第 3 章 自定义检视面板

本章将介绍如何使用 CustomEditor 特性和 Editor 类自定义组件的检视面板，以及如何使用 CustomPropertyDrawer 特性和 PropertyDrawer 类自定义属性在检视面板中的绘制。为了帮助读者更好地理解这些知识，在本章中提供了详细的示例。

3.1 创建自定义编辑器类

如果想要自定义一个组件的检视面板（Inspector）如何绘制，则需要为组件创建自定义编辑器类，该类继承了 Editor 类，并为其添加了 CustomEditor 特性，特性的构造函数的代码如下：

```
public CustomEditor (Type inspectedType);
public CustomEditor (Type inspectedType, bool editorForChildClasses);
```

参数 inspectedType 表示检视的类型，即自定义哪种类型组件的检视面板，editorForChildClasses 表示是否为其派生类也使用同样的检视面板，默认值为 false。

以 CustomComponent 组件为例，为其创建自定义编辑器类 CustomComponentEditor，代码如下：

```
using UnityEngine;
using UnityEditor;

[CustomEditor(typeof(CustomComponent))]
public class CustomComponentEditor : Editor { }
```

类似于 MonoBehaviour 的 Awake()、OnEnable()、OnDisable()、OnDestroy()等生命周期函数，Editor 类中也有相应的回调方法，见表 3-1。

表 3-1 Editor 类中的回调方法

方　法	详　解
Awake()	当组件所挂载的物体被选中时，该函数会被调用
OnEnable()	当组件所挂载的物体被选中时，该函数会被调用，晚于 Awake()执行

续表

方　法	详　解
OnValidate()	当组件的检视面板的值被修改时，该函数会被调用
OnDisable()	当组件所挂载的物体被取消选中时，该函数会被调用
OnDestroy()	当组件所挂载的物体被取消选中时，该函数会被调用，晚于 OnDisable()执行

其中，Awake()、OnEnable()方法用于进行初始化操作，例如，如果想要获取所检视的组件对象，则可以在 OnEnable()中将 Editor 中的 target 对象转换为目标类型，target 表示所检视的对象，类型为 Object，示例代码如下：

```
//第 3 章/CustomComponentEditor.cs

using UnityEngine;
using UnityEditor;

[CustomEditor(typeof(CustomComponent))]
public class CustomComponentEditor : Editor
{
    private CustomComponent component;

    private void OnEnable()
    {
        component = target as CustomComponent;
    }
}
```

3.1.1 如何自定义检视面板中的 GUI 内容

Editor 类中的虚方法 OnInspectorGUI()定义了组件的检视面板如何绘制，因此重写该方法即可实现自定义。如果想要扩展检视面板，则可以保留 base.OnInspectorGUI()的调用，再添加扩展的内容，也可以不保留，完全自定义。

首先给 CustomComponent 组件添加一些不同类型的变量，再来展示如何在 OnInspectorGUI()方法中添加控件来编辑这些变量的值，代码如下：

```
//第 3 章/CustomComponent.cs

using UnityEngine;

public class CustomComponent : MonoBehaviour
{
    public int intValue;
    [SerializeField] private string stringValue;
    [SerializeField] private bool boolValue;
    [SerializeField] private GameObject go;
```

```
    public enum ExampleEnum
    {
        Enum1,
        Enum2,
        Enum3,
    }
    [SerializeField] private ExampleEnum enumValue;
}
```

因为整数类型的字段 intValue 使用了 public 进行修饰，所以在编辑器类中可以直接访问和修改其值。那么其他使用 private 修饰的私有类型的字段，应该如何在编辑器类中访问和修改它们的值呢？

有两种方式，一种是使用 SerializedProperty 序列化属性；另一种是使用反射，一般推荐使用前者。

序列化属性通过 Editor 中的 serializedObject 属性调用 FindProperty()方法获取，该属性指的是当前检视的序列化对象，当调用 FindProperty()方法时将要获取的序列化属性的字段名称作为参数传入即可，代码如下：

```
//第 3 章/CustomComponentEditor.cs

using UnityEngine;
using UnityEditor;
using System.Reflection;

[CustomEditor(typeof(CustomComponent))]
public class CustomComponentEditor : Editor
{
    private CustomComponent component;
    private SerializedProperty stringValueProperty;
    private FieldInfo boolValueFieldInfo;
    private SerializedProperty gameObjectProperty;
    private SerializedProperty enumValue;

    private void OnEnable()
    {
        component = target as CustomComponent;
        stringValueProperty = serializedObject.FindProperty("stringValue");
        boolValueFieldInfo = typeof(CustomComponent).GetField(
            "boolValue", BindingFlags.Instance | BindingFlags.NonPublic);
        gameObjectProperty = serializedObject.FindProperty("go");
        enumValue = serializedObject.FindProperty("enumValue");
    }

    public override void OnInspectorGUI()
```

```
        {
            CustomExample();
        }
        private void CustomExample()
        {
            //public 修饰的字段，可以直接访问和修改其值
            component.intValue = EditorGUILayout.IntField(
                "Int Value", component.intValue);
            //private 修饰的字段，通过序列化属性的方式访问和修改其值
            stringValueProperty.stringValue = EditorGUILayout.TextField(
                "String Value", stringValueProperty.stringValue);
            //private 修饰的字段，通过反射的方式访问和修改其值
            boolValueFieldInfo.SetValue(component, EditorGUILayout.Toggle(
                "Bool Value", (bool)boolValueFieldInfo.GetValue(component)));
            EditorGUILayout.PropertyField(gameObjectProperty);
            enumValue.enumValueIndex = EditorGUILayout.Popup(
                "Enum Value", enumValue.enumValueIndex, enumValue.enumNames);
        }
    }
```

