LERNEN

unity3d

Free unaffiliated eBook created from
Stack Overflow contributors.

#unity3d

Inhaltsverzeichnis

Über 1

Kapitel 1: Erste Schritte mit unity3d 2

Bemerkungen 2

Versionen 2

Examples 5

Installation oder Setup 5

Überblick 5

Installieren 6

Mehrere Versionen von Unity installieren 6

Grundlegender Editor und Code 6

Layout 6

Linux-Layout 7

Grundlegende Verwendung 7

Grundlegendes Scripting 8

Editor-Layouts 8

Anpassen Ihres Arbeitsbereichs 10

Kapitel 2: Android Plugins 101 - Eine Einführung 13

Einführung 13

Bemerkungen 13

Beginnend mit Android-Plugins 13

Übersicht über das Erstellen eines Plugins und einer Terminologie 13

Wahl zwischen den Plugin-Erstellungsmethoden 14

Examples 14

UnityAndroidPlugin.cs 14

UnityAndroidNative.java 14

UnityAndroidPluginGUI.cs 15

Kapitel 3: Animation der Einheit 16

Examples 16

Grundlegende Animation zum Laufen 16

Animationsclips erstellen und verwenden 17

2D-Sprite-Animation 19

Animationskurven 21

Kapitel 4: Anzeigenintegration 24

Einführung 24

Bemerkungen 24

Examples 24

Grundlagen der Unity-Anzeigen in C # 24

Unity-Anzeigen-Grundlagen in JavaScript 25

Kapitel 5: Asset Store 26

Examples 26

Zugriff auf den Asset Store 26

Kauf von Vermögenswerten 26

Assets importieren 27

Veröffentlichen von Assets 27

Bestätigen Sie die Rechnungsnummer eines Kaufs 28

Kapitel 6: Attribute 29

Syntax 29

Bemerkungen 29

SerializeField 29

Examples 30

Allgemeine Inspektorattribute 30

Komponentenattribute 32

Laufzeitattribute 33

Menüattribute 34

Editorattribute 36

Kapitel 7: Audiosystem 40

Einführung 40

Examples 40

Audioklasse - Audio abspielen 40

Kapitel 8: Coroutinen 41

Syntax 41

Bemerkungen 41

Überlegungen zur Leistung 41
Reduzieren Sie den Müll durch Zwischenspeichern von YieldInstructions 41
Examples 41
Coroutinen 41
Beispiel 43
Eine Coroutine beenden 43
MonoBehaviour-Methoden, die Coroutines sein können 45
Verketten von Coroutinen 45
Wege zum Nachgeben 47
Kapitel 9: CullingGroup-API 50
Bemerkungen 50
Examples 50
Entfernungen von Objekten abschneiden 50
Objektsichtbarkeit aussortieren 52
Begrenzte Entfernungen 53
Begrenzungsabstände visualisieren 53
Kapitel 10: Designmuster 55
Examples 55
Muster für Modellansicht-Controller (MVC) 55
Kapitel 11: Eingabesystem 59
Examples 59
Lesetaste drücken und Unterschied zwischen GetKey, GetKeyDown und GetKeyUp 59
Beschleunigungssensor lesen (Basic) 60
Beschleunigungssensor lesen (Advance) 61
Beschleunigungssensor lesen (Präzision) 61
Lesen Sie die Maustaste (Links, Mitte, Rechts) 62
Kapitel 12: Einheitsbeleuchtung 65
Examples 65
Arten von Licht 65
Flächenlicht 65
Richtungslicht 65

Punktlicht 66

Punktlicht 67

Notiz über Schatten 68

Emission 69

Kapitel 13: Erweitern des Editors 71

Syntax 71

Parameter 71

Examples 71

Benutzerdefinierter Inspektor 71

Benutzerdefinierte Eigenschaftsschublade 73

Menüpunkte 76

Gizmos 81

Beispiel eins 81

Beispiel zwei 83

Ergebnis 84

Nicht ausgewählt 84

Ausgewählt 84

Editor-Fenster 85

Warum ein Editorfenster? 85

Erstellen Sie ein einfaches Editorfenster 85

Einfaches Beispiel 85

Tiefer gehen 86

Fortgeschrittene Themen 89

In der SceneView zeichnen 89

Kapitel 14: GameObjects finden und sammeln 94

Syntax 94

Bemerkungen 94

Welche Methode ist zu verwenden? 94

Tiefer gehen 94

Examples 95

Suche nach dem Namen von GameObject 95

Suche nach den Tags von GameObject 95

Im Bearbeitungsmodus in Skripts eingefügt 95

Finden von GameObjects anhand von MonoBehaviour-Skripts 95

Finden Sie GameObjects nach Namen aus untergeordneten Objekten 96

Kapitel 15: Implementierung der MonoBehaviour-Klasse 97

Examples 97

Keine überschriebenen Methoden 97

Kapitel 16: Importeure und (Post-) Prozessoren 98

Syntax 98

Bemerkungen 98

Examples 98

Textur-Postprozessor 98

Ein einfacher Importeur 99

Kapitel 17: Kollision 103

Examples 103

Colliders 103

Box Collider 103

Eigenschaften 103

Beispiel 104

Sphere Collider 104

Eigenschaften 104

Beispiel 105

Capsule Collider 105

Eigenschaften 106

Beispiel 106

Wheel Collider 106

Eigenschaften 106

Federungsfeder 107

Beispiel 107

Mesh Collider 107

Eigenschaften 108

Beispiel 109
Wheel Collider 109
Trigger-Colliders 111
Methoden 111
Trigger Collider Scripting 112
Beispiel 112
Kapitel 18: Kommunikation mit dem Server 113
Examples 113
Erhalten 113
Einfache Buchung (Post-Felder) 113
Post (Datei hochladen) 114
Laden Sie eine Zip-Datei auf den Server hoch 114
Eine Anfrage an den Server senden 114
Kapitel 19: Mobile Plattformen 118
Syntax 118
Examples 118
Berührung erkennen 118
TouchPhase 118
Kapitel 20: Multiplattform-Entwicklung 120
Examples 120
Compiler-Definitionen 120
Plattformspezifische Methoden für Teilklassen organisieren 120
Kapitel 21: Objekt-Pooling 122
Examples 122
Objektpool 122
Einfacher Objektpool 125
Ein weiterer einfacher Objektpool 126
Kapitel 22: Optimierung 128
Bemerkungen 128
Examples 128
Schnelle und effiziente Prüfungen 128

Entfernungs- / Entfernungsprüfungen 128
Bounds Checks 128
Vorsichtsmaßnahmen 128
Coroutine Power 129
Verwendungszweck 129
Aufteilen langlaufender Routinen auf mehrere Frames 129
Kostspielige Aktionen seltener durchführen 129
Häufige Fehler 130
Zeichenketten 130
String-Operationen bauen Müll auf 130
Zwischenspeichern Sie Ihre Stringoperationen 130
Die meisten Stringoperationen sind Debug-Nachrichten 131
Stringvergleich 132
Referenzen zwischenspeichern 132
Vermeiden Sie den Aufruf von Methoden, die Strings verwenden 133
Vermeiden Sie leere Einheitsmethoden 134
Kapitel 23: Physik 135
Examples 135
Starre Körper 135
Überblick 135
Eine Rigidbody-Komponente hinzufügen 135
Verschieben eines Rigidbody-Objekts 135
Masse 135
Ziehen 135
isKinematic 136
Einschränkungen 136
Kollisionen 136
Schwerkraft im starren Körper 137
Kapitel 24: Prefabs 139
Syntax 139

Examples 139
Einführung 139
Prefabs erstellen 139
Prefab-Inspektor 140
Prefabs instanziieren 141
Design-Zeitinstanziierung 141
Laufzeit-Instantiierung 142
Verschachtelte Prefabs 142
Kapitel 25: Quaternionen 147
Syntax 147
Examples 147
Einführung zu Quaternion vs Euler 147
Quaternion Look Rotation 147
Kapitel 26: Raycast 149
Parameter 149
Examples 149
Physik-Raycast 149
Physics2D Raycast2D 149
Raycast-Anrufe kapseln 150
Lesen Sie weiter 151
Kapitel 27: Ressourcen 152
Examples 152
Einführung 152
Ressourcen 101 152
Einführung 152
Alles zusammen setzen 153
Abschließende Anmerkungen 153
Kapitel 28: Schichten 155
Examples 155
Layer-Nutzung 155
LayerMask-Struktur 155

Kapitel 29: ScriptableObject 157
Bemerkungen 157
ScriptableObjects mit AssetBundles 157
Examples 157
Einführung 157
ScriptableObject-Assets erstellen 157
Erstellen Sie ScriptableObject-Instanzen durch Code 158
ScriptableObjects werden auch im PlayMode im Editor serialisiert 158
Vorhandene ScriptableObjects zur Laufzeit finden 159
Kapitel 30: Singletons in der Einheit 160
Bemerkungen 160
Lesen Sie weiter 160
Examples 161
Implementierung mit RuntimeInitializeOnLoadMethodAttribute 161
Ein einfaches Singleton MonoBehaviour in Unity C # 161
Advanced Unity Singleton 162
Singleton Implementierung durch Basisklasse 165
Singleton-Pattern mit Unitys Entity-Component-System 166
MonoBehaviour & ScriptableObject-basierte Singleton-Klasse 167
Kapitel 31: So verwenden Sie Asset-Pakete 172
Examples 172
Asset-Pakete 172
.Unitypackage importieren 172
Kapitel 32: Sofortiges grafisches Benutzeroberflächensystem (IMGUI) 174
Syntax 174
Examples 174
GUILayout 174
Kapitel 33: Stichworte 175
Einführung 175
Examples 175
Tags erstellen und anwenden 175

Tags im Editor einstellen 175
Tags über Skript einstellen 175
Benutzerdefinierte Tags erstellen 176
Suche nach GameObjects nach Tag: 177
Ein einzelnes GameObject 177
Suchen eines Arrays von GameObject Instanzen 178
Tags vergleichen 178
Kapitel 34: Unity Profiler 180
Bemerkungen 180
Verwenden des Profilers auf einem anderen Gerät 180
Android 180
iOS 181
Examples 181
Profiler Markup 181
Verwenden der Profiler- Klasse 181
Kapitel 35: User Interface System (UI) 183
Examples 183
Ereignis im Code abonnieren 183
Hinzufügen von Maus-Listenern 183
Kapitel 36: Vector3 185
Einführung 185
Syntax 185
Examples 185
Statische Werte 185
Vector3.zero und Vector3.one 185
Statische Anweisungen 186
Index 188
Einen Vector3 erstellen 188
Konstrukteure 188
Konvertieren von einem Vector2 oder Vector4 189
Bewegung anwenden 189

Lerp und LerpUnclamped....189
MoveTowards....191
SmoothDamp....192
Kapitel 37: Vernetzung....195
Bemerkungen....195
Headless-Modus in Unity....195
Examples....196
Erstellen eines Servers, eines Clients und Senden einer Nachricht....196
Die Klasse, die wir zur Serialisierung verwenden....196
Server erstellen....196
Der Kunde....198
Kapitel 38: Verwandelt sich....200
Syntax....200
Examples....200
Überblick....200
Erziehung und Kinder....201
Kapitel 39: Verwenden der Git-Quellcodeverwaltung mit Unity....203
Examples....203
Verwenden von Git Large File Storage (LFS) mit Unity....203
Vorwort....203
Git & Git-LFS installieren....203
Option 1: Verwenden Sie eine Git-GUI-Anwendung....203
Option 2: Installieren Sie Git & Git-LFS....204
Git Large File Storage für Ihr Projekt konfigurieren....204
Einrichten eines Git-Repositorys für Unity....204
Ordner ignorieren....204
Unity-Projekteinstellungen....205
Zusätzliche Konfiguration....205
Szenen und Prefabs zusammenführen....206
Kapitel 40: Virtuelle Realität (VR)....207
Examples....207

VR-Plattformen 207
SDKs: 207
Dokumentation: 207
Aktivieren der VR-Unterstützung 207
Hardware 208
Credits **210**

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: unity3d

It is an unofficial and free unity3d ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official unity3d.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit unity3d

Bemerkungen

Unity bietet eine plattformübergreifende Spieleentwicklungsumgebung für Entwickler. Entwickler können für die Programmierung des Spiels C # -Sprache und / oder JavaScript-basierte UnityScript verwenden. Target Deployment-Plattformen können einfach im Editor gewechselt werden. Der Kerncode des Spiels bleibt bis auf einige plattformabhängige Funktionen gleich. Eine Liste aller Versionen sowie die entsprechenden Downloads und Versionshinweise finden Sie hier: https://unity3d.com/get-unity/download/archive .

Versionen

Ausführung	Veröffentlichungsdatum
Einheit 2017.1.0	2017-07-10
5.6.2	2017-06-21
5.6.1	2017-05-11
5.6.0	2017-03-31
5.5.3	2017-03-31
5.5.2	2017-02-24
5.5.1	2017-01-24
5.5	2016-11-30
5.4.3	2016-11-17
5.4.2	2016-10-21
5.4.1	2016-09-08
5.4.0	2016-07-28
5.3.6	2016-07-20
5.3.5	2016-05-20
5.3.4	2016-03-15
5.3.3	2016-02-23

Ausführung	Veröffentlichungsdatum
5.3.2	2016-01-28
5.3.1	2015-12-18
5.3.0	2015-12-08
5.2.5	2016-06-01
5.2.4	2015-12-16
5.2.3	2015-11-19
5.2.2	2015-10-21
5.2.1	2015-09-22
5.2.0	2015-09-08
5.1.5	2015-06-07
5.1.4	2015-10-06
5.1.3	2015-08-24
5.1.2	2015-07-16
5.1.1	2015-06-18
5.1.0	2015-06-09
5.0.4	2015-07-06
5.0.3	2015-06-09
5.0.2	2015-05-13
5.0.1	01.04.2015
5.0.0	2015-03-03
4.7.2	2016-05-31
4.7.1	2016-02-25
4.7.0	2015-12-17
4.6.9	2015-10-15
4.6.8	2015-08-26

Ausführung	Veröffentlichungsdatum
4.6.7	2015-07-01
4.6.6	08.06.2015
4.6.5	2015-04-30
4.6.4	2015-03-26
4.6.3	2015-02-19
4.6.2	2015-01-29
4.6.1	2014-12-09
4.6.0	2014-11-25
4.5.5	2014-10-13
4.5.4	2014-09-11
4.5.3	2014-08-12
4.5.2	2014-07-10
4.5.1	2014-06-12
4.5.0	2014-05-27
4.3.4	2014-01-29
4.3.3	2014-01-13
4.3.2	2013-12-18
4.3.1	2013-11-28
4.3.0	2013-11-12
4.2.2	2013-10-10
4.2.1	2013-09-05
4.2.0	2013-07-22
4.1.5	2013-06-08
4.1.4	2013-06-06
4.1.3	2013-05-23

Ausführung	Veröffentlichungsdatum
4.1.2	2013-03-26
4.1.0	2013-03-13
4.0.1	2013-01-12
4.0.0	2012-11-13
3.5.7	2012-12-14
3.5.6	2012-09-27
3.5.5	2012-08-08
3.5.4	2012-07-20
3.5.3	2012-06-30
3.5.2	2012-05-15
3.5.1	2012-04-12
3.5.0	2012-02-14
3.4.2	2011-10-26
3.4.1	2011-09-20
3.4.0	2011-07-26

Examples

Installation oder Setup

Überblick

Unity läuft unter Windows und Mac. Es gibt auch eine Linux-Alpha-Version .

Es gibt 4 verschiedene Zahlungspläne für Unity:

1. **Persönlich** - Kostenlos *(siehe unten)*
2. **Plus** - 35 USD pro Monat pro Sitz *(siehe unten)*
3. **Pro** - $ 125 USD pro Monat pro Sitzplatz - Nachdem Sie den Pro-Plan für 24 aufeinanderfolgende Monate abonniert haben, haben Sie die Möglichkeit, das Abonnement zu beenden und Ihre Version beizubehalten.
4. **Enterprise** - Wenden Sie sich an Unity, um weitere Informationen zu erhalten

*Laut EULA: Unternehmen oder Kapitalgesellschaften, die im letzten Geschäftsjahr einen Umsatz von mehr als 100.000 USD erzielt haben, müssen **Unity Plus** (oder eine höhere Lizenz) verwenden. über 200.000 US-Dollar müssen sie **Unity Pro** (oder Enterprise) verwenden.*

Installieren

1. Laden Sie den Unity-Download-Assistenten herunter .
2. Führen Sie den Assistenten aus und wählen Sie die Module aus, die Sie herunterladen und installieren möchten, z. B. Unity-Editor, MonoDevelop-IDE, Dokumentation und gewünschte Plattformerstellungsmodule.

Wenn Sie eine ältere Version haben, können Sie auf die neueste stabile Version aktualisieren .

Wenn Sie Unity ohne den Unity-Download-Assistenten installieren möchten, erhalten Sie die Installationskomponenten der **Komponenten in den** Versionshinweisen zu Unity 5.5.1 .

Mehrere Versionen von Unity installieren

Es ist oft erforderlich, mehrere Versionen von Unity gleichzeitig zu installieren. Um dies zu tun:

- Ändern Sie unter Windows das Standardinstallationsverzeichnis in einen leeren Ordner, den Sie zuvor erstellt haben, z. B. `Unity 5.3.1f1` .
- Auf einem Mac wird das Installationsprogramm immer unter `/Applications/Unity` installiert. Benennen Sie diesen Ordner für Ihre vorhandene Installation um (z. B. in `/Applications/Unity5.3.1f1`), bevor Sie das Installationsprogramm für die andere Version `/Applications/Unity5.3.1f1` .
- Sie können beim Starten von Unity die `Alt-` Taste gedrückt halten, um die Auswahl eines zu öffnenden Projekts zu erzwingen. Andernfalls wird versucht, das letzte geladene Projekt zu laden (sofern verfügbar), und Sie werden möglicherweise aufgefordert, ein Projekt zu aktualisieren, das Sie nicht aktualisieren möchten.

Grundlegender Editor und Code

Layout

Der grundlegende Editor von Unity wird wie folgt aussehen. Die grundlegenden Funktionen einiger Standardfenster / Registerkarten werden im Bild beschrieben.

Objects that will be used in a scene are placed here to show in game world. Example: Light, Camera objects in this scene.

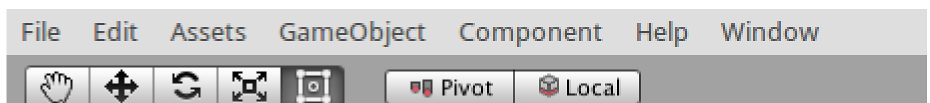
When "Play" is pressed "Game View to show the objects in the scene th camera.

This is game world or Scene view w physically placed. Example: For des assets are placed here physically as interation etc in the game.

All game assets are placed here. Example: Mo textures, prefabricated objects, scenes, script materials etc are located here for using in the

Linux-Layout

Es gibt einen kleinen Unterschied im Menü-Layout der Linux-Version, wie in der Abbildung unten.



Grundlegende Verwendung

Erstellen Sie ein leeres `GameObject` indem Sie mit der rechten Maustaste in das Hierarchiefenster klicken und `Create Empty` auswählen. Erstellen Sie ein neues Skript, indem Sie mit der rechten Maustaste in das Projektfenster klicken und `Create` > `C# Script` auswählen. Benennen Sie es nach Bedarf um.

Wenn das leere `GameObject` im Hierarchiefenster ausgewählt ist, ziehen Sie das neu erstellte Skript per Drag & Drop in das Inspektorfenster. Jetzt wird das Skript im Hierarchiefenster an das Objekt angehängt. Öffnen Sie das Skript mit der Standard-MonoDevelop-IDE oder Ihren Wünschen.

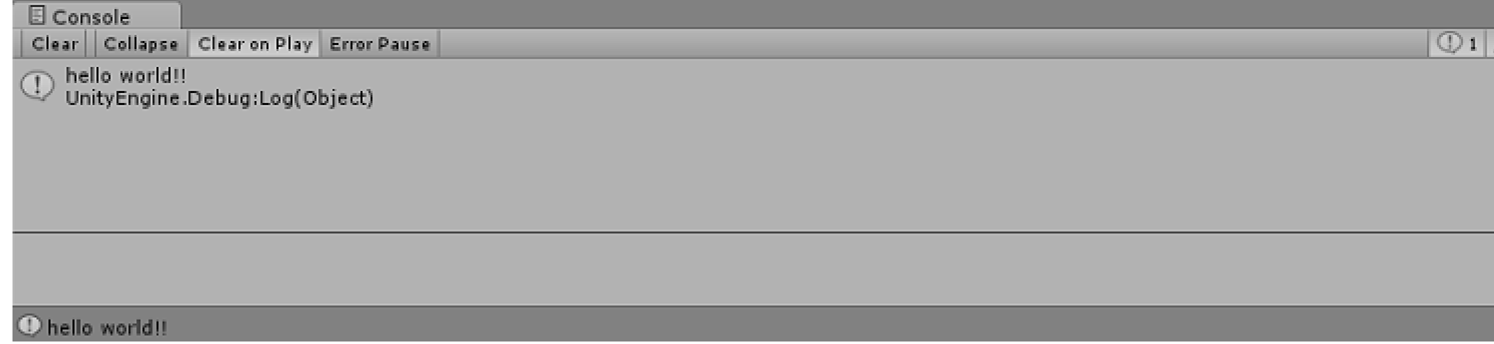
Grundlegendes Scripting

Der grundlegende Code sieht wie folgt aus, mit Ausnahme der Zeile `Debug.Log("hello world!!");` .

```
using UnityEngine;
using System.Collections;

public class BasicCode : MonoBehaviour {

    // Use this for initialization
    void Start () {
        Debug.Log("hello world!!");
    }

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

    }
}
```

Fügen Sie die Zeile `Debug.Log("hello world!!");` in der `void Start()` -Methode. Speichern Sie das Skript und kehren Sie zum Editor zurück. Starten Sie es, indem Sie oben im Editor auf **Play** drücken.

