第 5 章

2D 图像与 3D 物体检测跟踪

对 2D 图像进行检测与跟踪是 AR 技术应用最早的领域之一，利用设备摄像头获取的图像数据，通过计算机图像算法对图像中的特定 2D 图像部分进行检测识别与姿态跟踪，并利用 2D 图像的姿态叠加虚拟物体对象，这种方法也称为基于标识的 AR（Marker Based AR），是 AR 早期最主要的形式，经过多年的发展，2D 图像检测跟踪标识物已经从特定的 Marker 发展到普通图像。对 3D 物体检测跟踪则是对真实环境中的三维物体而不是 2D 图像进行检测识别跟踪，其利用人工智能技术实时对环境中的 3D 物体进行检测并评估姿态，相比 2D 图像，3D 物体检测识别跟踪对设备软硬件要求高得多。本章主要学习利用 AR Foundation 检测识别跟踪 2D 图像与 3D 物体的方法。

5.1 2D 图像检测跟踪

2D 图像检测跟踪首先需要准备一张被检测目标物图像（下文称目标图像），实施时通过设备摄像头采集的图像对目标图像进行检测和姿态评估（Pose Estimation），并确定其在图像中的位置，然后以目标图像中心为原点建立坐标系，称为模板坐标系（Marker Coordinate）。在建立这个坐标系之后就可以利用其渲染虚拟物体，从而实现 AR 效果，因此，在 AR 中，2D 图像检测跟踪技术是指通过计算机图像处理技术对设备摄像头中拍摄到的 2D 图像进行检测识别定位，并对其姿态进行跟踪的技术。

5.1.1 图像检测跟踪基本操作

图像检测跟踪技术的基础是图像识别，图像识别是指识别和检测出数字图像或视频中对象或特征的技术，图像识别技术是信息时代的一门重要技术，也是其他众多计算机图像视觉处理技术的基础。

在 AR Foundation 中，图像检测跟踪系统依据参考图像库中的图像信息尝试在设备摄像头捕获的图像中检测匹配 2D 图像并跟踪，在 AR Foundation 的图像跟踪处理中，一些特定的术语如表 5-1 所示。

表 5-1　2D 图像检测识别跟踪术语

术　　语	描 述 说 明
参考图像（Reference Image）	识别 2D 图像的过程实际上是一个特征值对比的过程，AR Foundation 将从设备摄像头中获取的图像信息与参考图像库中的图像特征值信息进行对比，存储在参考图像库中的用于对比的图像就叫作参考图像。一旦对比成功，真实环境中的图像将与参考图像库的参考图像建立对应关系，每个真实 2D 图像的姿态信息也一并被检测
参考图像库（Reference Image Library）	参考图像库用来存储一系列的参考图像，用于对比，每个图像跟踪程序都必须有一个参考图像库，但需要注意的是，参考图像库中存储的实际是参考图像的特征值信息而不是原始图像，这有助于提高对比速度和稳健性。参考图像库越大，图像对比就会越慢，建议参考图像库的图像数量不要超过 1000 张，通常应该控制在 100 张以内
跟踪组件提供方（Provider）	AR Foundation 架构在底层 SDK 图像跟踪 API 之上，也就是说 AR Foundation 并不具体负责图像识别算法，它只提供一个接口，具体图像检测识别由算法提供方提供

ARKit 图像检测识别主要技术指标如表 5-2 所示。

表 5-2　ARKit 图像检测识别主要技术指标

序号	描　　述
1	每个参考图像库可以存储最多几百张参考图像的特征点信息
2	ARKit 可以在环境中同时跟踪多张图像，但无法跟踪同一图像的多个实例
3	环境中的物理图像必须至少为 15cm×15cm 且必须平坦（如不能起皱或卷绕在瓶子上）
4	在物理图像被跟踪后，ARKit 会提供对位置、方向和物理大小的估计值，随着 ARKit 收集的数据增多，这些估计值会被持续优化
5	ARKit 可以跟踪移动中的图像
6	所有跟踪都在设备上完成，无须网络连接，支持在设备端或通过网络更新参考图像及参考图像库，无须重新安装应用

在 AR Foundation 中，使用 2D 图像检测跟踪基本操作分成两步：第 1 步是建立一个参考图像库；第 2 步是在场景中挂载 AR Tracked Image Manager 组件，并将一个需要实例化的预制体赋给其 Tracked Image Prefab 属性。

按照上述步骤，下面演示使用 2D 图像检测跟踪功能的基本流程。在 Unity 工程中，首先建立一个参考图像库，在工程窗口（Project 窗口）中的 ImageLib 文件夹下右击并依次选择 Create→XR→Reference Image Library 创建一个参考图像库，命名为 RefImageLib，如图 5-1 所示。

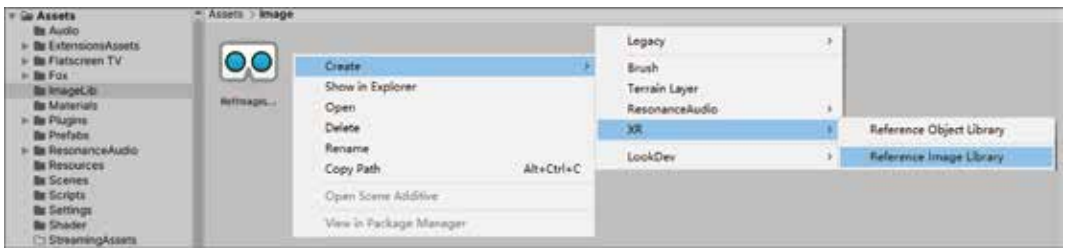


图 5-1　新建参考图像库

选择新建的 RefImageLib 参考图像库，在属性窗口（Inspector 窗口）中，单击 Add Image 按钮添加参考图像，将参考图像拖动到左侧图像框中[①]，如图 5-2 所示。

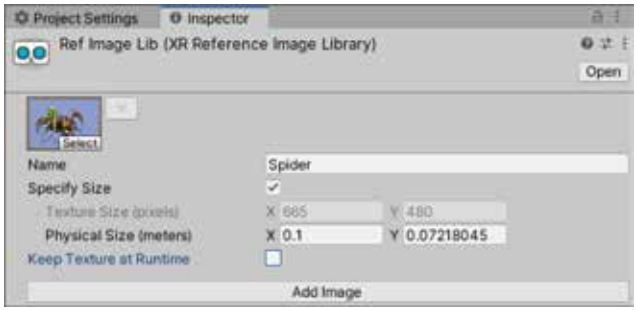


图 5-2　添加参考图像

在图 5-2 中，每个参考图像除了图像信息外还有若干其他属性，其具体含义如表 5-3 所示。

表 5-3　参考图像属性术语

术　语	描 述 说 明
Name	一个标识参考图像的名字，这个名字在进行图像对比时没有作用，但在比对匹配成功后可以通过参考图像名字获知是哪个参考图像，参考图像名字可以重复，因为在跟踪时，跟踪系统还会给每个参考图像一个 referenceImage.guid 值，利用这个 GUID 值唯一标识每个参考图像
Specify Size	为加速图像检测识别过程，ARKit 要求提供一个 2D 待检测图像的真实物理尺寸，所以如果要设置，这个值一定会是一个大于 0 的长和宽值对，当一个值发生变化时，Unity 会根据参考图像的长宽比例自动调整另一个值
Keep Texture at Runtime	一个默认的纹理，这个纹理可以用于修改预制体模型的外观

完成上述工作之后，在层级窗口中选择 XR Origin 游戏对象，并为其挂载 AR Tracked Image Manager 组件，将第 1 步制作的 RefImageLib 参考图像库拖动到其 Serialized Library 属性中，并设置相应的需要在检测到 2D 图像后实例化的预制体，如图 5-3 所示。

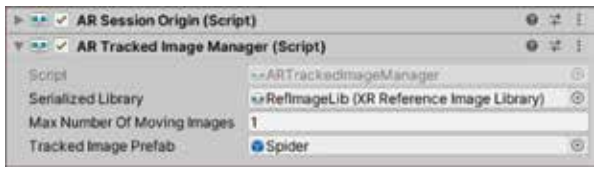


图 5-3　挂载图像检测跟踪组件

参考图像库也可以在运行时动态设置，不过 AR Tracked Image Manager 组件一旦启动图像检测跟踪，参考图像库就必须有设定值，不能为 null，即要求在 AR Tracked Image Manager

① 参考图像被添加到图像框中后，后台会进行参考图像的特征信息提取，实质上参考图像库中存储的是参考图像的特征信息。

组件 Enable 之前设置参考图像库，后文会详细阐述。Max Number of Moving Images 属性指定了最大可跟踪的动态图像的数量，这里的动态图像是指被跟踪图像可以旋转、平移等，即被跟踪图像姿态可以发生变化。动态图像跟踪是一个非常消耗 CPU 性能的任务，过多的动态图像跟踪会导致应用性能下降。

