第 5 章

图像检测跟踪

对二维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二维图像部分进行检测识别与姿态跟踪，并利用二维图像的姿态叠加虚拟物体对象，这种方法称为 Marker Based AR（基于标识的 AR），是 AR 早期最主要的形式，经过多年的发展，二维图像检测跟踪标识物已经从特定的 Marker 发展到普通图像。本章主要学习利用 AR Foundation 检测识别与跟踪二维图像的方法。

5.1 图像检测跟踪基本操作

在 AR 中，图像跟踪技术是指通过图像处理技术对设备摄像头中拍摄到的二维图像进行检测识别定位，并对其姿态进行跟踪的技术。图像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，其产生的目的是为了让计算机代替人类去处理大量的图形图像及真实物体信息，是众多其他技术的基础。

在 AR Foundation 中，图像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二维图像并跟踪之，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　图像跟踪术语表

术　　语	描 述 说 明
参考图像 （Reference Image）	识别二维图像的过程实际上是一个特征值比对的过程，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，每个真实二维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

续表

术　　语	描 述 说 明
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

ARCore 图像检测识别主要技术指标如表 5-2 所示。

表 5-2　ARCore 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 1000 张参考图像的特征点信息
2	ARCore 可以在环境中同时跟踪最多 20 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，ARCore 会提供对位置、方向和物理大小的估计值，随着 ARCore 收集的数据增多，这些估计值会持续优化
5	ARCore 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，使用二维图像检测跟踪时，基本操作分成两步：第一步是建立一个参考图像库；第二步是在场景中挂载 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，下面演示使用二维图像检测跟踪的基本流程。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。

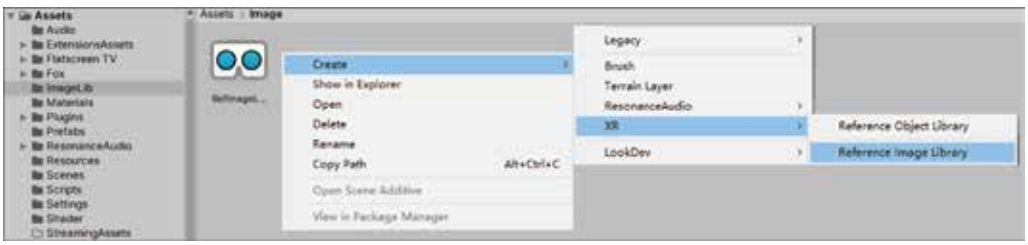

图 5-1　新建参考图像库

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。

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。

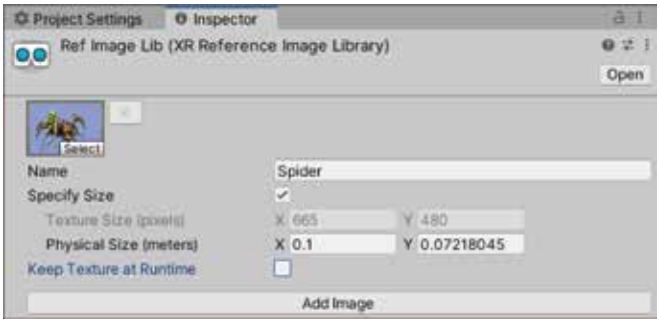

图 5-2　添加参考图像

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　参考图像属性术语表

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	这是个可选值，为加速图像检测识别过程，一些底层 SDK 要求提供一个二维待检测图像的真实物理尺寸，所以如果要设置，则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 AR Session Origin 游戏对象，并为其挂载 AR Tracked Image Manager 组件，将第一步制作的 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二维图像后实例化的预制体，如图 5-3 所示。

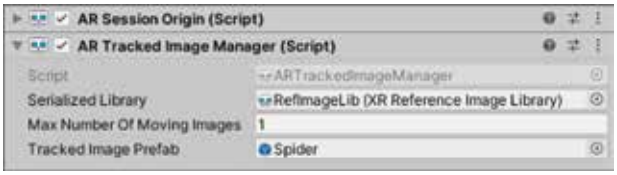

图 5-3　挂载图像检测跟踪组件

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 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二维图像，跟踪效果如图 5-4 所示。仔细观察实例化后的虚拟蜘蛛对象，会发现模型在 *Y* 轴上旋转了 180°，这是由于 ARCore 所采用的坐标系与 Unity 采用的坐标系不同造成的，为解决这个问题，在 ARCore 中实例化虚拟对象时需要将其绕自身 *Y* 轴再旋转 180°，但在 AR Foundation 中，实例化虚拟对象是由 AR Tracked Image Manager 自动完成的，无法干预，所以目前使用这种方式实例化虚拟对象时，需要先把模型在 *Y* 轴上进行 180° 旋转再使用[①]。

提示

如果将 Tracked Image Prefab 属性设置为一个播放视频的预制体，在检测到二维图像时，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二维图像大小与姿态，在检测到的二维图像上播放视频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 11 章。