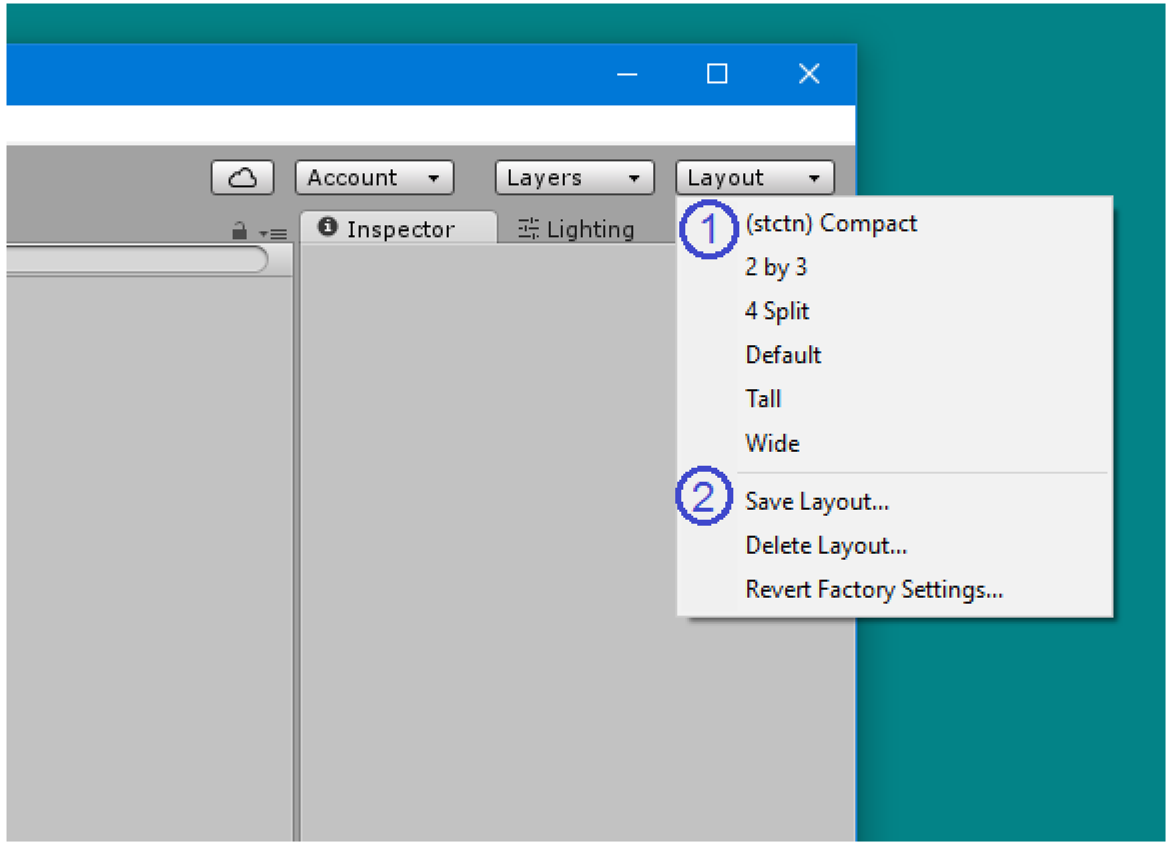
Das Ergebnis sollte im Konsolenfenster wie folgt aussehen:



Editor-Layouts

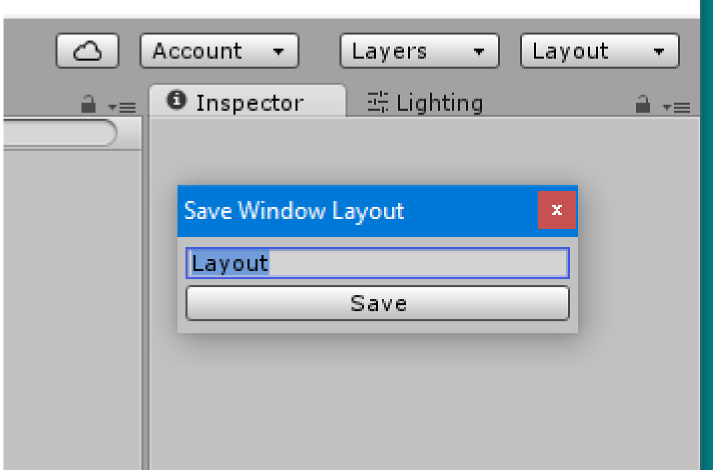
Sie können das Layout Ihrer Registerkarten und Fenster speichern, um Ihre Arbeitsumgebung zu standardisieren.

Das Layout-Menü befindet sich in der oberen rechten Ecke des Unity Editors:

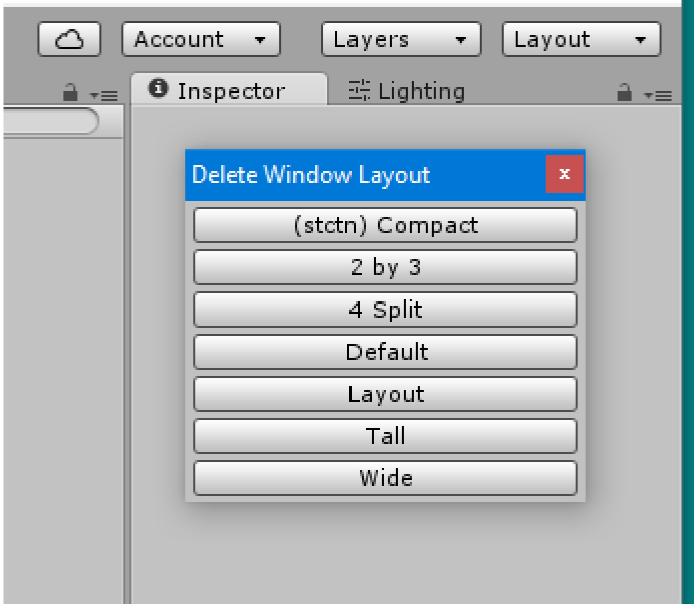


Unity wird mit 5 Standardlayouts geliefert (2 x 3, 4 geteilt, Standard, hoch, breit) *(mit 1 markiert)* . In der Abbildung oben ist neben den Standardlayouts auch ein benutzerdefiniertes Layout oben zu sehen.

Sie können Ihre eigenen Layouts hinzufügen, indem Sie im Menü auf die Schaltfläche "**Layout speichern**" klicken *(mit 2 gekennzeichnet)* :



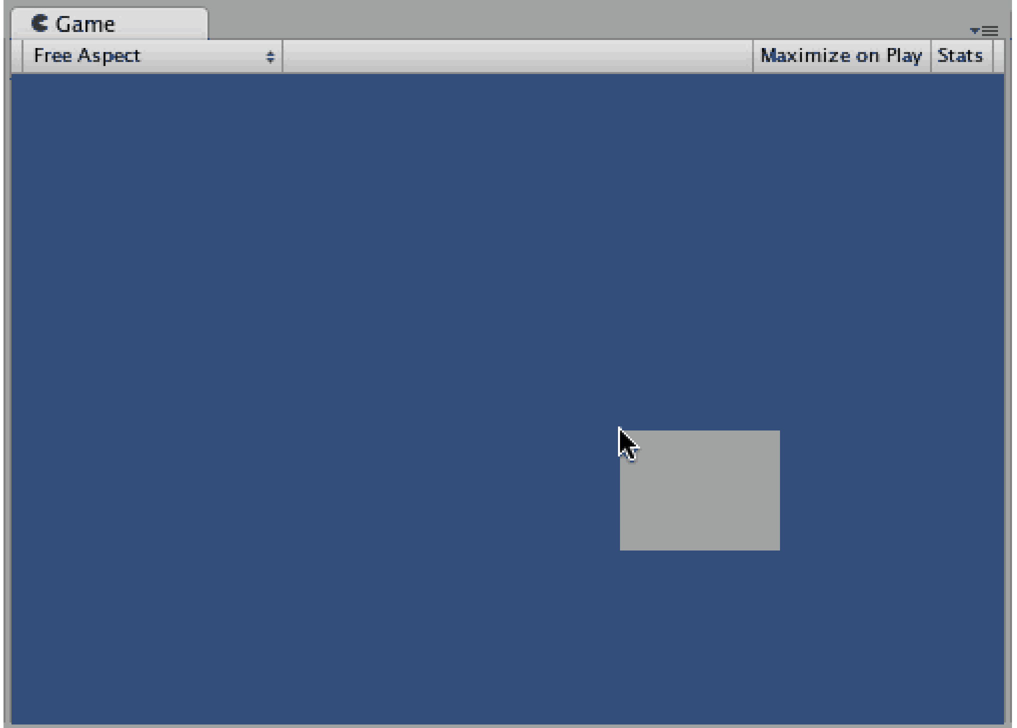
Sie können jedes Layout auch löschen, indem Sie im Menü auf die Schaltfläche "**Layout** löschen " klicken *(mit 2 gekennzeichnet)* :



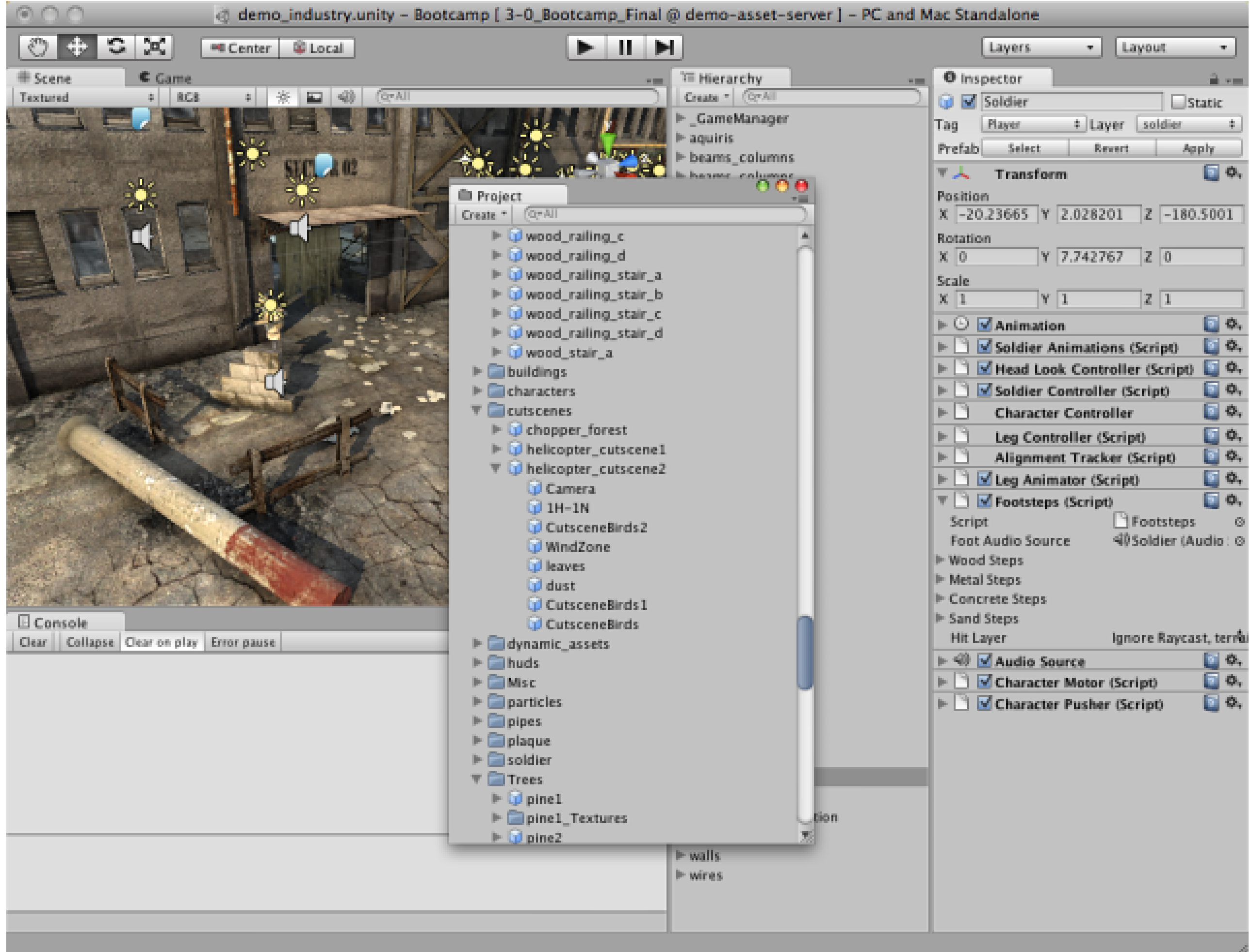
Mit der Schaltfläche "**Werkseinstellungen** wiederherstellen **...**" **werden** alle benutzerdefinierten Layouts entfernt und die Standardlayouts *(mit 2 gekennzeichnet)* wiederhergestellt.

Anpassen Ihres Arbeitsbereichs

Sie können das Layout der Ansichten anpassen, indem Sie die Tabulatortaste einer beliebigen Ansicht durch Ziehen an eine von mehreren Stellen ziehen. Durch das Ablegen einer Registerkarte im Registerkartenbereich eines vorhandenen Fensters wird die Registerkarte neben den vorhandenen Registerkarten hinzugefügt. Wenn Sie einen Tab in einer beliebigen Dockzone ablegen, wird die Ansicht in einem neuen Fenster hinzugefügt.

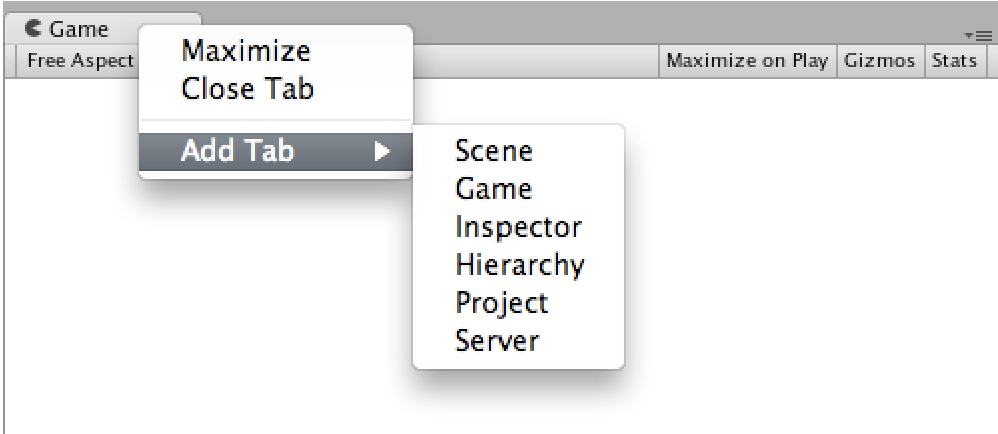


Registerkarten können auch vom Haupteditorfenster getrennt und in einem eigenen schwebenden Editorfenster angeordnet werden. Floating Windows kann wie das Haupteditorfenster Anordnungen von Ansichten und Registerkarten enthalten.



Wenn Sie ein Editor-Layout erstellt haben, können Sie das Layout speichern und jederzeit wiederherstellen. In diesem Beispiel finden Sie Editorlayouts .

Sie können jederzeit mit der rechten Maustaste auf die Registerkarte einer beliebigen Ansicht klicken, um zusätzliche Optionen wie Maximieren anzuzeigen oder eine neue Registerkarte zu demselben Fenster hinzuzufügen.



Erste Schritte mit unity3d online lesen: https://riptutorial.com/de/unity3d/topic/846/erste-schritte-mit-unity3d

Kapitel 2: Android Plugins 101 - Eine Einführung

Einführung

Dieses Thema ist der erste Teil einer Serie zum Erstellen von Android-Plugins für Unity. Beginnen Sie hier, wenn Sie noch wenig oder keine Erfahrung mit der Erstellung von Plug-ins und / oder dem Android-Betriebssystem haben.

Bemerkungen

In dieser Serie verwende ich ausgiebig externe Links, die ich zum Lesen anregen möchte. Zwar werden hier paraphrasierte Versionen des relevanten Inhalts eingefügt, aber es kann vorkommen, dass die zusätzliche Lektüre hilfreich ist.

Beginnend mit Android-Plugins

Derzeit bietet Unity zwei Möglichkeiten, native Android-Codes aufzurufen.

1. Schreiben Sie nativen Android-Code in Java und rufen Sie diese Java-Funktionen mit C # auf.
2. Schreiben Sie C # -Code, um direkt Funktionen des Android-Betriebssystems aufzurufen

Um mit nativem Code zu interagieren, stellt Unity einige Klassen und Funktionen bereit.

- AndroidJavaObject - Dies ist die Basisklasse, die Unity für die Interaktion mit nativem Code bereitstellt. Nahezu jedes Objekt, das aus nativem Code zurückgegeben wird, kann als AndroidJavaObject gespeichert werden
- AndroidJavaClass - Inherits from AndroidJavaObject. Dies wird verwendet, um Klassen in Ihrem nativen Code zu referenzieren
- Abrufen / Festlegen von Werten einer Instanz eines nativen Objekts und der statischen GetStatic / SetStatic- Versionen
- Rufen Sie / CallStatic auf, um native nicht statische und statische Funktionen aufzurufen

---

Übersicht über das Erstellen eines Plugins und einer Terminologie

1. Schreiben Sie nativen Java-Code in Android Studio
2. Exportieren Sie den Code in eine JAR / AAR-Datei (Schritte hier für JAR-Dateien und AAR-Dateien)

3. Kopieren Sie die JAR / AAR-Datei in Ihr Unity-Projekt unter **Assets / Plugins / Android**
4. Schreiben Sie Code in Unity (C # war hier immer der Weg), um Funktionen im Plugin aufzurufen

Beachten Sie, dass die ersten drei Schritte NUR gelten, wenn Sie ein natives Plugin haben möchten!

Von jetzt an verweise ich auf die JAR / AAR-Datei als **natives Plugin** und das C # -Skript als **C # -wrapper**

Wahl zwischen den Plugin-Erstellungsmethoden

Es ist sofort offensichtlich, dass die erste Möglichkeit, Plugins zu erstellen, lange dauert, so dass die Auswahl Ihrer Route ziemlich umstritten ist. Methode 1 ist jedoch die EINZIGE Möglichkeit, benutzerdefinierten Code aufzurufen. Wie wählt man also aus?

Einfach gesagt, macht dein Plugin

1. Benutzerdefinierten Code einbeziehen - Wählen Sie Methode 1 aus
2. Nur native Android-Funktionen aufrufen? - Wählen Sie Methode 2

Versuchen Sie **NICHT** , die beiden Methoden zu "mischen" (dh einen Teil des Plugins mit Methode 1 und den anderen mit Methode 2)! Obwohl durchaus möglich, ist es oft unpraktisch und schmerzhaft zu handhaben.

Examples

UnityAndroidPlugin.cs

Erstellen Sie ein neues C # -Skript in Unity und ersetzen Sie dessen Inhalt durch Folgendes

```
using UnityEngine;
using System.Collections;

public static class UnityAndroidPlugin {

}
```

UnityAndroidNative.java

Erstellen Sie eine neue Java-Klasse in Android Studio und ersetzen Sie deren Inhalt durch Folgendes

```
package com.axs.unityandroidplugin;
import android.util.Log;
import android.widget.Toast;
```

```
import android.app.ActivityManager;
import android.content.Context;


public class UnityAndroidNative {


}
```

UnityAndroidPluginGUI.cs

Erstellen Sie ein neues C # -Skript in Unity und fügen Sie diese Inhalte ein

```
using UnityEngine;
using System.Collections;

public class UnityAndroidPluginGUI : MonoBehaviour {

    void OnGUI () {

    }

}
```

Android Plugins 101 - Eine Einführung online lesen:
https://riptutorial.com/de/unity3d/topic/10032/android-plugins-101---eine-einfuhrung

Kapitel 3: Animation der Einheit

Examples

Grundlegende Animation zum Laufen

Dieser Code zeigt ein einfaches Beispiel für eine Animation in Unity.

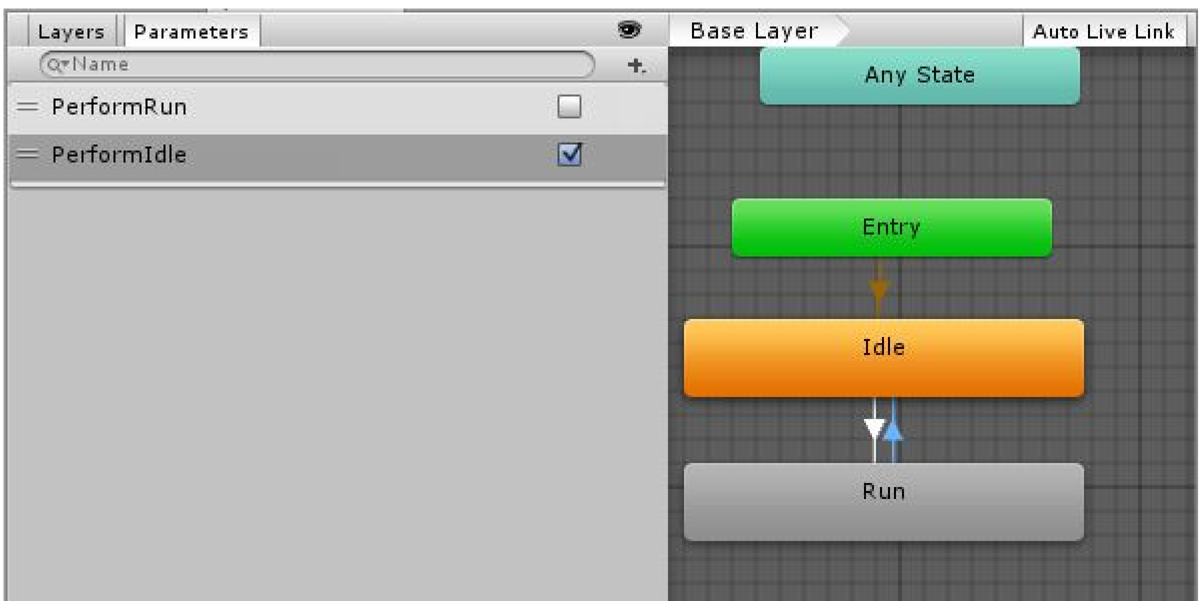
Für dieses Beispiel sollten Sie zwei Animationsclips haben. Run and Idle. Diese Animationen sollten Stand-In-Place-Bewegungen sein. Wenn Sie die Animationsclips ausgewählt haben, erstellen Sie einen Animator-Controller. Fügen Sie diesen Controller dem Spieler oder Spielobjekt hinzu, das Sie animieren möchten.

