第 3 章

AIGC 增强的游戏交互与动画

本章讨论关于游戏交互的一些内容，主要包括输入控制与动画资源。在游戏中通过输入设备将命令或信息输入，然后用一个动画效果作为视觉反馈可以构成一个基础的交互方式。

电子游戏能够快速地响应玩家的输入，比如常见的玩家在游戏终端上通过输入来控制主角行走、跳跃或攻击等操作，而且能够实时获得反馈，这种交互行为是频繁发生的。在 Unity 中提供了 Input 类用于监听控制事件，包括鼠标操作事件、键盘事件、游戏摇杆事件、移动平台上的触屏事件等。本书在前面章节中介绍过 Unity 支持的平台非常多，而不同平台的输入设备也各不相同，相应的控制方式也都不同。由于篇幅的限制，本章不可能将所有的输入控制事件都详细地一一叙述，因此本章以 PC 平台下的输入操作为例讲解输入和控制的基础事件。

在游戏中除了需要能够对输入控制事件进行监听，还需要能够对监听到的操作事件进行合适的反馈。例如，当玩家操作主角行走时，除了让主角的坐标改变，还需要播放一个行走的动画作为视觉反馈；或当主角受到攻击时，除了主角的血量发生变化，往往还伴随着受到攻击的动画效果。类似的动画效果视觉反馈在游戏中非常丰富，这样的输入控制和动画反馈构成游戏的基础交互效果。

3.1　游戏脚本与 AIGC 优化

游戏脚本在游戏的开发中起着至关重要的作用。游戏中逻辑层面的操作几乎都是由脚本来实现的，小到控制游戏的背景音乐的音量大小、对角色的种种控制等，大到敌人的 AI 逻辑等，可以说游戏脚本是组成游戏的骨架，脚本的好坏直接关系着游戏的质量。在第 1 章中，本书虽然对 Unity 的脚本有一些基础的交代，但还远远不足以进行游戏的开发，因此在本节中，将对游戏脚本进行更为详细的介绍。

3.1.1　创建脚本

游戏与动画电影类似，都是由大量静止画面组成的，这样的静止画面被称为“帧（frame）”，不同的是，动画和电影的每一帧都是提前制作好的，而游戏则要根据玩家的操作实时地制作出每一帧画面，这被称为实时渲染（real-time rendering）技术。而游戏脚本，其中很大一部分作用就是要规定每一帧的逻辑规则，这个说法并不严谨，但是能够大概表达出脚本的作用。

根据每个开发者的开发习惯不同，其在脚本程序设计上也略有区别，主要有两个倾向：一种是将游戏中大部分逻辑放到一个脚本中，另一种是按照每个游戏对象的不同功能分别编写脚本并绑定在相应的游戏对象上。这两种方式各有优劣，第一种方式便于对脚本进行管理，但大部分逻辑编写在同一个脚本中，会导致程序过于冗长。第二种方式将逻辑过度细分，会造成后期管理不便。所以，用户要根据两种方式的优缺点综合使用，扬长避短、切合实际。

在第 1 章中，已经对如何创建脚本做了简单的介绍，这里不再赘述。当脚本创建好之后会自动分配给该脚本一个默认命名 NewBehaviourScript，然而根据脚本的具体功能不同且为了便于管理，需要根据脚本的功能进行重命名，这时可以通过快捷键 F2 或在脚本上右击，选择 Rename 命令进行重命名。但需要注意的是，为了代码的可读性和团队协作，强烈建议脚本名与脚本中的类名（class name）保持一致，如图 3-1 所示。

在 Unity 2020.3 版本之前，如脚本文件名与类名不同，则会提示错误信息 Can’t add script，如图 3-2 所示。此时需要修改脚本名或双击打开脚本，在脚本编辑器中对类名进行修改。所以，当给脚本重命名之后一定要检查文件名与脚本中的类名是否一致。

