第 3 章

ARCore 功能特性与开发基础

ARCore 是一个高级 AR 应用开发引擎，高内聚、易使用，特别是集成到 AR Foundation 框架后，提供了简洁统一的使用界面，这使利用其开发 AR 应用变得非常高效。ARCore 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），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 ARCore 本身的技术能力和与 ARCore 扩展包相关的知识。

3.1 ARCore 概述及主要功能

2017 年 6 月，苹果公司发布了 ARKit SDK，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 ARKit 只能用于苹果公司自家的移动终端，无法应用到 Android 平台。2017 年 8 月，谷歌公司正式发布对标 ARKit SDK 的 ARCore[①]，将 AR 能力带入 Android 平台，ARCore 也是一套用来创建 AR 的 SDK，利用该工具包可以为现有及将来的 Android 手机提供 AR 功能，它通过采集环境信息及传感器数据使手机 / 平板具备感知环境、了解现实世界、支持虚实交互的能力，ARCore 还为 Android 和 iOS 平台同时提供了 API，支持 AR 体验共享。

ARCore 主要做两件工作，即跟踪用户设备姿态和构建对现实环境的理解。ARCore 利用 SLAM 技术进行运动跟踪并构建环境三维结构信息，除此之外，ARCore 还支持检测平坦的表面（如桌面或地面）、估计周围环境的光照信息。借助 ARCore 对现实世界的理解，我们能够以一种与现实世界无缝融合的方式添加虚拟物体、注释或其他信息，例如可以将一只打盹的小猫放在咖啡桌的一角，或者利用艺术家的生平信息为一幅画添加注释。

虽然 ARCore 出现时间比 ARKit 晚，但事实上 ARCore 项目开展比 ARKit 早，在 2014 年，谷歌公司展示了其 Tango 项目成果，Tango 是谷歌公司基于 FlyBy 公司 VIO 技术发展起来的 AR 技术，技术比 ARCore 更复杂，融合了更多传感器，需要额外的硬件辅助实现增强现实，过高的门槛使消费者甚至开发者难以触及，这导致 Tango 技术并未能大规模推广。苹

① ARCore 后来被更名为 Google Play Services for AR，本书遵循原名，仍使用 ARCore 进行描述。

果公司的 ARKit SDK 发布后，由于其基于现有移动终端硬件平台，不需要额外硬件支持，大大降低了 AR 使用者的门槛，取得了良好的市场反响。为抢占技术高点，谷歌公司也迅速在 Tango 的基础上推出了 ARCore，并提供了 Android、iOS、Unity、Unreal、Java 多个开发平台的 API。

ARCore 是在移动手机广泛普及的情况下，谷歌公司适应时代潮流推出的 AR 开发工具，但其可以追溯到 Tango 这个已经研究很长时间的 AR 技术项目，事实上 ARKit 也基于 FlyBy 公司的技术，因此，ARKit 和 ARCore 技术同源且具有基本相似的功能。

ARCore 通过 3 个主要功能融合虚拟内容与现实场景，即运动跟踪、环境理解、光照估计。这 3 个功能是 ARCore 实现各项 AR 功能的基础，也是 ARCore 技术框架中最重要的支柱。

3.1.1　运动跟踪

在三维空间中跟踪设备运动并最终定位其位姿是任何 AR 应用的基础，ARCore 通过 SLAM 算法采集来自设备摄像头的图像数据和来自运动传感器的运动数据，综合分析并估测设备随着时间推移而相对于周围世界的姿态变化，如图 3-1 所示。通过将渲染三维内容的虚拟相机姿态与 SLAM 算法计算出的设备摄像头姿态对齐，就能够从正确的透视角度渲染虚拟内容，将渲染的虚拟内容叠加到从设备摄像头获取的图像上，从而实现虚实场景的几何一致性，使虚拟内容看起来就像真实世界的一部分。

图 3-1　ARCore 运动跟踪示意图

3.1.2　环境理解

ARCore 通过检测环境特征点和平面来不断改进它对现实世界环境的理解，通过检查位于常见水平或垂直表面（如桌子或墙壁）上的成簇特征点构建平面，并确定平面的边界，利用这些数据信息就可以将虚拟物体放置于平坦的真实环境表面上，实现虚拟物体和真实环境无缝融合，营造虚实一致的体验，如图 3-2 所示。

图 3-2　ARCore 通过检测特征点和平面来改进对环境的理解

3.1.3　光照估计

ARCore 可以检测设备所在空间环境光的相关信息，估计光照强度和色彩，利用这些光照信息，可以使用与真实环境光相同的光照信息渲染虚拟物体，使虚拟物体和周边真实物体光照效果保持一致，提升虚拟物体的真实感，如图 3-3 所示，处于强光中的虚拟猫模型和处于阴影中的虚拟猫模型光照效果保持了与真实场景中光照的一致性。ARCore 还可以估计光源的位置和方向，从而支持虚拟物体生成与真实光照一致的阴影，进一步提升虚拟物体的真实感。

图 3-3　ARCore 光照估计效果示意图

3.1.4　ARCore 的不足

1. 运动跟踪失效

ARCore SDK 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，ARCore 的运动跟踪在以下情况时会失效。

1）在运动中进行运动跟踪

如果用户在火车或者电梯中使用 ARCore，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），这样将导致 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

如果用户将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，这将引起图像特征点提取匹配问题，进而导致跟踪失败。

3）热漂移

相机感光元器件与 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一个或者几个特定温度，但在用户设备使用过程中，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，表现出来的效果就是跟踪的物体漂移。

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，进而影响跟踪的准确性，这也会导致基于 VIO 的运动跟踪失败。

2. ARCore 存在的不足

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

1）环境理解需要时间

ARCore 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，ARCore 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 ARCore 更快地完成初始化。

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 ARCore 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

