

# Custom Views & Event Handling

Lecture 6



# Last Time...

## App Lifecycle and Structure

- Navigation in SwiftUI: NavigationStack, NavigationLink, TabView
- Modal Presentations: .sheet, .alert
- MVVM architecture (which is what again?)
- @Observable, @Bindable, @Environment



# This Week

## Custom Views & Event Handling

- GeometryReader, safe area
- SwiftUI shapes, .fill/.stroke, .clipShape, .contentShape
- Understanding event propagation and handling
- Keyboard handling and text input events
- Custom gesture recognition in SwiftUI



# GeometryReader


Why?

Lets us build/customize responsive layouts based on different screen sizes and orientations

How SwiftUI determines app structure:

1. Parent view proposes size for child view
2. Child view uses that to determine its size
3. Parent uses that size to position the child appropriately

How do we know what size though?





# GeometryReader

```
struct ContentView: View {
    var body: some View {
        GeometryReader { proxy in
            Text("Hello, World!")
                .frame(width: proxy.size.width * 0.9)
                .background(.red)
        }
    }
}
```

What is this View doing?

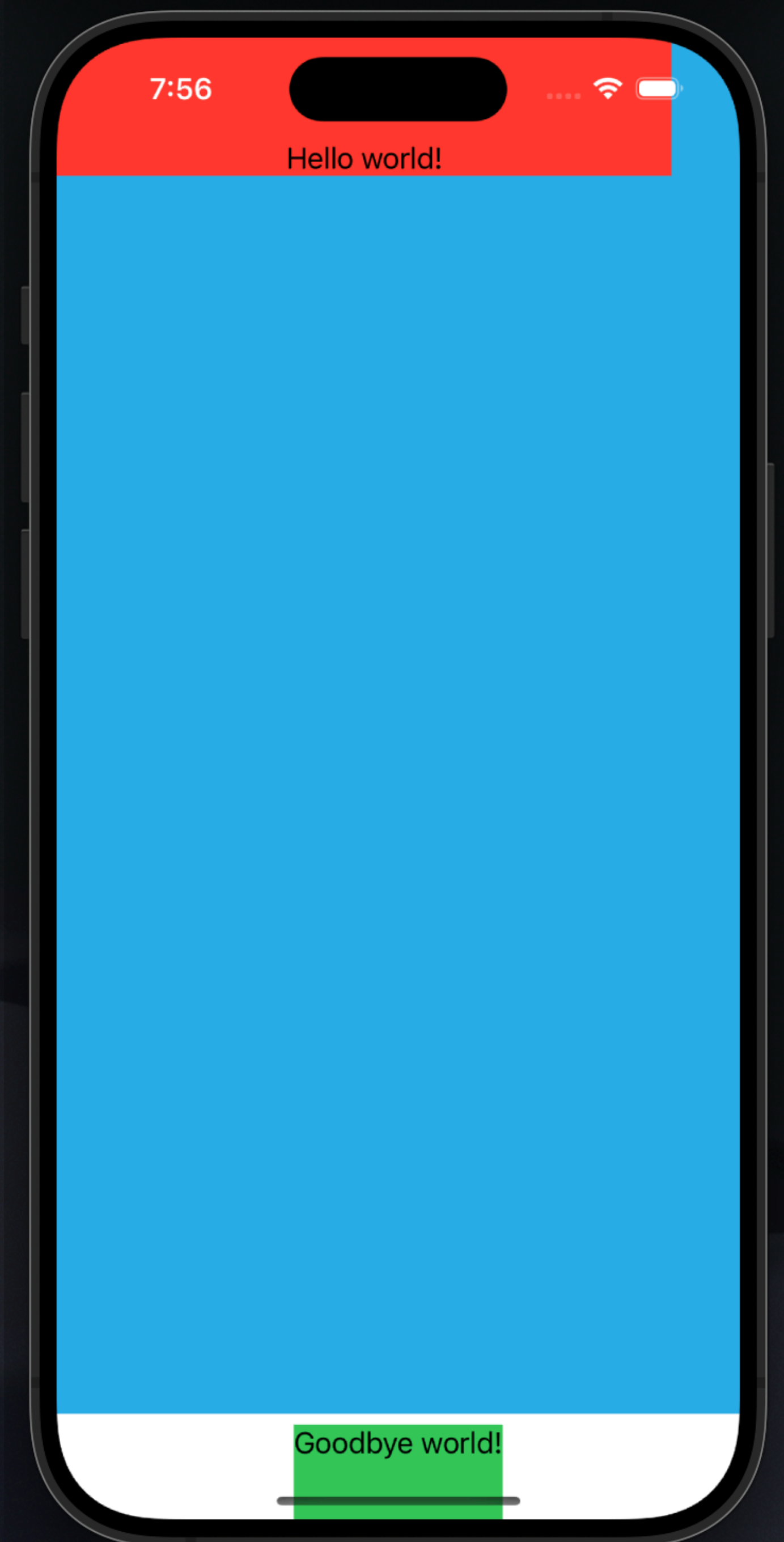


# GeometryReader

## Note of Caution

GeometryReader can take up all available space by default

```
struct ContentView: View {
    var body: some View {
        VStack {
            GeometryReader { proxy in
                Text("Hello world!")
                    .frame(width: proxy.size.width * 0.9)
                    .background(.red)
            }
            .background(.cyan)
            Text("Goodbye world!")
                .background(.green)
        }
    }
}
```





# GeometryReader

Proxy variable has type GeometryProxy

What if you want to get the coordinates of the View? Why might we want to do this?

- Use `.frame!` (more details in live demo)

## Accessing geometry characteristics

```
func bounds(of: NamedCoordinateSpace) -> CGRect?
```

Returns the given coordinate space's bounds rectangle, converted to the local coordinate space.

```
func frame(in: CoordinateSpace) -> CGRect
```

Returns the container view's bounds rectangle, converted to a defined coordinate space.

```
func frame(in: some CoordinateSpaceProtocol) -> CGRect
```

Returns the container view's bounds rectangle, converted to a defined coordinate space.

```
var size: CGSize
```

The size of the container view.

```
var safeAreaInsets: EdgeInsets
```

The safe area inset of the container view.

```
subscript<T>(Anchor<T>) -> T
```

Resolves the value of an anchor to the container view.

```
func transform(in: some CoordinateSpaceProtocol) -> AffineTransform3D?
```

