第 5 章

人脸检测跟踪

在计算机人工智能(Artificial Intelligence,AI)物体检测识别领域,最先研究的是人脸检测识别,目前技术发展最成熟的也是人脸检测识别。人脸检测识别已经广泛应用于安防、机场、车站、闸机、人流控制、安全支付等众多社会领域,也广泛应用于直播特效、美颜、Animoji 等娱乐领域。

5.1 人脸检测基础

ARKit 支持人脸检测,并且支持多人脸同时检测,还支持表情属性和 BlendShapes。但需要注意的是,ARKit 人脸检测跟踪需要配备前置深度摄像头(TrueDepth Camera)或者 A12 及以上处理器的设备,因此,并不是所有 iPhone/iPad 设备都支持人脸检测,目前支持 ARKit 人脸检测的设备如表 5-1 所示。

表 5-1 支持人脸检测的设备

移 动 设 备	支持的设备
iPhone	iPhone X
	iPhone Xr、iPhone Xs、iPhone Xs Max
	iPhone 11、iPhone 11 Pro、iPhone 11 Pro Max
	iPhone SE2
	iPhone 12mini、iPhone 12、iPhone 12 Pro、iPhone 12 Pro Max
iPad	iPad Pro 12.9 英寸第 1 代、第 2 代、第 3 代、第 4 代
	iPad Pro 11 英寸
	iPad Pro 10.5 英寸
	iPad Pro 9.7 英寸

5.1.1 人脸检测概念

人脸检测(Face Detection)是利用计算机视觉处理技术在数字图像或视频中自动定位人脸的过程,人脸检测不仅检测人脸在图像或视频中的位置,还应该检测出其大小与方向(姿态)。人脸检测是有关人脸图像分析应用的基础,包括人脸识别和验证、监控场合的人脸跟踪、面部表情分析、面部属性识别(性别、年龄、微笑、痛苦)、面部光照调整和变形、面部形状重建、图像视频检索等。近几年,随着机器学习技术的发展,人脸检测成功率与准确率大幅度提高,并开始大规模实用,如机场和火车站人脸验票、人脸识别身份认证等。

人脸识别(Face Recognition)是指利用人脸检测技术确定两张人脸是否对应同一个人,人脸识别技术是人脸检测技术的扩展和应用,也是很多其他应用的基础。目前,ARKit 仅提供人脸检测,而不提供人脸识

别功能。

人脸跟踪(Face Tracking)是指将人脸检测扩展到视频序列,跟踪同一张人脸在视频序列中的位置。理论上讲,任何出现在视频中的人脸都可以被跟踪,也即是说,在连续视频帧中检测到的人脸可以被识别为同一个人。人脸跟踪不是人脸识别的一种形式,它是根据视频序列中人脸的位置和运动推断不同视频帧中的人脸是否为同一人的技术。

人脸检测属于模式识别的一类,但人脸检测成功率受到很多因素的影响,影响人脸检测成功率的因素主要有表 5-2 中所述情形。

表 5-2 影响人脸检测成功率的部分因素

术 语	描 述 说 明
图像大小	人脸图像过小会影响检测效果,人脸图像过大会影响检测速度,图像大小反映在实际应用场景中就是人脸离摄像头的距离
图像分辨率	越低的图像分辨率越难检测,图像大小与图像分辨率直接影响摄像头识别距离。目前 4K 摄像机看清人脸的最远距离是 10m 左右,移动手机检测距离更小一些
光照环境	过亮或过暗的光照环境都会影响人脸检测效果
模糊程度	实际场景中主要是运动模糊,人脸相对于摄像机的移动经常会产生运动模糊
遮挡程度	五官无遮挡、脸部边缘清晰的图像有利于人脸检测。有遮挡的人脸会对人脸检测成功率造成影响
采集角度	人脸相对于摄像机角度不同也会影响人脸检测效果。正脸最有利于检测,偏离角度越大越不利于检测

随着人工智能技术的持续发展,在全球信息化、云计算、大数据的支持下,人脸检测识别技术也会越来越成熟,同时应用面会越来越广,可以预见,以人脸检测为基础的人脸识别将会呈现网络化、多识别融合、云互联的发展趋势。

5.1.2 人脸检测技术基础

人体头部是一个三维结构体,而眼、嘴、额头在这个三维结构体中又有比较固定的位置,因此在 ARKit 中使用了两个坐标系来处理与人体头部相关的工作,一个是世界坐标系(World Coordinates Space),这个坐标系就是 ARKit 启动时建立的以启动时设备所在位置为原点的坐标系,而另一个称为人脸坐标系(Face Coordinate Space)。

在 ARKit 检测到人脸后会生成一个 ARFaceAnchor,其 transform 属性指定了相对于世界坐标系的人脸位置与方向,利用该属性就可以在人脸上挂载虚拟元素。除此之外,ARKit 还会生成一个相对于人体头部的坐标系,该坐标系也以米为测量单位,利用该坐标系可以更精细地定位眼、嘴、鼻等位置从而实现更好的虚拟元素定位效果。ARKit 人脸坐标系也采用右手坐标系,如图 5-1 所示。

