第 3 章

ARKit 功能特性与开发基础

ARKit 是一个高级 AR 应用开发引擎，得益于苹果公司强大的软硬件整合能力和应用生态，ARKit 一经推出即在科技圈引发极大关注，一方面是苹果公司在科技界的巨大影响力；另一方面是 ARKit 在移动端实现的堪称惊艳的 AR 效果。ARKit 的面世，直接将 AR 技术带到了亿万用户眼前，更新了人们对 AR 的印象。在集成到 AR Foundation 框架后，提供了简洁统一的使用界面，这使得利用其开发 AR 应用变得非常高效。ARKit 运动跟踪稳定性好，并且支持多传感器融合（如深度传感器、双目相机、LiDAR），性能消耗低，有利于营造沉浸性更好的 AR 体验。本章主要阐述 ARKit 本身的技术能力和功能特性。

3.1 ARKit 概述及主要功能

2017 年 6 月，苹果公司发布 ARKit SDK（Software Development Kit，软件开发工具包），它能够帮助用户在移动端快速实现 AR 功能。ARKit 的发布推动了 AR 概念的普及，但与其他苹果生态 SDK 一样，ARKit 只能用于苹果公司自家的移动终端（包括 iPhone、iPad、iPod 及未来的 iGlass）。苹果公司官方对 ARKit 的描述为通过整合设备摄像头图像信息与设备运动传感器（包括 LiDAR）信息，在应用中提供 AR 体验的开发套件。对开发人员而言，更通俗的理解即 ARKit 是一种用于开发 AR 应用的 SDK。

从本质上讲，AR 是将 2D 或者 3D 元素（文字、图片、模型、音视频等）放置于设备摄像头所采集的图像中，营造一种虚拟元素真实存在于现实世界中的假象。ARKit 整合了摄像头图像采集、图像视觉处理、设备运动跟踪、场景渲染等技术，提供了简单易用的 API（Application Programming Interface，应用程序接口）以方便开发人员开发 AR 应用，开发人员不需要再关注底层的技术实现细节，从而大大降低了 AR 应用开发难度。

ARKit 通过移动设备（包括手机与平板电脑）单目摄像头采集的图像信息（包括 LiDAR 采集的信息），实现了平面检测识别、场景几何识别、环境光估计、环境光反射、2D 图像识别、3D 物体识别、人脸检测、人体动作捕捉等高级功能，在这些基础上就能够创建虚实融合的场景。如将一个虚拟的数字机器人放置在桌面上，虚拟机器人将拥有与现实世界真实物体一样的外观、物理效果、光影效果，并能依据现实世界中的照明条件动态地调整自身的光照

信息以便更好地融合到环境中，如图 3-1 所示。

图 3-1　在桌面上放置虚拟机器人的 AR 效果

得益于苹果公司强大的软硬件整合能力及其独特的生态，ARKit 得以充分挖掘 CPU/GPU 的潜力，在跟踪精度、误差消除、场景渲染方面做到了同时期的最好水平，表现在用户体验上就是 AR 跟踪稳定性好、渲染真实度高、人机交互自然。

ARKit 出现后，数以亿计的 iPhone、iPad 设备一夜之间拥有了最前沿的 AR 功能，苹果公司 iOS 平台也一举成为最大的移动 AR 平台。苹果公司还与 Unity、Unreal 合作，进一步扩大 AR 开发平台，拓宽 iOS AR 应用的开发途径，奠定了其在移动 AR 领域的领导者地位。

3.1.1　ARKit 功能

技术层面上，ARKit 通过整合 AVFoundation、CoreMotion、CoreML 这 3 个框架，在此基础上融合扩展而成，如图 3-2 所示，其中 AVFoundation 是处理基于时间的多媒体数据框架，CoreMotion 是处理加速度计、陀螺仪、LiDAR 等传感数据信息框架，CoreML 则是机器学习框架。ARKit 融合了来自 AVFoundation 的视频图像信息与来自 CoreMotion 的设备运动传感器数据，再借助于 CoreML 计算机视觉图像处理与机器学习技术，为开发者提供稳定的三维数字环境。

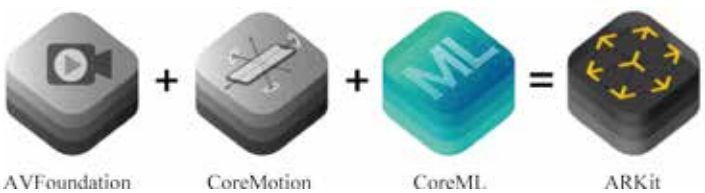

图 3-2　ARKit 通过融合扩展多个框架

经过多次迭代升级，ARKit 技术日渐完善，功能也日益拓展。目前，ARKit 主要提供如表 3-1 所示的功能。

表 3-1　ARKit 主要功能

功　能	描　述
特征点检测 （Feature Point）	检测并跟踪从设备摄像头采集的图像中的特征点信息，并利用这些特征点构建对现实世界的理解
平面检测 （Plane Detect）	检测并跟踪现实世界中的平整表面，ARKit 支持水平平面与垂直平面检测
图像检测识别跟踪 （Image Tracking）	检测识别并跟踪预扫描的 2D 图像，ARKit 最大支持同时检测 100 张图像
3D 物体检测跟踪 （Object Tracking）	检测识别并跟踪预扫描的 3D 物体
光照估计 （Light Estimation）	利用从摄像头图像采集的光照信息估计环境中的光照，并依此调整虚拟物体的光照效果
环境光反射 （Environment Probes & Environment Reflection）	利用从摄像头图像中采集的信息生成环境光探头（Environment Probe），并利用这些图像信息反射真实环境中的物体，以达到更真实的渲染效果
世界地图 （World Map）	支持保存与加载真实场景的空间映射数据，以便在不同设备之间共享体验
人脸检测跟踪 （Face Tracking）	检测跟踪摄像头图像中的人脸，ARKit 支持同时跟踪 3 张人脸。ARKit 还支持眼动跟踪，并支持人脸 BlendShape 功能，可以驱动虚拟人像模型
眼动跟踪 （Eye Tracking）	检测跟踪摄像头图像中的人眼运动，支持检测眼球凝视方向、双眼注视点
射线检测 （Ray Casting & Hit Testing）	从设备屏幕发射射线检测虚拟对象或者平面
人体动作捕捉 （Motion Capture）	检测跟踪摄像头图像中的人形，捕获人形动作，并用人形动作驱动虚拟模型，ARKit 支持 2D 和 3D 人形动作捕捉跟踪
人形遮挡 （People Occlusion）	分离摄像头图像中的人形区域，并能估计人形区域深度，以实现与虚拟物体的深度对比，从而实现正确的遮挡关系
多人协作 （Collaborative Session）	多设备间实时通信以共享 AR 体验
同时开启前后摄像头 （Simultaneous Front and Rear Camera）	允许同时开启设备前后摄像头，并可利用前置摄像头采集到的人脸检测数据驱动后置摄像头场景中的模型
3D 音效 （3D Audio）	模拟真实空间中的 3D 声音传播效果
景深 （Scene Depth）	模拟摄像机采集图像信息时的景深效果，实现焦点转移
相机噪声 （Camera Noise）	模拟相机采集图像时出现的不规则噪声

续表

功　能	描　述
运动模糊 （Motion Blur）	模拟摄像机在拍摄运动物体时出现的模糊拖尾现象
自定义渲染 （Custom Display）	支持对所有 ARKit 特性的自定义渲染
场景几何 （Scene Geometry）	使用 LiDAR 实时捕获场景深度信息并转换为场景几何网格
场景语义 （Scene Understanding）	支持检测到的平面和场景表面几何网格的语义分类
场景深度 （Depth API）	使用 LiDAR 实时捕获场景深度信息
视频纹理 （Video Texture）	采用视频图像作为纹理，可以实现视频播放、动态纹理效果
地理位置锚点 （Geographical Location Anchor）	利用 GPS 与地图在特定的地理位置上放置虚拟物体