ARCore 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。

4）鬼影现象

虽然 ARCore 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还不够准确，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。

3.2 运动跟踪原理

在第 1 章中，我们对 AR 技术原理进行过简要学习，ARCore 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。本节将更加深入地学习 ARCore 运动跟踪原理，从而加深对 ARCore 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 ARCore 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。

在进一步学习运动跟踪之前，先了解一下 ARCore 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）。ARCore 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 3-4 所示。

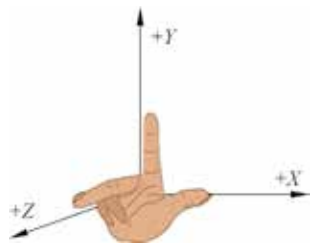

图 3-4 ARCore 采用右手坐标系

当用户在实际空间中移动时，ARCore 坐标系上的距离增减遵循表 3-1 所示的规律。

表 3-1 ARCore 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 ARCore 运动跟踪分类

ARCore 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 3-2 所示。

表 3-2　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的使用，通常还会使用四元数来表达旋转，四元数允许以简单的形式定义三维旋转的所有方面。

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 值使用 3DoF 方向跟踪，即只跟踪设备在 X、Y、Z 轴上的旋转运动，如图 3-5 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。

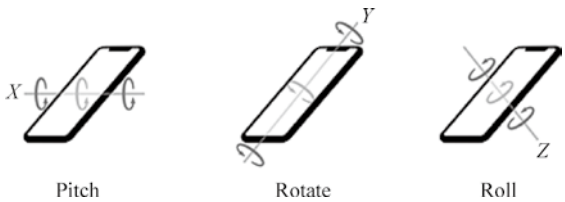

图 3-5　ARCore 中 3DoF 跟踪示意图

当采用 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 Mode 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，既跟踪设备在 *X*、*Y*、*Z* 轴上的旋转运动，也跟踪设备在 *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 3-6 所示。

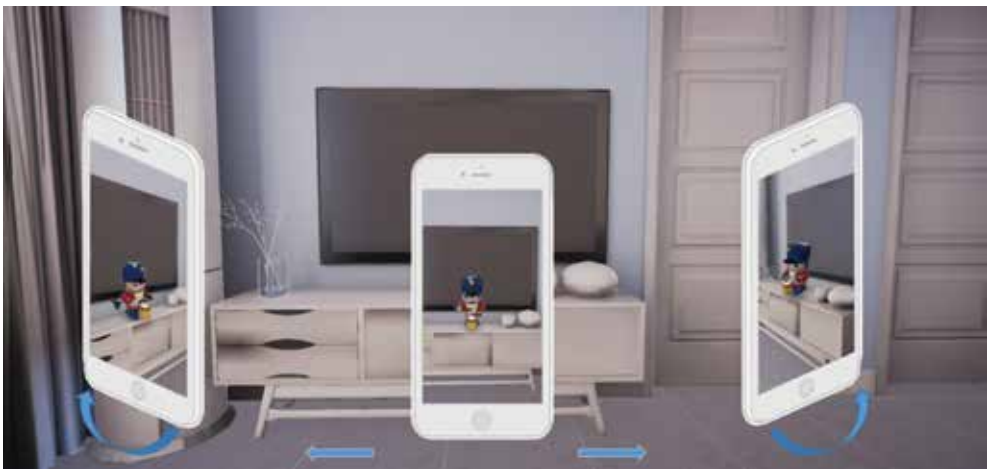

图 3-6　ARCore 中 6DoF 跟踪示意图

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 ARCore 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二维图像等。

3.2.3　ARCore 运动跟踪

ARCore 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。

ARCore 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。ARCore 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，对相机运动速度非常敏感，因为快速的相机运动会导致采集的图像模糊。

ARCore 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小时间跨度的跟踪，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 信息来自于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 3-7 所示。

图 3-7　VIO 进行空间计算原理图

在 AR 应用启动后，ARCore 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），ARCore 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，如图 3-7 所示，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，换而言之，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，从而无法解算出空间信息，因此，若要在 AR 应用中使用 VIO，则设备必须移动位置（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

被加入，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 ARCore 初始化时的设备位置，点云地图就是现实世界在 ARCore 中的数字表达。

VIO 跟踪的流图如图 3-8 所示，从流图可以看到，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。

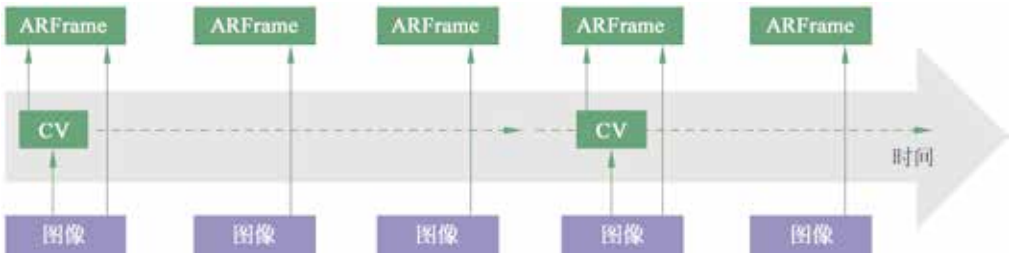

图 3-8　VIO 跟踪流程示意图

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，ARCore 使用锚点（Anchor）的方式绑定虚拟元素与现实空间环境，同时 ARCore 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），ARCore 就会对特征点的信息进行修正，从而优化跟踪（回环检测）。

ARCore 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 ARCore 制作的 AR 应用体验非常好。

3.2.4　ARCore 使用运动跟踪的注意事项

通过对 ARCore 运动跟踪原理的学习，现在可以很容易地理解前文所列 ARCore 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，不然会使用户困惑。ARCore 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暂停（PAUSED）、停止（STOPPED）、