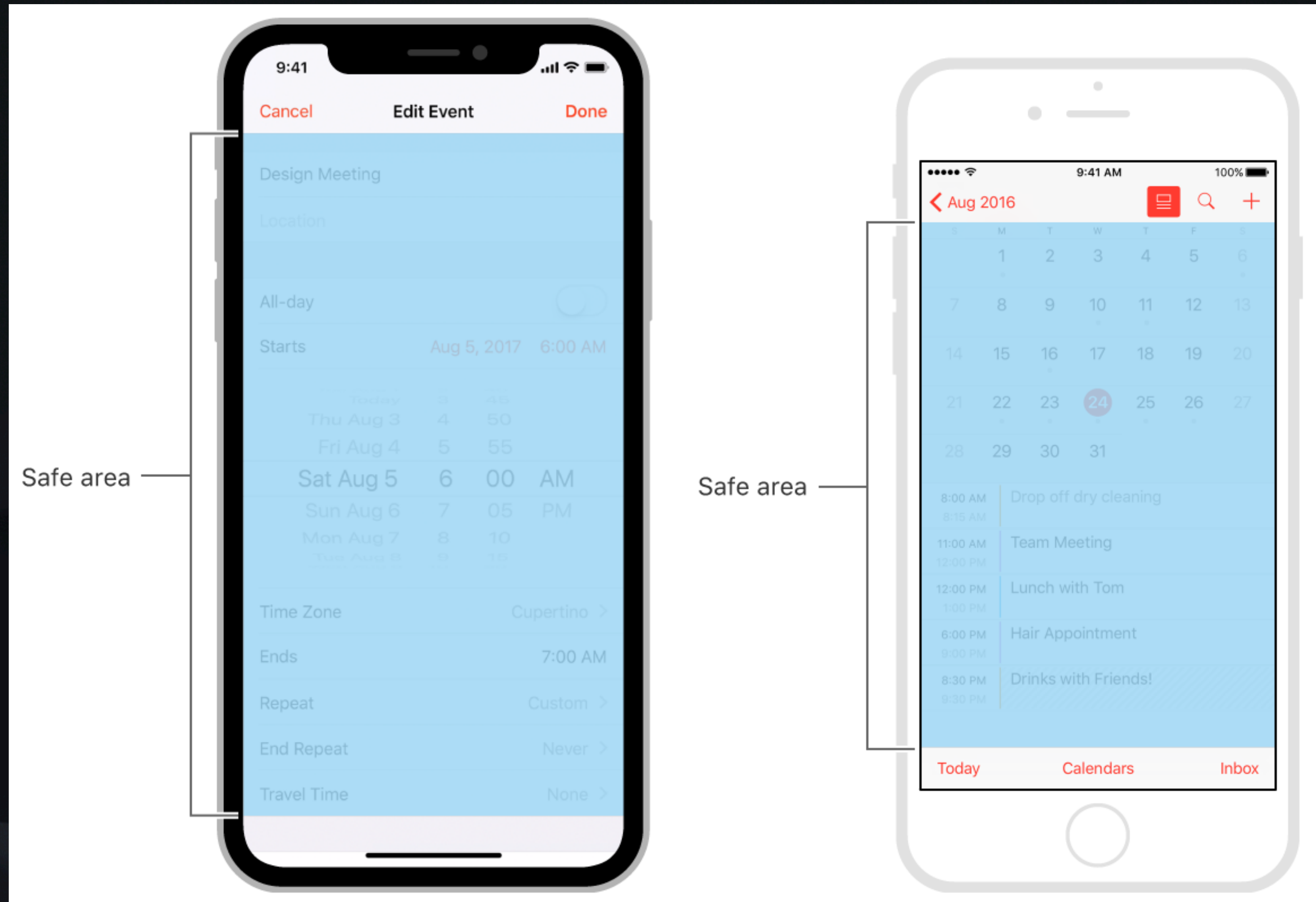
The container view's 3D transform converted to a defined coordinate space.



# Safe Areas in SwiftUI

By default, SwiftUI ensures views are placed in “safe” areas of the screen where navigation elements won’t display

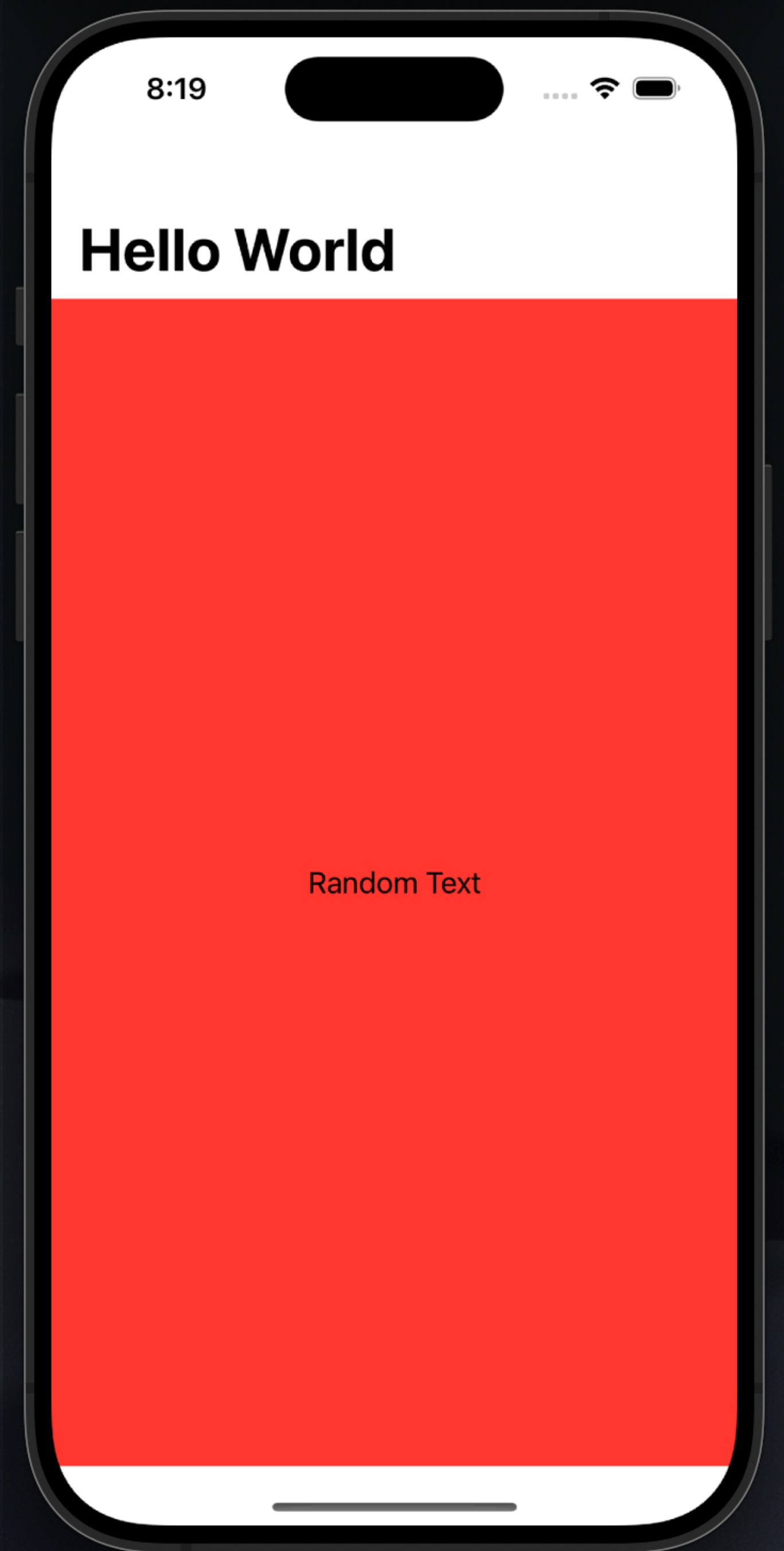
(e.g. avoiding the navigation bar, tab bar, toolbar, navigation title, etc.)