编译运行，将设备摄像头对准需要检测的 2D 图像，检测识别效果如图 5-4 所示。

图 5-4　图像检测跟踪效果图

提示

如果将 Tracked Image Prefab 属性设置为一个播放视频的预制体，则在检测到 2D 图像时，AR 应用就可以播放相应视频，并且 AR Foundation 会自动调整视频尺寸和姿态以适应 2D 图像大小尺寸与姿态，在检测到的 2D 图像上播放视频这一功能在一些场景中很有用，例如数字名片。视频播放功能可参见第 12 章。

AR Foundation 使用了 3 种状态表示当前图像检测跟踪状态，各状态如表 5-4 所示，可以在应用运行时获取图像跟踪状态信息。

表 5-4　2D 图像跟踪状态

状　态	描 述 说 明
None	图像还未被跟踪，这也可能是第 1 次检测到图像时的状态
Limited	图像已经被跟踪，但当前属于跟踪受限状态。在以下两种情况时图像跟踪会进入受限状态：①图像在当前帧中不可见；②当前图像被检测到，但并未被跟踪，如跟踪图像数量超过了 maxNumberOfMovingImages 属性设定的值
Tracking	图像正在被跟踪中

除获取图像跟踪状态信息，我们也能获取当前图像检测跟踪的其他信息，典型代码如下：

```
//第 5 章 /5-1
using UnityEngine;
```

```
using UnityEngine.XR.AR Foundation;

[RequireComponent(typeof(ARTrackedImageManager))]
public class TrackedImageInfoManager : MonoBehaviour
{
    ARTrackedImageManager mTrackedImageManager;

    void Awake()
    {
        mTrackedImageManager = GetComponent<ARTrackedImageManager>();
    }
    //注册图像检测事件
    void OnEnable()
    {
        mTrackedImageManager.trackedImagesChanged += OnTrackedImagesChanged;
    }
    //取消图像检测事件注册
    void OnDisable()
    {
        mTrackedImageManager.trackedImagesChanged -= OnTrackedImagesChanged;
    }
    //输出图像跟踪信息
    void UpdateInfo(ARTrackedImage trackedImage)
    {
        Debug.Log(string.Format(
            "参考图像名：{0}，图像跟踪状态：{1}，GUID 值：{2}，参考图像尺寸：{3} cm，
实际图像尺寸：{4} cm，空间位置：{5}",
            trackedImage.referenceImage.name,
            trackedImage.trackingState,
            trackedImage.referenceImage.guid,
            trackedImage.referenceImage.size * 100f,
            trackedImage.size * 100f,
            trackedImage.transform.position.ToString()));
    }
    //图像检测跟踪状态处理方法
    void OnTrackedImagesChanged(ARTrackedImagesChangedEventArgs eventArgs)
    {
        foreach (var trackedImage in eventArgs.added)
        {
            //设置默认图像尺寸比例，检测到的图像的单位是厘米，需要缩放
            trackedImage.transform.localScale = new Vector3(0.01f, 1f, 0.01f);
            UpdateInfo(trackedImage);
        }
        foreach (var trackedImage in eventArgs.updated)
            UpdateInfo(trackedImage);
    }
}
```

通过分析上述代码输出，可以了解到图像检测跟踪状态的变化情况，当被检测图像移出视野后，图像跟踪状态并不是 None 状态，而是 Limited 状态，当该图像再次被检测到时，其跟踪状态会再次变为 Tracking，但 GUID 值保持不变。

5.1.2　图像检测跟踪功能的启用与禁用

在 AR Foundation 中，实例化生成的虚拟对象并不会随着被跟踪 2D 图像的消失而消失（在实例化虚拟对象后移走 2D 图像，虚拟对象并不会被销毁，而是保持在最后一次 2D 图像被检测到的空间位置），虚拟对象停留在原来的位置上，在某些应用场景下是合理的，但在另一些场景下也会变得很不合适；另外，图像检测跟踪是一个非常消耗性能的操作，在不使用图像检测跟踪时一般应当把图像检测跟踪功能关闭。与平面检测功能的启用与禁用一样，也可以通过脚本代码来控制图像检测跟踪功能的启用与禁用及所跟踪对象的显示与隐藏，典型代码如下：

```
// 第 5 章 /5-2
public Text mToggleImageDetectionText;                    // 显示当前显隐状态
private ARTrackedImageManager mARTrackedImageManager;
void Awake()
{
    mARTrackedImageManager = GetComponent<ARTrackedImageManager>();
}
// 启用与禁用图像跟踪
public void ToggleImageTracking()
{
    mARTrackedImageManager.enabled = !mARTrackedImageManager.enabled;

    string ImageDetectionMessage = "";
    if (mARTrackedImageManager.enabled)
    {
        ImageDetectionMessage = " 禁用图像跟踪 ";
        SetAllImagesActive(true);
    }
    else
    {
        ImageDetectionMessage = " 启用图像跟踪 ";
        SetAllImagesActive(false);
    }

    if (mToggleImageDetectionText != null)
        mToggleImageDetectionText.text = ImageDetectionMessage;
 }

// 显示或者隐藏所有已实例化的虚拟对象
void SetAllImagesActive(bool value)
```

```
    {
        foreach (var img in mARTrackedImageManager.trackables)
            img.gameObject.SetActive(value);
    }
```

与平面检测功能的启用与禁用类似，可以使用一个按钮进行状态切换，从而在运行时控制图像检测跟踪功能的启用与禁用。

5.1.3 多图像检测跟踪

在 AR Tracked Image Manager 组件中，Tracked Image Prefab 属性指定了需要实例化的虚拟对象。在默认情况下，AR Foundation 支持多图像跟踪，通过在 RefImageLib 参考图像库中添加多张参考图像，在这些参考图像被检测到时，每个被检测到的参考图像都会生成一个虚拟对象，如图 5-5 所示。

图 5-5 AR Foundation 默认支持多图像跟踪

但在 AR 应用运行时，只能有一个 AR Tracked Image Manager 组件运行（多个 AR Tracked Image Manager 组件同时运行会导致跟踪冲突），因此只能设置一个 Tracked Image Prefab 属性，即不能实例化多种虚拟对象，这将极大地限制图像跟踪的实际应用，所以为了实例化多种虚拟对象，我们只能动态地修改 Tracked Image Prefab 属性。经过测试，我们发现在 AR Foundation 中，AR Tracked Image Manager 组件在 trackedImagesChanged 事件触发之前就已经实例化了虚拟对象，因此无法通过在 trackedImagesChanged 事件中修改 Tracked Image Prefab 属性达到实时改变需要实例化的虚拟对象的目的。

鉴于此，我们的解决思路是：在 AR Tracked Image Manager 组件 Tracked Image Prefab 中设置第 1 个需要实例化的虚拟对象预制体，然后在 trackedImagesChanged 事件中捕捉到图像 added 操作时，将 Tracked Image Prefab 属性更改为下一个需要实例化的虚拟对象预制体，达到动态调整虚拟对象的目的。如正常将 Tracked Image Prefab 设置为 Spider 预制体，在检测到 Spider 图像后，将 Tracked Image Prefab 修改为 Cat 预制体，在检测到 Cat 图像后就会实例化 Cat 预制体了。新建一个 C# 脚本，命名为 MultiImageTracking，代码如下：

```
//第 5 章 /5-3
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.AR Foundation;

public class MultiImageTracking : MonoBehaviour
{
    ARTrackedImageManager mImgTrackedmanager;
    public GameObject[] ObjectPrefabs;        //虚拟对象预制体数组

     private void Awake()
     {
         mImgTrackedmanager = GetComponent<ARTrackedImageManager>();
     }

     //注册图像跟踪事件
     private void OnEnable()
     {
         mImgTrackedmanager.trackedImagesChanged += OnTrackedImagesChanged;
     }
     //取消图像跟踪事件注册
     void OnDisable()
     {
         mImgTrackedmanager.trackedImagesChanged -= OnTrackedImagesChanged;
     }
     //图像跟踪状态改变处理方法
     void OnTrackedImagesChanged(ARTrackedImagesChangedEventArgs eventArgs)
     {
         foreach (var trackedImage in eventArgs.added)
         {
             OnImagesChanged(trackedImage.referenceImage.name);
         }
        //foreach (var trackedImage in eventArgs.updated)
        //{
        //OnImagesChanged(trackedImage.referenceImage.name);
        //}
     }
     //动态设置 trackedImagePrefab 属性
     private void OnImagesChanged(string referenceImageName)
     {
         if (referenceImageName == "Spider")
         {
             mImgTrackedmanager.trackedImagePrefab = ObjectPrefabs[1];
             Debug.Log("Tracked Name is .." + referenceImageName);
             Debug.Log("Prefab Name is .." + mImgTrackedmanager.trackedImagePrefab.
name);
```

```
            }
            if (referenceImageName == "Cat")
            {
                mImgTrackedmanager.trackedImagePrefab = ObjectPrefabs[0];
            }
        }
    }
```