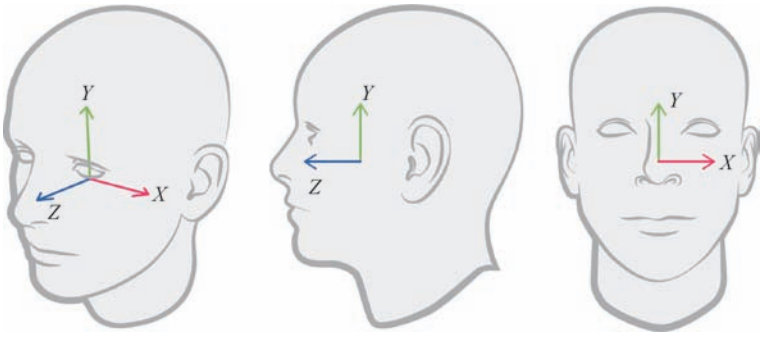

图 5-1 ARKit 人脸坐标系示意图

在人脸坐标系中，X 轴正向指向观察者的右侧（检测到人脸的左侧），Y 轴向上（这里的向上相对于人脸而不是世界坐标系），Z 轴指向人脸面向方向。

人脸检测技术的复杂性之一是人体头部是一个三维结构体，并且是一个动态的三维结构体，摄像机捕捉到的人脸图像很多时候都不是正面，而是有一定角度且时时处于变化中的侧面。当然，人脸检测的有利条件是人脸有很多特征，如图 5-2 所示，可以利用这些特征做模式匹配。但需要注意的是，在很多人脸检测算法中，人脸特征并不是人脸轮廓检测的前提，换句话说，人脸检测是独立于人脸特征的，且通常是先检测出人脸轮廓再进行特征检测，因为特征检测需要花费额外的时间，会对人脸检测效率产生影响。

图 5-2　人脸特征点示意图

人脸结构具有对称性，人脸特征会分布在 Y 轴两侧一定角度内，通常来说，人脸特征分布情况符合表 5-3 所示规律。

表 5-3　人脸特征分布情况

Y 欧拉角	人 脸 特 征	Y 欧拉角	人 脸 特 征
小于－36°	左眼、左嘴角、左耳、鼻底、左脸颊	12°～36°	右眼、右嘴角、鼻底、下嘴唇、右脸颊
－36°～－12°	左眼、左嘴角、鼻底、下嘴唇、左脸颊	大于 36°	右眼、右嘴角、右耳、鼻底、右脸颊
－12°～12°	左嘴角、右嘴角、上下嘴唇、鼻底		

人脸检测不仅需要找出人脸轮廓，还需要检测出人脸姿态（包括人脸位置和面向方向）。为了解决人脸姿态问题，一般的做法是制作一个三维人脸正面“标准模型”，这个模型需要非常精细，因为它将影响到人脸姿态估计的精度。有了这个三维标准模型之后，对人脸姿态检测的思路是在检测到人脸轮廓后对标准模型进行旋转，以期标准模型上的特征点与检测到的人脸特征点重合匹配。从这个思路可以看到，对姿态的检测其实是个不断尝试的过程，选取特征点吻合得最好的标准模型姿态作为人脸姿态。形象地说，就是先制作一个人皮面具，努力尝试将人皮面具套在人脸上，如果成功则人皮面具的姿态必定是人脸姿态。

如前所述，虽然人脸的结构具有稳定性，还有很多特征点可供校准，但由于姿态和表情的变化、不同人的外观差异、光照、遮挡等影响，准确地检测处于各种条件下的人脸仍然是较为困难的事情。幸运的是，随着深度神经网络的发展，在一般环境条件下，目前人脸检测准确率有了非常大的提高，甚至在某些条件下超过了人类。

ARKit 人脸检测由于使用了深度摄像头（或者带神经处理单元 NPU 的设备），检测精度非常高，除了具有通常意义下的人脸检测功能，还具备一些独特的功能特性，具体功能特性如表 5-4 所示。

表 5-4　ARKit 人脸检测跟踪功能

序号	描　　述
1	检测人脸位置与方向
2	提供检测到人脸的几何网络（ARFaceGeometry），因此可以渲染人脸模型
3	提供双眼姿态，因此可以独立地跟踪每一只眼睛
4	基于人脸位置，可以在检测到的人脸上挂载虚拟元素（贴纸或者模型）

续表

序号	描　　述
5	可以从检测到的人脸评估环境光照信息
6	支持 BlendShape，因此可以实现利用表情驱动模型功能
7	提供人眼凝视（LookAtPoint），因此可以实现眼动控制
8	允许在人脸检测跟踪的同时启用世界跟踪（World Tracking）

5.2　人脸检测配置

ARKit 可以使用人脸跟踪（ARFaceTrackingConfiguration）和世界跟踪（ARWorldTrackingConfiguration）两种配置方式开启人脸检测跟踪功能。

ARWorldTrackingConfiguration 配置中有一个 userFaceTrackingEnabled 属性，该属性为布尔值，默认为 false，如果设置为 true，则可以在进行世界跟踪的同时启动人脸检测跟踪。

ARFaceTrackingConfiguration 是专为人脸检测跟踪优化的配置，其中，maximumNumberOfTrackedFaces 属性用于设置最大同时检测跟踪的人脸数，当前最大值为 3；isWorldTrackingEnabled 设置是否在人脸检测跟踪的同时启动世界跟踪，isLightEstimationEnabled 设置是否启用环境光照评估。