# Safe Areas in SwiftUI

```
struct ContentView: View {  
    var body: some View {  
        NavigationStack {  
            ZStack {  
                Color.red  
                Text("Random Text")  
            }  
            .navigationTitle("Hello World")  
        }  
    }  
}
```





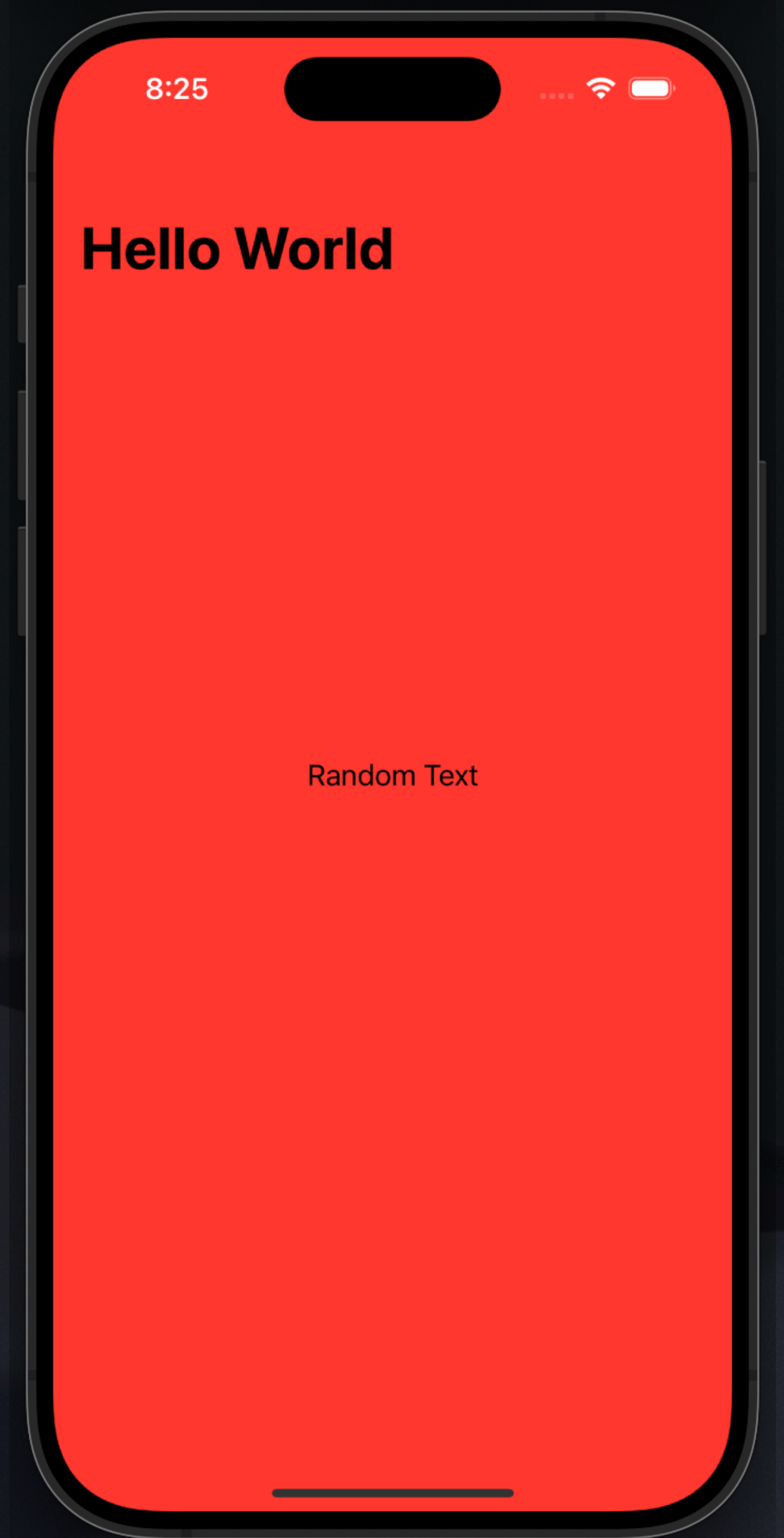
How do we ignore safe areas though?

`.ignoresSafeArea()` of course!



# Safe Areas in SwiftUI

```
struct ContentView: View {  
    var body: some View {  
        NavigationStack {  
            ZStack {  
                Color.red  
                Text("Random Text")  
            }  
            .ignoresSafeArea()  
            .navigationTitle("Hello World")  
        }  
    }  
}
```





# Safe Areas in SwiftUI

- You can customize the direction and region of ignored safe area
- Also see `.safeAreaInset()`, it lets you place a view *outside* the safe area



