第 3 章 相机控制

本章介绍各类型视角控制相机脚本的实现，包括第一人称视角、第三人称视角和自由控制类型视角，自由控制类型视角又分为观察者视角和漫游视角。除此之外，本章还将介绍虚拟摄像系统 Cinemachine，并通过示例展示虚拟相机如何与 Timeline 配合使用制作镜头动画。

3.1 第一人称类型相机

FPS 类型游戏中的视角是典型的第一人称视角，它模拟人眼真实的观察效果，用户可以自由地控制镜头运动和视角变化。通常情况下，在第一人称视角中，鼠标光标会被锁定到游戏窗口的中心并隐藏，使其不可见，该功能需要通过 Cursor 类中的 visible 和 lockState 两个静态变量实现，代码如下：

```
//第 3 章/FirstPersonCameraController.cs

//初始化时隐藏鼠标光标
[SerializeField] private bool hideCursorOnStart =true;

private void Start()
{
    if (hideCursorOnStart)
    {
        Cursor.visible=false;
        Cursor.lockState=CursorLockMode.Locked;
    }
}
```

其中，visible 用于设置鼠标光标是否可见，lockState 为枚举类型，包含 3 个枚举值：None 为默认状态；Locked 表示光标被锁定在游戏窗口的中心；Confined 表示光标被限制在视图中，无法移出游戏窗口。

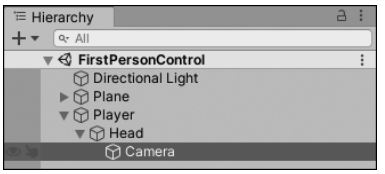


图 3-1　Player 与 Camera 的层级关系

在第一人称视角中，相机是随 Player(用户人物角色)移动的，通常情况下会将相机设为 Player 的子物体，如图 3-1 所示，无须在代码中控制其坐标值，只需通过代码控制相机的旋转值，使用户实现镜头操控自由。

实时获取鼠标在水平、垂直方向上的输入，视角跟随鼠标移动方向进行转动。实时获取输入的代码可以写在 MonoBehaviour 的生命周期函数 Update()中，需要注意的是，该函数每帧被调用一次，而由于不同时间或不同设备的性能不同，该函数每秒被调用的次数也会不同，因此为了保证匀速变化，需要将获取的输入值乘以 Time 类中的静态变量 deltaTime，该变量表示上一帧到当前帧的时间间隔，以秒为单位，Update()执行的次数越少，该时间间隔对应的值也就越大。除此之外，为了灵活地控制移动的灵敏度，通常还会乘以表示灵敏度的变量，代码如下：

```
//第 3 章/FirstPersonCameraController.cs

//灵敏度
[SerializeField, Range(1f, 10f)]
private float sensitivity=3f;

private void Update()
{
    float horizontal=Input.GetAxis("Mouse X")
        * Time.deltaTime * 100f * sensitivity;
    float vertical=Input.GetAxis("Mouse Y")
        * Time.deltaTime * 100f * sensitivity;
}
```

相机在水平方向上的角度不受任何限制，而在垂直方向上的角度需要有最大值、最小值的限制，防止出现镜头翻转的现象，如图 3-2 所示。

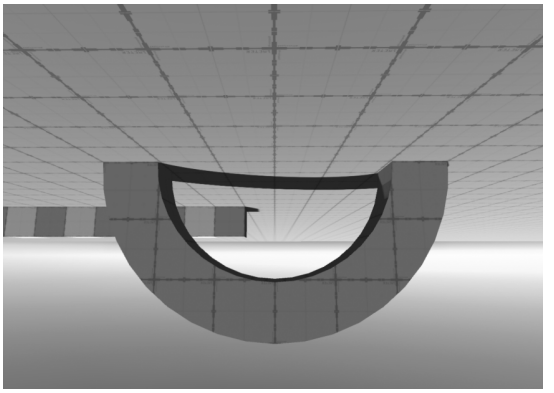


图 3-2　镜头翻转

使用两个 float 类型变量 rotY、rotX 处理鼠标在水平方向和垂直方向上的角度值，形成的欧拉角(rotX，rotY，0f)便是相机的目标角度值，最终通过 Quaternion 类中的静态函数 Euler()将该欧拉角转换为四元数，代码如下：

```
//第 3 章/FirstPersonCameraController.cs

//垂直方向角度最小值限制
[SerializeField, Range(-80f,-10f)]
private float rotXMinLimit=-40f;
//垂直方向角度最大值限制
[SerializeField, Range(10f, 80f)]
private float rotXMaxLimit=70f;
private float rotX, rotY;

private void Update()
{
    //...
    rotY+=horizontal;
    rotX-=vertical;
    rotX=Mathf.Clamp(rotX, rotXMinLimit, rotXMaxLimit);
    Quaternion targetRotation=Quaternion.Euler(rotX, rotY, 0f);
    transform.rotation=targetRotation;
}
```