Öffnen Sie das Animator-Fenster von Windows aus. Ziehen Sie die 2 Animationsclips in das Animator-Fenster. Es werden 2 Status erstellt. Nach dem Erstellen können Sie auf der linken Registerkarte "Parameter" zwei Parameter hinzufügen, beide als Bool. Nennen Sie einen als "PerformRun" und einen anderen als "PerformIdle". Setzen Sie "PerformIdle" auf "true".

Machen Sie Übergänge vom Ruhezustand zu Run und Run to Idle (Siehe Abbildung). Klicken Sie auf Leerlauf-> Übergang ausführen, und deaktivieren Sie HasExit im Inspektorfenster. Machen Sie dasselbe für den anderen Übergang. Fügen Sie für Idle-> Run-Übergang eine Bedingung hinzu: PerformIdle. Fügen Sie für Run-> Idle eine Bedingung hinzu: PerformRun. Fügen Sie das unten angegebene C # -Skript zum Spielobjekt hinzu. Es sollte mit der Aufwärts-Taste mit Animation laufen und mit den Links- und Rechts-Tasten drehen.

```
using UnityEngine;
using System.Collections;

public class RootMotion : MonoBehaviour {

//Public Variables
[Header("Transform Variables")]
public float RunSpeed = 0.1f;
public float TurnSpeed = 6.0f;


Animator animator;

void Start()
{
    /**
    * Initialize the animator that is attached on the current game object i.e. on which you
will attach this script.
    */
    animator = GetComponent<Animator>();
}

void Update()
{
    /**
    * The Update() function will get the bool parameters from the animator state machine and
set the values provided by the user.
```

```
     * Here, I have only added animation for Run and Idle. When the Up key is pressed, Run
animation is played. When we let go, Idle is played.
     */

     if (Input.GetKey (KeyCode.UpArrow)) {
               animator.SetBool ("PerformRun", true);
               animator.SetBool ("PerformIdle", false);
          } else {
               animator.SetBool ("PerformRun", false);
               animator.SetBool ("PerformIdle", true);
          }
}

void OnAnimatorMove()
     {
          /**
           * OnAnimatorMove() function will shadow the "Apply Root Motion" on the animator. Your
game objects psoition will now be determined
           * using this fucntion.
           */
          if (Input.GetKey (KeyCode.UpArrow)){
               transform.Translate (Vector3.forward * RunSpeed);
               if (Input.GetKey (KeyCode.RightArrow)) {
                    transform.Rotate (Vector3.up * Time.deltaTime * TurnSpeed);
               }
               else if (Input.GetKey (KeyCode.LeftArrow)) {
                    transform.Rotate (-Vector3.up * Time.deltaTime * TurnSpeed);
               }
          }

     }



}
```

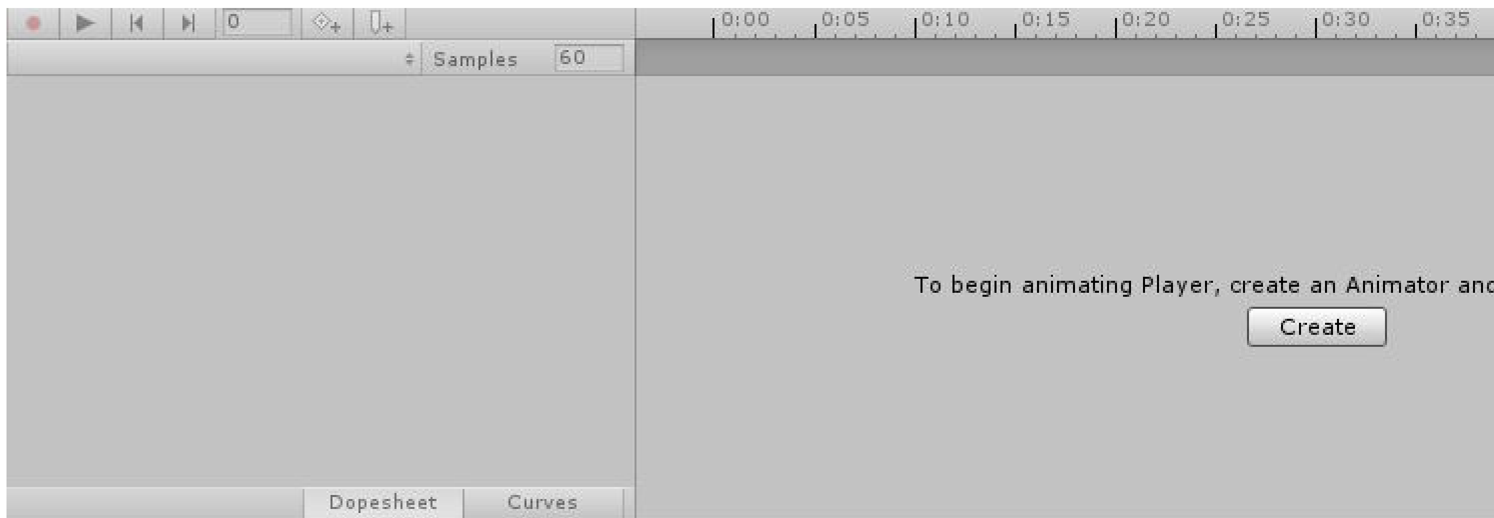


Animationsclips erstellen und verwenden

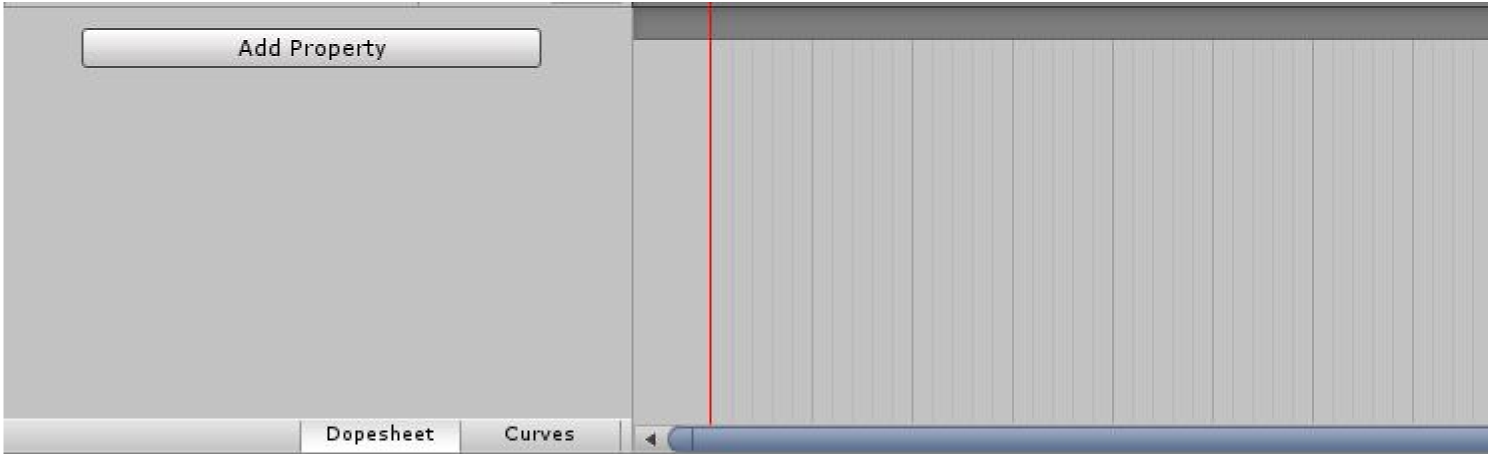
Dieses Beispiel zeigt, wie Animationsclips für Spielobjekte oder Spieler erstellt und verwendet werden.

Beachten Sie, dass die in diesem Beispiel verwendeten Modelle vom Unity Asset Store heruntergeladen werden. Der Player wurde unter folgendem Link heruntergeladen: https://www.assetstore.unity3d.com/de/#!/content/21874 .

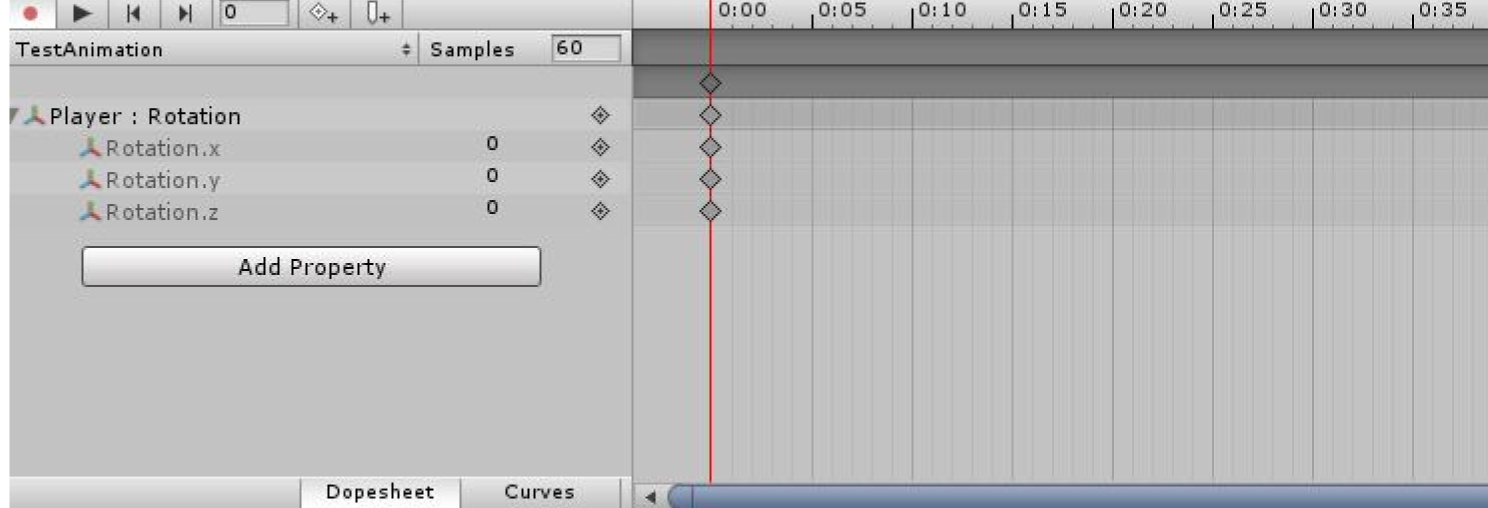
Um Animationen zu erstellen, öffnen Sie zuerst das Animationsfenster. Sie können es öffnen, indem Sie auf Fenster und Animation auswählen klicken oder Strg + 6 drücken. Wählen Sie im Hierarchiefenster das Spielobjekt aus, auf das Sie den Animationsclip anwenden möchten, und klicken Sie dann im Animationsfenster auf die Schaltfläche Erstellen.



Benennen Sie Ihre Animation (wie IdlePlayer, SprintPlayer, DyingPlayer usw.) und speichern Sie sie. Klicken Sie nun im Animationsfenster auf die Schaltfläche Eigenschaft hinzufügen. Auf diese Weise können Sie die Eigenschaften des Spielobjekts oder Spielers zeitlich ändern. Dies kann Transformationsformen wie Rotation, Position und Skalierung sowie andere Eigenschaften enthalten, die an das Spielobjekt angehängt sind, z. B. Collider, Mesh Renderer usw.



Um eine laufende Animation für ein Spielobjekt zu erstellen, benötigen Sie ein humanoides 3D-Modell. Sie können das Modell über den obigen Link herunterladen. Befolgen Sie die obigen Schritte, um eine neue Animation zu erstellen. Fügen Sie eine Transformationseigenschaft hinzu und wählen Sie Rotation für einen der Charakterbeine.



In diesem Moment wären Ihre Play-Schaltfläche und die Rotation-Werte in der Spielobjekteigenschaft rot. Klicken Sie auf den Dropdown-Pfeil, um die X-, Y- und Z-Werte für die Drehung anzuzeigen. Die Standardanimationszeit ist auf 1 Sekunde eingestellt. Animationen verwenden Keyframes, um zwischen Werten zu interpolieren. Fügen Sie zum Animieren Schlüssel zu verschiedenen Zeitpunkten hinzu und ändern Sie die Rotationswerte im Inspektorfenster. Zum Beispiel kann der Rotationswert zum Zeitpunkt 0,0s 0,0 sein. Zur Zeit 0,5 s kann der Wert für X 20,0 sein. Zur Zeit 1,0 s kann der Wert 0,0 sein. Wir können unsere Animation bei 1.0s beenden.

Ihre Animationslänge hängt von den letzten Schlüsseln ab, die Sie der Animation hinzufügen. Sie können weitere Tasten hinzufügen, um die Interpolation weicher zu gestalten.

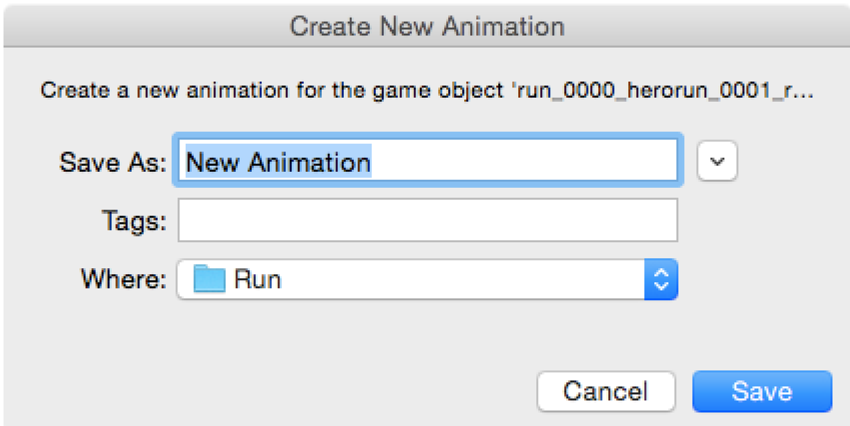
2D-Sprite-Animation

Bei der Sprite-Animation wird eine vorhandene Folge von Bildern oder Frames angezeigt.

Importieren Sie zunächst eine Folge von Bildern in den Asset-Ordner. Erstellen Sie entweder einige Bilder von Grund auf oder laden Sie sie im Asset Store herunter. (In diesem Beispiel wird dieses kostenlose Asset verwendet .)

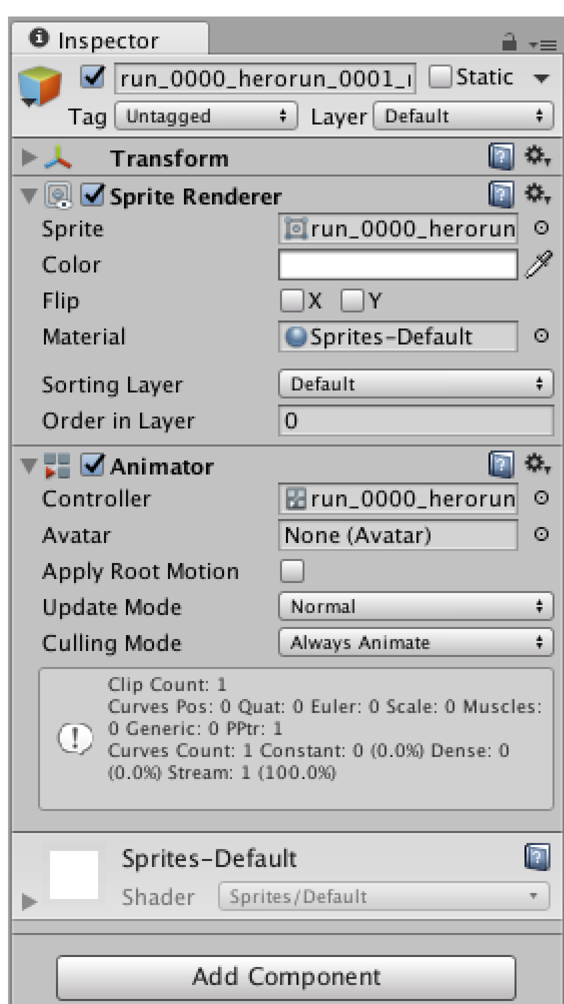
run_0000_herorun_0001_run_anim_001.png
run_0001_herorun_0002_run_anim_002.png
run_0002_herorun_0003_run_anim_003.png
run_0003_herorun_0004_run_anim_004.png
run_0004_herorun_0005_run_anim_005.png
run_0005_herorun_0006_run_anim_006.png
run_0006_herorun_0007_run_anim_007.png
run_0007_herorun_0008_run_anim_008.png
run_0008_herorun_0009_run_anim_009.png
run_0009_herorun_0010_run_anim_010.png
run_0010_herorun_0011_run_anim_011.png
run_0011_herorun_0012_run_anim_012.png
run_0012_herorun_0013_run_anim_013.png
run_0013_herorun_0014_run_anim_014.png

Ziehen Sie jedes einzelne Bild einer einzelnen Animation aus dem Assets-Ordner in die Szenenansicht. Unity zeigt ein Dialogfeld zum Benennen des neuen Animationsclips an.

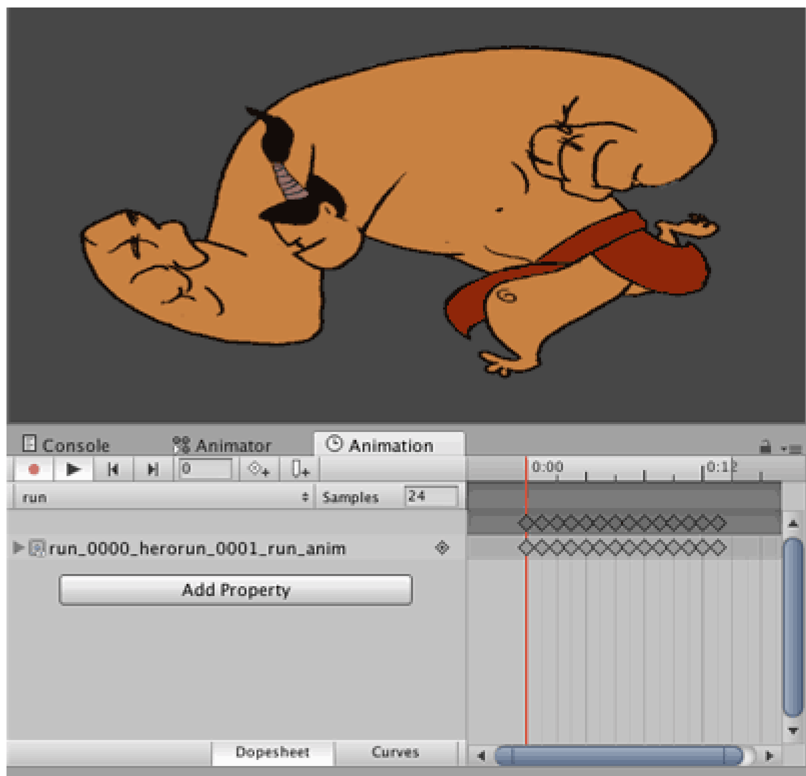


Dies ist eine nützliche Verknüpfung für:

- Neue Spielobjekte erstellen
- Zuweisen von zwei Komponenten (ein Sprite-Renderer und ein Animator)
- Animations-Controller erstellen (und die neue Animator-Komponente mit ihnen verknüpfen)
- Animationsclips mit den ausgewählten Frames erstellen



Vorschau der Wiedergabe in der Animationsregisterkarte durch Klicken auf Wiedergabe:



Dieselbe Methode kann verwendet werden, um neue Animationen für dasselbe Spielobjekt zu erstellen und dann das neue Spielobjekt und den Animationscontroller zu löschen. Fügen Sie den neuen Animationsclip auf dieselbe Weise wie bei der 3D-Animation zum Animationscontroller dieses Objekts hinzu.

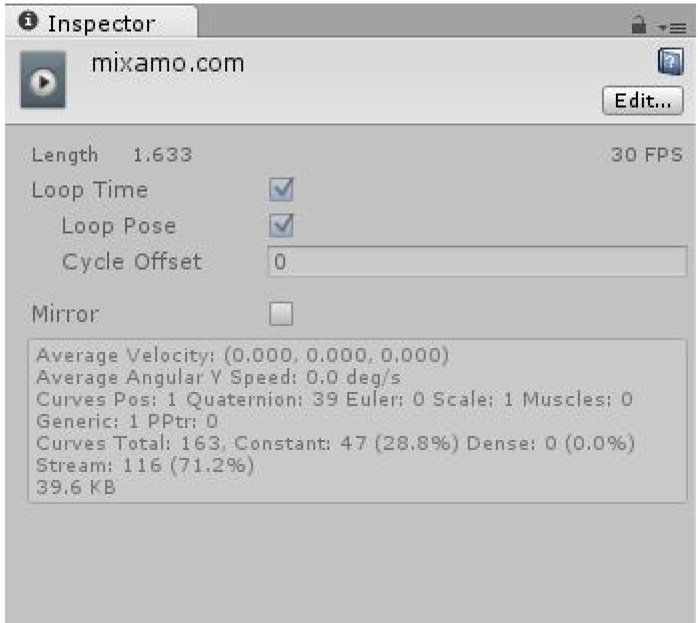
Animationskurven

Mit Animationskurven können Sie einen Float-Parameter während der Wiedergabe der Animation ändern. Wenn beispielsweise eine Animation mit einer Länge von 60 Sekunden vorhanden ist und Sie einen Float-Wert / -Parameter wünschen, nennen Sie ihn X, um durch die Animation zu variieren (wie bei Animationszeit = 0,0s; X = 0,0, bei Animationszeit = 30,0s; X = 1,0, zur Animationszeit = 60,0 s; X = 0,0).

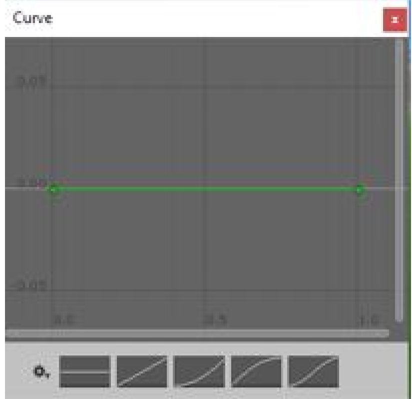
Sobald Sie den Float-Wert haben, können Sie ihn zum Übersetzen, Drehen, Skalieren oder auf andere Weise verwenden.

