Sample Code

Analyzing a selfie and visualizing its content

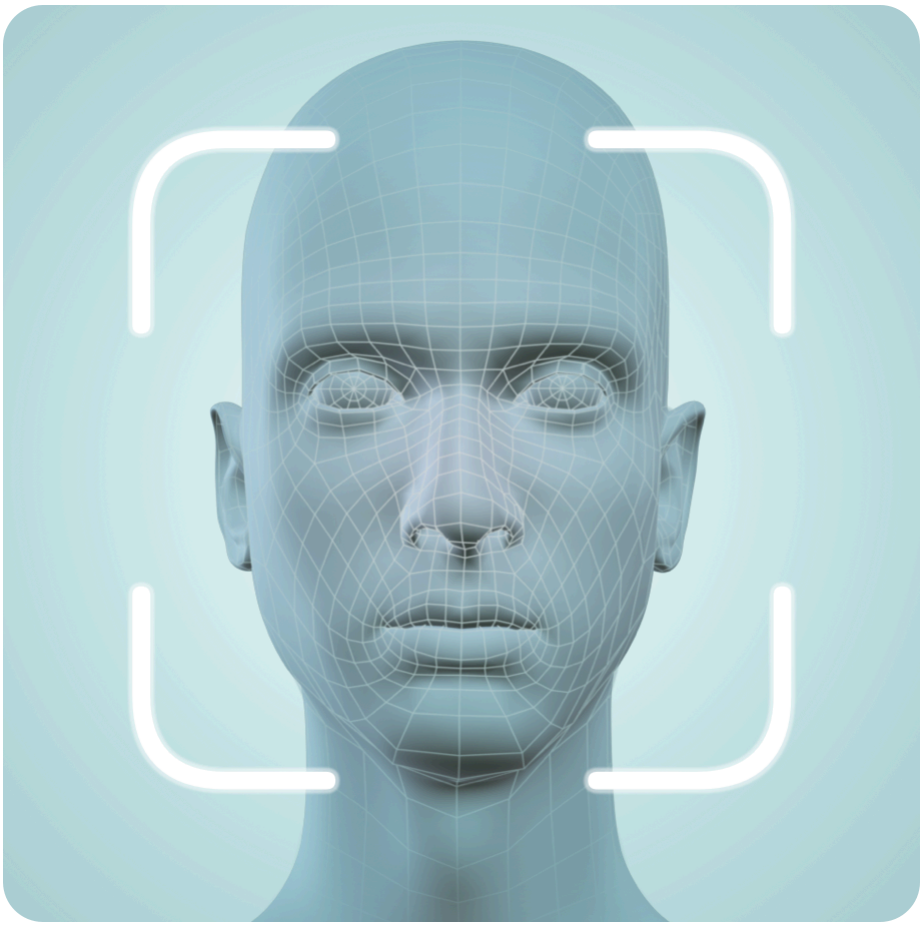
Calculate face-capture quality and visualize facial features for a collection of images using the Vision framework.

Download

iOS 18.0+ | iPadOS 18.0+ | Xcode 16.1+

Overview

Use the `Vision` framework to detect faces and facial features in a photo. This framework can analyze a photo to retrieve metrics such as face-capture quality and visual information like facial landmarks and face rectangles. This sample demonstrates how to locate all the faces in a selfie through the `DetectFaceRectanglesRequest`. The sample then uses `DetectFaceCaptureQualityRequest` to obtain capture-quality scores, and `DetectFaceLandmarksRequest` to display outlines around each facial landmark, like the eyes or nose.



Face-capture quality is a holistic measure that considers scene lighting, blur, occlusion, expression, pose, focus, and more. It provides a score that the app uses to sort the collection of selfies from best to worst. The pretrained machine-learning model scores a capture lower if, for example, the image contains low light or bad focus, or if the person has a negative expression. These scores are floating-point values between 0.0 and 1.0.

Configure the sample code project

To run this sample app, you need the following:

- Xcode 16 or later
- iPhone with iOS 18 or later

Selecting the selfies

The sample uses `PhotosPicker` to allow a person to select the selfies to analyze, and sets the maximum number of images to 5 through the `maxSelectionCount` parameter. For more information on using `PhotosPicker`, see Bringing Photos picker to your SwiftUI app:

```
PhotosPicker(selection: $selectedPhotos, maxSelectionCount: 5, matching: .images) {
    Text("Select Selfies")
}
```

The sample performs the `Vision` requests and displays the images using data, so the app converts each `PhotosPickerItem` to data:

```
for photo in selectedPhotos {
    if let image = try? await photo.loadTransferable(type: Data.self) {
        selectedPhotosData.append(image)
    }
}
```

Perform the requests and analyze the selfies

To analyze a selfie, the sample first instantiates the three `Vision` requests. The sample then performs `DetectFaceRectanglesRequest` to locate the faces in the photo, and uses the returned `FaceObservation` objects as the objects the other two requests process. The app sets this functionality through the `inputFaceObservations` property.

By default, `DetectFaceLandmarksRequest` and `DetectFaceCaptureQualityRequest` need to locate the faces first before performing the rest of the request. Setting the `inputFaceObservations` property prevents the sample from performing `DetectFaceRectanglesRequest` more than once (which is unnecessary).