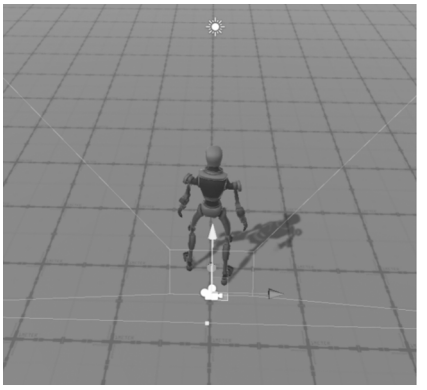
3.2　第三人称类型相机

在第三人称视角中，相机跟随人物角色进行移动，人物角色的驱动控制在 Update()函数中实现，而相机跟随的逻辑通常写在 LateUpdate()函数中，该函数晚于 Update()函数执行，这样可以确保相机的运动始终在人物运动之后执行。

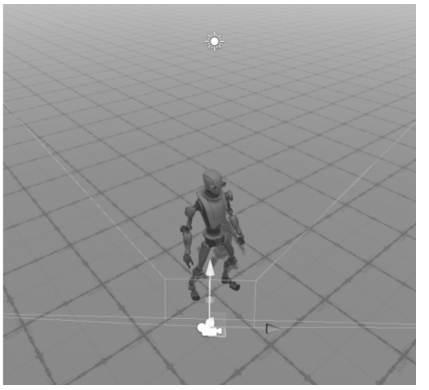
相机在水平方向上的朝向可以与角色的朝向保持一致，如图 3-3(a)所示，也可以由用户自由控制朝向，观察角色的不同方位，如图 3-3(b)所示，下面介绍这两种视角控制的实现方式。

3.2.1　通过角色朝向控制视角

如果想要让相机在水平方向上的朝向与角色朝向保持一致，则不需要获取鼠标在水平方向上的输入，只需根据人物角色的欧拉角设置相机的欧拉角。如果允许相机在垂直方向上进行一定的旋转，与第一人称类型相机一致，则需要对最大值和最小值进行限制，代码如下：



（a）朝向人物角色的前方



（b）自由控制朝向

图 3-3　相机的朝向

```
//第 3 章/ ThirdPersonCameraController.cs

//垂直方向角度最小值限制
[SerializeField, Range(-80f,-10f)]
private float rotXMinLimit=-40f;
//垂直方向角度最大值限制
[SerializeField, Range(10f, 80f)]
private float rotXMaxLimit=70f;
private float rotX;

private void LateUpdate()
{
    //鼠标右键按下时旋转视角
    if (Input.GetMouseButton(1))
    {
        vertical=Input.GetAxis("Mouse Y")
            * Time.deltaTime * 100f * sensitivity;
        rotX-=vertical;
        rotX=Mathf.Clamp(rotX, rotXMinLimit, rotXMaxLimit);
    }
    Quaternion targetRotation=Quaternion.Euler(
        rotX, avatar.eulerAngles.y, 0f); //y 值取人物角色的欧拉角 y 值
}
```

为旋转添加平滑过渡的过程，使用 Quaternion 类中的插值函数 Lerp()，根据旋转速度向目标值进行插值运算，代码如下：

```
targetRotation=Quaternion.Lerp(transform.rotation,
```

```
    targetRotation, Time.deltaTime * rotationSpeed);
```

假设通过鼠标滚轮可以调整相机与人物角色之间的距离，那么需要获取鼠标滚轮的滚动值，并动态地调整控制距离的变量，但是距离不能无限制地拉近或拉远，同样应该由最大值和最小值进行限制。

通常情况下，鼠标滚轮向上滚动时，相机向前方移动，鼠标滚轮向下滚动时，相机向后方移动，也可以通过一个 bool 类型的变量，控制是否反转移动的方向，当其为 true 时，为输入值乘以－1，代码如下：

```
//第 3 章/ ThirdPersonCameraController.cs

//鼠标滚轮灵敏度
[SerializeField]
private float scollSensitivity=2f;
//是否反转滚动方向
[SerializeField]
private bool invertScrollDirection;
//最小距离
[SerializeField]
private float minDistanceLimit=2f;
//最大距离
[SerializeField]
private float maxDistanceLimit=5f;
//当前距离
private float currentDistance=2f;
//目标距离
private float targetDistance;

private void Start()
{
    currentDistance=Mathf.Clamp(currentDistance,
        minDistanceLimit, maxDistanceLimit);
    targetDistance=currentDistance;
}
private void LateUpdate()
{
    //...
    //鼠标滚轮滚动时改变距离
    currentDistance-=Input.GetAxis("Mouse ScrollWheel")
        * Time.deltaTime * 100f * scollSensitivity
        * (invertScrollDirection ? -1f : 1f);
```

```
        //距离限制
        currentDistance=Mathf.Clamp(currentDistance,
            minDistanceLimit, maxDistanceLimit);
        //插值
        targetDistance=Mathf.Lerp(targetDistance,
            currentDistance, Time.deltaTime);
    }
```