如图 3-1 所示，自定义编辑器已经将组件的序列化属性绘制在检视面板中，但是此时修改它们的值并不会被保存，因为当使用自定义编辑器修改组件的序列化属性时，需要调用相应的方法来应用这些修改并保存，而调用这些方法之前需要先检测检视面板或序列化属性是否发生了变更。

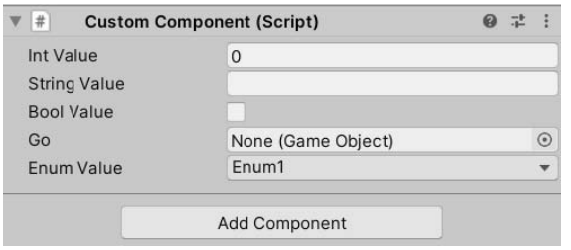

图 3-1　自定义检视面板

3.1.2　如何检测和应用修改

判断序列化属性是否发生变化的方法有多种，可以使用 GUI 类中的 changed 属性，如果任何控件的值发生了变更，则 changed 属性的返回值为 true。也可以使用 EditorGUI 类中的 BeginChangeCheck()方法来开始一块代码块以检测 GUI 的变更，调用 EndChangeCheck()方法结束代码块，如果在该代码块中 GUI 发生了变更，则 EndChangeCheck()方法的返回值将为 true。除此之外，也可以直接对比控件交互修改后的值与属性值是否一致，代码如下：

```
    //第 3 章/CustomComponentEditor.cs
```

```
public override void OnInspectorGUI()
{
    ChangeCheckExample();
}
private void ChangeCheckExample()
{
    //开启一块代码块以检测 GUI 是否变更
    EditorGUI.BeginChangeCheck();
    //public 修饰的字段，可以直接访问和修改其值
    component.intValue = EditorGUILayout.IntField(
        "Int Value", component.intValue);
    //如果在代码块中 GUI 发生了变更，则返回值为 true
    if (EditorGUI.EndChangeCheck())
    {
        Debug.Log("IntValue 发生变更");
    }
    //private 修饰的字段，通过序列化属性的方式访问和修改其值
    //接收控件的返回值
    string newStringValue = EditorGUILayout.TextField(
        "String Value", stringValueProperty.stringValue);
    //对比是否一致，如果不一致，则更新
    if (newStringValue != stringValueProperty.stringValue)
    {
        stringValueProperty.stringValue = newStringValue;
        Debug.Log("StringValue 发生变更");
    }
    //private 修饰的字段，通过反射的方式访问和修改其值
    boolValueFieldInfo.SetValue(component, EditorGUILayout.Toggle(
        "Bool Value", (bool)boolValueFieldInfo.GetValue(component)));
    EditorGUILayout.PropertyField(gameObjectProperty);
    enumValue.enumValueIndex = EditorGUILayout.Popup(
        "Enum Value", enumValue.enumValueIndex, enumValue.enumNames);
    //有任何控件发生了变更
    if (GUI.changed)
    {
        Debug.Log("GUI 发生了变更");
    }
}
```

当检测到发生变更后，应该调用序列化对象中的 ApplyModifiedProperties()方法来应用修改，应用后，还需要调用 EditorUtility 中的 SetDirty()方法将对象标记为“脏”，也就是表示它发生了变更，此时再使用快捷键 Ctrl+S 才最终完成保存操作，代码如下：

```
//第 3 章/CustomComponentEditor.cs

public override void OnInspectorGUI()
{
```

```
        ApplyModificationExample();
    }
    private void ApplyModificationExample()
    {
        //public 修饰的字段，可以直接访问和修改其值
        component.intValue = EditorGUILayout.IntField(
            "Int Value", component.intValue);
        //private 修饰的字段，通过序列化属性的方式访问和修改其值
        stringValueProperty.stringValue = EditorGUILayout.TextField(
            "String Value", stringValueProperty.stringValue);
        //private 修饰的字段，通过反射的方式访问和修改其值
        boolValueFieldInfo.SetValue(component, EditorGUILayout.Toggle(
            "Bool Value", (bool)boolValueFieldInfo.GetValue(component)));
        EditorGUILayout.PropertyField(gameObjectProperty);
        enumValue.enumValueIndex = EditorGUILayout.Popup(
            "Enum Value", enumValue.enumValueIndex, enumValue.enumNames);

        //有任何控件发生了变更
        if (GUI.changed)
        {
            //应用修改
            serializedObject.ApplyModifiedProperties();
            //标记为“脏”
            EditorUtility.SetDirty(component);
        }
    }
```