从 Unity 2020.3 版本开始，Unity 允许脚本文件名和类名不一致了。现在 Unity 编辑器可以通过解析 C# 代码，自动识别文件中定义的 public 类，并将其与 .meta 文件关联。可以在 Inspector 中正常看到并使用该脚本组件，即使类名与文件名不同。虽然技术上允许不一致，但为了代码的可读性和团队协作，强烈建议文件名与主类名保持一致，这是 C# 社区和 Unity 社区的通用约定。

除此之外，目前 Unity 只会识别 public 类。如果类是 internal 或没有访问修饰符，可能无法在编辑器中正确显示。如果一个脚本文件中定义了多个 public 类，Unity 会将第一个 public 类作为该脚本组件的类型，因此建议一个文件只定义一个 public 类，避免混淆。

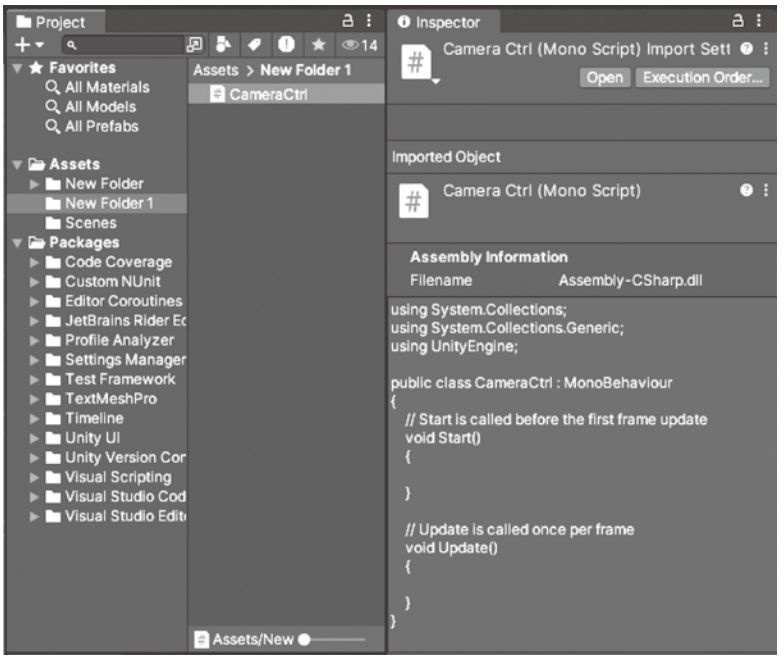

图 3-1
脚本名与脚本中的类名需保持一致

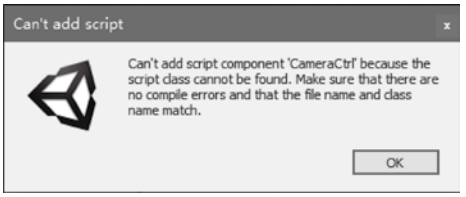

图 3-2
错误信息无法添加脚本到对象

每次新建一个 C# 脚本，脚本中都会自动地生成一些基础的代码，具体如下：

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

public class CameraCtrl : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

代码顶部使用 using 关键字引入了三个命名空间，分别是 System.Collections、System.Collections.Generic 和 UnityEngine。前两个命名空间中定义了 C# 的一些基础类和接口，如

果读者对 C# 有使用经验，那么对前两个命名空间应该比较熟悉。UnityEngine 命名空间则是属于 Unity 引擎的，里面定义了许多引擎中的类、对象和接口。这三个命名空间并不是全部，只是其中定义了许多在开发中非常常用的一些类，所以每当创建一个脚本时都会自动生成这些内容。

引入命名空间后则是声明一个类，在本例中，类名 CameraCtrl 是 MonoBehaviour 类的派生类（子类）。在 Unity 中的所有脚本都要继承此类。此类中包含下方已经自动生成的 Start() 方法和 Update() 方法，此外，还包含其他开发中的常用方法，具体的名称和解释如表 3-1 所示。