除了表 3-1 中我们所能看到的 ARKit 提供的能力，ARKit 还提供了我们看不见但对渲染虚拟物体和营造虚实融合异常重要的尺寸度量系统。ARKit 的尺寸度量系统非常稳定、精准，ARKit 尺度空间中的 1 单位等于真实世界中的 1m，因此，我们能在 ARKit 虚拟空间中营造与真实世界体验一致的物体尺寸，并能正确地依照与现实空间中近大远小视觉特性同样的规律渲染虚拟物体尺寸，实现虚实场景的几何一致性。

需要注意的是，ARKit 并不包含图形渲染 API，即 ARKit 没有图形渲染能力，它只提供设备的跟踪和真实物体表面检测功能。对虚拟物体的渲染由第三方框架提供，如 RealityKit、SceneKit、SpriteKit、Metal 等，这提高了灵活性，同时降低了 ARKit 的复杂度，减小了包体大小。

3.1.2　ARKit 三大基础能力

ARKit 整合了 SLAM、计算机视觉、机器学习、传感器融合、表面估计、光学校准、特征匹配、非线性优化等大量的底层技术，提供给开发者简捷易用的程序界面。ARKit 提供的能力总体可以分为 3 部分：运动跟踪、场景理解和渲染，如图 3-3 所示，在这三大基础能力之上，构建了形形色色的附加功能。

图 3-3　ARKit 三大基础核心能力

1. 运动跟踪

实时跟踪用户设备在现实世界中的运动是 ARKit 的核心功能之一，利用该功能可以实时获取用户设备姿态信息。在运动跟踪精度与消除误差积累方面，ARKit 控制得非常好，表现在使用层面就是加载的虚拟元素不会出现漂移、抖动、闪烁现象。ARKit 的运动跟踪整合了 VIO 和 IMU，即图像视觉跟踪与运动传感器跟踪，提供 6DoF 跟踪能力，不仅能跟踪设备位移，还能跟踪设备旋转。

更重要的是，ARKit 运动跟踪没有任何前置要求，不需要对环境的先验知识，也没有额外的传感器要求，仅凭现有的移动设备硬件就能满足 ARKit 运动跟踪的所有要求。

2. 场景理解

场景理解建立在运动跟踪、计算机视觉、机器学习等技术之上。场景理解提供了关于设备周边现实环境的属性相关信息，如平面检测功能，提供了对现实环境中物体表面（如地面、桌面等）检测平面的能力，如图 3-4 所示。从技术上讲，ARKit 通过检测特征点和平面来不断改进它对现实世界环境的理解。ARKit 可以检测位于常见的水平或垂直表面（例如桌子或墙）上的成簇特征点，并允许将这些表面用作应用程序的工作平面，ARKit 也可以确定每个平面的边界，并将该信息提供给应用，使用此信息可以将虚拟物体放置于平坦的表面上而不超出平面的边界。场景理解是一个渐进的过程，随着设备探索的环境不断拓展而不断加深，并可随着探索的进展不断修正误差。

图 3-4　ARKit 对平坦表面的检测识别

ARKit 通过 VIO 检测特征点来识别平面，因此它无法正确检测像白墙一样没有纹理的平坦表面。当加入 LiDAR 传感器后，ARKit 对环境的感知能力得到大幅提高，不仅可以检测平坦表面，还可以检测非平坦有起伏的表面，由于 LiDAR 的特性，其对弱纹理、光照不敏感，可以构建现实环境的高精度几何网格。

场景理解还提供了射线检测功能，利用该功能可以与场景中的虚拟对象、检测到的平面、特征点交互，如将虚拟元素放置到指定位置、旋转移动虚拟物体等。场景理解还对现实环境中的光照信息进行评估，并利用这些光照估计信息修正场景中的虚拟元素光照。除此之外，场景理解还实现了反射现实物理环境功能以提供更具沉浸性的虚实融合体验。

3. 渲染

严格意义上讲，ARKit 并不包含渲染功能，AR 的渲染由第三方框架提供，但除提供场景理解能力之外，ARKit 还提供连续的摄像头图像流，这些图像流可以方便地融合进任何第三方渲染框架，如 RealityKit、SceneKit、SpriteKit、Metal，或者自定义的渲染器。本书主要讲述利用 Unity 开发引擎进行 AR 场景搭建和渲染，因此无须关注底层的渲染技术。

运动跟踪、场景理解、渲染紧密协作，形成了稳定、健壮、智能的 ARKit，在这三大基

础能力之上，ARKit 还提供了诸如 2D 图像识别跟踪、3D 物体识别跟踪、物理仿真、基于 GPS 地理位置的 AR 等实用功能。

在苹果公司的强力推动下，ARKit 正处于快速发展中，性能更强的硬件和新算法的加入，提供了更快的检测速度（如平面检测、人脸检测、3D 物体检测等）和更多更强的功能特性（如人形遮挡、人体动作捕捉、人脸 BlendShapes、场景几何网格等）。ARKit 适用的硬件范围也在拓展，可以预见，ARKit 适用的硬件一定会拓展到苹果公司的 AR 眼镜产品。

3.1.3 ARKit 的不足

ARKit 提供了稳定的运动跟踪功能，也提供了高精度的环境感知功能，但受限于视觉 SLAM 技术的技术水平和能力，ARKit 的运动跟踪在以下情况时会失效。

1. 在运动中做运动跟踪

如果用户在火车或者电梯中使用 ARKit，这时 IMU 传感器获取的数据不仅包括用户的移动数据（实际是加速度），也包括火车或者电梯的移动数据（实际是加速度），则将导致 SLAM 数据融合问题，从而引起漂移甚至完全失败。

2. 跟踪动态的环境

如果用户设备对着波光粼粼的湖面，这时从设备摄像头采集的图像信息是不稳定的，会引发图像特征点提取匹配问题进而导致跟踪失败。

3. 热漂移

相机感光器件与 IMU 传感器都是对温度敏感的元器件，其校准通常只会针对某个或者几个特定温度，但在用户设备的使用过程中，随着时间的推移设备会发热，发热会影响摄像头采集图像的颜色 / 强度信息和 IMU 传感器测量的加速度信息的准确性，从而导致误差，表现出来就是跟踪物体的漂移。

4. 昏暗环境

基于 VIO 的运动跟踪效果与环境光照条件有很大关系，昏暗环境采集的图像信息对比度低，这对提取图像特征点信息非常不利，进而影响到跟踪的准确性，这也会导致基于 VIO 的运动跟踪失败。

除运动跟踪问题外，由于移动设备资源限制或其他问题，ARKit 还存在以下不足。

1. 环境理解需要时间

ARKit 虽然对现实场景表面特征点提取与平面检测进行了非常好的优化，但还是需要一个相对比较长的时间，在这个过程中，ARKit 需要收集环境信息构建对现实世界的理解。这是一个容易让不熟悉 AR 的使用者产生困惑的阶段，因此，AR 应用中应当设计良好的用户引

导，同时帮助 ARKit 更快地完成初始化。在配备 LiDAR 传感器的高端设备上，由于 LiDAR 可以实时获取场景表面几何网格信息，从而大大加速了这个过程，也无须用户移动设备获取视差就能完成对周边环境的理解。

2. 运动处理有滞后

当用户设备移动过快时会导致摄像头采集的图像模糊，从而影响 ARKit 对环境特征点的提取，进而造成运动跟踪误差，表现为跟踪不稳定或者虚拟物体漂移。

3. 弱纹理表面检测问题

ARKit 使用的 VIO 技术很难在光滑、无纹理、反光的场景表面提取所需的特征值，因而无法构建对环境的理解和检测识别平面，如很难检测跟踪光滑大理石地面和白墙。当然，该问题也会因为有 LiDAR 传感器的加入而得到良好的改善。

4. 鬼影现象

虽然 ARKit 在机器学习的辅助下对平面边界预测作了很多努力，但由于现实世界环境的复杂性，检测到的平面边界仍然不够准确，因此，添加的虚拟物体可能会出现穿越墙壁的现象，所以对开发者而言，应当鼓励使用者在开阔的空间或场景中使用 AR 应用程序。