3.1.3 编辑器操作的撤销与恢复

如果想要使编辑器中的操作支持撤销和恢复，则需要用到 Undo 类中的方法，常用的几种方法和作用见表 3-2。

表 3-2 Undo 类中的方法和作用

方　法	作　用
RecordObject()	记录对象的状态
AddComponent()	为游戏物体添加组件并针对这一操作注册撤销操作
RegisterCreatedObjectUndo()	针对新创建的对象注册撤销操作
DestoryObjectImmediate()	销毁对象并注册撤销操作，以便能够重新创建该对象
SetTransformParent()	更改 Transform 的父级，并注册撤销操作

1. RecordObject()

RecordObject()是最常用的方法，当在编辑器中与控件交互修改某个序列化属性的值时，为了使这次修改操作可撤销，在赋新值之前需要先调用该方法来记录属性的状态，这样在修

改之后使用快捷键 Ctrl+Z 便可撤销本次修改操作，恢复之前的值。

此方法的代码如下，参数 objectToUndo 表示的是要修改的对象的引用，name 表示的是在撤销历史记录中记录的本次操作的名称。

```
public static void RecordObject (Object objectToUndo, string name);
```

以 CustomComponent 组件中的 intValue 为例，使用一个新的 int 值记录控件被编辑后的值，对比新旧值是否一致，如果不一致，则在更新值之前调用 RecordObject()方法，第 1 个参数用于传入对象的引用，第 2 个参数用于传入该项操作的命名，代码如下：

```
//第 3 章/CustomComponentEditor.cs

public override void OnInspectorGUI()
{
    RecordObjectExample();
}
private void RecordObjectExample()
{
    int newIntValue = EditorGUILayout.IntField(
        "Int Value", component.intValue);
    if (newIntValue != component.intValue)
    {
        Undo.RecordObject(component, "Change Int Value");
        component.intValue = newIntValue;
        serializedObject.ApplyModifiedProperties();
        EditorUtility.SetDirty(component);
    }
}
```

本次操作的名称记录在历史记录中，打开 Edit 菜单可以查看，如图 3-2 所示。

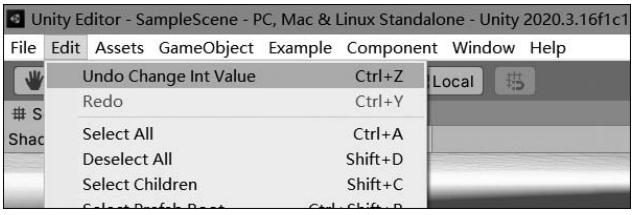

图 3-2　Undo Change Int Value

2. AddComponent()

当在编辑器脚本中为某个游戏物体添加组件时，为了使添加组件这项操作支持撤销与恢复，需要使用 Undo 类中的 AddComponent()方法，而非 GameObject 中的 AddComponent()方法。方法的返回值是添加的新组件，执行撤销操作后，新添加的组件会被销毁。

此方法的代码如下，参数 gameObject 表示要添加组件的游戏物体，type 则是要添加的组件的类型。

```
public static Component AddComponent(GameObject gameObject, Type type);
```

在示例脚本中创建一个 GUI 按钮，当单击该按钮时，使用该方法为组件所在的游戏物体添加一个 BoxCollider 组件。这时如果在编辑器中使用快捷键 Ctrl+Z 执行撤销操作，则这个 BoxCollider 组件将会被销毁，代码如下：

```
//第 3 章/CustomComponentEditor.cs

public override void OnInspectorGUI()
{
    AddComponentExample();
}
private void AddComponentExample()
{
    if (GUILayout.Button("Add BoxCollider"))
    {
        Undo.AddComponent(component.gameObject,
            typeof(BoxCollider));
    }
}
```

操作历史记录如图 3-3 所示。

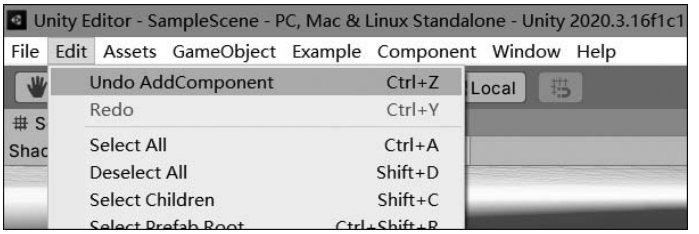

图 3-3　Undo AddComponent

3. RegisterCreatedObjectUndo()

当在编辑器脚本中创建一个新的游戏物体时，为了使这项创建操作支持撤销与恢复，需要在创建游戏物体之后，调用 Undo 类中的 RegisterCreatedObjectUndo()方法，以便为新创建的游戏物体注册撤销操作。

此方法的代码如下，参数 objectToUndo 表示新创建的游戏物体，当执行撤销操作时，该物体节点将被销毁，name 表示在历史记录中记录的本次操作的名称。

```
public static void RegisterCreatedObjectUndo (Object objectToUndo, string
name);
```

在示例脚本中创建一个 GUI 按钮，当单击该按钮时，在场景中创建一个新的游戏物体，并调用该方法注册撤销操作。这时如果在编辑器中使用快捷键 Ctrl+Z 执行撤销操作，则这个新建的物体节点将被销毁，代码如下：

```
//第 3 章/CustomComponentEditor.cs

public override void OnInspectorGUI()
{
    RegisterCreatedObjectUndoExample();
}
private void RegisterCreatedObjectUndoExample()
{
    if (GUILayout.Button("Create New GameObject"))
    {
        GameObject go = new GameObject();
        Undo.RegisterCreatedObjectUndo(go,
            "Create New GameObject");
    }
}
```