Using the `score` method on a `FaceObservation`, the sample sets the selfie's score. The function returns the new `Selfie` object, which holds the photo, the score, and the results of the `DetectFaceLandmarksRequest`:

```
func processSelfie(photo: Data) async throws -> Selfie {
    /// Instantiate the `Vision` requests.
    let detectFacesRequest = DetectFaceRectanglesRequest()
    var qualityRequest = DetectFaceCaptureQualityRequest()
    var landmarksRequest = DetectFaceLandmarksRequest()

    /// Perform `DetectFaceRectanglesRequest` to locate all faces in the photo.
    let handler = ImageRequestHandler(photo)
    let faceObservations = try await handler.perform(detectFacesRequest)

    /// Set the faces that `DetectFaceLandmarksRequest` and `DetectFaceCaptureQualit
    landmarksRequest.inputFaceObservations = faceObservations
```

```
    qualityRequest.inputFaceObservations = faceObservations

    /// Perform `DetectFaceCaptureQualityRequest` and `DetectFaceLandmarksRequest` c
    let (qualityResults, landmarksResults) = try await handler.perform(qualityReques

    var score: Float = 0
    /// Set the capture-quality score of the photo if `Vision` detects one face.
    if qualityResults.count == 1 {
        score = qualityResults[0].captureQuality!.score
    /// Set the average capture-quality score if `Vision` detects multiple faces.
    } else if qualityResults.count > 1 {
        for face in qualityResults {
            score += face.captureQuality!.score
        }
        score /= Float(qualityResults.count)
    }

    return Selfie(photo: photo, score: score, landmarksResults: landmarksResults)
}
```

Analyzing a large collection of selfies with `Vision` requests can take time, so the app uses Swift concurrency to help with speed and efficiency. Using `TaskGroup`, the app processes the images in parallel. When the `processSelfie` method returns a new `Selfie` object, the app adds it to the `selfies` array. After the app processes all the images, it sorts the `selfies` array by capture-quality score. The function returns the new array of `Selfie` objects:

```
func processAllSelfies(photos: [Data]) async throws -> [Selfie] {
    var selfies = [Selfie]()

    try await withThrowingTaskGroup(of: Selfie.self) { group in
        for photo in photos {
            group.addTask {
                return try await processSelfie(photo: photo)
            }
        }

        /// Only add the photo to the `selfies` array if Vision detects a face.
        for try await selfie in group where selfie.facesDetected > 0 {
            selfies.append(selfie)
        }
    }

    /// Sort the selfies in descending order of their capture-quality scores.
```

```
    selfies.sort { $0.score > $1.score }

    return selfies
}
```

Display face rectangles

The sample provides custom `Shape` implementations to draw a rectangle around each face, and the face landmarks. For face rectangles, the app uses the `boundingBox` property on a `FaceObservation`. The `boundingBox` property contains the location and dimensions of the box in the form of a `NormalizedRect`. The sample converts the `NormalizedRect` to a `CGRect`, and returns a `Path` to draw the rectangle:

```
struct BoundingBox: Shape {
    private let normalizedRect: NormalizedRect

    init(observation: any BoundingBoxProviding) {
        normalizedRect = observation.boundingBox
    }

    func path(in rect: CGRect) -> Path {
        let rect = normalizedRect.toImageCoordinates(rect.size, origin: .upperLeft)
        return Path(rect)
    }
}
```

The sample creates a `BoundingBox` object for each face in the photo, and overlays them on the image:

```
.overlay {
    ForEach(selfie.landmarkResults, id: \.self) { observation in
        BoundingBox(observation: observation)
            .stroke(.red, lineWidth: 2)
    }
}
```

Display face landmarks

To create and display face landmarks on the image, the sample uses the custom `FaceLandmark` structure. Each `FaceObservation` from the `DetectFaceLandmarksRequest` contains a

collection of landmarks as regions. A region contains all the points the sample needs to draw the outline. The possible regions are `faceContour`, `innerLips`, `leftEye`, `leftEyebrow`, `leftPupil`, `medianLine`, `nose`, `noseCrest`, `outerLips`, `rightEye`, `rightEyebrow`, and `rightPupil`.

The sample converts a region's `NormalizedPoint` collection to a `CGPoint` collection, and draws a path from one point to the next. When it reaches the last point, the sample closes the path:

```
struct FaceLandmark: Shape {
    let region: FaceObservation.Landmarks2D.Region

    func path(in rect: CGRect) -> Path {
        let points = region.pointsInImageCoordinates(rect.size, origin: .upperLeft)
        let path = CGMutablePath()

        path.move(to: points[0])

        for index in 1..<points.count {
            path.addLine(to: points[index])
        }

        if region.pointsClassification == .closedPath {
            path.closeSubpath()
        }

        return Path(path)
    }
}
```

For each face `Vision` detects in an image, the sample creates `FaceLandmark` objects and overlays them on the image:

```
.overlay {
    ForEach(selfie.landmarksResults, id: \.self) { observation in
        FaceLandmark(region: observation.landmarks!.faceContour)
            .stroke(.white, lineWidth: 2)

        // ..
    }
}
```

See Also

Face and body detection

`struct DetectFaceRectanglesRequest`

A request that finds faces within an image.

`struct DetectFaceLandmarksRequest`

An image-analysis request that finds facial features like eyes and mouth in an image.

`struct DetectFaceCaptureQualityRequest`

A request that produces a floating-point number that represents the capture quality of a face in a photo.

`struct DetectHumanRectanglesRequest`

A request that finds rectangular regions that contain people in an image.