第5章

编程基础

本章首先介绍 Unity3D 编程基础知识，然后分别介绍 HTC VIVE 开发环境，投影式 VR 系统开发环境以及 HoloLens 开发环境等常见的 VR 编程环境。

5.1 Unity3D 编程基础

5.1.1 Unity3D 简介

Unity3D 是目前最为流行的游戏引擎之一，具有强大的跨平台发布功能，同时 Unity3D 对各类虚拟现实、混合现实硬件设备提供了良好的开发接口，可以方便地对 Oculus Rift、HTC VIVE 及 HoloLens 等主流 VR/MR 硬件设备进行集成开发，适合进行虚拟现实应用开发。本书统一采用 Unity3D 作为实例开发平台。

本章介绍 Unity3D 引擎开发基础知识，包括 Unity3D 的基本操作、动画角色的基本控制及虚拟相机的设置与使用，在后续的实战篇章节中将结合虚拟现实实例开发，详细讲述 Unity3D 开发知识。

Unity3D 由 Unity Technology 公司开发，2004 年推出最初版本。近年来，Unity3D 强化了对各类交互硬件设备的开发支持，从传统的游戏开发逐步扩展到各类虚拟现实及人机交互等系统开发。经过十余年的发展，目前全球已有数百万开发者使用 Unity3D 进行产品开发。

Unity3D 具有如下特点。

1. 支持跨平台发布

Unity3D 支持跨平台发布，开发者不用过多考虑各平台间的差异，只需开发一套工程，就可以在 Windows、Linux、Mac OS X、iOS、Android、Xbox One 和 PS4 等不同系统下跨平台发布运行。在 Unity3D 中选择菜单 File→Build Settings，在弹出窗口中可以选择不同的发布平台，无须额外的二次开发与移植工作，可以节省大量的开发时间和精力。

2. 丰富的插件支持

在 Unity3D 官方资源商店 Asset Store 中提供了大量的材质、粒子特效及物理仿真方面的插件，由此可实现许多视觉特效制作，节省开发时间，提高视觉效果。同时 Unity3D 也具有各类 VR 设备及交互设备的插件支持，例如免费的 Steam VR 插件及 Kinect with MS-SDK 插件，支持在 Unity3D 平台中使用 Oculus Rift、HTC VIVE 等头戴式 VR 设备的开发及 Kinect

交互设备的开发。

3. 良好的集成开发界面

Unity3D 提供了图形化的集成开发界面，用户可以通过鼠标拖曳完成代码、材质及各类组件与游戏模型之间的绑定工作。同时 Unity3D 提供了与主流三维动画平台 Maya 及 3ds Max 类似的三维模型交互操作界面，可以方便地将三维模型资源组合成所需游戏场景，如图 5.1 所示。

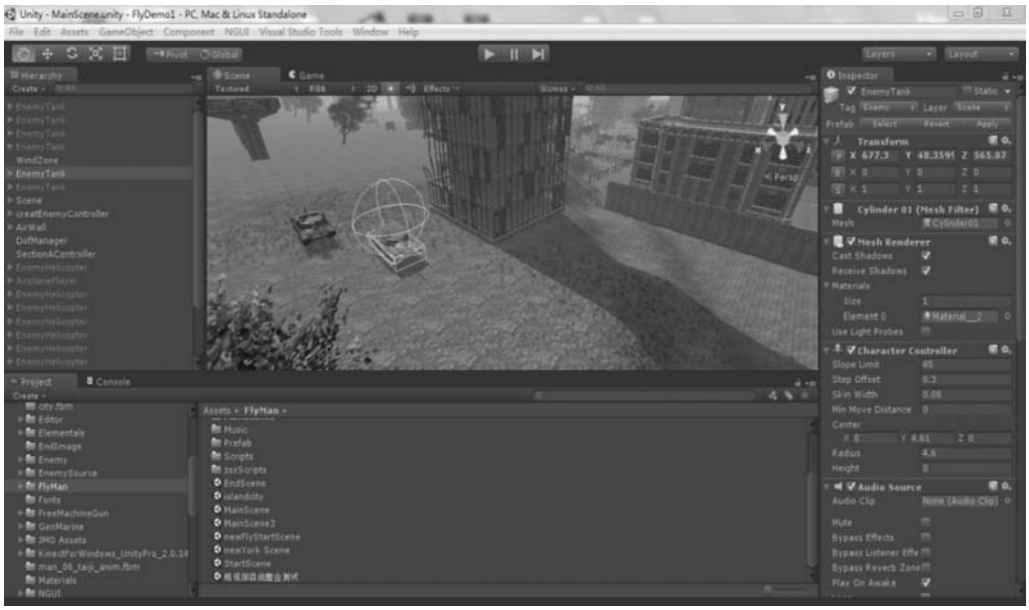

图 5.1　Unity3D 的编辑器

4. 方便的代码开发与调试

早期版本的 Unity(如 Unity 5.x)支持采用 JavaScript、Boo 及 C＃语言进行代码开发，新版本 Unity(如 Unity 2019.x)只支持 C＃语言代码开发。其中，C＃语言封装性好，学习上手快，便于快速开发和第三方代码移植，是进行 Unity3D 代码开发的首选。本书采用 C＃语言进行代码示例与开发。Unity3D 的代码编辑与调试工作可以在微软的 Visual Studio 平台中进行，具有良好的代码输入提示功能，同时具有良好的断点跟踪及变量检查等代码调试功能，便于开发与调试。

5.1.2　Unity3D 集成开发界面基本操作

本章主要介绍 Unity3D 的主要操作界面以及脚本编辑器。

1. 主要操作界面

1) 层次视图

层次视图(见图 5.2)显示当前打开场景文件(Scene)在场景视图(Scene View)中显示或隐藏的所有游戏物体。

图 5.2　层次视图

5.1.2 节

2）场景视图

场景视图(Scene View)(见图 5.3)可视化显示游戏场景中的所有对象,可以在此视图中操纵所有物体的位置、旋转和尺寸。在层次视图(Hierarchy)中选择某对象后按 F 键(Frame Selected),可在场景视图中快速找到该物体。

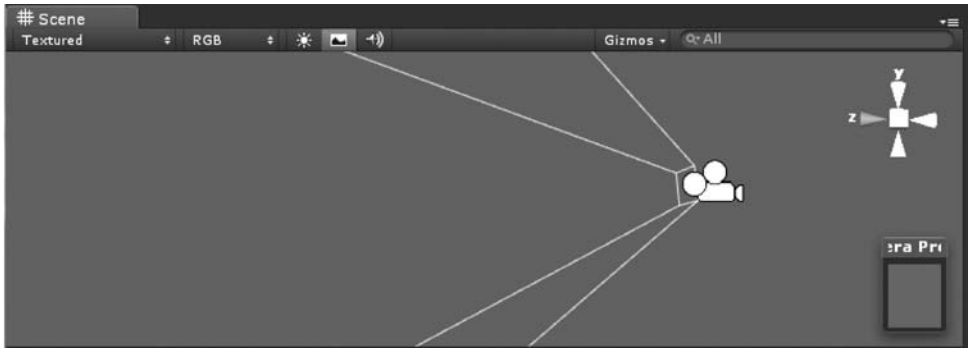

图 5.3 场景视图

3）工程视图

工程视图(Project)(见图 5.4)是用于存储所有资源文件的地方,无须担心数据量,只有在导出场景中用到的资源才会被打包编译。

4）检视面板

检视面板(Inspector)(见图 5.5)用于显示当前物体附带的所有组件(脚本也属于组件)。展开组件显示所有组件的属性数值。

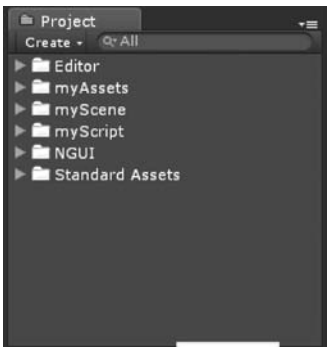

图 5.4 工程视图

图 5.5 检视面板

5）游戏视图

游戏视图(Game View)(见图 5.6)可以把场景视图中所有相机看到的视角在这里显示,并执行所有物体中运行的组件和脚本。单击“播放”按钮进入播放模式。

2. 脚本编辑器

Unity3D 采用 C# 作为脚本语言,在编写脚本时,使用 VS 作为默认开发环境。VS 功能强大,可以同时支持以上三种脚本语言,图 5.7 是一段简单的脚本结构分析。

图 5.6　游戏视图

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class NewBehaviourScript : MonoBehaviour {
5
6     // Use this for initialization
7     void Start () {
8
9     }
10
11    // Update is called once per frame
12    void Update () {
13
14    }
15 }
16
```

图 5.7　一段简单的脚本结构分析

图 5.7 中列举了 Unity3D 脚本中常用函数 Start()和 Update()。Unity3D 脚本中的重要函数均为回调函数，Unity 在执行过程中会自动调用相关函数。就本例而言，图 5.7 中的 Start()函数仅在 Update()函数第一次被调用前调用。而 Update()函数在其所在脚本启用后，将在之后的每一帧被调用。Unity3D 还提供其他大量在不同时刻调用的回调函数，读者可自行查阅相关资料。


5.1.3 节

5.1.3　动画角色控制

在虚拟现实应用开发中，虚拟动画角色的导入与动作控制是最基本的环节，用户可以通过各类体感输入设备控制虚拟动画角色完成不同的动作，本节以一个官方实例角色的导入与动作控制为例介绍相关基础知识。在本节中，用户仍使用传统的键盘交互进行角色控制，在后面章节介绍的系统中，用户将进一步通过体感输入设备进行更为自然的交互与控制，但基本的编程开发流程是一样的。

对于虚拟动画角色的导入与动作控制，Unity3D 官方提供了一组标准资源，包括若干基本动画角色与基本特效，供用户进行学习与测试。自 Unity 2018 版本开始，官方标准资源(Standard Assets)需要开发者在官方资源商店网站 Asset Store 中进行下载和导入。

在 Window 菜单中选择 Asset Store 选项，在场景窗口上方的面板栏就出现了 Asset Store 栏，进入该栏后 Unity3D 将自动链接到官方 Asset Store 网站，用户可以下载各类资源，如图 5.8 所示。

在搜索栏中输入 Standard Assets 关键字进行搜索，选择并进入官方标准资源包的下载页面，如图 5.9 所示，这一资源包是完全免费供开发者学习和使用的。

在页面中单击 Import 按钮，官方标准资源包将下载和导入至本地 Unity3D 工程目录中。

现在回到场景面板栏，在 Project 窗口中找到并进入 Standard Assets→Characters→ThirdPersonCharacter→Models 目录，将其中名为 Ethan 的项目用鼠标左键拖动到 Hierarchy 窗口中，将 Unity3D 官方提供的男孩模型 Ethan 导入场景之中，如图 5.10 所示。

接下来将为动画角色 Ethan 添加运动控制与动画控制。首先要说明的是，Unity3D 导入的动画角色模型，一般包含一组基本动画供开发者调用。动画控制就是开发者通过代码来控制何时播放哪一种动画动作及动画动作之间的转换。这不仅是游戏控制的基本环节，也是虚拟现实交互的基本环节。首先让 Ethan 做一个简单的游戏动画(Idle)，即一个角色在无输入控制时自然站立、身体自然轻微运动的动画，表示角色在等待用户进行动作控制。

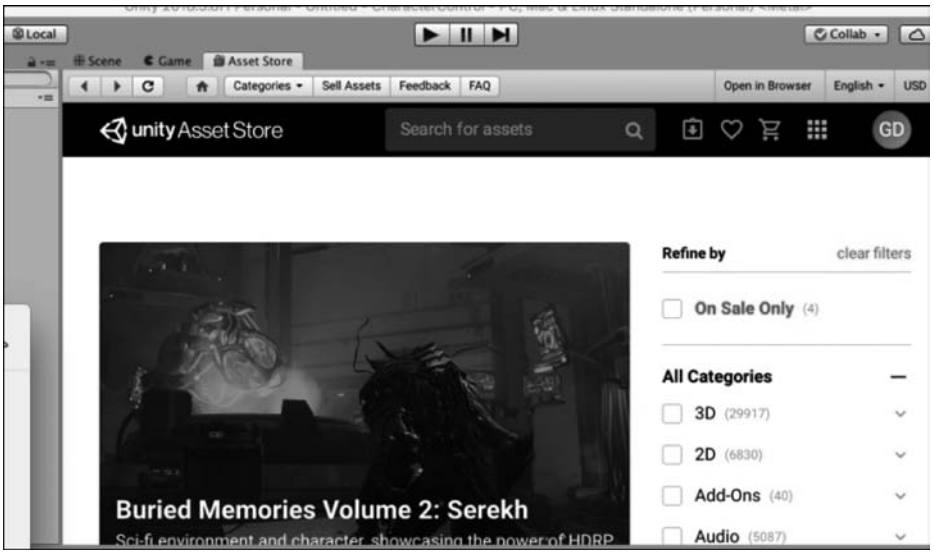

图 5.8 链接到官方 Asset Store 网站

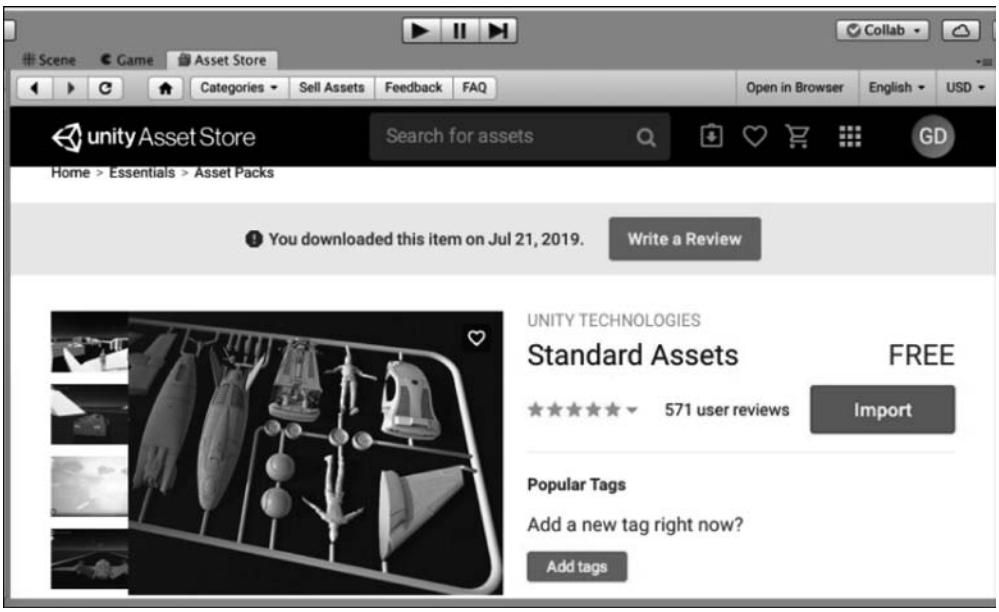

图 5.9 进入官方 Standard Assets 资源包下载页面

首先在 Assets 窗口中右击出现浮动菜单,选择 Create→Animator Controller 命令,新建一个动画控制器元件,将名字对应改为 Ethan,并将该项拖动至角色 Ethan 的 Inspector 面板中 Animator 组件下的 Controller 栏中,表明将通过该动画控制器控制角色 Ethan 的动画运动,如图 5.11 所示。

接下来在动画控制器中进行动画控制的设置,在 Inspector 栏中双击 Animator 组件下的 Controller 命令,场景窗口自动切换至 Animator 栏,这是一个以节点图方式呈现的动画控制界面,每一个方块形的节点表示一个动画状态,通过节点设置及代码控制可以实现不同动画之间的切换,使角色根据用户输入完成指定的动画动作。

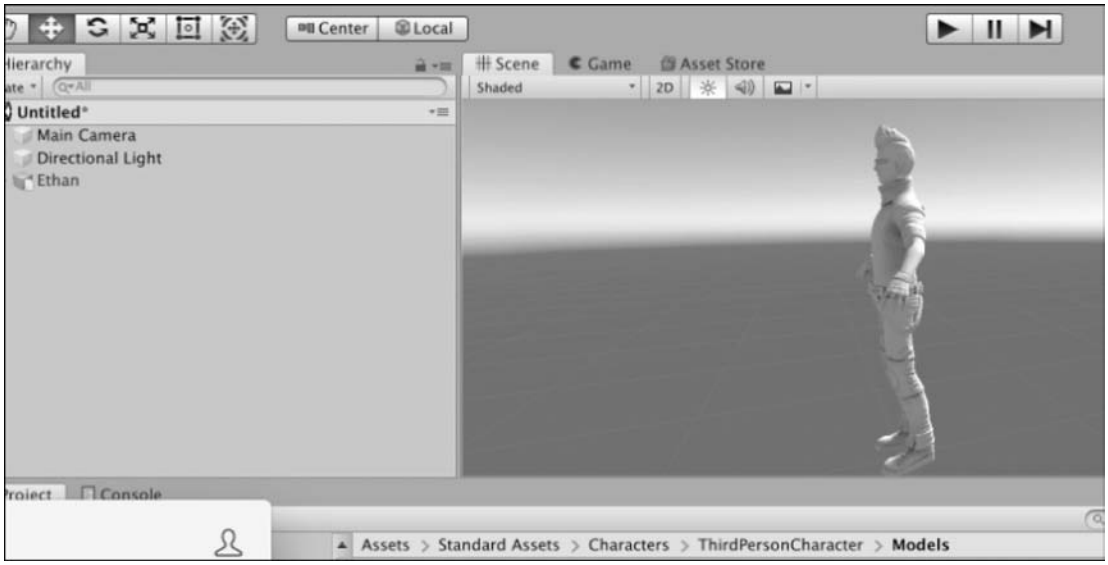

图 5.10　将 Unity3D 官方提供的模型 Ethan 导入场景

图 5.11　为角色创建一个动画控制器

首先添加第一个动画节点，右击 Create State→Empty，新建一个空的动画节点，在右侧 Inspect 栏中将节点名称改为 Idle，表示此节点对应于角色等待动画。随后单击 Motion 栏，在出现的弹出菜单栏中，可以看到角色 Ethan 附带的一系列动画动作，选择其中的 HumanoidIdle 动画，如图 5.12 所示，运行游戏会发现角色 Ethan 开始执行等待动作。

接下来的控制分为两个步骤：首先实现通过键盘上的上下左右箭头四个按键控制角色 Ethan 在场景中移动，控制动画角色在虚拟场景中的漫游。其中，上下箭头控制角色位置前进

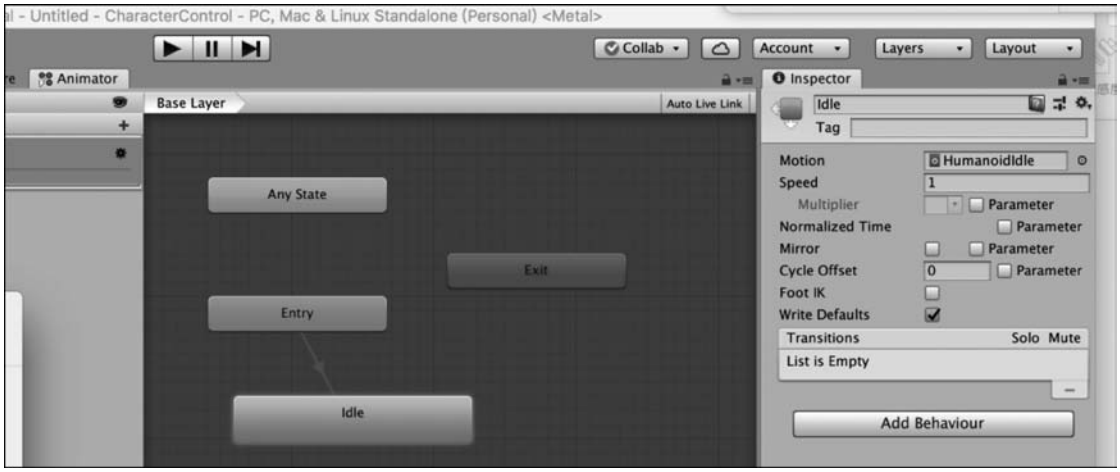

图 5.12　在动画控制面板中设置节点与动画

与后退,左右箭头控制角色的朝向进行左右转动;随后在动画控制器中添加相应的动画动作,控制角色执行前后走动(跑动)及转向走动(跑动)。经过这两个步骤,就可以控制角色 Ethan 进行场景漫游。

在 Assets 窗口中新建一个 C# 脚本,更名为 EthanControl.cs,将脚本拖动至 Ethan 角色,并双击脚本图标在 Visual Studio 中打开,并在 Update()函数体中输入如下代码。