操作历史记录如图 3-4 所示。

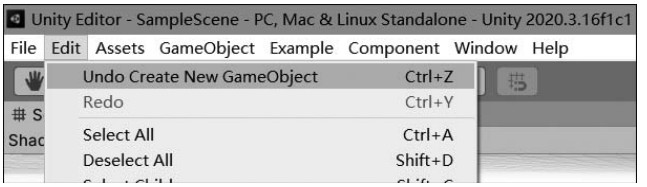

图 3-4　Undo Create New GameObject

4. DestroyObjectImmediate()

当在编辑器脚本中销毁一个对象时，为了使这项销毁操作支持撤销与恢复，需要使用 Undo 类中的 DestoryObjectImmediate()方法，而非 Object 中的 DestroyImmediate()方法。销毁的对象被保存在撤销缓存区中，以便在执行撤销操作时能够重新创建该对象。

此方法的代码如下，参数 objectToUndo 表示要销毁的对象。

```
public static void DestroyObjectImmediate (Object objectToUndo);
```

同样地，在示例脚本中创建一个 GUI 按钮，当单击该按钮时，使用该方法将当前的对象销毁。这时如果在编辑器中使用快捷键 Ctrl+Z 执行撤销操作，则被销毁的对象将被重建，代码如下：

```
//第 3 章/CustomComponentEditor.cs

public override void OnInspectorGUI()
{
    DestroyObjectImmediateExample();
}
private void DestroyObjectImmediateExample()
{
    if (GUILayout.Button("Destroy"))
```

```
        {
            Undo.DestroyObjectImmediate(component);
        }
    }
```

操作历史记录如图 3-5 所示。

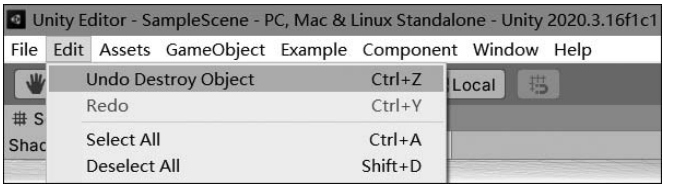

图 3-5　Undo Destroy Object

5. SetTransformParent()

当在编辑器脚本中修改某个 Transform 组件的父级时，为了使这项修改操作支持撤销与恢复，需要使用 Undo 类中的 SetTransformParent()方法，而非 Transform 中的 SetParent()方法。

此方法的代码如下，参数 transform 表示要修改父级的 Transform 组件，newParent 表示指定的父级，name 则是在操作历史记录中记录的本次操作的名称。

```
    public static void SetTransformParent (Transform transform, Transform
newParent, string name);
```

同样地，在示例脚本中创建一个 GUI 按钮，当单击该按钮时，调用该方法将当前游戏物体的父级修改为根级。这时如果在编辑器中使用快捷键 Ctrl+Z 执行撤销操作，则当前游戏物体的父级将由根级恢复为操作之前的父级，代码如下：

```
    //第 3 章/CustomComponentEditor.cs

    public override void OnInspectorGUI()
    {
        SetTransformParentExample();
    }
    private void SetTransformParentExample()
    {
        if (GUILayout.Button("Set As Root"))
        {
            Undo.SetTransformParent(component.transform,
                null, "Change Transform Parent");
        }
    }
```

操作历史记录如图 3-6 所示。

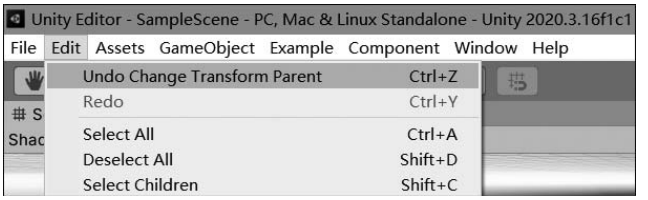

图 3-6　Undo Change Transform Parent

3.1.4　实现 DoTween 动画参数的编辑

DoTween 是一个免费开源的高效动画工具，相信大部分开发者或多或少会用到它。使用 DoTween 工具可以实现 Transform 组件的移动、旋转和缩放动画，在调用相关方法时，可以设置动画时长、延时、起始值、目标值等参数。

本节针对 Transform 封装一个动画组件，并在它的编辑器类中自定义这些动画参数的绘制，为它们提供控件，以便进行交互修改。

首先定义移动、旋转和缩放动画相关的参数，以旋转动画为例，toggle 表示动画的开关，startValue 与 endValue 分别表示起始值和目标值，duration 表示动画的时长，delay 表示动画的延时时长，ease 则表示 DoTween 动画的类型，不同的类型对应不同的动画效果。以上这些参数在移动动画和缩放动画中都包含，而 mode 是旋转动画中特有的字段，表示旋转的类型，代码如下：

```
//第 3 章/RotateAnimation.cs

using System;
using UnityEngine;
using DG.Tweening;

[Serializable]
public class RotateAnimation
{
    public bool toggle;
    public Vector3 startValue;
    public Vector3 endValue;
    public float duration = 1f;
    public float delay = 0f;
    public Ease ease = Ease.Linear;
    public RotateMode mode = RotateMode.Fast;
}
```