编译运行，先扫描检测识别 Spider 图像，待实例化 Spider 虚拟对象后再扫描检测 Cat 图像，这时就会实例化 Cat 虚拟对象了，效果如图 5-6 所示。

图 5-6　实例化多种虚拟对象

但这种方式其实有个很大的弊端，即必须按顺序检测图像，因为我们无法在用户检测图像之前预测用户可能会扫描检测的 2D 图像。为解决这个问题，就不能使用 AR Tracked Image Manager 组件实例化对象了，而需由我们自己负责虚拟对象的实例化。将 AR Tracked Image Manager 组件下的 Tracked Image Prefab 属性置空，为 MultiImageTracking 脚本的 ObjectPrefabs 数组赋上相应的预制体对象，对代码进行修改，修改后的代码如下：

```
//第 5 章 /5-4
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.AR Foundation;

public class MultiImageTracking : MonoBehaviour
{
   ARTrackedImageManager mImgTrackedmanager;
   public GameObject[] ObjectPrefabs;          //虚拟对象预制体数组

    private void Awake()
    {
        mImgTrackedmanager = GetComponent<ARTrackedImageManager>();
    }
    //注册图像跟踪事件
    private void OnEnable()
```

```
        {
            mImgTrackedmanager.trackedImagesChanged += OnTrackedImagesChanged;
        }
        // 取消图像跟踪事件注册
        void OnDisable()
        {
            mImgTrackedmanager.trackedImagesChanged -= OnTrackedImagesChanged;
        }
        void OnTrackedImagesChanged(ARTrackedImagesChangedEventArgs eventArgs)
        {
            foreach (var trackedImage in eventArgs.added)
            {
                OnImagesChanged(trackedImage);
            }
           //foreach (var trackedImage in eventArgs.updated)
           //{
           //OnImagesChanged(trackedImage.referenceImage.name);
           //}
        }
        // 根据检测到的参考图像名确定需要实例化的虚拟对象
        private void OnImagesChanged(ARTrackedImage referenceImage)
        {
            if (referenceImage.referenceImage.name == "Spider")
            {
                Instantiate(ObjectPrefabs[0], referenceImage.transform);
            }
            if (referenceImage.referenceImage.name == "Cat")
            {
                Instantiate(ObjectPrefabs[1], referenceImage.transform);
            }
        }
    }
```

因为 Tracked Image Prefab 属性为空，AR Tracked Image Manager 组件不会实例化任何虚拟对象，需要我们自己负责虚拟对象的实例化。在上述代码中，我们在 trackedImagesChanged 事件中捕捉到图像 added 操作，根据参考图像的名字在被检测图像的位置实例化虚拟对象，达到了预期目的，不再要求用户按顺序扫描图像。但现在还面临一个问题，我们不可能使用 if-else 或者 switch-case 语句遍历所有可用的模型，因此可修改为动态加载模型的方式，代码如下：

```
    // 第 5 章 /5-5
    using System.Collections;
    using System.Collections.Generic;
    using UnityEngine;
    using UnityEngine.XR.AR Foundation;

    public class MultiImageTracking : MonoBehaviour
```

```
    {
        ARTrackedImageManager mImgTrackedManager;
        private Dictionary<string, GameObject> mPrefabs =  new Dictionary<string,
GameObject>();                  //虚拟对象预制体字典

        private void Awake()
        {
            mImgTrackedManager = GetComponent<ARTrackedImageManager>();
        }

        void Start()
        {
            mPrefabs.Add("Cat", Resources.Load("Cat") as GameObject);
            mPrefabs.Add("Spider", Resources.Load("Spider") as GameObject);
        }
        //注册图像跟踪事件
        private void OnEnable()
        {
            mImgTrackedManager.trackedImagesChanged += OnTrackedImagesChanged;
        }
        //取消图像跟踪事件注册
        void OnDisable()
        {
            mImgTrackedManager.trackedImagesChanged -= OnTrackedImagesChanged;
        }
        void OnTrackedImagesChanged(ARTrackedImagesChangedEventArgs eventArgs)
        {
            foreach (var trackedImage in eventArgs.added)
            {
                OnImagesChanged(trackedImage);
            }
           //foreach (var trackedImage in eventArgs.updated)
           //{
           //OnImagesChanged(trackedImage.referenceImage.name);
           //}
        }
        //根据检测到的参考图像名实例化虚拟对象
        private void OnImagesChanged(ARTrackedImage referenceImage)
        {
            Debug.Log(" 参考图像名 :"+ referenceImage.referenceImage.name);
            Instantiate(mPrefabs[referenceImage.referenceImage.name], referenceImage
.transform);
        }
    }
```

为了实现代码所描述的功能，还要完成两项工作：第 1 项工作是将虚拟对象预制体放置到 Resources 文件夹中方便动态加载；第 2 项工作是保证 mPrefabs 字典中的 key 值与 RefImageLib

参考图像库中的参考图像名一致。至此，我们已实现自由的多图像多模型功能。

但仔细思考，其实还有问题没有解决：①前文提到过，参考图像名在同一个参考图像库中是允许重名的，利用参考图像名为字典 key 值就只能有一个虚拟对象可以被使用，开发人员需要确保参考图像名的唯一性；②现在每个参考图像对应一个虚拟对象，这是在开发时就已经确定的，无法在运行时动态切换虚拟对象。

解决第 1 个问题可以利用每个参考图像的 GUID 值，这个值是在添加参考图像时由算法生成的，可以确保唯一性，并且每个参考图像对应一个 GUID 值，在图像跟踪丢失后再次被跟踪时，这个 GUID 值也不会发生变化，可以利用这个值作为字典的 key 值。

解决第 2 个问题也可以利用被检测图像的 GUID 值，在检测到图像实例化后保存该虚拟对象，然后在需要时另外实例化其他虚拟对象并替换原虚拟对象。

以上两个问题解决方案的实现并不复杂，读者可自行实现。

5.1.4　运行时创建参考图像库

参考图像库可以在开发时创建并设置好，这种方式简单方便，但使用时不太灵活，有时无法满足应用需求，实际上，AR Foundation 支持在运行时动态地创建参考图像库。从前面的学习我们知道，AR Tracked Image Manager 组件在启动时其参考图像库必须不为 null，否则该组件不会启动，因此，在使用图像检测识别跟踪功能时，必须确保在 AR Tracked Image Manager 组件启用之前设置好参考图像库，所以在使用动态图像库时，一般会在需要时动态地添加 AR Tracked Image Manager 组件，而不是在开发时预先添加该组件。动态添加 AR Tracked Image Manager 组件及参考图像库的典型代码如下（后文将实际演示）：

```
    //第 5 章 /5-6
    mTrackImageManager = gameObject.AddComponent<ARTrackedImageManager>();
    XRReferenceImageLibrary runtimeImageLibrary;
    //创建参考图像库
    mTrackImageManager.referenceLibrary = mTrackImageManager.CreateRuntimeLibrary
(runtimeImageLibrary);
    mTrackImageManager.requestedMaxNumberOfMovingImages = 2;
    mTrackImageManager.trackedImagePrefab = mPrefabOnTrack;
    mTrackImageManager.enabled = true;
```

在添加完 AR Tracked Image Manager 组件并设置好相应参考图像库及其他属性后，需要显式地设置 mTrackImageManager.enabled = true，以便启用该组件。

参考图像库可以是 XRReferenceImageLibrary 或者 RuntimeReferenceImageLibrary 类型，XRReferenceImageLibrary 类型可以在 Unity 编辑器中创建，但不能在运行时修改，不过在运行时 XRReferenceImageLibrary 类型会自动转换成 RuntimeReferenceImageLibrary 类型。在使用时，也可以直接从 XRReferenceImageLibrary 创建 RuntimeReferenceImageLibrary，典型代码如下：

```
    // 第 5 章 /5-7
    XRReferenceImageLibrary serializedLibrary = new XRReferenceImageLibrary();
    RuntimeReferenceImageLibrary runtimeLibrary = mTrackImageManager.CreateRuntimeLibrary
(serializedLibrary);
```

在从 XRReferenceImageLibrary 类型向 RuntimeReferenceImageLibrary 类型转换时，参考图像的顺序并不确定，即无法预先确定参考图像的次序，但参考图像名称及其 GUID 值不会发生变化。