典型的启动人脸检测跟踪功能的代码如代码清单 5-1 所示。

代码清单 5-1

```
1.  guard ARFaceTrackingConfiguration.isSupported else { return }
2.  let configuration = ARFaceTrackingConfiguration()
3.  configuration.isLightEstimationEnabled = true
4.  arView.session.run(configuration, options: [.resetTracking, .removeExistingAnchors])
```

由于并非所有支持 ARKit 的设备都支持人脸检测跟踪，因此在开启人脸检测跟踪之前，首先应当检测用户设备是否支持人脸检测，如果支持，再设置诸如 isLightEstimationEnabled、maximumNumberOfTrackedFaces 等属性，然后启动 ARSession。

5.2.1　人脸网格

除了人脸姿态，ARKit 还提供了每个已检测到的人脸网格（ARFaceGeometry），该网络包含 1220 个顶点，网格数据包括顶点（vertices）、索引（triangleIndices）、三角形数量（triangleCount）、纹理坐标（textureCoordinates）等相关信息，如图 5-3 所示。利用人脸网格，开发者就可以渲染出人脸形状，或者对人脸网络进行自定义贴图等。

图 5-3　人脸网格示意图

提示

人脸网格顶点总数不会变，每个检测到的人脸都会生成一张 1220 个顶点的网格，因此 vertexCount、textureCoordinateCount、triangleCount 都不会变，对每一张特定的人脸网格，变化的只是顶点（vertices）的位置，因为 ARKit 会根据用户面部形状与表情调整网格。

到目前为止，RealityKit 并不支持人脸网格几何生成与渲染，本节我们将使用 SceneKit 进行演示，但是由于本书的主题，我们只关注与人脸网格相关处理，其他 SceneKit 相关技术细节，需读者自行查阅 SceneKit 资料。

ARKit 会根据每个检测到的人脸提供与之相应形状、尺寸、表情的网格信息，在使用 SceneKit 渲染人脸网格时，有 3 个类非常重要：ARFaceAnchor、ARFaceGeometry、ARSCNFaceGeometry。

ARFaceAnchor 继承自 ARAnchor，是专门用于锚定人脸的锚点，其 transform 属性指定相对于世界坐标系的人脸位置与方向，利用它就可以锚定生成的人脸网格。

ARFaceGeometry 包含 ARKit 生成的人脸网格信息，包括顶点、索引、UV 坐标等所有信息。

ARSCNFaceGeometry 则是利用 ARFaceGeometry 网格数据生成 SCNGeometry，可以直接作为 SceneKit 场景中的节点。

检测与渲染人脸网格的典型代码如代码清单 5-2 所示。

代码清单 5-2

```
import UIKit
import SceneKit
import ARKit

class ViewController: UIViewController {
    @IBOutlet var sceneView: ARSCNView!
    override func viewDidLoad() {
        super.viewDidLoad()
        guard ARFaceTrackingConfiguration.isSupported else {
            fatalError("当前设备不支持人脸检测!")
        }
        sceneView.delegate = self
    }

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        let configuration = ARFaceTrackingConfiguration()
        configuration.isLightEstimationEnabled = true
        configuration.providesAudioData = false
        configuration.isWorldTrackingEnabled = false
        configuration.maximumNumberOfTrackedFaces = 1
        sceneView.session.run(configuration)
    }

    override func viewWillDisappear(_ animated: Bool) {
        super.viewWillDisappear(animated)
        sceneView.session.pause()
    }
}

extension ViewController: ARSCNViewDelegate {
    func renderer(_ renderer: SCNSceneRenderer, nodeFor anchor: ARAnchor) -> SCNNode? {
        guard let device = sceneView.device else {return nil }
```

```
        let faceGeometry = ARSCNFaceGeometry(device: device)
        let node = SCNNode(geometry: faceGeometry)
        node.geometry?.firstMaterial?.fillMode = .lines
        return node
    }

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

}
```

在代码清单 5-2 中，首先检查了当前设备对人脸检测的支持情况，然后使用 ARFaceTrackingConfiguration 配置并运行了人脸检测 ARSession，当 ARKit 检测到人脸时，我们将从 ARSCNFaceGeometry 对象得到的人脸几何网格并使用线框的渲染模式进行渲染，检测效果如图 5-4 左图所示。

图 5-4　分别使用线框与纹理贴图渲染检测到的人脸网格示意图

在 AR 应用运行时，ARKit 会根据检测到的人脸方向、表情实时更新人脸网格，为显示出人脸网格的实时变化，我们使用 renderer(_:didUpdate:for:)代理方法对人脸网格进行了实时更新。

检测到的人脸网格不仅包括几何顶点信息，也包括 UV 坐标信息，因此，我们不仅可以以线框模式渲染网格，还可以使用静态、动态的纹理贴图进行渲染，只需要将代码清单 5-2 中 renderer(: nodeFor:)代理方法中线框渲染模式变更为使用材质纹理，典型代码如代码清单 5-3 所示。

代码清单 5-3

```
guard let device = sceneView.device else {return nil }
let faceGeometry = ARSCNFaceGeometry(device: device)
```

```
let node = SCNNode(geometry: faceGeometry)
let material = node.geometry?.firstMaterial!
material?.diffuse.contents = "face.scnassets/face.png"
node.geometry?.firstMaterial?.fillMode = .fill
```

使用线框模式与使用纹理贴图模式渲染的人脸网格如图 5-4 所示，利用 ARKit 人脸网格贴图可以实现很多有意思的贴纸效果，如腮红、口红、额纹等，在电子商务试妆方面也可以应用。

