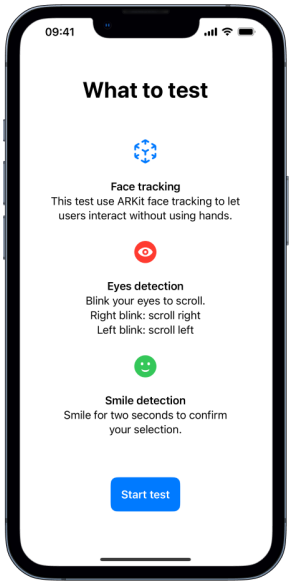
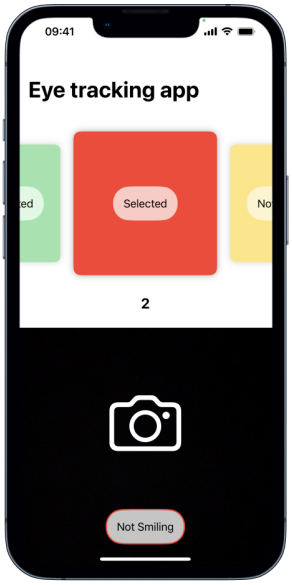
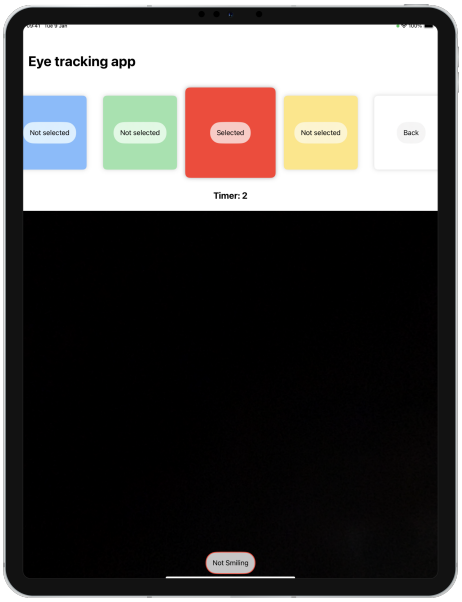
EyeTracking App

Concept by Alessandro Esposito Vulgo Gigante

Repository Link: https://github.com/alexvulgo/EyeTracking







The purpose of this project is to conduct a speculative analysis in order to produce a fully interactive application that doesn't require the use of hands. The main aim is to provide a solution for individuals who have motor disabilities or difficulty using their upper limbs. I am of the opinion that technological advancement should always be supported by strong accessibility.
I think that the proposed solution can be useful for both the use of devices like iPhone and iPad and as a substitute for manual gestures on devices like the VisionPro.

This test was executed by me using **SwiftUI** and the **ARKit**, **RealityKit frameworks** provided by **Apple**.

I wanted to focus on three specific points:

- **Scrolling** of elements
- **Selection** of elements
- **Navigation** between the views

A generic array is displayed on the main screen using multiple cards, each representing a different color.
By **blinking** your right eye, you can scroll items to the right. Similarly, using the left eye, you can scroll to the left.

The vector will return to its startl or final position if you reach the right or left margin.
The camera located at the bottom of the screen is capable of observing eye closure and determining whether the user is smiling or not.
If the smile has been detected, the user is informed by a button.
A smile for two seconds is necessary to confirm the selection of an item.

A timer is visible below the cards, which allows the user to check the time needed to confirm their selection.
Selecting an element causes the card's opacity to increase and the word 'selected' to appear.

The last card is an example to show how is possible to move between different views using this eye tracking mechanism and works as a back button.

An onboarding page when starting the application summarizes the commands that were just listed.

Eye Tracking

Real-time facial expression recognition can be achieved by analyzing the user's face using ARKit and RealityKit.

ARKit: *"Detect faces in a front-camera AR experience, overlay virtual content, and animate facial expressions in real-time."*