然后创建 Transform 的 Tween 动画组件，move、rotate、scale 分别表示移动动画、旋转动画、缩放动画，代码如下：

```
//第 3 章/TransformTweenAnimation.cs
```

```
using UnityEngine;

public class TransformTweenAnimation : MonoBehaviour
{
    public MoveAnimation move;
    public RotateAnimation rotate;
    public ScaleAnimation scale;
}
```

将组件挂载于游戏物体，默认检视面板如图 3-7 所示。

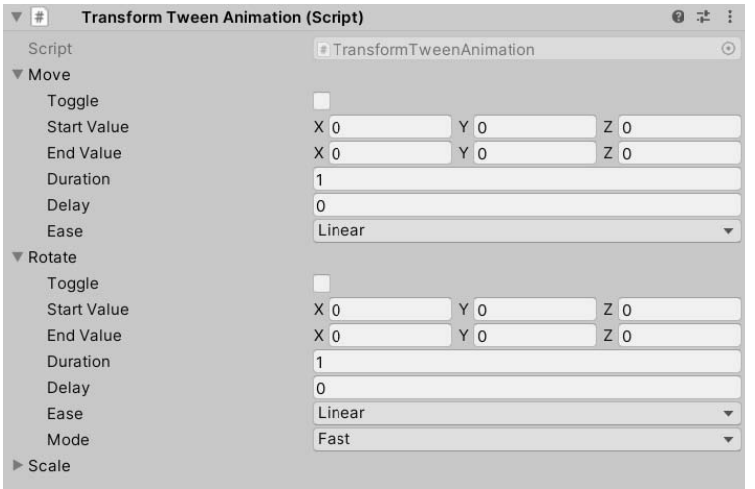

图 3-7　TransformTweenAnimation 组件的默认检视面板

为 TransformTweenAnimation 组件创建自定义编辑器类，注意需要为编辑器类添加 CustomEditor 特性，否则它无法生效。在 OnGUI()方法中添加菜单栏，使用按钮控制各动画的开关，开启水平布局，将这些按钮水平排列，还需要使用 Undo 类中的 RecordObject()方法记录属性的状态，以便用户使用快捷键 Ctrl+Z 与 Ctrl+Y 实现编辑操作的撤销与恢复，代码如下：

```
//第 3 章/TransformTweenAnimationEditor.cs

using UnityEngine;
using UnityEditor;

[CustomEditor(typeof(TransformTweenAnimation))]
public class TransformTweenAnimationEditor : Editor
{
    private TransformTweenAnimation animation;
    private readonly static float alpha = .4f;

    private void OnEnable()
```

```
    {
        animation = target as TransformTweenAnimation;
    }
    public override void OnInspectorGUI()
    {
        OnMenuGUI();
    }
    private void OnMenuGUI()
    {
        GUILayout.BeginHorizontal();
        Color cacheColor = GUI.color;
        Color alphaColor = new Color(cacheColor.r,
            cacheColor.g, cacheColor.b, alpha);
        //Move
        GUI.color = animation.move.toggle ? cacheColor : alphaColor;
        if (GUILayout.Button(EditorGUIUtility.IconContent("MoveTool"),
            "ButtonLeft", GUILayout.Width(25f)))
        {
            Undo.RecordObject(target, "Move Toggle");
            animation.move.toggle = !animation.move.toggle;
            EditorUtility.SetDirty(target);
        }
        //Rotate
        GUI.color = animation.rotate.toggle ? cacheColor : alphaColor;
        if (GUILayout.Button(EditorGUIUtility.IconContent("RotateTool"),
            "ButtonMid", GUILayout.Width(25f)))
        {
            Undo.RecordObject(target, "Rotate Toggle");
            animation.rotate.toggle = !animation.rotate.toggle;
            EditorUtility.SetDirty(target);
        }
        //Scale
        GUI.color = animation.scale.toggle ? cacheColor : alphaColor;
        if (GUILayout.Button(EditorGUIUtility.IconContent("ScaleTool"),
            "ButtonRight", GUILayout.Width(25f)))
        {
            Undo.RecordObject(target, "Scale Toggle");
            animation.scale.toggle = !animation.scale.toggle;
            EditorUtility.SetDirty(target);
        }
        GUI.color = cacheColor;
        GUILayout.EndHorizontal();
    }
}
```