提示

本节讨论的 ARKit 不足为不使用 LiDAR 传感器时存在的先天技术劣势，在配备有 LiDAR 传感器的设备上，由于 LiDAR 传感器并不受到弱纹理、灯光等因素的影响，因此 ARKit 能实时精准高效地重建场景几何网格，可以弥补由于 VIO 原因给 ARKit 带来的不足，但 LiDAR 只对场景重建有帮助，并不能解决如昏暗环境跟踪失效、热漂移、运动中跟踪等问题。对开发人员而言，了解 ARKit 的优劣势才能更好地扬长避短，在适当的时机通过适当的引导最佳化用户体验。

3.2　运动跟踪原理

第 1 章对 AR 技术原理进行过简要介绍，ARKit 运动跟踪所采用的技术路线与其他移动端 AR SDK 相同，也采用 VIO 与 IMU 结合的方式进行 SLAM 定位跟踪。本节将更加深入地讲解 ARKit 运动跟踪原理，从而加深对 ARKit 在运动跟踪方面优劣势的理解，并在开发中尽量避免劣势或者采取更加友好的方式扬长避短。

3.2.1　ARKit 坐标系

实现虚实融合最基本的要求是实时跟踪用户（设备）的运动，始终保持虚拟相机与设备

摄像头的对齐，并依据运动跟踪结果实时更新虚拟元素的姿态，这样才能在现实世界与虚拟世界之间建立稳定精准的联系。运动跟踪的精度与质量直接影响 AR 的整体效果，任何延时、误差都会造成虚拟元素抖动或者漂移，从而破坏 AR 的真实感和沉浸性。

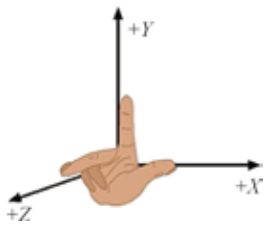

图 3-5　ARKit 采用右手坐标系

在进一步讲解运动跟踪之前，先了解一下 ARKit 空间坐标系，在不了解 AR 坐标系的情况下，阅读或者实现代码可能会感到困惑（如在 AR 空间中放置的虚拟物体会在 *Y* 轴上有 180° 偏转）。ARKit 采用右手坐标系（包括世界坐标系、相机坐标系、投影坐标系，这里的相机指渲染虚拟元素的相机），而 Unity 使用左手坐标系。右手坐标系 *Y* 轴正向朝上，*Z* 轴正向指向观察者自己，*X* 轴正向指向观察者右侧，如图 3-5 所示。

当用户在实际空间中移动时，ARKit 坐标系上的距离增减遵循表 3-2 所示的规律。

表 3-2　ARKit 采用的坐标系与设备移动关系

移 动 方 向	描　　述
向右移动	*X* 增加
向左移动	*X* 减少
向上移动	*Y* 增加
向下移动	*Y* 减少
向前移动	*Z* 减少
向后移动	*Z* 增加

3.2.2　ARKit 运动跟踪分类

ARKit 运动跟踪支持 3DoF 和 6DoF 两种模式，3DoF 只跟踪设备旋转，因此是一种受限的运动跟踪方式，通常，我们不会使用这种运动跟踪方式，但在一些特定场合或者 6DoF 跟踪失效的情况下，也有可能会用到。

DoF 概念与刚体在空间中的运动相关，可以解释为“刚体运动的不同基本方式”。在客观世界或者虚拟世界中，都采用三维坐标系来精确定位一个物体的位置。如一个有尺寸的刚体放置在坐标系的原点，那么这个物体的运动整体上可以分为平移与旋转两类（刚体不考虑缩放），同时，平移又可以分为 3 个度：前后（沿 *Z* 轴移动）、左右（沿 *X* 轴移动）、上下（沿 *Y* 轴移动）；旋转也可以分 3 个度：俯仰（围绕 *X* 轴旋转）、偏航（围绕 *Y* 轴旋转）、翻滚（围绕 *Z* 轴旋转）。通过分析计算，刚体的任何运动方式均可由这 6 个基本运动方式来表达，即 6DoF 的刚体可以完成所有的运动形式，具有 6DoF 的刚体物体在空间中的运动是不受限的。

具有 6DoF 的刚体可以到达三维坐标系的任何位置并且可以朝向任何方向。平移相对来讲比较好理解，即刚体沿 *X*、*Y*、*Z* 3 个轴之一运动，旋转其实也是以 *X*、*Y*、*Z* 3 个轴之一为旋转轴进行旋转。在计算机图形学中，常用一些术语来表示特定的运动，这些术语如表 3-3 所示。

表 3-3　刚体运动术语及其含义

术　语	描　述
Down	向下
Up	向上
Strafe	左右
Walk	前进后退
Pitch	围绕 *X* 轴旋转，即上下打量，也叫俯仰角
Rotate	围绕 *Y* 轴旋转，即左右打量，也叫偏航角（Yaw）
Roll	围绕 *Z* 轴旋转，即翻滚，也叫翻滚角

在 AR 空间中描述物体的位置和方向时经常使用姿态（Pose）这个术语，姿态的数学表达为矩阵，既可以用矩阵来表示物体平移，也可以用矩阵来表示物体旋转。为了更好地平滑及优化内存使用，通常还会使用四元数来表达旋转，四元数允许我们以简单的形式定义 3D 旋转的所有方面。

1. 方向跟踪

在 AR Foundation 中，可以通过将场景中 AR Session 对象上的 AR Session 组件 Tracking Mode 属性设置为 Rotation Only 值、XR Origin → Camera Offset → Main Camera 对象上的 Tracked Pose Driver 组件 Tracking Type 属性设置为 Rotation Only 值使用 3DoF 方向跟踪[①]，即只跟踪设备在 *X*、*Y*、*Z* 轴上的旋转运动，如图 3-6 所示，表现出来的效果类似于站立在某个点上下左右观察周围环境。方向跟踪只跟踪方向变化而不跟踪设备位置变化，由于缺少位置信息，无法从后面去观察放置在地面上的桌子，因此这是一种受限的运动跟踪方式。在 AR 中采用这种跟踪方式时虚拟元素会一直飘浮在摄像头图像之上，即不能固定于现实世界中。

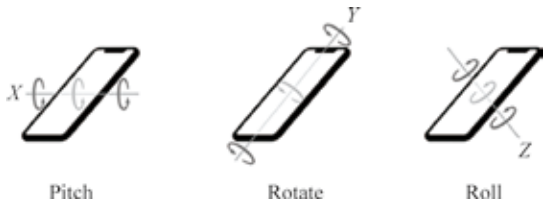

图 3-6　ARKit 中 3DoF 跟踪示意图

采用 3DoF 进行运动跟踪时，无法使用平面检测功能，也无法使用射线检测功能。

2. 世界跟踪

在 AR Foundation 中，可以通过将场景中 AR Session 对象上的 AR Session 组件 Tracking

① 这两个组件实际上分别对应老的 Input Manager 输入管理器和新的 Input System 输入管理器，即 AR Session 组件会在老的 Input Manager 起作用时有效，而 Tracked Pose Driver 组件会在新的 Input System 起作用时有效。

Mode 属性设置为 Position And Rotation 值、XR Origin → Camera Offset → Main Camera 对象上的 Tracked Pose Driver 组件 Tracking Type 属性设置为 Position And Rotation 值使用 6DoF 跟踪，这是默认跟踪方式，使用这种方式，既能跟踪设备在 *X*、*Y*、*Z* 轴上的旋转运动，也能跟踪设备在 *X*、*Y*、*Z* 轴上的平移运动，能实现对设备姿态的完全跟踪，如图 3-7 所示。