相机的位置根据目标旋转值向后方平移目标距离，并加上人物角色的坐标值得到。由于人物角色的轴心点在底部，因此再加入一个控制高度的变量，代码如下：

```
//第 3 章/ ThirdPersonCameraController.cs

//高度
[SerializeField, Range(1f, 5f)]
private float height=2f;

private void LateUpdate()
{
    //...
    //目标位置
    Vector3 targetPosition=targetRotation * Vector3.back
        * targetDistance +avatar.position +Vector3.up * height;
    //赋值
    transform.rotation=targetRotation;
    transform.position=targetPosition;
}
```

3.2.2 通过用户输入控制视角

在 3.2.1 节中实现了通过角色朝向控制的第三人称视角，如果想要通过用户输入控制视角，自由地观察人物角色的各方位，则需要在此基础上获取鼠标在水平方向上的输入，将输入值应用于欧拉角 y 值。增加一个 bool 类型变量，用于表示是否由角色朝向控制相机水平方向上的朝向，当其为 true 时，使用人物角色的欧拉角 y 值；当其为 false 时，根据输入值设置欧拉角 y 值，代码如下：

```
//第 3 章/ ThirdPersonCameraController.cs

//...
private float rotY;
[SerializeField] private bool backOfAvatar;
```

```
private void LateUpdate()
{
    //当鼠标右键按下时旋转视角
    if (Input.GetMouseButton(1))
    {
        float horizontal=Input.GetAxis("Mouse X")
            * Time.deltaTime * 100f * sensitivity;
        float vertical=Input.GetAxis("Mouse Y")
            * Time.deltaTime * 100f * sensitivity;
        rotY+=horizontal;
        rotX-=vertical;
        rotX=Mathf.Clamp(rotX, rotXMinLimit, rotXMaxLimit);
    }
    Quaternion targetRotation=Quaternion.Euler(
        rotX, backOfAvatar ? avatar.eulerAngles.y : rotY, 0f);
    //...
}
```

接下来实现这样一个功能，当相机的视角根据用户输入进行旋转时，人物角色的头部同步进行转动，如图 3-4 所示。

头部转动的角度需要进行限制，避免出现头部转动到身后等怪异现象，如图 3-5 所示。当角度达到限制值时，可以自动回正头部朝向，不再跟随相机视角。

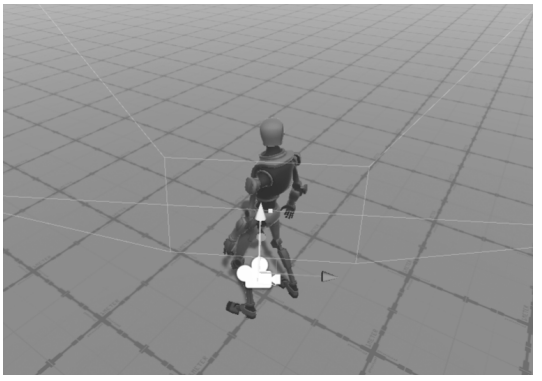


图 3-4　角色头部随相机视角转动

图 3-5　角色头部转动到身后

相机坐标加相机前方一定单位获得的点是头部需要看向的点，该点减去角色头部坐标点形成看向的方向，根据该方向可以求得目标角度值，判断角度值是否在限制值范围内。

由于旋转是 360°的，以 180°为例，当角度为－180°时与朝向和角度为 180°时是一样的，因此需要一种方法进行调整。当角度大于 180°时，将其减去 360°，当角度小于－180°时，将其加上 360°，这样使角度的取值范围始终为[－180，180]，代码如下：

```
//第 3 章/HeadTracker.cs

private Camera mainCamera; //主相机
private Transform head; //头部
[SerializeField] private Animator animator;
[Tooltip("水平方向上的角度限制"), SerializeField]
private Vector2 horizontalAngleLimit=new Vector2(-70f, 70f);
[Tooltip("垂直方向上的角度限制"), SerializeField]
private Vector2 verticalAngleLimit=new Vector2(-60f, 60f);

private void Start()
{
    mainCamera=Camera.main !=null
        ?Camera.main : FindObjectOfType<Camera>();
    head=animator.GetBoneTransform(HumanBodyBones.Head);
}
//获取看向的位置
private Vector3 GetLookAtPosition()
{
    //相机前方一定单位的位置
    Vector3 position=mainCamera.transform.position
        +mainCamera.transform.forward * 100f;
    //看向的方向
    Quaternion lookRotation=Quaternion.LookRotation(
        position-head.position, animator.transform.up);
    Vector3 angle=lookRotation.eulerAngles
        -animator.transform.eulerAngles;
    float x=NormalizeAngle(angle.x);
    float y=NormalizeAngle(angle.y);
    //是否在限制值范围内
    bool isInRange=x>=verticalAngleLimit.x && x<=verticalAngleLimit.y
        && y>=horizontalAngleLimit.x && y<=horizontalAngleLimit.y;
    return isInRange ?position
        : (head.position +animator.transform.forward);
}
//角度标准化
private float NormalizeAngle(float angle)
{
    if (angle>180) angle-=360f;
    else if (angle<-180) angle+=360f;
    return angle;
}
```