跟踪（TRACKING）3 种，分别指示运动跟踪状态的 3 种情况（在整合进 AR Foundation 时，状态由 ARSessionState 枚举描述）。为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，并不是所有的 Android 手机都支持 ARCore，因此，通常在使用 ARCore 之前需要进行一次设备支持性检查。在 AR Foundation 框架中，可使用 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 3-3 所示。

表 3-3　ARSessionState 枚举值

枚 举 值	描　　述
None	状态未知，未安装 ARCore
Unsupported	当前设备不支持 ARCore
CheckingAvailability	系统正在进行设备支持性检查
NeedsInstall	设备硬件支持 ARCore，但需要安装 ARCore 工具包

续表

枚 举 值	描　　述
Installing	设备正在安装 ARCore
Ready	设备支持 ARCore 并且已做好使用准备
SessionInitialized	ARCore 会话正在初始化，还未建立运动跟踪
SessionTracking	ARCore 会话运动跟踪中

在 ARCore 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，此外还可以通过订阅 ARSession.stateChanged 事件处理运动跟踪变化情况。

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。在启动 AR 应用后，ARCore 需要一点时间来收集足够多的视觉特征点信息，在这个过程中，ARCore 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），ARCore 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 3-9 所示，在刚启动 AR 应用时，ARCore 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。

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

图 3-9　ARCore 跟踪开始后的状态变化

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，ARCore 的跟踪状态会发生变化，如图 3-10 所示。

图 3-10　ARCore 跟踪状态会受到设备及环境的影响

在 ARCore 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、二维图像检测等。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，ARCore 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 ARCore 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 ARCore 中断结束后会自动尝试进行重定位（Relocalization），重定位如果成功，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 3-11 所示。

图 3-11　ARCore 跟踪中断及重定位时的状态变化

提示

重定位是一件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 事件获取跟踪状态变化，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.ARFoundation;
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 各枚举值如表 3-4 所示。

表 3-4　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	环境中特性点不足
ExcessiveMotion	设备移动太快，造成图像模糊
Unsupported	Provider 不支持跟踪受限原因
CameraUnavailable	设备摄像头不可用

3.5　ARCore 扩展包

AR Foundation 是一个上层封装框架，由于要考虑不同底层的共通性，不能做到对底层 SDK 所有功能特性都完全支持，因此，ARCore 提供了一个扩展包（ARCore Extensions），用于实现其特定的功能和更精细的控制，如对硬件摄像头、云锚点、会话录制（Session Record）的配置控制，并支持 iOS 端使用云锚点与 Android 互通，实现 AR 体验跨端共享。

ARCore 扩展包是一个可选包，如果不需要上文提及的功能，则可以不引入，ARCore 扩展包并不能直接在 Unity 包管理器（Unity Package Manager，UPM）中导入，需要先下载[①]或者从 Unity 包管理器中填写 ARCore 扩展包的 GitHub 地址进行导入。下面以使用 UPM 导入进行演示，在 Unity 菜单栏中，依次选择 Window → Package Manager，打开 UPM 对话框，单击该对话框左上角的“+”符号，在下拉菜单中选择 Add package from git URL 选项，如图 3-12 所示，在弹出的网址输入框中填写 https://github.com/google-ar/arcore-unity-extensions.git。

图 3-12　直接从 git URL 中导入 ARCore 扩展包

在导入 ARCore 扩展包后，为了方便统一管理配置文件，在工程窗口（Project 窗口）中新建一个文件夹，命名为 Config，然后在空白处右击，在弹出菜单中依次选择 Create → XR → ARCore Extensions Config 创建 ARCoreExtensionsConfig 配置文件，以同样的操作，在弹出菜单中依次选择 Create → XR → Camera Config Filter 创建 ARCoreExtensionsCameraConfigFilter 配置文件。这两个配置文件，一个用于配置云锚点，另一个用于配置设备摄像头使用参数，在后文使用时再详细阐述。

然后在层级窗口（Hierarchy 窗口）中右击，在弹出菜单中依次选择 XR → ARCore Extensions 创建 ARCore Extensions 对象，选择该对象，在属性窗口（Inspector 窗口）中为 ARCore Extensions 组件各属性赋值（将层级窗口中各相应对象和上文创建的两个配置文件设

① 下载最新版本的网址为 https://github.com/google-ar/arcore-unity-extensions/releases。

置到相应的属性列中），如图 3-13 所示。

图 3-13　为 ARCore Extensions 组件各属性赋值

至此，已经导入并完成了 ARCore 扩展包配置，在具体使用时再详细阐述各功能配置的使用流程。

3.6　相机配置

移动设备基本有前置摄像头、后置摄像头、IMU 传感器，ARCore 利用这些传感器数据探索和理解现实环境。设备可能还有深度传感器，能够检测目标距离、尺寸、范围等数据。这些传感器还有不同的性能技术参数，如摄像头可以以 30 帧 / 秒或者 60 帧 / 秒运行，所有这些构成了移动设备硬件环境的多样性。事实上，ARCore 可以充分利用这些传感器数据，融合各类数据提供更好的使用体验。也可以通过 ARCore 扩展包进行相应配置以优化性能，如可以使用相机图像元数据、帧缓冲和相机访问共享优化性能。

在 AR Foundation 中，摄像头配置（Camera Configurations）描述了设备硬件传感器属性，可以通过其子系统的 XRCameraConfiguration 类访问这些属性，在 Android 平台，ARCore 扩展包在 XRCameraConfiguration 类基础之上提供了 XRCameraConfigurationExtensions 配置类，用于获取特定于 ARCore 的属性，可以使用这些属性进行更加细致的控制。