5.1.5 运行时切换参考图像库

在 AR 应用运行时，也可以动态地切换参考图像库，下面演示运行时如何动态地切换参考图像库。首先制作好两个参考图像库 ReferenceImageLibrary_1 和 ReferenceImageLibrary_2，然后使用两个按钮事件在运行时动态地切换所使用的参考图像库。新建一个 C# 脚本，命名为 ChangeImageLib，代码如下：

```
    // 第 5 章 /5-8
    using System;
    using UnityEngine;
    using UnityEngine.XR.AR Foundation;
    using UnityEngine.XR.ARSubsystems;
    using UnityEngine.UI;

    public class ChangeImageLib : MonoBehaviour
    {
        [SerializeField]
        private Button BtnFirst, BtnSecond;      //UI 按钮
        [SerializeField]
        private GameObject mPrefabOnTrack;       // 虚拟对象预制体

        [SerializeField]
        XRReferenceImageLibrary[] mReferenceImageLibrary;
        private int currentSelectedLibrary = 0;
        private ARTrackedImageManager mTrackImageManager;
        void Start()
        {
            mTrackImageManager= gameObject.AddComponent<ARTrackedImageManager>();
            mTrackImageManager.referenceLibrary = mTrackImageManager.CreateRuntimeLibrary
(mReferenceImageLibrary[0]);
            mTrackImageManager.requestedMaxNumberOfMovingImages = 3;
            mTrackImageManager.trackedImagePrefab = mPrefabOnTrack;
            mTrackImageManager.enabled = true;
            BtnFirst.onClick.AddListener(() => SetReferenceImageLibrary(0));
            BtnSecond.onClick.AddListener(() => SetReferenceImageLibrary(1));
            Debug.Log(" 初始化完成 !");
```

```
        }
        // 动态切换参考图像库
        public void SetReferenceImageLibrary(int selectedLibrary = 0)
        {
            mTrackImageManager.referenceLibrary = mTrackImageManager.CreateRuntimeLibrary
(mReferenceImageLibrary[selectedLibrary]);
            Debug.Log(String.Format(" 切换参考图像库 {0} 成功 !", selectedLibrary));
        }
    }
```

在层级窗口中选择 XR Origin 对象，将该脚本挂载到此对象上，并设置好相关属性，将创建并设置好的 ReferenceImageLibrary_1 和 ReferenceImageLibrary_2 参考图像库赋给 mReferenceImageLibrary 对象，如图 5-7 所示。

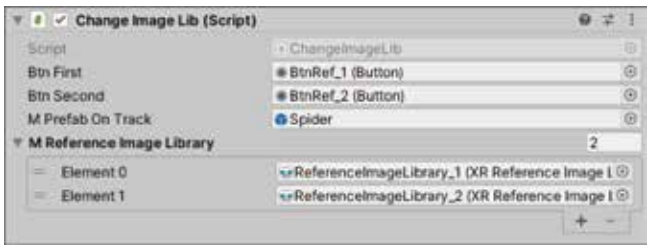


图 5-7　设置参考图像库及其他属性

在本演示中，ReferenceImageLibrary_1 和 ReferenceImageLibrary_2 参考图像库中分别添加了不同的参考图像，通过切换参考图像库，AR 应用会及时地做出反应，对新切换参考图像库中的参考图像进行检测识别跟踪，效果如图 5-8 所示。

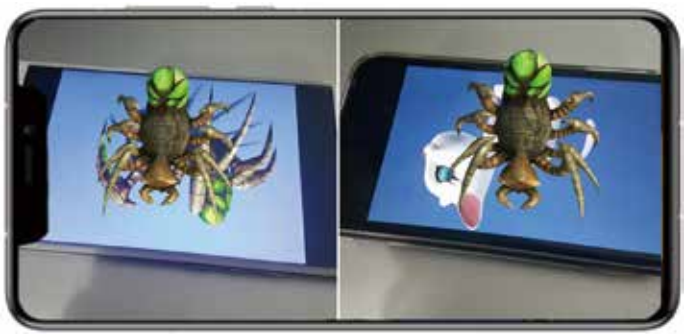


图 5-8　AR 应用能及时对新切换参考图像库中的图像进行检测识别

5.1.6　运行时添加参考图像

ARKit 支持在运行时动态地将新的参考图像添加到参考图像库中，这时 RuntimeReferenceImageLibrary 类型实际上就是一个 MutableRuntimeReferenceImageLibrary 类型，在具体使用时，

也需要将 RuntimeReferenceImageLibrary 转换为 MutableRuntimeReferenceImageLibrary 类型再使用。如果需要在运行时创建新的参考图像库，则可以使用无参数的 CreateRuntimeLibrary() 方法，然后将其转换为 MutableRuntimeReferenceImageLibrary 类型以供使用，典型代码如下：

```
// 第 5 章 /5-9
var library = trackedImageManager.CreateRuntimeLibrary();
if (library is MutableRuntimeReferenceImageLibrary mutableLibrary)
{
     // 将图像添加到参考图像库
}
```

因为需要提取参考图像的特征值信息，运行时动态地添加参考图像是一个计算密集型任务，这需要花费几十毫秒时间，需要很多帧才能完成添加，所以为了防止同步操作造成应用卡顿，可以利用 Unity Job 系统异步处理这种操作。

添加参考图像时，需要使用 ScheduleAddImageWithValidationJob() 方法将参考图像添加到 MutableRuntimeReferenceImageLibrary 库中，该方法可以是同步的，也可以是异步的，取决于开发者的使用方式，该方法的原型如下：

```
// 第 5 章 /5-10
public static AddReferenceImageJobState ScheduleAddImageWithValidationJob
(
this MutableRuntimeReferenceImageLibrary library,      // 参考图像库
Texture2D texture,                                     // 需要添加的参考图像
string name,                                           // 参考图像名称
float? widthInMeters,                                  // 待检测的图像物理尺寸宽（单位：米）
JobHandle inputDeps = default                          // 输入描述信息
);
```

可以通过该方法一次性地将单张或者多张参考图像添加到参考图像库，即使当前的参考图像库正在跟踪图像、处于使用中也没关系。

动态添加参考图像对图像有一些特定要求，第 1 个要求是参考图像可读写，因为添加图像时需要提取图像特征值信息；第 2 个要求是图像格式必须为应用平台上支持的格式，通常选择 RGB24 或者 RGBA32 格式。这需要在图像的导入设置（Import Setting）中设置[①]，如图 5-9 所示[②]。

经过测试，动态添加参考图像对图像编码格式也有要求，通常只支持 JPG、PNG 格式，但一些 JPG、PNG 类型图像的编码格式也不支持，如 ETC_RGB4、Alpha8 等，需要仔细挑选作为参考图像的图像文件。

---

① 在 Unity 工程窗口中选择作为参考图像的图像，然后在属性窗口中会打开导入设置面板，展开 Advanced 卷展栏，进行高级设置。

② 如果作为动态添加的参考图像没有启用 Read/Write Enabled 功能，则编译时将提示 The texture must be readable to be used as the source for a reference image；如果所选定的格式平台不支持，则编译时将提示 The texture format ETC_RGB4 is not supported by the current image tracking subsystem。

图 5-9　设置图片相应属性

下面演示如何在运行时动态地添加参考图像，新建一个 C# 脚本，命名为 DynamicImage-Tracking，代码如下：

```
// 第 5 章 /5-11
using Unity.Collections;
using UnityEngine;
using UnityEngine.XR.AR Foundation;
using UnityEngine.XR.ARSubsystems;
using UnityEngine.UI;
using Unity.Jobs;

public class DynamicImageTrackingAsync : MonoBehaviour
{
    // 用于异步图像操作的结构体
    struct DeallocateJob : IJob
    {
        [DeallocateOnJobCompletion]
        public NativeArray<Byte> data;
        public void Execute() { }
    }

    [SerializeField]
    private Button BtnAddImage;                                    //UI 按钮

    [SerializeField]
    private GameObject mPrefabOnTrack;                             // 虚拟对象预制体
    private Vector2 scaleFactor = new Vector3(0.3f, 0.3f);         // 参考图像尺寸
    private XRReferenceImageLibrary runtimeImageLibrary;           // 运行时参考图像库
    private ARTrackedImageManager mTrackImageManager;

    [SerializeField]
    private Texture2D mAddedImage;

    void Start()
    {
        mTrackImageManager = gameObject.AddComponent<ARTrackedImageManager>();
```

```
            // 创建参考图像库
            mTrackImageManager.referenceLibrary = mTrackImageManager.CreateRuntimeLibrary
(runtimeImageLibrary);
            mTrackImageManager.requestedMaxNumberOfMovingImages = 2;
            mTrackImageManager.trackedImagePrefab = mPrefabOnTrack;
            mTrackImageManager.enabled = true;
            mTrackImageManager.trackedImagesChanged += OnTrackedImagesChanged;
            BtnAddImage.onClick.AddListener(() => AddImageJobAsync(mAddedImage));
        }
        // 取消图像检测事件注册
        void OnDisable()
        {
            mTrackImageManager.trackedImagesChanged -= OnTrackedImagesChanged;
        }
        // 添加参考图像同步
        public void AddImageJob(Texture2D texture2D)
        {

            try
            {
                MutableRuntimeReferenceImageLibrary mutableRuntimeReferenceImageLibrary =
mTrackImageManager.referenceLibrary as MutableRuntimeReferenceImageLibrary;
                mutableRuntimeReferenceImageLibrary.ScheduleAddImageWithValidationJob(
                    texture2D,                          // 参考图像
                    "Spider",                           // 参考图像名
                    scaleFactor.x);                     // 参考图像的宽
            }
            catch (System.Exception e)
            {
                Debug.Log(" 出现错误 :"+e.Message);
            }
        }


        // 异步添加参考图像
        void AddImageJobAsync(Texture2D refImage)
        {
            Byte[] colorBuffer = refImage.GetRawTextureData();
            NativeArray<Byte> image = new NativeArray<Byte>(colorBuffer,  Allocator.
TempJob);
            if (mTrackImageManager.referenceLibrary is MutableRuntimeReferenceImageLibrary
mutableLibrary)
            {
                var referenceImage = new XRReferenceImage(
                    SerializableGuid.empty,          //GUID 值
                    SerializableGuid.empty,          // 默认纹理
                    scaleFactor,                     // 物理尺寸设置
```

```
                "Spider",                          // 参考图像名称
                null);                             // 参考图像，这里先不设置

            var jobState = mutableLibrary.ScheduleAddImageWithValidationJob(
                image,
                new Vector2Int(refImage.width, refImage.height),
                TextureFormat.ARGB32,              // 设置图像格式
                referenceImage);

            // 启动一个任务，在任务结束后销毁相关资源
            new DeallocateJob { data = image }.Schedule(jobState.jobHandle);
        }
        else
        {
            // 销毁图像资源
            image.Dispose();
        }
    }

    // 图像检测事件
    void OnTrackedImagesChanged(ARTrackedImagesChangedEventArgs eventArgs)
    {
        foreach (ARTrackedImage trackedImage in eventArgs.added)
        {
            Debug.Log(" 检测到图像 :"+trackedImage.name);
            trackedImage.transform.Rotate(Vector3.up, 180);
        }
        foreach (ARTrackedImage trackedImage in eventArgs.updated)
        {
            trackedImage.transform.Rotate(Vector3.up, 180);
        }
    }
}
```