获得看向的位置后，在 LateUpdate()函数中让头部看向该位置，代码如下：

```
//第 3 章/HeadTracker.cs

private float angleX;
private float angleY;
[Tooltip("插值速度"), SerializeField]
private float lerpSpeed=5f;

private void LateUpdate()
{
    LookAtPosition(GetLookAtPosition());
}
//<summary>
//看向某点
//</summary>
//<param name="position">看向的点</param>
public void LookAtPosition(Vector3 position)
{
    Quaternion lookRotation=Quaternion.LookRotation(
        position-head.position);
    Vector3 eulerAngles=lookRotation.eulerAngles
        -animator.transform.rotation.eulerAngles;
    float x=NormalizeAngle(eulerAngles.x);
    float y=NormalizeAngle(eulerAngles.y);
    angleX=Mathf.Clamp(Mathf.Lerp(angleX, x,
        Time.deltaTime * lerpSpeed),
        verticalAngleLimit.x, verticalAngleLimit.y);
    angleY=Mathf.Clamp(Mathf.Lerp(angleY, y,
        Time.deltaTime * lerpSpeed),
        horizontalAngleLimit.x, horizontalAngleLimit.y);
    Quaternion rotY=Quaternion.AngleAxis(
        angleY, head.InverseTransformDirection(animator.transform.up));
    head.rotation *=rotY;
    Quaternion rotX=Quaternion.AngleAxis(
        angleX, head.InverseTransformDirection(
            animator.transform.TransformDirection(Vector3.right)));
    head.rotation *=rotX;
}
```

3.3 自由控制类型相机

本节介绍两种自由类型相机控制脚本的实现，分别为观察者视角控制和漫游视角控制，二者的核心区别在于，前者在全局坐标系中进行前、后、左、右及上、下方向的移动，视角围绕与地面的交点旋转，后者在本地坐标系中沿自身的前、后、左、右及上、下方向进行移动，视角围绕自身进行旋转，它们适用于不同的场景。

3.3.1 观察者视角控制

在观察者视角控制中，可以通过键盘上的 W、A、S、D、Q 和 E 键进行移动，对应方向分别为前、左、后、右、下和上。在方法中声明一个 Vector3 类型的变量，表示移动的方向，当对应的按键被按下时，为该变量赋值，代码如下：

```
//第 3 章/GodlikeCameraController.cs

//移动方向
Vector3 motion=Vector3.zero;
if (Input.GetKey(KeyCode.W)) //向前移动
    motion+=Vector3.forward;
if (Input.GetKey(KeyCode.S)) //向后移动
    motion+=Vector3.back;
if (Input.GetKey(KeyCode.A)) //向左移动
    motion+=Vector3.left;
if (Input.GetKey(KeyCode.D)) //向右移动
    motion+=Vector3.right;
if (Input.GetKey(KeyCode.Q)) //向下移动
    motion+=Vector3.down;
if (Input.GetKey(KeyCode.E)) //向上移动
motion+=Vector3.up;
```

还可以通过按住鼠标左键进行拖曳，根据鼠标拖曳产生的偏移量进行移动。声明一个全局的 Vector2 类型变量，用于记录上一帧的鼠标位置，当前帧的鼠标位置减去上一帧的鼠标位置便可以获得鼠标移动的偏移量，代码如下：

```
//第 3 章/GodlikeCameraController.cs

//用于记录上一帧的鼠标位置
private Vector2 lastMousePosition;

private void Update()
```

```
{
    //...
    //鼠标左键是否被按住,鼠标位置是否处于窗口内
    if (Input.GetMouseButton(0)
        && Screen.safeArea.Contains(Input.mousePosition))
    {
        //当前帧的鼠标位置减去上一帧的鼠标位置得到偏移量
        Vector3 mousePosDelta=new Vector2(
            Input.mousePosition.x-lastMousePosition.x,
            Input.mousePosition.y-lastMousePosition.y);
        motion+=Vector3.left * mousePosDelta.x
            +Vector3.back * mousePosDelta.y;
    }
    //记录鼠标位置
    lastMousePosition=Input.mousePosition;
}
```

除此之外,还可以通过鼠标滚轮向视角的前方或后方进行移动,获取移动方向的代码如下:

```
float wheelValue=Input.GetAxis("Mouse ScrollWheel");
motion+=wheelValue==0 ? Vector3.zero
    : ((wheelValue>0 ? Vector3.forward : Vector3.back)
        * (invertScrollDirection ? -1f : 1f));
```

获取移动的方向后,计算目标坐标值,由于通过鼠标滚轮移动是沿自身的前方或后方进行移动的,而其他移动是沿世界空间中的前、后、左、右等方向进行移动的,因此需要进行相应区分。