```
void Update()
{
    float h = Input.GetAxis("Horizontal");
    float v = Input.GetAxis("Vertical");

    Vector3 LocalVelocity = new Vector3(0, 0, v);
    Vector3 Velocity = this.transform.TransformDirection(LocalVelocity);
    this.transform.localPosition += Velocity*Time.deltaTime;
    this.transform.Rotate(0, h * 2, 0);
}
```

在上述代码中,前两行是 Unity3D 提供的标准键盘输入代码。变量 h 的取值为$-1\sim+1$,表示按上下箭头的强度:按上箭头时,h 取值为 $0\sim1$,取值大小表示前进幅度;按下箭头时,h 取值为 $0\sim-1$,取值大小表示后退幅度;不按键时 h 取值为 0,表示角色静止。同样地,变量 v 的取值也为$-1\sim+1$,表示按下左右箭头的强度:按右箭头时,v 取值为 $0\sim1$,取值大小表示右转幅度;按左箭头时,v 取值为 $0\sim-1$,取值大小表示左转幅度;不按键时 v 取值为 0,表示角色朝向正前方。在下面的步骤中,将通过键盘输入的 v 值与 h 值控制角色的位置、朝向及动作。

此时运行游戏,可以通过上下箭头控制角色前进后退,通过左右箭头控制角色转向。但此时角色在移动过程中,只是执行等待动画,呈现一种“漂浮”移动的感觉。接下来需要在动画控制器中为角色添加相应动画控制,使角色在受键盘控制移动漫游时,能够执行相应的走动和跑动动画。

首先分析一下动作控制的目标,轻按上箭头时,角色 Ethan 位置慢速前进,应相应执行向前走的动画,同时按左右箭头时,角色 Ethan 位置慢速转向,应相应执行向左走及向右走的动画;长按上箭头时,角色 Ethan 位置快速前进,应相应执行向前跑的动画,同时按左右箭头时,角色 Ethan 位置快速转向,应相应执行向左跑及向右跑的动画;最后按下下箭头时,角色 Ethan

位置后退，应相应执行向后走的动画。为实现上述动画控制，在 Animator 面板中设置混合树节点(Blend Tree)，用于处理上下左右四个箭头按键的混合控制。

打开 Animator 面板，单击鼠标右键新建一个动画节点，更名为 Move，用于控制向前、向左、向右三个方向上的走动及跑动的动画控制。再新建一个动画节点，更名为 WalkBack，相应的 Motion 选项设置为 Humanoid Walk Back，用于控制后退走动的动画控制，如图 5.13 所示。

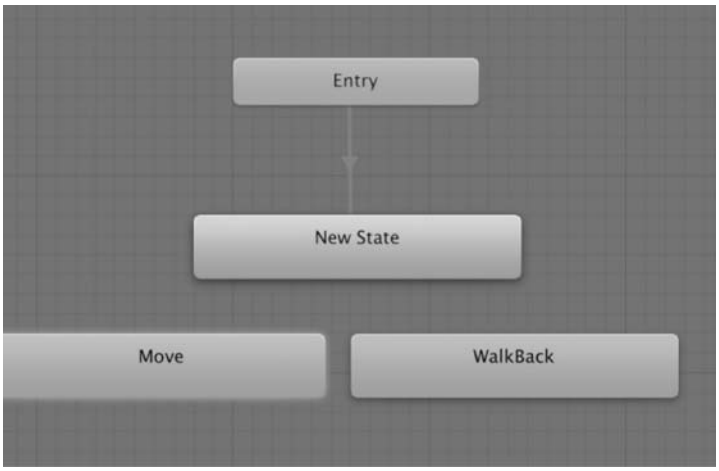

图 5.13　新建两个动画节点

鼠标单击 Move 节点，在弹出菜单中选择 Create New BlendTree in State 命令，为 Move 节点创建一个混合树(Blender Tree)结构，用于混合控制向前走/跑、向左走/跑及向右走/跑共 6 组动画的切换。这 6 组动画通过两级混合树进行控制，第一级根据上箭头按键强度(变量 h 的取值大小)，决定角色 Ethan 执行走还是跑的动画，第二级根据左右箭头按键强度(变量 v 的取值大小)，决定角色 Ethan 执行左转还是右转的动画，如图 5.14 所示。

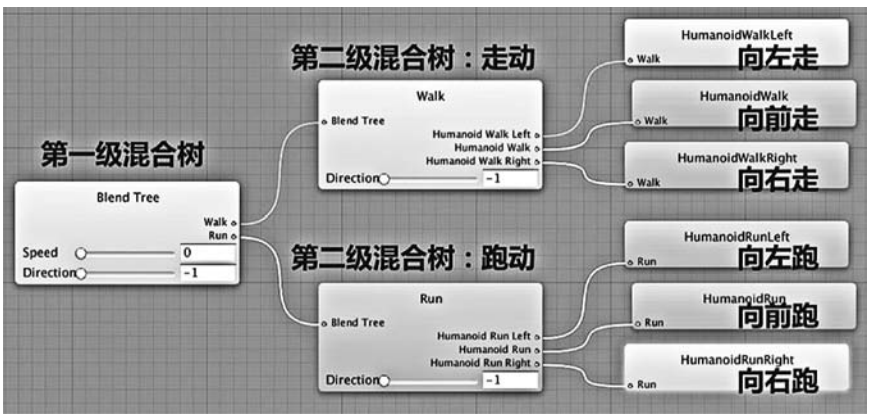

图 5.14　设置两级混合树控制

可以看到，现在 Animator 面板中的混合树可以通过节点上的两个滑块控制 6 种动画之间进行切换。两个滑块分别名为 Speed 与 Direction：Speed 滑块对应键盘前进速度，Direction 滑块对应键盘左右转向角度。后面可以通过代码中的 v 变量取值及 h 变量取值对应控制

Speed 与 Direction 两个滑块的取值,实现代码控制。

现在回到 Animator 面板的根界面,在静止等待节点 Idle 与移动动画节点 Move 之间设置两条往返箭头,进行动画状态转换设置:由静止等待切换到移动状态的条件是 Speed>0.1,即一旦按下上箭头,角色 Ethan 就开始向前、左、右三个方向移动;由移动状态反向切换到静止等待的条件是 Speed<0.1,即一旦松开上箭头,角色 Ethan 就开始执行静止等待动画。

接下来在静止等待节点 Idle 与后退动画节点 WalkBack 之间同样设置两条往返箭头。当 Speed<−0.1 时,即一旦按下箭头,角色 Ethan 就开始倒退行走,反之松开下箭头,角色 Ethan 就开始执行静止等待动画,如图 5.15 所示。

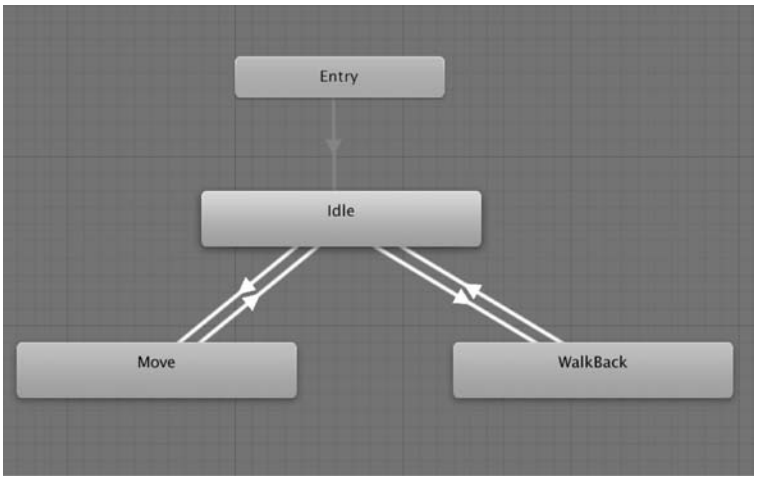

图 5.15 设置动画节点之间的转换路径

接下来在动画控制脚本 EthanControl.cs 中的 Update()函数体中添加如下代码。