接下来为各动画的参数添加交互修改的控件，实现动画参数的编辑，仍然以旋转动画为例，代码如下：

```
//第 3 章/TransformTweenAnimationEditor.cs

readonly static float labelWidth = 60f;
readonly static GUIContent duration = new GUIContent("Duration", "动画时长");
readonly static GUIContent delay = new GUIContent("Delay", "延时时长");
readonly static GUIContent from = new GUIContent("From", "初始值");
readonly static GUIContent to = new GUIContent("To", "目标值");
readonly static GUIContent ease = new GUIContent("Ease");
readonly static GUIContent rotateMode = new GUIContent("Mode", "旋转模式");

public override void OnInspectorGUI()
{
    OnMenuGUI();
    OnMoveAnimationGUI();
    GUILayout.Space(3f);
    OnRotateAnimationGUI();
    GUILayout.Space(3f);
    OnScaleAnimationGUI();
}
private void OnRotateAnimationGUI()
{
    if (animation.rotate.toggle)
    {
        GUILayout.BeginHorizontal("Badge");
        {
            GUILayout.BeginVertical();
            GUILayout.Space(40f);
            GUILayout.Label(EditorGUIUtility.IconContent("RotateTool"));
            GUILayout.EndVertical();

            GUILayout.BeginVertical();
            {
                //Duration、Delay
                GUILayout.BeginHorizontal();
                GUILayout.Label(duration, GUILayout.Width(labelWidth));
                var newDuration = EditorGUILayout.FloatField(
                    animation.rotate.duration);
                if (newDuration != animation.rotate.duration)
                {
                    Undo.RecordObject(target, "Rotate Duration");
                    animation.rotate.duration = newDuration;
                    EditorUtility.SetDirty(target);
                }

                GUILayout.Label(delay, GUILayout.Width(labelWidth - 20f));
                var newDelay = EditorGUILayout.FloatField(
```

```
            animation.rotate.delay);
        if (newDelay != animation.rotate.delay)
        {
            Undo.RecordObject(target, "Rotate Delay");
            animation.rotate.delay = newDelay;
            EditorUtility.SetDirty(target);
        }
        GUILayout.EndHorizontal();

        //From
        GUILayout.BeginHorizontal();
        GUILayout.Label(from, GUILayout.Width(labelWidth));
        Vector3 newStartValue = EditorGUILayout.Vector3Field(
            GUIContent.none, animation.rotate.startValue);
        if (newStartValue != animation.rotate.startValue)
        {
            Undo.RecordObject(target, "Rotate From");
            animation.rotate.startValue = newStartValue;
            EditorUtility.SetDirty(target);
        }
        GUILayout.EndHorizontal();

        //To
        GUILayout.BeginHorizontal();
        GUILayout.Label(to, GUILayout.Width(labelWidth));
        Vector3 newEndValue = EditorGUILayout.Vector3Field(
            GUIContent.none, animation.rotate.endValue);
        if (newEndValue != animation.rotate.endValue)
        {
            Undo.RecordObject(target, "Rotate To");
            animation.rotate.endValue = newEndValue;
            EditorUtility.SetDirty(target);
        }
        GUILayout.EndHorizontal();

        //Rotate Mode
        GUILayout.BeginHorizontal();
        GUILayout.Label(rotateMode, GUILayout.Width(labelWidth));
        var newRotateMode = (RotateMode)EditorGUILayout.EnumPopup(
            animation.rotate.mode);
        if (newRotateMode != animation.rotate.mode)
        {
            Undo.RecordObject(target, "Rotate Mode");
            animation.rotate.mode = newRotateMode;
            EditorUtility.SetDirty(target);
        }
        GUILayout.EndHorizontal();
```

```
                //Ease
                GUILayout.BeginHorizontal();
                GUILayout.Label(ease, GUILayout.Width(labelWidth));
                var newEase = (Ease)EditorGUILayout.EnumPopup(
                    animation.rotate.ease);
                if (newEase != animation.rotate.ease)
                {
                    Undo.RecordObject(target, "Rotate Ease");
                    animation.rotate.ease = newEase;
                    EditorUtility.SetDirty(target);
                }
                GUILayout.EndHorizontal();
            }
            GUILayout.EndVertical();
        }
        GUILayout.EndHorizontal();
    }
}
```

最终效果如图 3-8 所示。

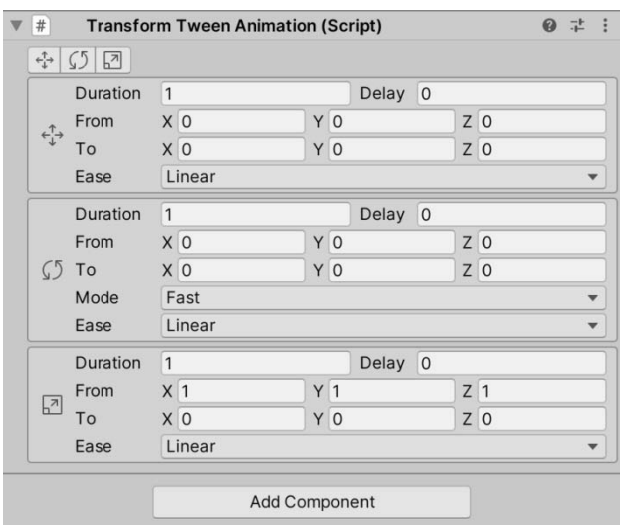

图 3-8　自定义 TransformTweenAnimation 组件检视面板

3.1.5　如何自定义预览窗口

预览窗口是指当选中模型、动画或音频等类型的资源时，在检视窗口的下方出现的小窗口，如图 3-9 所示。在默认情况下，在组件对象的检视窗口中是没有预览窗口的，如果想开启预览窗口，则需要重写 Editor 类中的虚方法 HasPreviewGUI()，当方法的返回值为 true 时就可以打开预览窗口，而预览窗口中的内容需要重写 OnPreviewGUI()方法实现。