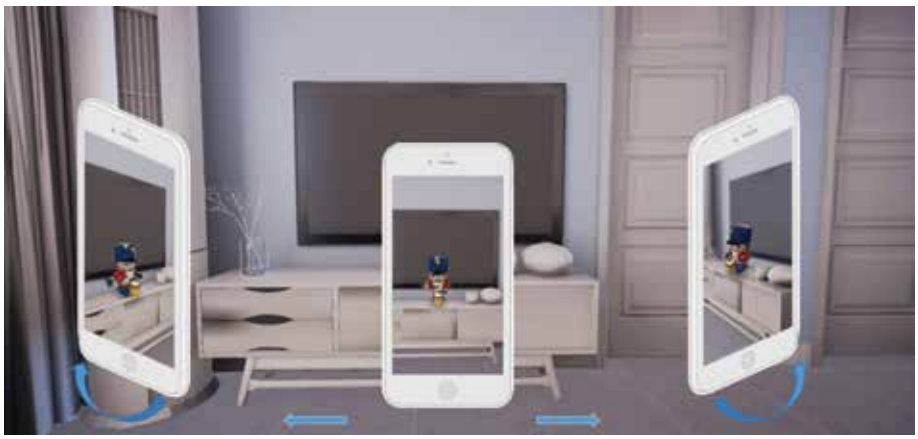

图 3-7　ARKit 中 6DoF 跟踪示意图

6DoF 的运动跟踪方式（世界跟踪）可以营造完全真实的 AR 体验，通过世界跟踪，能从不同距离、方向、角度观察虚拟物体，就好像虚拟物体真正存在于现实世界中一样。在 ARKit 中，通常通过世界跟踪方式创建 AR 应用，使用世界跟踪时，支持平面检测、射线检测，还支持检测识别摄像头采集图像中的 2D 图像等。

3.2.3　ARKit 运动跟踪

ARKit 通过 VIO + IMU 方式进行运动跟踪，图像数据来自设备摄像头，IMU 数据来自运动传感器，包括加速度计和陀螺仪，它们分别用于测量设备的实时加速度与角速度。运动传感器非常灵敏，每秒可进行 1000 次以上的数据检测，能在短时间跨度内提供非常及时准确的运动信息；但运动传感器也存在测量误差，由于检测速度快，这种误差累积效应就会非常明显（微小的误差以每秒 1000 次的速度累积会迅速变大），因此，在较长的时间跨度后，跟踪就会变得完全失效。

ARKit 为了消除 IMU 存在的误差累积漂移，采用 VIO 的方式进行跟踪校准，VIO 基于计算机视觉计算，该技术可以提供非常高的计算精度，但付出的代价是计算资源与计算时间。ARKit 为了提高 VIO 跟踪精度采用了机器学习方法，因此，VIO 处理速度相比于 IMU 要慢得多，另外，计算机视觉处理对图像质量要求非常高，对设备运动速度非常敏感，因为快速的摄像机运动会导致采集的图像模糊。

ARKit 充分吸收利用了 VIO 和 IMU 各自的优势，利用 IMU 的高更新率和高精度进行较短时间跨度的跟踪，利用 VIO 对较长时间跨度 IMU 跟踪进行补偿，融合跟踪数据向上提供运

动跟踪服务。IMU 信息来自运动传感器的读数，精度取决于传感器本身。VIO 信息来自计算机视觉处理结果，因此精度受到较多因素的影响，下面主要讨论 VIO，VIO 进行空间计算的原理如图 3-8 所示。

图 3-8　VIO 进行空间计算原理图

在 AR 应用启动后，ARKit 会不间断地捕获从设备摄像头采集的图像信息，并从图像中提取视觉差异点（特征点），ARKit 会标记每个特征点（每个特征点都有 ID），并会持续地跟踪这些特征点。当设备从另一个角度观察同一空间时（设备移动了位置），特征点就会在两张图像中呈现视差，利用这些视差信息和设备姿态偏移量就可以构建三角测量，从而计算出特征点缺失的深度信息。换言之，可以通过从图像中提取的二维特征进行三维重建，进而实现跟踪用户设备的目的。

从 VIO 工作原理可以看到，如果从设备摄像头采集的图像不能提供足够的视差信息，则无法进行空间三角计算，因此无法解算出空间信息。若要在 AR 应用中使用 VIO，则设备必须移动一定的距离（X、Y、Z 方向均可），无法仅仅通过旋转达到目的。

在通过空间三角计算后，特征点的位置信息被解算出来，这些位置信息会存储到对应特征点上。随着用户在现实世界中探索的进行，一些不稳定的特征点被剔除，一些新的特征点会加入，并逐渐形成稳定的特征点集合，这个特征点集合称为点云，点云坐标原点为 ARKit 初始化时的设备位置，点云地图就是现实世界在 ARKit 中的数字表达。

VIO 跟踪流程图如图 3-9 所示。从流程图可以看到，为了优化性能，计算机视觉计算并不是每帧都执行。VIO 跟踪主要用于校正补偿 IMU 在时间跨度较长时存在的误差累积，每帧执行视觉计算不仅会消耗大量计算资源，而且没有必要。

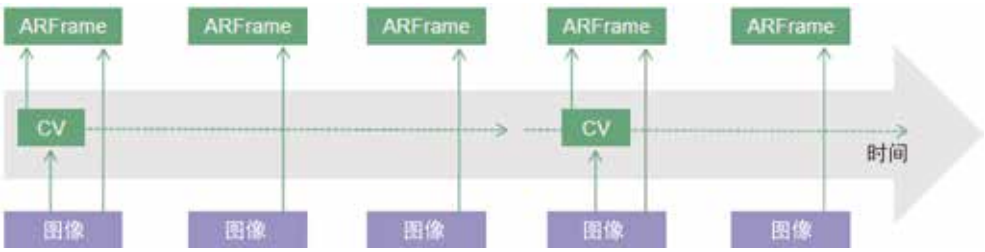

图 3-9　VIO 跟踪流程示意图

VIO 也存在误差，这种误差随着时间的积累也会变得很明显，表现出来就是放置在现实空间中的虚拟元素会出现一定的漂移。为抑制这种漂移，ARKit 使用锚点的方式绑定虚拟元素与现实空间环境，同时 ARKit 也会实时地对设备摄像头采集的图像进行匹配计算，如果发现当前采集的图像与之前某个时间点采集的图像匹配（用户在同一位置以同一视角再次观察现实世界时），则 ARKit 会对特征点的信息进行修正，从而优化跟踪（回环检测）。

ARKit 融合了 VIO 与 IMU 跟踪各自的优势，提供了非常稳定的运动跟踪能力，也正是因为稳定的运动跟踪使利用 ARKit 制作的 AR 应用体验非常好。

3.2.4 ARKit 使用运动跟踪的注意事项

通过对 ARKit 运动跟踪原理的学习，我们现在可以很容易地理解前文所列 ARKit 的不足，因此，为了得到更好的跟踪质量，需要注意以下事项。

（1）运动跟踪依赖于不间断输入的图像数据流与传感器数据流，某一种方式短暂地受到干扰不会对跟踪造成太大的影响，如用手偶尔遮挡摄像头图像采集不会导致跟踪失效，但如果中断时间过长，则跟踪就会变得很困难。

（2）VIO 跟踪精度依赖于采集图像的质量，低质量的图像（如光照不足、纹理不丰富、模糊）会影响特征点的提取，进而影响跟踪质量。

（3）当 VIO 数据与 IMU 数据不一致时会导致跟踪问题，如视觉信息不变而运动传感器数据变化（如在运动的电梯里），或者视觉信息变化而运动传感器数据不变（如摄像头对准波光粼粼的湖面），这都会导致数据融合障碍，进而影响跟踪效果。

开发人员很容易理解以上内容，但这些信息，使用者在进行 AR 体验时可能并不清楚，因此，必须实时地给予引导和反馈，否则会使用户困惑。ARKit 为辅助开发人员了解 AR 运动跟踪状态提供了实时的状态监视，将运动跟踪状态分为受限、正常、不可用（limited、normal、notAvailable）3 种，分别指示运动跟踪状态质量不佳、正常、当前不可用 3 种情况，并在跟踪受限时给出原因（在整合进 AR Foundation 时，状态由 ARSessionState 枚举描述）。为了提升用户的使用体验，应当在跟踪受限或者不可用时给出明确的原因和操作建议，引导使用者提高运动跟踪的精度和稳定性。

3.3 设备可用性检查

从第 1 章可知，只有 iPhone 6s 及以上 iPhone、第 5 代以上 iPad、第 7 代以上 iPod 才支持 ARKit，并且还有 iOS 版本要求，因此，通常在使用 ARKit 之前需要进行一次设备支持性检查以确保 AR 应用能正常运行。在 AR Foundation 框架中，使用 ARSession 类静态属性检查设备支持性，典型代码如下：

```
// 第 3 章 /3-1
public class DeviceCheck
```

```
    {
        [SerializeField]
        private ARSession mSession;

        IEnumerator Check() {
            if ((ARSession.state == ARSessionState.None) ||
                (ARSession.state == ARSessionState.CheckingAvailability))
            {
                yield return ARSession.CheckAvailability();    // 检查设备支持性
            }

            if (ARSession.state == ARSessionState.Unsupported)
            {
                // 设备不支持，需要启用其他备用方案
            }
            else
            {
                // 设备可用，启动会话
                mSession.enabled = true;
            }
        }
    }
```

