Unity Editor Scripting ← Hyperlink

1. Generelles

Für Zugriff auf die Editor-API muss der UnityEditor-Namespace benutzt werden. Der UnityEditor-Namespace wird nicht bei Builds kompiliert, Skripte die diesen referenzieren werden nicht kompilierbar und der Build bricht ab.

Unity ignoriert bei Builds alle Ordner, die den Namen “Editor” tragen. Somit bietet es sich an, Editor-Skripte dort abzulegen. Skripte außerhalb eines “Editor”-Ordners können nicht auf Skripte innerhalb dieser zugreifen, Skripte innerhalb dieser Ordner haben jedoch Zugriff auf alle Skripte.

Eine weitere Alternative sind Compiler-Directives, doch die Methode “Editor”-Ordner ist meistens die bessere Wahl.

2. Editor Windows

Der Unity Editor besteht aus Editor Windows wie z.B. dem Scene View, dem Game View, der Hierarchy, der Konsole usw. Diese können frei bewegt, gedockt, geschlossen werden um das Layout des Editors anzupassen. Eigene Editor Windows kann man dadurch erstellen, indem man ein Skript von der Klasse `EditorWindow` erben lässt. Editor Windows sind `ScriptableObjects` und Instanzen sind so lange aktiv, wie ein dazugehöriges Window offen ist.

Um ein EditorWindow zu öffnen, gibt es zwei Wege, beides sind statische Methoden definiert in der Klasse EditorWindow. `EditorWindow.GetWindow<T>()`, wobei T der Typ des Windows ist, erstellt ein Fenster, wenn noch keines existiert oder fokussiert ein bereits vorhandenes. Die Funktion gibt die Instanz des Fensters zurück, sodass man dort Daten übergeben kann. Der zweite Weg ist `EditorWindow.CreateWindow<T>()`. Hierbei wird bei jedem Aufruf ein neues Fenster erstellt.

```
var windowInstance = EditorWindow.GetWindow<MyWindow>();
windowInstance.publicProperty = someValue;
```

In der Regel bietet es sich an, eine `public static void` Methode in der Klasse des jeweiligen EditorWindows zu deklarieren, um dieses von anderen Skripten aus öffnen zu können. Ist die Methode parameterlos kann sie zusätzlich mit einem MenuItem-Attribut versehen werden, was das Öffnen über ein Dropdown-Menu in Unity ermöglicht.

Ein EditorWindow bekommt, insofern die Methoden existieren, einen OnEnable-Call beim Öffnen und einen OnDisable-Call beim Schließen.

3. IMGUI und (Editor)GUILayout

Das Layout eines EditorWindows entsteht nicht durch eine Art von Markup Language und Stylesheet, sondern durch sukzessive programmatische Draw-Calls von Elementen. Dies trägt den Namen IMGUI und wurde in früheren Versionen von Unity sogar für das UI in Spielen verwendet. Was für Runtime-Scripts die Methode `Update` ist, ist bei IMGUI `OnGUI`. Falls benötigt, kann `Repaint` aufgerufen werden, um sofort das Fenster neu zu zeichnen.

Um einen Text darzustellen, kann man die Methode LabelField in EditorGUILayout benutzen:

```
EditorGUILayout.LabelField("hello world");
```

Weitere Parameter erlauben es, das Aussehen und Layout weiter anzupassen:

```
EditorGUILayout.LabelField(
    label: "hello world",
    style: EditorStyles.boldLabel,
    options: GUILayout.Width(200)  // you can add more options after this
);
```

Auffällig ist hier, dass neben EditorGUILayout auch GUILayout, welches im UnityEngine-Namespace liegt, benutzt wird. Wie erwähnt, wurde so früher das gesamte UI in Unity-Games geschrieben. Weitere Methoden ermöglichen es, im EditorWindow Felder für Strings, Zahlen, Assets und vieles mehr zu zeichnen.

```
string myString = string.Empty;
float myFloat = 0;
float myOtherFloat = 0;

void OnGUI () {
    myString = EditorGUILayout.TextField(myString);
    myFloat = EditorGUILayout.FloatField(myFloat);
    myOtherFloat = EditorGUILayout.Slider(myOtherFloat, -1, 1);
}
```

Einige wichtige Methoden:

`EditorGUILayout.LabelField` um Text darzustellen. Für mehrere Zeilen muss ein GUIStyle mit wordWrap = true verwendet werden.

`EditorGUILayout.Float/Int/Text/Object/.../Field` um Zahlen, Text, Objekte und weiteres einzugeben.