假设通过键盘左侧的 Shift 按键可以加速移动,那么在将移动方向归一化后,可以通过向量数乘得到最终的移动向量。通过旋转四元数乘以移动向量得到目标坐标值,从当前坐标值向目标坐标值进行插值运算,实现平滑移动,代码如下:

```
//移动速度
[SerializeField]
private float moveSpeed=50f;
//加速系数,当 Shift 键被按下时起作用
[SerializeField]
private float boostFactor=3.5f;
//目标坐标值
private Vector3 targetPos;
//目标角度值
```

```
private Vector3 targetRot;
//插值速度
[SerializeField]
private float lerpSpeed=10f;

private void OnEnable()
{
    targetPos=transform.position;
    targetRot=transform.eulerAngles;
    lastMousePosition=Input.mousePosition;
}
private void Update()
{
    //...
    //归一化
    motion=motion.normalized;
    //按住左 Shift 键时移动加速
    if (Input.GetKey(KeyCode.LeftShift))
        motion * =boostFactor;
    motion * =Time.deltaTime * moveSpeed;
    targetPos+=(wheelValue !=0f ? Quaternion.Euler(targetRot)
        : Quaternion.Euler(0f, targetRot.y, targetRot.z)) * motion;
    transform.position=Vector3.Lerp(transform.position,
        targetPos, Time.deltaTime * lerpSpeed);
    transform.rotation=Quaternion.Lerp(transform.rotation,
        Quaternion.Euler(targetRot), Time.deltaTime * lerpSpeed);
    //记录鼠标位置
    lastMousePosition=Input.mousePosition;
}
```

当按住鼠标右键拖曳鼠标时，视角绕与地平面的交点进行旋转，交点在鼠标右键开始被按下的帧中进行计算，计算过程需要用到向量的点乘运算。

2.2.4 节介绍了向量的点乘运算，如果两个向量均是单位化的向量，模长均为 1，则点乘结果就是这两个向量夹角的余弦值。基于此，可以通过 transform.forward 与 Vector3.down 向量求得相机前方与世界空间中正下方向量的余弦值，从而可以根据该余弦值和相机的坐标 y 值求得从相机位置沿相机前方到地平面的距离，最终得到交点坐标，代码如下：

```
//第 3 章/GodlikeCameraController.cs

//鼠标右键拖动时围绕该点进行旋转
private Vector3 center=Vector3.zero;
```

```
private void Update()
{
    //鼠标右键开始被按下时计算视角围绕的点
    if (Input.GetMouseButtonDown(1))
    {
        //相机前方与世界空间中正下方向量的余弦值
        float cos=Vector3.Dot(transform.forward, Vector3.down);
        //根据余弦值和相机坐标 y 值求得从相机位置沿相机前方到地平面的距离
        float distance=transform.position.y / cos;
        distance=distance<0f ?0f : distance;
        center=transform.position +transform.forward * distance;
        center=!float.IsNaN(center.magnitude) ?center : Vector3.zero;
    }
    //...
}
```

运行结果如图 3-6 所示。

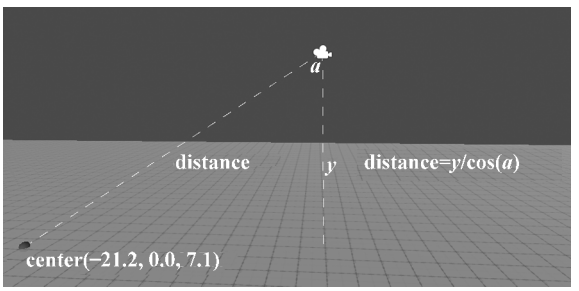


图 3-6　求解中心点

当鼠标右键持续被按下时，调用 Transform 类中的 RotateAround()方法即可实现按交点进行旋转，代码如下：

```
//第 3 章/GodlikeCameraController.cs

private void Update()
{
    //...
    else if (Input.GetMouseButton(1))
    {
        //当前帧与上一帧鼠标位置发生的偏移量
        Vector3 mousePosDelta=new Vector2(
            Input.mousePosition.x-lastMousePosition.x,
```

```
            Input.mousePosition.y-lastMousePosition.y);
        //鼠标右键被按下拖动时绕交点旋转
        transform.RotateAround(center, Vector3.up, mousePosDelta.x
            * Time.deltaTime * mouseMovementSensitivity);
        transform.RotateAround(center, transform.right,-mousePosDelta.y
            * Time.deltaTime * mouseMovementSensitivity);
        targetPos=transform.position;
        targetRot=transform.eulerAngles;
    }
    //...
}
```

在RTS类型的游戏中通常会有这样的功能，当鼠标光标移动到屏幕边缘时，视角也会向该边缘方向进行移动，实现该功能首先要定义屏幕的边缘，假设边缘宽度为2，屏幕分辨率为1920×1080，那么鼠标坐标x值在[0，2]或[1918，1920]取值范围内时，或者坐标y值在[0，2]或[1078，1080]取值范围内时，可以判定鼠标光标在屏幕边缘，代码如下：

```
//第3章/GodlikeCameraController.cs

//是否启用鼠标光标处于屏幕边缘时向外移动
[SerializeField]
private bool enableScreenEdgeMove;
//该值用于定义屏幕边缘区域
[SerializeField]
private float screenEdgeDefine=2f;

private void Update()
{
    //...
    if (enableScreenEdgeMove)
    {
        bool isMouseOnHorizontalScreenEdge =
            Input.mousePosition.x<=screenEdgeDefine ||
            Input.mousePosition.x>=Screen.width-screenEdgeDefine;
        bool isMouseOnVerticalScreenEdge =
            Input.mousePosition.y<=screenEdgeDefine ||
            Input.mousePosition.y>=Screen.height-screenEdgeDefine;
        if (isMouseOnHorizontalScreenEdge)
            motion+=Input.mousePosition.x<=screenEdgeDefine
                ?Vector3.left : Vector3.right;
        if (isMouseOnVerticalScreenEdge)
```

```
            motion+=Input.mousePosition.y<=screenEdgeDefine
                ?Vector3.back : Vector3.forward;
        }
        //...
    }
```