# SwiftUI Default Shapes

```
struct ContentView: View {
    var body: some View {
        VStack {
            Rectangle()
                .fill(.red)
                .frame(width: 200, height: 100)

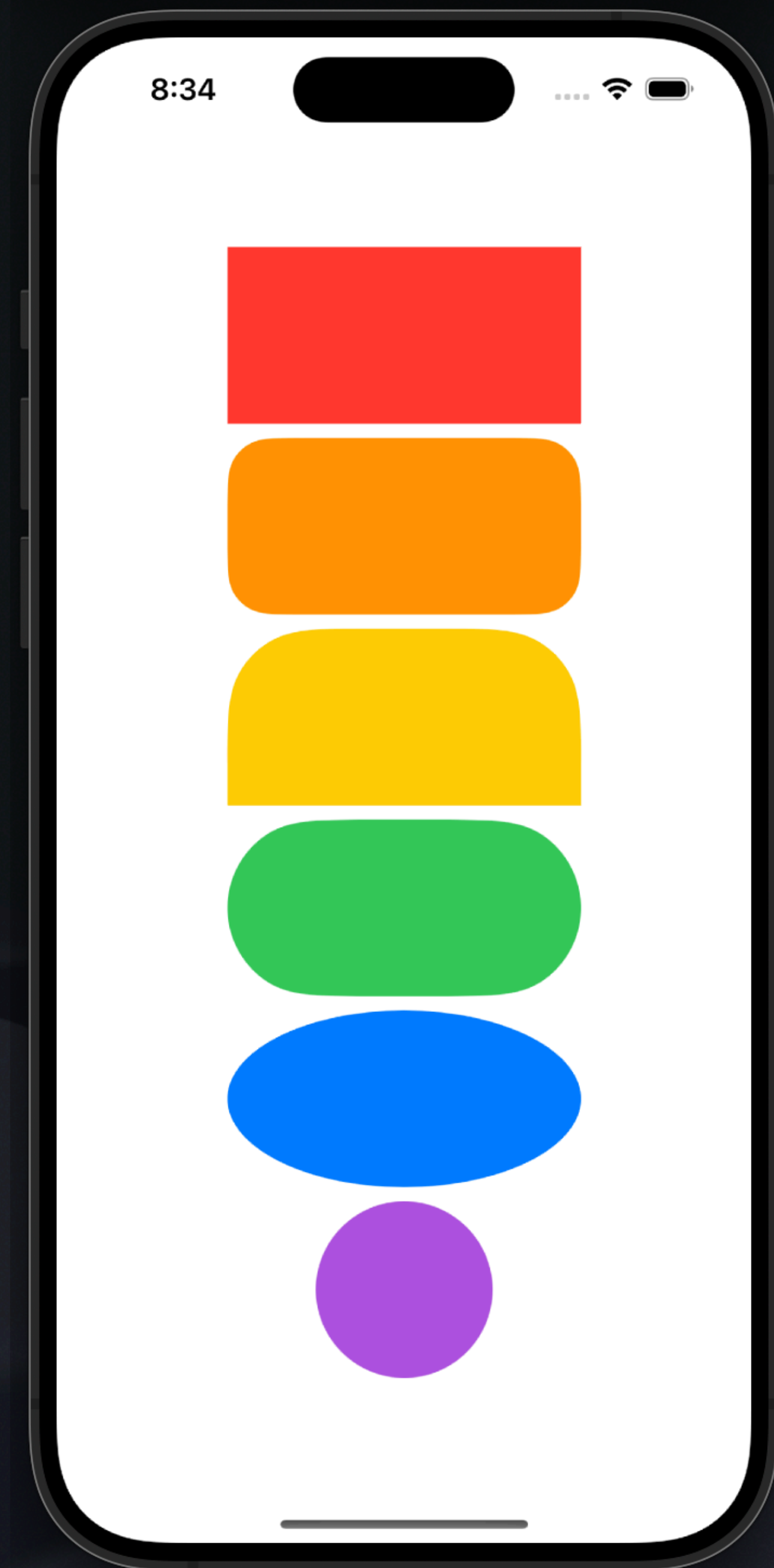
            RoundedRectangle(cornerRadius: 25)
                .fill(.orange)
                .frame(width: 200, height: 100)

            UnevenRoundedRectangle(cornerRadius: .init(topLeading: 50, topTrailing: 50))
                .fill(.yellow)
                .frame(width: 200, height: 100)

            Capsule()
                .fill(.green)
                .frame(width: 200, height: 100)

            Ellipse()
                .fill(.blue)
                .frame(width: 200, height: 100)

            Circle()
                .fill(.purple)
                .frame(width: 200, height: 100)
        }
    }
}
```