`GUILayout.Button` um einen Button zu zeichnen. Die Methode gibt true zurück, wenn dieser gedrückt wird.

`GUILayout.Space` um etwas leeren Raum einzufügen.

`EditorGUILayout.Begin/EndHorizontal` um Elemente seitlich nebeneinander anzuordnen.

`GUI.enabled, GUI.color, GUI.backgroundColor` um Interaktion zu deaktivieren oder die Farbe der gezeichneten Elemente anzupassen.

Zusätzlich zu den Begin-/End-Methoden gibt es oft auch Scopes in EditorGUI und EditorGUILayout, die den gleichen Effekt haben, jedoch mit etwas saubererem Code. Hier ein Beispiel:

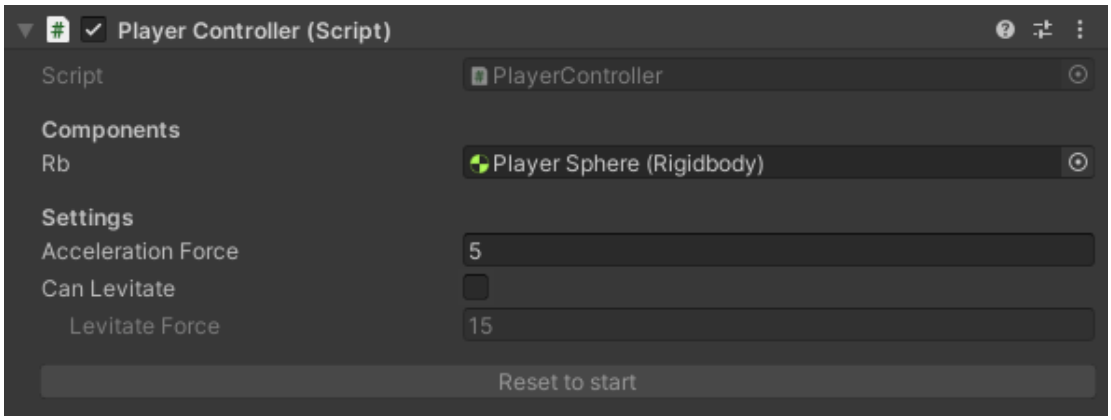
```
EditorGUILayout.BeginHorizontal();
// draw stuff
EditorGUILayout.EndHorizontal();
```

```
using(new EditorGUILayout.HorizontalScope()){
    // draw stuff
}
```

Im Projekt der Aufgabenstellung gibt es zusätzliche Scopes in den Klassen EditorTools und EditorGUITools.

4. Custom Editors

Die Darstellung von Components, zu denen MonoBehavious gehören, oder Scriptable Objects im Inspector Window wird durch einen Editor beschrieben. Mittels eines Custom Editors kann diese angepasst werden.



Ein Custom Editor für eine Klasse benötigt ein Skript, das von der Klasse Editor erbt und mit einem CustomEditor-Attribute versehen ist, wobei der Typ der Klasse, die den Editor bekommen soll, angegeben werden muss.

```
public class MyClass : MonoBehaviour {
    // gameplay code here
}

[CustomEditor(typeof(MyClass))]
public class MyEditor : Editor {
    // editor code here
}
```

In einem Editor kann man auch wieder OnEnable und OnDisable calls bekommen, doch statt OnGUI wird hier OnInspectorGUI zum Zeichnen der Elemente verwendet und muss als public override deklariert werden.

```
public override void OnInspectorGUI () {
    DrawDefaultInspector();
}
```

Die Property `target` vom Typ `UnityEngine.Object` enthält das Objekt, das bearbeitet wird. Sind die Felder, die man bearbeiten will public, so kann man diese wie zuvor mit EditorGUILayout zeichnen und Werte zuweisen. Allerdings werden somit vorgenommene Änderungen von Unity nicht als Änderungen des Assets registriert und können beim Schließen des Programms, dem Szenenwechsel oder vielen anderen Situationen verloren gehen. Mittels der Klasse Undo können Änderungen registriert und sogar revertierbar gemacht werden. Jedoch lassen sich somit, da der Editor eine separate Klasse vom editierten Objekt ist, nur public Fields und Properties anpassen.

Serialized Objects (nicht zu verwechseln mit Scriptable Objects) sind Repräsentationen der serialisierten Daten von Unity Objects (`UnityEngine.Object`). Der Konstruktor erlaubt es, von jedem Unity Object ein Serialized Object zu erstellen zur Bearbeitung *aller* serialisierten Felder mittels der Serialized Properties. `SerializedObject.Update` stellt sicher, dass die Serialized Properties die korrekten Werte haben und `SerializedObject.ApplyModifiedProperties` wendet alle Änderungen an , registriert die Änderung per Undo und markiert das Objekt als dirty, sodass Änderungen auch gespeichert werden.