3.3.2　漫游视角控制

漫游视角控制与观察者视角控制的实现过程类似，只不过在漫游视角控制中处理移动时是沿相机的自身方向进行移动，这与观察者视角控制中通过鼠标滚轮移动的方式一致。处理旋转时以自身为轴心进行旋转，不需要求鼠标位置发生的偏移量及中心点等，根据鼠标在水平方向和垂直方向上的输入值进行旋转即可，代码如下：

```
//第 3 章/RoamCameraController.cs

using UnityEngine;

//<summary>
//漫游类型相机控制
//</summary>
public class RoamCameraController : MonoBehaviour
{
    //移动速度
    [SerializeField]
    private float moveSpeed=50f;
    //加速系数，当 Shift 键被按下时起作用
    [SerializeField]
    private float boostFactor=3.5f;
    //是否反转方向，用于鼠标滚轮移动
    [SerializeField]
    private bool invertScrollDirection=false;
    //鼠标转动的灵敏度
    [Range(0.1f, 20f), SerializeField]
    private float mouseMovementSensitivity=10f;
    //反转水平转动方向
    [SerializeField] private bool invertY=false;
    //插值速度
    [SerializeField]
    private float lerpSpeed=10f;
    //目标坐标值
    private Vector3 targetPos;
```

```
    //目标旋转值
    private Vector3 targetRot;

    private void OnEnable()
    {
        targetPos=transform.position;
        targetRot=transform.eulerAngles;
    }
    private void Update()
    {
        //当按住鼠标右键拖动时旋转视角
        if (Input.GetMouseButton(1))
        {
            float x=Input.GetAxis("Mouse X");
            float y=Input.GetAxis("Mouse Y");
            targetRot+=new Vector3(y * (invertY ? 1f :-1f), x, 0f)
                * mouseMovementSensitivity * Time.deltaTime * 100f;
        }
        //移动方向
        Vector3 motion=Vector3.zero;
        if (Input.GetKey(KeyCode.W)) //向前移动
            motion+=Vector3.forward;
        if (Input.GetKey(KeyCode.S)) //向后移动
            motion+=Vector3.back;
        if (Input.GetKey(KeyCode.A)) //向左移动
            motion+=Vector3.left;
        if (Input.GetKey(KeyCode.D)) //向右移动
            motion+=Vector3.right;
        if (Input.GetKey(KeyCode.Q)) //向下移动
            motion+=Vector3.down;
        if (Input.GetKey(KeyCode.E)) //向上移动
            motion+=Vector3.up;

        float wheelValue=Input.GetAxis("Mouse ScrollWheel");
        motion+=wheelValue==0 ? Vector3.zero
            : (wheelValue>0 ? Vector3.forward : Vector3.back)
                * (invertScrollDirection ? -1 : 1);

        motion=motion.normalized;
        if (Input.GetKey(KeyCode.LeftShift))
            motion *=boostFactor;
```

```
            motion * =Time.deltaTime * moveSpeed;
            targetPos+=Quaternion.Euler(targetRot) * motion;
            transform.position=Vector3.Lerp(transform.position,
                targetPos, Time.deltaTime * lerpSpeed);
            transform.rotation=Quaternion.Lerp(transform.rotation,
                Quaternion.Euler(targetRot), Time.deltaTime * lerpSpeed);
        }
    }
```

3.4　Cinemachine

Cinemachine 是官方在 2017 年推出的一套专门控制相机的模块工具，它解决了摄像机间的复杂控制、混合、切换等复杂数学和逻辑问题，减少了在开发过程中开发者对相机的脚本控制编写所需的时间成本，并能快速实现一些摄像机功能。该工具需要在 Package Manager 中下载导入，如图 3-7 所示。