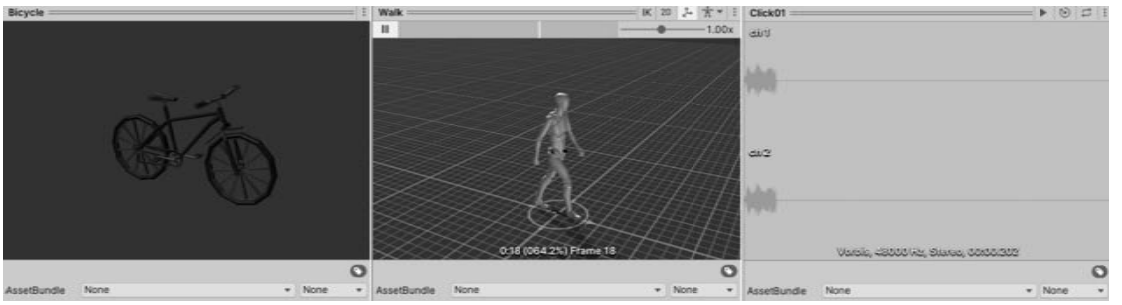

图 3-9　预览窗口

除此之外，Editor 类中还有两个重要的与预览窗口相关的虚方法，即 GetPreviewTitle()和 OnPreviewSettings()，前者可以设置预览窗口的标题，后者可以在标题栏的右侧添加设置相关的控件。

仍然以 CustomComponent 组件为例，让 HasPreviewGUI()方法在程序运行时返回值 true，并在 OnPreviewGUI()方法中将各序列化属性的值绘制出来，代码如下：

```
//第 3 章/CustomComponentEditor.cs

public override bool HasPreviewGUI()
{
    return EditorApplication.isPlaying;
}
public override GUIContent GetPreviewTitle()
{
    return new GUIContent("这里是窗口的标题");
}
public override void OnPreviewSettings()
{
    GUILayout.Button("Button1", EditorStyles.toolbarButton);
    GUILayout.Button("Button2", EditorStyles.toolbarButton);
}
public override void OnPreviewGUI(Rect r, GUIStyle background)
{
    GUILayout.Label(string.Format("Int Value: {0}",
        example.intValue));
    GUILayout.Label(string.Format("String Value: {0}",
        stringValueProperty.stringValue));
    GUILayout.Label(string.Format("Bool Value: {0}",
        (bool)boolValueFieldInfo.GetValue(example)));
    GUILayout.Label(string.Format("Go Value: {0}",
        gameObjectProperty.objectReferenceValue));
    GUILayout.Label(string.Format("Enum Value: {0}",
        enumValue.enumNames[enumValue.enumValueIndex]));
}
```

程序运行后，选中 CustomComponent 组件所在的物体，查看 Inspector 中的预览窗口，结果如图 3-10 所示。

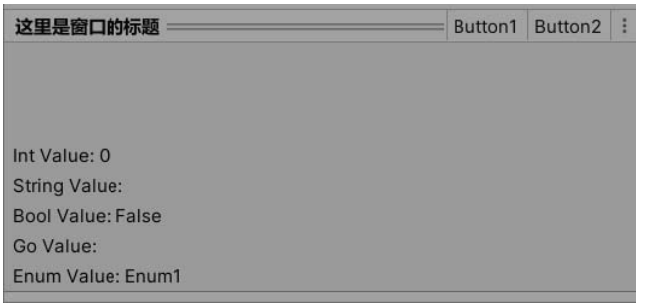

图 3-10　自定义预览窗口

OnPreviewGUI()方法的第 1 个参数为 Rect 类型，它表示预览窗口的矩形区域。通常情况下，在根据矩形区域绘制内容时，需要使用 GUI 或 EditorGUI 中的方法，GUILayout 类和 EditorGUILayout 类中的方法不再适用。GUI 类位于 Unity Engine 命名空间中，EditorGUI 位于 Unity Editor 命名空间中，与 GUILayout 和 EditorGUILayout 一样，它们有不同的适用范围。

1. EditorGUI

EditorGUI 类中包含 3 个公开的静态变量，其中 actionKey 表示当前是否按下了 Ctrl 键，这对于使用快捷键实现某些功能是十分有用的，它在 macOS 系统中对应的是 Command 键。indentLevel 用于控制字段标签的缩进级别。showMixedValue 表示编辑多个不同值的外观，例如，在层级窗口中选中多个游戏物体，如果这些游戏物体的 Transform 组件拥有不同的坐标、旋转和缩放值，则在检视面板中这些变量的值将使用“—”来表示，如图 3-11 所示。