```
mAnimator.SetFloat("Speed", v);
mAnimator.SetFloat("Direction", h);
```

分别将变量 v 与变量 h 的取值赋给 Animator 面板中的动画切换滑块 Speed 与 Direction,从而通过用户的键盘操作控制角色 Ethan 的动画切换过程。

最后在 Hierarchy 面板栏中将主相机 Main Camera 拖动到角色 Ethan 的里面。该操作的意义是将主相机设置为角色 Ethan 的子物体,即主相机将跟随角色 Ethan 前后移动及转向。由于现在主相机位于角色脑后方向,因此执行效果是以第三人称视角进行场景漫游。

现在可以通过上、下、左、右箭头按键控制角色进行动画并漫游。

本节通过官方自带动画角色,实现了一种简单的第三人称漫游控制。在例子中用户是通过键盘进行控制的,在后面章节中可以替换为使用深度相机、交互手柄进行控制,但动画切换控制的原理基本上相同,只不过更换了输入手段。


5.1.4 节

5.1.4 虚拟相机设置

1. 相机参数设置

在 Hierarchy 窗口中选择场景中的相机,在右侧 Inspector 窗口中就显示出相机的各项设

置，如图 5.16 所示。

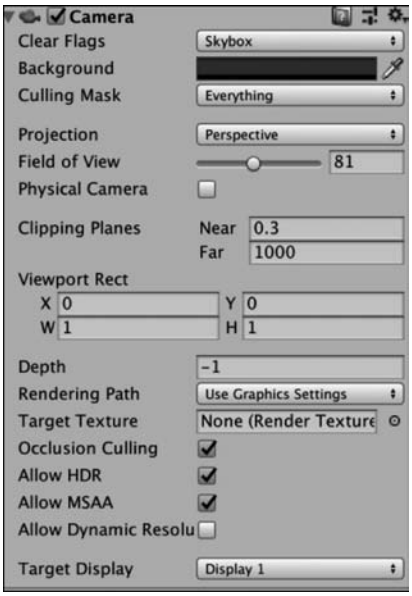

图 5.16　相机的各项参数设置

其中第一栏 Clear Flags 设置了虚拟相机背景画面类型，单击下拉箭头可以看到对应选项，默认选项为 Sky Box，即设置相机渲染背景为天空盒图像；第 2 选项为 Solid Color，设置相机渲染背景为用户指定的单色图像，背景色在第二项 Background 栏中设置；第 3 选项为 Depth Only，设置相机渲染输出为表示场景 3D 深度次序的灰度图像，应用该项设置时相机输出画面用于特效叠加处理；第 4 选项为 Don't Clear，设置相机的输出图像为前后帧叠加效果，用于产生运动模糊特效。通常情况下采用默认的 Sky Box 选项。

第三栏 Culling Mask 设置相机的渲染掩模，即设置相机可以看到虚拟场景中的哪些物体与角色。其默认选项为 Everything，设置相机可以看到所有物体。单击下拉箭头可以按照虚拟场景物体的分层设置情况，具体选择相机只渲染指定层中的物体，而不处理其他层中的物体。这一设置的意义是可以根据漫游情况忽略过远处的物体，提高实时渲染效率。Culling Mask 的设置既可以手动调整，也可以在游戏运行过程中通过代码实时调整。

第四栏 Projection 项设置相机的视景体模式。其默认选项为 Perspective，相机设置为透视相机，渲染画面呈现近大远小的透视效果，用于显示 3D 场景；如果设置为 Orthographic，相机被设置为垂直相机，渲染画面无透视效果，用于渲染显示 2D 视图。

第五栏 Field of View 项用于设置透视相机的视角大小。视角越大相机视野也就越大，通常设置为 60～90°，超过 90°则会出现广角镜头效果。

第七栏 Clipping Planes 用于设置透视相机的近切面与远切面。透视相机只对近切面与远切面之间的物体进行渲染，超出范围的物体则不予处理。

第八栏 Viewport Rect 用于设置相机渲染输出画面在整个屏幕中的占比，具体有四个选项，其中，X 与 Y 选项设置渲染画面左上角在屏幕中的位置，例如(X，Y)＝(0，0)表示画面左上角位于屏幕左上角，(X，Y)＝(0.5，0.5)表示画面左上角位于屏幕中心；W 与 H 选项设置渲染画面的宽度与高度，例如(W，H)＝(1，1)表示渲染画面的宽度、高度与屏幕尺寸相同，(W，H)＝(0.5，1)表示渲染画面的宽度为屏幕一半，高度与屏幕相同。

第九栏 Depth 用于设置相机深度序号，用于处理场景中多个相机画面的深度叠加。Depth 数值大的相机为前景相机，Depth 数值小的相机为背景相机。前景相机画面会遮挡背景相机画面。

第十栏 Rendering Path 用于设置相机的渲染顺序，影响相机对场景中透明物体的渲染效果。

第十一栏 Target Texture 用于指定相机的渲染目标纹理，即相机的渲染输出可以不直接输出到屏幕，而是暂存于指定的目标纹理区域中，以便进行合成处理后再输出至屏幕。该选项常用来进行立体视频中左右眼画面的合成。

2. 渲染次序控制

在游戏开发场景中通常只有一台主相机负责画面实时渲染,但在 VR 应用中经常需要在同一个场景中使用多台虚拟相机,例如,实现立体显示时需要两台虚拟相机来模拟用户的左右眼,或者在虚拟漫游过程中需要从不同角度观察虚拟环境,都需要设置多台虚拟相机。本节介绍如何使用多台相机并控制其渲染次序。

在 5.1.3 节的角色控制实例中添加一台虚拟相机,该相机作为角色 Ethan 的子物体,放置在角色头部正中跟随角色运动。该相机以角色自身眼睛的视角观察场景,称之为第一人称相机。而之前主相机被放置在角色头部后上方跟随角色运动,不仅能看到前方的虚拟环境,同时也能看到虚拟角色的动作,称之为第三人称相机。下面以此为例讲述一下如何在场景中设置多台相机,从不同视角渲染场景并进行视角切换控制。后续章节中设置立体投影双目相机的基本原理与此相同,只是需要进一步细致设置双目相机位置绑定及调整视景体。

找到绑定在角色 Ethan 上的场景主相机 MainCamera,通过菜单 GameObject→Camera,在场景中新建两台相机作为第一人称相机与第三人称相机,分别命名为 FirstPersonCamera 与 ThirdPersonCamera。将这两台相机拖动至 MainCamera 下,成为主相机的子物体,第一人称相机位于角色 Ethan 头部,第三人称相机位于角色 Ethan 头部后上方,选择这三台相机观察各种渲染视角,如图 5.17 所示。

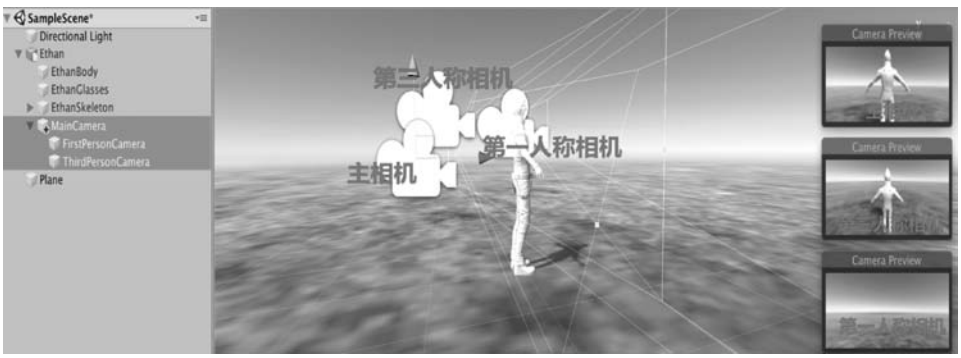

图 5.17 选择这三台相机观察各种渲染视角

现在运行游戏程序,发现虽然场景中存在三台相机,但运行时输出的只有主相机视角画面,现在通过代码脚本控制第一人称相机与第三人称相机进行渲染输出,并使用户可以对两个视角进行切换显示控制。

为主相机 MainCamera 添加一个新建的 C# 脚本 CameraControl.cs,在脚本类声明之后,添加两个 Public 型的 Camera 变量,用于访问和控制第一人称相机与第三人称相机。将两台相机图标拖到 Inspector 窗口中脚本对应的 Public 项中,建立相机对象与 Public 型变量之间的关联,如图 5.18 所示。

随后在脚本中再声明两个 RenderTexture 型变量。RenderTexture 是指一种特定的内存空间,可以暂存虚拟相机的渲染画面,也就是说,虚拟相机的渲染结果可以不必直接输出至显示设备,而是暂存于指定内存区域中,待进行后续处理之后再推送至显示设备。脚本中定义的两个 RenderTexture 型变量分别用于存储第一人称相机与第三人称相机的渲染结果。然后根据用户控制选择其中一个视角画面进行显示输出。随后再定义一个 int 型变量 ViewFlag 用于记录当前视角选择,初始值设为 3 表示以第三人称视角显示输出,如图 5.19 所示。

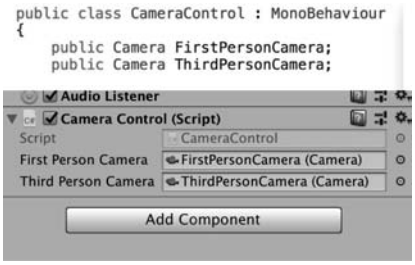

图 5.18　在相机控制脚本中添加 Public 型变量

```
public class CameraControl : MonoBehaviour
{
    public Camera FirstPersonCamera;
    public Camera ThirdPersonCamera;

    RenderTexture FirstPersonCamRT;
    RenderTexture ThirdPersonCamRT;
    int ViewFlag=3;
```

图 5.19　在相机控制脚本中声明 RenderTexture 型变量

接下来，在脚本的 Start()函数体中添加如下代码，如图 5.20 所示。其中，第 1、2 行代码分别为声明的 RenderTexture 变量申请内存区域，内存区域大小与显示设备屏幕分辨率相同；第 3、4 行分别将第一人称相机与第三人称相机的渲染输出指向对应的 RenderTexture 内存区；第 5、6 行分别禁止第一人称相机与第三人称相机进行自动渲染，由用户指定某一个视角相机进行受控渲染。

```
void Start()
{
    FirstPersonCamRT = new RenderTexture(Screen.width, Screen.height, 24);
    ThirdPersonCamRT = new RenderTexture(Screen.width, Screen.height, 24);

    FirstPersonCamera.targetTexture = FirstPersonCamRT;
    ThirdPersonCamera.targetTexture = ThirdPersonCamRT;

    FirstPersonCamera.enabled = false;
    ThirdPersonCamera.enabled = false;

}
```

图 5.20　对第一人称相机与第三人称相机进行设置

在 Update 函数体中添加如图 5.21 所示的代码，使用户可以通过键盘(F1 键/F3 键)控制输出视角。

最后在脚本中新建一个名为 OnRenderImage 的函数体，如图 5.22 所示。OnRenderImage()函数体与 Update()函数体一样，在每一个游戏帧中进行调用，作用是在相机渲染输出至显示设备之前对渲染结果进行再次处理。其调用顺序是在 Update 函数体之后。

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.F1))
    {
        ViewFlag = 1;
    }
    if (Input.GetKeyDown(KeyCode.F3))
    {
        ViewFlag = 3;
    }
}
```

图 5.21　用户通过键盘控制输出视角

```
void OnRenderImage(RenderTexture source, RenderTexture destination)
{

    if (ViewFlag == 1)
    {
        FirstPersonCamera.Render();
        Graphics.Blit(FirstPersonCamRT, destination);
    }

    if(ViewFlag==3)
    {
        ThirdPersonCamera.Render();

        Graphics.Blit(ThirdPersonCamRT, destination);

    }
```

图 5.22　在 OnRenderImage()函数体中控制两台相机渲染输出

其中第一个 if 判断的作用是，如果当前用户指定第一人称视角画面输出，则控制第一人称相机进行一次渲染，并调用 Blit()函数将存放于 RenderTexture 内存区中的渲染画面输出至显示器 destination；第二个 if 判断的作用是，如果当前用户指定第三人称视角画面输出，则

控制第三人称相机进行一次渲染,并调用 Blit()函数将存放于 RenderTexture 内存区中的渲染画面输出至显示器 destination,从而实现两台相机画面的切换输出。

保存脚本执行游戏,按 F1 键屏幕上显示第一人称视角画面,按 F3 键屏幕上显示第三人称视角画面,用户可以选择切换使用第一人称视角或第三人称视角进行虚拟漫游,如图 5.23 所示。

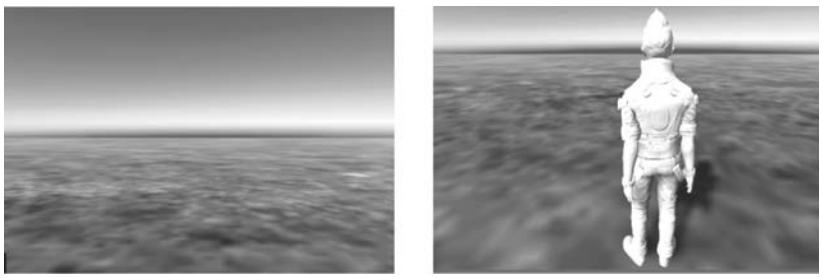

图 5.23 用户控制虚拟漫游视角

5.1.5 节

5.1.5 Unity3D 中函数体的执行顺序

在游戏和 VR 应用中,虚拟环境中的各种事件总是并行出现的,虚拟角色本身会进行各种运动,而用户的交互也会改变虚拟环境与角色的状态,因此游戏引擎和 VR 开发都需要具有并行处理各类事件的机制,本节介绍 Unity3D 中并行处理各类事件时函数体的执行顺序,理清这一问题对深入理解 Unity3D 开发有着重要意义。

虽然各类操作系统中的多线程技术已非常成熟,但 Unity3D 仍然是以单线程运行为主的,实际上目前大多数游戏引擎也都是基于单线程的,其原因何在呢? 多线程的一个重要优点是可以利用 CPU 空闲时间处理多个任务,提高资源利用率,但在游戏和 VR 应用中,角色事件处理与画面更新在响应时间上需要有很强的确定性与实时性,如果在逻辑更新与画面更新处理中使用多线程模式,那么处理多线程同步会大大增加开发消耗,所以目前大多数游戏引擎及 VR 开发的主逻辑循环部分都是单线程的。

在 Unity3D 中,每个角色对象都可以通过所绑定的代码脚本来控制和更新其行为,一个对象可以绑定多个脚本,共同控制该对象的多个或一个行为属性,在一个对象的脚本中也可以去控制和更新其他对象的行为属性。在每个脚本中最重要也最常用的就是 Update()函数体,Unity3D 主逻辑循环会根据硬件实际运行速率对每个脚本中的 Update()函数体进行调用,在 Update()函数体中来响应各种交互指令,并更新所属对象的行为属性,这样虚拟环境中的所有对象、所有角色看起来都像是在并行地运作了。

在代码脚本中除了 Update()之外还有其他重要的函数体,图 5.24 中列出了脚本的基本类 MonoBehaviour 中的若干重要成员函数体的执行顺序,下面结合该图来进行介绍。

图 5.24 中列出的函数体都会在游戏或 VR 应用运行过程的特定时刻,被自动地调用执行,这里着重介绍下面几个常用的函数体:Awake()函数体与 Start()函数体在整个游戏运行过程中只会被执行一次,用于对所属对象进行初始化工作。FixedUpdate()、Update()及 LateUpdate()函数体则会被不断反复调用执行,用于对所属对象的属性进行实时更新,从而控制所属对象的行为。OnPreRender()、OnPostRender()及 OnRenderImage()函数体只能绑定在虚拟相机对象上,用于在每一帧中进行实时渲染的不同阶段进行相机设置、绘图处理及图

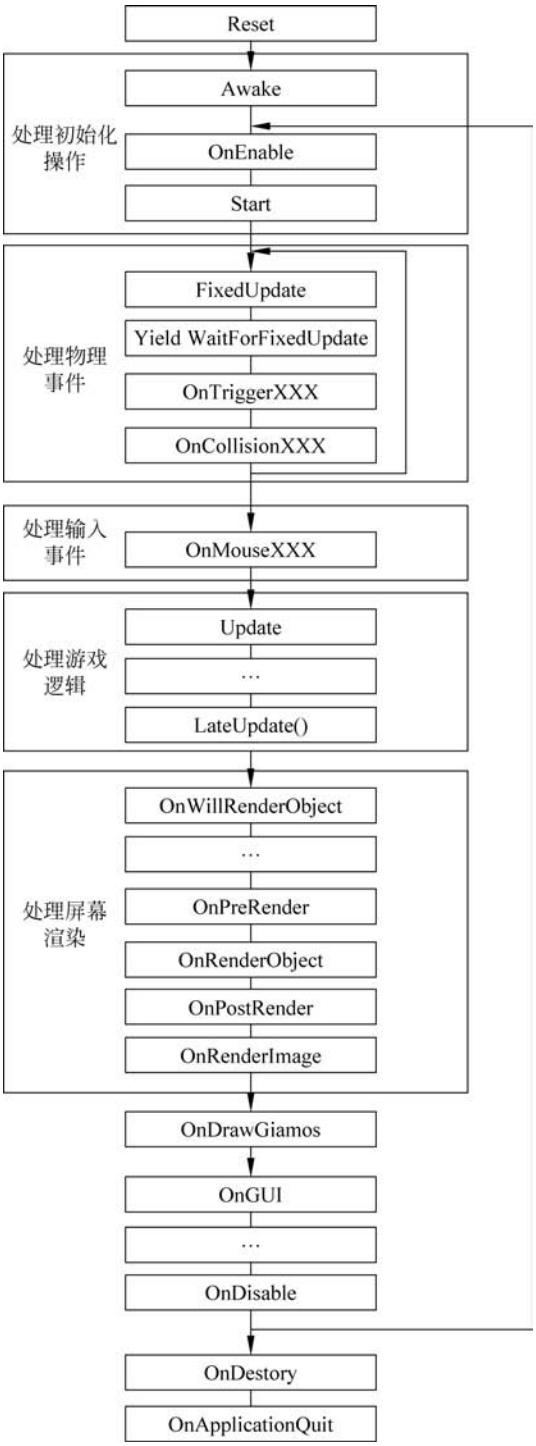

图 5.24 Unity3D 中 MonoBehaviour 类成员函数执行流程图

像特效处理。

图 5.24 中列出的是一个脚本中各函数体的执行顺序，看起来简单明了，但要深入了解其中的机制并正确使用，就需要在多个对象多个脚本并行执行的背景下来理解。在虚拟场景中存在多个对象(例如多种角色及多个道具)，而每一个对象又可以绑定多个脚本。下面介绍一

下多个对象、多个脚本中的这些函数体的执行原则。

首先看用作初始化操作的 Awake()与 Start()两个函数体,我们已经知道在单个脚本中是先执行 Awake()之后,再执行 Start()。但在多对象多脚本情况下,一个脚本中的 Start()并不是在该脚本中的 Awake()执行之后就会被立即执行,而是要等到其他所有脚本中的 Awake()都被执行完毕之后,才会开始调用执行,也就是说,所有脚本中的 Awake()要在首轮中全部执行完毕,才会开始下一轮 Start()的执行,如此设置是为了规范和调整各对象、各脚本之间的初始化顺序。下面来解释一下。

各脚本中的 Awake()函数体是在场景中所有对象都被实例化创建之后被自动调用执行的,但各脚本中的 Awake()的执行顺序是随机选定的。如果在 Awake()中进行初始化的属性变量并不依赖于其他对象或其他脚本,就没有问题,但如果一个属性变量的初始取值依赖于其他对象或脚本中的某个属性取值,就可能会出现错误。例如,需要把场景中所有大精灵 B 的初始速度设置为小精灵 A 初始速度的一半,那么在大精灵 B 脚本中的 Awake()函数体里进行这一设置时,小精灵 A 脚本中的 Awake()可能还没有被调用,其初始速度值也没有初始化,这样就导致大精灵 B 的速度被初始化为错误数值(见图 5.25)。此时就需要把大精灵 B 的初始速度设置工作放在 Start()函数体中,这是因为当大精灵 B 脚本的 Start()函数体开始执行时,所有脚本的 Awake()函数体都已经执行完毕了,大精灵 B 的初始速度设置也就不会出现问题了(见图 5.26)。