ARKit 只提供人脸检测功能，并不支持人脸识别，如果需要此功能，还需要结合其他技术共同完成。

5.2.2　挂载虚拟元素

在 RealityKit 中，在检测到的人脸面部挂载虚拟元素的实现方式有两种：一种是通过遵循 ARSessionDelegate 协议，执行 session(_ session: ARSession, didAdd anchors: [ARAnchor])方法，在获取的 ARFaceAnchor 上挂载虚拟元素；另一种是与 Reality Composer 结合使用。

在使用第一种方式时，可以利用 ARFaceAnchor 初始化一个 AnchorEntity 类型实例，这样，ARFaceAnchor 的姿态信息就可以直接被使用，典型的使用代码如代码清单 5-4 所示。

代码清单 5-4

```
public func session(_ session: ARSession, didAdd anchors: [ARAnchor]) {
    for anchor in anchors {
        guard let anchor = anchor as? ARFaceAnchor else { continue }
        do {
            let faceEntity =  try Entity.load(named: "toy_biplane.usdz")
            let faceAnchor = AnchorEntity(anchor: anchor)
            faceAnchor.addChild(faceEntity)
            self.scene.addAnchor(faceAnchor)
        } catch {
            print("找不到文件")
        }
    }
}
```

在检测到的人脸上挂载虚拟元素使用 RealityKit 与 Reality Composer 结合的方式更方便直观，特别是需要在很多虚拟元素之间进行切换时，可以大大简化代码逻辑。使用第二种方式的操作步骤如下：

(1) 打开 Reality Composer，并创建一个锚定到人脸的工程(Reality Composer 具体操作参阅第 10 章)，如图 5-5 所示。

(2) 导入需要挂载的 USDZ 或者 Reality 模型文件并调整到参考人脸理想的位置，然后给场景命名(命名时建议使用英文字母或者英文字母与数字组合，方便在 RealityKit 中调用)，如图 5-6 所示。

(3) 在 Reality Composer 菜单中依次选择“文件”→“保存”(或者使用快捷键 Command+S)保存工程为 FaceMask.rcproject 文件(工程名根据需要自行命名)。

(4) 使用 RealityKit 加载工程文件到内存，直接获取工程文件中的锚点信息并将其作为 ARAnchor 添加到 ARVeiw.scene 场景中即可。这里需要注意的是，ARKit 会在检测到人脸后自动在指定的位置挂载虚拟元素，但 ARKit 并不会自动运行人脸检测的 ARSession，因此，需要手动运行人脸检测的 ARSession 以开启人脸检测功能，典型代码如代码清单 5-5 所示。

图 5-5　在 Reality Composer 中创建锚定到人脸的工程

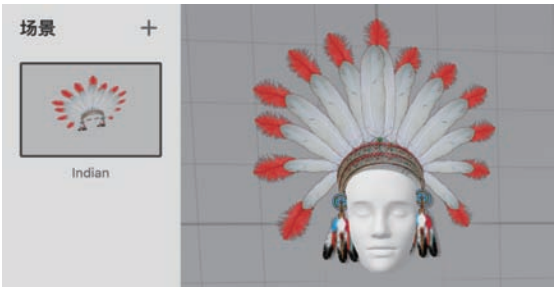

图 5-6　调整模型到参考人脸合适位置并重命名场景

代码清单 5-5

```
struct ARViewContainer: UIViewRepresentable {
    func makeUIView(context: Context) -> ARView {
        let arView = ARView(frame: .zero)
        if ARFaceTrackingConfiguration.isSupported {
            let faceConfig = ARFaceTrackingConfiguration()
            faceConfig.maximumNumberOfTrackedFaces = 1
            arView.session.delegate = arView
            let faceAnchor = try! FaceMask.loadGlass1()
            arView.addGuesture()
            arView.scene.addAnchor(faceAnchor)
            arView.session.run(faceConfig, options: [.resetTracking,.removeExistingAnchors])
        }
        return arView
    }
    func updateUIView(_ uiView: ARView, context: Context) {
        uiView.scene.anchors.removeAll()
    }
}
var faceMaskCount = 0
let numberOfMasks = 6

extension ARView : ARSessionDelegate{
    func addGuesture(){
        let gesture = UISwipeGestureRecognizer()
        gesture.addTarget(self, action: #selector(changeGlass(gesture:)))
        self.addGestureRecognizer(gesture)
    }

    @objc func changeGlass(gesture: UISwipeGestureRecognizer){
        faceMaskCount += 1
        faceMaskCount %= numberOfMasks
        switch faceMaskCount {
        case 0:
            let g = try! FaceMask.loadGlass2()
            self.scene.anchors.removeAll()
```

```
            self.scene.addAnchor(g)
        case 1:
            let g = try! FaceMask.loadIndian()
            self.scene.anchors.removeAll()
            self.scene.addAnchor(g)
        ...
       case 5:
            let g = try! FaceMask.loadFaceMesh()
            self.scene.anchors.removeAll()
            self.scene.addAnchor(g)
        default:
            break
        }
    }
}
```

在代码清单 5-5 中，首先检查设备对人脸检测的支持情况，在设备支持时运行人脸检测配置开启人脸检测功能，然后加载由 Reality Composer 配置好的虚拟模型。本示例我们在 Reality Composer 中创建了多个场景，每一个场景使用了一个虚拟元素，为方便切换不同的虚拟元素，我们使用了滑动手势控制场景切换，实现的效果如图 5-7 所示。