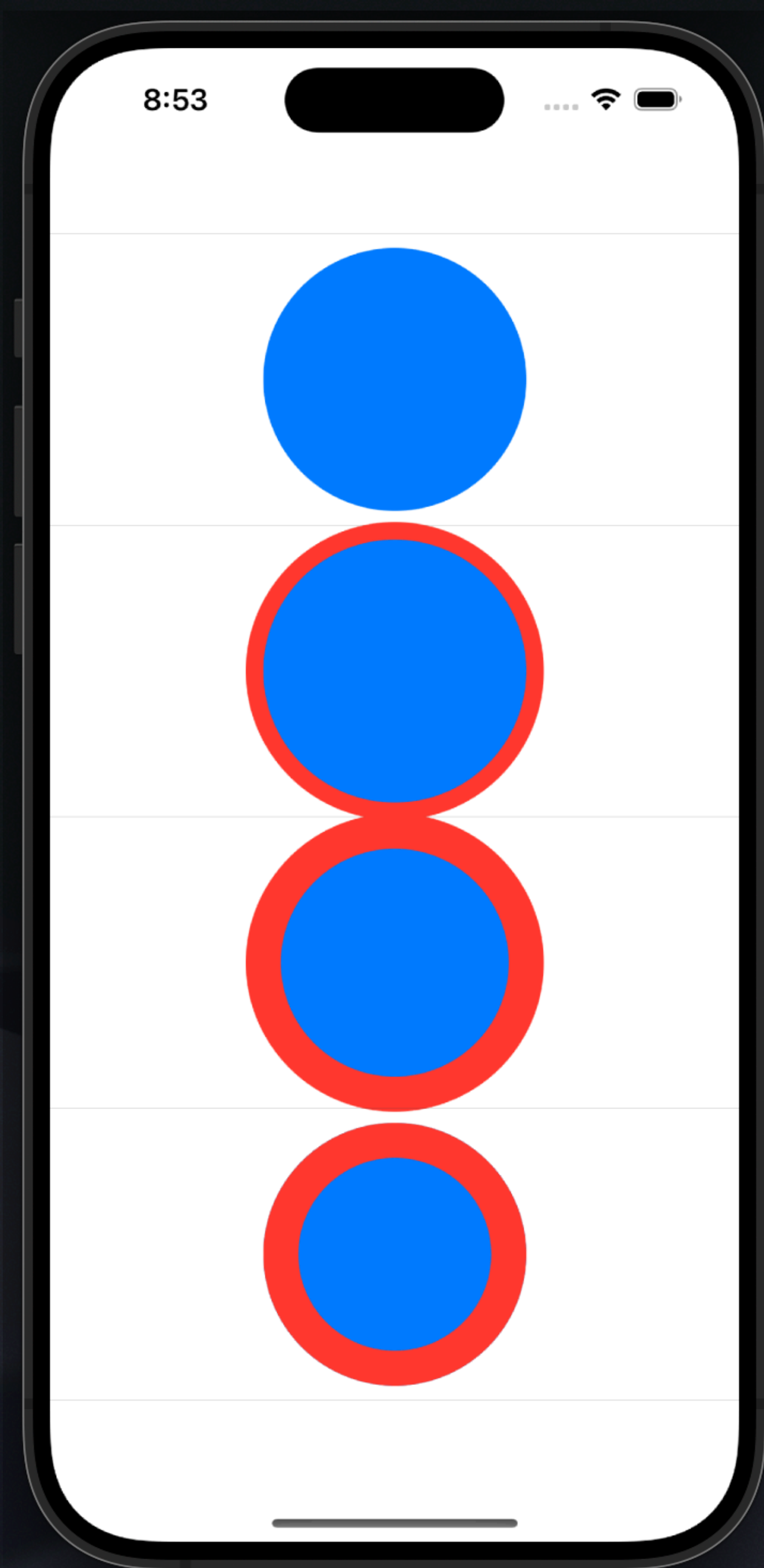
# Shape Modifiers

- `.fill()` does what you think, fills the Shape with the provided color (or gradient using `LinearGradient`)
- `.stroke()` does what you think. It draws a border **centered on the view's edge**, so half of the border will be inside the view and half outside
- `.strokeBorder()` **insets your view**, and then draws a border **entirely inside the original size**



# SwiftUI Default Shapes

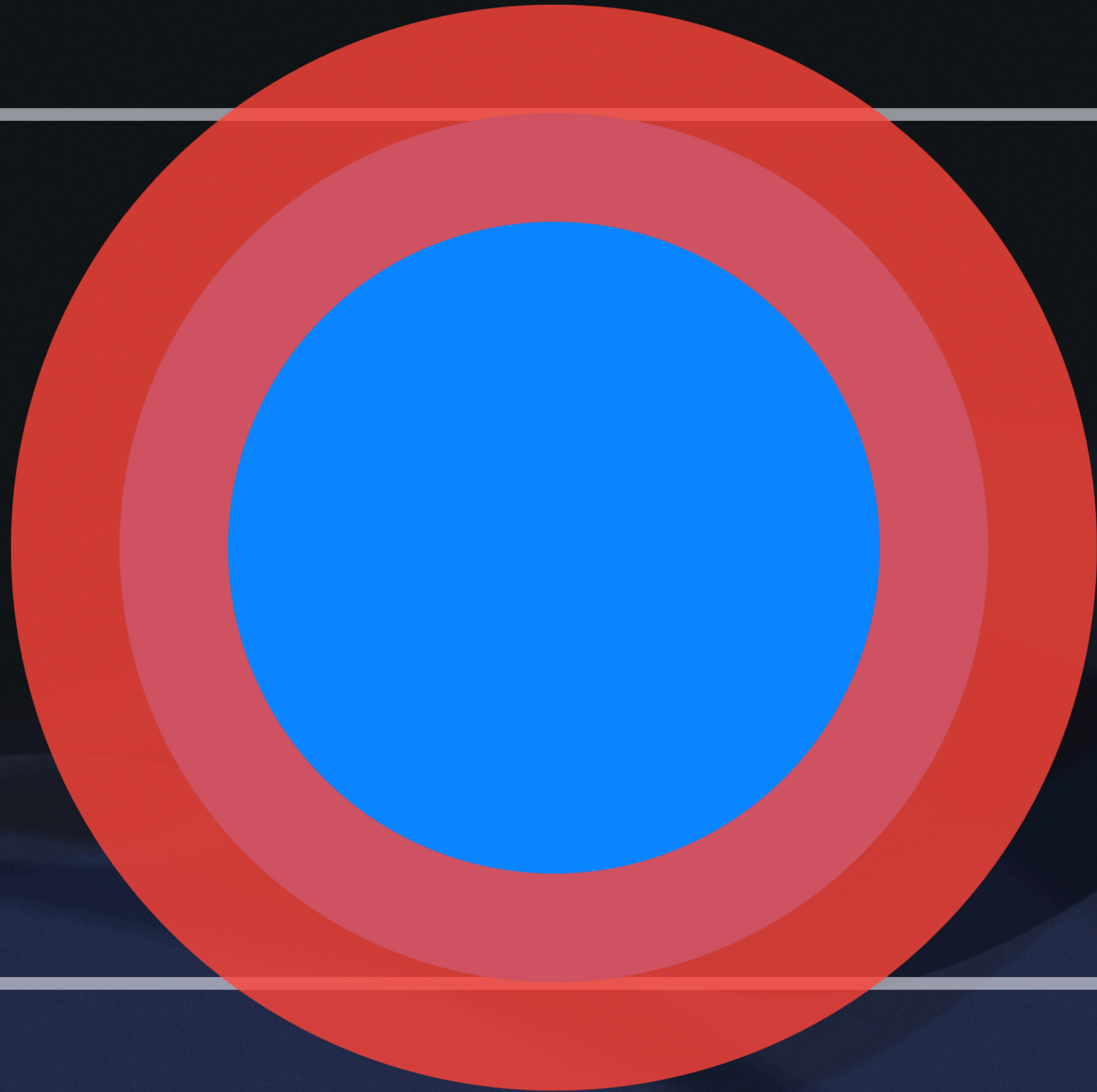
```
struct ContentView: View {
    var body: some View {
        VStack {
            Divider()
            Circle()
                .fill(.blue)
                .frame(width: 150, height: 150)
            Divider()
            Circle()
                .stroke(.red, lineWidth: 20)
                .fill(.blue)
                .frame(width: 150, height: 150)
            Divider()
            Circle()
                .fill(.blue)
                .stroke(.red, lineWidth: 20)
                .frame(width: 150, height: 150)
            Divider()
            Circle()
                .fill(.blue)
                .strokeBorder(.red, lineWidth: 20)
                .frame(width: 150, height: 150)
            Divider()
        }
    }
}
```





`.stroke`

`.strokeBorder`



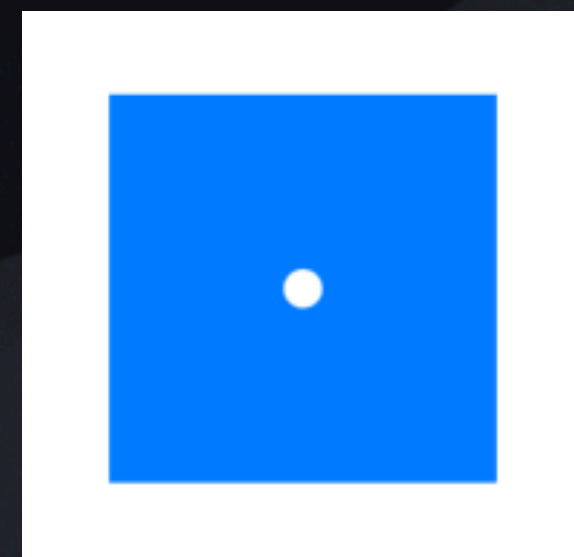


# Sidenote: Canvas

Immediate mode drawing, just like CIS 1200

```
struct ContentView: View {
  var body: some View {
    Canvas { gctx, size in
      gctx.translateBy(x: 45, y: 45)

      let path = Path(ellipseIn: CGRect(x: 0, y: 0, width: 10, height: 10))
      gctx.fill(path, with: .color(.white))
    }
    .frame(width: 100, height: 100)
    .background(.blue)
  }
}
```