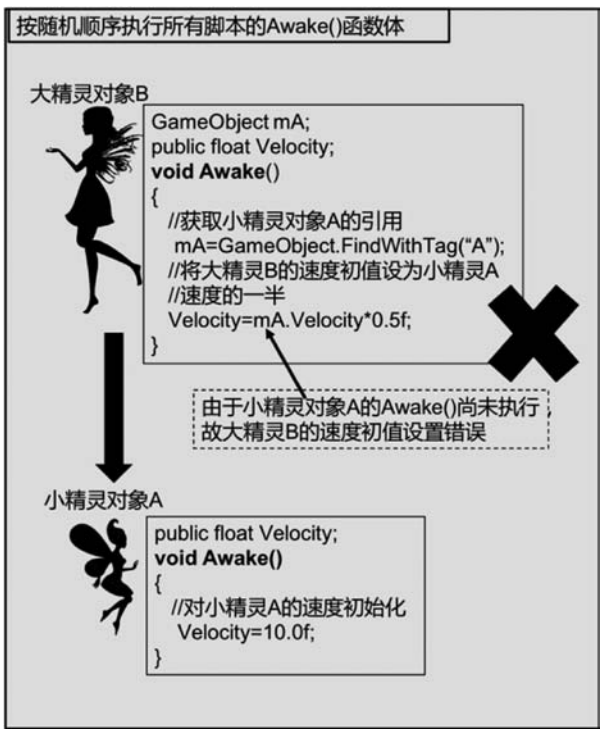

图 5.25　在 Awake()函数体中进行初始化存在潜在错误可能性

需要补充说明的是,由于 Awake()函数体是在场景中所有对象都实例化创建之后才开始执行,所以在 Awake()之中可以正确安全地使用 GameObject.Find()或 GameObject.FindWithTag()函数去查找和获取其他对象的引用,但是通过引用去控制和改变其他对象的属

性时，最好是放在 Start()函数体或更后面的 Update()函数体中执行才是安全的(见图 5.26)。

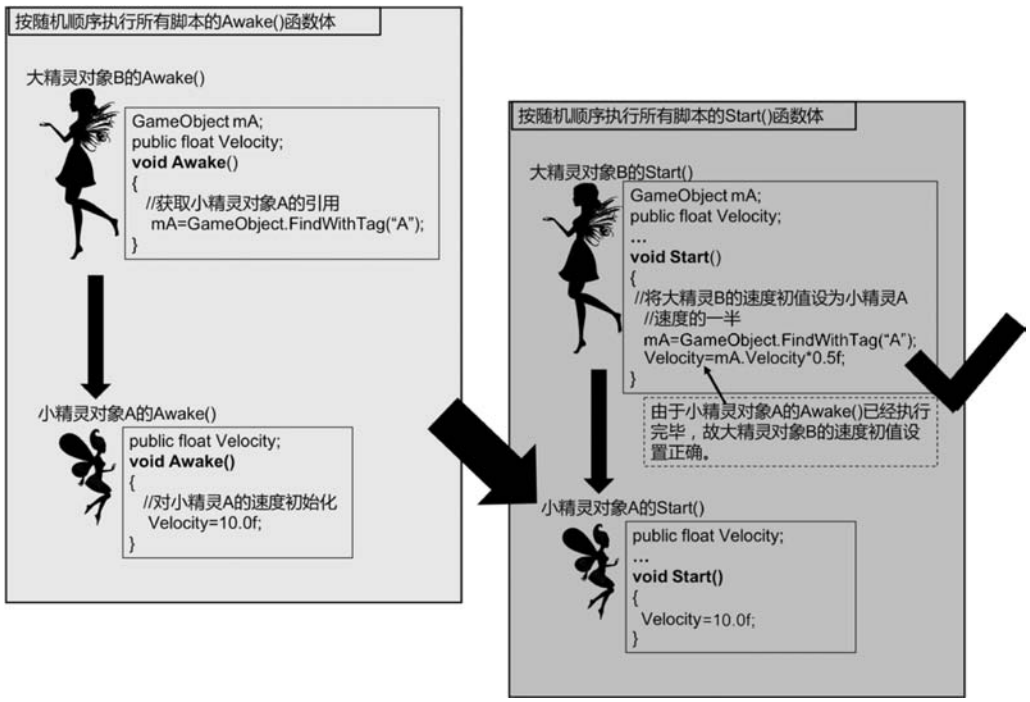

图 5.26　利用 Awake()与 Start()的执行先后顺序正确进行初始化

理解了 Awake()函数体与 Start()函数体之间的执行顺序机制之后，可以类推地理解 FixedUpdate()、Update()及 LateUpdate()三个函数体的执行顺序。首先，Unity3D 自动调用执行所有脚本中的 FixedUpdate()(按照随机顺序)，完毕之后再自动调用执行所有脚本中的 Update()(按照随机顺序)，完毕之后再自动调用执行所有脚本中的 LateUpdate()(按照随机顺序)。这三个函数体对应了游戏对象属性的三个更新轮次：首先第一轮是 FixedUpdate()，该函数体是按照固定时间间隔反复自动调用的，两次调用之间的时间间隔不受硬件情况影响，在该函数体中适于完成对精确物理属性的更新工作。例如，动力学模拟中的受力更新或速度更新。后续两轮 Update()与 LateUpdate()的调用时间间隔则受到硬件情况影响，无法保证完全恒定。设置 LateUpdate()的一个重要作用是为了正确处理对象之间的多重影响，例如在运行过程中，对象 C 的位置要同时受到对象 A 与 B 的影响，而对象 D 需要始终瞄准对象 C。如果上述操作都在各对象的 Update()函数体中执行，那么在同一帧中，对象 D 瞄准对象 C 在前，对象 B 调整对象 C 的位置在后，于是造成了对象 D 没有瞄准本帧中对象 C 的最终位置(见图 5.27)。为此应该在对象 A 与 B 的 Update()中去更新对象 C 的位置属性，等到所有脚本的 Update()函数体都执行完毕后，在相机 D 的 LateUpdate()函数体中再获取对象 C 的位置，并指向 C，实现正确的瞄准效果(见图 5.28)。

需要指出的是，Unity3D 提供了通过手动或代码设置 Script Execution Order(脚本执行顺序)属性的方式，允许用户自行设置和规定多个代码脚本的先后顺序，也就相应设置了各脚本之间 Update()及 LateUpdate()函数体的执行顺序。但对于场景中有众多角色对象的情况，利用 Update()与 LateUpdate()的先后轮次顺序来处理对象间的多重影响，避免造成冲突，这是一种合理而方便的开发技巧。

图 5.27　仅使用 Update()函数体，无法确保正确处理对象间的多重影响

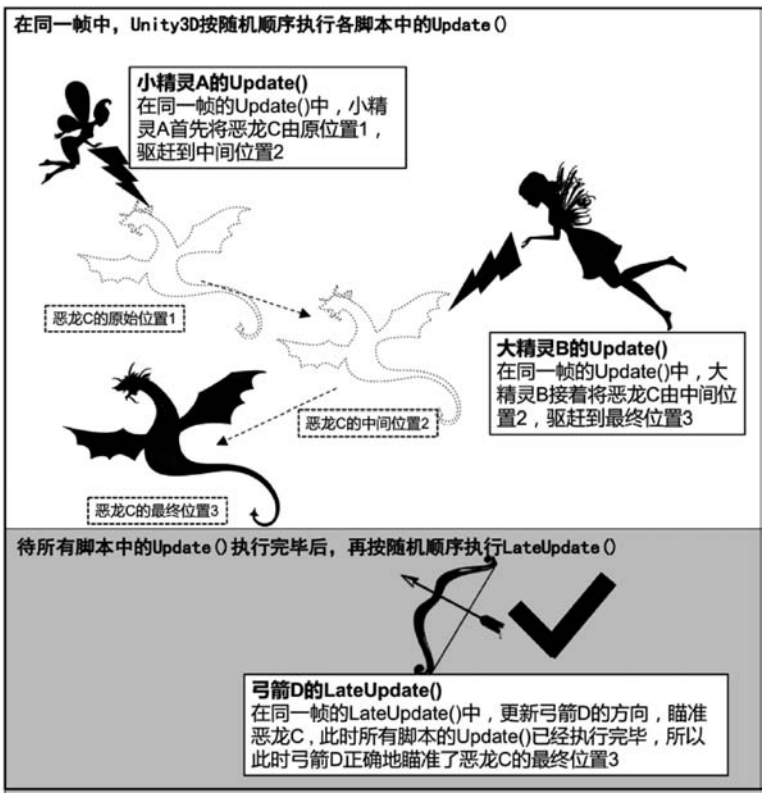

图 5.28　利用 Update()与 LateUpdate()的执行轮次顺序，正确处理对象间的多重影响

最后再介绍一下 OnPreRender()、OnPostRender()及 OnRenderImage()三个函数体的作用。这三个函数体所属脚本必须绑定在场景中的某个虚拟相机对象上,它们的调用顺序同样是按照先后轮次来执行的。OnPreRender()函数体在相机开始渲染之前被自动调用,在其中可以更改设置相机的某些参数,如开/关相机的雾效渲染功能,而 OnPostRender()则在相机完成渲染之后被自动调用,用户可以在其中调用图形库函数,在渲染画面上附加绘制图标图形,如果这一操作放在 OnPreRender()函数体中,那么用户绘制的图标图形就会被渲染画面所覆盖。所有渲染绘制操作完成之后,在将结果输出到显示设备之前,OnRenderImage()函数体被自动调用,允许用户在其中对渲染画面进行后期特效处理,例如对画面进行高斯平滑、运动模糊或画面泛光(Bloom)处理。

5.2 投影式 VR 系统开发环境

在第 3 章和第 4 章中介绍了立体显示的基本原理,本节中进一步介绍如何通过 Unity3D 平台实现立体投影显示功能。首先分析立体图像视差与立体显示效果的关系,在此基础上再介绍如何正确渲染生成立体图像,最后介绍如何按 120Hz 刷新率顺序显示立体图像,形成立体视频流。

5.2.1 视差与立体显示效果的关系

左右眼图像中的视差使用户产生了立体视觉。在目前常用的立体显示技术中,只有水平视差,而没有垂直视差,虚拟场景中的一个物体在左眼图像中的水平位置为 L,在右眼图像中的水平位置为 R,则定义该物体的视差 $P=R-L$。当物体距离较远时,会穿入屏幕出现在屏幕后方,称为入屏效果;当物体距离较近时,会穿出屏幕出现在屏幕前方,称为出屏效果。出屏还是入屏取决于物体的视差,而视差有零视差、正视差、负视差及发散视差四种情况。下面结合图 5.29 进行具体分析。

零视差:如图 5.29(a)所示,物体在左右眼图像中的水平位置重合,视差 $P=0$,此时物体出现在屏幕上。

正视差:如图 5.29(b)所示,当左右眼图像叠放在一起时,物体在右眼图像中的水平位置 R 位于左眼图像中的水平位置 L 的右侧,视差 $P>0$。这时,物体出现在屏幕后方,即产生入屏效果。此时物体的距离与视差成正比——视差越大物体越远,当视差与用户眼间距相等时,物体出现在无穷远处。

负视差:如图 5.29(c)所示,当左右眼图像叠放在一起时,物体在右眼图像中的水平位置 R 位于左眼图像中的水平位置 L 的左侧,与用户的左右眼呈现交叉,视差 $P<0$。这时,物体出现在屏幕前方,即产生出屏效果。此时物体的距离与视差的绝对值成反比——视差绝对值越大物体越近。当视差与用户眼间距相等时,物体恰好出现在用户到屏幕距离的一半处。

发散视差:如图 5.29(d)所示,当视差值大于两眼的瞳孔距时会产生发散视差。在真实世界中,该情况是不存在的。在立体显示时,此类情况即使存在很短的一段时间,也会使眼睛产生极为不舒服的感觉。因此,在立体显示时应该避免此类情况。

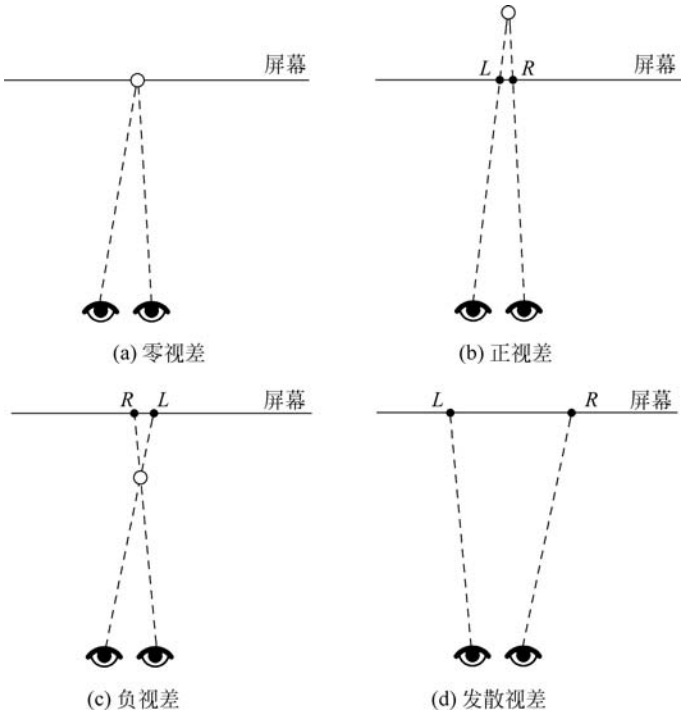

图 5.29 视差与立体显示效果

5.2.2 渲染立体图像

5.2.1 节中分析了视差与立体显示效果之间的关系，本节介绍如何在虚拟场景中设置立体相机的透视投影矩阵，保证立体相机能够渲染生成正确的立体图像。

普通虚拟场景中只需要设置一台相机，而为了立体渲染，就需要设置左、右两台相机，对应于观众的左、右眼。左、右两台相机在水平方向上有一定的间隔距离，相当于人的眼间距(Interaxial)，此时左、右眼图像中就出现了视差。

需要注意的是，如果左、右相机仅进行单纯平移，两台相机的视锥体之间就不存在一个公共的截面(如图 5.30(a)中的左右相机单纯平移和图 5.30(c)中左右相机满足零视差面约束的情况下)。此时渲染出的左、右图像中就只有负视差情况，而没有零视差和正视差情况[①]。为了修正这一问题，就需要为左、右相机的视锥体指定一个公共的截面(如图 5.30(b)中的左右相机单纯平移后的视锥体和图 5.30(d)中左右相机满足零视差面约束时的视锥体)，即零视差面。虚拟场景中位于零视差面前面的点产生了负视差，位于零视差面后面的点产生了正视差，而恰好位于零视差上面的点产生了零视差[②]，即位于零视差面上的点在左、右图像中汇聚为一点，因此称相机到零视差面的距离为汇聚(Convergence)距离。

左右相机的视锥体在零视差面上重合，称为零视差面约束，对比图 5.30(a)与图 5.30(b)，在满足零视差面约束条件下，两台相机的视锥体从原先的对称情况变成了非对称情况，这就需要重新计算左、右相机的透视投影矩阵。在前面介绍过一台虚拟相机的透视投影矩阵是由该相机近切面的上(Top)、下(Bottom)、左(Left)、右(Right)，及近切面距离(Near)和远切面距