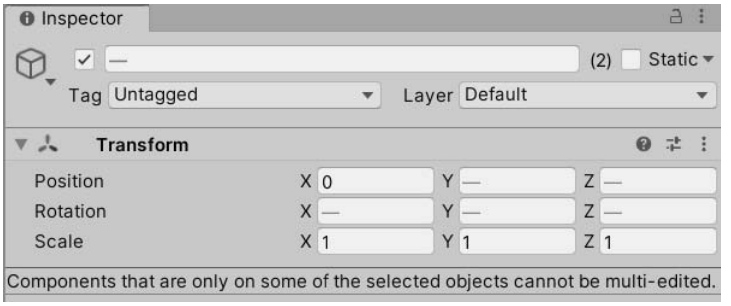

图 3-11　批量编辑

EditorGUI 类中包含的静态方法见表 3-3。

表 3-3　EditorGUI 类中的静态方法

方　　法	作　　用
BeginChangeCheck()	开启一个新代码块来检查 GUI 是否发生了变更，与 EndChangeCheck 配合使用
BeginDisabledGroup()	创建一组禁用的控件，与 EndDisabledGroup 配合使用，表示组内的控件无法进行交互
BeginFoldoutHeaderGroup()	创建一个折叠栏，与 EndFoldoutHeaderGroup 配合使用
BeginProperty()	创建一个属性封装器，可用于使常规 GUI 控件与 SerializedProperty 配合使用
BoundsField()	创建一个用于编辑 Bounds 类型字段的控件
BoundsIntField()	创建一个用于编辑 BoundsInt 类型字段的控件
CanCacheInspectorGUI()	确定能否缓存 SerializedProperty 的检视面板 GUI
ColorField()	创建一个用于编辑 Color 类型字段的控件
CurveField()	创建一个用于编辑 AnimationCurve 类型字段的控件
DelayedDoubleField()	创建一个用于编辑 double 类型字段的延迟类型输入框
DelayedFloatField()	创建一个用于编辑 float 类型字段的延迟类型输入框
DelayedIntField()	创建一个用于编辑 int 类型字段的延迟类型输入框
DelayedTextField()	创建一个用于编辑 string 类型字段的延迟类型输入框
DoubleField()	创建一个用于编辑 double 类型字段的输入框
DrawPreviewTexture()	在矩形内绘制纹理
DrawRect()	在当前编辑器窗口中的指定位置以指定大小绘制一个着色的矩形
DrawTextureAlpha()	在矩形内绘制纹理的 Alpha 通道
DropdownButton()	创建一个用于打开下拉列表样式的按钮
DropShadowLabel()	绘制带有投影的文本
EndChangeCheck()	结束由 BeginChangeCheck 开启的代码块，并检查 GUI 是否发生了变更
EndDisabledGroup()	结束由 BeginDisabledGroup 开始的禁用组
EndFoldoutHeaderGroup()	结束由 BeginFoldoutHeaderGroup 开启的折叠栏
EndProperty()	结束由 BeginProperty 开始的属性封装器
EnumFlagsField()	创建一个用于编辑位掩码的控件
EnumPopup()	创建一个用于编辑枚举类型字段的控件
FloatField()	创建一个用于编辑 float 类型字段的输入框
FocusTextInControl()	将键盘焦点移动到指定的文本字段，并开始编辑内容
Foldout()	创建一个折叠栏
GetPropertyHeight()	获取 PropertyField 控件所需的高度
GradientField()	创建一个用于编辑 Gradient 类型字段的控件
HandlePrefixLabel()	创建一个显示在特定控件前的文本

续表

方　法	作　用
HelpBox()	创建一个带有发送给用户的消息的帮助框
InspectorTitlebar()	创建一个类似于 Inspector 窗口的标题栏
IntField()	创建一个用于编辑 int 类型字段的输入框
IntPopup()	创建一个用于弹出下拉列表的控件
IntSlider()	创建一个用于编辑 int 类型字段的滑动条
LabelField()	创建一个文本（用于显示只读信息）
LayerField()	创建一个用于编辑 Layer 层级的控件
LongField()	创建一个用于编辑 long 类型字段的输入框
MaskField()	创建一个用于编辑掩码的控件
MinMaxSlider()	创建一个滑动条
MultiFloatField()	创建一个用于编辑多个 float 类型字段的控件，用于在同一行中编辑多个浮点值
MultiIntField()	创建一个用于编辑多个 int 类型字段的控件，用于在同一行中编辑多个整数
MultiPropertyField()	创建一个用于编辑多个序列化属性的控件
ObjectField()	创建一个用于编辑对象的控件（通过拖曳对象或使用对象选择器分配对象）
PasswordField()	创建一个用于编辑密码类型文本的输入框
Popup()	创建一个用于弹出下拉列表的控件
PrefixLabel()	创建一个显示在特定控件前的文本
ProgressBar()	创建一个进度条
PropertyField()	创建一个用于编辑可序列化属性的控件
RectField()	创建一个用于编辑 Rect 类型字段的控件
RectIntField()	创建一个用于编辑 RectInt 类型字段的控件
SelectableLabel()	创建一个可选择的文本（用于显示可复制粘贴的只读信息）
Slider()	创建一个滑动条
TagField()	创建一个用于编辑物体 Tag 标签的控件
TextArea()	创建一个用于编辑 string 类型字段的文本域
TextField()	创建一个用于编辑 string 类型字段的输入框
Toggle()	创建一个开关
ToggleLeft()	创建一个开关，开关位于左侧，标签紧随其右
Vector2Field()	创建一个用于编辑 Vector2 类型字段的控件
Vector2IntField()	创建一个用于编辑 Vector2Int 类型字段的控件
Vector3Field()	创建一个用于编辑 Vector3 类型字段的控件
Vector3IntField()	创建一个用于编辑 Vector3Int 类型字段的控件
Vector4Field()	创建一个用于编辑 Vector4 类型字段的控件