Sample Code

Tracking and visualizing faces

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

Download

iOS 11.0+ | iPadOS 11.0+ | Xcode 16.1+

Overview

This sample app presents a simple interface allowing you to choose between five augmented reality (AR) visualizations on devices with a TrueDepth front-facing camera.

- An overlay of x/y/z axes indicating the ARKit coordinate system tracking the face (and in iOS 12, the position and orientation of each eye).
- The face mesh provided by ARKit, showing automatic estimation of the real-world directional lighting environment, as well as a texture you can use to map 2D imagery onto the face.
- Virtual 3D content that appears to attach to (and interact with) the user's real face.
- Live camera video texture-mapped onto the ARKit face mesh, with which you can create effects that appear to distort the user's real face in 3D.
- A simple robot character whose facial expression animates to match that of the user, showing how to use ARKit's animation blend shape values to create experiences like the system Animoji app.

Use the tab bar to switch between these modes.

Important

Face tracking is available on all devices with Apple Neural Engine in iOS 14 & iPad OS 14, and requires a device with TrueDepth camera on iOS 13 & iPadOS 13 and below. ARKit is not available in iOS Simulator.

Start a face-tracking session in a scenekit view

Like other uses of ARKit, face tracking requires configuring and running a session (an `ARSession` object) and rendering the camera image together with virtual content in a view. This sample uses `ARSCNView` to display 3D content with SceneKit, but you can also use SpriteKit or build your own renderer using Metal (see `ARSKView` and Displaying an AR Experience with Metal.

Face tracking differs from other uses of ARKit in the class you use to configure the session. To enable face tracking, create an instance of `ARFaceTrackingConfiguration`, configure its properties, and pass it to the `run(_:options:)` method of the AR session associated with your view, as shown here:

```
guard ARFaceTrackingConfiguration.isSupported else { return }
let configuration = ARFaceTrackingConfiguration()
if #available(iOS 13.0, *) {
    configuration.maximumNumberOfTrackedFaces = ARFaceTrackingConfiguration.supporte
}
configuration.isLightEstimationEnabled = true
sceneView.session.run(configuration, options: [.resetTracking, .removeExistingAnchor
```

Track the Position and Orientation of a Face

When face tracking is active, ARKit automatically adds`ARFaceAnchor` objects to the running AR session, containing information about the user's face, including its position and orientation. (ARKit detects and provides information about only face at a time. If multiple faces are present in the camera image, ARKit chooses the largest or most clearly recognizable face.)

In a SceneKit-based AR experience, you can add 3D content corresponding to a face anchor in the `renderer(_:nodeFor:)` or `renderer(_:didAdd:for:)` delegate method. ARKit manages a SceneKit node for the anchor, and updates that node's position and orientation on each frame, so any SceneKit content you add to that node automatically follows the position and orientation of the user's face.

```
func renderer(_ renderer: SCNSceneRenderer, nodeFor anchor: ARAnchor) -> SCNNode? {
    // This class adds AR content only for face anchors.
    guard anchor is ARFaceAnchor else { return nil }

    // Load an asset from the app bundle to provide visual content for the anchor.
    contentNode = SCNReferenceNode(named: "coordinateOrigin")

    // Add content for eye tracking in iOS 12.
    self.addEyeTransformNodes()

    // Provide the node to ARKit for keeping in sync with the face anchor.
    return contentNode
}
```

This example uses a convenience extension on `SCNReferenceNode` to load content from an `.scn` file in the app bundle. The `renderer(_:nodeFor:)` method provides that node to `ARSCNView`, allowing ARKit to automatically adjust the node's position and orientation to match the tracked face.

Use face geometry to model the user's face

ARKit provides a coarse 3D mesh geometry matching the size, shape, topology, and current facial expression of the user's face. ARKit also provides the `ARSCNFaceGeometry` class, offering an easy way to visualize this mesh in SceneKit.

Your AR experience can use this mesh to place or draw content that appears to attach to the face. For example, by applying a semitransparent texture to this geometry you could paint virtual tattoos or makeup onto the user's skin.