① 详细推导见附录 C。

② 详细推导见附录 C。

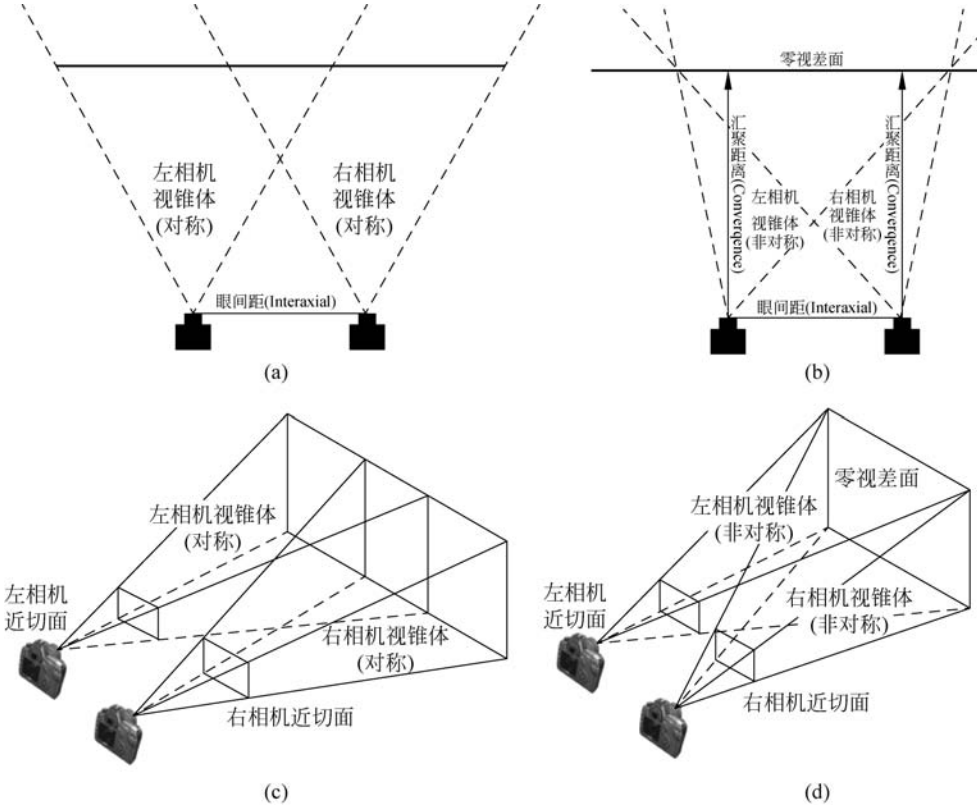

图 5.30　立体相机的视锥体结构

离(Far)共 6 个参数决定的，而在左、右两台相机组成立体相机时，相机的透视投影矩阵还与两台相机之间的间距(Interaxial)及汇聚距离(Convergence)两个参数相关，左、右相机的透视投影矩阵参数分别为[①]：

$$\text{左相机：}\begin{cases}\text{Top}=\text{Near}\times\tan\left(\dfrac{\text{Fov}}{2}\right)\\ \text{Bottom}=-\text{Near}\times\tan\left(\dfrac{\text{Fov}}{2}\right)\\ \text{Left}=-\text{Aspect}\times\text{Top}+\dfrac{\text{Near}\times\text{Interaxial}}{2\times\text{Convergence}}\\ \text{Right}=\text{Aspect}\times\text{Top}+\dfrac{\text{Near}\times\text{Interaxial}}{2\times\text{Convergence}}\end{cases}$$

$$\text{右相机：}\begin{cases}\text{Top}=\text{Near}\times\tan\left(\dfrac{\text{Fov}}{2}\right)\\ \text{Bottom}=-\text{Near}\times\tan\left(\dfrac{\text{Fov}}{2}\right)\\ \text{Left}=-\text{Aspect}\times\text{Top}-\dfrac{\text{Near}\times\text{Interaxial}}{2\times\text{Convergence}}\\ \text{Right}=\text{Aspect}\times\text{Top}-\dfrac{\text{Near}\times\text{Interaxial}}{2\times\text{Convergence}}\end{cases}$$

① 详细推导见附录 C。

其中,Fov 是相机的视角,Aspect 是显示设备的高宽比。

5.2.3 播放立体视频

在 VR 开发中,通过设置左、右相机可以实时渲染生成立体视频,立体视频中的每一帧包含左眼和右眼两幅图像,需要根据投影仪中不同的立体播放模式,编写立体视频播放代码。

在前面章节中介绍过目前常用立体投影技术分为偏振式立体投影与主动式立体投影两类,偏振式立体投影主要在电影院环境中使用,而主动式立体投影更适合于搭建 VR 环境。目前商用投影仪基本都具备主动立体投影功能,并支持帧序列(见图 5.31(a))、左右并列(见图 5.31(b))、上下并列(见图 5.31(c))三种播放模式。帧序列模式是指按照左—右—左—右…顺序依次播放左眼画面与右眼画面,图像刷新率为 120Hz,即每秒各播放 60 帧左眼画面与 60 帧右眼画面。但帧序列模式不支持高清分辨率投影。左右并列格式是指左、右眼图像水平并列成一个双倍宽度的画面,例如,对于 1080p 分辨率的立体视频,每一帧图像的分辨率为 3840×1080,并按 60Hz 的刷新率发送到投影仪,投影仪端自动将画面一分为二,按左—右—左—右…顺序播放,因此实际图像刷新率仍为 120Hz。上下并列格式与左右并列格式类似,只是左、右眼图像垂直排列成一个双倍高度的画面。目前,左右并列、上下并列两种模式都支持 1080p 分辨率的立体视频播放。

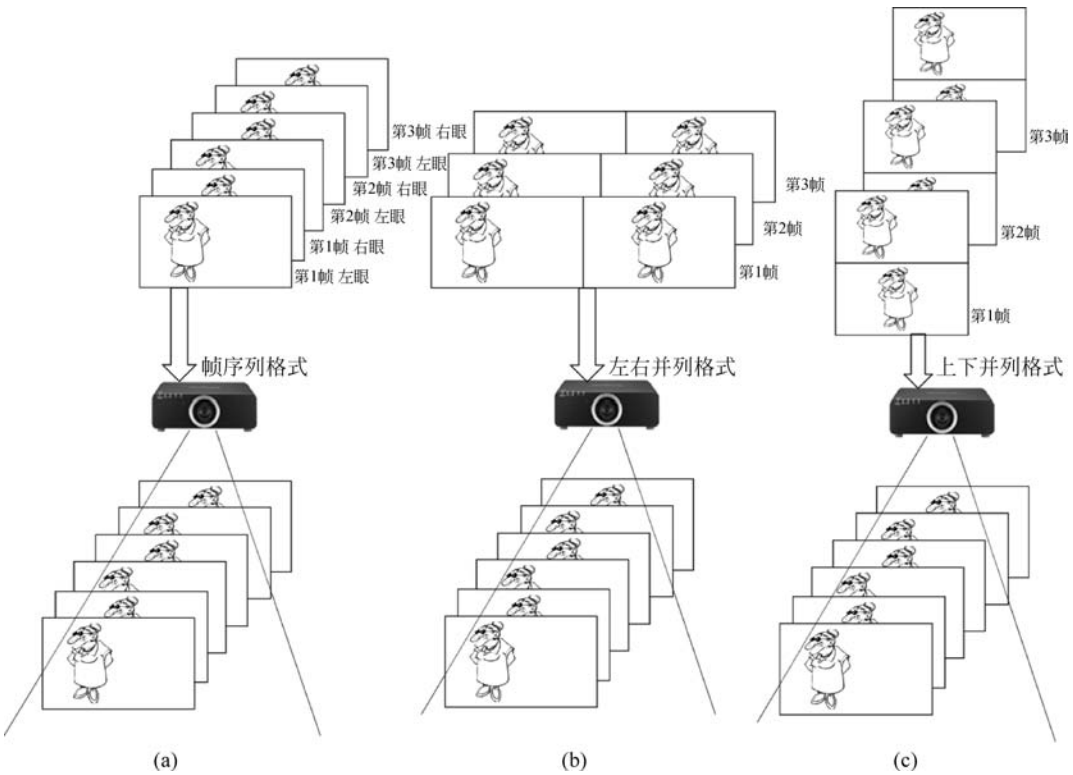

图 5.31 常用的三种立体视频格式比较

在 Unity3D 平台中,开发者可以根据需要自行编写上述三种模式的立体视频播放代码。下面给出 Unity3D 实现左右并列格式立体视频播放的实现方法与代码示例。

（1）**左右相机摆置**：为左右相机设置一个共同的父物体，父物体对应于虚拟角色双眼连线的中心位置，父物体跟随虚拟角色进行位移和旋转变换。左、右相机在父物体坐标系下，按眼间距参数在 x 轴水平方向分别向左和向右进行平移。父子层级设置保证从不同位置和不同视角观看（渲染）虚拟场景时，左、右眼相机始终保持固定的眼间距取值，父子层级结构的设置在 Hierarchy 栏中完成。

（2）**设置相机透视投影矩阵**：根据用户指定的眼间距与汇聚距离（Convergence）两个参数，按照 5.2.2 节中的公式，分别计算左、右相机的透视投影矩阵。在编写代码脚本时，眼间距与汇聚距离作为脚本的 public 型变量，可供用户自行指定设置，具体见本节的代码示例。

（3）**左右相机渲染控制**：由于 Unity3D 中默认场景中主相机自动进行渲染输出，因此在进行立体渲染时，就需要通过代码控制左、右相机交替进行渲染，分别生成每一帧的左、右眼图像。在 Unity3D 中可以通过调用 Camera.Render()函数控制指定相机完成一次渲染。

（4）**左右相机画面并列**：左、右相机渲染完成后，生成的左、右眼画面各自存放于显存中的 RenderTexture 区域中，可以通过调用 Graphics.CopyTexutre()函数将两幅画面复制到一个双倍宽度的 RenderTexture 区域中，形成左右并列格式立体画面，推送到投影仪端进行播放。这一过程可在 OnRenderImage()函数体中完成。该函数体每一帧被自动调用一次，用于在最终显示之前对相机渲染结果进行特效处理。

以左右并列格式为例，VR 立体视频的实时渲染与播放流程图如图 5.32 所示。

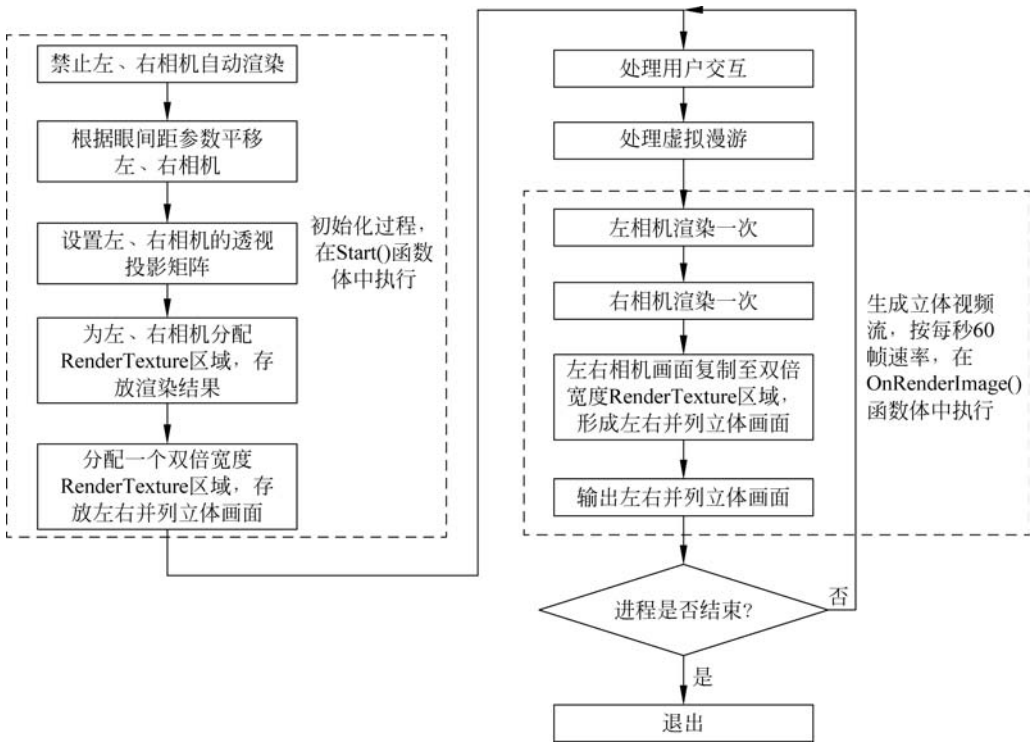

图 5.32　VR 立体视频的实时渲染与播放流程图（左右并列格式）

下面给出 Unity3D 中左右并列格式的立体视频实时渲染与播放的代码实例。对立体相机进行初始化设置的代码如下。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class _StereoCamera : MonoBehaviour
{
    //左右眼相机对象
    public Camera leftCamera;
    public Camera rightCamera;
    //左右眼相机的transform，用于根据眼间距来设置左右眼相机位置
    public Transform leftCamera_transform;
    public Transform rightCamera_transform;

    //左右眼相机的RenderTexture
    private RenderTexture leftCamera_texture;
    private RenderTexture rightCamera_texture;
    //双倍宽度的RenderTexture，用于左右相机画面并列
    private RenderTexture mixingCamera_texture;

    public float Interaxial = 0.06f;//左右眼视差
    public float Convergence = 6.0f;//汇聚距离(即相机到零视差面的距离)


    // Start is called before the first frame update
    void Start()
    {
        //立体相机的初始化工作
        //---------------------- (1) -----------------
        leftCamera.enabled = false;//禁止左相机自动渲染
        rightCamera.enabled = false;//禁止右相机自动渲染
        //---------------------- (2) -----------------
        //根据父物体位置及眼间距Interaixal，设置左右相机位置
        leftCamera_transform.position = transform.position + transform.TransformDirection(-Interaxial / 2, 0, 0);
        rightCamera_transform.position = transform.position + transform.TransformDirection(Interaxial / 2, 0, 0);
        //---------------------- (3) -----------------
        //根据眼间距Interaixal及零视差距离parallaxDist,设置左右相机的投影矩阵
        leftCamera.projectionMatrix = GetProjectionMatrix(leftCamera,true);
        rightCamera.projectionMatrix = GetProjectionMatrix(rightCamera,false);
        //---------------------- (4) -----------------
        //分配左眼相机对应的RenderTexutre
        leftCamera_texture = new RenderTexture(Screen.width, Screen.height, 24);
        //分配右眼相机对应的RenderTexutre
        rightCamera_texture = new RenderTexture(Screen.width, Screen.height, 24);
        //将左眼相机渲染目的指向左眼RenderTexture的入口，左眼相机渲染结果将存储于m_leftCamTexture
        leftCamera.targetTexture = leftCamera_texture;
        //将右眼相机渲染目的指向左眼RenderTexture的入口，右眼相机渲染结果将存储于m_RighttCamTexture
        rightCamera.targetTexture = rightCamera_texture;
        //---------------------- (5) -----------------
        //分配为双宽度的RenderTexture，用于将左、右眼图像形成左右并列格式进行输出显示
        mixingCamera_texture = new RenderTexture(Screen.width * 2, Screen.height, 24);
        //执行其他初始化工作
        //......
    }
```

在立体相机初始化过程中，通过调用函数体 GetProjectionMatrix()计算立体相机透视投影矩阵，该函数体代码如下。

```
//计算立体相机的投影矩阵
//输入1：左/右相机对象  输入2：bool变量，是否为左相机。
//输出：透视投影矩阵
Matrix4x4 GetProjectionMatrix(Camera cam,bool isLeftCam)
{
    //声明矩阵对象
    Matrix4x4 m = new Matrix4x4();
    //设置相机透视投影矩阵所需的6个参数
    float left,right,bottom,top,near,far;
    //相机的视角(弧度)
    float FOVrad;
    //相机的视角及宽高比
    float Aspect;
    //读取相机的视角并转化为弧度
    FOVrad = cam.fieldOfView / 180.0f * Mathf.PI;
    //读取相机的宽高比
    Aspect = cam.aspect;
    //读取相机的近切面距离
    near = cam.nearClipPlane;
    //读取相机的远切面距离
    far = cam.farClipPlane;
    //计算透视投影矩阵中的top与bottom参数
    top = near* Mathf.Tan(FOVrad * 0.5f);  bottom = -top;
    //计算左相机的left与right参数
    if (isLeftCam)
    {
```

```
        //立体相机的left与right参数与眼间距Interaxial及汇聚距离Convergence有关
        left = -top*Aspect + (Interaxial*near) / (2.0f * Convergence);
        right = top * Aspect + (Interaxial * near) / (2.0f * Convergence);
    }
    //计算右相机的left与right参数
    else
    {
        //立体相机的left与right参数与眼间距Interaxial及汇聚距离Convergence有关
        left = -top * Aspect - (Interaxial * near) / (2 * Convergence);
        right = top * Aspect - (Interaxial * near) / (2 * Convergence);
    }
    //根据相机的left,right,bottom,top,near,far共6个参数计算透视投影矩阵
    float x = (2.0f * near) / (right - left);float y = (2.0f * near) / (top - bottom);
    float a = (right + left) / (right - left);float b = (top + bottom) / (top - bottom);
    float c = -(far + near) / (far - near);float d = -(2.0f * far * near) / (far - near);
    //矩阵赋值
    m[0, 0] = x; m[0, 1] = 0; m[0, 2] = a; m[0, 3] = 0;
    m[1, 0] = 0; m[1, 1] = y; m[1, 2] = b; m[1, 3] = 0;
    m[2, 0] = 0; m[2, 1] = 0; m[2, 2] = c; m[2, 3] = d;
    m[3, 0] = 0; m[3, 1] = 0; m[3, 2] = -1.0f; m[3, 3] = 0;
    return m;//返回投影矩阵
}
```

控制立体相机实时渲染,生成立体视频流的代码如下。