设备支持状态由 ARSessionState 枚举描述，其枚举值如表 3-4 所示。

表 3-4　ARSessionState 枚举值

枚　举　值	描　　述
None	状态未知
Unsupported	当前设备不支持 ARKit
CheckingAvailability	系统正在进行设备支持性检查
NeedsInstall	设备硬件支持但需要安装 SDK 工具包，通常用于 ARCore
Installing	设备正在安装 SDK 工具包，通常用于 ARCore
Ready	设备支持 ARKit 并且已做好使用准备
SessionInitialized	ARKit 会话正在初始化，还未建立运动跟踪
SessionTracking	ARKit 会话运动跟踪中

在 ARKit 运动跟踪启动后，如果运动跟踪状态发生变化，则 ARSession.state 值也会发生变化，可以通过订阅 ARSession.stateChanged 事件处理运动跟踪状态变化的情况。

3.4　AR 会话生命周期管理与跟踪质量

AR 会话整合运动传感器数据和计算机视觉处理数据跟踪用户设备姿态，为得到更好的跟踪质量，AR 会话需要持续的运动传感器数据和视觉计算数据输入。在启动 AR 应用后，

ARKit 需要时间来收集足够多的视觉特征点信息，在这个过程中，ARKit 是不可用的。在 AR 应用运行过程中，由于一些异常情况（如摄像头被覆盖），ARKit 的跟踪状态也会发生变化，可以在需要时进行必要的处理（如显示 UI 信息）。

1. AR 会话生命周期

AR 会话的基本生命周期如图 3-10 所示，在刚启动 AR 应用时，ARKit 还未收集到足够多的特征点和运动传感器数据信息，无法计算设备的姿态，这时的跟踪状态是不可用状态。在经过几帧之后，跟踪状态会变为设备初始化状态（SessionInitialized），这种状态表明设备姿态已可用但精度可能会有问题。

图 3-10　ARKit 跟踪开始后的状态变化

再经过一段时间后，跟踪状态会变为正常状态（SessionTracking），说明 ARKit 已准备好，所有的功能都已经可用。

2. 跟踪受限

在 AR 应用运行过程中，由于环境的变化或者其他异常情况，ARKit 的跟踪状态会发生变化，如图 3-11 所示。

图 3-11　ARKit 跟踪状态会受到设备及环境的影响

在 ARKit 跟踪状态受限时，基于环境映射的功能将不可用，如平面检测、射线检测、2D 图像检测等。在 AR 应用运行过程中，由于用户环境变化或者其他异常情况，ARKit 可能在任何时间进入跟踪受限状态，如当用户将摄像头对准一面白墙或者房间中的灯突然关闭，这时 ARKit 就会进入跟踪受限状态。

3. 中断恢复

在 AR 应用运行过程中，AR 会话也有可能会被迫中断，如在使用 AR 应用的过程中突然来电话，这时 AR 应用将被切换到后台。当 AR 会话被中断后，虚拟元素与现实世界将失去关联。在 ARKit 中断结束后会自动尝试进行重定位（Relocalization），如果重定位成功，则虚拟元素与现实世界的关联关系会恢复到中断前的状态，包括虚拟元素的姿态及虚拟元素与现实世界之间的相互关系；如果重定位失败，则虚拟元素与现实世界的原有关联关系被破坏。

在重定位过程中，AR 会话的运动跟踪状态保持为受限状态，重定位成功的前提条件是使

用者必须返回 AR 会话中断前的环境中，如果使用者已经离开，则重定位永远也不会成功（环境无法匹配）。整个过程如图 3-12 所示。

图 3-12　ARKit 跟踪中断及重定位时的状态变化

提示

重定位是一个容易让使用者困惑的操作，特别是对不熟悉 AR 应用、没有 AR 应用使用经验的使用者而言，重定位会让他们感到迷茫。所以在进行重定位时，应当通过 UI 或者其他视觉信息告知使用者，并引导使用者完成重定位操作。

在 AR 应用运行中，可以通过 ARSession.stateChanged 事件获取运动跟踪状态变化情况，在跟踪受限时提示用户原因，引导用户进行下步操作，典型代码如下：

```
// 第 3 章 /3-2
using UnityEngine;
using UnityEngine.XR.AR Foundation;
using UnityEngine.XR.ARSubsystems;

public class TrackingReason : MonoBehaviour
{
    private bool mSessionTracking;          // 运动跟踪状态标识

    // 注册事件
    void OnEnable()
    {
        ARSession.stateChanged += ARSessionOnstateChanged;
    }
    // 取消事件注册
    void OnDisable()
    {
        ARSession.stateChanged -= ARSessionOnstateChanged;
    }

    //AR 会话状态变更事件
    void ARSessionOnstateChanged(ARSessionStateChangedEventArgs obj)
    {
        mSessionTracking = obj.state == ARSessionState.SessionTracking ? true :
false;
        if (mSessionTracking)
            return;
```

```
            switch (ARSession.notTrackingReason)
            {
                case NotTrackingReason.Initializing:
                    Debug.Log("AR 正在初始化 ");
                    break;
                case NotTrackingReason.Relocalizing:
                    Debug.Log("AR 正在进行重定位 ");
                    break;
                case NotTrackingReason.ExcessiveMotion:
                    Debug.Log(" 设备移动太快 ");
                    break;
                case NotTrackingReason.InsufficientLight:
                    Debug.Log(" 环境昏暗，光照不足 ");
                    break;
                case NotTrackingReason.InsufficientFeatures:
                    Debug.Log(" 环境无特性 ");
                    break;
                case NotTrackingReason.Unsupported:
                    Debug.Log(" 设备不支持跟踪受限原因 ");
                    break;
                case NotTrackingReason.None:
                    Debug.Log(" 运动跟踪未开始 ");
                    break;
            }
        }
    }
```

在 AR 运动跟踪受限时，AR Foundation 会通过 NotTrackingReason 枚举值标识跟踪受限原因，NotTrackingReason 枚举各值如表 3-5 所示。

表 3-5　NotTrackingReason 枚举值

枚　举　值	描　　述
None	运动跟踪未开始，状态未知
Initializing	正在进行跟踪初始化
Relocalizing	正在进行重定位
InsufficientLight	光照不足，环境昏暗
InsufficientFeatures	环境中特征点不足
ExcessiveMotion	设备移动太快，造成图像模糊
Unsupported	Provider 不支持跟踪受限原因描述
CameraUnavailable	设备摄像头不可用

3.5　基于地理位置的 AR

基于地理位置的 AR（Geography Location Based AR，以下简称地理 AR）因其巨大的潜在价值而受到广泛关注，通过将 AR 虚拟物体放置在真实世界经纬坐标上，可以实现如真实物体一样的自然效果，并支持持久共享。例如，可以在高速路两侧设置 AR 电子标志、虚拟路障、道路信息，一方面可以大大降低使用实物的成本；另一方面可以极大地提高信息反应速度，提高道路使用效率。

地理 AR 是非常具有前景的应用形态，但目前也面临很多问题，最主要的问题是定位系统的精度，在室外开阔的空间中，GPS/ 北斗等导航系统定位精度还不能满足 AR 需求，而且也受到很多因素的影响，如高楼遮挡、多径效应、方位不确定性等，无法仅凭 GPS/ 北斗等导航系统确定虚拟物体（或者移动设备）的姿态。

ARKit 已经在这方面进行了探索尝试，通过综合 GPS、设备电子罗盘、环境特征等各方面信息确定设备姿态。ARKit 实现地理 AR 非常依赖 GPS 与环境特征信息，如果这两者中的一个存在问题，则整体表现出来的效果就会大打折扣。GPS 信号来自卫星，环境特征信息来自地图，这里的地图不是普通意义上的地图，更确切的表述为点云地图，是事先采集的环境点云信息。当这两者均满足要求时，ARKit 就可以融合解算出设备的姿态，也就可以正确地加载虚拟物体，实现虚拟物体姿态与真实世界经纬度的对齐。