图 5-4　图像识别跟踪效果图

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　二维图像跟踪状态

术　语	描 述 说 明
None	图像还未被跟踪，这也可能是第一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，但并未跟踪，如跟踪图像数量超过了 maxNumberOfMovingImages 属性设定值
Tracking	图像正在跟踪中

① 将模型在 *Y* 轴上旋转 180° 有两种办法：①如果模型是自己制作的，或者熟悉常见三维建模软件，可以在三维建模软件中调整模型的旋转再导出；②在 Unity 层级窗口中新创建一个空对象，将模型作为其子对象，在 *Y* 轴上旋转模型，然后连同空对象一起保存为新的预制体使用。

除获取图像跟踪状态信息外，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.ARFoundation;

[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 值会保持不变。

5.2　图像跟踪功能启用与禁用

在 AR Foundation 中，实例化出来的虚拟对象并不会随着被跟踪二维图像的消失而消失（在实例化虚拟对象后取走二维图像，虚拟对象并不会被销毁，而是保持在最后一次二维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，而且，图像跟踪是一个非常消耗性能的操作，在不使用图像跟踪时一般应当把图像跟踪功能关闭。参考平面检测功能的启用与禁用，也可以通过脚本代码来控制图像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跟踪功能的启用与禁用。

5.3 多图像跟踪

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 组件会导致跟踪冲突），这样就只能设置一个 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，只能动态地修改 Tracked Image Prefab 属性。经过测试，会发现在 ARFoundation 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，因此这样就无法通过在 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之目的。

鉴于此，可采用的解决思路是：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，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，这样来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，这样，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，编写代码如下：

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

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　实例化多个虚拟对象示意图

但这种方式其实有一个很大的弊端，即必须按顺序检测图像，因为无法在用户检测图像之前预测用户可能会扫描检测的二维图像。为解决这个问题，就不能使用 AR Tracked Image Manager 组件实例化对象，而由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，并修改代码，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.ARFoundation;

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，所以需要自己负责虚拟对象的实例化。在上述代码中，会在 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，实现了目的，不再要求用户按顺序扫描图像，但现在还面临一个问题，即不可能使用 if-else 或者 switch-case 语句遍历所有可用的模型，因此应修改为动态加载模型的方式，编写代码如下：

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

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("Image name:"+ 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：第一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；第二项工作是保证 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，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，如果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，无法在运行时动态地切换虚拟对象。

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，然后在需要的时候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。

5.4　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 支持在运行时动态创建参考图像库。从前面的学习中知道，AR Tracked Image Manager 组件在启动时其参考图像库必须不为 null 值，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，一般是在需要的时候添加 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（后文将实际演示使用）：

```
// 第 5 章 /5-6
mTrackImageManager = gameObject.AddComponent<ARTrackedImageManager>();
XRReferenceImageLibrary runtimeImageLibrary;
// 创建参考图像库
mTrackImageManager.referenceLibrary = mTrackImageManager.CreateRuntimeLibrary(ru
ntimeImageLibrary);
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，启用该组件。

参考图像库可以是 XRReferenceImageLibrary 或者 RuntimeReferenceImageLibrary 类型，XRReferenceImageLibrary 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 会自动转换成 RuntimeReferenceImageLibrary。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.CreateRuntimeLi
brary(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。

5.5 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，下面演示运行时动态切换参考图像库操作。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，编写代码如下：

```
// 第 5 章 /5-8
using System;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
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.CreateRuntimeLi
brary(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.CreateRuntimeLi
brary(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 AR Session Origin 对象，将该脚本挂载于此对象上，并设置好相关属性，将创建设置好的 ReferenceImageLibrary_1 和 ReferenceImageLibrary_2 参考图像库赋给 mReferenceImageLibrary 对象，如图 5-7 所示。

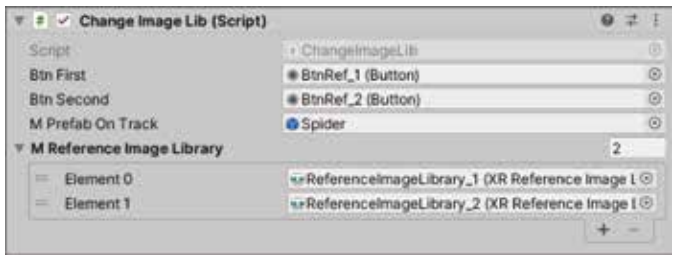

图 5-7　设置参考图像库及其他属性

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

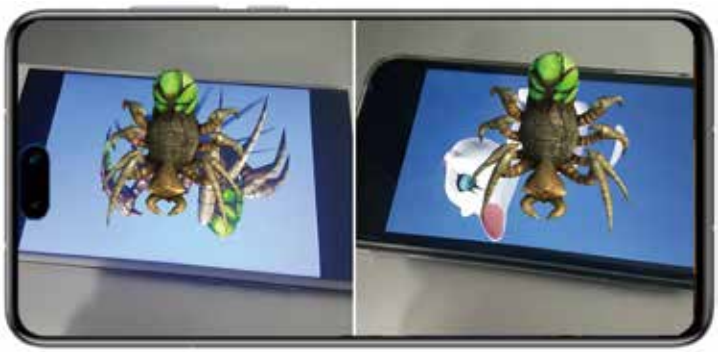

图 5-8　AR 应用能及时对新切换参考图像库中的图像进行检测识别

5.6 运行时添加参考图像

ARCore 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，则可以使用无参的 CreateRuntimeLibrary() 方法，然后将其转换为 MutableRuntimeReferenceImageLibrary 使用，典型代码如下：