Before defining the project's views, I had to set an ARContainer that would only recognize the facial information I needed.

```
struct CustomContainer: UIViewRepresentable {

    @Binding var LeftisWinking: Bool
    @Binding var RightisWinking: Bool

    @Binding var SmileLeftCheck: Bool
    @Binding var SmileRightCheck: Bool


    func makeUIView(context: Context) -> CustomARView {

        return CustomARView(LeftisWinking : $LeftisWinking , RightisWinking :
            $RightisWinking , SmileLeftCheck : $SmileLeftCheck,
            SmileRightCheck : $SmileRightCheck)

    }

    func updateUIView(_ uiView: CustomARView, context: Context) {}
}

class CustomARView: ARView, ARSessionDelegate {

    @Binding var LeftisWinking: Bool
```

```
    @Binding var RightisWinking: Bool

    @Binding var SmileLeftCheck: Bool
    @Binding var SmileRightCheck: Bool

    init(LeftisWinking: Binding<Bool> , RightisWinking:
      Binding<Bool>,SmileLeftCheck: Binding<Bool> , SmileRightCheck:
      Binding<Bool>) {

        _LeftisWinking = LeftisWinking
        _RightisWinking = RightisWinking

        _SmileLeftCheck = SmileLeftCheck
        _SmileRightCheck = SmileRightCheck

        super.init(frame: .zero)

        self.session.delegate = self

        let configuration = ARFaceTrackingConfiguration()
        self.session.run(configuration)
    }


    func session(_ session: ARSession, didUpdate anchors: [ARAnchor]) {

            guard let faceAnchor = anchors.compactMap({ $0 as?
            ARFaceAnchor }).first else {
            return
        }

        detectBlink(faceAnchor: faceAnchor)

        detectSmile(faceAnchor: faceAnchor)

    }
```

Scrolling

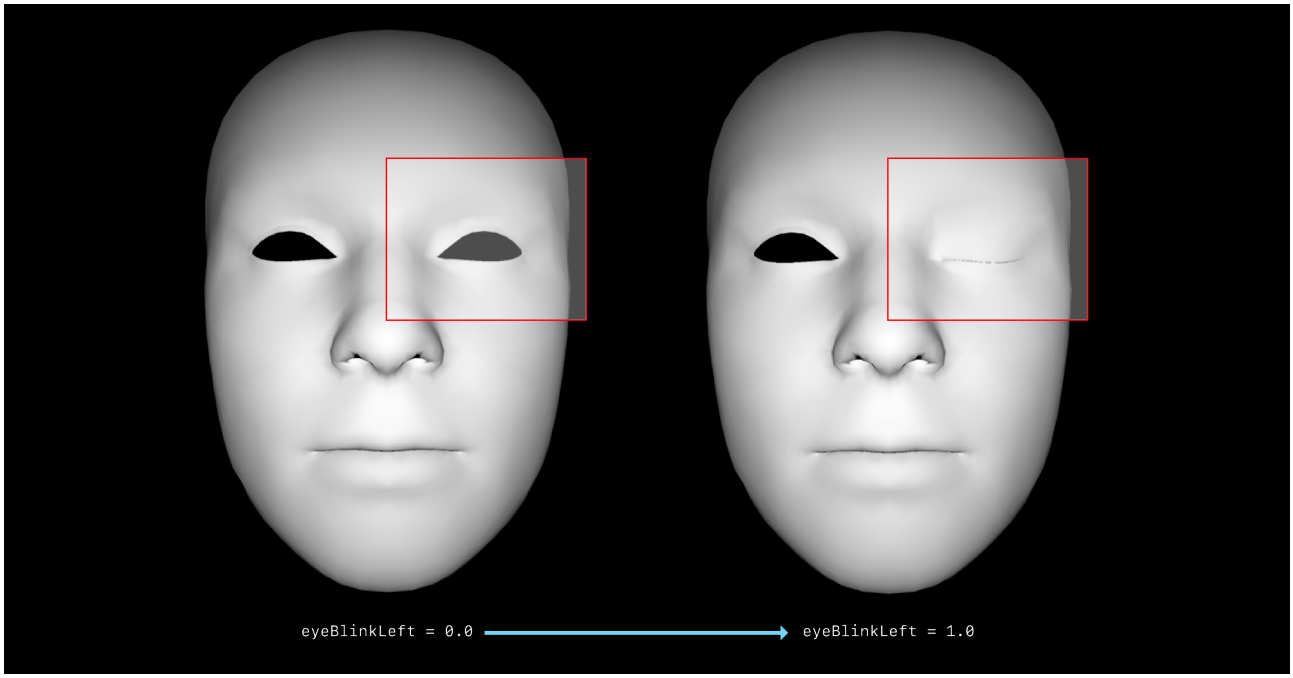
In order to navigate through the array elements, the goal was to use the right and left eye closures to change directions.
An object called **ARFaceAnchor** is provided by ARKit to accomplish this task.
The Apple documentation states:
"An anchor for a unique face that is visible in the front-facing camera."
Using the FaceAnchor class, you can use specific facial expressions recognized by the framework and exploited through variables called **blendShapes**.
*"The **blendShapes** dictionary provided by an **ARFaceAnchor** object describes the facial expression of a detected face in terms of the movements of specific facial features. For each key in the dictionary, the corresponding value is a floating point number indicating the current position of that feature relative to its neutral configuration, ranging from 0.0 (neutral) to 1.0 (maximum movement)."*

```
private func detectBlink(faceAnchor: ARFaceAnchor) {

        let blendShapes = faceAnchor.blendShapes
```

```
        if let leftEyeBlink = blendShapes[.eyeBlinkLeft] as? Float,
           let rightEyeBlink = blendShapes[.eyeBlinkRight] as? Float {

            if rightEyeBlink > 0.8 {
                RightisWinking = true
            } else {
                RightisWinking = false
            }

            if leftEyeBlink > 0.8 {
                LeftisWinking = true
            } else {
                LeftisWinking = false
            }

        }
    }
```