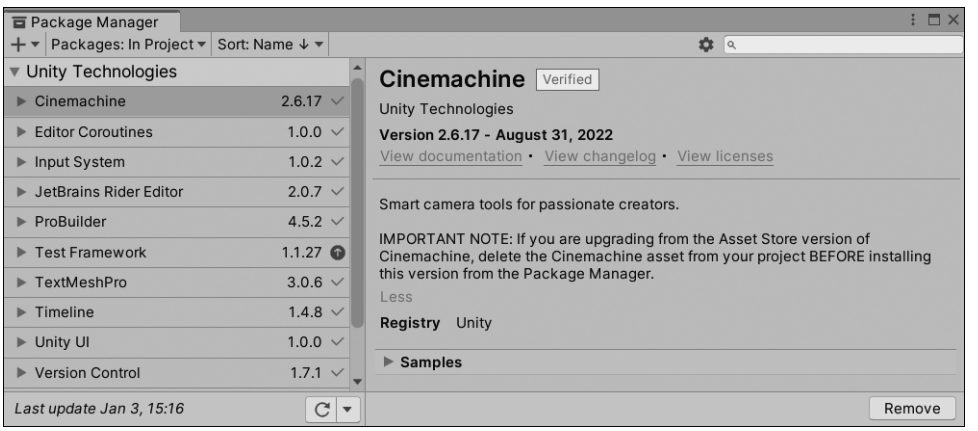


图 3-7　Cinemachine Package

在 Cinemachine 中，虚拟相机（Virtual Camera）是一个重要的概念。创建虚拟相机后，场景中的原相机会自动添加一个 CinemachineBrain 组件，如图 3-8 所示，该组件是整个模块的核心组件，可以称为“大脑”，它挂载于 Camera 组件所在的对象上，监控着场景中所有为活跃状态的虚拟相机，以实现镜头的切换及控制。此外，开发者还可以使用 Timeline 来控制虚拟相机，实现效果丰富的镜头动画。

3.4.1　基于虚拟相机实现第三人称视角

Cinemachine 中提供了第三人称相机解决方案，导入 Cinemachine 工具后，可以发现 Unity 窗口顶部多了一个 Cinemachine 菜单栏，通过菜单栏中的 Create Virtual Camera 选

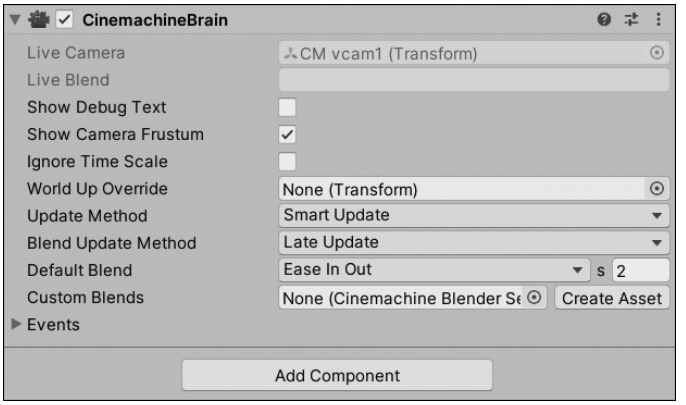


图 3-8　CinemachineBrain 组件

项即可在场景中创建一个虚拟相机。

创建完成后，可以发现场景发生了两个变化。第一，在主相机的层级中，出现了一个图标，如图 3-9 所示，该图标表示主相机挂载了 CinemachineBrain 组件，一个场景中只存在一个 CinemachineBrain 组件。第二，场景中多了一个新的游戏物体 CM vcam1，该游戏物体上挂载了 CinemachineVirtualCamera 组件，即虚拟相机，如图 3-10 所示，如果继续创建，则场景中便会多出新的挂载虚拟相机组件的游戏物体 CM vcam2，以此类推。

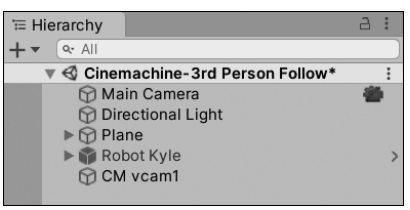


图 3-9　场景变化

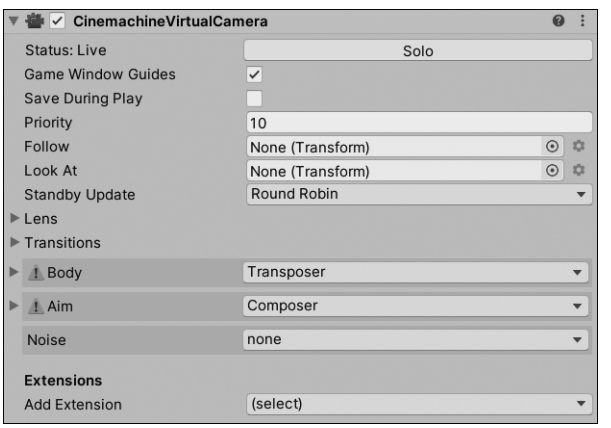


图 3-10　CinemachineVirtualCamera 组件

如果要将虚拟相机作为第三人称视角相机使用，则需要为 Follow 属性赋值，如图 3-11

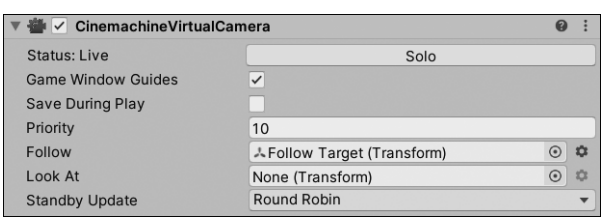


图 3-11　虚拟相机跟随的目标

所示，它表示相机跟随的目标。

设置跟随的目标后，将 Body 的类型设置为 3rd Person Follow，如图 3-12 所示，它表示将虚拟相机作为第三人称视角跟随相机使用，Body 中的属性详解见表 3-1。