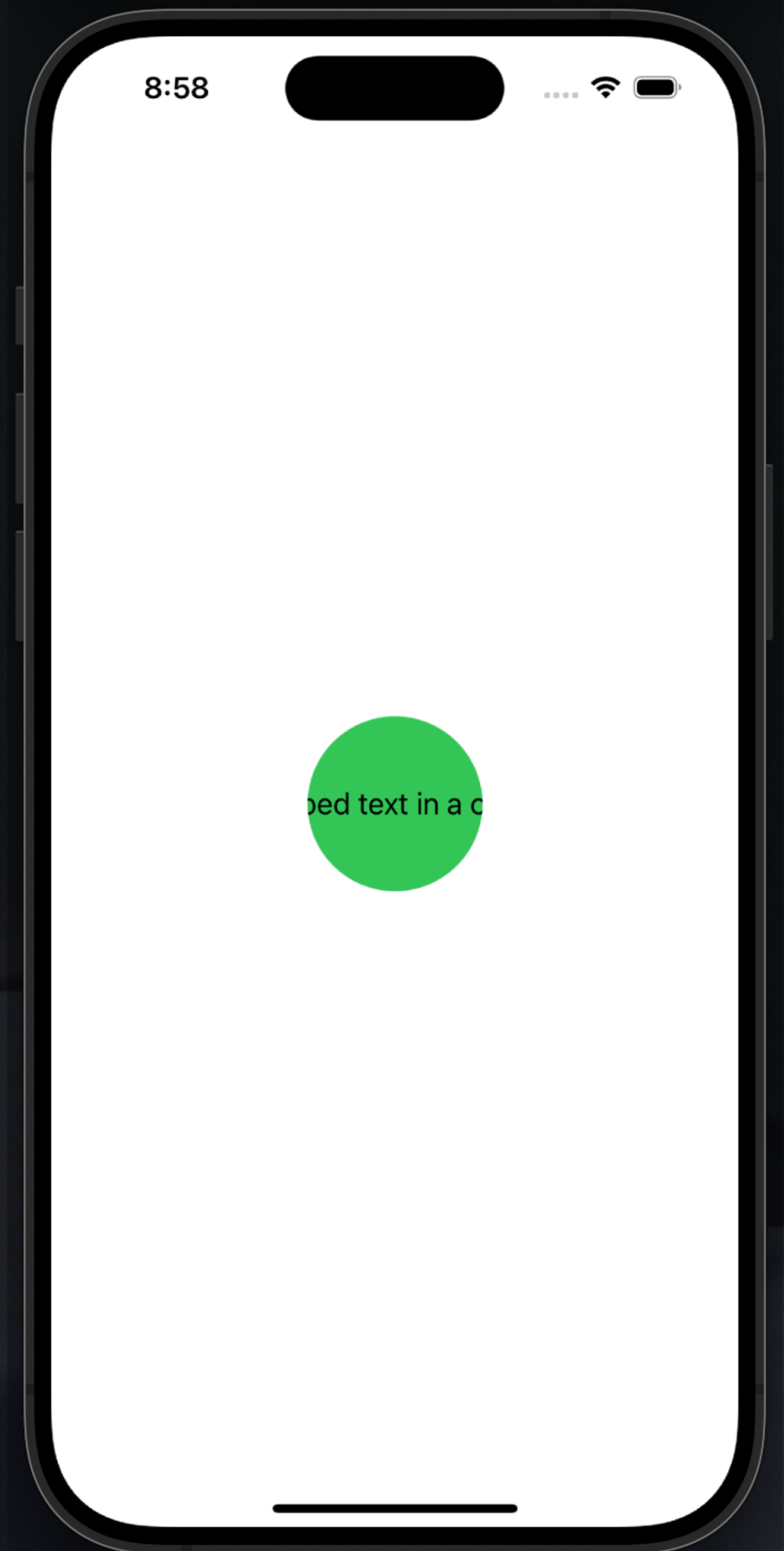




# .clipShape()

- Clips a view's borders to the shape given

```
struct ContentView: View {  
    var body: some View {  
        Text("Clipped text in a circle")  
            .lineLimit(1)  
            .frame(width: 200, height: 100)  
            .foregroundColor(Color.black)  
            .background(Color.green)  
            .clipShape(Circle())  
    }  
}
```





# Events and Gesture Handling

## In SwiftUI

- Events such as taps, swipes, and other gestures are automatically handled by Gestures such as:
  - onTapGesture, .onTapGesture(count: 3)
  - onLongPressGesture, onLongPressGesture(minimumDuration: 5)
  - DragGesture()
  - MagnificationGesture()
  - RotationGesture()
- More details in the live demo!

```
struct ContentView: View {  
    var body: some View {  
        Text("Tap me!")  
            .padding()  
            .background(.red)  
            .onTapGesture {  
                print("Tapped")  
            }  
    }  
}
```



# Events and Gesture Handling

What if nested views all have gesture handlers, which one is used?

The event starts out at the inner most view, and is handled by the inner most event recognizer. We say the event is “consumed,” so no other handlers on outer views get a chance to respond.

What if we want to change this behavior so multiple gestures work at the same time?

- Check out the `.simultaneousGesture()` for more control over priority!

What if there are multiple gesture handlers on the same view we want to control?