表 3-1　Unity 脚本中的常用方法

方法名	解　　释
Start	当脚本被激活（脚本已经绑定在对象上并勾选了脚本）时，仅在开始执行第一帧前调用一次，并且要在所有脚本的 Awake() 方法都执行完毕后才会被调用
Awake	与 Start() 方法类似，仅在开始执行第一帧前调用一次；不同的是，即便脚本没有被激活，只要绑定在游戏对象上也会被调用
Update	当脚本被激活时，每一帧都会被调用，是实现各种游戏行为最常用的函数
LateUpdate	当所有脚本中的 Update 函数都被调用后，该函数才会被调用。此方法可用于调整脚本执行顺序。例如，让物体在 Update() 方法里实现位移，而跟随物体的摄影机可以在 LateUpdate() 方法里实现位移
FixedUpdate	此方法用作处理物理效果，与 Update() 方法类似，但不是每帧调用一次，而是以固定的时间间隔进行调用
OnGUI	此方法用于处理 GUI 事件，与 Update() 方法类似，程序将会在每一帧被调用
OnEnable	当脚本被激活时调用该方法，不能用于协同程序
OnDisable	当脚本变为不可用或非激活状态时调用该方法，不能用于协同程序

表 3-1 只列举了少部分常用方法，除此之外，Unity 脚本中的常用方法还有很多，比如用于检测碰撞的 OnCollisionEnter() 等三个方法，以及用于显示界面 UI 的 OnGUI() 等若干个方法，在此无法一一详解，建议读者参考 Unity 官方提供的 Scripting API 帮助文档。

可以通过一个小例子让读者体验一下每种方法的不同——打开一个新的 Unity 项目，并创建一个 C# 脚本，输入以下代码：

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

public class Test : MonoBehaviour {
    // 首先被打印出一次
    void Awake() {
        print("唤醒");
    }
    // 脚本开始执行
```

```
    void Start () {
        print(" 开始 ");
    }
    // 每帧打印一行
    void Update () {
        print(" 执行 ");
    }
}
```

将脚本关联到场景中的 MainCamera 上，当然也可以关联到场景中任意“活动的”对象上，然后运行项目进行测试，能够看到在 Console 视窗中将会打印出消息行。

在本例中，可以看到“唤醒”方法的消息最先在 Console 视窗中被打印出一行后，“开始”消息被打印出来，之后“执行”消息在程序停止运行前的每一帧都会被打印出一行。以上列举的其他方法也可以以同样的方式进行测试。

3.1.2 使用 AIGC 辅助编程工具创建脚本

通义灵码针对 Unity 开发场景做了专项优化适配，全面支持 Unity 核心开发语言 C# 脚本的生成、调试与场景化代码优化，还能深度联动 Unity 原生 API，例如 Transform、Rigidbody 组件逻辑及 UI 交互等核心功能模块。

整理好上一小节的脚本需求后，在通义灵码的输入窗口中输入提示词：“帮我写一个 Unity 脚本，命名为 Test，包含 Awake()、Start()、Update() 三个生命周期方法，分别在各方法中打印“唤醒”开始“执行”。输入完成后，通义灵码会快速返回响应及专属脚本框，脚本框右上角设有三个功能按钮，从左至右依次为“插入”“复制”“创建新文件”。单击“创建新文件”按钮，即可在解决方案中自动新增一个 NewFile.cs 脚本，后续需手动修改脚本名称，并将其拖曳至 Unity 项目的 Assets 文件夹内。

3.2 键盘事件

使用键盘操作游戏是 PC 平台游戏的主要操作方式，如控制角色移动、跳跃、技能释放等，也可以使用其他一些快捷键。而要实现键盘对游戏的实时控制，则需要对键盘事件进行监听。根据游戏中需要的操作不同，要监听的事件也不同，主要有按下事件、抬起事件和长按事件，在本节中将详细介绍这三类事件。

3.2.1 按下事件

按下事件是游戏中最为常见的键盘事件，Unity 提供用于判断键盘被按下的布尔类型 Input.GetKeyDown() 方法。按键值作为参数传递到方法中，一旦按键被按下，方法将返回 True 值，否则返回 False 值。另外，由于监听按键是否被按下，需要游戏在开始运行时就

不间断地判断按键是否被按下，所以此方法需要放在脚本的 Update() 方法中。在游戏运行中使用条件判断语句需要对每一帧都进行判断，代码如下：

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

public class InputTestKeyDown : MonoBehaviour {
    void Start () {

    }
    // 每帧执行一遍 Update() 方法
    void Update () {
        // 当监听到按键被按下时打印信息
        if (Input.GetKeyDown(KeyCode.A)) {
            print("按下了 A 键 ");
        }
        if (Input.GetKeyDown(KeyCode.B)) {
            print("按下了 B 键 ");
        }
    }
}
```