通过 ARCameraManager.GetConfigurations() 方法可以获取所有当前设备支持的相机配置，通过 CameraManager.currentConfiguration 可以设置设备使用的相机参数，典型代码如下[①]：

```
// 第 3 章 /3-3
using UnityEngine.XR.ARFoundation;
using Google.XR.ARCoreExtensions;

public ARCameraManager cameraManager;   //ARCameraManager 组件

private void CameraConfigurations()
{
     using (NativeArray<XRCameraConfiguration> configurations = cameraManager.
GetConfigurations(Allocator.Temp))
    {
        if (!configurations.IsCreated || (configurations.Length <= 0))
```

① 关于相机配置的更详细信息参见第 9 章。

```
        {
            return;
        }

        //迭代获取最大分辨率配置
        var desiredConfig = configurations[0];
        for (int i = 1; i < configurations.Length; ++i)
        {
            //选择最大帧率和分辨率的配置，打开深度传感器
            if (configurations[i].GetFPSRange().y > desiredConfig.GetFPSRange().y &&
                configurations[i].GetTextureDimensions().x > desiredConfig.
GetTextureDimensions().x &&
                configurations[i].GetTextureDimensions().y > desiredConfig.
GetTextureDimensions().y &&
                configurations[i].CameraConfigDepthSensorUsage() ==
CameraConfigDepthSensorUsage.RequireAndUse)
            {
                desiredConfig = configurations[i];
            }
        }

        //设置相配配置
        if (desiredConfig != cameraManager.currentConfiguration)
        {
            cameraManager.currentConfiguration = desiredConfig;
        }
    }
}
```

选择 3.5 节创建的 ARCoreExtensionsCameraConfigFilter 配置文件，在属性窗口中展开其属性值，如图 3-14 所示。

图 3-14　ARCore 扩展包相机配置界面

Target Camera Framerate 属性用于限制相机最高帧率，在支持 60 帧 / 秒的设备上，ARCore 默认会使用高帧率，这会对应用性能带来挑战，也可以在下拉列表框中选择 30 帧 / 秒以限制最高帧率。

Depth Sensor Usage 属性用于设置是否使用深度传感器，在配备有深度传感器的设备上，ARCore 默认会开启深度传感器，这将提供更好的 AR 体验，也可以选择 DoNotUse 禁用深度

传感器。

Stereo Camera Usage 属性用于设置是否使用双目传感器，在配备了双目传感器的设备上，ARCore 默认会开启该传感器，这将提高 SLAM 跟踪稳定性和质量，也可以选择 DoNotUse 禁用双目传感器。

除了在编辑时静态配置，也可以在应用运行时通过 ARCoreExtensions.OnChooseXRCameraConfiguration 事件动态地进行配置，典型代码如下：

```
// 第 3 章 /3-4
public void Awake()
{
    …
    arcoreExtensions.OnChooseXRCameraConfiguration = SelectCameraConfiguration;
    …
}

// 执行动态相机配置
int SelectCameraConfiguration(List<XRCameraConfiguration> supportedConfigurations)
{
    int index = 0;
    …                    // 自定义相机配置逻辑
    return index;
}
```

需要注意的是，使用动态配置时，必须确保在 ARCore 会话启动之前完成配置，通常是在 Awake() 方法中进行，这样才能在 AR 运行开始前生效。

3.7 会话录制与回放

如前文所述，AR 应用调试是一件非常烦琐且费时费力的工作，特别是对需要到现场进行调试的应用，如文旅、导航等与现地位置相关的应用，非常不友好，为解决这个问题，ARCore 引入了会话录制与回放功能（Session Record & Playback），利用该功能可以预先录制场景数据信息，在调试 AR 应用时可以回放这些场景数据并进行相应操作，因此可以对录制的会话进行重用，加速调试过程。

会话录制时会录制满足正常 SLAM 运动跟踪的所有信息，包括图像数据、IMU 数据及其他传感器数据（如 ToF 传感器），因此录制完的会话可以在其他平台回放，如图 3-15 所示，在回放时，AR 应用可以正常实现其他功能，如平面检测、点云重建等，与使用真机设备进行测试一样，从而达到一次录制多平台回放、会话重用的目的。

通过会话录制与回放，可以大幅度降低实地现场测试的需求，从而提高应用迭代速度。在默认情况下，出于计算性能考虑，ARCore 使用 640 × 480 像素分辨率的摄像头图像进行 SLAM 运动跟踪，而不是渲染到屏幕上的高分辨率图像，因此，默认的会话录制也会使用

640 × 480 像素分辨率，通常这能满足回放的各项要求。如果确实希望录制高分辨率图像流，则必须对摄像头进行配置，此时 ARCore 进行运动跟踪依然会使用 640 × 480 像素分辨率图像，同时，还会采集配置文件中配置的高分辨率图像流，该图像流仅用于播放（呈现效果），并且会将高分辨率图像存储于视频主轨道，而将用于 ARCore 运动跟踪的低分辨率图像存储于其他轨道，需要注意的是，同时捕获两个图像流会影响应用程序的性能，并且不会提高会话回放的质量。