上述代码演示了同步和异步两种添加参考图像的方式，在层级窗口中选择 XR Origin 对象，将该脚本挂载在此对象上，并设置好相关属性，即可在运行时动态地添加参考图像了。动态切换参考图像库与动态添加参考图像是非常实用的功能，可以根据不同的应用场景切换到不同的参考图像库，或者添加新的参考图像而无须重新编译或者中断应用。

5.1.7　脱卡

从前文我们知道，在使用 ARKit 进行 2D 图像检测跟踪时，当检测识别到参考图像时，会实例化虚拟对象并叠加到物理图像之上，这样便能实时跟踪物理图像的移动、旋转。当物理图像移出视野范围（屏幕）后，实例化的虚拟对象也会跟随之移出视野（或者随 2D 图像移

动到屏幕边缘外），这在很多应用场景下是合适的。

但有时，我们也希望当物理图像移出视野后，虚拟对象能停留在屏幕中心，并能正常与用户进行交互，例如在博物馆门口通过检测识别特定图像后实例化虚拟导游，希望该虚拟导游能伴随整个游览过程，进行不间断的语音讲解。当物理图像消失后，实例化后的虚拟模型不随之消失，这个功能称为图像检测识别脱卡。

脱卡可以有很多种实现，例如当物理图像消失后，启用另一个渲染相机渲染实例化后的模型，例如通过当前渲染相机的前向向量在设备屏幕前的一定距离放置实例化模型等。

但在进行这个操作之前，需要定义物理图像消失事件，即什么情况下可认为物理图像消失。AR Tracked Image Manager 组件的 trackedImagesChanged 事件会提供这个检测识别结果的状态变化，结合前文知识，我们只需检查跟踪图像的状态值，当跟踪状态从 Tracking 变更到 Limited 时，就可以认为物理图像消失这个事件已经发生，但 AR Foundation 官方文档指出，可能有多种未知情况会导致图像检测跟踪状态从 Tracking 变更到 Limited，所以以此作为判断依据并不严谨。

但考虑到 2D 图像检测识别后，通常会实例化虚拟对象，虚拟对象会追踪 2D 图像的位置，当物理图像移出视野后，虚拟对象也会移出视野，利用这个特性，可以通过虚拟对象的可见性判断物理图像是否移出视野。典型代码如下：

```
// 第 5 章 /5-12
using UnityEngine;

public class ObjectTrackLost : MonoBehaviour
{
    private bool isTrackLost = false;    // 可见与不可见标识
    private Camera mCamera;              // 主相机
    private float distance = 0.5f;       // 放置在主相机前方的距离
    void Start()
    {
        mCamera = Camera.main;
    }
    // 当虚拟对象可见时触发
    public void OnBecameVisible()
    {
        Debug.Log(" 跟踪正常 ");
        //isTrackLost = false;
    }

    // 当虚拟对象不可见时触发
    public void OnBecameInvisible()
    {
        Debug.Log(" 跟踪丢失 ");
        isTrackLost = true;
    }
```

```
        void Update()
        {
            if(isTrackLost)
            {
                // 将虚拟对象放置于主相机前一定距离
                gameObject.transform.position = mCamera.transform.position +
mCamera.transform.forward.normalized * distance;
                gameObject.transform.rotation = mCamera.transform.rotation;
            }
        }
        // 当 2D 图像被跟踪时由外部调用，终止脱卡
        public void SetVisible()
        {
            isTrackLost = false;
        }
    }
```

在上述代码中，我们利用 OnBecameInvisible() 方法判断虚拟对象的可见性，然后在虚拟对象不可见（物理图像被移出）时，将虚拟对象放置在主相机前方指定位置。该代码需要挂载于虚拟对象上，并且虚拟对象必须有 Renderer 组件（对象一定要可被渲染）。需要注意的是，不能再使用 OnBecameVisible() 方法恢复图像跟踪，而必须由外部代码在图像被再次跟踪时调用 SetVisible() 方法恢复跟踪。

5.1.8　图像检测跟踪优化

在 2D 图像被检测到之后，ARKit 会跟踪该图像的姿态（位置与方向），因此，可以实现虚拟对象与 2D 图像绑定的效果（虚拟元素的姿态会随 2D 图像的姿态发生变化），如在一张别墅的图片上加载一个虚拟的别墅模型，旋转、移动该别墅图片，虚拟别墅模型也会跟着旋转或者移动，就像虚拟模型粘贴在图片上一样。

图像检测跟踪效果与很多因素相关，为了更好地在应用中使用 2D 图像检测跟踪，提高用户体验，应当注意以下事项。

1. 有关参考图像

参考图像应当具有丰富纹理、高对比度、纹理不重复等特征，特征丰富的参考图像有利于 ARKit 进行图像检测，具体注意事项如表 5-5 所示。

表 5-5　参考图像一般注意事项

序号	描　　述
1	参考图像支持 PNG 和 JPG 文件格式，对于 JPG 文件，为了获得最佳性能，需避免过度压缩
2	参考图像特征提取仅基于高对比度的点，所以彩色和黑白图像都会被检测到，对物理图像颜色不敏感

续表

序号	描　　述
3	参考图像的分辨率至少应为 300 × 300 像素
4	使用高分辨率的参考图像不会提升性能
5	避免使用特征稀疏、无纹理的参考图像
6	避免使用具有重复特征、重复纹理的参考图像

在参考图像的选择上，肉眼很难分辨是否是高质量的参考图像，参考图像的选择有一定的技巧，包含高度重复特征的图像很容易导致特征点误匹配，因此建议采用细节丰富、纹理不重复的图像作为参考图像[①]。

2. 有关参考图像库创建

ARKit 参考图像库会存储参考图像的特征值信息，每张参考图像会占据大约 6KB 空间。在运行时将一张参考图像添加到参考图像库大约需要 30ms，因此在运行时添加参考图像需要使用异步操作。另外，不要在参考图像库中存储不使用的参考图像，因为这会对应用性能产生一定的影响。

3. 有关跟踪优化

在 ARKit 初次识别图像时，物理图像大约需要占据设备摄像头图像面积的 40% 或以上，需要及时提示用户将物理图像放置在摄像头取景范围内并保持合适大小。

一般而言，为了更好地优化 2D 图像检测跟踪，需要从参考图像的选择、参考图像库的设计、整体设计方面进行考虑。

（1）尽量指定待检测图像的预期物理尺寸。此元数据可以提升检测识别性能，特别是对于较大的物理图像（长和宽超过 75cm），ARKit 会使用这些物理尺寸评估 2D 图像到用户设备的距离，不正确的物理尺寸会影响检测跟踪精度，从而影响加载的虚拟模型与 2D 图像的贴合度。

（2）物理图像尽量展平。卷曲的图像，如包裹酒瓶的海报非常不利于 ARKit 检测或者导致检测出的位姿不正确。

（3）确保需要检测的物理图像照明条件良好。光线昏暗或者反光（如玻璃橱窗里的海报）会影响图像检测。

（4）通常情况下，每个参考图像库的参考图像数量不应该超过 25 个。如果数量过多，则会影响检测准确性和检测速度。一般情况下，可以将大型的参考图像库拆分为小的参考图像库，然后根据 AR 应用运行时的条件动态地切换参考图像库。