将脚本保存后，将其绑定到任意游戏对象上执行程序。在程序执行时，当按下键盘上的 A 键或 B 键，就会在 Console 视窗中打印相应的信息，如图 3-3 所示。

图 3-3
根据键盘按下事件打印提示信息

3.2.2　抬起事件

所谓抬起事件，是指当键盘按键按下时并不触发事件，而是在抬起时触发事件，Unity 提供了 Input.GetKeyUp() 方法用来处理这一类型的操作。与按下事件方法一致，参数为按键值，按键抬起时触发事件方法返回 True 值，否则返回 False 值，这需要在 Update() 方法中调用，代码如下：

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

public class InputTestKeyUp : MonoBehaviour {
   void Start () {

   }
   void Update () {
       // 当监测到按键抬起时打印信息
```

```
            if (Input.GetKeyUp(KeyCode.LeftControl)) {
                print("抬起左 Ctrl 键");
            }
            if (Input.GetKeyUp(KeyCode.RightAlt)) {
                print("抬起右 Alt 键");
            }
            if (Input.GetKeyUp(KeyCode.Space)) {
                print("抬起空格键");
            }
        }
    }
```

将编写后的脚本保存后绑定到游戏对象上，执行后获得如图 3-4 所示的输出。

图 3-4
根据键盘抬起事件打印提示信息

3.2.3 长按事件

在游戏中，偶尔会用到长按某一按键的操作情况，比如要实现足球游戏中射门按键的蓄力功能，就需要通过长按事件对按键持续按下的状态进行监听，Unity 为此提供了 Input.GetKey() 方法来判断键盘是否持续处于按下状态。此方法与前文介绍的按下事件、抬起事件类似，不同的是，此方法最好配合抬起事件作为整个长按操作的结束，具体在脚本中的使用请参考以下代码：

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

public class InputTestLongPress : MonoBehaviour {
    // 设定激活帧数
    int limitFrame = 5;
    // 声明帧数自增变量
    int pressFrame = 0;
    void Start () {
    }
    void Update () {
        // 当按键持续按下时被激活
        if (Input.GetKey(KeyCode.W)) {
```

```
            // 计算按下的帧数
            pressFrame++;
            // 当按下帧数大于规定激活帧数时打印文字
            if (pressFrame > limitFrame) {
                print(" 长按激活 ");
            }
        }
        // 当按键抬起时帧数自增变量归零并将取值打印
        if (Input.GetKeyUp(KeyCode.W)) {
            pressFrame = 0;
            print("pressFrame:" + pressFrame);
        }
    }
}
```

当按键被持续按下后，变量 pressFrame 会在每帧进行一次自加 1 的运算，直到 pressFrame 的取值大于规定的帧数值，将持续打印出“长按激活”的文字信息。然后在按键抬起时，将变量 pressFrame 的取值归零，以便用于下次长按打印信息的操作，并且将 pressFrame 的当前取值打印出来，这表示一次长按操作结束。程序运行效果如图 3-5 所示。