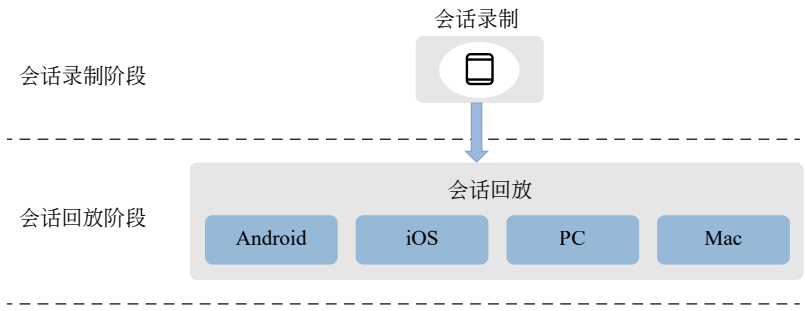

图 3-15　通过会话录制可以达到一次录制、多平台回放的目的

除此之处，还可以采集其他传感器数据，如设备的硬件深度传感器数据（如果有）。下面分成两种实现方式阐述会话录制与回放功能。

3.7.1　AR Foundation 录制与回放

AR Foundation 支持 ARCore 会话录制与回放[①]，该功能已内置到其子系统中，因此使用起来非常方便，在 Unity 中新建 SessionRecordAndPlayback.cs 脚本，编写代码如下：

```
// 第 3 章 /3-5
using System.IO;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARCore;

public class SessionRecordAndPlayback : MonoBehaviour
{
    private ARSession mSession;                    // 会话组件
    private string mVideoFilePath;                 // 文件存储路径
    void Awake()
    {
        mSession = GetComponent<ARSession>();
        mVideoFilePath = Path.Combine(Application.persistentDataPath,
"ARCoreSession.mp4");
```

① 该功能需要 AR Foundation 4.2 及以上版本。

```
    }
    //开始录制会话
    public void StartRecord()
    {
        if (mSession.subsystem is ARCoreSessionSubsystem subsystem)
        {
            var session = subsystem.session;
            if (session == null)
                return;
            var recordingStatus = subsystem.recordingStatus;

            if (!recordingStatus.Recording())
            {
                using (var config = new ArRecordingConfig(session))
                {
                    config.SetMp4DatasetFilePath(session, mVideoFilePath);
                    config.SetRecordingRotation(session, GetRotation());
                    var status = subsystem.StartRecording(config);
                    DeBug.Log($"开始录制会话 {config.GetMp4DatasetFilePath
(session)} => {status}");
                }
            }
        }
    }
    //结束会话录制
    public void StopRecord()
    {
        if (mSession.subsystem is ARCoreSessionSubsystem subsystem)
        {
            var session = subsystem.session;
            if (session == null)
                return;
            var recordingStatus = subsystem.recordingStatus;
            if (recordingStatus.Recording())
            {
                var status = subsystem.StopRecording();
                if (status == ArStatus.Success)
                {
                    DeBug.Log(File.Exists(mVideoFilePath)
                        ? $"会话文件已保存到 {mVideoFilePath} ({GetFileSize(mVideo
FilePath)})"
                        : "会话结束，但无会话文件");
                }
            }
        }
    }
```

```
    //开始回放会话
    public void StartPlayback()
    {
        if (mSession.subsystem is ARCoreSessionSubsystem subsystem)
        {
            var session = subsystem.session;
            if (session == null)
                return;
            var playbackStatus = subsystem.playbackStatus;

            if (!playbackStatus.Playing() && File.Exists(mVideoFilePath))
            {
                var status = subsystem.StartPlayback(mVideoFilePath);
                DeBug.Log($"开始回放会话 ({mVideoFilePath}) => {status}");
            }
        }
    }
    //结束会话回放
    public void StopPlayback()
    {
        if (mSession.subsystem is ARCoreSessionSubsystem subsystem)
        {
            var session = subsystem.session;
            if (session == null)
                return;
            var playbackStatus = subsystem.playbackStatus;
            if (playbackStatus.Playing() || playbackStatus == ArPlaybackStatus.
Finished)
            {
                var status = subsystem.StopPlayback();
                if(status == ArStatus.Success)
                   DeBug.Log($"结束会话 => {status}");
            }
        }
    }
    #region Helper
    //根据手机旋转对录制的会话视频进行反向旋转
    private static int GetRotation() => Screen.orientation switch
    {
        ScreenOrientation.Portrait => 0,
        ScreenOrientation.LandscapeLeft => 90,
        ScreenOrientation.PortraitUpsideDown => 180,
        ScreenOrientation.LandscapeRight => 270,
        _ => 0
    };
    //获取文件大小
    private static string GetFileSize(string path)
```

```
    {
        var ByteCount = new FileInfo(path).Length;
        const long kiloBytes = 1024;
        const long megaBytes = kiloBytes * kiloBytes;

        if (ByteCount / megaBytes > 0)
        {
            var size = (float)ByteCount / megaBytes;
            return $"{size:F2} MB";
        }

        if (ByteCount / kiloBytes > 0)
        {
            var size = (float)ByteCount / kiloBytes;
            return $"{size:F2} KB";
        }
        return $"{ByteCount} Bytes";
    }
    #endregion
}
```

在使用时，将该脚本挂载到场景中的 AR Session 对象上，使用按钮事件触发开始录制、结束录制、开始回放、结束回放 4 个相应的方法即可。

在真机设备运行时，录制会话后会采集设备图像、IMU 等各传感器数据，并将这些数据保存为 MPEG-4 格式视频文件，在回放会话时，应用会重放录制的视频（IMU 等传感器数据也一并进行重放），在回放会话时，ARCore 运动跟踪数据来源于录制的会话，与真机设备实时数据无关，真机设备可以保持不动，但回放的会话就跟使用真机扫描环境一样，同样也可以与场景进行交互，实现如平面检测、虚拟物体放置等功能和操作，与使用真机设备进行测试效果无异。