Für mein Beispiel zeige ich ein laufendes Spielobjekt. Wenn die Animation für einen Lauf abgespielt wird, sollte sich die Übersetzungsgeschwindigkeit des Players im Verlauf der Animation erhöhen. Wenn die Animation ihr Ende erreicht hat, sollte sich die Übersetzungsgeschwindigkeit verringern.

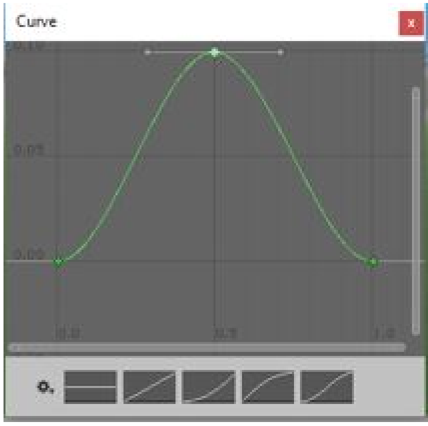
Ich habe einen laufenden Animationsclip erstellt. Wählen Sie den Clip aus und klicken Sie dann im Inspektorfenster auf Bearbeiten.



Wenn Sie dort sind, scrollen Sie nach unten zu Curves. Klicken Sie auf das Pluszeichen, um eine Kurve hinzuzufügen. Benennen Sie die Kurve, zB ForwardRunCurve. Klicken Sie auf die Miniaturkurve rechts. Es öffnet sich ein kleines Fenster mit einer Standardkurve.



Wir wollen eine parabelförmige Kurve, in der sie steigt und dann fällt. Standardmäßig befinden sich 2 Punkte auf der Linie. Sie können weitere Punkte hinzufügen, indem Sie auf die Kurve doppelklicken. Ziehen Sie die Punkte, um eine Form ähnlich der folgenden zu erstellen.



Fügen Sie im Animator-Fenster den laufenden Clip hinzu. Fügen Sie außerdem einen Float-Parameter mit demselben Namen wie die Kurve hinzu, z. B. ForwardRunCurve.

Wenn die Animation abgespielt wird, ändert sich der Float-Wert entsprechend der Kurve. Der folgende Code zeigt, wie der Float-Wert verwendet wird:

```
using UnityEngine;
using System.Collections;

public class RunAnimation : MonoBehaviour {

Animator animator;
float curveValue;

void Start()
{
    animator = GetComponent<Animator>();
}

void Update()
{
    curveValue = animator.GetFloat("ForwardRunCurve");

    transform.Translate (Vector3.forward * curveValue);
}

}
```

Die Variable curveValue enthält den Wert der Kurve (ForwardRunCruve) zu einem bestimmten Zeitpunkt. Wir verwenden diesen Wert, um die Geschwindigkeit der Übersetzung zu ändern. Sie können dieses Skript an das Spielobjekt des Spielers anhängen.

Animation der Einheit online lesen: https://riptutorial.com/de/unity3d/topic/5448/animation-der-einheit

Kapitel 4: Anzeigenintegration

Einführung

In diesem Thema wird die Integration von Werbediensten von Drittanbietern wie Unity Ads oder Google AdMob in ein Unity-Projekt beschrieben.

Bemerkungen

Dies gilt für Unity-Anzeigen .

> **Stellen Sie sicher, dass der Testmodus für Unity Ads während der Entwicklung aktiviert ist**

Sie als Entwickler dürfen keine Impressionen oder Installationen erstellen, indem Sie in Ihrem eigenen Spiel auf Anzeigen klicken. Dies verstößt gegen die Unity Ads-Nutzungsbedingungen , und Sie werden vom Unity Ads-Netzwerk wegen Betrugsversuchen gesperrt.

Weitere Informationen finden Sie in der Unity Ads-Nutzungsbedingungen .

Examples

Grundlagen der Unity-Anzeigen in C

```
using UnityEngine;
using UnityEngine.Advertisements;

public class Example : MonoBehaviour
{
    #if !UNITY_ADS // If the Ads service is not enabled
    public string gameId; // Set this value from the inspector
    public bool enableTestMode = true; // Enable this during development
    #endif

    void InitializeAds () // Example of how to initialize the Unity Ads service
    {
        #if !UNITY_ADS // If the Ads service is not enabled
        if (Advertisement.isSupported) { // If runtime platform is supported
            Advertisement.Initialize(gameId, enableTestMode); // Initialize
        }
        #endif
    }

    void ShowAd () // Example of how to show an ad
    {
        if (Advertisement.isInitialized || Advertisement.IsReady()) { // If the ads are ready
to be shown
            Advertisement.Show(); // Show the default ad placement
        }
```

```
    }
}
```

Unity-Anzeigen-Grundlagen in JavaScript

```
#pragma strict
import UnityEngine.Advertisements;

#if !UNITY_ADS // If the Ads service is not enabled
public var gameId : String; // Set this value from the inspector
public var enableTestMode : boolean = true; // Enable this during development
#endif

function InitializeAds () // Example of how to initialize the Unity Ads service
{
    #if !UNITY_ADS // If the Ads service is not enabled
    if (Advertisement.isSupported) { // If runtime platform is supported
        Advertisement.Initialize(gameId, enableTestMode); // Initialize
    }
    #endif
}

function ShowAd () // Example of how to show an ad
{
    if (Advertisement.isInitialized && Advertisement.IsReady()) { // If the ads are ready to
be shown
        Advertisement.Show(); // Show the default ad placement
    }
}
```

Anzeigenintegration online lesen: https://riptutorial.com/de/unity3d/topic/9796/anzeigenintegration

Kapitel 5: Asset Store

Examples

Zugriff auf den Asset Store

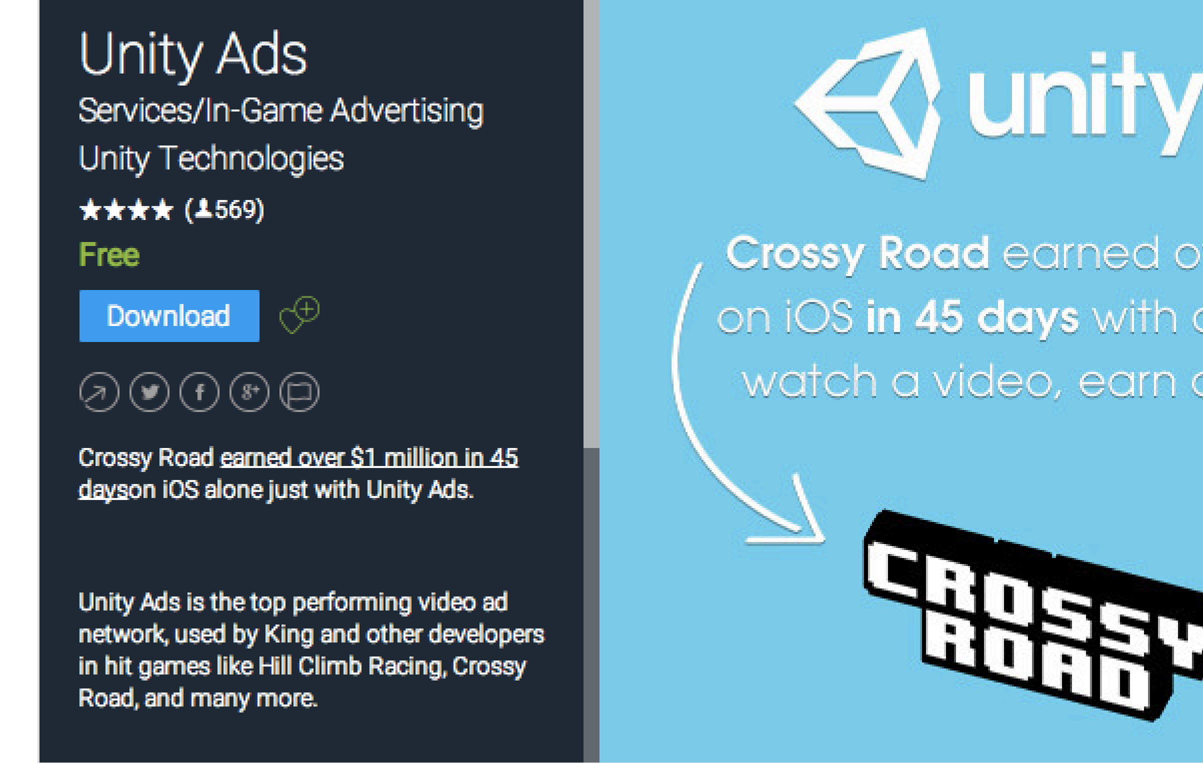
Es gibt drei Möglichkeiten, auf den Unity Asset Store zuzugreifen:

- Öffnen Sie das Asset Store-Fenster, indem Sie im Hauptmenü von Unity Fenster → Asset Store auswählen.
- Verwenden Sie die Tastenkombination (Strg + 9 unter Windows / 9 unter Mac OS).
- Durchsuchen Sie das Webinterface: https://www.assetstore.unity3d.com/

Sie werden möglicherweise aufgefordert, ein kostenloses Benutzerkonto zu erstellen oder sich anzumelden, wenn Sie zum ersten Mal auf den Unity Asset Store zugreifen.

Kauf von Vermögenswerten

Klicken Sie nach dem Zugriff auf den Asset Store und nach dem Anzeigen des gewünschten Assets auf die Schaltfläche **Download** . Der Schaltflächentext kann auch " **Jetzt kaufen" sein,** wenn das Asset mit Kosten verbunden ist.



Wenn Sie den Unity Asset Store über die Weboberfläche anzeigen, wird der Text der Schaltfläche " **Herunterladen"** möglicherweise als " **In Unity öffnen"** angezeigt. Wenn Sie diese Schaltfläche auswählen, wird eine Instanz von Unity gestartet und das Asset wird im *Asset Store-Fenster* angezeigt.

Möglicherweise werden Sie aufgefordert, ein kostenloses Benutzerkonto zu erstellen oder sich anzumelden, wenn Sie zum ersten Mal im Unity Asset Store einkaufen.

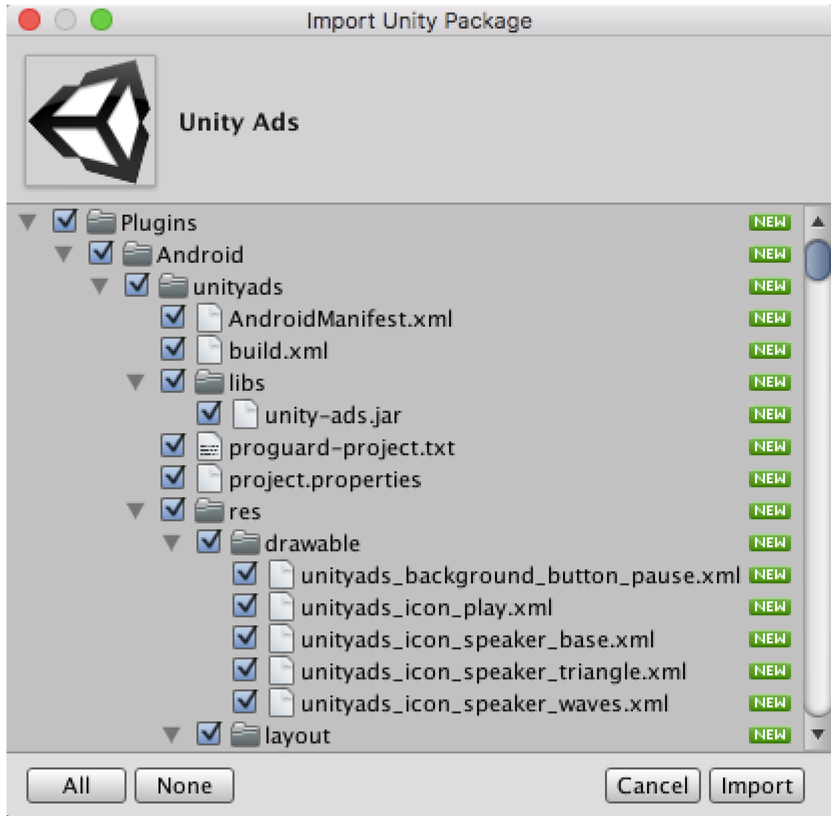
Unity akzeptiert dann ggf. Ihre Zahlung.

Assets importieren

Nachdem das Asset in Unity heruntergeladen wurde, ändert sich die Schaltfläche **Herunterladen** oder **Jetzt kaufen** in **Importieren** .

Wenn Sie diese Option auswählen, wird der Benutzer mit einem Fenster zum *Importieren von Unity-Paketen* aufgefordert, in dem der Benutzer die Asset-Dateien auswählen kann, deren Import er in sein Projekt importieren möchte.

Wählen Sie **Importieren aus** , um den Vorgang zu bestätigen. Platzieren Sie die ausgewählten Asset-Dateien im Ordner Assets, der im *Fenster Projektansicht* angezeigt *wird* .



Veröffentlichen von Assets

1. Erstellen Sie ein Publisher-Konto
2. Fügen Sie dem Publisher-Konto ein Asset hinzu
3. Laden Sie die Asset-Store-Tools herunter (aus dem Asset-Store).
4. Gehen Sie zu "Asset Store Tools"> "Paket-Upload".
5. Wählen Sie im Asset-Tool-Fenster das richtige Paket und den richtigen Projektordner aus
6. Klicken Sie auf Hochladen
7. Reichen Sie Ihr Asset online ein

TODO - Bilder hinzufügen, mehr Details

Bestätigen Sie die Rechnungsnummer eines Kaufs

Die Rechnungsnummer wird verwendet, um den Verkauf für Publisher zu überprüfen. Viele Publisher von bezahlten Assets oder Plugins fragen auf Anfrage nach der Rechnungsnummer. Die Rechnungsnummer wird auch als Lizenzschlüssel verwendet, um ein Asset oder Plugin zu aktivieren.

Die Rechnungsnummer kann an zwei Stellen gefunden werden:

1. Nachdem Sie das Asset gekauft haben, erhalten Sie eine E-Mail mit dem Betreff "Unity Asset Store-Kaufbestätigung ...". Die Rechnungsnummer befindet sich im PDF-Anhang dieser E-Mail.

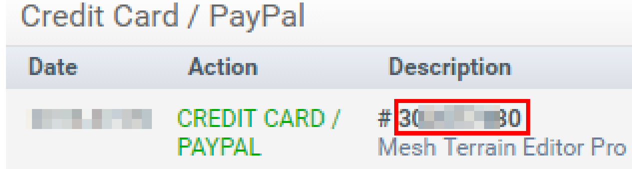
UNITY3D.COM

Unity Technologies ApS
Vendersgade 28
1363 København K
Danmark

INVOICE

Invoice No.
Date
Due Date
Order No.

2. Öffnen Sie https://www.assetstore.unity3d.com/#!/account/transactions , dann finden Sie die Rechnungsnummer in der Spalte *Beschreibung* .



Credit Card / PayPal

Date	Action	Description
	CREDIT CARD / PAYPAL	#30 30 Mesh Terrain Editor Pro

Asset Store online lesen: https://riptutorial.com/de/unity3d/topic/5705/asset-store

Kapitel 6: Attribute

Syntax

- [AddComponentMenu (Zeichenfolge Menüname)]
- [AddComponentMenu (Zeichenfolge menuName, int order)]
- [CanEditMultipleObjects]
- [ContextMenuItem (Stringname, Stringfunktion)]
- [ContextMenu (Name der Zeichenfolge)]
- [CustomEditor (Typ inspectedType)]
- [CustomEditor (Typ inspectedType, bool editorForChildClasses)]
- [CustomPropertyDrawer (Typtyp)]
- [CustomPropertyDrawer (Typtyp, bool useForChildren)]
- [DisallowMultipleComponent]
- [DrawGizmo (GizmoType-Gizmo)]
- [DrawGizmo (GizmoType-Gizmo, Typ drawGizmoType)]
- [ExecuteInEditMode]
- [Header (String-Header)]
- [HideInInspector]
- [InitializeOnLoad]
- [InitializeOnLoadMethod]
- [MenuItem (String itemName)]
- [MenuItem (Zeichenfolge itemName, bool isValidateFunction)]
- [MenuItem (String itemName, bool isValidateFunction, int-Priorität)]
- [Mehrzeilig (int lines)]
- [PreferenceItem (Stringname)]
- [Reichweite (Schwimmer min, Schwimmer max)]
- [RequireComponent (Typ)
- [RuntimeInitializeOnLoadMethod]
- [RuntimeInitializeOnLoadMethod (RuntimeInitializeLoadType loadType)]
- [SerializeField]
- [Space (Schwimmerhöhe)]
- [TextArea (int minLines, int maxLines)]
- [Tooltip (String-Tooltip)]

Bemerkungen

SerializeField

Das Unity-Serialisierungssystem kann für folgende Aufgaben verwendet werden:

- **Kann** öffentliche nicht statische Felder (von serialisierbaren Typen) serialisieren
- **Kann** nicht öffentliche, nicht statische Felder serialisieren, die mit dem Attribut

[SerializeField] gekennzeichnet sind
- Statische Felder **können nicht** serialisiert werden
- Statische Eigenschaften **können nicht** serialisiert werden

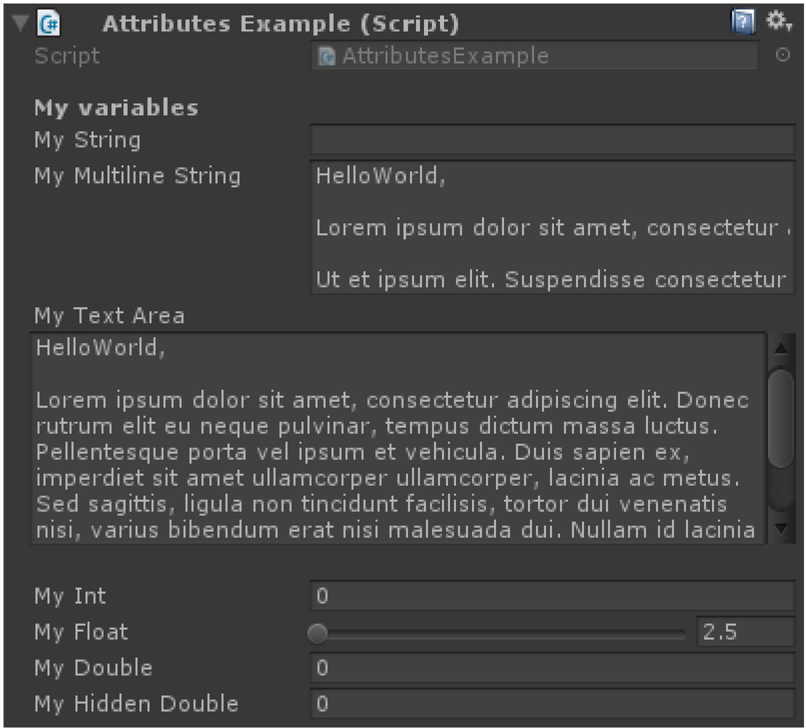
Ihr Feld wird, selbst wenn es mit dem SerializeField-Attribut markiert ist, nur dann zugewiesen, wenn es von einem Typ ist, den Unity serialisieren kann.

- Alle Klassen, die von UnityEngine.Object erben (zB GameObject, Component, MonoBehaviour, Texture2D)
- Alle grundlegenden Datentypen wie int, string, float, bool
- Einige eingebaute Typen wie Vector2 / 3/4, Quaternion, Matrix4x4, Color, Rect, LayerMask
- Arrays eines serialisierbaren Typs
- Liste eines serialisierbaren Typs
- Aufzählungen
- Structs

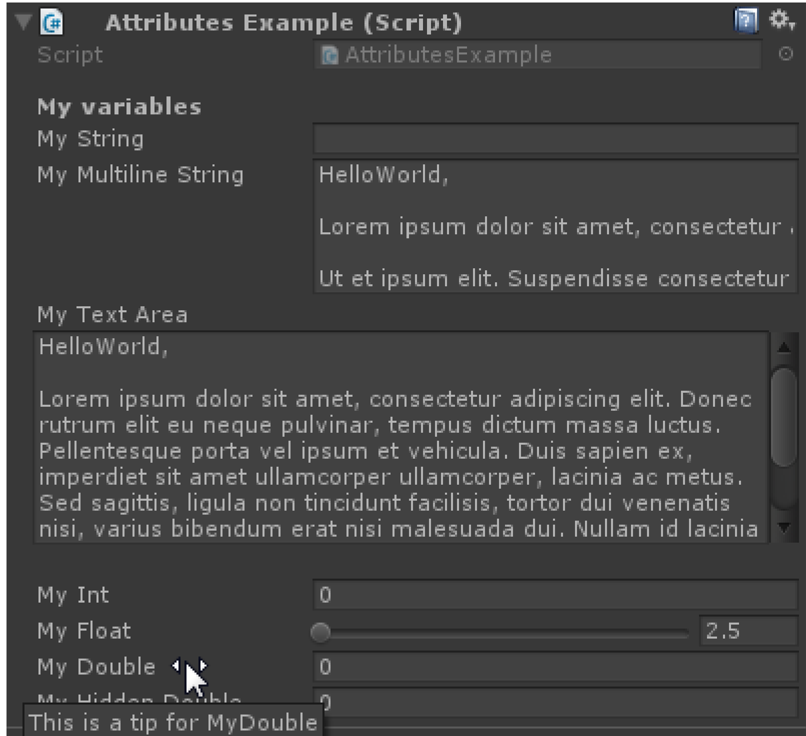
Examples

Allgemeine Inspektorattribute

```
[Header( "My variables" )]
public string MyString;

[HideInInspector]
public string MyHiddenString;

[Multiline( 5 )]
public string MyMultilineString;

[TextArea( 2, 8 )]
public string MyTextArea;

[Space( 15 )]
public int MyInt;

[Range( 2.5f, 12.5f )]
public float MyFloat;

[Tooltip( "This is a tip for MyDouble" )]
public double MyDouble;

[SerializeField]
private double myHiddenDouble;
```



Wenn Sie den Mauszeiger über die Beschriftung eines Feldes bewegen:



```
[Header( "My variables" )]
public string MyString;
```

In der Kopfzeile wird eine fett gedruckte Beschriftung mit dem Text über dem zugewiesenen Feld angezeigt. Dies wird häufig für Kennzeichnungsgruppen verwendet, um sie von anderen Kennzeichnungen abzuheben.

```
[HideInInspector]
```

```
public string MyHiddenString;
```

HideInInspector verhindert, dass öffentliche Felder im Inspektor angezeigt werden. Dies ist nützlich für den Zugriff auf Felder aus anderen Teilen des Codes, wo sie sonst nicht sichtbar oder veränderbar sind.

```
[Multiline( 5 )]
public string MyMultilineString;
```

Multiline erstellt ein Textfeld mit einer bestimmten Anzahl von Zeilen. Bei Überschreitung dieses Betrags wird das Feld weder erweitert noch der Text umgebrochen.

```
[TextArea( 2, 8 )]
public string MyTextArea;
```

TextArea ermöglicht mehrzeiligen Text mit automatischem **Zeilenumbruch** und Bildlaufleisten, wenn der Text den zugewiesenen Bereich überschreitet.

```
[Space( 15 )]
public int MyInt;
```

Der Weltraum zwingt den Inspektor, zusätzlichen Abstand zwischen vorherigen und aktuellen Elementen einzufügen, was beim Unterscheiden und Trennen von Gruppen hilfreich ist.

```
[Range( 2.5f, 12.5f )]
public float MyFloat;
```

Range zwingt einen numerischen Wert zwischen einem Minimum und einem Maximum. Dieses Attribut funktioniert auch für Ganzzahlen und Doubles, obwohl min und max als Gleitkommazahlen angegeben sind.

```
[Tooltip( "This is a tip for MyDouble" )]
public double MyDouble;
```

Der Tooltip zeigt eine zusätzliche Beschreibung, wenn die Beschriftung des Felds **überfahren** wird.

```
[SerializeField]
private double myHiddenDouble;
```

SerializeField zwingt Unity zur Serialisierung des Felds - nützlich für private Felder.