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

因为需要提取参考图像的特征值信息，运行时动态添加参考图像是一个计算密集型任务，这大概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 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，该方法的原型代码如下：

```
// 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（单位为米）
    JobHandle inputDeps = default                       // 输入描述信息
);
```

可以通过该方法将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及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，第一要求参考图像可读写，因为添加图像时需要提取图像特征值信息；第二要求图像格式必须为应用平台支持的格式，通常选择 RGB24 或者 RGBA32 格式。这个需要在图像的导入设置（import setting）中设置[①]，如图 5-9 所示[②]。

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 Enable 功能，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，编译时将提示：The texture format ETC_RGB4 is not supported by the current image tracking subsystem。

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。

图 5-9　设置图片相应属性

下面演示在运行时动态添加参考图像，新建一个 C# 脚本，命名为 DynamicImageTracking，并编写代码如下：

```
// 第 5 章 /5-11
using Unity.Collections;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
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.CreateRuntimeLi
brary(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);     //参考图像尺寸宽
        }
        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 MutableRuntimeReferenceImageL
ibrary 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 AR Session 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。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。

5.7 脱卡

从前文中已经知道，在使用 ARCore 进行二维图像检测跟踪时，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，并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（或者随二维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，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，并能与用户正常进行交互，例如在博物馆门口通过检测识别特定图像后实例化的虚拟导游，我们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方式，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，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事件已经发生，但 AR Foundation 指出，有多种未知情况可能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。

但考虑到二维图像检测识别后，通常会实例化虚拟对象，虚拟对象会追踪二维图像的位置，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端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指定的距离
            gameObject.transform.position = mCamera.transform.position +
mCamera.transform.forward.normalized * distance;
            gameObject.transform.rotation = mCamera.transform.rotation;
        }
    }
    // 当二维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，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的距离。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。

5.8　图像跟踪优化

在二维图像被检测到之后，ARCore 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二维图像绑定的效果 (虚拟元素的姿态会随二维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二维图像检测跟踪，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 ARCore 进行图像检测，其他注意事项如表 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
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，因此 ARCore 提供了 ARCoreIMG 工具，使用 ARCoreIMG 工具可以获取每个参考图像的得分情况，取值范围为 [0,100]，建议参考图像的得分至少为 75 分。

ARCoreIMG 是一个命令行工具，它不仅可以获取参考图像得分，还可以从一组参考图像生成参考图像库文件，此工具主要用于检查参考图像的质量。该工具支持 Windows、Linux 系统，使用命令行操作。在 Windows 下检查参考图像质量的命令示例如下[①]：

```
arcoreimg  eval-img  -input_image_path=4.jpg
```

ARCoreIMG 工具执行命令后会返回相关结果，常见结果如表 5-6 所示。

表 5-6　ARCoreIMG 返回结果及原因

返 回 结 果	原　　因
返回数值（如 55）	参考图像评估得分，参考值为 [0，100]，得分越高，检测识别成功率越高，建议选择分值为 75 分以上图像作为参考图像
input_image_path is not specified!	命令行中需要添加 input_image_path 参数
Unsupported image format.	格式不支持，只支持 JPG、PNG 两种图像格式，并且与具体图像编码格式有关，有时即使为 JPG 格式，也会报这个错误
Failed to get enough keypoints from target image.	图像纹理不丰富，无法提取足够的特征值，建议更换参考图像

参考图像的选择有一定的技巧，包含高度重复特征的图像得分会很低，因此建议采用细节丰富、纹理不重复的图像作为参考图像。

2. 有关参考图像库创建

ARCore 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。在运行时添加参考图像，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

① 官方给出的检查参考图像质量命令示例为 arcoreimg eval-img dog.png，经过测试，使用该命令会提示如下错误：input_image_path is not specified，正确的命令应该加上 input_image_path 参数。

加参考图像需要使用异步操作。另外，不要在参考图像库中保存不使用的参考图像，因为这会对应用性能产生一定的影响。

3. 有关跟踪优化

在 ARCore 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，为了更好地优化二维图像检测跟踪，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，此元数据可以提升识别跟踪性能，特别是对于较大的物理图像（长和宽超过 75cm），ARCore 会使用这些物理尺寸评估二维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二维图像的贴合度。

（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 ARCore 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 ARCore 的检测。

（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 100 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切换参考图像库。