3.7.2 ARCore 扩展录制与回放

使用 AR Foundation 内置功能进行录制和回放，使用简单，但无法进行更多的操作，如添加自定义数据、在其他平台回放会话等。使用 ARCore 扩展包也可以进行会话录制与回放，并且提供了更多的操作灵活性，使用 ARCore 扩展包进行会话录制与回放，首先需要在场景中正确配置该扩展包（详见 3.5 节），然后利用该扩展包提供的 AR Recording Manager 和 AR Playback Manager 组件进行功能处理，进行会话录制与回放的典型代码如下：

```
// 第 3 章 /3-6
// 会话录制
// 通过 ARCoreRecordingConfig 可编程对象配置会话录制相关属性
ARCoreRecordingConfig recordingConfig = ScriptableObject.CreateInstance<ARCoreReco
rdingConfig>();
```

```
recordingConfig.Mp4DatasetUri = /* 存储会话录制后文件的路径 */;
// 开始录制
recordingManager.StartRecording(recordingConfig);
// 结束录制
recordingManager.StopRecording();

// 会话回放
session.enabled = false;
playbackManager.SetPlaybackDatasetUri(uri);    // 设置会话文件路径
// 开始回放
session.enabled = true;

// 结束会话回放实际上是将原会话文件路径置空
session.enabled = false;
playbackManager.SetPlaybackDatasetUri(null);
session.enabled = true;
```

通过上述代码可以看到，会话回放实际上暂停了 AR 会话，然后使用录制好的会话进行 AR 会话重启。

ARCore 扩展包会话录制与回放的一大优势是在录制视频和传感器数据之外还可以同时录制自定义数据，这些录制的数据可以在回放的时候被提取出来，并且在相同的时刻被回放，如通过 RecordTrackData() 方法在 00:05:37 时刻录制了自定义数据，在会话回放时，该数据也会在 00:05:37 时刻被回放。

ARCore 不会自动添加任何开发人员自定义的数据，但允许开发者在特定的帧添加数据并在回放到该帧时取回数据，通过这种方式，增强了会话录制与回放功能的实用性。如可以在室外录制会话时同时录制 GPS 信息，在会话回放时这些 GPS 信息同样有效；或者可以通过录制会话，添加自定义信息并共享给其他人。

ARCore 扩展包在录制与回放开发人员自定义数据时借助了视频技术中轨道（track）的数据通道，一个轨道就是一个独立的数据通道，类似于图像处理软件中的图层，在视频播放时，多轨道数据通过时钟信号同步。录制自定义数据实际上就是添加一个独立的数据轨道，并将其设置到配置文件中，为了方便地提取到相关轨道，每个轨道都使用一个 GUID 值标识，典型代码如下：

```
// 第 3 章 /3-7
var track = new Track {
 Id = Guid.Parse("de5ec7a4-09ec-4c48-b2c3-a98b66e71893")    // 生成的 GUID 值
};

List<Track> tracks = new List<Track>();
tracks.Add(track);
recordingConfig.Tracks = tracks;
```

元数据是用于描述轨道的数据，与图像帧和时间无关，通常用于描述轨道数据来源、位置、

用途等信息，添加元数据的典型代码如下：

```
//第 3 章 /3-8
Byte[] metadata = System.Text.Encoding.Default.GetBytes("新的元数据");
track.Metadata = metadata;
```

默认的元数据类型为 application/text，当然也可以是其他数据类型，如逗号分隔值（Comma Separated Values，CSV）格式等，用于存储更多的描述数据，如果使用 CSV 格式，则需要将 MIME 类型修改为 text/csv，代码如下：

```
//第 3 章 /3-9
track.MimeType = "text/csv";
```

元数据与时间无关，与时间相关的自定义数据在存储时与图像帧同步，即这些数据与时间轴相关，录制会话时添加自定义数据的时间与回放会话时提取该数据的时间保持一致，添加自定义数据的典型代码如下：

```
//第 3 章 /3-10
public void onRecordCustomData()
{
    var customData = new Byte[] { 12,34,56,78 };
    mRecordingManager.RecordTrackData(track.Id, customData);
}
```

因为自定义数据与时间轴、图像帧相关，在提取这些数据时，需要监视每帧会话中的数据轨道，在轨道中有数据时将其提取出来，所以提取自定义数据操作通常要放在 update() 这类每帧都会执行的方法中执行，典型代码如下：

```
//第 3 章 /3-11
if(isPlayback)
{
    var trackDataList = mPlaybackManager.GetUpdatedTrackData(track.Id);
    foreach (TrackData trackData in trackDataList)
    {
        var data = trackData.Data;
        var customData = (int)(data[0]);
        DeBug.Log($"元数据是{track.Metadata.ToString()}：自定义数据是：{customData}");
    }
}
```