```
void OnRenderImage(RenderTexture source, RenderTexture destination)
{
    //左眼相机渲染一次,生成当前帧的左眼图像
    leftCamera.Render();
    //右眼相机渲染一次，生成当前帧的右眼图像
    rightCamera.Render();
    //将左右眼相机渲染的结果，分别复制到一个双倍宽度的RenderTexture中
    //生成左右并列格式的立体画面，输出到显卡进行立体投影
    Graphics.CopyTexture(leftCamera_texture, 0, 0,
                         0, 0, Screen.width, Screen.height,
                         mixingCamera_texture, 0, 0, 0, 0); //复制左眼画面
    Graphics.CopyTexture(rightCamera_texture, 0, 0,
                         0, 0, Screen.width, Screen.height,
                         mixingCamera_texture, 0, 0, Screen.width, 0); //复制右眼画面
    //将双倍画幅的立体画面推送到输出显示端
    Graphics.Blit(mixingCamera_texture, destination);

}
```

根据上述代码实例的实现原理,读者也可以自己编写帧顺序及上下并列格式的立体视频播放代码。

5.3 HTC VIVE 开发环境

HTC VIVE 是目前常用的头戴式 VR 设备,不仅具有良好的沉浸式立体视觉体验,同时也为用户提供了良好的定位跟踪及交互功能。本节介绍基于 HTC VIVE 进行系统开发的基本过程,包括环境配置、头盔显示以及手柄开发等内容。

5.3.1 环境配置


5.3.1 节

1. HTC VIVE 的安装

可在官网上下载 VIVEPORT 按照安装教程进行 HTC VIVE 安装,下载过程中会安装 SteamVR。SteamVR 是 HTC VIVE 的运行插件,如图 5.33 所示。

也可以在计算机中安装 Steam 以及 SteamVR。首先从官网下载并安装 Steam(见图 5.34),

图 5.33 安装 HTC VIVE

安装完成后进入 Steam 的商店界面，搜索并下载 SteamVR。安装成功后，在 Steam 界面的右上角会有 VR 标志(见图 5.35)。在对 HTC VIVE 进行开发时启动 SteamVR 设备，第一次启动时设置 VR 设备的房间环境，按照房间设置的一步一步提示完成即可。

图 5.34 安装 Steam

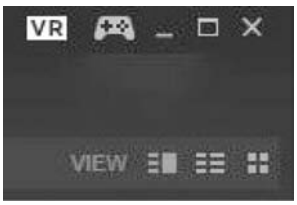

图 5.35 SteamVR 安装成功

2. Unity3D 中 SteamVR SDK 插件的下载与配置

在 Unity3D 中下载和配置 SteamVR SDK。SteamVR SDK 是一个由 VIVE 提供的官方库，以简化 VIVE 开发。在布置好的场景中，单击 Window→Asset Store(见图 5.36)，打开官网商店，搜索 SteamVR Plugin，下载即可(见图 5.37)；然后单击 Import 按钮，并单击编辑器中的 Accept All 按钮，如图 5.38 所示。

注意在使用 VIVE 调试时，先对项目进行设置，单击菜单栏中的 Edit→Project Settings→Player，找到 Other Settings，勾选 Virtual Reality Supported 复选框，查看 Virtual Reality SDKs 中是否有 OpenVR。如果没有，单击右下角的“+”进行添加，如图 5.39 所示。启用 OpenVR 后，即可运行虚拟场景。

3. Unity3D 中手柄开发插件 VIVE Input Utility 的下载

VIVE 手柄开发可使用多种方式多种插件，这里介绍 VIVE Input Utility 插件的使用。首先在 Asset Store 中下载并导入插件 VIVE Input Utility，如图 5.40 所示。VIVE Input

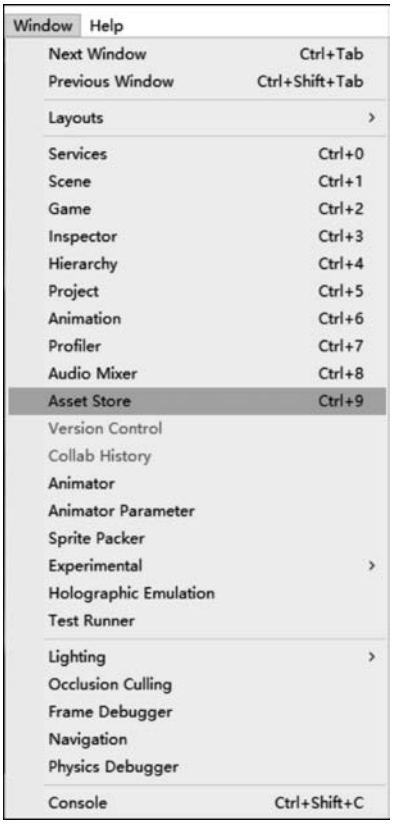

图 5.36　Asset Store

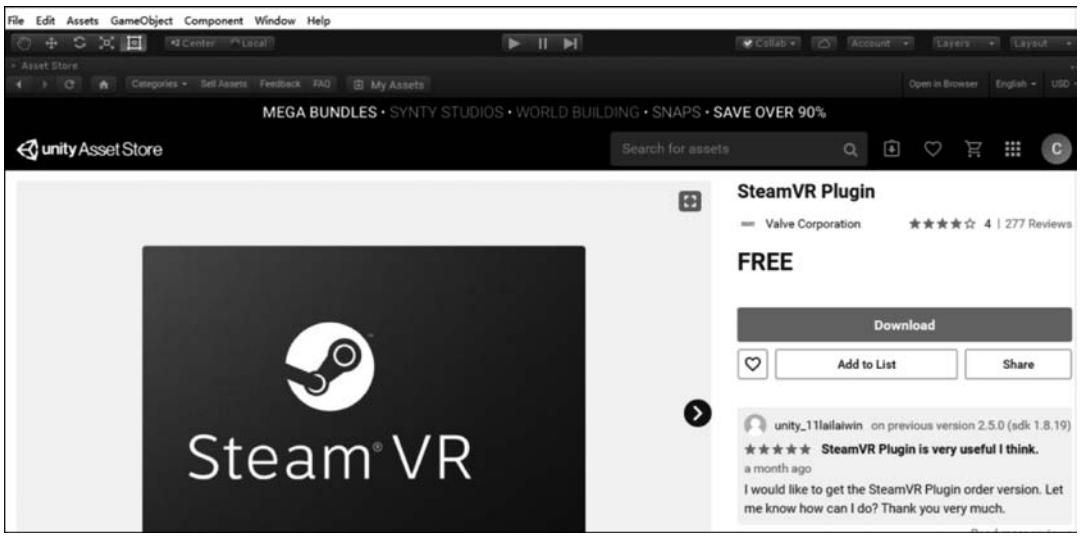

图 5.37　下载 SteamVR Plugin

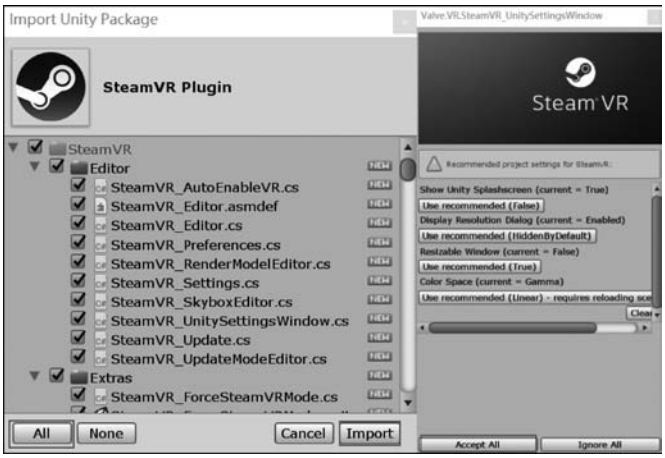

图 5.38 SteamVR Plugin 的导入以及编辑器设置

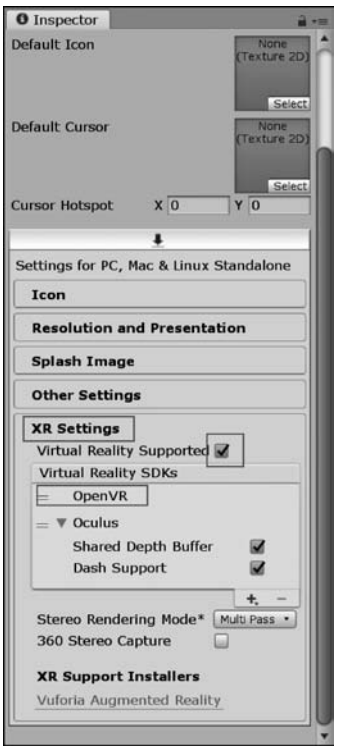

图 5.39 Unity3D 设置

Utility 是一个基于 SteamVR 插件的开发工具,使开发者更方便地控制 VIVE 设备。本案例中使用该插件简单有效地开发 VIVE 手柄。

4. 场景中虚拟相机以及手柄的设置

首先,由于 HTC VIEV 场景中需要的是 3D 相机,所以将场景中自动生成的 Main Camera 删除。然后选择 SteamVR/Prefabs/[CameraRig]预制体并添加到场景中(见图 5.41)。这个预制体是 SteamVR 提供的 3D 虚拟相机,与 VIVE 头盔直接绑定,可以直接在 HTC

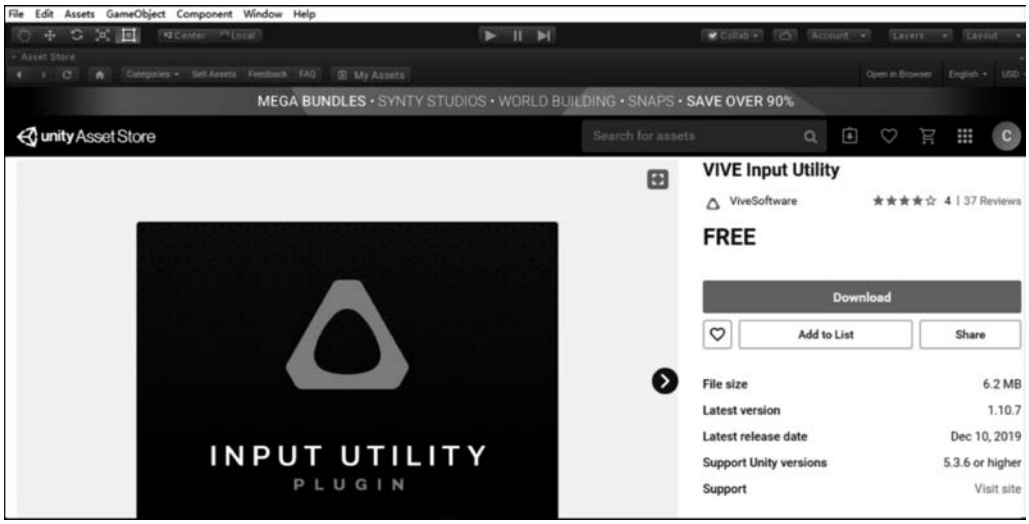

图 5.40　VIVE Input Utility 的下载

VIVE 中提供 3D 效果。为了方便后续设置，将它在场景中的位置设置为(0，0，0)。

图 5.41　CameraRig 预制体

然后将 VIVE Input Utility 插件中 Prefabs 中的 VivePointers 预制体(见图 5.42)添加到场景中，以备开发手柄功能使用。

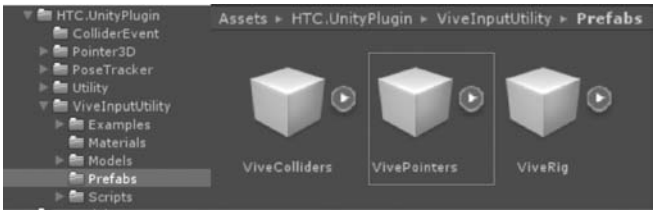

图 5.42　VivePointers 预制体

5.3.2　HTC VIVE 头盔

HTC VIVE 头盔通过精确地跟踪定位以及逼真地呈现虚拟场景，给玩家带来真实的、高品质的沉浸式体验。VIVE 头盔的跟踪定位是通过使用 HTC VIVE 定位器捕获头盔的位置信息与头部转向等信息，将信息映射到虚拟场景中，为虚拟场景中相应玩家的位置和动作提供数据，确定当前视场中的目标物、用户视点的位置和朝向，实现虚拟现实空间定位并提供浸入式体验。

HTC VIVE 头盔与虚拟相机[CameraRig]直接绑定，头盔移动时的位置和旋转角度等信

图 5.43 [CameraRig]的层次展开

息也会反映在虚拟相机上，所以头盔位置移动时相机位置也会随之移动。虚拟相机[CameraRig]可控制头盔的位置和旋转角度，包含的子物体如图 5.43 所示。其中，Controller(left)和 Controller(right)是左右手柄，对应玩家的左右手，Camera(head)是玩家的头部，Camera(eye)对应玩家的眼睛视角。给[CameraRig]添加相关脚本设置其位置以及角度，控制相机的位置和旋转角度，使玩家的 Head(即场景中视角)达到系统设定的 Transform 信息。可在 C# 脚本的 Update()函数中调用方法 transform.position 或者 GetComponent<Transform>().position 获取相机的位置，调用 transform.rotation 或者 GetComponent<Transform>().rotation 获取角度信息。

5.3.3 HTC VIVE 手柄交互

HTC VIVE 手柄(见图 5.44)可以像鼠标一样选中某个物体，也可以对物体进行抓取，也可以根据系统需要进行交互语义定义。一套 VIVE 设备中有两个手柄，分左右，开发的时候也是分左右的。每个手柄上面有一个圆盘和 4 个按钮。

图 5.44 HTC VIVE 手柄

系统按钮：用来打开手柄。这个按钮不可以开发(默认)。在游戏中按下该按钮是调出系统默认的菜单，用来关闭和切换游戏用。

menu 按钮：默认用来打开游戏菜单。

grip 按钮：每个手柄有两个 grip 按钮，左右侧各有一个。

trigger 按钮：扳机按钮，用的最多，可以有力度等级区别。

pad：触摸屏＋鼠标的功能，可触摸，可点击。

后面三种是开发时常用的。

1. 按钮开发

首先，需要在 C# 脚本中进行引用，具体代码为 using HTC.UnityPlugin.Vive。然后在该脚本的 Update()代码段中调用相关按钮的相关方法，实现特定的功能。每个按钮都有 GetPress(按住时一直返回 true)、GetPressDown(按下时触发事件)、GetPressUp(放开时触发事件)三种方法，用 HandRole 枚举来确定左右手柄，用 ControllerButton 枚举来确定是哪个按钮。具体实现代码如图 5.45 所示。

其中，trigger 按钮除了上述常见方法，还可以通过 GetTriggerValue 方法获得其模拟值 triggervalue，范围是 0～1，不同数值代表不同程度的按压，具体数值对应如表 5.1 所示。

pad 有接触、按下两组方法，可返回事件点的位置等信息，如 GetPadTouchAxis 是返回接触点的位置信息，常用到的有六种方法：GetPadTouchAxis，GetPadTouchDelta，GetPadTouchVector，GetPadToPressAxis，GetPadToPressDelta，GetPadToPressVector。

其中，Axis 是坐标位置，Delta 是最后一帧移动位置，Vector 是移动的向量。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using HTC.UnityPlugin.Vive;
0 个引用
public class fff : MonoBehaviour {
    // Update is called once per frame
    0 个引用
    void Update () {
        if(ViveInput.GetPressUp(HandRole.RightHand, ControllerButton.Trigger))
        {
            Debug.Log
        }
    }
}
```

GetPress / GetPressDown / GetPressDownEx / GetPressDownEx<> / GetPressEx / GetPressEx<> / GetPressUp / GetPressUpEx / GetPressUpEx<>

LeftHand / RightHand

FullTrigger / Grip / HairTrigger / Menu / Pad / PadTouch / Trigger

图 5.45　手柄按钮开发实现

表 5.1　triggervalue 数值对应按压力度

triggervalue＝0	没按	无
triggervalue＝0.1～0.2	轻按	HairTrigger
triggervalue＞0.5	中度按	Trigger
triggervalue＝1	全部按下	FullTrigger

2. UGUI 开发

在场景中添加预制体[CameraRig]和[VivePointers]后，新建一个 UI 按钮，如图 5.46 所示。

图 5.46　添加 Button

禁用 Canvas 对象下的两个脚本，并设置模式为 World Space，为 Canvas 添加 Canvas Raycast Target 脚本，如图 5.47 所示。