Komponentenattribute

```
[DisallowMultipleComponent]
[RequireComponent( typeof( Rigidbody ) )]
public class AttributesExample : MonoBehaviour
{
```

```
    [...]
}
```

```
[DisallowMultipleComponent]
```

Das DisallowMultipleComponent-Attribut verhindert, dass Benutzer mehrere Instanzen dieser Komponente zu einem GameObject hinzufügen.

```
[RequireComponent( typeof( Rigidbody ) )]
```

Mit dem RequireComponent-Attribut können Sie eine oder mehrere weitere Komponenten als Anforderungen angeben, wenn diese Komponente einem GameObject hinzugefügt wird. Wenn Sie diese Komponente zu einem GameObject hinzufügen, werden die erforderlichen Komponenten automatisch hinzugefügt (sofern noch nicht vorhanden). Diese Komponenten können erst entfernt werden, wenn die Komponente, für die sie erforderlich ist, entfernt wird.

Laufzeitattribute

```
[ExecuteInEditMode]
public class AttributesExample : MonoBehaviour
{

    [RuntimeInitializeOnLoadMethod]
    private static void FooBar()
    {
        [...]
    }

    [RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.BeforeSceneLoad )]
    private static void Foo()
    {
        [...]
    }

    [RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.AfterSceneLoad )]
    private static void Bar()
    {
        [...]
    }

    void Update()
    {
        if ( Application.isEditor )
        {
            [...]
        }
        else
        {
            [...]
        }
    }
}
```

```
[ExecuteInEditMode]
public class AttributesExample : MonoBehaviour
```

Das ExecuteInEditMode-Attribut zwingt Unity, die Zaubermethoden dieses Skripts auszuführen, auch wenn das Spiel nicht läuft.

Die Funktionen werden nicht ständig wie im Wiedergabemodus aufgerufen

- Update wird nur aufgerufen, wenn sich etwas in der Szene geändert hat.
- OnGUI wird aufgerufen, wenn die Game View ein Ereignis empfängt.
- OnRenderObject und die anderen Renderback-Callback-Funktionen werden bei jeder Neulackierung der Szenenansicht oder Spielansicht aufgerufen.

```
[RuntimeInitializeOnLoadMethod]
private static void FooBar()

[RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.BeforeSceneLoad )]
private static void Foo()

[RuntimeInitializeOnLoadMethod( RuntimeInitializeLoadType.AfterSceneLoad )]
private static void Bar()
```

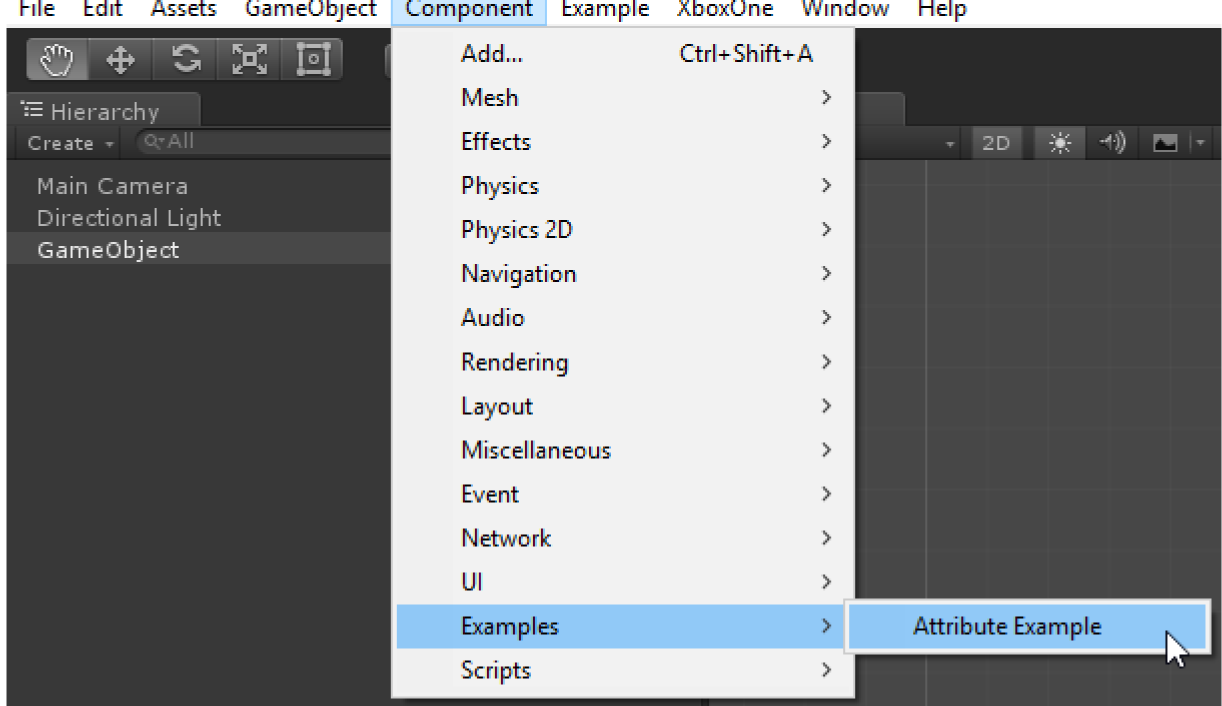
Mit dem Attribut RuntimeInitializeOnLoadMethod kann eine Laufzeitklassenmethode aufgerufen werden, wenn das Spiel die Laufzeit ohne Interaktion des Benutzers lädt.

Sie können angeben, ob die Methode vor oder nach dem Laden der Szene aufgerufen werden soll (after ist Standard). Die Reihenfolge der Ausführung ist für Methoden, die dieses Attribut verwenden, nicht garantiert.

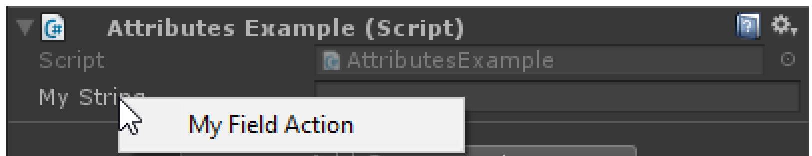
Menüattribute

```
[AddComponentMenu( "Examples/Attribute Example" )]
public class AttributesExample : MonoBehaviour
{
    [ContextMenuItem( "My Field Action", "MyFieldContextAction" )]
    public string MyString;

    private void MyFieldContextAction()
    {
        [...]
    }

    [ContextMenu( "My Action" )]
    private void MyContextMenuAction()
    {
        [...]
    }
}
```

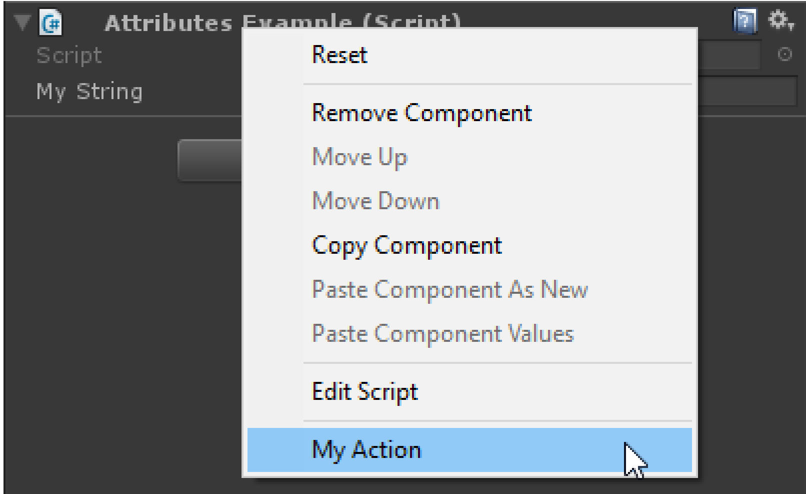
Das Ergebnis des Attributs [AddComponentMenu]



Das Ergebnis des Attributs [ContextMenuItem]



Das Ergebnis des Attributs [ContextMenu]



```
[AddComponentMenu( "Examples/Attribute Example" )]
public class AttributesExample : MonoBehaviour
```

Mit dem AddComponentMenu-Attribut können Sie Ihre Komponente anstelle des Component-> Scripts-Menüs an einer beliebigen Stelle im Component-Menü platzieren.

```
[ContextMenuItem( "My Field Action", "MyFieldContextAction" )]
public string MyString;
```

```
private void MyFieldContextAction()
{
    [...]
}
```

Mit dem Attribut ContextMenuItem können Sie Funktionen definieren, die dem Kontextmenü eines Felds hinzugefügt werden können. Diese Funktionen werden nach Auswahl ausgeführt.

```
[ContextMenu( "My Action" )]
private void MyContextMenuAction()
{
    [...]
}
```

Mit dem ContextMenu-Attribut können Sie Funktionen definieren, die dem Kontextmenü der Komponente hinzugefügt werden können.

Editorattribute

```
[InitializeOnLoad]
public class AttributesExample : MonoBehaviour
{

    static AttributesExample()
    {
        [...]
    }

    [InitializeOnLoadMethod]
    private static void Foo()
    {
        [...]
    }
}
```

```
[InitializeOnLoad]
public class AttributesExample : MonoBehaviour
{

    static AttributesExample()
    {
        [...]
    }
```

Das Attribut InitializeOnLoad ermöglicht dem Benutzer das Initialisieren einer Klasse ohne Interaktion des Benutzers. Dies geschieht immer dann, wenn der Editor gestartet oder neu kompiliert wird. Der statische Konstruktor garantiert, dass dies vor allen anderen statischen Funktionen aufgerufen wird.

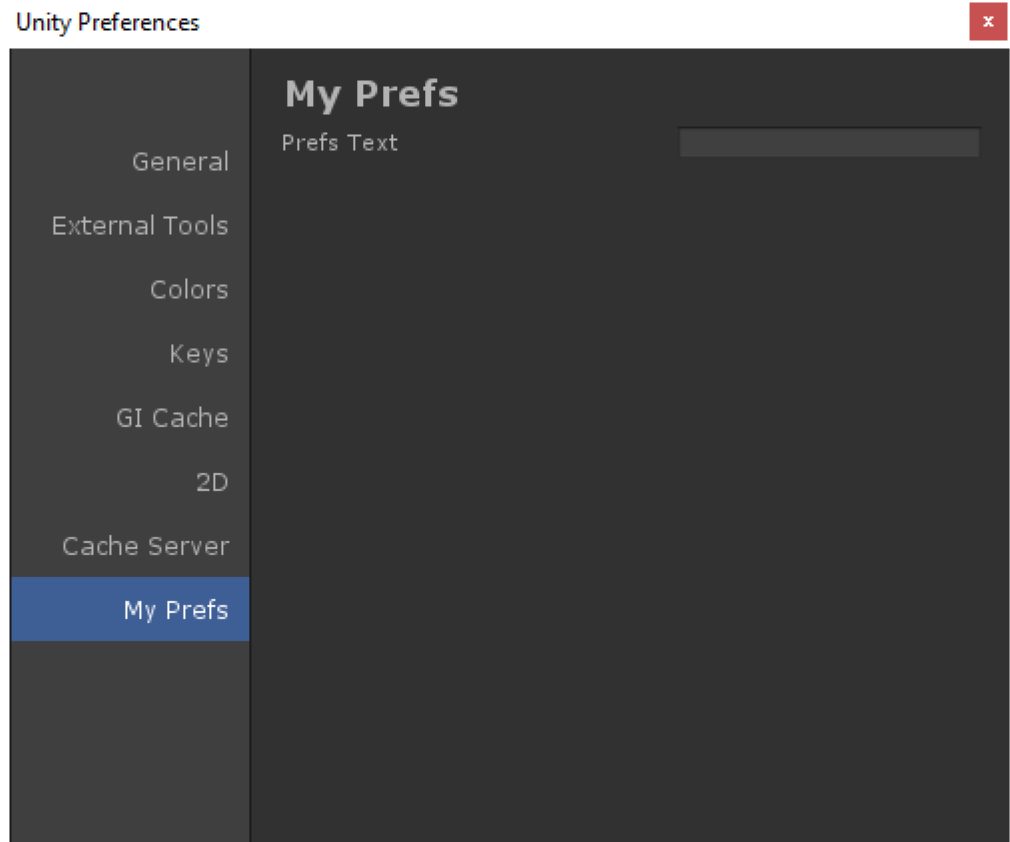
```
[InitializeOnLoadMethod]
private static void Foo()
{
```

```
    [...]
}
```

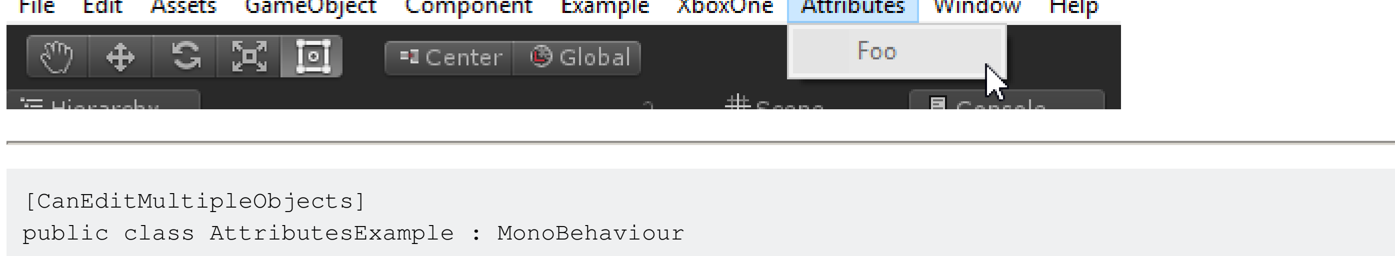
Das Attribut InitializeOnLoad ermöglicht dem Benutzer das Initialisieren einer Klasse ohne Interaktion des Benutzers. Dies geschieht immer dann, wenn der Editor gestartet oder neu kompiliert wird. Die Reihenfolge der Ausführung ist für Methoden, die dieses Attribut verwenden, nicht garantiert.

```
[CanEditMultipleObjects]
public class AttributesExample : MonoBehaviour
{

    public int MyInt;

    private static string prefsText = "";

    [PreferenceItem( "My Prefs" )]
    public static void PreferencesGUI()
    {
        prefsText = EditorGUILayout.TextField( "Prefs Text", prefsText );
    }

    [MenuItem( "Attributes/Foo" )]
    private static void Foo()
    {
        [...]
    }

    [MenuItem( "Attributes/Foo", true )]
    private static bool FooValidate()
    {
        return false;
    }
}
```

Das Ergebnis des [PreferenceItem] -Attributs



Das Ergebnis des Attributs [MenuItem]




```
[CanEditMultipleObjects]
public class AttributesExample : MonoBehaviour
```

Mit dem CanEditMultipleObjects-Attribut können Sie Werte von Ihrer Komponente über mehrere GameObjects bearbeiten. Ohne diese Komponente wird Ihre Komponente bei der Auswahl mehrerer GameObjects nicht normal angezeigt. Stattdessen wird die Meldung "Bearbeitung mehrerer Objekte wird nicht unterstützt" angezeigt.

> Dieses Attribut ist für benutzerdefinierte Editoren vorgesehen, die die Mehrfachbearbeitung unterstützen. Nicht-benutzerdefinierte Editoren unterstützen automatisch die Mehrfachbearbeitung.

```
[PreferenceItem( "My Prefs" )]
public static void PreferencesGUI()
```

Mit dem PreferenceItem-Attribut können Sie im Unity-Voreinstellungsmenü ein zusätzliches Element erstellen. Die empfangende Methode muss statisch sein, damit sie verwendet werden kann.

```
[MenuItem( "Attributes/Foo" )]
```

```
private static void Foo()
{
    [...]
}

[MenuItem( "Attributes/Foo", true )]
private static bool FooValidate()
{
    return false;
}
```

Mit dem MenuItem-Attribut können Sie benutzerdefinierte Menüelemente erstellen, um Funktionen auszuführen. In diesem Beispiel wird auch eine Validatorfunktion verwendet (die immer false zurückgibt), um die Ausführung der Funktion zu verhindern.

```
[CustomEditor( typeof( MyComponent ) )]
public class AttributesExample : Editor
{
    [...]
}
```

Mit dem CustomEditor-Attribut können Sie benutzerdefinierte Editoren für Ihre Komponenten erstellen. Diese Editoren werden zum Zeichnen Ihrer Komponente im Inspektor verwendet und müssen von der Editor-Klasse abgeleitet werden.

```
[CustomPropertyDrawer( typeof( MyClass ) )]
public class AttributesExample : PropertyDrawer
{
    [...]
}
```

Mit dem CustomPropertyDrawer-Attribut können Sie ein benutzerdefiniertes Eigenschaftenfach für den Inspektor erstellen. Sie können diese Schubladen für Ihre benutzerdefinierten Datentypen verwenden, damit sie im Inspektor verwendet werden können.

```
[DrawGizmo( GizmoType.Selected )]
private static void DoGizmo( AttributesExample obj, GizmoType type )
{
    [...]
}
```

Mit dem DrawGizmo-Attribut können Sie benutzerdefinierte Komponenten für Ihre Komponenten zeichnen. Diese Gizmos werden in der Szenenansicht gezeichnet. Sie können entscheiden, wann das Gizmo gezeichnet werden soll, indem Sie den Parameter GizmoType im DrawGizmo-Attribut verwenden.

> Die Empfangsmethode erfordert zwei Parameter. Der erste ist die Komponente, für die das Gizmo gezeichnet werden soll, und die zweite ist der Zustand, in dem sich das Objekt befindet, in dem das Gizmo gezeichnet werden soll.

Attribute online lesen: https://riptutorial.com/de/unity3d/topic/5535/attribute

Kapitel 7: Audiosystem

Einführung

Dies ist eine Dokumentation zum Abspielen von Audio in Unity3D.

Examples

Audioklasse - Audio abspielen

```
using UnityEngine;

public class Audio : MonoBehaviour {
    AudioSource audioSource;
    AudioClip audioClip;

    void Start() {
        audioClip = (AudioClip)Resources.Load("Audio/Soundtrack");
        audioSource.clip = audioClip;
        if (!audioSource.isPlaying) audioSource.Play();
    }
```

Audiosystem online lesen: https://riptutorial.com/de/unity3d/topic/8064/audiosystem

Kapitel 8: Coroutinen

Syntax

- public Coroutine StartCoroutine (Routine IEnumerator);
- public Coroutine StartCoroutine (String methodName, Objektwert = null);
- public void StopCoroutine (Zeichenfolge Methodenname);
- public void StopCoroutine (Routine IEnumerator);
- public void StopAllCoroutines ();

Bemerkungen

Überlegungen zur Leistung

Es ist am besten, Coroutines in Maßen zu verwenden, da die Flexibilität mit Leistungskosten einhergeht.

- Coroutines verlangen in großer Zahl mehr von der CPU als von Standard-Update-Methoden.
- In einigen Unity-Versionen gibt es ein Problem, bei dem Coroutines bei jedem Aktualisierungszyklus Müll erzeugen, da Unity den `MoveNext` Rückgabewert `MoveNext` . Dies wurde zuletzt in 5.4.0b13 beobachtet. (Fehlerbericht)

Reduzieren Sie den Müll durch Zwischenspeichern von YieldInstructions

Ein üblicher Trick, um den in Coroutinen erzeugten Müll zu reduzieren, besteht im `YieldInstruction` der `YieldInstruction` .

```
IEnumerator TickEverySecond()
{
    var wait = new WaitForSeconds(1f); // Cache
    while(true)
    {
        yield return wait; // Reuse
    }
}
```

Das Nachgeben von `null` erzeugt keinen zusätzlichen Müll.

Examples

Coroutinen

Zunächst ist es wichtig zu verstehen, dass Spiel-Engines (wie Unity) nach einem "Frame-

basierten" Paradigma arbeiten.

Code wird bei jedem Frame ausgeführt.

Dazu gehört der eigene Code von Unity und Ihr Code.

Beim Nachdenken über Frames ist es wichtig zu verstehen, dass es ***absolut*** keine Garantie dafür gibt, wann Frames auftreten. Sie passieren ***nicht*** regelmäßig. Die Lücken zwischen Frames könnten zum Beispiel 0,02632, dann 0,021167, dann 0,029778 usw. sein. In dem Beispiel sind sie alle etwa 1/50 einer Sekunde, aber sie sind alle unterschiedlich. Und Sie erhalten zu jeder Zeit einen Frame, der viel länger dauert oder kürzer ist. und Ihr Code kann jederzeit innerhalb des Rahmens ausgeführt werden.

In Anbetracht dessen fragen Sie vielleicht: Wie greifen Sie auf diese Frames in Ihrem Code zu, in Unity?