图 5-7　在检测到的人脸上挂载虚拟元素效果图

5.3　BlendShapes

苹果公司在 iPhone X 及后续机型上增加了一个深度相机（TrueDepth Camera），利用这个深度相机可以更加精准捕捉用户的面部表情，提供更详细的面部特征点信息。在 ARKit 4 后，利用机器学习技术对前置摄像头捕获的图像进行处理，使不具备深度相机的设备也可以进行人脸检测和 BlendShapes（需要 A12 及以上处理器）。

5.3.1 BlendShapes 基础

利用前置摄像头采集到的用户面部表情特征，ARKit 提供了一种更加抽象的表示面部表情的方式，这种表示方式叫作 BlendShapes，BlendShapes 可以翻译成形状融合，在 3ds Max 中也叫变形器，这个概念原本用于描述通过参数控制模型网格的位移，苹果公司借用了这个概念，在 ARKit 中专门用于表示通过人脸表情因子驱动模型的技术。BlendShapes 在技术上是一组存储了用户面部表情特征运动因子的字典，共包含 52 组特征运动数据，ARKit 会根据摄像机采集的用户表情特征值实时地设置对应的运动因子。利用这些运动因子可以驱动 2D 或者 3D 人脸模型，这些模型即可呈现与用户一致的表情。

ARKit 实时提供全部 52 组运动因子，这 52 组运动因子中包括 7 组左眼运动因子数据、7 组右眼运动因子数据、27 组嘴与下巴运动因子数据、10 组眉毛脸颊鼻子运动因子数据、1 组舌头运动因子数据。但在使用时可以选择利用全部或者只利用其中的一部分，如只关注眼睛运动，则只利用眼睛相关运动因子数据即可。

每一组运动因子表示一个 ARKit 识别的人脸表情特征，每一组运动因子都包括一个表示人脸特定表情的定位符与一个表示表情程度的浮点类型值，表情程度值的范围为[0,1]，其中 0 表示没有表情，1 表示完全表情。如图 5-8 所示，在图中，这个表情定位符为 mouthSmileRight，代表右嘴角的表情定位，左图中表情程度值为 0，即没有任何右嘴角表情，右图中表情值为 1，即为最大的右嘴角表情运动，而 0 到 1 之间的中间值则会对网格进行融合，形成一种过渡表情，这也是 BlendShapes 名字的由来。ARKit 会实时捕捉到这些运动因子，利用这些运动因子我们可以驱动 2D、3D 人脸模型，这些模型会同步用户的面部表情，当然，我们可以只取其中的一部分所关注的运动因子，但由于人脸表情通常与若干组表情因子相关联，如果想精确地模拟用户的表情，建议使用全部运动因子数据。

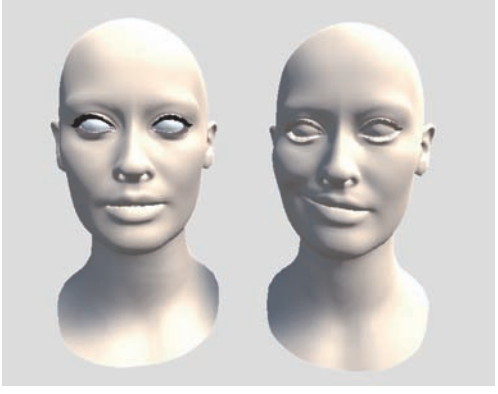

图 5-8 运动因子对人脸表情的影响

5.3.2 BlendShapes 技术原理

在 ARKit 中，对人脸表情特征信息定义了 52 组运动因子数据，其使用 BlendShapeLocation 作为表情定位符，表情定位符定义了特定表情，如 mouthSmileLeft、mouthSmileRight 等，与其对应的运动因子则表示表情程度，这 52 组运动因子数据如表 5-5 所示。

表 5-5 BlendShapeLocation 表情定位符及其描述

区　域	表情定位符	描　述
Left Eye(7)	eyeBlinkLeft	左眼眨眼
	eyeLookDownLeft	左眼目视下方
	eyeLookInLeft	左眼注视鼻尖
	eyeLookOutLeft	左眼向左看
	eyeLookUpLeft	左眼目视上方
	eyeSquintLeft	左眼眯眼
	eyeWideLeft	左眼睁大