To create a SceneKit face geometry, initialize an `ARSCNFaceGeometry` object with the Metal device your SceneKit view uses for rendering, and assign that geometry to the SceneKit node tracking the face anchor.

```
func renderer(_ renderer: SCNSceneRenderer, nodeFor anchor: ARAnchor) -> SCNNode? {
    guard let sceneView = renderer as? ARSCNView,
        anchor is ARFaceAnchor else { return nil }

    #if targetEnvironment(simulator)
    #error("ARKit is not supported in iOS Simulator. Connect a physical iOS device a
    #else
    let faceGeometry = ARSCNFaceGeometry(device: sceneView.device!)!
    let material = faceGeometry.firstMaterial!

    material.diffuse.contents = #imageLiteral(resourceName: "wireframeTexture") // E
    material.lightingModel = .physicallyBased

    contentNode = SCNNode(geometry: faceGeometry)
    #endif
    return contentNode
}
```

Note

This example uses a texture with transparency to create the illusion of colorful grid lines painted onto a real face. You can use the `wireframeTexture.png` image included with this sample code project as a starting point to design your own face textures.

ARKit updates its face mesh conform to the shape of the user's face, even as the user blinks, talks, and makes various expressions. To make the displayed face model follow the user's expressions, retrieve an updated face meshes in the `renderer(_:didUpdate:for:)` delegate callback, then update the `ARSCNFaceGeometry` object in your scene to match by passing the new face mesh to its `update(from:)` method:

```
func renderer(_ renderer: SCNSceneRenderer, didUpdate node: SCNNode, for anchor: ARA
    guard let faceGeometry = node.geometry as? ARSCNFaceGeometry,
        let faceAnchor = anchor as? ARFaceAnchor
        else { return }

    faceGeometry.update(from: faceAnchor.geometry)
}
```

Place 3D content on the user's face

Another use of the face mesh that ARKit provides is to create *occlusion geometry* in your scene. An occlusion geometry is a 3D model that doesn't render any visible content (allowing the camera image to show through), but obstructs the camera's view of other virtual content in the scene.

This technique creates the illusion that the real face interacts with virtual objects, even though the face is a 2D camera image and the virtual content is a rendered 3D object. For example, if you place an occlusion geometry and virtual glasses on the user's face, the face can obscure the frame of the glasses.

To create an occlusion geometry for the face, start by creating an `ARSCNFaceGeometry` object as in the previous example. However, instead of configuring that object's SceneKit material with a visible appearance, set the material to render depth but not color during rendering:

```
let faceGeometry = ARSCNFaceGeometry(device: sceneView.device!)!
faceGeometry.firstMaterial!.colorBufferWriteMask = []
occlusionNode = SCNNode(geometry: faceGeometry)
occlusionNode.renderingOrder = -1
```

Because the material renders depth, other objects rendered by SceneKit correctly appear in front of it or behind it. But because the material doesn't render color, the camera image appears in its place.

The sample app combines this technique with a SceneKit object positioned in front of the user's eyes, creating an effect where the user's nose realistically obscures the object. This object uses physically-based materials, so it automatically benefits from the real-time directional lighting information that `ARFaceTrackingConfiguration` provides.

> Note
>
> The `ARFaceGeometry.obj` file included in this sample project represents ARKit's face geometry in a neutral pose. You can use this as a template to design your own 3D art assets for placement on a real face.

Map camera video onto 3D face geometry

For additional creative uses of face tracking, you can texture-map the live 2D video feed from the camera onto the 3D geometry that ARKit provides. After mapping pixels in the camera video onto the corresponding points on ARKit's face mesh, you can modify that mesh, creating illusions such as resizing or distorting the user's face in 3D.