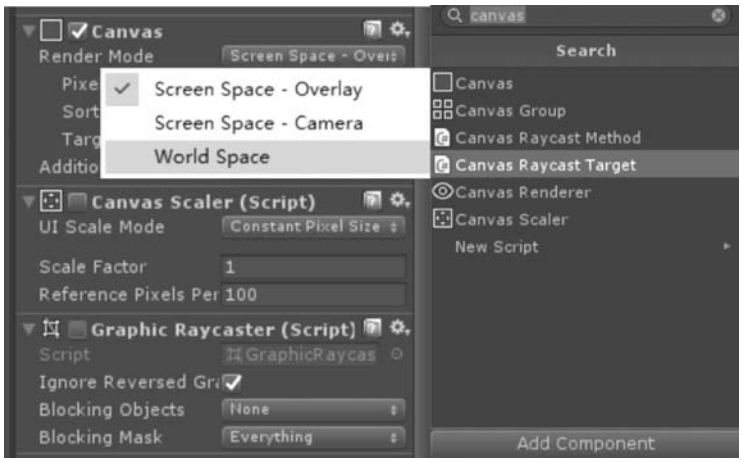

图 5.47　设置 Canvas 并添加脚本

将 Canvas 和 Button 调整至合适的大小和位置后，运行场景。运行以后，手柄会发出射线，当射线照射到按钮时，会有一个黄色的球，如图 5.48 所示。此时按 Trigger 按钮，就可以实现单击按钮的动作，按钮触发的具体事件可自行定义。

3. 通过射线远距离拖动物体

手柄射线照射到远处的 3D 物体时，可以通过 Trigger 按钮抓住物体并拖动。同样是在场景中添加预制体[CameraRig]和[VivePointers]后，新建一个 3D 物体，如 Cube，为其添加脚本 Draggable，若不能自动添加 Rigidbody 刚体组件，便手动添加上，如图 5.49 所示。

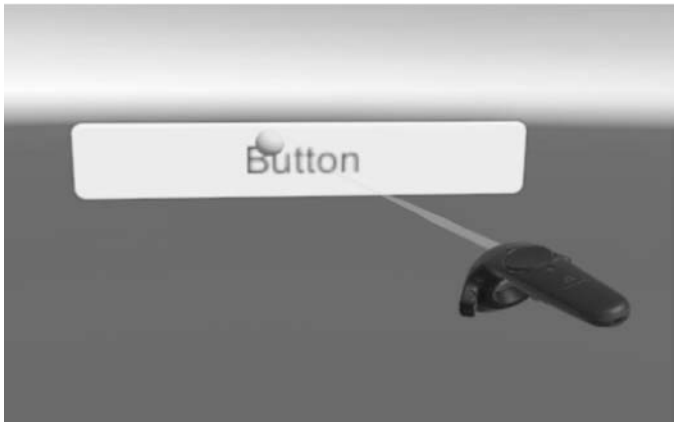

图 5.48　手柄点击按钮场景实例

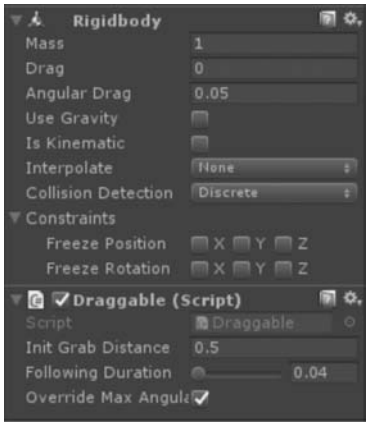

图 5.49　为手柄点击按钮添加脚本

4. 触碰和拾取

在场景中添加预制体[CameraRig]和[VivePointers]后，新建一个 3D 对象，默认可以被触碰，再为其添加 Rigidbody 组件和 Basic Grabbable，如图 5.50 所示，则该对象可以被拾取。

在 3D 物体上添加脚本 Material Changer，可自行设置该物体在被触碰和拾取时的效果，设置参数如图 5.51 所示。

图 5.50 为可拖曳物体添加刚体组件和拖曳脚本

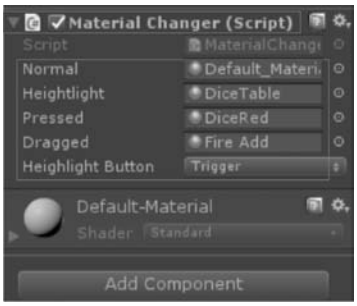

图 5.51 Material Changer 脚本的参数设置

其中，Normal 是默认贴图，Heightlight 是触碰后的贴图，Pressed 是按下按钮时的贴图，Dragged 是拖曳时的贴图，Heightlight Button 是指定的可响应按钮，默认是 Trigger。实例如图 5.52 所示。

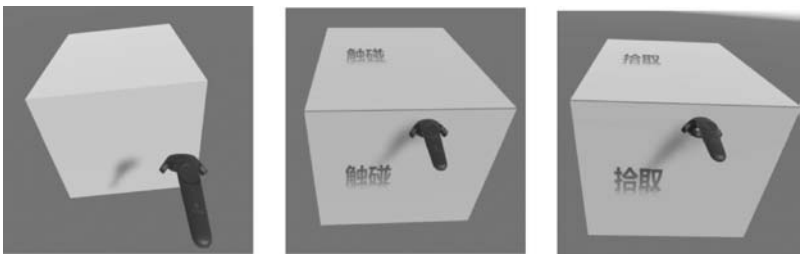

图 5.52 HTC VIVE 手柄触碰和拾取物体实例

5.4 HoloLens 开发环境

HoloLens 允许开发者在 Unity3D 中创建自己的混合现实应用程序，并部署在 HoloLens 中运行。本节将介绍如何在 Unity3D 中开发 HoloLens 应用程序并使其运行在 HoloLens 上。

5.4.1 环境配置

1. 安装 Visual Studio

从官网下载 Visual Studio，安装时需要勾选“通用 Windows 平台开发”“使用 Unity3D 的游戏开发”“使用 C++的游戏开发”以及“Visual Studio 扩展开发”。如图 5.53 所示为安装界面右侧安装组件的详细信息。使用 Unity3D 的游戏开发有一个可选的 Unity3D 编辑器，由于 HoloLens 官方建议使用 LTS 版本的 Unity3D 编辑器，因此此处不勾选。

图 5.53 Visual Studio 2017 安装详细信息

安装 Unity3D

2. 安装 Unity3D

下载 HoloLens 官方建议的 LST 版本，安装时需要勾选支持 UWP 平台打包的组件，如图 5.54 所示。

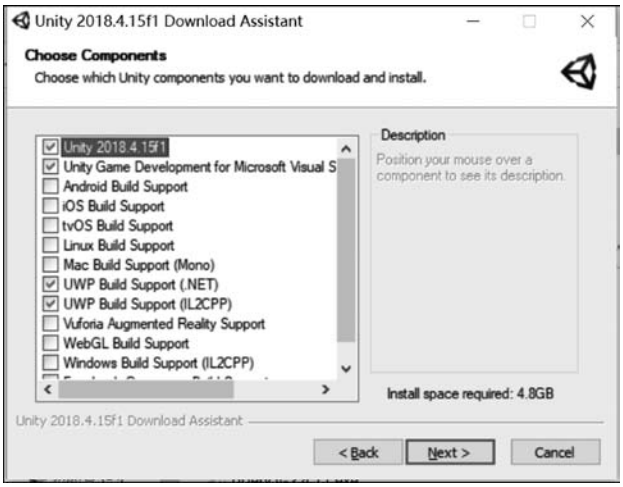

图 5.54 Unity3D 安装过程中的组件选择

3. 安装模拟器(可选)

微软提供了 HoloLens 模拟器，允许开发人员在 PC 上测试 HoloLens 应用程序。模拟器的运行对于 PC 的硬件配置有一定要求，要求如下。

(1) 操作系统：Windows 10 专业版、企业版或教育版。

(2) CPU：4 核及以上 64 位。

(3) 内存：8GB 及以上。

(4) GPU：支持 DirectX 11.0 或以上，驱动为 WDDM 1.2 及以上。

其安装过程如下。

1) Windows 10 启用开发人员模式

Windows 10 系统中启用开发人员模式的过程如图 5.55 所示：在“设置”窗口中选择“更新和安全”→“开发者选项”，选中“开发人员模式”单选按钮。

2) 设置虚拟化

如图 5.56 所示，重启计算机打开 BISO 界面，找到虚拟化选项并打开。

3) 启用 Hyper-V

选择“控制面板”→“程序”→“程序和功能”→“启用或关闭 Windows 功能”，勾选 Hyper-V 及其子项前面的复选框，如图 5.57 所示。

4) HoloLens 模拟器安装

下载 HoloLens 模拟器，安装。模拟器运行效果如图 5.58 所示。

4. 引入 MixedRealityToolkit

MixedRealityToolkit(MRTK)是微软提供的混合现实开发工具包，旨在加速针对 Microsoft HoloLens 和 Windows 混合现实沉浸式设备应用程序的开发。其源码在 GitHub 上开放。

图 5.55 Windows 中设置开发人员模式

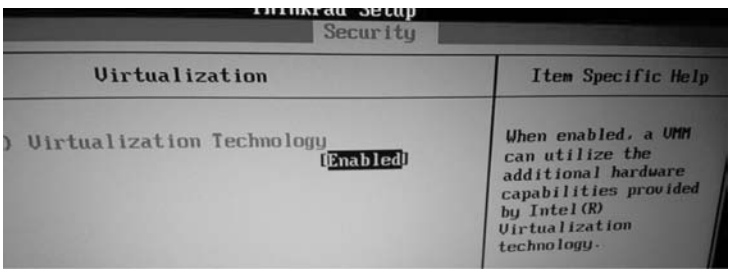

图 5.56 BIOS 开启虚拟化

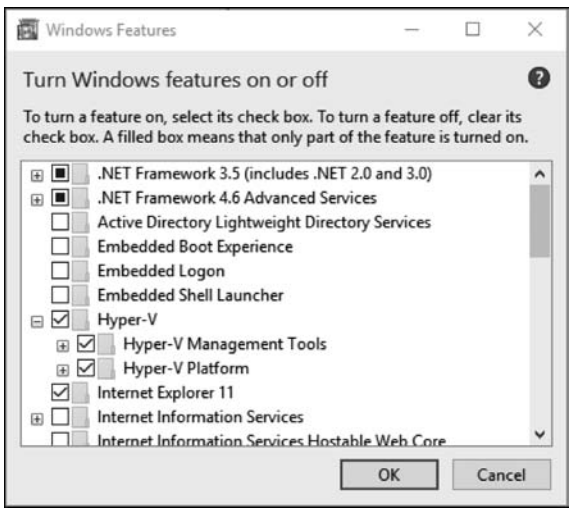

图 5.57 启用 Hyper-V

图 5.58 HoloLens 模拟器运行界面

由于 MRTK 基于 Windows 10 SDK 18362+,因此需要先下载安装相应的 Windows 10 SDK。在 Unity3D 中引入 MRTK 的步骤如下。

(1) GitHub 下载 MixedRealityToolkit-Unity。网址 https://github.com/microsoft/MixedRealityToolkit-Unity/releases 提供了 MRTK 的各种发布版本。目前已经更新到 v2.3.0。其中,Microsoft.Mixed Reality.Toolkit.Unity.Foundation.2.3.0.unitypackage 必选,其他包可选,如图 5.59 所示。

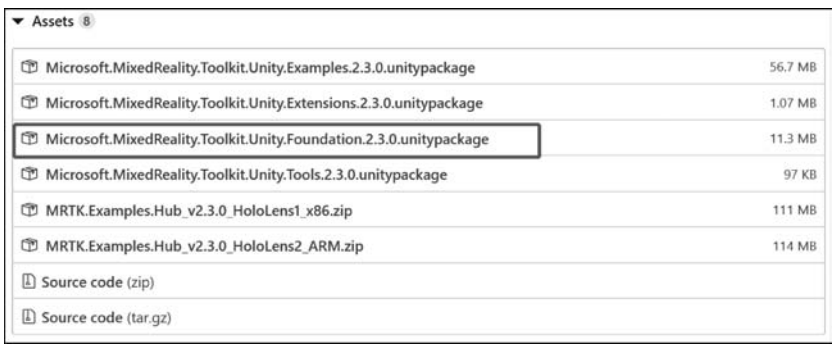

图 5.59 MixedRealityToolkit-Unity 资源包下载页

(2) 资源包导入 Unity3D。如图 5.60 所示,在 Unity3D 中新建工程,选择 Assets→Import Package→Custom Package 命令,然后浏览到资源包下载位置,选中之后进行导入。导入完成后弹出 MRTK 配置选项,单击 Apply 按钮。

导入成功后,Assets 目录结构如图 5.61 所示。


5.4.2 节

5.4.2 开发实例

本节介绍如何在 Unity3D 中构建简单的 HoloLens 混合现实应用实例。

1. MR 场景及配置

新建场景,在编辑器导航栏选择 Mixed Reality Toolkit→Add to Scene and Configure 命令,如图 5.62 所示。

图 5.60　MRTK 导入过程

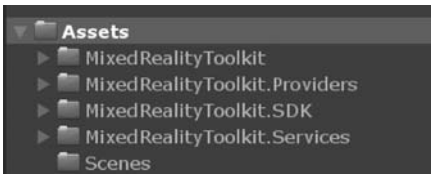

图 5.61　MRTK 导入后的文件结构

图 5.62 场景中添加 Mixed Reality Toolkit 配置

该操作完成后场景中包含以下物体,如图 5.63 所示,主相机成为 MixedRealityPlayspace 的子物体。

图 5.63 场景配置结果

在场景中简单放置一个 Cube 用于测试,位置在设备正前方 1m 处,大小为 25cm×25cm×25cm,分别绕 x,y,z 轴旋转 45°,如图 5.64 所示。

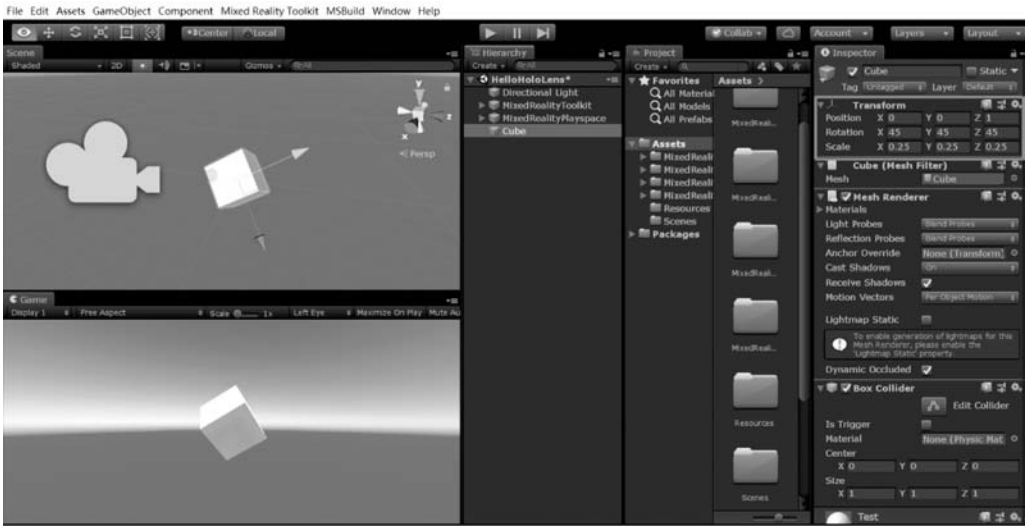

图 5.64 场景中全息物体的设置

2. 工程打包

如图 5.65 所示,选择 File→Build Settings 命令,将目标平台改为 UWP,选中 Universal Windows Platform 后单击 Switch Platform 按钮。单击 Add Open Scenes 按钮添加场景,然后单击 Player Settings 按钮进行项目配置。

在 PlayerSettings 中输入 Company Name、Product Name 和 Version,其中 Product Name 为 HoloLens 中显示的应用程序的名称,如图 5.66 所示。