3.5.1　技术基础

在 ARKit 中，为了更好地使用地理 AR，需要使用 ARGeoTrackingConfiguration 配置类和 ARGeoAnchor 锚点类，前者专用于处理与地理位置相关的 AR 事宜，后者用于将虚拟物体锚定在一个指定经纬度、海拔高度的位置上。

使用 ARGeoTrackingConfiguration 配置类启动的 AR 会话会综合 GPS、设备电子罗盘、环境特征点信息数据进行设备姿态解算，如果所有条件都符合，则会输出设备的姿态。使用该配置类运行的 AR 会话也可以进行诸如平面检测、2D 图像检测、3D 物体检测之类的功能。另外，与所有其他类型应用一样，需要使用一个锚点将虚拟物体锚定到特定的位置，在地理 AR 中，这个锚点就是 ARGeoAnchor。

ARKit 为地理 AR 应用提供了与其他应用基本一致的操作界面，由于是锚定地理空间中的虚拟物体，所以 ARGeoAnchor 不仅需要有 Transform 信息，还需要有地理经纬坐标信息，并且 ARGeoAnchor 自身的 X 轴需要与地理东向对齐，Z 轴需要与地理南向对齐，Y 轴垂直向上，如图 3-13 所示。与其他所有锚点一样，ARGeoAnchor 放置以后也不可以修改。

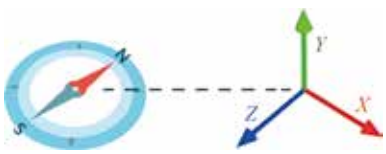

图 3-13　ARGeoAnchor 自身坐标轴与地理坐标轴对齐

进行地理位置定位不可控因素非常多，如 GPS 信号被遮挡、点云地图当前位置不可用等，

都有可能导致设备位姿解算失败，ARKit 引入了 ARGeoTrackingState 枚举，用于描述当前设备定位状态，AR 应用运行时将处于表 3-6 中的某种状态。

表 3-6　ARGeoTrackingState 枚举值

枚　举　值	描　　述
ARGeoTrackingStateInitializing	地理位置定位初始化
ARGeoTrackingStateLocalized	地理位置定位成功
ARGeoTrackingStateLocalizing	正在进行定位
ARGeoTrackingStateNotAvailable	状态不可用

在 AR 应用运行过程中，设备定位状态也会由于各种因素发生变化，如图 3-14 所示，只有当定位状态为 ARGeoTrackingStateLocalized 时才能有效地跟踪 ARGeoAnchor。

图 3-14　定位状态会在运行时不断地变化

当定位状态不可用时，ARKit 使用 ARGeoTrackingStateReason 枚举描述出现问题的原因，开发人员可以实时地获取这些值，引导用户进行下一步操作，该枚举所包含的值如表 3-7 所示。

表 3-7　ARGeoTrackingStateReason 枚举值

枚　举　值	描　　述
ARGeoTrackingStateReasonNone	没有检测到问题原因
ARGeoTrackingStateReasonNotAvailableAtLocation	当前位置无法进行地理定位
ARGeoTrackingStateReasonNeedLocationPermissions	GPS 使用未授权
ARGeoTrackingStateReasonDevicePointedTooLow	设备摄像头所拍摄角度太低，无法进行特征点匹配
ARGeoTrackingStateReasonWorldTrackingUnstable	跟踪不稳定，设备姿态不可靠
ARGeoTrackingStateReasonWaitingForLocation	设备正在等待 GPS 信号
ARGeoTrackingStateReasonGeoDataNotLoaded	正在下载点云地图，或者点云地图不可用
ARGeoTrackingStateReasonVisualLocalizationFailed	点云匹配失败

为营造更好的用户体验，即使在定位成功（定位状态为 ARGeoTrackingStateLocalized）时，ARKit 也会根据当前定位的准确度将状态划分为若干精度级，由 ARGeoTrackingStatus.Accuracy 枚举描述，具体如表 3-8 所示。

表 3-8　ARGeoTrackingStatus.Accuracy 枚举值

枚　举　值	描　　述
high	定位精度非常高
undetermined	定位精度不明确
low	定位精度较低
medium	定位精度中等

ARKit 对定位精度进行划分的目的是希望开发者能依据不同的精度制定不同的应对方案，定位精度越低，虚拟物体与定位点之间的误差就越大，表现出来就是虚拟物体偏移，如原来放置在公园大门前的虚拟物体会偏移到广场中间。针对不同的定位精度可以采用不同的策略，在定位精度较低时使虚拟物体不要过分依赖特定点，如飘浮在空中的热气球就比放置在门口的小木偶更适合，更不容易让使用者察觉到定位点的偏移。

提示

本节讨论的基于位置的 AR 相关技术均为原生 ARKit 类，目前在 AR Foundation 中，并没有完全实现相应的方法和枚举，因此需要开发者自行进行处理。

3.5.2　实践

由于 AR Foundation 当前并不直接支持地理 AR，因此需要通过 Object-C 原生代码进行桥接，在 ARKit 初始化前通过 ConfigurationChooser 类进行配置，引导 ARKit 使用 ARGeoTrackingConfiguration 配置进行初始化。同时，为了在原生代码与托管代码间传递 ARSession 对象，也需要对 XRSessionSubsystem 类 nativePtr 指针进行转换。

另外，使用地理 AR 需要 iOS 14 以上操作系统及 A12 以上处理器，而且只有苹果公司提供点云地图的区域（目前区域有限，仅北美地区若干城市和伦敦提供）才可以使用，由于需要使用 GPS，因此需要开启 GPS 权限和支持网络连接。

下面以一个简单示例进行使用说明，该示例中，在获取当前设备位置姿态定位信息后，通过单击屏幕，在该地理位置添加一个 ARGeoAnchor 锚点[①]。

首先通过 Object-C 原生代码实现互操作类及方法，新建一个代码脚本文件，命名为 GeoAnchorsNativeInterop.mm，编写代码如下：

```
// 第 3 章 /3-3
#import<ARKit/ARKit.h>

// 获取 ARGeoTrackingConfiguration 配置类
```

① 有了锚点后我们就可以在该锚点上挂载虚拟物体、进行对象渲染展示等操作。

```
Class ARGeoTrackingConfiguration_class() {
    // 地理 AR 需要 iOS 14 及以上版本
    if (@available(iOS 14, *)) {
        if (ARGeoTrackingConfiguration.isSupported) {
            return [ARGeoTrackingConfiguration class];
        } else {
            NSLog(@"ARGeoTrackingConfiguration 在当前设备不支持 ");
        }
    }
    return NULL;
}

// 在 ARSession 会话中添加 ARGeoAnchor
void ARSession_addGeoAnchor(void* self, CLLocationCoordinate2D coordinate,
double altitude) {
    if (@available(iOS 14, *)) {
        // 将 void* 类型 ARSession 转换回 ARSession 类型
        ARSession* session = (__bridge ARSession*)self;

        // 执行添加 ARGeoAnchor 操作
        ARGeoAnchor* geoAnchor = [[ARGeoAnchor alloc] initWithCoordinate:
coordinate altitude:altitude];

        // 在当前 ARSession 中添加 geoAnchor
        [session addAnchor:geoAnchor];

        NSLog(@" 在纬度：%f, 经度：%f, 高度：%f 米处添加了 ARGeoAnchor 锚点 ",
coordinate.latitude, coordinate.longitude, altitude);
    }
}

// 在 ARKit 右手坐标系与 Unity 左手坐标系之间进行转换
static inline simd_float4x4 FlipHandedness(simd_float4x4 transform) {
    const simd_float4* c = transform.columns;
    return simd_matrix(simd_make_float4( c[0].xy, -c[0].z, c[0].w),
                       simd_make_float4( c[1].xy, -c[1].z, c[1].w),
                       simd_make_float4(-c[2].xy,  c[2].z, c[2].w),
                       simd_make_float4( c[3].xy, -c[3].z, c[3].w));
}

// 演示地理 AR 的操作使用，本示例只是简单地遍历所有 ARGeoAnchor 并打印其 transforms 信息
void DoSomethingWithSession(void* sessionPtr) {
    if (@available(iOS 14, *)) {
        ARSession* session = (__bridge ARSession*)sessionPtr;

        for (ARAnchor* anchor in session.currentFrame.anchors) {
            if ([anchor isKindOfClass:[ARGeoAnchor class]]) {
```

```
                ARGeoAnchor* geoAnchor = (ARGeoAnchor*)anchor;
                const simd_float4x4 transform = FlipHandedness(geoAnchor.transform);
                const simd_float4* c = transform.columns;
                NSLog(@"ARGeoAnchor %@ transform:\n"
                      "[%+f %+f %+f %+f]\n"
                      "[%+f %+f %+f %+f]\n"
                      "[%+f %+f %+f %+f]\n"
                      "[%+f %+f %+f %+f]\n",
                      [geoAnchor.identifier UUIDString],
                      c[0].x, c[1].x, c[2].x, c[3].x,
                      c[0].y, c[1].y, c[2].y, c[3].y,
                      c[0].z, c[1].z, c[2].z, c[3].z,
                      c[0].w, c[1].w, c[2].w, c[3].w);
            }
        }
    }
}
// 检查当前系统版本是否是 iOS 14 及以上
bool AR FoundationSamples_IsiOS14OrLater() {
    if (@available(iOS 14, *)) {
        return true;
    }
    return false;
}
```