First, create an `ARSCNFaceGeometry` for the face and assign the camera image to its main material. `ARSCNView` automatically sets the scene's `background` material to use the live video feed from the camera, so you can set the geometry to use the same material.

```
// Show video texture as the diffuse material and disable lighting.
let faceGeometry = ARSCNFaceGeometry(device: sceneView.device!, fillMesh: true)!
let material = faceGeometry.firstMaterial!
material.diffuse.contents = sceneView.scene.background.contents
material.lightingModel = .constant
```

To correctly align the camera image to the face, you'll also need to modify the texture coordinates that SceneKit uses for rendering the image on the geometry. One easy way to perform this mapping is with a SceneKit shader modifier (see the `SCNShadable` protocol). The shader code here applies the coordinate system transformations needed to convert each vertex position in the mesh from 3D scene space to the 2D image space used by the video texture:

```
// Transform the vertex to the camera coordinate system.
float4 vertexCamera = scn_node.modelViewTransform * _geometry.position;

// Camera projection and perspective divide to get normalized viewport coordinates (
float4 vertexClipSpace = scn_frame.projectionTransform * vertexCamera;
vertexClipSpace /= vertexClipSpace.w;

// XY in clip space is [-1,1]x[-1,1], so adjust to UV texture coordinates: [0,1]x[0,
// Image coordinates are Y-flipped (upper-left origin).
float4 vertexImageSpace = float4(vertexClipSpace.xy * 0.5 + 0.5, 0.0, 1.0);
vertexImageSpace.y = 1.0 - vertexImageSpace.y;

// Apply ARKit's display transform (device orientation * front-facing camera flip).
float4 transformedVertex = displayTransform * vertexImageSpace;

// Output as texture coordinates for use in later rendering stages.
_geometry.texcoords[0] = transformedVertex.xy;
```

When you assign a shader code string to the `geometry` entry point, SceneKit configures its renderer to automatically run that code on the GPU for each vertex in the mesh. This shader code also needs to know the intended orientation for the camera image, so the sample gets that from the ARKit `displayTransform(for:viewportSize:)` method and passes it to the shader's `displayTransform` argument:

```
// Pass view-appropriate image transform to the shader modifier so
// that the mapped video lines up correctly with the background video.
let affineTransform = frame.displayTransform(for: .portrait, viewportSize: sceneView
let transform = SCNMatrix4(affineTransform)
faceGeometry.setValue(SCNMatrix4Invert(transform), forKey: "displayTransform")
```

Note

This example's shader modifier also applies a constant scale factor to all vertices, causing the user's face to appear larger than life. Try other transformations to distort the face in other ways.

Animate a character with blend shapes

In addition to the face mesh shown in the earlier examples, ARKit also provides a more abstract representation of the user's facial expressions. You can use this representation (called *blend shapes*) to control animation parameters for your own 2D or 3D assets, creating a character that follows the user's real facial movements and expressions.

As a basic demonstration of blend shape animation, this sample includes a simple model of a robot character's head, created using SceneKit primitive shapes. (See the `robotHead.scn` file in the source code.)

To get the user's current facial expression, read the `blendShapes` dictionary from the face anchor in the `renderer(_:didUpdate:for:)` delegate callback. Then, examine the key-value pairs in that dictionary to calculate animation parameters for your 3D content and update that content accordingly.

```
let blendShapes = faceAnchor.blendShapes
guard let eyeBlinkLeft = blendShapes[.eyeBlinkLeft] as? Float,
    let eyeBlinkRight = blendShapes[.eyeBlinkRight] as? Float,
    let jawOpen = blendShapes[.jawOpen] as? Float
    else { return }
eyeLeftNode.scale.z = 1 - eyeBlinkLeft
eyeRightNode.scale.z = 1 - eyeBlinkRight
jawNode.position.y = originalJawY - jawHeight * jawOpen
```

There are more than 50 unique `ARFaceAnchor.BlendShapeLocation` coefficients, of which your app can use as few or as many as necessary to create the artistic effect you want. In this sample, the `BlendShapeCharacter` class performs this calculation, mapping the `eyeBlinkLeft` and `eyeBlinkRight` parameters to one axis of the `scale` factor of the robot's eyes, and the `jawOpen` parameter to offset the position of the robot's jaw.

See Also

Face Tracking

{} Combining user face-tracking and world tracking

Track the user’s face in an app that displays an AR experience with the rear camera.

`class ARFaceAnchor`

An anchor for a unique face that is visible in the front-facing camera.