In my case, I opted for:

- **blendShapes[.eyeBlinkLeft] ->**The coefficient describing closure of the eyelids over the left eye.
- **blendShapes[.eyeBlinkRight] ->** The coefficient describing closure of the eyelids over the right eye.



When the left eye is completely open, the variable **eyeBlinkLeft** corresponding value is **0.0**, and when it is completely closed, it's **1.0**. Similar reasoning applies to the **eyeBlinkRight** variable.

Checking if the variable representing this coefficient is greater than 0.9 is the only way to confirm the actual closure or “blink” of the eye.
In order to ensure easy accessibility, I decreased the value of this test to increase control fluidity even when the eye is not completely closed.

When the value of the variables eyeBlinkLeft and eyeBlinkRight correspond to the closing of the respective eyelid, the boolean variables **LeftisWinking** and **RightisWinking** become **true.** This two variables allows to control the array in the contentView.

```
} .onChange(of: LeftisWinking) {

   scroll_right()

  }
  .onChange(of: RightisWinking) {

   scroll_left()

  }
```

The function **.onchange** is used to trigger the respective function when a value changes its state.

```
func scroll_right() {
        if (RightisWinking == false && LeftisWinking == true) {
            withAnimation {
                if(currentIndex < viewModel.items.count - 1){
                    currentIndex = currentIndex + 1
                } else {
                    currentIndex = 0
                }
            }
        }
    }
```

```
func scroll_left() {
        if (RightisWinking == true && LeftisWinking == false) {
            withAnimation {
                if(currentIndex <= viewModel.items.count - 1 && currentIndex>0){
                    currentIndex = currentIndex - 1
                } else {
                    currentIndex = viewModel.items.count-1
                }
            }
        }
    }
```

Analyzing the **scroll_right()** function, we can see that the first thing to state is that only the right eye is recognized as closed. The reason for this is that the application constantly monitors the movement of the user's eyes, so the action is only activated when one eye is closed.
After that, the array index is incremented and the next element is shown. The same logic is applied to the **scroll_left()** function.

Selection

When it comes to selecting an element, I replaced the tap gesture with an analysis of the user's smile. This way, when the application recognizes that the user is smiling, it performs an action (in this case, selecting an element from the array). More precisely, the framework acknowledges the two-part smile, with the left corner of the mouth and the right corner of the mouth. These two coefficients were broken down into two variables: **smileRight** and **smileLeft.**