一个完整的利用 ARCore 扩展包进行会话录制与回放的代码如下：

```
//第 3 章 /3-12
using System.IO;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using Google.XR.ARCoreExtensions;
```

```
using System.Collections;
using System.Collections.Generic;

public class SessionRecordAndPlayback : MonoBehaviour
{
    private ARSession mSession;                    // 会话组件
    private ARRecordingManager mRecordingManager; //ARRecordingManager 组件
    private ARPlaybackManager mPlaybackManager;   //ARPlaybackManager 组件
    private string mVideoFilePath;                 // 会话文件存储路径
    private Track track;                           // 视频文件轨道，用于存储自定义数据
    private bool isPlayback = false;
    void Start()
    {
        mSession = GetComponent<ARSession>();
        mRecordingManager = GetComponent<ARRecordingManager>();
        mPlaybackManager = GetComponent<ARPlaybackManager>();
        mVideoFilePath = Path.Combine(Application.persistentDataPath,
“ARCoreSession2.mp4”);
        track = new Track
        {
            Id = System.Guid.NewGuid()    // 该 GUID 应当保存起来
        };

    }
    private void Update()
    {
        if(isPlayback)
        {
            var trackDataList = mPlaybackManager.GetUpdatedTrackData(track.Id);
            foreach (TrackData trackData in trackDataList)
            {
                var data = trackData.Data;
                var customData = (int)(data[0]);
                DeBug.Log($”元数据是 {track.Metadata.ToString()}：自定义数据是：
{customData}”);
            }
        }
    }
    // 开始录制会话
    public void StartRecord()
    {
        ARCoreRecordingConfig recordingConfig = ScriptableObject.CreateInstance<AR
CoreRecordingConfig>();
        recordingConfig.Mp4DatasetUri = new System.Uri(mVideoFilePath);
        // 添加元数据
        Byte[] metadata = System.Text.Encoding.Default.GetBytes(" 新的元数据 " );
        track.Metadata = metadata;
```

```
        List<Track> tracks = new List<Track>();
        tracks.Add(track);
        recordingConfig.Tracks = tracks;

        mRecordingManager.StartRecording(recordingConfig);
        DeBug.Log("开始录制会话");
    }
    //添加自定义数据
    public void onRecordCustomData()
    {
        var customData = new Byte[] { 12,34,56,78 };
        mRecordingManager.RecordTrackData(track.Id, customData);
    }
    //结束录制会话
    public void StopRecord()
    {

        if (mRecordingManager.RecordingStatus == RecordingStatus.OK)
            mRecordingManager.StopRecording();
        DeBug.Log(File.Exists(mVideoFilePath)
                        ? $"会话文件已保存到 {mVideoFilePath} ({GetFileSize(mVideo
FilePath)})"
                        : "会话结束，但无会话文件");
    }
    //开始回放会话
    public void StartPlayback()
    {
        DeBug.Log("开始回放会话");
        if (mPlaybackManager.PlaybackStatus == PlaybackStatus.None && File.
Exists(mVideoFilePath))
        {
            StartCoroutine(StartSessionPlayback());
        }
    }
    //结束会话回放
    public void StopPlayback()
    {
        isPlayback = false;
        if (mPlaybackManager.PlaybackStatus == PlaybackStatus.OK ||
mPlaybackManager.PlaybackStatus == PlaybackStatus.FinishedSuccess)
        {
            StartCoroutine(StopSessionPlayback());
        }
    }

    #region Helper
    //使用协程辅助延时
```

```
    private IEnumerator StartSessionPlayback()
    {
        mSession.enabled = false;
        while (true)
        {
            var result = mPlaybackManager.SetPlaybackDatasetUri(new System.
Uri(mVideoFilePath));
            if (result == PlaybackResult.ErrorPlaybackFailed || result ==
PlaybackResult.SessionNotReady)
            {
                yield return new WaitForSeconds(0.5f);  // 等待 500ms
            }
            else
            {
                break;
            }
        }
        mSession.enabled = true;
        isPlayback = true;
        DeBug.Log(" 开始回放会话 ");
    }
    // 使用协程辅助延时
    private IEnumerator StopSessionPlayback()
    {
        mSession.enabled = false;
        yield return new WaitForSeconds(0.5f);          // 等待 500ms
        while (true)
        {
            var result = mPlaybackManager.SetPlaybackDatasetUri((System.Uri)
null);
            if (result == PlaybackResult.ErrorPlaybackFailed || result ==
PlaybackResult.SessionNotReady)
            {
                yield return new WaitForSeconds(0.5f);  // 等待 500ms
            }
            else
            {
                break;
            }
        }
        mSession.enabled = true;
        DeBug.Log(" 结束回放会话 ");
    }
        // 获取文件大小
    private static string GetFileSize(string path)
    {
        var ByteCount = new FileInfo(path).Length;
```

```
        const long kiloBytes = 1024;
        const long megaBytes = kiloBytes * kiloBytes;

        if (ByteCount / megaBytes > 0)
        {
            var size = (float)ByteCount / megaBytes;
            return $"{size:F2} MB";
        }

        if (ByteCount / kiloBytes > 0)
        {
            var size = (float)ByteCount / kiloBytes;
            return $"{size:F2} KB";
        }
        return $"{ByteCount} Bytes";
    }
    #endregion
}
```

使用时，将该脚本挂载到场景中的 AR Session 对象上，并且将 AR Recording Manager 和 AR Playback Manager 组件也挂载到该对象上，使用按钮事件触发开始录制、结束录制、添加自定义数据、开始回放、结束回放 5 个相应的方法。可以看到会话的录制与回放，也可以正确录制和提取自定义数据[①]。

使用会话录制与回放功能可以更方便地调试 AR 应用，例如，可以录制保存几个不同的场景会话，利用这些场景会话，就可以调试 AR 应用在不同场景中的表现而不用亲自到实际场景中去测试。

3.8 即时放置

通常，用户在空间中放置 AR 虚拟对象时，需要先扫描周围环境，进行平面检测，然后通过手势放置虚拟对象，但这个过程并不非常直观，有时也会让用户感到困惑，通过即时放置（Instant Placement）功能允许用户无须移动设备，在 ARCore 建立运动跟踪之前放置虚拟物体，使用户更早地进入 AR 虚实融合氛围。在用户放置虚拟对象后，随着环境探索的进行，一旦 ARCore 建立运动跟踪，在能够确定 AR 虚拟对象正确姿态时，它就会更新虚拟对象的姿态和跟踪方法，从而自动完成过渡，如图 3-16 所示。

图 3-16 显示在 ARCore 中未检测到平面时，可以在现实世界桌子上放置一个姿态预估的虚拟物体模型，当 ARCore 正确检测到平面后，就可以确定更精确的虚拟模型姿态，并使用新的精确姿态进行对象跟踪。使用即时放置功能允许通过预先显示一个示意对象，减少用户焦虑，