图 3-5
根据键盘长按事件打印提示信息

3.2.4　按任意键事件

“请按任意键继续游戏……”是游戏中的常见提示，通常表示地图或资源加载完毕后按任意键继续进行游戏的操作，为此 Unity 也提供了两个布尔类型的属性，即 Input.anyKeyDown 和 Input.anyKey。两者的区别在于前者是按下任意键触发，后者是长按任意键触发。使用的代码如下：

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

public class InputTestAnyKey : MonoBehaviour {
    // 按下任意键帧数
    int anyKeyDownFrame = 0;
    // 长按任意键帧数
    int anyKeyFrame = 0;
    void Start () {
    }
    void Update () {
        // 当有任意键被按下 anyKeyDownFrame 开始自加 1 并打印
```

```
            if (Input.anyKeyDown) {
                anyKeyDownFrame++;
                print("按下任意键帧数：" + anyKeyDownFrame);
            }
            // 当有任意键被长按 anyKeyFrame 开始自加 1 并打印
            if (Input.anyKey) {
                anyKeyFrame++;
                print(“长按任意键帧数：" + anyKeyFrame);
            }
        }
    }
```

执行程序后用户会发现，每次按下任意键，Input.anyKeyDown 的取值都为 True，否则取值为 False，而当取值为 True 时，anyKeyDownFrame 变量都会自加 1。而长按任意键，则会使变量 anyKeyFrame 每帧自加 1，如图 3-6 所示。

图 3-6
按下和长按任意键打印提示信息

3.3 鼠标事件

在 PC 游戏中，鼠标的操作起着举足轻重的作用，有些游戏甚至不需要使用键盘，只需鼠标就能够完成所有操作，而且目前绝大部分 PC 游戏的鼠标使用率都是非常高的。常见的鼠标操作有单击、滑动滚轮、右击等，Unity 引擎能够监听这些操作，也提供了相对应的事件响应方法。在本节中着重介绍鼠标单击的三个事件响应方法。

3.3.1 按下事件

在常见的 FPS 射击类游戏中，用户瞄准目标后单击发射出子弹的这类操作通常是通过鼠标按下事件的响应实现的。在 Unity 中提供布尔类型 Input.GetMouseButtonDown() 方法用来判断鼠标是否被按下，如果被按下，则返回 True 值，否则，返回 False。在此方法中，数字参数用来判断按下的具体是哪一个鼠标按键，填入参数 0，表示判断鼠标左键是否被按下；填入参数 1，表示判断鼠标右键是否被按下；填入参数 2，表示判断鼠标中键是否被按下。

具体使用方法如下代码：

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

public class MouseInputTest : MonoBehaviour {
    void Start () {
    }
    void Update()
    {
        if (Input.GetMouseButtonDown(0))
            print("按下鼠标左键");
        if (Input.GetMouseButtonDown(1))
            print("按下鼠标右键");
        if (Input.GetMouseButtonDown(2))
            print("按下鼠标中键");
    }
}
```

将脚本关联到场景中的任意游戏对象上并执行，将鼠标悬停在 Game 视窗中分别单击鼠标左键、右键、中键，输出结果会在 Console 视窗中打印出来，如图 3-7 所示。

图 3-7
按下鼠标左键、右键、中键后的打印信息

3.3.2　抬起事件

与按下事件类似，Unity 也提供了 Input.GetMouseButtonUp() 方法用来判断鼠标按键按下后是否抬起，方法参数与鼠标按下事件完全一样：当抬起鼠标按键时，方法返回 True 值，否则返回 False。具体使用方法如下：

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

public class MouseInputTest : MonoBehaviour {
    void Start () {
    }
    void Update ()
    {
        if (Input.GetMouseButtonUp(0))
```