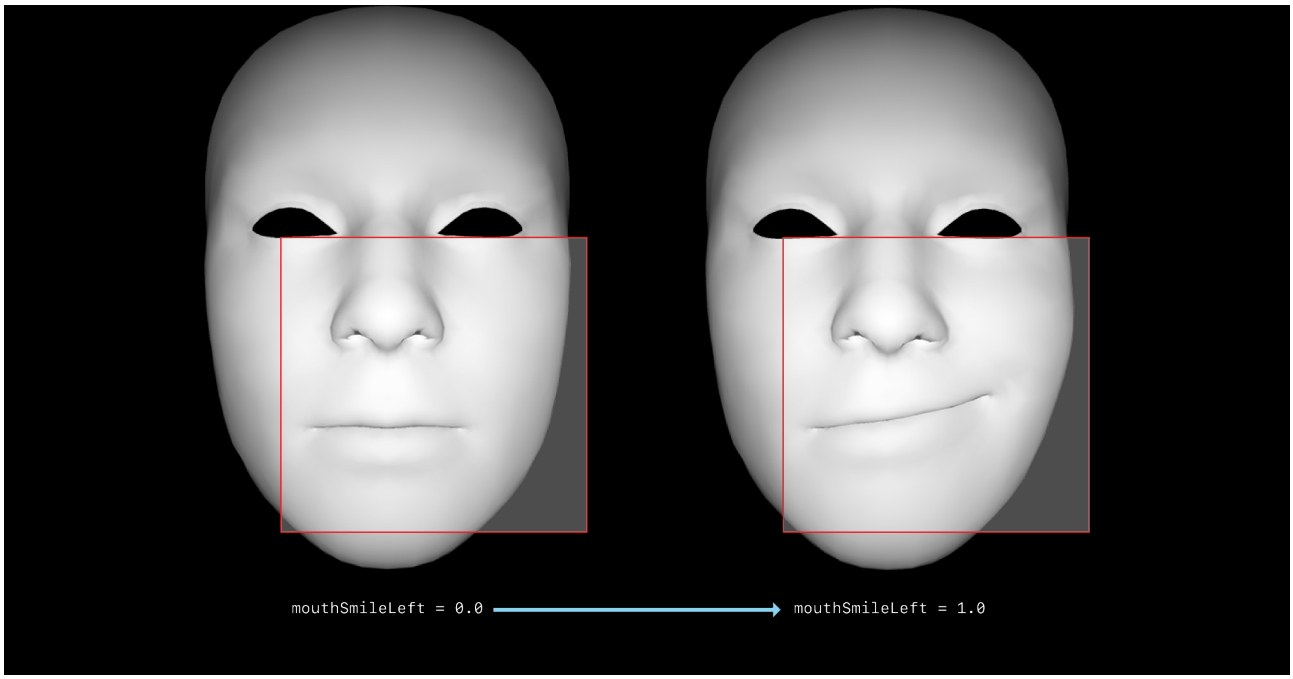
- **blendShapes[.mouthSmileLeft] ->**The coefficient describing upward movement of the left corner of the mouth.
- **blendShapes[.mouthSmileRight] ->** The coefficient describing upward movement of the right corner of the mouth.

Checking that the two values are higher than 0.9, is the least effective way to get a smile, but I wanted the system to recognize subtle mouth movements instead of a full smile with fully open mouth. Thus, the control coefficient is decreased to 0.6.

```
private func detectSmile(faceAnchor: ARFaceAnchor) {

        let blendShapes = faceAnchor.blendShapes

        if let smileLeft = blendShapes[.mouthSmileLeft] as? Float,
           let smileRight = blendShapes[.mouthSmileRight] as? Float {

            if smileRight > 0.6 {
                SmileRightCheck = true
            } else {
                SmileRightCheck = false
            }

            if smileLeft > 0.6 {
                SmileLeftCheck = true
            } else {
                SmileLeftCheck = false
            }

        }
    }
```

Just like scrolling, the **boolean variables smileRightCheck** and **smileLeftCheck** are always monitoring these values in the main view and are utilized in the function **.onchange()** verifying if the user is smiling or not.

```
.onChange(of: SmileLeftCheck || SmileRightCheck){

 confirm()

 }

func confirm() {

       if (load == false){
           load = true
           startCountdown()
       } else if (load == true) {
           load = false
           resetCountdown()
       }

   }
```



A confirmation function is triggered when the smile is detected due to changes in the boolean control variables, which will manage the selection of an element via a **timer**. To keep unwanted movements from interfering with the array, the user is required to keep a smile for two seconds. Interrupting the smile causes the timer to become invalid and the user must repeat the procedure.

Navigation

Switching between different views was another problem that I wanted to address while developing this solution. My inability to interact with buttons manually prevented me from using a simple navigation link to move between the views. To overcome this issue, I developed a custom view manager that utilizes an **EnvironmentObject** and a **switch** function.

During testing, only *onboarding* and *first view* were established.

EyeTrackingApp.swift

```
import SwiftUI


enum AppState {
    case onboarding
    case firstView
}


class StateManager: ObservableObject{

    @Published var currentState = AppState.onboarding

}



@main
struct EyeTrackingApp: App {

    private var stateManager = StateManager()

    init() {

    }


    var body: some Scene {
        WindowGroup {
            SceneContainerView()
                .environmentObject(stateManager)
        }
    }
}
```

SceneContainerView.swift

```
import SwiftUI

struct SceneContainerView: View {

    @EnvironmentObject var stateManager : StateManager

    var body: some View {


        switch(stateManager.currentState){

        case .onboarding:
            OnboardingView()
                .environmentObject(stateManager)


        case .firstView:
            ContentView(viewModel: itemViewModel())
                .environmentObject(stateManager)

        }
```

```
    }
}
```

Invoking a function that sets a change of state is sufficient to change the view. Consequently, it is possible to create an object within the application that, recognizing the user's smile, invokes the function and alters the view.
In my case, the final element in the array when selected serves as a back button and brings the user back to the onboarding view.

```
func back() {
        stateManager.currentState = .onboarding
    }
```