① 建议在创建参考图像库时通过 Xcode 对参考图像进行验证，只有通过 Xcode 验证的图像才可作为参考图像添加到参考图像库中，具体验证方式可查阅 ARKit 官方文档。

5.2　3D 物体检测跟踪

3D 物体检测跟踪技术，是指通过计算机图像处理和人工智能技术对设备摄像头拍摄到的 3D 物体检测识别定位并对其姿态进行跟踪的技术。3D 物体检测跟踪技术的基础也是图像检测识别，但比前述 2D 图像检测识别跟踪要复杂得多，原因在于现实世界中的物体是三维的，从不同角度看到的物体形状、纹理都不一样，在进行图像特征值对比时需要的数据和计算比 2D 图像要大得多。

在 ARKit 中，3D 物体检测识别跟踪通过预先记录 3D 物体的空间特征信息，在真实环境中寻找对应的 3D 真实物体对象，并对其姿态进行跟踪。与 2D 图像检测跟踪类似，在 AR Foundation 中实现 3D 物体检测跟踪也需要一个参考物体库，这个参考物体库中的每个对象都是一个 3D 物体的空间特征信息。获取参考物体空间特征信息可以通过扫描真实 3D 物体采集其特征信息，生成 .arobject 参考物体空间特征信息文件。.arobject 文件只包括参考物体的空间特征信息，而不是参考物体的数字模型，也不能用该文件复原参考物体三维结构。参考物体空间特征信息对快速、准确检测识别 3D 物体起着关键作用。

5.2.1　获取参考物体空间特征信息

苹果公司提供了一个获取物体空间特征信息的扫描工具，但扫描工具是一个 Xcode 工程源码，需要自己编译，源码名为 Apple's Object Scanner App，读者可自行下载并使用 Xcode 编译[①]。该工具的主要功能是扫描真实世界中的物体并导出 .arobject 文件，该文件可作为 3D 物体检测识别的参考物体。

使用扫描工具进行扫描的过程其实是对物体表面 3D 特征值信息与空间位置信息的采集过程，这是一个计算密集型的工作，为确保扫描过程流畅、高效，建议使用高性能的 iOS 设备。当然扫描工作可以在任何支持 ARKit 的设备上进行，但高性能 iOS 设备可以更好地完成这一任务。

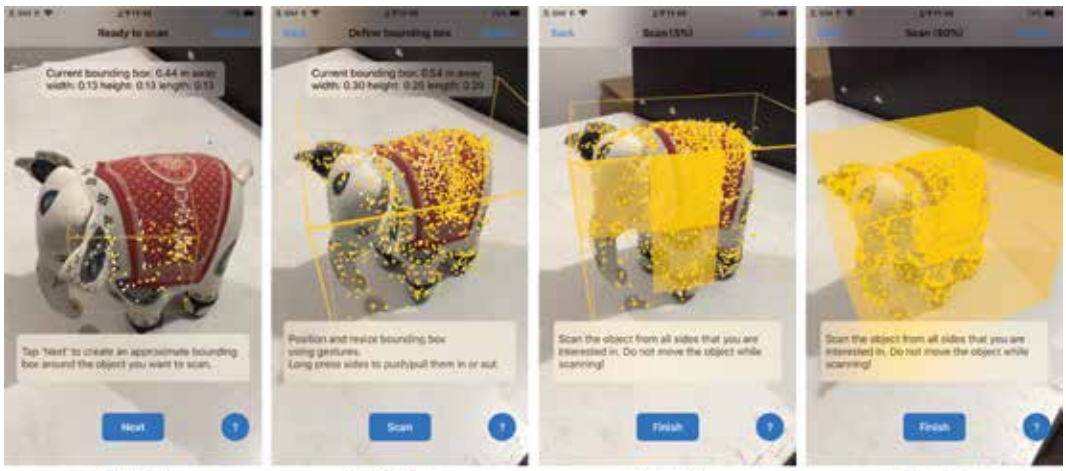
参考物体空间特征信息对后续 3D 物体检测识别速度、准确性有直接影响，因此，正确地扫描生成 .arobject 文件非常重要，遵循下述步骤操作可以提高扫描成功率。下面，将引导大家一步一步完成这个扫描过程。

（1）将需要扫描的物体放置在一个背景干净不反光的平整面上（如桌面、地面），运行扫描工具，将被扫描物体放置在摄像头正中间位置，在扫描工具检测到物体时会出现一个空心长方体（包围盒），移动手机，将长方体大致放置在物体的正中间位置，如图 5-10（a）所示，屏幕上也会提示包围盒的相关信息，但这时包围盒可能与实际物体尺寸不匹配，单击 Next 按钮可调整包围盒大小。

（2）扫描工具只采集包围盒内的物体空间特征信息，因此，包围盒大小对采集信息的完

① 下载地址为 https://developer.apple.com/documentation/arkit/scanning_and_detecting_3d_objects。

整性非常关键。围绕着被扫描物体移动手机，扫描工具会尝试自动调整包围盒的大小。如果自动调整结果不是很理想，则可以手动进行调整，方法是长按长方体的一个面，当这个面出现延长线时拖动该面即可移动，长方体的 6 个面都可以采用类似方法进行调整。包围盒不要过大或过小，如果过小，则采集不到完整的物体空间特征信息；如果过大，则可能会采集到周围环境中的物体信息，不利于快速检测识别 3D 物体。调整好后单击 Scan 按钮开始对物体空间特征信息进行采集，如图 5-10（b）所示。



(a) 第(1)步　(b) 第(2)步　(c) 第(3)步　(d) 第(4)步

图 5-10　扫描采集 3D 参考物体空间特征信息（一）

（3）在开始扫描物体后，扫描工具会给出可视化的信息采集提示，将采集成功的区域用淡黄色标识出来，引导用户完成全部信息采集工作，如图 5-10（c）所示。

（4）缓慢移动手机（保持被扫描物体不动），从不同角度扫描物体，确保包围盒的所有面都被成功扫描（通常底面不需要扫描，只需扫描前、后、左、右、上 5 个面）。如图 5-10（d）所示，扫描工具会在所有面的信息采集完后自动进入下一步，或者可以在采集完所有信息后手动单击 Finish 按钮进入下一步。如果在未完整采集到所需信息时单击 Finish 按钮，则会提示采集信息不足，如图 5-11 所示。

（5）在采集完物体空间特征信息后，扫描工具会在物体上显示一个 XYZ 的三维彩色坐标轴，如图 5-12（a）所示。这个坐标轴的原点表示这个物体的原点（这个原点代表的就是模型局部坐标系原点），可以通过拖动三个坐标轴边上的小圆球调整坐标轴的原点位置。在图 5-12（a）中可以看到 Load Model 按钮，单击该按钮可以加载一个 USDZ 格式模型文件，加载完后会在三维坐标轴原点显示该模型，就像是在真实环境中检测到 3D 物体并加载数字模型一样。通过加载模型可以直观地看到数字模型与真实三维物体之间的位置关系，如果位置不合适，则可以重复步骤（5）调整三维坐标轴原点位置，直到加载后的数字模型与真实三维物体位置关系达到预期要求。

（6）在调整好坐标系原点后可以对采集的空间特征信息进行测试验证，单击 Test 按钮进行

测试，如图 5-12（b）所示。将被扫描物体放置到不同的环境、不同的光照条件下，使用设备摄像头从不同的角度查看该物体，看能否正确地检测出物体的位置及姿态。如果验证时出现无法检测识别的问题，则说明信息采集不太完整或有问题，需要重新扫描一次；如果验证无问题，则可导出使用。单击 Share 按钮导出该单个物体采集的空间特征信息 .arobject 格式文件，也可以单击左上角的 Merge Scans 合并多个物体空间特征信息文件，如图 5-12（c）所示。合并可以是合并之前采集后导出的 .arobject 文件，也可以开始新的物体扫描，合并两次扫描结果。

（7）单击 Share 按钮后该扫描工具会将采集的物体空间特征信息导出为 .arobject 文件，在打开的导出对话框中，可以选择不同的导出方式，可以保存到云盘，也可以通过邮件、微信等媒介发送给他人，如图 5-12（d）所示，还可以通过 AirDrop（隔空投送）的方式直接投送到 Mac 计算机或其他 iOS 设备上。在使用 AirDrop 投送到 Mac 计算机上时，只需在计算机上打开 Finder，选择“隔空投送”，接收来自移动设备发送的文件便可（需要打开 Mac 计算机的蓝牙 / 网络并完成设备配对），如图 5-13 所示，接收的文件存储在下载文件夹中，后缀为 .arobject。