单击 Build Settings 中的 Build 按钮,在弹出的文件资源管理器中新建文件夹,将程序打包在该文件夹中,如图 5.67 所示。

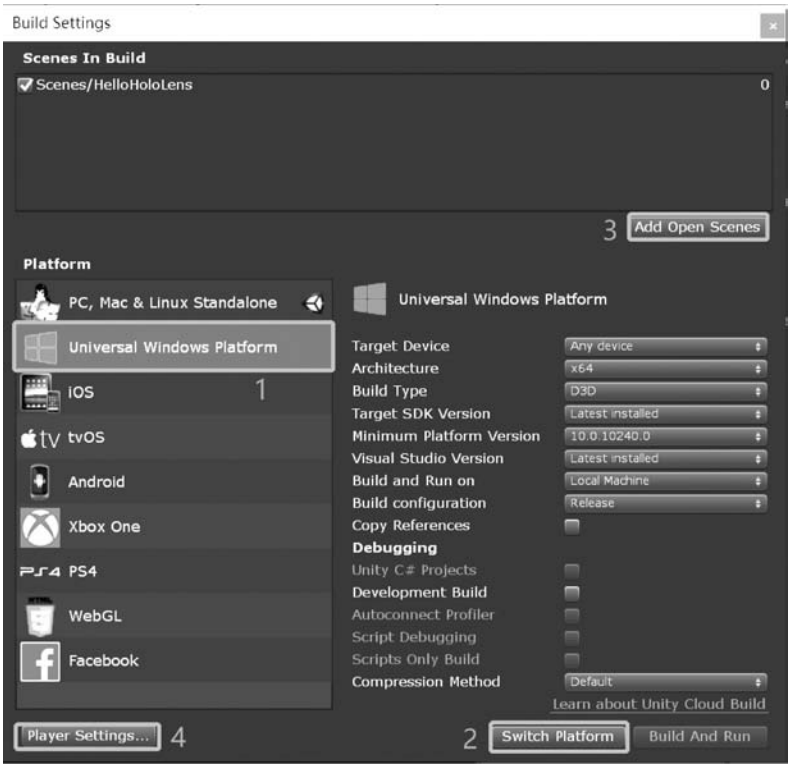

图 5.65 打包平台配置

图 5.66 Play Settings 中项目设置

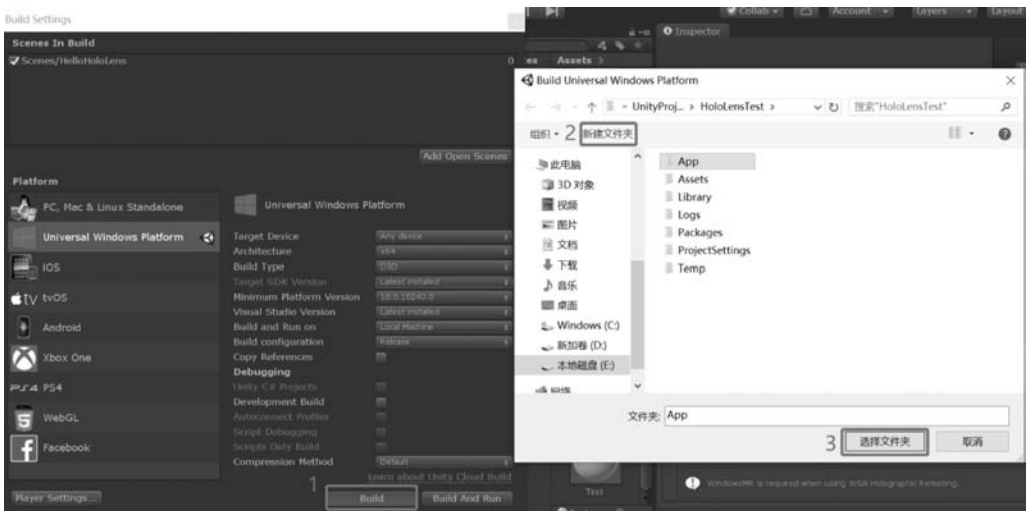

图 5.67 新建文件夹保存打包文件

3. 应用程序部署

要将应用程序部署在 HoloLens 上，首先需要在 HoloLens 中打开开发者模式，打开过程类似于在 Windows 10 中打开开发者模式。程序打包完成后，用 VS 打开后缀为 sln 的文件，编译平台选择 Release x86，如图 5.68 所示。开发人员可以选择将程序部署在 HoloLens 真机或者 HoloLens 模拟器上。

图 5.68 VS 中调试选项

(1) 若在真机中调试，调试器选择 Device。真机调试状态下，HoloLens 通过 USB 连接到 PC，在第一次进行部署时需要将 PC 和 HoloLens 进行配对。如图 5.69 所示，在 HoloLens 开发者选项中单击 Pair 按钮，会显示一个六位数字 PIN 码，单击 VS 中的 Device 开始部署程序，部署过程中按提示输入 PIN，如图 5.70 所示，即可将程序部署在 HoloLens 中。

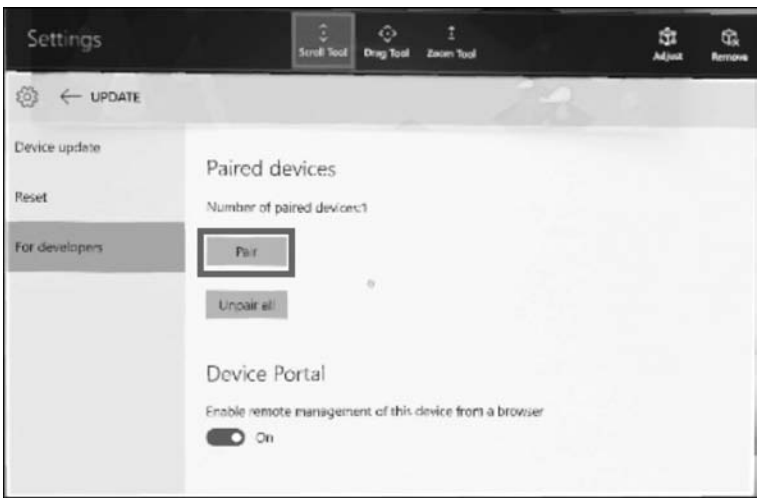

图 5.69 HoloLens 中配对

图 5.70 VS 中提示输入 PIN

HoloLens 中运行效果如图 5.71 所示。

(2) 若在 HoloLens 模拟器中调试,调试器选择 HoloLens Emulator。运行结果如图 5.72 所示。

图 5.71 HoloLens 真机中的运行效果

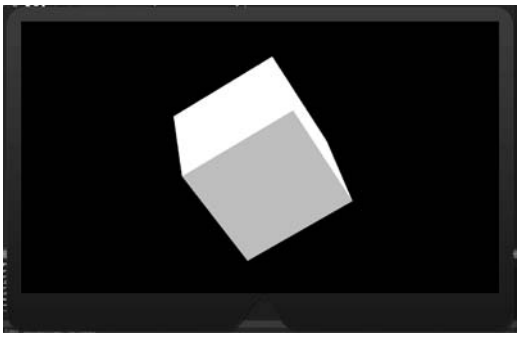

图 5.72 HoloLens 模拟器中的运行效果

5.4.3 交互实现

5.4.3 节

本节介绍如何在 Unity3D 中使用 MRTK 进行 HoloLens 的交互操作开发。MRTK 对于不同的交互方式提供了不同的接口,接口定义在 Microsoft. MixedReality. Toolkit. Input 命名空间中。开发过程中只需实现这些接口即可进行不同交互方式的开发。以下将介绍 HoloLens 的三种基本的交互操作开发,分别是凝视、手势和语音交互。如 5.4.2 节所述,建立场景,在场景中放置一个 cube 作为交互对象,在场景中放置 3D Text 用于显示交互状态。

1. 凝视

凝视功能由 IMixedRealityFocusHandler 接口提供。本节实现凝视点进入和离开物体,物体颜色改变的效果,使用文字显示凝视点状态。图 5.73 为凝视交互的实现代码。首先,引入 Microsoft. MixedReality. Toolkit. Input 命名空间,然后实现 IMixedRealityFocusHandler 接口中 OnFocusEnter()和 OnFocusExit()两个函数。函数中为设置 cube 颜色和 3D Text 文本的代码。

```
using Microsoft.MixedReality.Toolkit.Input;
using UnityEngine;

0 个引用
public class GazeTest : MonoBehaviour,IMixedRealityFocusHandler
{
    public TextMesh GazeState;
    13 个引用
    public void OnFocusEnter(FocusEventData eventData)
    {
        GetComponent<MeshRenderer>().material.color = Color.red;
        GazeState.text = "凝视点进入";
    }

    13 个引用
    public void OnFocusExit(FocusEventData eventData)
    {
        GetComponent<MeshRenderer>().material.color = Color.white;
        GazeState.text = "凝视点退出";
    }
}
```

图 5.73 凝视交互的代码

图 5.74 为 HoloLens 模拟器中的运行效果。

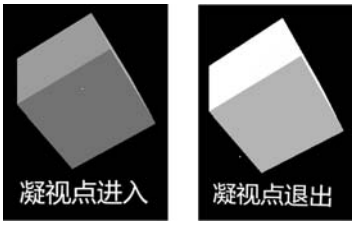

图 5.74 模拟器中的运行效果

2. 手势

手势功能由 IMixedRealityInputHandler 接口提供。本节实现用户凝视物体并执行 Airtap 手势时物体颜色改变,手势执行结束后恢复初始颜色,文字显示用户交互状态。图 5.75 为手势交互实现的代码。首先,引入 Microsoft. MixedReality. Toolkit. Input 命名空间,然后实现 IMixedRealityInputHandler 接口中 OnInputUp()和 OnInputDown()两个函数。函数中为设置 cube 颜色和 3D Text 文本的代码。

```
using Microsoft.MixedReality.Toolkit.Input;
using UnityEngine;

0 个引用
public class GestureTest : MonoBehaviour,IMixedRealityInputHandler
{
    public TextMesh GestureState;

    16 个引用
    public void OnInputDown(InputEventData eventData)
    {
        GetComponent<MeshRenderer>().material.color = Color.red;
        GestureState.text = "用户点击物体";
    }

    16 个引用
    public void OnInputUp(InputEventData eventData)
    {
        GetComponent<MeshRenderer>().material.color = Color.white;
        GestureState.text = "点击结束";
    }
}
```

图 5.75 手势交互的代码

图 5.76 为 HoloLens 模拟器中的运行效果。

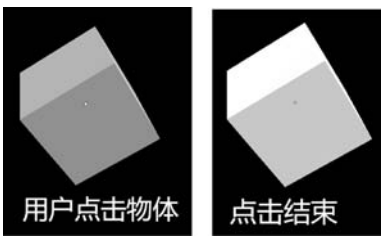

图 5.76 HoloLens 模拟器中的运行效果

3. 语音

语音交互实现过程如下。

1) 语音指令设置

选中场景中 MixedRealityToolkit 物体,在物体上的 MRTK 组件中选择 Input 选项。在 Input 选项下配置 Speech 交互方式,单击 Add a New Speech Command 按钮,输入新的语音指令。如图 5.77 所示,此处以改变物体颜色为例,加入 Change Color 指令。

2) 绑定语音交互事件

加入指令后,在 cube 上添加 Speech Input Handler 脚本,添加语音指令,并绑定触发事件,如图 5.78 所示。

触发事件在脚本中定义,并绑定在场景中的物体上,图 5.79 为触发事件的代码,代码控制物体颜色改变和交互状态的改变。

图 5.80 为 HoloLens 模拟器中的运行效果。

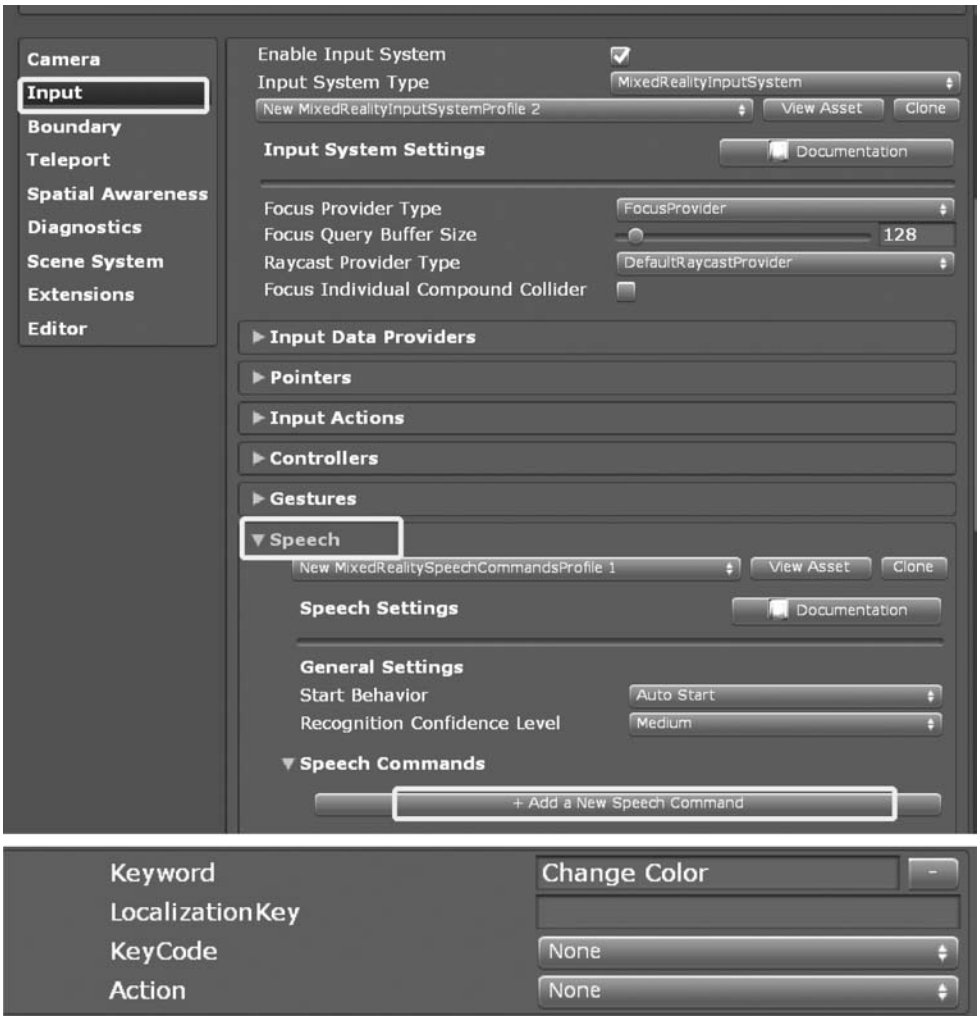

图 5.77 语音指令设置

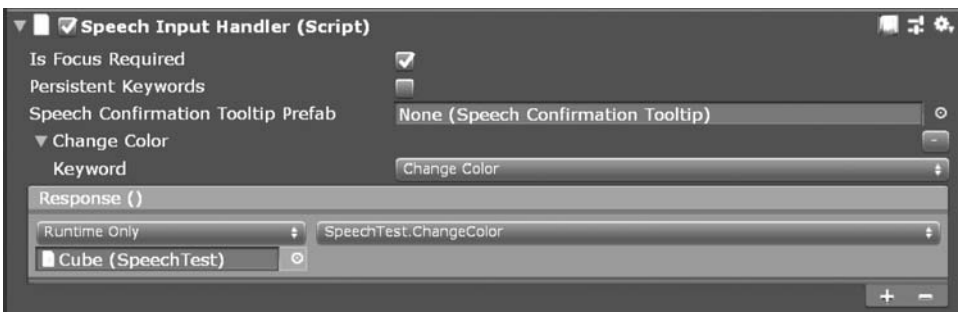

图 5.78 语音交互事件绑定

```
using UnityEngine;

0 个引用
public class SpeechTest : MonoBehaviour
{
    public TextMesh SpeechState;

    // Start is called before the first frame update
    0 个引用
    public void ChangeColor()
    {
        GetComponent<MeshRenderer>().material.color = Color.red;
        SpeechState.text = "语音控制颜色改变";
    }
}
```

图 5.79　语音交互事件代码

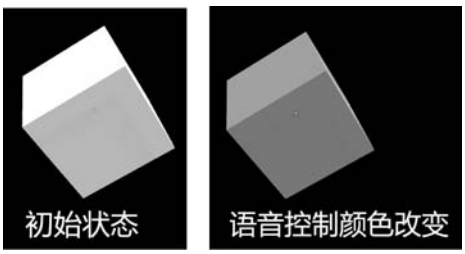

图 5.80　HoloLens 模拟器中的运行效果

习题

1. 用 Unity3D 进行编程,实现一个基于 HTC VIVE 的赛车 VR 游戏。

2. 用 Unity3D 进行编程,实现一个基于 HoloLens 的混合现实系统,对房间中的汽车进行观察。

3. 分别编写 Unity3D 代码,实现帧顺序与上下并列格式的立体视频播放。