Sie verwenden ganz einfach entweder den Aufruf Update () oder eine Coroutine. (In der Tat - sie sind genau das Gleiche: Sie ermöglichen, dass Code für jeden Frame ausgeführt wird.)

Der Zweck einer Coroutine ist:

Sie können etwas Code ausführen und dann "stop and wait" ***bis zu einem zukünftigen Frame*** .

Sie können bis **zum nächsten Frame** warten, können Sie für **eine Reihe von Frames** warten, oder Sie können für einige ***ungefähre*** Zeit in Sekunden in der Zukunft warten.

Zum Beispiel können Sie auf "etwa eine Sekunde" warten, was bedeutet, dass es etwa eine Sekunde warten wird, und dann den Code in etwa einer Sekunde in einen Frame stellen. (Und tatsächlich kann der Code innerhalb dieses Rahmens jederzeit ausgeführt werden.) Wiederholen: Es wird nicht genau eine Sekunde dauern. Genaues Timing ist in einer Spiel-Engine ohne Bedeutung.

In einer Coroutine:

Einen Frame warten:

```
// do something
yield return null;  // wait until next frame
// do something
```

Drei Frames warten:

```
// do something
yield return null;  // wait until three frames from now
yield return null;
yield return null;
// do something
```

Um **etwa eine** halbe Sekunde zu warten:

```
// do something
yield return new WaitForSeconds (0.5f); // wait for a frame in about .5 seconds
// do something
```

Mach jeden einzelnen Frame etwas:

```
while (true)
{
    // do something
    yield return null;  // wait until the next frame
}
```

Dieses Beispiel ist buchstäblich identisch mit dem Einfügen von etwas in Unitys "Update" -Aufruf: Der Code bei "etwas tun" wird bei jedem Frame ausgeführt.

Beispiel

Ticker an ein `GameObject` . Während dieses Spielobjekt aktiv ist, wird der Haken ausgeführt. Beachten Sie, dass das Skript die Coroutine vorsichtig stoppt, wenn das Spielobjekt inaktiv wird. Dies ist normalerweise ein wichtiger Aspekt für die korrekte Verwendung der Coroutinen.

```
using UnityEngine;
using System.Collections;

public class Ticker:MonoBehaviour {

    void OnEnable()
    {
        StartCoroutine(TickEverySecond());
    }

    void OnDisable()
    {
        StopAllCoroutines();
    }

    IEnumerator TickEverySecond()
    {
        var wait = new WaitForSeconds(1f); // REMEMBER: IT IS ONLY APPROXIMATE
        while(true)
        {
            Debug.Log("Tick");
            yield return wait;  // wait for a frame, about 1 second from now
        }
    }
}
```

Eine Coroutine beenden

Oft entwerfen Sie Coroutinen so, dass sie auf natürliche Weise enden, wenn bestimmte Ziele erreicht werden.

```
IEnumerator TickFiveSeconds()
{
    var wait = new WaitForSeconds(1f);
    int counter = 1;
    while(counter < 5)
    {
        Debug.Log("Tick");
        counter++;
        yield return wait;
    }
    Debug.Log("I am done ticking");
}
```

Um zu verhindern, dass eine Coroutine "innerhalb" der Coroutine "steht", können Sie nicht einfach "zurückkehren", als würden Sie eine gewöhnliche Funktion früher verlassen. Stattdessen verwenden Sie den `yield break` .

```
IEnumerator ShowExplosions()
{
    ... show basic explosions
    if(player.xp < 100) yield break;
    ... show fancy explosions
}
```

Sie können auch alle vom Skript gestarteten Coroutinen zwingen, vor dem Beenden anzuhalten.

```
void OnDisable()
{
    // Stops all running coroutines
    StopAllCoroutines();
}
```

Die Methode, um eine *bestimmte* Coroutine vom Anrufer zu stoppen, hängt davon ab, wie Sie sie gestartet haben.

Wenn Sie eine Coroutine mit dem Namen der Zeichenfolge gestartet haben:

```
StartCoroutine("YourAnimation");
```

Dann können Sie es stoppen, indem Sie StopCoroutine mit demselben String-Namen aufrufen:

```
StopCoroutine("YourAnimation");
```

Alternativ können Sie einen Verweis halten , um *entweder* die `IEnumerator` durch die Koroutine Verfahren zurückgegeben wird , *oder* das `Coroutine` Objekt durch zurück `StartCoroutine` und rufen `StopCoroutine` auf eine der beiden:

```
public class SomeComponent : MonoBehaviour
{
    Coroutine routine;

    void Start () {
        routine = StartCoroutine(YourAnimation());
```

```
    }

    void Update () {
        // later, in response to some input...
        StopCoroutine(routine);
    }

    IEnumerator YourAnimation () { /* ... */ }
}
```

MonoBehaviour-Methoden, die Coroutines sein können

Es gibt drei MonoBehaviour-Methoden, die zu Coroutinen gemacht werden können.

1. Start()
2. OnBecameVisible ()
3. OnLevelWasLoaded ()

Damit können Sie beispielsweise Skripts erstellen, die nur ausgeführt werden, wenn das Objekt für eine Kamera sichtbar ist.

```
using UnityEngine;
using System.Collections;

public class RotateObject : MonoBehaviour
{
    IEnumerator OnBecameVisible()
    {
        var tr = GetComponent<Transform>();
        while (true)
        {
            tr.Rotate(new Vector3(0, 180f * Time.deltaTime));
            yield return null;
        }
    }

    void OnBecameInvisible()
    {
        StopAllCoroutines();
    }
}
```

Verketten von Coroutinen

Coroutinen können in sich nachgeben und auf **andere Coroutinen** warten.

Sie können also Sequenzen verketten - "eine nach der anderen".

Dies ist sehr einfach und eine grundlegende Technik in Unity.

In Spielen ist es absolut selbstverständlich, dass bestimmte Dinge "in Ordnung" passieren müssen. Fast jede "Runde" eines Spiels beginnt mit einer bestimmten Reihe von Ereignissen über einen bestimmten Zeitraum und in einer bestimmten Reihenfolge. So starten Sie ein Autorennen:

```
IEnumerator BeginRace()
{
  yield return StartCoroutine(PrepareRace());
  yield return StartCoroutine(Countdown());
  yield return StartCoroutine(StartRace());
}
```

Wenn Sie also BeginRace anrufen ...

```
 StartCoroutine(BeginRace());
```

Es wird Ihre Routine "Vorbereitungsrennen" ausführen. (Vielleicht blinken einige Lichter und ein paar Menschengeräusche, Zurücksetzen von Ergebnissen usw.) Wenn dies beendet ist, wird der Countdown ausgeführt, in dem Sie möglicherweise einen Countdown auf der Benutzeroberfläche animieren. Wenn dies abgeschlossen ist, wird der Startcode für das Rennen ausgeführt, bei dem Sie möglicherweise Soundeffekte ausführen, einige KI-Treiber starten, die Kamera auf eine bestimmte Weise bewegen und so weiter.

Verstehen Sie zur Klarheit, dass die drei Aufrufe

```
  yield return StartCoroutine(PrepareRace());
  yield return StartCoroutine(Countdown());
  yield return StartCoroutine(StartRace());
```

muss selbst **in** einer Coroutine sein. Das heißt, sie müssen vom Typ `IEnumerator` . In unserem Beispiel ist das also `IEnumerator BeginRace` . Aus "normalem" Code starten Sie diese Coroutine mit dem Aufruf " `StartCoroutine` .

```
 StartCoroutine(BeginRace());
```

Um die Verkettung besser zu verstehen, haben wir hier eine Funktion, die Coroutinen verkettet. Sie übergeben eine Reihe von Coroutinen. Die Funktion führt so viele Coroutinen aus, wie Sie nacheinander durchlaufen.

```
// run various routines, one after the other
IEnumerator OneAfterTheOther( params IEnumerator[] routines )
{
    foreach ( var item in routines )
    {
        while ( item.MoveNext() ) yield return item.Current;
    }

    yield break;
}
```

So würden Sie das nennen ... Nehmen wir an, Sie haben drei Funktionen. Erinnern Sie sich, dass sie alle `IEnumerator` sein `IEnumerator` :

```
IEnumerator PrepareRace()
{
    // codesay, crowd cheering and camera pan around the stadium
```

```
    yield break;
}

IEnumerator Countdown()
{
    // codesay, animate your countdown on UI
    yield break;
}

IEnumerator StartRace()
{
    // codesay, camera moves and light changes and launch the AIs
    yield break;
}
```

Du würdest es so nennen

```
StartCoroutine( MultipleRoutines( PrepareRace(), Countdown(), StartRace() ) );
```

oder vielleicht so

```
IEnumerator[] routines = new IEnumerator[] {
     PrepareRace(),
     Countdown(),
     StartRace() };
StartCoroutine( MultipleRoutines( routines ) );
```

Um es zu wiederholen, eine der grundlegendsten Anforderungen in Spielen ist, dass bestimmte Dinge "in einer Sequenz" im Laufe der Zeit nacheinander passieren. Das erreichen Sie in Unity sehr einfach mit

```
  yield return StartCoroutine(PrepareRace());
  yield return StartCoroutine(Countdown());
  yield return StartCoroutine(StartRace());
```

Wege zum Nachgeben

Sie können bis zum nächsten Frame warten.

```
yield return null; // wait until sometime in the next frame
```

Sie können mehrere dieser Anrufe in einer Reihe haben, um einfach auf beliebig viele Frames zu warten.

```
//wait for a few frames
yield return null;
yield return null;
```

Warten Sie ***ungefähr*** n Sekunden. Es ist äußerst wichtig zu verstehen, dass dies nur **eine ungefähre Zeit ist** .

```
yield return new WaitForSeconds(n);
```

Es ist absolut nicht möglich, den Aufruf "WaitForSeconds" für genaue Timing-Zeiten zu verwenden.

Oft möchten Sie Aktionen verketten. Also, mach etwas, und wenn das fertig ist, mach etwas anderes, und wenn das fertig ist, mach etwas anderes. Um dies zu erreichen, warten Sie auf eine andere Coroutine:

```
yield return StartCoroutine(coroutine);
```

Verstehen Sie, dass Sie dies nur von einer Coroutine aus aufrufen können. So:

```
StartCoroutine(Test());
```

So beginnen Sie eine Coroutine mit einem "normalen" Code.

Dann in dieser laufenden Coroutine:

```
Debug.Log("A");
StartCoroutine(LongProcess());
Debug.Log("B");
```

Dadurch wird A gedruckt, der lange Prozess gestartet und **sofort B gedruckt** . Es wird **nicht** warten, bis der lange Prozess abgeschlossen ist. Auf der anderen Seite:

```
Debug.Log("A");
yield return StartCoroutine(LongProcess());
Debug.Log("B");
```

Dann wird A gedruckt, der lange Vorgang gestartet, **gewartet, bis** der Vorgang abgeschlossen **ist** , und dann wird B gedruckt.

Es lohnt sich immer daran zu erinnern, dass Coroutinen keinerlei Verbindung zu Threads haben. Mit diesem Code:

```
Debug.Log("A");
StartCoroutine(LongProcess());
Debug.Log("B");
```

Man kann sich leicht vorstellen, dass es so ist, als würde man LongProcess in einem anderen Thread im Hintergrund starten. Das ist aber absolut falsch. Es ist nur eine Coroutine. Game Engines basieren auf Frames und "Coroutines" in Unity ermöglichen Ihnen einfach den Zugriff auf die Frames.

Es ist sehr einfach zu warten, bis eine Webanfrage abgeschlossen ist.

```
void Start() {
    string url = "http://google.com";
```

```
    WWW www = new WWW(url);
    StartCoroutine(WaitForRequest(www));
}

IEnumerator WaitForRequest(WWW www) {
    yield return www;

    if (www.error == null) {
        //use www.data);
    }
    else {
        //use www.error);
    }
}
```

Der Vollständigkeit halber: In sehr seltenen Fällen verwenden Sie ein festes Update in Unity. Es gibt einen `WaitForFixedUpdate()` Aufruf, der normalerweise niemals verwendet würde. Es gibt einen bestimmten Aufruf (`WaitForEndOfFrame()` in der aktuellen Version von Unity), der in bestimmten Situationen beim `WaitForEndOfFrame()` von Bildschirmaufnahmen während der Entwicklung verwendet wird. (Der genaue Mechanismus ändert sich mit der Entwicklung von Unity geringfügig, also google nach den neuesten Informationen, falls relevant.)

Coroutinen online lesen: https://riptutorial.com/de/unity3d/topic/3415/coroutinen

Kapitel 9: CullingGroup-API

Bemerkungen

Da die Verwendung von CullingGroups nicht immer sehr unkompliziert ist, kann es hilfreich sein, den Großteil der Logik hinter einer Manager-Klasse zu kapseln.

Nachfolgend finden Sie einen Plan, wie ein solcher Manager funktionieren kann.

```
using UnityEngine;
using System;
public interface ICullingGroupManager
{
    int ReserveSphere();
    void ReleaseSphere(int sphereIndex);
    void SetPosition(int sphereIndex, Vector3 position);
    void SetRadius(int sphereIndex, float radius);
    void SetCullingEvent(int sphereIndex, Action<CullingGroupEvent> sphere);
}
```

Das Wichtigste dabei ist, dass Sie eine Sperrkugel vom Manager reservieren, die den Index der reservierten Kugel zurückgibt. Sie verwenden dann den angegebenen Index, um Ihre reservierte Kugel zu manipulieren.

Examples

Entfernungen von Objekten abschneiden

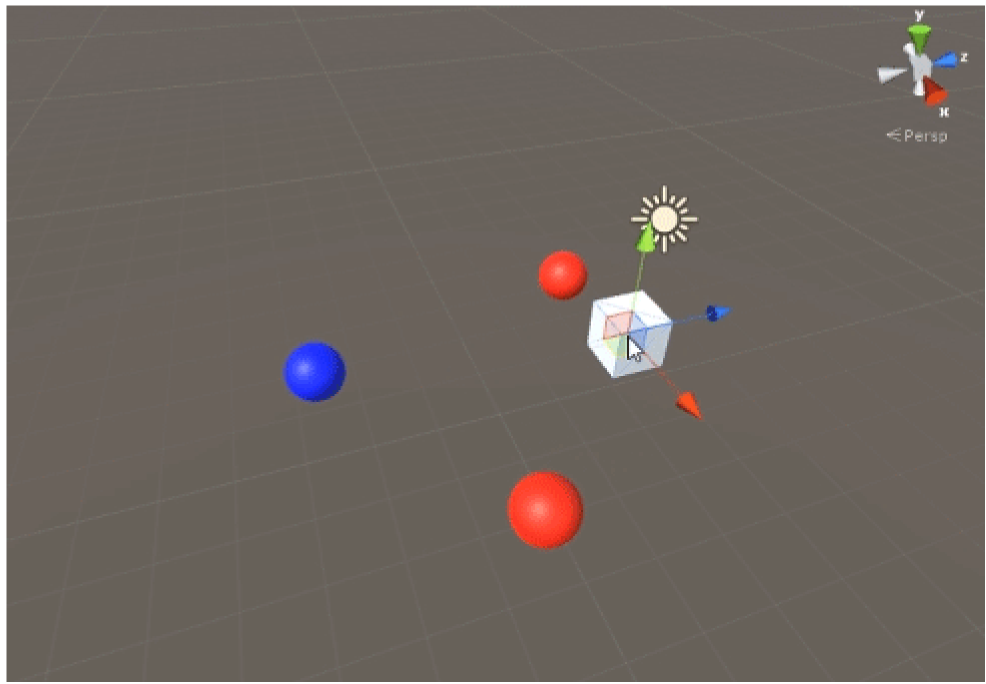
Das folgende Beispiel veranschaulicht die Verwendung von CullingGroups zum Abrufen von Benachrichtigungen anhand des Entfernungsreferenzpunkts.

> Dieses Skript wurde aus Gründen der Kürze vereinfacht und verwendet mehrere Methoden, die die Performance stark beeinträchtigen.

```
using UnityEngine;
using System.Linq;

public class CullingGroupBehaviour : MonoBehaviour
{
    CullingGroup localCullingGroup;

    MeshRenderer[] meshRenderers;
    Transform[] meshTransforms;
    BoundingSphere[] cullingPoints;

    void OnEnable()
    {
        localCullingGroup = new CullingGroup();

        meshRenderers = FindObjectsOfType<MeshRenderer>()
                .Where((MeshRenderer m) => m.gameObject != this.gameObject)
```

```
                .ToArray();

        cullingPoints = new BoundingSphere[meshRenderers.Length];
        meshTransforms = new Transform[meshRenderers.Length];

        for (var i = 0; i < meshRenderers.Length; i++)
        {
            meshTransforms[i] = meshRenderers[i].GetComponent<Transform>();
            cullingPoints[i].position = meshTransforms[i].position;
            cullingPoints[i].radius = 4f;
        }

        localCullingGroup.onStateChanged = CullingEvent;
        localCullingGroup.SetBoundingSpheres(cullingPoints);
        localCullingGroup.SetBoundingDistances(new float[] { 0f, 5f });
        localCullingGroup.SetDistanceReferencePoint(GetComponent<Transform>().position);
        localCullingGroup.targetCamera = Camera.main;
    }

    void FixedUpdate()
    {
        localCullingGroup.SetDistanceReferencePoint(GetComponent<Transform>().position);
        for (var i = 0; i < meshTransforms.Length; i++)
        {
            cullingPoints[i].position = meshTransforms[i].position;
        }
    }

    void CullingEvent(CullingGroupEvent sphere)
    {
        Color newColor = Color.red;
        if (sphere.currentDistance == 1) newColor = Color.blue;
        if (sphere.currentDistance == 2) newColor = Color.white;
        meshRenderers[sphere.index].material.color = newColor;
    }

    void OnDisable()
    {
        localCullingGroup.Dispose();
    }
}
```

Fügen Sie das Skript einem GameObject (in diesem Fall einem Cube) hinzu, und klicken Sie auf Spielen. Jedes andere GameObject in der Szene ändert seine Farbe entsprechend der Entfernung zum Referenzpunkt.



Objektsichtbarkeit aussortieren

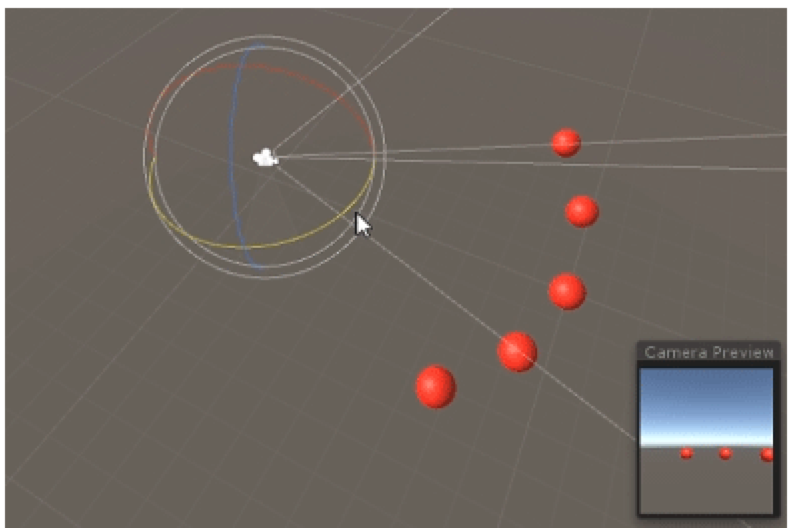
Das folgende Skript veranschaulicht, wie Ereignisse entsprechend der Sichtbarkeit einer festgelegten Kamera empfangen werden.

> Dieses Skript verwendet aus Gründen der Kürze mehrere leistungsstarke Methoden.

```
using UnityEngine;
using System.Linq;

public class CullingGroupCameraBehaviour : MonoBehaviour
{
    CullingGroup localCullingGroup;

    MeshRenderer[] meshRenderers;

    void OnEnable()
    {
        localCullingGroup = new CullingGroup();

        meshRenderers = FindObjectsOfType<MeshRenderer>()
            .Where((MeshRenderer m) => m.gameObject != this.gameObject)
            .ToArray();

        BoundingSphere[] cullingPoints = new BoundingSphere[meshRenderers.Length];
        Transform[] meshTransforms = new Transform[meshRenderers.Length];

        for (var i = 0; i < meshRenderers.Length; i++)
        {
            meshTransforms[i] = meshRenderers[i].GetComponent<Transform>();
            cullingPoints[i].position = meshTransforms[i].position;
            cullingPoints[i].radius = 4f;
        }

        localCullingGroup.onStateChanged = CullingEvent;
        localCullingGroup.SetBoundingSpheres(cullingPoints);
        localCullingGroup.targetCamera = Camera.main;
```

```
    }

    void CullingEvent(CullingGroupEvent sphere)
    {
        meshRenderers[sphere.index].material.color = sphere.isVisible ? Color.red :
Color.white;
    }

    void OnDisable()
    {
        localCullingGroup.Dispose();
    }
}
```

Fügen Sie das Skript der Szene hinzu und klicken Sie auf Wiedergabe. Alle Geometrie in der Szene ändert ihre Farbe entsprechend ihrer Sichtbarkeit.



> Ein ähnlicher Effekt kann mit der `MonoBehaviour.OnBecameVisible()` Methode erzielt werden, wenn das Objekt eine `MeshRenderer` Komponente hat. Verwenden Sie CulingGroups, wenn Sie leere GameObjects-, `Vector3` Koordinaten `Vector3` oder eine zentralisierte Methode zum Verfolgen von Objektsichtbarkeiten verwenden möchten.

Begrenzte Entfernungen

Sie können Begrenzungsabstände über dem Radius der Auslagerungspunkte hinzufügen. Sie sind in gewisser Weise zusätzliche Auslösebedingungen außerhalb des Hauptradius der Sammelpunkte wie "nah", "weit" oder "sehr weit".

```
cullingGroup.SetBoundingDistances(new float[] { 0f, 10f, 100f});
```

> Begrenzungsentfernungen wirken sich nur bei Verwendung eines Entfernungsbezugspunkts aus. Sie haben keine Auswirkungen während des Kamera-Ausschlusses.

Begrenzungsabstände visualisieren

Was anfangs für Verwirrung sorgen kann, ist, wie Begrenzungsabstände über den Kugelradien hinzugefügt werden.