续表

区　域	表情定位符	描　述
Right Eye(7)	eyeBlinkRight	右眼眨眼
	eyeLookDownRight	右眼目视下方
	eyeLookInRight	右眼注视鼻尖
	eyeLookOutRight	右眼向左看
	eyeLookUpRight	右眼目视上方
	eyeSquintRight	右眼眯眼
	eyeWideRight	右眼睁大
Mouth and Jaw(27)	jawForward	努嘴时下巴向前
	jawLeft	撇嘴时下巴向左
	jawRight	撇嘴时下巴向右
	jawOpen	张嘴时下巴向下
	mouthClose	闭嘴
	mouthFunnel	稍张嘴并双唇张开
	mouthPucker	抿嘴
	mouthLeft	向左撇嘴
	mouthRight	向右撇嘴
	mouthSmileLeft	左撇嘴笑
	mouthSmileRight	右撇嘴笑
	mouthFrownLeft	左嘴唇下压
	mouthFrownRight	右嘴唇下压
	mouthDimpleLeft	左嘴唇向后
	mouthDimpleRight	右嘴唇向后
	mouthStretchLeft	左嘴角向左
	mouthStretchRight	右嘴角向右
	mouthRollLower	下嘴唇卷向里
	mouthRollUpper	下嘴唇卷向上
	mouthShrugLower	下嘴唇向下
	mouthShrugUpper	上嘴唇向上
	mouthPressLeft	下嘴唇压向左
	mouthPressRight	下嘴唇压向右
	mouthLowerDownLeft	下嘴唇压向左下
	mouthLowerDownRight	下嘴唇压向右下
	mouthUpperUpLeft	上嘴唇压向左上
	mouthUpperUpRight	上嘴唇压向右上
Eyebrows(5)	browDownLeft	左眉向外
	browDownRight	右眉向外
	browInnerUp	蹙眉
	browOuterUpLeft	左眉向左上
	browOuterUpRight	右眉向右上
Cheeks(3)	cheekPuff	脸颊向外
	cheekSquintLeft	左脸颊向上并回旋
	cheekSquintRight	右脸颊向上并回旋

续表

区　域	表情定位符	描　述
Nose(2)	noseSneerLeft	左蹙鼻子
	noseSneerRight	右蹙鼻子
Tongue(1)	tongueOut	吐舌头

需要注意的是，在表 5-5 中表情定位符的命名是基于人脸方向的，如 eyeBlinkRight 定义的是人脸右眼眨眼，但在呈现 3D 模型时我们镜像了模型，看到的人脸模型右眼其实在左边。

有了表情特征运动因子后，就可以使用 SceneKit 中的 SCNMorpher. SetWeight()方法进行网格融合，该方法原型为：

```
setWeight(_ weight: CGFloat, forTargetNamed targetName: Int);
```

该方法有两个参数，forTargetNamed 参数为需要融合的网格变形器名，即上文中的 BlendShapeLocation 名；weight 参数为需要设置的 BlendShape 权重值，取值范围为[0,1]。

该方法主要功能是用于设置指定网格变形器的 BlendShape 权重值，这个值表示从源网格到目标网格的过渡(源网格与目标网格拥有同样的拓扑结构，但顶点位置两者有差异)，最终值符合以下公式：

$$v_{fin} = (1 - value) * v_{src} + value * v_{des}$$

因此，通过设置网格的 BlendShape 权重值可以将网格从源网格过渡到目标网格，如图 5-9 所示。

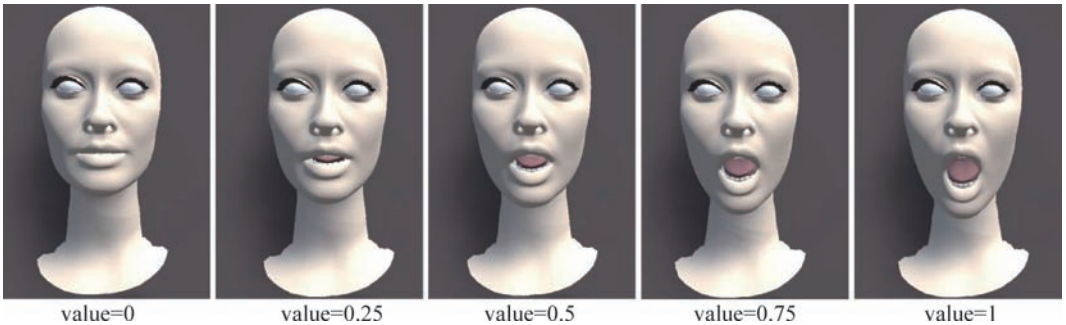

图 5-9　BlendShape 权重值对网格的影响

5.3.3　BlendShapes 使用

使用 ARKit 的 BlendShapes 功能需要满足两个条件：第一是有一个配备有深度相机或者 A12 及以上处理器的移动设备；第二是有一个 BlendShapes 已定义好的模型，为简化操作，这个模型的 BlendShapes 名称定义应与表 5-5 完全对应。为模型添加 BlendShapes 可以在 3ds Max 软件中定义变形器，并做好对应的网格变形。

在满足以上两个条件后，使用 BlendShapes 就变得相对简单了，实现的思路如下：

(1) 获取 ARKit 表情特征运动因子。这可以通过检查 ARFaceAnchor 获取相应数据，在检测到人脸时，ARFaceAnchor 会返回一个 blendShapes 集合，该集合包含所有 52 组表情特征运动因子数据。