将该文件放置到 Unity 工程 Assets/Plugins/iOS 文件夹下（这里只是为了便于管理，放其他路径下也没关系），选中该文件，在属性窗口（Inspector 窗口）中，勾选使用平台 iOS 和该平台框架 ARKit 后的复选框，如图 3-15 所示，以使 GeoAnchorsNativeInterop 代码能正确地在 iOS 平台下编译运行。

图 3-15　选择原生代码运行平台与所需特性

实现地理 AR 操作还是比较简单的，首先使用 ARGeoTrackingConfiguration 配置初始化 ARKit，如果当前配置不是 ARGeoTrackingConfiguration 配置，则需要进行替换和重新初始化。在此基础上，打开设备 GPS 定位功能，如果定位成功，则可以在该地理坐标位置添加 ARGeoAnchor 锚点，进而可以添加虚拟物体。也可以通过查询该地理位置所有的 ARGeoAnchor 锚点，获取锚点后就可恢复所有的虚拟物体对象，从而实现持久化的基于真实地理位置的 AR 体验。创建一个 C# 脚本，命名为 GeoAR，编写代码如下：

```
// 第 3 章 /3-4
using System;
using Unity.Collections;
```

```
using UnityEngine;
using UnityEngine.XR.AR Foundation;
using UnityEngine.XR.ARSubsystems;

#if UNITY_IOS
using System.Runtime.InteropServices;
using UnityEngine.XR.ARKit;
#endif

namespace Davidwang.Chapter3
{
    [RequireComponent(typeof(ARSession))]
    public class GeoAR : MonoBehaviour
    {
#if UNITY_IOS && !UNITY_EDITOR
        public static bool IsSupported => ARGeoAnchorConfigurationChooser.
ARGeoTrackingConfigurationClass != IntPtr.Zero;

        // 原生指针数据，用于原生代码与托管代码转换
        public struct NativePtrData
        {
            public int version;
            public IntPtr sessionPtr;
        }

        // 经纬坐标
        public struct CLLocationCoordinate2D
        {
            public double latitude;
            public double longitude;
        }

        // 开启 GPS 定位
        void Start() => Input.location.Start();

        void OnGUI()
        {
            GUI.skin.label.fontSize = 50;
            GUILayout.Space(100);

            if (ARGeoAnchorConfigurationChooser.ARGeoTrackingConfigurationClass ==
IntPtr.Zero)
            {
                GUILayout.Label("ARGeoTrackingConfiguration 在当前设备不支持 ");
                return;
            }
```

```
            switch (Input.location.status)
            {
                case LocationServiceStatus.Initializing:
                    GUILayout.Label("正在开启 GPS...");
                    break;
                case LocationServiceStatus.Stopped:
                    GUILayout.Label("定位服务已停止，无法使用地理 AR。");
                    break;
                case LocationServiceStatus.Failed:
                    GUILayout.Label("定位失败，无法使用地理 AR。");
                    break;
                case LocationServiceStatus.Running:
                    GUILayout.Label("定位成功，单击屏幕添加 ARGeoAnchor 锚点。");
                    break;
            }
        }

        void Update()
        {
            if (Input.location.status != LocationServiceStatus.Running)
                return;

            if (GetComponent<ARSession>().subsystem is ARKitSessionSubsystem
subsystem)
            {
                if (!(subsystem.configurationChooser is ARGeoAnchorConfiguratio
nChooser))
                {
                    // 检查初始配置文件，使用 ARGeoTrackingConfiguration 进行配置
                    subsystem.configurationChooser = new ARGeoAnchorConfigurati
onChooser();
                }

                // 使用设备电子罗盘进行方位对齐
                subsystem.requestedWorldAlignment = ARWorldAlignment
.GravityAndHeading;

                // 检查子系统指针情况
                if (subsystem.nativePtr == IntPtr.Zero)
                    return;

                // 进行 ARSession 类型转换
                var session = Marshal.PtrToStructure<NativePtrData>(subsystem
.nativePtr).sessionPtr;
                if (session == IntPtr.Zero)
                    return;
                // 检查屏幕手指单击操作
```

```
                    var screenTapped = Input.touchCount > 0 && Input.GetTouch(0)
.phase == TouchPhase.Ended;
                    if (screenTapped)
                    {
                        // 获取 GPS 数据
                        var locationData = Input.location.lastData;

                        // 调用原生代码，添加 ARGeoAnchor
                        AddGeoAnchor(session, new CLLocationCoordinate2D
                        {
                            latitude = locationData.latitude,
                            longitude = locationData.longitude
                        }, locationData.altitude);
                    }
                    // 执行自定义操作演示方法
                    DoSomethingWithSession(session);
                }
            }
            // 引入原生方法
            [DllImport("__Internal")]
            static extern void DoSomethingWithSession(IntPtr session);

            [DllImport("__Internal", EntryPoint = "ARSession_addGeoAnchor")]
            static extern void AddGeoAnchor(IntPtr session, CLLocationCoordinate2D
coordinate, double altitude);
    #else
            public static bool IsSupported => false;
    #endif
        }

    #if UNITY_IOS && !UNITY_EDITOR
        // 自定义 ConfigurationChooser，引导 ARKit 使用 ARGeoTrackingConfiguration 配置
        class ARGeoAnchorConfigurationChooser : ConfigurationChooser
        {
            static readonly ConfigurationChooser s_DefaultChooser = new
    DefaultConfigurationChooser();
            // 配置描述符
            static readonly ConfigurationDescriptor s_ARGeoConfigurationDescriptor =
new ConfigurationDescriptor(
                ARGeoTrackingConfigurationClass,
                Feature.WorldFacingCamera |
                Feature.PositionAndRotation |
                Feature.ImageTracking |
                Feature.PlaneTracking |
                Feature.ObjectTracking |
                Feature.EnvironmentProbes,
                0);
```

```
            public override Configuration ChooseConfiguration(NativeSlice<Configurat
ionDescriptor> descriptors, Feature requestedFeatures)
            {
                // 检测 GPS 状态，通过指针请求使用 ARGeoTrackingConfiguration 配置
                return Input.location.status == LocationServiceStatus.Running
                    ? new Configuration(s_ARGeoConfigurationDescriptor,
requestedFeatures.Intersection(s_ARGeoConfigurationDescriptor.capabilities))
                    : s_DefaultChooser.ChooseConfiguration(descriptors,
requestedFeatures);
            }
            // 桥接原生类
            public static extern IntPtr ARGeoTrackingConfigurationClass
            {
                [DllImport("__Internal", EntryPoint = "ARGeoTrackingConfiguration_
class")]
                get;
            }
        }
    #endif
    }
```