Zunächst berechnet die Culling-Gruppe die *Fläche* sowohl der Begrenzungssphäre als auch der Begrenzungsentfernung. Die beiden Bereiche werden addiert und das Ergebnis ist der Triggerbereich für das Entfernungsband. Der Radius dieses Bereichs kann verwendet werden, um das Wirkungsfeld der Begrenzungsentfernung zu visualisieren.

```
float cullingPointArea = Mathf.PI * (cullingPointRadius * cullingPointRadius);
float boundingArea = Mathf.PI * (boundingDistance * boundingDistance);
float combinedRadius = Mathf.Sqrt((cullingPointArea + boundingArea) / Mathf.PI);
```

CullingGroup-API online lesen: https://riptutorial.com/de/unity3d/topic/4574/cullinggroup-api

Kapitel 10: Designmuster

Examples

Muster für Modellansicht-Controller (MVC)

Der Model-View-Controller ist ein sehr verbreitetes Designmuster, das es schon seit einiger Zeit gibt. Dieses Muster konzentriert sich auf die Reduzierung von *Spaghetti*- Code, indem Klassen in Funktionsteile unterteilt werden. Ich habe kürzlich mit diesem Entwurfsmuster in Unity experimentiert und möchte ein grundlegendes Beispiel darstellen.

Ein MVC-Design besteht aus drei Kernteilen: Modell, Ansicht und Controller.

Modell: Das Modell ist eine Klasse, die den Datenteil Ihres Objekts darstellt. Dies kann ein Spieler, Inventar oder ein ganzes Level sein. Bei korrekter Programmierung sollten Sie dieses Skript auch außerhalb von Unity verwenden können.

Beachten Sie ein paar Dinge zum Modell:

- Es sollte nicht von Monobehaviour erben
- Es sollte keinen Unity-spezifischen Code für die Portabilität enthalten
- Da wir Unity-API-Aufrufe vermeiden, kann dies Dinge wie implizite Konverter in der Model-Klasse verhindern (Problemumgehungen sind erforderlich).

Spieler.cs

```
using System;

public class Player
{
    public delegate void PositionEvent(Vector3 position);
    public event PositionEvent OnPositionChanged;

    public Vector3 position
    {
        get
        {
            return _position;
        }
        set
        {
            if (_position != value) {
                _position = value;
                if (OnPositionChanged != null) {
                    OnPositionChanged(value);
                }
            }
        }
    }
    private Vector3 _position;
}
```

Vector3.cs

Eine benutzerdefinierte Vector3-Klasse für unser Datenmodell.

```
using System;

public class Vector3
{
    public float x;
    public float y;
    public float z;

    public Vector3(float x, float y, float z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

Ansicht: Die Ansicht ist eine Klasse, die den mit dem Modell verknüpften Ansichtsteil darstellt. Dies ist eine geeignete Klasse, um von Monobehaviour abgeleitet zu werden. Dieser sollte Code enthalten, der direkt mit Unity-spezifischen APIs wie `OnCollisinEnter` , `Start` , `Update` usw. interagiert.

- Üblicherweise erbt er von Monobehaviour
- Enthält Unity-spezifischen Code

PlayerView.cs

```
using UnityEngine;

public class PlayerView : Monobehaviour
{
    public void SetPosition(Vector3 position)
    {
        transform.position = position;
    }
}
```

Controller: Der Controller ist eine Klasse, die Modell und Ansicht miteinander verbindet. Controller halten sowohl Model und View als auch die Interaktion mit dem Laufwerk synchron. Der Controller kann Ereignisse von beiden Partnern überwachen und entsprechend aktualisieren.

- Bindet sowohl das Modell als auch die Ansicht, indem der Status synchronisiert wird
- Kann die Interaktion zwischen Partnern fördern
- Controller sind möglicherweise portabel (möglicherweise müssen Sie hier Unity-Code verwenden)
- Wenn Sie sich dafür entscheiden, Ihren Controller nicht portabel zu machen, sollten Sie ihn als Monobehaviour in Betracht ziehen

PlayerController.cs

```
using System;

public class PlayerController
{
    public Player model { get; private set; }
    public PlayerView view { get; private set; }

    public PlayerController(Player model, PlayerView view)
    {
        this.model = model;
        this.view = view;

        this.model.OnPositionChanged += OnPositionChanged;
    }

    private void OnPositionChanged(Vector3 position)
    {
        // Sync
        Vector3 pos = this.model.position;

        // Unity call required here! (we lost portability)
        this.view.SetPosition(new UnityEngine.Vector3(pos.x, pos.y, pos.z));
    }

    // Calling this will fire the OnPositionChanged event
    private void SetPosition(Vector3 position)
    {
        this.model.position = position;
    }
}
```

Endgültige Verwendung

Nun, da wir alle Hauptteile haben, können wir eine Fabrik schaffen, die alle drei Teile erzeugt.

PlayerFactory.cs

```
using System;

public class PlayerFactory
{
    public PlayerController controller { get; private set; }
    public Player model { get; private set; }
    public PlayerView view { get; private set; }

    public void Load()
    {
        // Put the Player prefab inside the 'Resources' folder
        // Make sure it has the 'PlayerView' Component attached
        GameObject prefab = Resources.Load<GameObject>("Player");
        GameObject instance = GameObject.Instantiate<GameObject>(prefab);
        this.model = new Player();
        this.view = instance.GetComponent<PlayerView>();
        this.controller = new PlayerController(model, view);
    }
}
```

Und schließlich können wir die Fabrik von einem Manager anrufen ...

Manager.cs

```
using UnityEngine;

public class Manager : Monobehaviour
{
    [ContextMenu("Load Player")]
    private void LoadPlayer()
    {
        new PlayerFactory().Load();
    }
}
```

Hängen Sie das Manager-Skript an ein leeres GameObject in der Szene an, klicken Sie mit der rechten Maustaste auf die Komponente und wählen Sie "Player laden".

Für eine komplexere Logik können Sie Vererbung mit abstrakten Basisklassen und Schnittstellen für eine verbesserte Architektur einführen.

Designmuster online lesen: https://riptutorial.com/de/unity3d/topic/10842/designmuster

Kapitel 11: Eingabesystem

Examples

Lesetaste drücken und Unterschied zwischen GetKey, GetKeyDown und GetKeyUp

Der Eingang muss von der Aktualisierungsfunktion gelesen werden.

Referenz für alle verfügbaren Keycode-Nummern .

1. `Input.GetKey` drücken mit `Input.GetKey` :

`Input.GetKey` gibt **wiederholt** `true` während der Benutzer den angegebenen Schlüssel gedrückt hält. Dies kann verwendet werden, um eine Waffe **wiederholt** abzufeuern, während Sie die angegebene Taste gedrückt halten. Unten ist ein Beispiel für das automatische Abschießen von Kugeln, wenn die Leertaste gedrückt gehalten wird. Der Spieler muss die Taste nicht immer wieder drücken und loslassen.

```
public GameObject bulletPrefab;
public float shootForce = 50f;

void Update()
{
    if (Input.GetKey(KeyCode.Space))
    {
        Debug.Log("Shooting a bullet while SpaceBar is held down");

        //Instantiate bullet
        GameObject bullet = Instantiate(bulletPrefab, transform.position, transform.rotation)
as GameObject;

        //Get the Rigidbody from the bullet then add a force to the bullet
        bullet.GetComponent<Rigidbody>().AddForce(bullet.transform.forward * shootForce);
    }
}
```

2. Lesetaste drücken mit `Input.GetKeyDown` :

`Input.GetKeyDown` wird nur **einmal** wahr, wenn die angegebene Taste gedrückt wird. Dies ist der Hauptunterschied zwischen `Input.GetKey` und `Input.GetKeyDown` . Ein Beispiel dafür ist das Ein- und Ausschalten einer Benutzeroberfläche oder einer Taschenlampe oder eines Elements.

```
public Light flashLight;
bool enableFlashLight = false;

void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        //Toggle Light
```

```
        enableFlashLight = !enableFlashLight;
        if (enableFlashLight)
        {
            flashLight.enabled = true;
            Debug.Log("Light Enabled!");
        }
        else
        {
            flashLight.enabled = false;
            Debug.Log("Light Disabled!");
        }
    }
}
```

3. Lesetaste drücken mit `Input.GetKeyUp` :

Dies ist das genaue Gegenteil von `Input.GetKeyDown` . Es wird verwendet, um zu erkennen, wann der Tastendruck losgelassen / aufgehoben wird. Genau wie `Input.GetKeyDown` gibt es nur **einmal** `true` . Beispielsweise können Sie `enable` Licht `enable` wenn die Taste mit `Input.GetKeyDown` gedrückt wird, und das Licht deaktivieren, wenn die Taste mit `Input.GetKeyUp` .

```
public Light flashLight;
void Update()
{
    //Disable Light when Space Key is pressed
    if (Input.GetKeyDown(KeyCode.Space))
    {
        flashLight.enabled = true;
        Debug.Log("Light Enabled!");
    }

    //Disable Light when Space Key is released
    if (Input.GetKeyUp(KeyCode.Space))
    {
        flashLight.enabled = false;
        Debug.Log("Light Disabled!");
    }
}
```

Beschleunigungssensor lesen (Basic)

`Input.acceleration` wird zum Lesen des Beschleunigungssensors verwendet. Es gibt `Vector3` als ein Ergebnis , das enthält `x` , `y` und `z` - Achse - Werte im 3D - Raum.

```
void Update()
{
    Vector3 acclerometerValue = rawAccelValue();
    Debug.Log("X: " + acclerometerValue.x + "  Y: " + acclerometerValue.y + "  Z: " +
acclerometerValue.z);
}

Vector3 rawAccelValue()
{
    return Input.acceleration;
}
```

Beschleunigungssensor lesen (Advance)

Die Verwendung von Rohwerten direkt vom Beschleunigungssensor zum Bewegen oder Drehen eines GameObjects kann zu Problemen wie ruckartigen Bewegungen oder Vibrationen führen. Es wird empfohlen, die Werte vor der Verwendung zu glätten. Tatsächlich sollten die Werte des Beschleunigungssensors vor der Verwendung immer geglättet werden. Dies kann mit einem Tiefpassfilter erreicht werden. `Vector3.Lerp` kommt `Vector3.Lerp` zum Einsatz.

```
//The lower this value, the less smooth the value is and faster Accel is updated. 30 seems
fine for this
const float updateSpeed = 30.0f;

float AccelerometerUpdateInterval = 1.0f / updateSpeed;
float LowPassKernelWidthInSeconds = 1.0f;
float LowPassFilterFactor = 0;
Vector3 lowPassValue = Vector3.zero;

void Start()
{
    //Filter Accelerometer
    LowPassFilterFactor = AccelerometerUpdateInterval / LowPassKernelWidthInSeconds;
    lowPassValue = Input.acceleration;
}

void Update()
{

    //Get Raw Accelerometer values (pass in false to get raw Accelerometer values)
    Vector3 rawAccelValue = filterAccelValue(false);
    Debug.Log("RAW X: " + rawAccelValue.x + "  Y: " + rawAccelValue.y + "  Z: " +
rawAccelValue.z);

    //Get smoothed Accelerometer values (pass in true to get Filtered Accelerometer values)
    Vector3 filteredAccelValue = filterAccelValue(true);
    Debug.Log("FILTERED X: " + filteredAccelValue.x + "  Y: " + filteredAccelValue.y + "  Z: "
+ filteredAccelValue.z);
}

//Filter Accelerometer
Vector3 filterAccelValue(bool smooth)
{
    if (smooth)
        lowPassValue = Vector3.Lerp(lowPassValue, Input.acceleration, LowPassFilterFactor);
    else
        lowPassValue = Input.acceleration;

    return lowPassValue;
}
```

Beschleunigungssensor lesen (Präzision)

Lesen Sie den Beschleunigungssensor präzise ab.

In diesem Beispiel wird Speicher zugewiesen:

```
void Update()
```

```
{
    //Get Precise Accelerometer values
    Vector3 accelValue = preciseAccelValue();
    Debug.Log("PRECISE X: " + accelValue.x + "  Y: " + accelValue.y + "  Z: " + accelValue.z);

}

Vector3 preciseAccelValue()
{
    Vector3 accelResult = Vector3.zero;
    foreach (AccelerationEvent tempAccelEvent in Input.accelerationEvents)
    {
        accelResult = accelResult + (tempAccelEvent.acceleration * tempAccelEvent.deltaTime);
    }
    return accelResult;
}
```

In diesem Beispiel wird **kein** Speicher zugewiesen:

```
void Update()
{
    //Get Precise Accelerometer values
    Vector3 accelValue = preciseAccelValue();
    Debug.Log("PRECISE X: " + accelValue.x + "  Y: " + accelValue.y + "  Z: " + accelValue.z);

}

Vector3 preciseAccelValue()
{
    Vector3 accelResult = Vector3.zero;
    for (int i = 0; i < Input.accelerationEventCount; ++i)
    {
        AccelerationEvent tempAccelEvent = Input.GetAccelerationEvent(i);
        accelResult = accelResult + (tempAccelEvent.acceleration * tempAccelEvent.deltaTime);
    }
    return accelResult;
}
```

Beachten Sie, dass dies nicht gefiltert wird. Bitte sehen Sie hier, wie Sie die Beschleunigungsmesser-Werte glätten, um Rauschen zu entfernen.

Lesen Sie die Maustaste (Links, Mitte, Rechts)

Diese Funktionen werden verwendet, um Mausklicks zu überprüfen.

- `Input.GetMouseButton(int button);`
- `Input.GetMouseButtonDown(int button);`
- `Input.GetMouseButtonUp(int button);`

Sie haben alle den gleichen Parameter.

- 0 = Linke Maustaste.
- 1 = Klicken Sie mit der rechten Maustaste.
- 2 = mittlerer Mausklick.

`GetMouseButton` wird verwendet, um zu erkennen, wann die Maustaste gedrückt *gehalten* wird. Es

wird `true` wenn die angegebene Maustaste gedrückt wird.

```
void Update()
{
    if (Input.GetMouseButton(0))
    {
        Debug.Log("Left Mouse Button Down");
    }

    if (Input.GetMouseButton(1))
    {
        Debug.Log("Right Mouse Button Down");
    }

    if (Input.GetMouseButton(2))
    {
        Debug.Log("Middle Mouse Button Down");
    }
}
```

`GetMouseButtonDown` wird verwendet, um festzustellen, wann ein Mausklick erfolgt. Wenn sie **einmal** gedrückt wird, wird `true` . Es wird erst dann wieder `true` zurückgegeben, wenn die Maustaste losgelassen und erneut gedrückt wird.

```
void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        Debug.Log("Left Mouse Button Clicked");
    }

    if (Input.GetMouseButtonDown(1))
    {
        Debug.Log("Right Mouse Button Clicked");
    }

    if (Input.GetMouseButtonDown(2))
    {
        Debug.Log("Middle Mouse Button Clicked");
    }
}
```

`GetMouseButtonUp` wird verwendet, um zu erkennen, wann die angegebene Maustaste losgelassen wird. Dies ist nur `true` wenn die angegebene Maustaste losgelassen wird. Um wieder wahr zu sein, muss es erneut gedrückt und losgelassen werden.

```
void Update()
{
    if (Input.GetMouseButtonUp(0))
    {
        Debug.Log("Left Mouse Button Released");
    }

    if (Input.GetMouseButtonUp(1))
    {
        Debug.Log("Right Mouse Button Released");
    }
```