(2) 绑定 ARKit 的表情特征定位符与模型中的变形器，使其保持一致。

(3) 当人脸 ARFaceAnchor 发生更新时，实时更新所有与表情特征运动因子相关联的模型变形器。

核心示例代码如代码清单 5-6 所示。

代码清单 5-6

```
import UIKit
import SceneKit
import ARKit

class ViewController: UIViewController {
    @IBOutlet var sceneView: ARSCNView!
    var contentNode: SCNReferenceNode?
    private lazy var head = contentNode!.childNode(withName: "head", recursively: true)!

    override func viewDidLoad() {
        super.viewDidLoad()
        guard ARFaceTrackingConfiguration.isSupported else {
            fatalError("当前设备不支持人脸检测!")
        }
        sceneView.delegate = self
    }

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        let configuration = ARFaceTrackingConfiguration()
        configuration.isLightEstimationEnabled = true
        configuration.providesAudioData = false
        configuration.isWorldTrackingEnabled = false
        configuration.maximumNumberOfTrackedFaces = 1
        sceneView.autoenablesDefaultLighting = true
        sceneView.allowsCameraControl = false
        sceneView.session.run(configuration)

    }
    override func viewWillDisappear(_ animated: Bool) {
        super.viewWillDisappear(animated)
        sceneView.session.pause()
    }
}

extension ViewController: ARSCNViewDelegate {
   func modelSetup() {
      if let filePath = Bundle.main.path(forResource: "BlendShapeFace", ofType: "scn") {
          let referenceURL = URL(fileURLWithPath: filePath)
          self.contentNode = SCNReferenceNode(url: referenceURL)
          self.contentNode?.load()
          self.head.morpher?.unifiesNormals = true
          self.contentNode?.scale = SCNVector3(0.01,0.01,0.01)
          self.contentNode?.position.y += 0.02
       }
    }
   func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNode, for anchor: ARAnchor) {
       guard let faceAnchor = anchor as? ARFaceAnchor else { return }
```

```
49.         modelSetup()
50.         node.addChildNode(self.contentNode!)
51.     }
52.
53.     func renderer(_ renderer: SCNSceneRenderer, didUpdate node: SCNNode,for anchor: ARAnchor) {
54.         guard let faceAnchor = anchor as? ARFaceAnchor else { return }
55.         DispatchQueue.main.async {
56.             for (key, value) in faceAnchor.blendShapes {
57.                 if let fValue = value as? Float {
58.                     self.head.morpher?.setWeight(CGFloat(fValue), forTargetNamed: key.rawValue)
59.                 }
60.             }
61.         }
62.     }
63. }
```

实现 BlendShapes 核心逻辑很清晰，即使用检测到的人脸表情驱动模型对应的表情，因此人脸表情与模型变形器(BlendShapes)必须建立一一对应关系，我们可以选择手动逐个绑定，也可以在建模时将变形器名与 ARKit 中的 BlendShapeLocation 定位符名按表 5-5 所示完全对应以简化手动绑定。为实现实时驱动的效果，需要实时地更新 ARKit 检测到的人脸表情因子到模型变形器。

运行本示例，AR 应用启动后会自动开启前置摄像头，当检测到人脸时就会在该人脸位置挂载虚拟头像，当人脸表情发生变化时，虚拟头像模型对应表情也会发生变化，BlendShapes 效果如图 5-10 所示。

图 5-10　BlendShapes 效果图

注意

由于目前 USDZ 及 Reality 模型格式并不支持变形器(morpher target，blendShapes)，RealityKit 也不支持直接使用 Morpher.SetWeight()进行网格变形，所以本节中我们使用了 SceneKit，可以看到，当模型变形器命名与表 5-5 中命名完全一致时，使用 BlendShapes 比较简单。5.4 节将演示使用 RealityKit 获取表情因子驱动模型。

5.4 同时开启前后摄像头

iOS设备配备了前后两个摄像头，在运行AR应用时，需要选择使用哪个摄像头作为图像输入。最常见的AR体验使用设备后置摄像头进行世界跟踪、虚实融合，通常使用ARWorldTrackingConfiguration配置跟踪使用者的真实环境。除了进行虚实融合，我们通常还利用后置摄像头采集的图像信息评估真实世界中的光照情况、对真实环境中的2D图像或者3D物体进行检测等。

对具备前置深度相机(TrueDepth Camera)或者A12及以上处理器的设备，使用ARFaceTrackingConfiguration配置可以实时进行人脸检测跟踪，实现人脸姿态和表情的捕捉。

拥有前置深度相机或A12及以上处理器硬件的iPhone/iPad，在运行iOS 13及以上系统时，还可以同时开启设备前后摄像头，即同时进行人脸检测和世界跟踪。这是一项非常有意义且实用的功能，意味着使用者可以使用表情控制场景中的虚拟物体，实现除手势与语音之外的另一种交互方式。

在RealityKit中，同时开启前后摄像头需要使用ARFaceTrackingConfiguration配置或者ARWorldTrackingConfiguration配置之一。使用ARFaceTrackingConfiguration配置时将其supportsWorldTracking属性设置为true，使用ARWorldTrackingConfiguration配置时将其userFaceTrackingEnabled属性设置为true都可以在支持人脸检测的设备上同时开启前后摄像头。

同时开启前后摄像头后，RealityKit会使用后置摄像头跟踪现实世界，同时也会通过前置摄像头实时检测人脸信息，包括人脸表情信息。

需要注意的是，并不是所有设备都支持同时开启前后摄像头，只有符合前文所描述的设备才支持该功能，因此，在使用之前也应当对该功能的支持情况进行检查。在不支持同时开启前后摄像头的设备上应当执行另外的策略，如提示用户进行只使用单个摄像头的操作。

在下面的演示中，我们会利用后置摄像头的平面检测功能，在检测到的水平平面上放置机器头像模型，然后利用从前置摄像头中捕获的人脸表情信息驱动头像模型。核心代码如代码清单5-7所示。