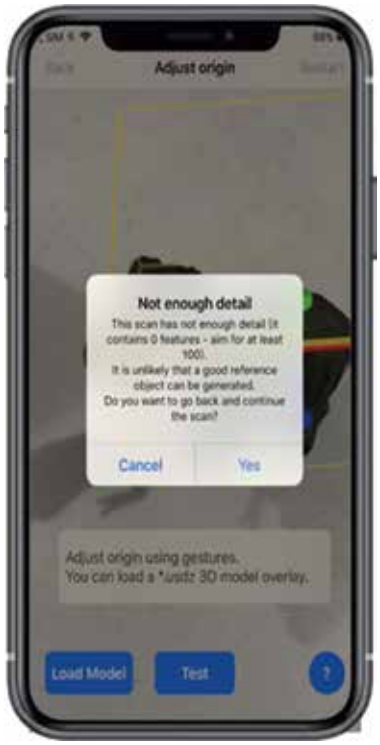
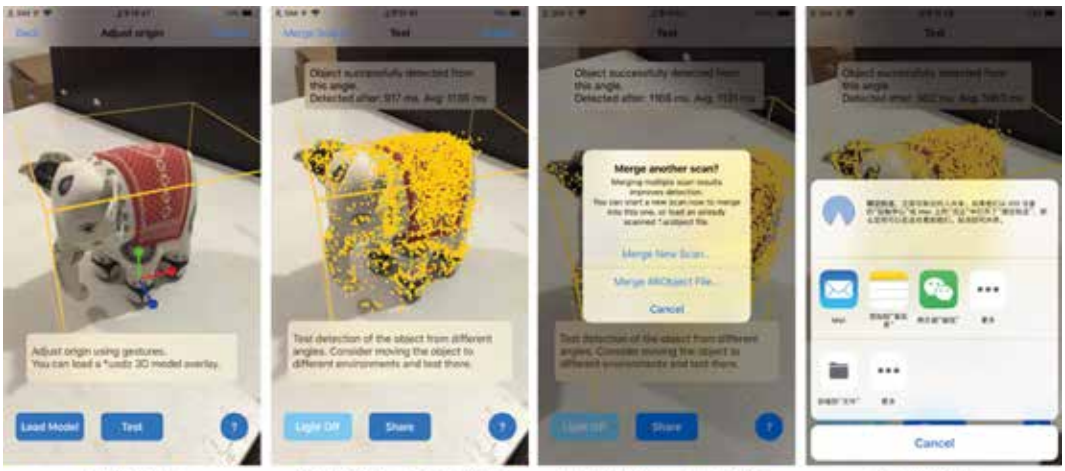


图 5-11　未能采集到足够信息的提示



(a) 第(5)步　(b) 第(6)步：测试验证　(c) 第(7)步：导出文件　(d) 第(8)步

图 5-12　扫描采集 3D 参考物体空间特征信息（二）

在得到参考物体空间特征信息文件（.arobject 文件）后，就可以将其用于后续的 3D 物体检测识别中了。

图 5-13　采用“隔空投送”方式将采集的 .arobject 文件发送到 Mac 计算机上

5.2.2　扫描获取物体空间特征信息的注意事项

如前所述，参考物体空间特征信息对 3D 物体检测识别的速度、准确性有非常大的影响，因此，在扫描获取参考物体空间特征信息时，遵循以下原则可大大地提高参考物体空间特征信息的可用性及保真度。

1. 扫描环境

（1）确保扫描时的照明条件良好、被扫描物体有足够的光照，通常要在 250 ～ 400lm，良好照明有利于采集物体特征值信息。

（2）使用白光照明，避免暖色或冷色灯光照明。

（3）背景干净，最好是无反光、非粗糙的中灰色背景，干净的背景有利于分离被扫描物体与周边环境。

2. 被扫描物体

（1）将被扫描物体放置在摄像机镜头正中间，最好与周边物体分开一段距离。

（2）被扫描物体最好有丰富的纹理细节，无纹理、弱纹理、反光物体不利于特征值信息提取。

（3）被扫描物体大小适中，不过大或过小。ARKit 扫描或检测识别 3D 物体时对可放在桌面的中等尺寸物体进行过特殊优化。

（4）被扫描物体最好是刚体，不会在扫描或检测识别时发生变形、折叠、扭曲等影响特征值和空间信息的形变。

（5）扫描时的环境光照与检测识别时的环境光照信息一致时效果最佳，应防止扫描与检测识别时光照差异过大。

（6）在扫描物体时应逐面缓慢扫描，不要大幅度快速移动手机。

在获取参考物体的空间特征信息 .arobject 文件后就可以将其作为参考物体进行真实环境 3D 物体的检测识别跟踪了。虽然 3D 物体检测识别跟踪在技术上与 2D 图像检测识别跟踪有非常大的差异，但在 AR Foundation 中，3D 物体检测识别跟踪与 2D 图像检测识别跟踪在使用界面、操作步骤上几乎完全一致，极大地方便了开发者使用。

5.2.3　AR Tracked Object Manager 组件

在 AR Foundation 中，3D 物体检测识别跟踪系统依据参考物体库中的参考物体空间特征信息尝试在设备摄像头拍摄的图像中检测匹配 3D 物体并跟踪，与 2D 图像检测识别跟踪类似，3D 物体检测识别跟踪也有一些特定的术语，如表 5-6 所示。

表 5-6　3D 物体检测识别跟踪术语

术　　语	描 述 说 明
参考物体（Reference Object）	检测识别 3D 物体的过程也是一个特征值对比的过程，AR Foundation 将从摄像头中获取的图像信息与参考物体库中的参考物体空间特征值信息进行对比，存储在参考物体库中的用于对比的物体空间特征信息叫作参考物体（物体空间特征信息并不是数字模型，也不能据此恢复出 3D 物体）。一旦对比成功，真实环境中的 3D 物体将与参考物体库中的参考物体建立对应关系，每个真实 3D 物体的姿态信息也一并被检测
参考物体库（Reference Object Library）	参考物体库用于存储一系列的参考物体空间特征信息，此信息用于对比，每个 3D 物体跟踪程序都必须有一个参考物体库，但需要注意的是，参考物体库中存储的实际是参考物体的空间特征值信息而不是原始 3D 物体网格信息，这有助于提高对比速度与稳健性。参考物体库越大，3D 物体检测对比就会越慢，相比与 2D 图像检测识别，3D 物体检测识别需要比对的数据量更大、计算也更密集，因此，在同等条件下，参考物体库中可容纳的参考物体数量要比 2D 图像库中的参考图像数量少得多
跟踪组件提供方（Provider）	AR Foundation 架构在底层 SDK 3D 物体检测识别跟踪 API 之上，也就是说 AR Foundation 并不具体负责 3D 物体检测识别过程的算法，它只提供一个接口，具体 3D 物体检测识别由算法提供方提供
AR 物体锚点（ARObjectAnchor）	记录真实世界中被检测识别的 3D 物体位置与姿态的锚点，该锚点由 AR 会话在检测识别到 3D 物体后自动添加到每个被检测到的对象上。通过该锚点，可以将虚拟物体对象渲染到指定的空间位置上

在 AR Foundation 中，3D 物体属于可跟踪对象，由 AR Tracked Object Manager 组件进行统一管理，该组件通常挂载在 XR Origin 对象上，其有 Reference Library 和 Tracked Object Prefab 两个属性，如图 5-14 所示。

图 5-14　AR Tracked Object Manager 组件

AR Tracked Object Manager 组件负责对 3D 物体的检测识别和跟踪进行管理，并可以在已检测到的 3D 物体位置渲染虚拟对象，该组件依据参考物体库中的参考物体空间特征信息不断尝试在环境中检测 3D 物体，因此，只有预置在参考物体库中的 3D 物体才可能被检测到。

1. Reference Library

参考物体库，AR Foundation 检测 3D 物体的依据，可以在开发时静态地设置也可以在运行时动态地添加，但只要 AR Tracked Object Manager 组件开始启动 3D 物体检测跟踪，参考物体库就不能为 null。

2. Tracked Object Prefab

在检测到 3D 物体后需要被实例化的预制体，在实例化时，AR Foundation 会确保每个实例化后的对象都挂载一个 AR Tracked Object 组件，如果预制体没有挂载该组件，则 AR Tracked Object Manager 组件会负责自动为其挂载一个，也可以在运行时通过代码获取该实例化对象。

5.2.4 3D 物体检测识别跟踪基本操作

在 AR Foundation 中，3D 物体检测跟踪与 2D 图像检测跟踪操作步骤基本一致，分为两步：第 1 步是建立一个参考物体库；第 2 步是在场景中挂载 AR Tracked Object Manager 组件，并将一个需要实例化的预制体赋给其 Tracked Object Prefab 属性。下面我们来具体操作。

按上述步骤，首先建立一个参考物体库，右击工程窗口中的 ObjectLib 文件夹，依次选择 Create → XR → Reference Object Library 新建一个参考物体库，并命名为 RefObjectLib，如图 5-15 所示。