上述代码逻辑比较简单，使用时只需将该脚本挂载于场景中 ARSession 对象上。

使用 ARKit 地理 AR 需要满足 GPS 和点云地图同时可用才能正常运行，而基于真实地理位置的点云地图需要苹果公司预先对场景进行扫描并更新到地图中，由于地理信息的敏感性，国内使用还需时日。另外，开启 GPS 使用需要用户授权，所以 AR 应用应当在使用前进行 GPS 定位授权申请。

3.6　热管理

iOS 系统中有一个温度控制模块（Thermal Module）负责设备热管理，如环境温度过低时增加发热，而在设备温度过高时主动降温。温控模块通过监控设备温度状态，制定 CPU、GPU、DDR、GPS、蓝牙控制策略，如在设备温度非常高时会通过降低 CPU 频率降温，以防止蓝屏或者死机。

对 AR 应用而言，我们更关心设备温度过高的情况，因为 AR 应用是计算机密集型应用，其底层的 SLAM 算法对资源消耗很大，2D 图像检测跟踪、3D 物体检测跟踪、人体动捕、人形遮挡、光照估计、场景几何网格等功能都是性能消耗大户，使用时间稍长就会导致设备温度大幅度升高。通过前文的学习，我们知道 SLAM 运动跟踪对设备传感器参数极为敏感，温度的升高会导致跟踪热漂移，出现抖动、闪烁等问题，严重时会导致应用卡顿、假死。

为了防止出现类似问题，在设计 AR 应用时，一个比较合理的方案是根据设备热状态动态启停功能特性，如点云渲染、平面检测、人体动捕、光照估计等功能特性，可以根据设备

热状态开启或者关闭，动态进行性能效果平衡调节，防止性能恶化影响主功能的使用。

AR Foundation 并不直接支持获取 iOS 设备热状态，因此，我们采取与 3.5 节同样的思路，先通过原生代码做桥接，然后通过 C# 代码获取设备当前热状态，并根据当前设备热状态启用或者停用 AR 功能特性。首先，创建一个 ThermalStateForIOSProvider.mm 原生文件，编写代码如下：

```
   // 第 3 章 /3-5
   #import<Foundation/NSProcessInfo.h>
   #define EXPORT(_returnType_) extern "C" _returnType_ __attribute__ ((visibility
("default")))

   namespace
   {
       // 枚举，主要是将 iOS 热状态类型转换为 C# 枚举，枚举序号需要与 C# 中的 ThermalState 枚举
       // 保持一致
       enum ThermalState
       {
           kThermalStateUnknown = 0,
           kThermalStateNominal = 1,
           kThermalStateFair = 2,
           kThermalStateSerious = 3,
           kThermalStateCritical = 4,
       };
       // 将 iOS 热状态类型转换为枚举值
       inline ThermalState ConvertThermalState(NSProcessInfoThermalState thermalState)
       {
           ThermalState returnValue;

           switch (thermalState)
           {
               case NSProcessInfoThermalStateNominal:
                   returnValue = kThermalStateNominal;
                   break;
               case NSProcessInfoThermalStateFair:
                   returnValue = kThermalStateFair;
                   break;
               case NSProcessInfoThermalStateSerious:
                   returnValue = kThermalStateSerious;
                   break;
               case NSProcessInfoThermalStateCritical:
                   returnValue = kThermalStateCritical;
                   break;
               default:
                   returnValue = kThermalStateUnknown;
                   break;
           }
```

```
            return returnValue;
        }
    }
    // 主要的导出方法，获取 iOS 设备热状态并转换成枚举值
    EXPORT(ThermalState) Native_GetCurrentThermalState()
    {
        return ::ConvertThermalState([[NSProcessInfo processInfo] thermalState]);
    }
```

将该文件放置到 Unity 工程 Assets/Plugins/iOS 文件夹下，选中该文件，在属性窗口中，勾选使用平台 iOS 复选框。

下面演示在运行时根据设备热状态情况动态开启、关闭平面检测功能[①]，新建一个 C# 脚本，命名为 ThermalManager，代码如下：

```
    // 第 3 章 /3-6
    using System;
    using System.ComponentModel;
    using System.Runtime.InteropServices;
    using UnityEngine;
    using UnityEngine.XR.AR Foundation;

    public class ThermalManager : MonoBehaviour
    {
        [SerializeField]
        private ARPlaneManager mPlaneManager;                    // 平面管理器
        private ThermalState mPreviousThermalState = ThermalState.Unknown;
                                                                 // 前一个设备热状态

        void Update()
        {
            ThermalState thermalState = NativeApi.GetCurrentThermalState();
            // 状态发生改变并且当前发热严重
            if (mPreviousThermalState != thermalState && thermalState > ThermalState
.Serious)
            {
                mPlaneManager.enabled = false;                   // 关闭平面检测功能
            }
            else if (mPreviousThermalState != thermalState && thermalState <
ThermalState.Serious)
            {
                mPlaneManager.enabled = true;
            }
            mPreviousThermalState = thermalState;
```

① 第 4 章会详细阐述平面检测管理相关知识。

```
        }

        // 枚举，这里的序号需要与原生代码中的 ThermalState 枚举保持一致
        public enum ThermalState
        {
            [Description("Unknown")]
            Unknown = 0,
            [Description("Nominal")]
            Nominal = 1,
            [Description("Fair")]
            Fair = 2,
            [Description("Serious")]
            Serious = 3,
            [Description("Critical")]
            Critical = 4,
        }
        // 桥接原生方法
        static class NativeApi
        {
    #if UNITY_IOS && !UNITY_EDITOR
            // 导入原生方法
            [DllImport("__Internal", EntryPoint = "Native_GetCurrentThermalState")]
            public static extern ThermalState GetCurrentThermalState();
    #else
            public static ThermalState GetCurrentThermalState() => ThermalState
.Unknown;
    #endif
        }
    }
```

上述代码逻辑非常简单，首先通过原生 API 获取当前设备热状态，如果当前设备热状态与前一帧状态不同，则说明热状态发生了改变，这时比较当前设备热状态与设定的阈值，如果超过指定阈值，则关闭平面检测功能；如果低于指定阈值，则开启平面检测功能。

在使用时，只需将 ThermalManager 脚本挂载到场景对象上，并设置好平面管理器属性，AR 应用运行时会实时检查设备热状态并执行平面检测功能开启或关闭操作。

热管理对维持应用正常运行非常重要，特别是对 AR 应用这类计算密集型应用，通过在功能特性与设备性能之间进行动态调节，可以将用户体验维持在一个比较高的水平。在效果与性能之间进行折中，这也是 3D 游戏开发经常使用的技巧。

3.7 AR 轻应用

随着以用户体验为中心的软件设计思想的进步和网络通信技术的发展，各类小程序、快应用、即时应用获得了极大成功，用户不再需要为每个应用安装一个 App，通过一个统一的

入口即可以以随用随开启、用完即走的方式使用一段功能、完成一项任务，这是以用户体验为中心的思想在应用形态上的表现。

苹果轻应用（App Clips）与此类似，将执行代码放置在 App Store 仓库中，当用户需要使用时，将执行代码从 App Store 仓库下载到本地运行，使用后即可释放，不要求用户安装特定应用。通过这种方式，也可以实现 AR 轻应用，用户通过扫描如图 3-16 所示的 App 二维码即可触发 AR 体验。

图 3-16　App 轻应用二维码示例

通过轻应用开发（App Clips Codes），就可以先将 AR 应用执行代码及资源文件上传到 App Store 仓库，生成 App 轻应用二维码（以下简称二维码），生成的这些二维码可以打印后粘贴到海报、产品、场所等地方，使用时，用户只需打开其相机扫描这些二维码便可触发 AR 体验。这种 AR 体验方式具有广阔的应用前景，例如在产品包装盒上印上二维码，用户扫描后即可在真实场景中呈现产品，这对用户直观了解产品外观、尺寸、细节等非常有帮助，而且这些二维码还可以与 NFC（Near Field Communication，近场通信）技术结合，自动触发 AR 体验，人机交互更自然。

目前AR轻应用只支持原生开发，具体细节不再详述，读者如感兴趣则可以查阅相关资料。