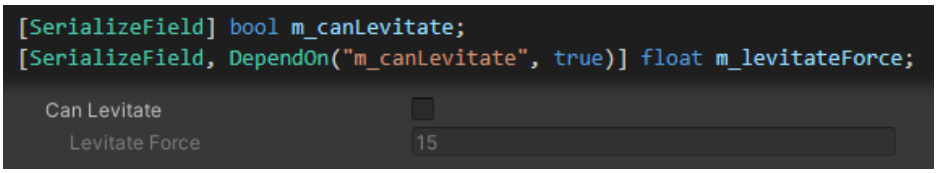
Die Klasse Editor bietet neben dem bearbeiteten Unity Object per `target` auch ein SerializedObject dessen über die Property `serializedObject`. Mit der Funktion SerializedObject.FindProperty können einzelne Properties bekommen werden. Die Werte sind über die verschiedenen value-Properties zugänglich.

```
public class MyClass : MonoBehaviour {
    [SerializeField] private int m_myInt;
}

[CustomEditor(typeof(MyClass))]
public class MyEditor : Editor {

    public override void OnInspectorGUI () {
        serializedObject.Update();
        var intProperty = serializedObject.FindProperty("m_myInt");
        intProperty.intValue = EditorGUILayout.IntField(intProperty.intValue);
        serializedObject.ApplyModifiedProperties();
    }

}
```

Zusätzlich kann hierbei von der Funktion `EditorGUILayout.PropertyField` Gebrauch gemacht werden. Diese zeichnet automatisch die korrekte Art von Feld inklusive der Attribute, die auf die Property angewandt wurden, wie zum Beispiel `Header`, `TextArea`, `Range` oder eigene Attribute.

Oftmals sind bei einem Custom Editor die Properties, die speziell behandelt werden in der Minderheit, und durch die hardgecodeten Propertynamen ist der Editor Code bei Änderungen in der Klasse des bearbeiteten Unity Objects unflexibel. Im Projekt der Aufgabenstellung finden Sie die Klassen `GenericEditor` und `GenericEditor<T>`, welche das Schreiben von Custom Editors per Iteration über die Serialized Properties deutlich erleichtern.

5. Custom Property Drawers



Wenn per `EditorGUILayout.PropertyField` oder `EditorGUI.PropertyField` eine SerializedProperty gezeichnet wird, benutzt Unity dafür einen Property Drawer. Wichtig hierbei ist, dass innerhalb von Property Drawers kein `EditorGUILayout` oder `GUILayout` verwendet werden sollte, sondern die Klassen `EditorGUI` und `GUI`. Dies erfordert Rects, um die Positionen und Größen der einzelnen Elemente explizit anzugeben, statt wie zuvor implizit z.B. mittels Horizontal Scopes . Die Manipulation von Rects ist oft umständlich, im Projekt der Aufgabenstellung bietet Ihnen die Klasse `EditorGUITools` diverse Möglichkeiten, Rects aufzuteilen.

Property Drawers müssen, wie Editors, mit einem Attribut versehen werden, das beschreibt, was dieser Property Drawer zeichnet. Der Zieltyp kann entweder ein PropertyAttribute sein, wie z.B. Range oder TextArea oder ein eigener Datentyp, der serialisierbar ist, bei der Aufgabenstellung wäre dies zum Beispiel die Klasse DialogueLine. Bei Property Drawers muss die Methode OnGUI(Rect, SerializedProperty, GUIContent) overriden werden. GetPropertyHeight kann auch überschrieben werden, sofern eine andere Höhe als eine Zeile gewünscht ist.

Um Properties einer Property zu bekommen, zum Beispiel im Falle einer serialisierten serialisierbaren Klasse oder Structs, kann man mit `SerializedProperty.FindPropertyRelative` bekommen.

```
public class MyClass : MonoBehaviour {
    [SerializeField] MyCompound m_myCompound;

    [System.Serializable]
    public class MyCompound {
        [SerializeField] int m_myInt;
        [SerializeField] float m_myFloat;
    }
}

[CustomPropertyDrawer(typeof(MyClass.MyCompound))]
public class MyCompoundDrawer : PropertyDrawer {
    public override void OnGUI (Rect position, SerializedProperty property, GUIContent label) {
        var intProp = property.FindPropertyRelative("m_myInt");
        var floatProp = property.FindPropertyRelative("m_myFloat");
        // ...
    }
}
```