① 为演示简单，代码使用了简单的方式，因此需要在同一个会话内完成会话录制与回放操作，实际使用时应当存储数据轨道 GUID 值。

提升 AR 使用体验。

图 3-16　即时放置效果示意图

在 AR Foundation 中使用 ARCore 即时放置功能需要使用持续射线检测（Persistent Raycast），持续射线是指该射线每一帧都会进行射线检测[①]，直到该射线被移除。在具体代码操作时，只需将持续射线添加到 AR Raycast Manager 组件便可以启用该持续射线并在每一帧进行更新，因此，实现即时放置功能的基本思路是设置好持续射线，预置一个放置位置，AR Raycast Manager 组件会持续地跟踪该射线的检测情况，在 ARCore 运动跟踪建立之前，使用预置的位置放置虚拟对象，而当检测到平面或者其他可跟踪对象时，该组件会自动进行姿态纠正，使用正确的姿态更新虚拟对象，代码如下：

```
//第 3 章 /3-13
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;

[RequireComponent(typeof(ARRaycastManager))]
public class InstancePlacement : MonoBehaviour
{
    private ARRaycastManager mRaycastManager;          //射线检测组件
    private readonly float estimateDistance = 1.2f;    //射线长度
    private bool placementPoseIsValid = false;         //是否可放置指示标志
    void Awake()
    {
```

① 射线检测更详细阐述可参见第 4 章。

```
        mRaycastManager = GetComponent<ARRaycastManager>();
    }
    void Update()
    {
        Touch touch;
        //无操作时返回
        if (Input.touchCount < 1 || (touch = Input.GetTouch(0)).phase !=
TouchPhase.Began)
        {
            return;
        }
        //单击在UI上时返回
        if (EventSystem.current.IsPointerOverGameObject(touch.fingerId))
        {
            return;
        }
        //已设置过示例对象时返回
        if (placementPoseIsValid)
            return;
        //添加持续射线
        var raycast = mRaycastManager.AddRaycast(touch.position,
estimateDistance);
        if (raycast != null)
        {
            placementPoseIsValid = true;
        }
    }
}
```

使用时，将 InstancePlacement 脚本挂载到场景中的 AR Session Origin 对象上，并同时在该对象上挂载 AR Plane Manager 组件和 AR Raycast Manager 组件，并为 AR Raycast Manager 组件的 Raycast Prefab 属性赋上需要放置的虚拟物体预制体。在 AR 应用启动后检测到平面之前，可以通过单击手势在场景中放置虚拟物体对象，而当应用检测到平面可放置点后，则会自动使用新的姿态更新虚拟物体。

通过以上方法可以实现即时放置功能，使用简单，但这种方式用两个问题：一是无法控制虚拟对象，如实现（如图 3-16 所示）预先展示示意物体，在检测到平面后放置真正的虚拟物体，也无法获取实例化后的虚拟对象；二是使用持续射线后，每一帧都进行射线检测，并且每一帧都会进行预制体的实例化操作，性能消耗极大，因此，改进思路是不通过 AR Raycast Manager 组件实例化虚拟物体，而是通过持续射线的 updated 事件进行示意模型与真正模型的转换操作，代码如下：

```
//第3章/3-14
using UnityEngine;
using UnityEngine.EventSystems;
```

```
using System.Collections.Generic;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;

[RequireComponent(typeof(ARRaycastManager))]
public class InstancePlacement : MonoBehaviour
{
    public GameObject mPreObjectPrefab;  // 在真实平面检测到之前用于示意的虚拟物体预制体
    public GameObject mObjectPrefab;                    // 真正需要放置的虚拟物体预制体

    private ARRaycastManager mRaycastManager;           // 组件对象
    private readonly float estimateDistance = 1.2f;  // 预先指定的放置示例物体距离
    private GameObject prePlacementObject;              // 实例化后的指示物体
    private bool placementPoseIsValid = false;         // 是否可放置指示器
    private ARRaycast mRaycast = null;                  // 构建的永久射线
    private Camera mCamera;                             // 场景渲染相机
    private static List<ARRaycastHit> Hits;             // 碰撞结果
    void Awake()
    {
        mRaycastManager = GetComponent<ARRaycastManager>();
        Hits = new List<ARRaycastHit>();
        mCamera = Camera.main;
    }
    void Update()
    {
        Touch touch;
        // 无操作时返回
        if (Input.touchCount < 1 || (touch = Input.GetTouch(0)).phase !=
TouchPhase.Began)
        {
            return;
        }
        // 单击在 UI 上时返回
        if (EventSystem.current.IsPointerOverGameObject(touch.fingerId))
        {
            return;
        }
        // 已设置过示例对象时返回
        if (placementPoseIsValid)
            return;
        // 添加持续射线，实例化示例对象
        mRaycast = mRaycastManager.AddRaycast(touch.position, estimateDistance);
        if (mRaycast != null)
        {
            placementPoseIsValid = true;
            prePlacementObject = Instantiate(mPreObjectPrefab, mRaycast.
transform.position, Quaternion.identity);
```

```
            mRaycast.updated += onRaycastUpdated;
        }
    }
    // 持续射线对象事件，每一帧都会触发
    private void onRaycastUpdated(ARRaycastUpdatedEventArgs obj)
    {
        DeBug.Log("跟踪状态 :" + mRaycast.trackingState);
        if (mRaycast.trackingState == TrackingState.Tracking)
        {
            Ray ray = new Ray(mCamera.transform.position, prePlacementObject.
transform.position);
            if (mRaycastManager.Raycast(ray, Hits, TrackableType.
PlaneWithinPolygon | TrackableType.PlaneWithinBounds))
            {
                var hitPose = Hits[0].pose;
                Instantiate(mObjectPrefab, hitPose.position, hitPose.rotation);
                mRaycastManager.RemoveRaycast(mRaycast);
                Destroy(prePlacementObject);
            }
        }
    }
}
```

在上述代码中，使用了两个预制体，一个用于在未能找到合适放置点之前预先展示，另一个是真正需要放置的虚拟物体。通过持续射线 updated 事件再进行一次射线检测，当检测到平面后，实例化真正的虚拟物体，同时移除了持续射线，销毁示例物体[①]，既提高了性能，也优化了视觉表现。

① 代码使用了射线与平面的碰撞检测，因此在该代码挂载的同一个场景对象上需要挂载 AR Plane Manager 组件。