图 5-15 新建一个参考物体库

选择新建的 RefObjectLib 参考物体库，在属性窗口中，单击 Add Reference Object 按钮添加参考物体，将 5.2.1 节中导出的 .arobject 文件拖到工程中，并将其拖动到 Reference Object Assets 属性框中，如图 5-16 所示。

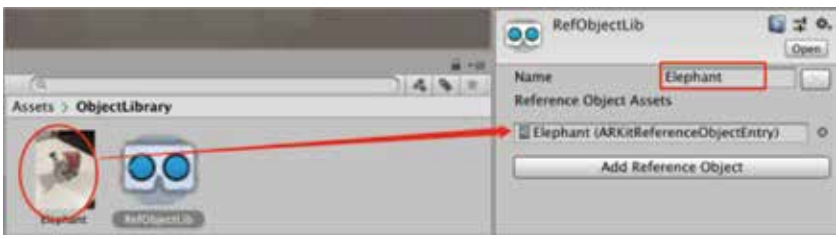


图 5-16 将参考物体空间特征信息文件添加到参考物体库中

每个参考物体都有一个 Name 属性，该属性用于标识参考物体，这个名字在做 3D 物体检测对比时没有作用，但在比对匹配成功后可以通过参考物体名字获知是哪个参考物体。参考物体名字可以重复，因为在添加参考物体时，跟踪系统还会为每个参考物体生成一个 GUID 值，这个 GUID 值可用于唯一标识一个参考物体。

完成上述工作之后，在层级窗口中选择 XR Origin 对象，并为其挂载 AR Tracked Object Manager 组件，将第 1 步制作的 RefObjectLib 参考物体库拖动到其 Reference Library 属性中，并设置好需要实例化的预制体，如图 5-17 所示。

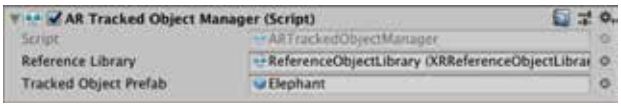


图 5-17　设置 AR Tracked Object Manager 组件属性

编译运行，效果如图 5-18 所示。

图 5-18　3D 物体检测识别跟踪效果图

3D 物体检测识别跟踪技术比 2D 图像检测识别跟踪技术要复杂得多，但 AR Foundation 对这两种技术在使用方式上进行了统一，提供给开发人员完全一致的使用界面，方便了应用开发。

5.2.5　3D 物体检测跟踪启用与禁用

在 AR Foundation 中，与 2D 图像检测跟踪一样，实例化出来的虚拟对象并不会随着被跟踪物体的消失而消失，而是会继续停留在原来的位置上，并且 3D 物体检测跟踪比 2D 图像检测跟踪消耗资源更多，在不需要时或者使用后应当关闭 3D 物体检测跟踪功能，参考 2D 图像检测识别跟踪功能的启用与禁用，类似地，可以控制 3D 物体检测跟踪功能的启用与禁用及加载的虚拟对象的显示和隐藏，代码如下：

```
// 第 5 章 /5-13
using System.Collections;
```

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.AR Foundation;
using UnityEngine.UI;

[RequireComponent(typeof(ARTrackedObjectManager))]
public class AppController : MonoBehaviour
{

    public Text m_ToggleObjectdDetectionText;
    private ARTrackedObjectManager mARTrackedObjectManager;
    void Awake()
    {
        mARTrackedObjectManager = GetComponent<ARTrackedObjectManager>();
    }

    #region 启用与禁用物体跟踪
    public void ToggleObjectTracking()
    {
        mARTrackedObjectManager.enabled = !mARTrackedObjectManager.enabled;

        string ObjectDetectionMessage = "";
        if (mARTrackedObjectManager.enabled)
        {
            ObjectDetectionMessage = "禁用物体跟踪";
            SetAllObjectsActive(true);
        }
        else
        {
            ObjectDetectionMessage = "启用物体跟踪";
            SetAllObjectsActive(false);
        }

        if (m_ToggleObjectdDetectionText != null)
            m_ToggleObjectdDetectionText.text = ObjectDetectionMessage;
    }

    void SetAllObjectsActive(bool value)
    {
        foreach (var obj in mARTrackedObjectManager.trackables)
            obj.gameObject.SetActive(value);
    }
    #endregion
}
```

在使用时，将该脚本挂载在 XR Origin 对象上，并使用一个按钮事件控制检测跟踪功能的启用与禁用，运行效果如图 5-19 所示。

图 5-19　3D 物体检测跟踪功能启用与禁用示意图

5.2.6　多物体检测识别跟踪

与 2D 图像检测跟踪相似，在 AR Tracked Object Manager 组件中，有一个 Tracked Object Prefab 属性，这个属性即为需要实例化的虚拟对象。默认 ARKit 支持多 3D 物体检测跟踪，即 ARKit 会在每个检测识别到的 3D 物体上实例化一个虚拟对象，如图 5-20 所示。

图 5-20　AR Foundation 默认支持多 3D 物体识别跟踪

为解决多参考物体多虚拟对象的问题，需要自己负责虚拟对象的实例化。首先将 AR Tracked Object Manager 组件下的 Tracked Object Prefab 属性置空，然后新建一个 C# 脚本文件，命名为 MultiObjectTracking，并编写代码如下：

```
// 第 5 章 /5-14
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.AR Foundation;

[RequireComponent(typeof(ARTrackedObjectManager))]
public class MultiObjectTracking : MonoBehaviour
```

```
    {
        ARTrackedObjectManager ObjTrackedManager;
        private Dictionary<string, GameObject> mPrefabs =  new Dictionary<string,
GameObject>();

        private void Awake()
        {
            ObjTrackedManager = GetComponent<ARTrackedObjectManager>();
        }
        void Start()
        {
            mPrefabs.Add("Book", Resources.Load("Book") as GameObject);
            mPrefabs.Add("Elephant", Resources.Load("Elephant") as GameObject);
        }
        private void OnEnable()
        {
            ObjTrackedManager.trackedObjectsChanged += OnTrackedObjectsChanged;
        }
        void OnDisable()
        {
            ObjTrackedManager.trackedObjectsChanged -= OnTrackedObjectsChanged;
        }
        void OnTrackedObjectsChanged(ARTrackedObjectsChangedEventArgs eventArgs)
        {
            foreach (var trackedObject in eventArgs.added)
            {
                OnImagesChanged(trackedObject);
            }
           //foreach (var trackedImage in eventArgs.updated)
           //{
           //OnImagesChanged(trackedImage.referenceImage.name);
           //}
        }

        private void OnImagesChanged(ARTrackedObject refObject)
        {
            Debug.Log(" 参考物体名 :"+ refObject.referenceObject.name);
            Instantiate(mPrefabs[refObject.referenceObject.name], refObject.transform);
        }
    }
```

该脚本从 Resources 文件夹下动态地加载虚拟模型，并根据检测识别到的参考物体名称实例化不同的虚拟对象。将该脚本挂载在 XR Origin 对象上，为确保代码正确运行，我们还要完成两项工作：第 1 项工作是将需要实例化的预制体放置到 Resources 文件夹中方便动态加载；第 2 项工作是确保脚本中 mPrefabs 字典的 key 值与 RefObjectLib 参考物体库中的参考物体名称一致，如图 5-21 所示。

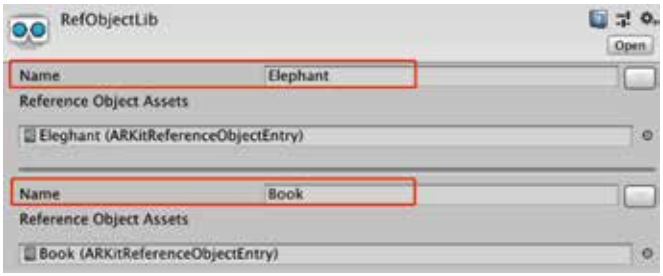


图 5-21　参考物体名称应与 mPrefabs 字典中的 key 值对应

至此，我们已实现自由的多参考物体多虚拟对象功能，编译运行，扫描检测 3D 物体，AR 应用会根据 3D 物体的不同加载不同的虚拟对象，效果如图 5-22 所示。

图 5-22　实例化多个虚拟对象

可以看到，3D 物体检测识别跟踪与 2D 图像检测识别跟踪使用方式极为相似，AR Foundation 屏蔽了这两者在底层实现上的巨大差异，提供了相同的使用界面，除了 3D 物体空间特征信息获取之外，其他使用方法遵循一样的流程和步骤，降低了开发者的使用难度。

现实世界原本就是三维的，3D 物体检测识别跟踪更符合人类认识事物的规律，因此在很多领域都有着广阔的应用前景，如博物馆文物展示，利用 3D 物体检测识别功能，就可以实现对静态展品的信息动态化，实现关联的模型动画、视频播放等，扩展对展品的背景知识、使用功能、内部结构、工作原理等演示。