Bei Custom Property Drawers für Attributes bekommt man per `attribute` das Attribute, muss es aber noch selbst casten, um die Werte eventueller Parameter zu erhalten.

```
public class MyClass : MonoBehaviour {
    [SerializeField, MyBool(true)] private int m_myInt;
}

public class MyBoolAttribute : PropertyAttribute {
    public readonly bool myBool;

    public MyBoolAttribute (bool boolValue) {
        this.myBool = boolValue;
    }
}

[CustomPropertyDrawer(typeof(MyBoolAttribute))]
public class MyBoolAttributeDrawer : PropertyDrawer {
    public override void OnGUI (Rect position, SerializedProperty property, GUIContent label) {
        var myAttr = (MyBoolAttribute)attribute;
        if(myAttr.myBool){
            // ...
        }else{
            // ...
        }
    }
}
```

Bei Serialized Properties gibt es einige Sachen zu beachten:

- Pro Property wird nur ein Property Drawer benutzt. Gibt es einen Property Drawer für eine Property, aber auch einen Property Drawer für ein Property Attribute, wird der Drawer für das Attribute benutzt. Bei mehreren Attributes mit Drawern wird nur eins benutzt.
- In der Regel reicht ein `EditorGUI.PropertyField` statt dem typspezifischen Feld aus.
- Unity serialisiert Enums über den `intValue` einer SerializedProperty. Dieser kann zum korrekten Enum gecastet werden. Die Property `SerializedProperty.enumValueIndex` zeigt stattdessen auf ein Feld in der Property `SerializedProperty.enumDisplayNames`.
- `SerializedProperty.objectReferenceValue` ist nur gültig für direkt serialisierte Unity Objects als Properties (GameObjects, MonoBehaviours, ScriptableObjects, ...)
- Sowohl bei Listen als auch bei Arrays als Serialized Property werden die Properties `arraySize`, `GetArrayElementAtIndex`, `MoveArrayElement` usw. zur Bearbeitung verwendet. `GetArrayElementAtIndex` gibt hierbei die Serialized Property für den Eintrag an der Stelle in der Collection.

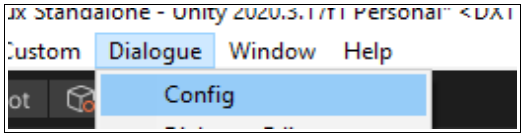
6. Aufgabenstellung

Ihre Aufgabe ist es, die Klasse `DialogueFileWindow` in ein Editor Window zu verwandeln, mit dem ein Dialogue File bearbeitet werden kann. Dabei muss ein Serialized Object des DialogueFiles erstellt werden und die Änderungen an den Serialized Properties dessen vorgenommen werden. Sie finden das Skript im Ordner *Scripts/Dialogue/Editor/Abgabe*. Legen Sie alle weiteren Skripte, die Sie erstellen auch in diesem Ordner ab und geben Sie ihn als Unitypackage exportiert ab.

Der Liste von Dialogue Lines sollen Elemente hinzugefügt und entfernt werden können, zudem sollen die Reihenfolge der Elemente veränderbar sein. Nur eine Dialogue Line soll zur gleichen Zeit bearbeitet werden können (nur ein Textfeld sichtbar). Versuchen Sie, das Fenster so zu designen, dass es benutzerfreundlich ist. Ein EditorGUI(Layout).PropertyField einer Dialogue Line reicht dafür nicht aus, ohne einen Custom Property Drawer zu implementieren. Alternativ können auch die Child-Properties der Dialogue Line direkt im Window gezeichnet werden.

Bei der Hintergrundmusik soll die bool-Property `m_loopBackgroundAudio` nur editierbar sein, wenn auch ein AudioClip für die Musik vorliegt.

Die Projektvorlage beinhaltet neben einigen Utilities auch Beispielwindows, -editors und -propertydrawers, die Sie als Referenz verwenden können.



Testen Sie, dass Ihre Version des DialogueFileWindow die Vorgaben erfüllt. Testen Sie auch, ob ein frisch erstelltes DialogueFile mit Ihrem Window editierbar ist, und beheben Sie Probleme, falls welche auftreten. Mit dem Config-Window können Sie einen neuen Dialog erstellen. In der Szene „Dialogue“ können Sie sich dann Ihr Resultat „ingame“ ansehen.

Letztlich ist hier eine Musterlösung, Ihr EditorWindow darf natürlich anders aussehen.