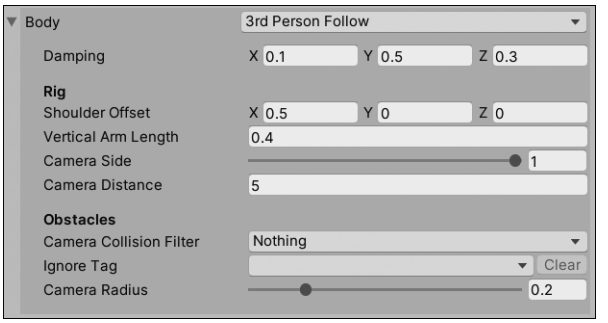


图 3-12　3rd Person Follow

表 3-1　3rd Person Follow Body 属性详解

属　　性	详　　解
Damping	相机跟随的阻尼值，值越大，相机到达目标位置的速度越慢
Shoulder Offset	肩部偏移值（相对于跟随目标位置的偏移量）
Vertical Arm Length	手相对于肩部的垂直偏移值，当相机在垂直方向上旋转时，该值会影响跟随目标在屏幕中的位置
Camera Side	指定相机偏向哪个肩部，取值范围为 0～1，0 表示左侧，1 表示右侧
Camera Distance	相机与跟随目标的距离
Camera Collision Filter	障碍物检测层级（相机将避开通过这些层级设定的障碍物）
Ignore Tag	设置为此标签的障碍物将被忽略
Camera Radius	相机半径（用于指定相机离障碍物的距离）

当跟随目标是角色本身时，相机朝向与角色朝向保持一致，假设想要实现通过用户输入使相机围绕角色旋转，可以将跟随目标设置为独立于角色旋转的空物体，通过旋转该空物体实现相机围绕角色进行旋转。

3.4.2　轨道路径与推轨相机

通过 Cinemachine 工具可以实现复杂的镜头动画，如图 3-13 所示，将 CinemachineVirtualCamera 组件中 Body 的类型设置为 Tracked Dolly 类型，可以实现让虚拟相机沿自定义的轨道路径移动，该类型 Body 中的属性详解见表 3-2。

启用 Auto Dolly 后，虚拟相机的位置会自动动画化到路径上最接近跟随目标的位置，这也意味着，使用 Tracked Dolly 时，不仅需要提供一个轨道路径，还需要指定一个跟随的目标。

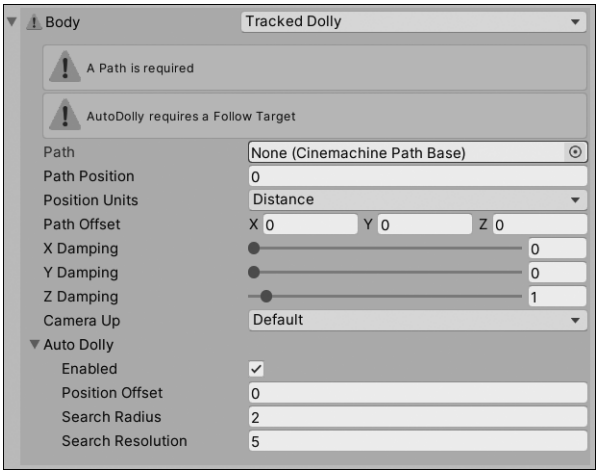


图 3-13　Tracked Dolly

表 3-2　Tracked Dolly Body 属性详解

属　　性	详　　解
Path	轨道路径
Path Position	虚拟相机在路径中的位置
Position Units	路径位置的度量单位
Path Offset	虚拟相机相对于路径的位置
X Damping	虚拟相机到达目标位置在 x 轴上的阻尼值
Y Damping	虚拟相机到达目标位置在 y 轴上的阻尼值
Z Damping	虚拟相机到达目标位置在 z 轴上的阻尼值
Camera Up	设置虚拟相机向上方向的类型
Auto Dolly	自动推轨的相关设置

工具中提供了两种轨道路径组件，分别为 CinemachinePath 和 CinemachineSmoothPath。路径通过贝塞尔曲线实现，如果切线设置不当，则会影响路径动画的平滑性和连续性，两种类型路径组件的主要区别在于 CinemachineSmoothPath 会自动设置切线，以确保完全平滑。

创建一个新的游戏物体并为其添加 CinemachineSmoothPath 组件，在该组件的检视面板中编辑路径点，如图 3-14 所示，也可以在场景中进行编辑，该组件对应的编辑器类中通过创建坐标操控柄为在场景中编辑路径点提供了支持。

路径点编辑完成后，当选中组件所在的游戏物体时，路径以轨道的形状展示在场景中，如图 3-15 所示。

创建另一个游戏物体并为其添加 CinemachineDollyCart 组件，将其作为虚拟相机的跟