- Check out `.exclusively()`, `.simultaneously()`, and `.sequentially()`

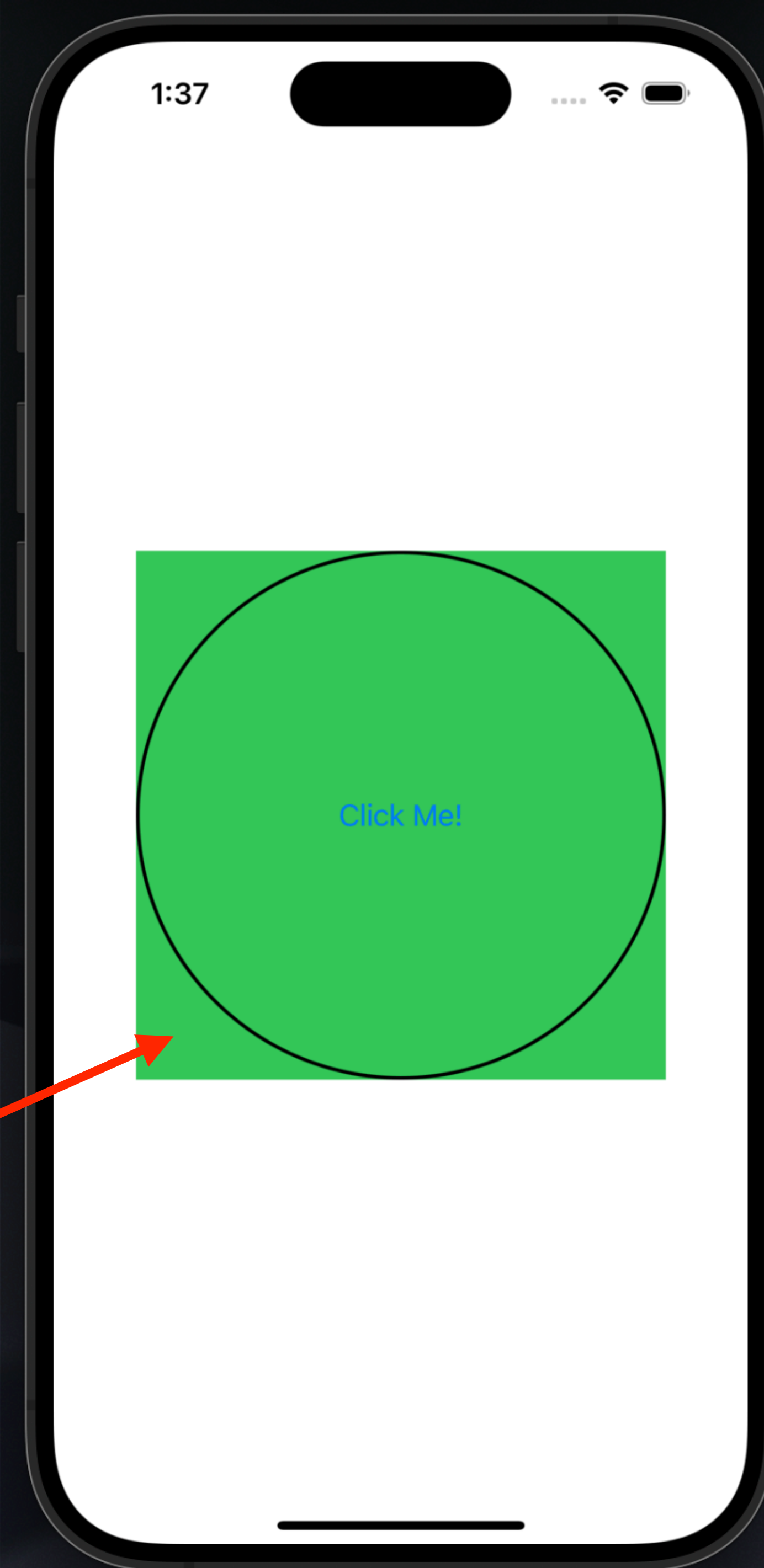


# .contentShape()

- Modifies the interactable shape of the view during hit-testing

```
struct ContentView: View {  
    var body: some View {  
        Button(action: {  
            print("Button clicked")  
        }) {  
            Text("Click Me!")  
                .frame(width: 300, height: 300)  
                .background(.green)  
        }  
        .contentShape(Circle())  
    }  
}
```

Clicking here  
doesn't trigger  
the button





# Keyboard Handling and Text Input Events

We can also handle inputs and events directly from the keyboard!

- Note: The view must be focusable to respond to key presses, see `.focusable()`. This brings up the keyboard when focused
- Also note the “`return .handled`” here. This is saying the event has been handled, so it becomes consumed. If we returned `.ignored` instead, then the event could still be handled by another handler outside this view

```
struct ContentView: View {
    @FocusState private var focused: Bool
    @State private var key = ""

    var body: some View {
        Text(key)
            .padding()
            .background(.green)
            .focusable()
            .focused($focused)
            .onKeyPress { press in
                key += press.characters
                print("\(press.characters) pressed!")
                return .handled
            }
            .onAppear {
                focused = true
            }
    }
}
```



# Custom Gesture Recognition

Each gesture has some of its own features/properties, so check out their documentation!

We can use `.onChanged` and `.onEnded` for many to perform actions based on the gestures.



# Gesture Customization

We can also customize our own gestures! What do you think this does?

```
struct ContentView: View {
    var body: some View {
        Text("Hello world!")
            .padding()
            .background(.green)
            .gesture(DragGesture(minimumDistance: 3.0, coordinateSpace: .local)
                .onEnded { value in
                    switch(value.translation.width, value.translation.height) {
                        case (...0, -30...30): print("left")
                        case (0..., -30...30): print("right")
                        case (-100...100, ...0): print("up")
                        case (-100...100, 0...): print("down")
                        default: print("none")
                    }
                }
            )
    }
}
```



# Gesture Creation

You can also create your own gestures, just make a struct that conforms to Gesture!

Most of the time this is not needed, as the default gestures cover almost everything you need.

Look up the documentation for it!