代码清单 5-7

```
var robotHead = RobotHead()
struct ARViewContainer: UIViewRepresentable {
    func makeUIView(context: Context) -> ARView {
        let arView = ARView(frame: .zero)
        if ARWorldTrackingConfiguration.supportsUserFaceTracking {
            let config = ARWorldTrackingConfiguration()
            config.userFaceTrackingEnabled = true
            config.planeDetection = .horizontal
            config.worldAlignment = .gravity
            arView.automaticallyConfigureSession = false
            arView.session.run(config,options: [])
            arView.session.delegate = arView
            arView.createRobotHead()
        }
        return arView
    }
    func updateUIView(_ uiView: ARView, context: Context) {
    }
```

```
}

extension ARView : ARSessionDelegate{
    func createRobotHead(){
        let planeAnchor = AnchorEntity(plane:.horizontal)
        planeAnchor.addChild(robotHead)
        self.scene.addAnchor(planeAnchor)
    }

    public func session(_ session: ARSession, didUpdate anchors: [ARAnchor]) {
        for anchor in anchors {
            guard let anchor = anchor as? ARFaceAnchor else { continue }
            robotHead.update(with: anchor)
        }
    }
}
```

在代码清单 5-7 中，我们首先对设备支持情况进行检查，在确保设备支持同时开启前后摄像头功能时使用 ARWorldTrackingConfiguration 配置并运行 AR 进程，然后在检测到平面时将机器头像模型放置于平面上，最后利用 session(didUpdate frame:)代理方法使用实时捕获到的人脸表情数据更新机器头像模型，从而达到了使用人脸表情驱动场景中模型的目的。需要注意的是代码中 userFaceTrackingEnabled 必须设置为 true，并且开启平面检测功能，另外，为更好地组织代码，我们将与模型及表情驱动相关的代码放到了 RobotHead 类中。

RobotHead 类用于管理机器头像模型加载及使用表情数据驱动模型的工作，关键代码如代码清单 5-8 所示。

代码清单 5-8

```
func update(with faceAnchor: ARFaceAnchor) {
    let blendShapes = faceAnchor.blendShapes
    guard let eyeBlinkLeft = blendShapes[.eyeBlinkLeft] as? Float,
        let eyeBlinkRight = blendShapes[.eyeBlinkRight] as? Float,
        let eyeBrowLeft = blendShapes[.browOuterUpLeft] as? Float,
        let eyeBrowRight = blendShapes[.browOuterUpRight] as? Float,
        let jawOpen = blendShapes[.jawOpen] as? Float,
        let upperLip = blendShapes[.mouthUpperUpLeft] as? Float,
        let tongueOut = blendShapes[.tongueOut] as? Float
        else { return }

    eyebrowLeftEntity.position.y = originalEyebrowY + 0.03 * eyeBrowLeft
    eyebrowRightEntity.position.y = originalEyebrowY + 0.03 * eyeBrowRight
    tongueEntity.position.z = 0.1 * tongueOut
    jawEntity.position.y = originalJawY - jawHeight * jawOpen
    upperLipEntity.position.y = originalUpperLipY + 0.05 * upperLip
    eyeLeftEntity.scale.z = 1 - eyeBlinkLeft
    eyeRightEntity.scale.z = 1 - eyeBlinkRight

```

```
20.     let cameraTransform = self.parent?.transformMatrix(relativeTo: nil)
21.     let faceTransformFromCamera = simd_mul(simd_inverse(cameraTransform!), faceAnchor.transform)
22.     let rotationEulers = faceTransformFromCamera.eulerAngles
23.     let mirroredRotation = Transform(pitch: rotationEulers.x, yaw: -rotationEulers.y + .pi, roll:
    rotationEulers.z)
24.     self.orientation = mirroredRotation.rotation
25. }
```

在代码清单 5-8 中，我们首先从 ARFaceAnchor 中获取 BlendShapes 表情运动因子集合，并从中取出感兴趣的表情运动因子，然后利用这些表情因子对机器头像模型中的子实体对象相关属性进行调整，最后还处理了人脸与模型旋转关系的对应问题。

在支持同时开启前置与后置摄像头的设备上编译运行，当移动设备在检测到的水平平面时放置好机器头像模型，将前置摄像头对准人脸，可以使用人脸表情驱动机器头像模型，当人体头部旋转时，机器头像模型也会相应地进行旋转，实现效果如图 5-11 所示。

图 5-11　使用前置摄像头检测到的人脸姿态控制后置摄像头中的机器头像模型效果

本节演示的是一个简单的案例，但完整实现了利用前置摄像头采集的人脸表情信息控制后置摄像头中机器头像模型的功能，并且我们也看到，在使用前置摄像头时，后置摄像头可以同时进行世界跟踪。

由于 RealityKit 目前没有控制网格变形的函数，要实现利用人脸表情控制驱动模型的功能，开发者只能手动进行人脸表情与模型状态变化的绑定，人工计算模型中各子对象的位置与方向，这是一个比较麻烦且容易出错的过程。

经过测试发现，ARKit 对人脸表情的捕捉还是比较准确的，在使用配备深度相机的设备时，捕捉精度比较高，可以应付一般应用需求。

在本章中，我们对 ARKit 人脸检测相关知识进行了学习，对人脸网格、人脸贴纸、挂载虚拟物体、BlendShapes 进行了演示，但人脸检测的应用远远比演示的要多，例如可以利用人脸特定表情解锁、使用人眼凝视点判断使用者关注点实现人眼凝视交互等。通过学习，我们也发现，目前 RealityKit 在一些人脸功能支持上还有不足，部分功能需要借助 SceneKit 或者其他技术才能实现。