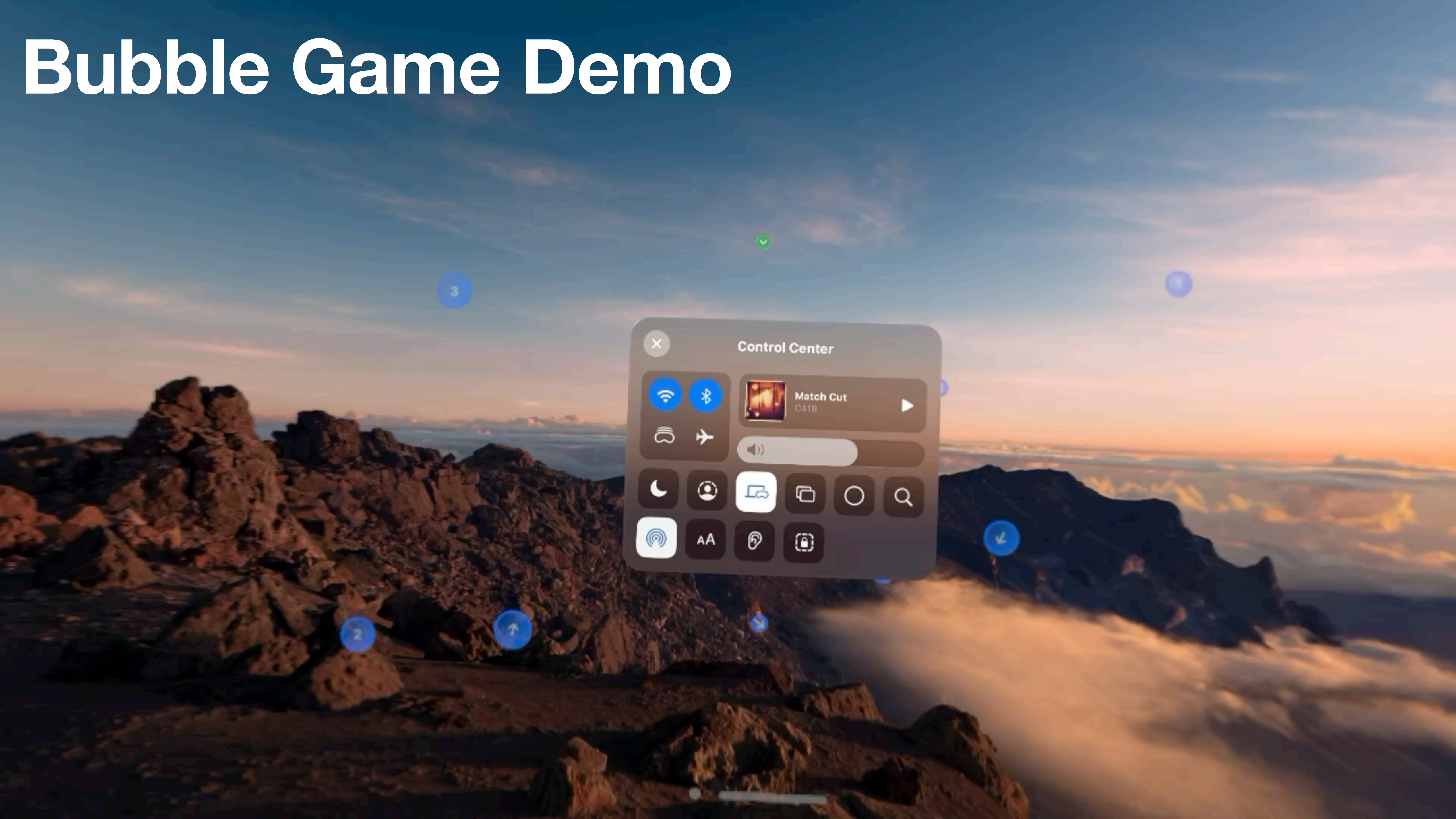


# Custom Gesture Recognition

Try creating your own in the live demo later!



# Bubble Game Demo



Control Center

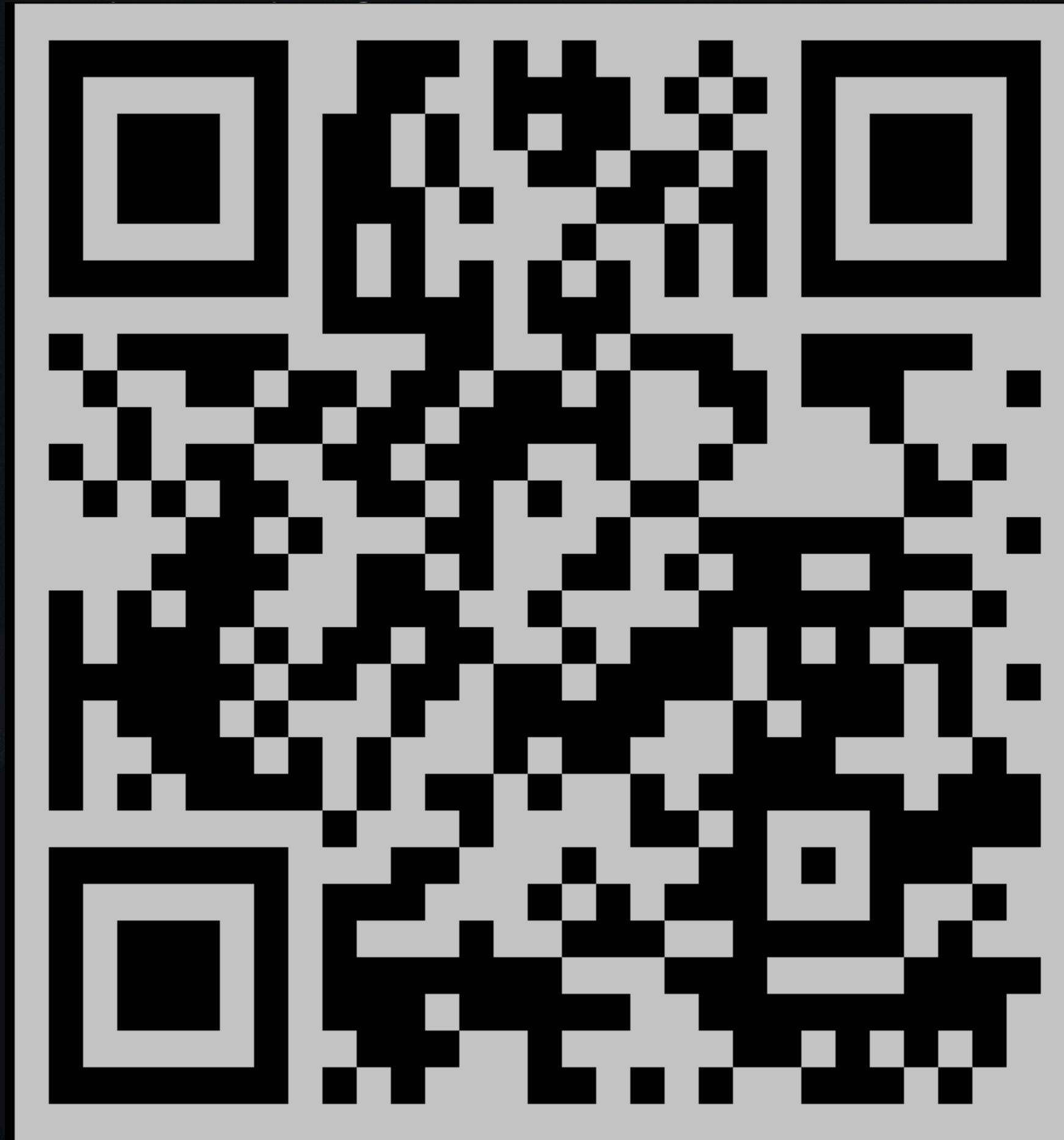
Match Cut C41B

AA

Control Center panel with various system icons and a music player interface.



# Bubble Game Demo



<https://github.com/cis1951/lec6-code>



# Recap

## Custom Views & Event Handling

- GeometryReader, safe area
- SwiftUI shapes, .fill/.stroke, .clipShape, .contentShape
- Understanding event propagation and handling
- Keyboard handling and text input events
- Custom gesture recognition in SwiftUI



# See you next week!

Homework 2 Trivia Game:

- Due on **Wednesday, 3/19**
- Focuses on **lectures 3-5**