```
    if (Input.GetMouseButtonUp(2))
    {
        Debug.Log("Middle Mouse Button Released");
    }
}
```

Eingabesystem online lesen: https://riptutorial.com/de/unity3d/topic/3413/eingabesystem

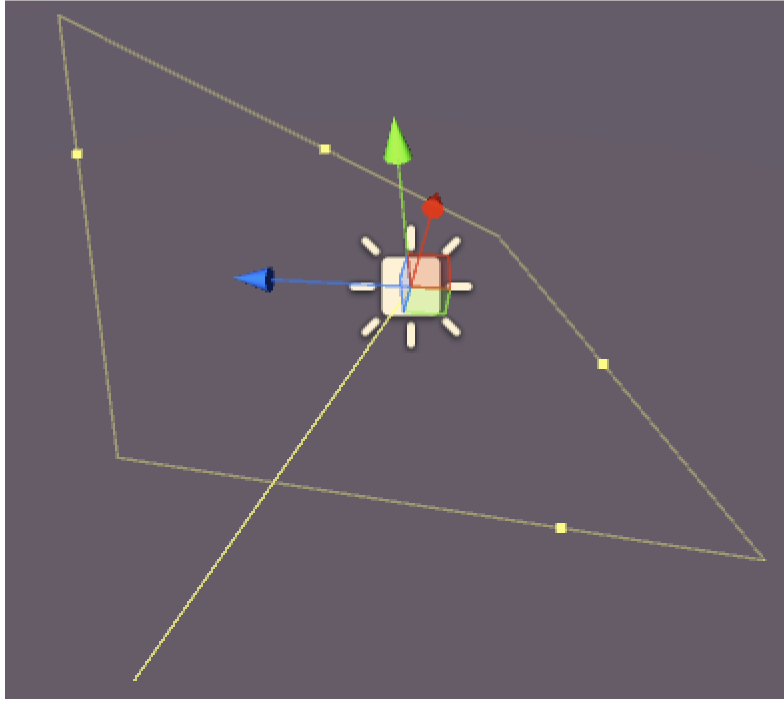
Kapitel 12: Einheitsbeleuchtung

Examples

Arten von Licht

Flächenlicht

Licht wird über die Oberfläche eines rechteckigen Bereichs ausgestrahlt. Sie werden nur gebacken, was bedeutet, dass Sie den Effekt erst sehen können, wenn Sie die Szene backen.

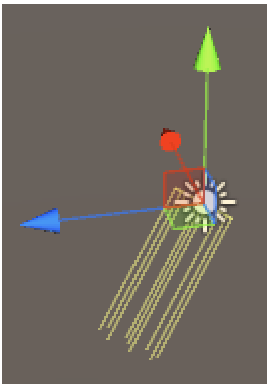


Flächenlichter haben folgende Eigenschaften:

- **Breite** - Breite der Lichtfläche.
- **Höhe** - Höhe der Lichtfläche.
- **Farbe** - Weisen Sie die Farbe des Lichts zu.
- **Intensität** - Wie stark ist das Licht von 0 - 8.
- **Bounce-Intensität** - Wie stark das *indirekte* Licht von 0 - 8 ist.
- **Halo** zeichnen - Zeichnet einen Heiligenschein um das Licht.
- **Flare** - Hiermit können Sie dem Licht einen Flare-Effekt zuweisen.
- **Render-Modus** - Auto, Wichtig, Nicht Wichtig.
- **Culling Mask** - Ermöglicht das selektive Beleuchten von Teilen einer Szene.

Richtungslicht

Richtungslichter strahlen Licht in eine einzige Richtung ab (ähnlich wie die Sonne). Es spielt keine Rolle, wo in der Szene das eigentliche GameObject platziert wird, da das Licht "überall" ist. Die Lichtintensität nimmt nicht wie bei den anderen Lichtarten ab.

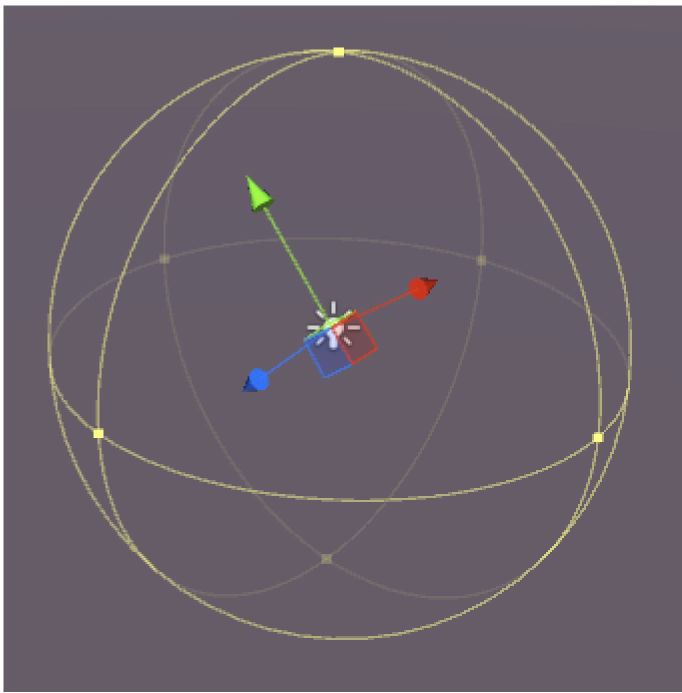


Ein Richtungslicht hat die folgenden Eigenschaften:

- **Backen** - Echtzeit, gebacken oder gemischt.
- **Farbe** - Weisen Sie die Farbe des Lichts zu.
- **Intensität** - Wie stark ist das Licht von 0 - 8.
- **Bounce-Intensität** - Wie stark das *indirekte* Licht von 0 - 8 ist.
- **Schattentyp** - Keine Schatten, harte Schatten oder weiche Schatten.
- **Cookie** - Erlaubt Ihnen, einen Cookie für das Licht zuzuweisen.
- **Cookie Size** - Die Größe des zugewiesenen Cookies.
- **Halo** zeichnen - Zeichnet einen Heiligenschein um das Licht.
- **Flare** - Hiermit können Sie dem Licht einen Flare-Effekt zuweisen.
- **Render-Modus** - Auto, Wichtig, Nicht Wichtig.
- **Culling Mask** - Ermöglicht das selektive Beleuchten von Teilen einer Szene.

Punktlicht

Ein Punktlicht emittiert Licht von einem Punkt im Raum in alle Richtungen. Je weiter vom Ursprungspunkt entfernt, desto weniger intensiv ist das Licht.

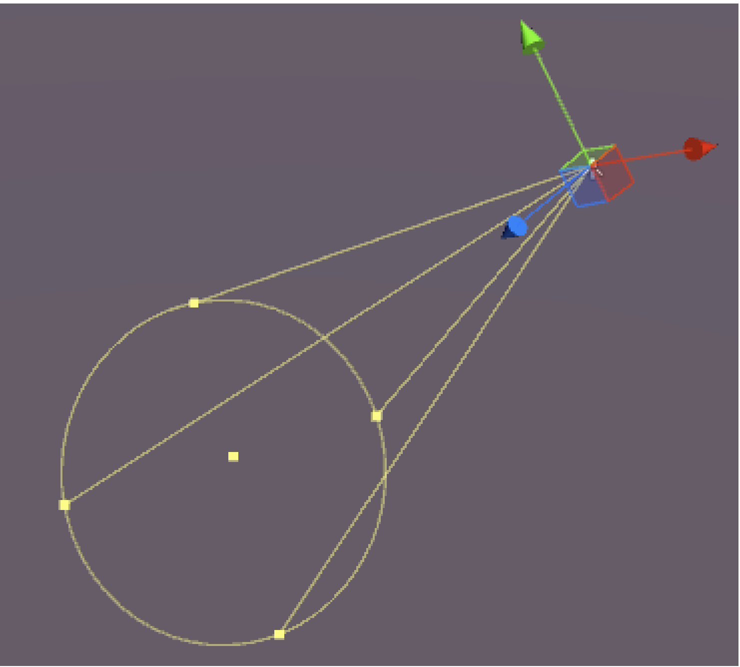


Punktlichter haben folgende Eigenschaften:

- **Backen** - Echtzeit, gebacken oder gemischt.
- **Reichweite** - Die Entfernung von dem Punkt, an dem das Licht nicht mehr erreicht wird.
- **Farbe** - Weisen Sie die Farbe des Lichts zu.
- **Intensität** - Wie stark ist das Licht von 0 - 8.
- **Bounce-Intensität** - Wie stark das *indirekte* Licht von 0 - 8 ist.
- **Schattentyp** - Keine Schatten, harte Schatten oder weiche Schatten.
- **Cookie** - Erlaubt Ihnen, einen Cookie für das Licht zuzuweisen.
- **Halo** zeichnen - Zeichnet einen Heiligenschein um das Licht.
- **Flare** - Hiermit können Sie dem Licht einen Flare-Effekt zuweisen.
- **Render-Modus** - Auto, Wichtig, Nicht Wichtig.
- **Culling Mask** - Ermöglicht das selektive Beleuchten von Teilen einer Szene.

Punktlicht

Ein Punktlicht ist einem Punktlicht sehr ähnlich, aber die Emission ist auf einen Winkel beschränkt. Das Ergebnis ist ein "Lichtkegel", der für Autoscheinwerfer oder Scheinwerfer nützlich ist.



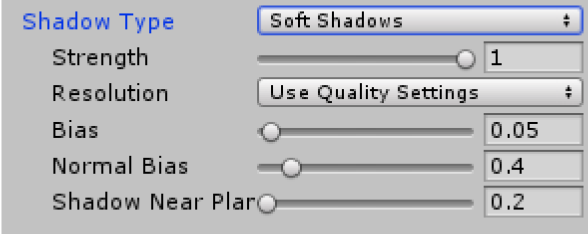
Scheinwerfer haben folgende Eigenschaften:

- **Backen** - Echtzeit, gebacken oder gemischt.
- **Reichweite** - Die Entfernung von dem Punkt, an dem das Licht nicht mehr erreicht wird.
- **Spot Angle** - Der Winkel der Lichtemission.
- **Farbe** - Weisen Sie die Farbe des Lichts zu.
- **Intensität** - Wie stark ist das Licht von 0 - 8.
- **Bounce-Intensität** - Wie stark das *indirekte* Licht von 0 - 8 ist.
- **Schattentyp** - Keine Schatten, harte Schatten oder weiche Schatten.
- **Cookie** - Erlaubt Ihnen, einen Cookie für das Licht zuzuweisen.
- **Halo** zeichnen - Zeichnet einen Heiligenschein um das Licht.
- **Flare** - Hiermit können Sie dem Licht einen Flare-Effekt zuweisen.
- **Render-Modus** - Auto, Wichtig, Nicht Wichtig.
- **Culling Mask** - Ermöglicht das selektive Beleuchten von Teilen einer Szene.

Notiz über Schatten

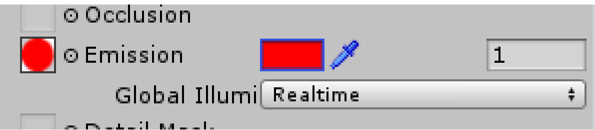
Wenn Sie Hard oder Soft Shadows auswählen, stehen im Inspector folgende Optionen zur Verfügung:

- **Stärke** - Wie dunkel sind die Schatten von 0 - 1.
- **Auflösung** - Wie detailliert Schatten sind.
- **Bias** - der Grad, bis zu dem Schattenwurfflächen vom Licht weggedrückt werden.
- **Normal Bias** - Der Grad, in dem Schattenwurfflächen entlang ihrer Normalen nach innen gedrückt werden.
- **Schatten in der Nähe von Flugzeug** - 0.1 - 10.



Emission

Emission ist, wenn eine Oberfläche (oder eher ein Material) Licht emittiert. Im Inspektorfenster für ein Material zu einem statischen Objekt, das den Standard-Shader verwendet, gibt es eine Emissionseigenschaft:

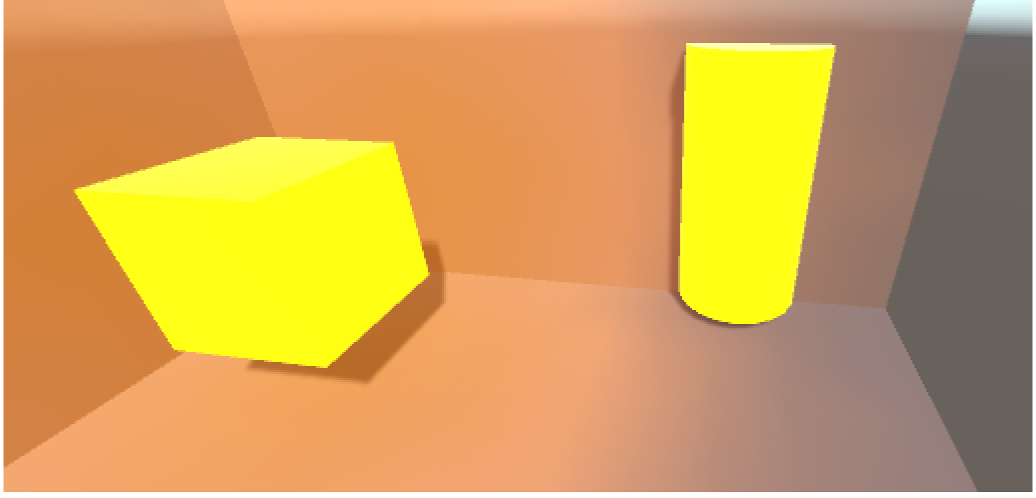


Wenn Sie diese Eigenschaft auf einen höheren Wert als den Standardwert 0 ändern, können Sie die Emissionsfarbe festlegen oder dem Material eine **Emissionskarte** zuweisen. Jede diesem Slot zugewiesene Textur ermöglicht es der Emission, ihre eigenen Farben zu verwenden.

Es gibt auch eine Option für die globale Beleuchtung, mit der Sie einstellen können, ob die Emission in der Nähe statischer Objekte verbacken wird oder nicht:

- **Gebacken** - Die Emission wird in die Szene eingebrannt
- **Echtzeit** - Die Emission wirkt sich auf dynamische Objekte aus
- **Keine** - Die Emission wirkt sich nicht auf Objekte in der Nähe aus

Wenn das Objekt *nicht* auf statisch gesetzt ist, wird der Effekt immer noch "glühen", aber es wird kein Licht ausgestrahlt. Der Würfel hier ist statisch, der Zylinder ist nicht:



Sie können die Emissionsfarbe wie folgt im Code einstellen:

```
Renderer renderer = GetComponent<Renderer>();
Material mat = renderer.material;
mat.SetColor("_EmissionColor", Color.yellow);
```

Das emittierte Licht fällt quadratisch ab und wird nur gegen statische Materialien in der Szene sichtbar.

Einheitsbeleuchtung online lesen: https://riptutorial.com/de/unity3d/topic/7884/einheitsbeleuchtung

Kapitel 13: Erweitern des Editors

Syntax

- [MenuItem (String itemName)]
- [MenuItem (Zeichenfolge itemName, bool isValidateFunction)]
- [MenuItem (String itemName, bool isValidateFunction, int-Priorität)]
- [ContextMenu (Name der Zeichenfolge)]
- [ContextMenuItem (Stringname, Stringfunktion)]
- [DrawGizmo (GizmoType-Gizmo)]
- [DrawGizmo (GizmoType-Gizmo, Typ drawGizmoType)]

Parameter

Parameter	Einzelheiten
MenuCommand	MenuCommand wird verwendet, um den Kontext für ein MenuItem zu extrahieren
MenuCommand.context	Das Objekt, das das Ziel des Menübefehls ist
MenuCommand.userData	Ein int, um benutzerdefinierte Informationen an einen Menüpunkt zu übergeben

Examples

Benutzerdefinierter Inspektor

Wenn Sie einen benutzerdefinierten Inspektor verwenden, können Sie die Art und Weise ändern, in der ein Skript im Inspektor gezeichnet wird. Manchmal möchten Sie zusätzliche Informationen im Inspektor für Ihr Skript hinzufügen, die mit einem benutzerdefinierten Eigenschaftsauszug nicht möglich sind.

Nachfolgend finden Sie ein einfaches Beispiel für ein benutzerdefiniertes Objekt, das bei Verwendung eines benutzerdefinierten Inspektors nützliche Informationen anzeigen kann.

```
using UnityEngine;
#if UNITY_EDITOR
using UnityEditor;
#endif

public class InspectorExample : MonoBehaviour {

    public int Level;
    public float BaseDamage;
```

```
    public float DamageBonus {
        get {
            return Level / 100f * 50;
        }
    }

    public float ActualDamage {
        get {
            return BaseDamage + DamageBonus;
        }
    }
}

#if UNITY_EDITOR
[CustomEditor( typeof( InspectorExample ) )]
public class CustomInspector : Editor {

    public override void OnInspectorGUI() {
        base.OnInspectorGUI();

        var ie = (InspectorExample)target;

        EditorGUILayout.LabelField( "Damage Bonus", ie.DamageBonus.ToString() );
        EditorGUILayout.LabelField( "Actual Damage", ie.ActualDamage.ToString() );
    }
}
#endif
```

Zuerst definieren wir unser benutzerdefiniertes Verhalten mit einigen Feldern

```
public class InspectorExample : MonoBehaviour {
    public int Level;
    public float BaseDamage;
}
```

Die oben gezeigten Felder werden automatisch (ohne benutzerdefinierten Inspektor) gezeichnet, wenn Sie das Skript im Inspektorfenster anzeigen.

```
public float DamageBonus {
    get {
        return Level / 100f * 50;
    }
}

public float ActualDamage {
    get {
        return BaseDamage + DamageBonus;
    }
}
```

Diese Eigenschaften werden von Unity nicht automatisch gezeichnet. Um diese Eigenschaften in der Inspektoransicht anzuzeigen, müssen Sie unseren benutzerdefinierten Inspektor verwenden.

Wir müssen unseren Custom Inspector zunächst so definieren

```
[CustomEditor( typeof( InspectorExample ) )]
```

```
public class CustomInspector : Editor {
```

Der benutzerdefinierte Inspektor muss vom *Editor abgeleitet sein* und benötigt das *CustomEditor*-Attribut. Der Parameter des Attributs ist der Typ des Objekts, für das der benutzerdefinierte Inspektor verwendet werden soll.

Als nächstes ist die OnInspectorGUI-Methode. Diese Methode wird aufgerufen, wenn das Skript im Inspektorfenster angezeigt wird.

```
public override void OnInspectorGUI() {
    base.OnInspectorGUI();
}
```

Wir rufen *base.OnInspectorGUI ()* auf, damit Unity die anderen Felder im Skript verarbeiten kann. Wenn wir das nicht nennen würden, müssten wir selbst mehr arbeiten.

Als nächstes sind unsere benutzerdefinierten Eigenschaften, die wir anzeigen möchten

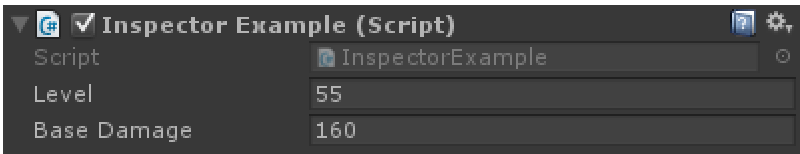
```
var ie = (InspectorExample)target;

EditorGUILayout.LabelField( "Damage Bonus", ie.DamageBonus.ToString() );
EditorGUILayout.LabelField( "Actual Damage", ie.ActualDamage.ToString() );
```

Wir müssen eine temporäre Variable erstellen, die ein auf unseren benutzerdefinierten Typ gegossenes Ziel enthält (Ziel ist verfügbar, da wir vom Editor abgeleitet sind).

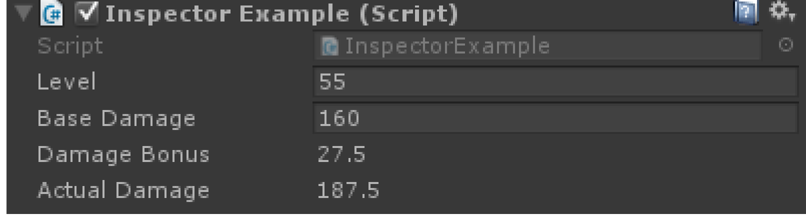
Als Nächstes können wir entscheiden, wie unsere Eigenschaften gezeichnet werden sollen. In diesem Fall sind zwei Labelfelder ausreichend, da wir nur die Werte anzeigen und nicht bearbeiten können.

Ergebnis

Vor



Nach dem



Benutzerdefinierte Eigenschaftsschublade

Manchmal haben Sie benutzerdefinierte Objekte, die Daten enthalten, aber nicht von

MonoBehaviour abgeleitet sind. Das Hinzufügen dieser Objekte als Feld in einer Klasse, die MonoBehaviour ist, hat keine visuellen Auswirkungen, es sei denn, Sie schreiben ein eigenes benutzerdefiniertes Eigenschaftsfeld für den Objekttyp.

Im Folgenden finden Sie ein einfaches Beispiel für ein benutzerdefiniertes Objekt, das zu MonoBehaviour hinzugefügt wurde, sowie ein benutzerdefiniertes Eigenschaftsfach für das benutzerdefinierte Objekt.

```
public enum Gender {
    Male,
    Female,
    Other
}

// Needs the Serializable attribute otherwise the CustomPropertyDrawer wont be used
[Serializable]
public class UserInfo {
    public string Name;
    public int Age;
    public Gender Gender;
}

// The class that you can attach to a GameObject
public class PropertyDrawerExample : MonoBehaviour {
    public UserInfo UInfo;
}

[CustomPropertyDrawer( typeof( UserInfo ) )]
public class UserInfoDrawer : PropertyDrawer {

    public override float GetPropertyHeight( SerializedProperty property, GUIContent label ) {
        // The 6 comes from extra spacing between the fields (2px each)
        return EditorGUIUtility.singleLineHeight * 4 + 6;
    }

    public override void OnGUI( Rect position, SerializedProperty property, GUIContent label )
{
        EditorGUI.BeginProperty( position, label, property );

        EditorGUI.LabelField( position, label );

        var nameRect = new Rect( position.x, position.y + 18, position.width, 16 );
        var ageRect = new Rect( position.x, position.y + 36, position.width, 16 );
        var genderRect = new Rect( position.x, position.y + 54, position.width, 16 );

        EditorGUI.indentLevel++;

        EditorGUI.PropertyField( nameRect, property.FindPropertyRelative( "Name" ) );
        EditorGUI.PropertyField( ageRect, property.FindPropertyRelative( "Age" ) );
        EditorGUI.PropertyField( genderRect, property.FindPropertyRelative( "Gender" ) );

        EditorGUI.indentLevel--;

        EditorGUI.EndProperty();
    }
}
```

Zunächst definieren wir das benutzerdefinierte Objekt mit allen Anforderungen. Nur eine einfache

Klasse, die einen Benutzer beschreibt. Diese Klasse wird in unserer PropertyDrawerExample-Klasse verwendet, die wir einem GameObject hinzufügen können.

```
public enum Gender {
    Male,
    Female,
    Other
}

[Serializable]
public class UserInfo {
    public string Name;
    public int Age;
    public Gender Gender;
}

public class PropertyDrawerExample : MonoBehaviour {
    public UserInfo UInfo;
}
```

> Die benutzerdefinierte Klasse benötigt das Serializable-Attribut, andernfalls wird CustomPropertyDrawer nicht verwendet

Als nächstes ist der CustomPropertyDrawer

Zuerst müssen wir eine von PropertyDrawer abgeleitete Klasse definieren. Die Klassendefinition benötigt außerdem das CustomPropertyDrawer-Attribut. Der übergebene Parameter ist der Typ des Objekts, für das diese Schublade verwendet werden soll.

```
[CustomPropertyDrawer( typeof( UserInfo ) )]
public class UserInfoDrawer : PropertyDrawer {
```

Als Nächstes überschreiben wir die GetPropertyHeight-Funktion. Dies ermöglicht uns, eine benutzerdefinierte Höhe für unser Grundstück festzulegen. In diesem Fall wissen wir, dass unser Eigentum aus vier Teilen besteht: Bezeichnung, Name, Alter und Geschlecht. Daher verwenden wir *EditorGUIUtility.singleLineHeight * 4* und fügen weitere 6 Pixel hinzu, da wir jedes Feld mit zwei Pixeln dazwischen platzieren möchten.

```
public override float GetPropertyHeight( SerializedProperty property, GUIContent label ) {
    return EditorGUIUtility.singleLineHeight * 4 + 6;
}
```

Als nächstes ist die tatsächliche OnGUI-Methode. Wir beginnen mit *EditorGUI.BeginProperty ([...])* und beenden die Funktion mit *EditorGUI.EndProperty ()* . Wir tun dies, damit, wenn diese Eigenschaft Teil eines Prefab sein würde, die eigentliche Prefab-Überschreibungslogik für alles zwischen diesen beiden Methoden funktionieren würde.

```
public override void OnGUI( Rect position, SerializedProperty property, GUIContent label ) {
    EditorGUI.BeginProperty( position, label, property );
```

Danach zeigen wir ein Label mit dem Namen des Feldes und definieren bereits die Rechtecke für

unsere Felder.

```
EditorGUI.LabelField( position, label );

var nameRect = new Rect( position.x, position.y + 18, position.width, 16 );
var ageRect = new Rect( position.x, position.y + 36, position.width, 16 );
var genderRect = new Rect( position.x, position.y + 54, position.width, 16 );
```

> Jedes Feld hat einen Abstand von 16 + 2 Pixeln und die Höhe beträgt 16 (was mit EditorGUIUtility.singleLineHeight identisch ist).

Als Nächstes rücken wir die Benutzeroberfläche mit einer Registerkarte für ein etwas schöneres Layout ein, zeigen die Eigenschaften an, lassen die *Einrückung* der GUI frei und enden mit *EditorGUI.EndProperty* .

```
EditorGUI.indentLevel++;

EditorGUI.PropertyField( nameRect, property.FindPropertyRelative( "Name" ) );
EditorGUI.PropertyField( ageRect, property.FindPropertyRelative( "Age" ) );
EditorGUI.PropertyField( genderRect, property.FindPropertyRelative( "Gender" ) );

EditorGUI.indentLevel--;

EditorGUI.EndProperty();
```

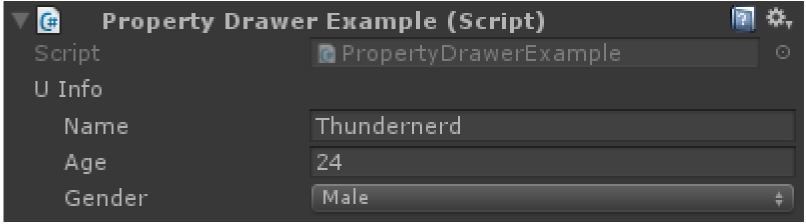
Wir zeigen die Felder mithilfe von *EditorGUI.PropertyField an,* das ein Rechteck für die Position und eine SerializedProperty für die Anzeige der *Eigenschaft* erfordert. Wir erwerben die Eigenschaft, indem *wir FindPropertyRelative ("...")* für die in der *OnGUI-* Funktion übergebene Eigenschaft *aufrufen* . Beachten Sie, dass diese Groß- und Kleinschreibung beachtet werden und nicht öffentliche Eigenschaften nicht gefunden werden können!

> Für dieses Beispiel speichere ich nicht die Eigenschaften, die von property.FindPropertyRelative ("...") zurückgegeben werden. Sie sollten diese in privaten Feldern in der Klasse speichern, um unnötige Anrufe zu vermeiden

Ergebnis

Vor

Nach dem



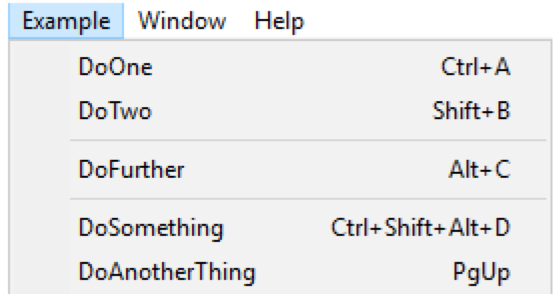
Menüpunkte

Menüelemente bieten eine hervorragende Möglichkeit, dem Editor benutzerdefinierte Aktionen hinzuzufügen. Sie können der Menüleiste Menüelemente hinzufügen, sie als Kontext-Klick auf bestimmte Komponenten oder sogar als Kontext-Klick auf Felder in Ihren Skripts verwenden.

Nachfolgend finden Sie ein Beispiel, wie Sie Menüelemente anwenden können.

```
public class MenuItemsExample : MonoBehaviour {

    [MenuItem( "Example/DoSomething %#&d" )]
    private static void DoSomething() {
        // Execute some code
    }

    [MenuItem( "Example/DoAnotherThing", true )]
    private static bool DoAnotherThingValidator() {
        return Selection.gameObjects.Length > 0;
    }

    [MenuItem( "Example/DoAnotherThing _PGUP", false )]
    private static void DoAnotherThing() {
        // Execute some code
    }

    [MenuItem( "Example/DoOne %a", false, 1 )]
    private static void DoOne() {
        // Execute some code
    }

    [MenuItem( "Example/DoTwo #b", false, 2 )]
    private static void DoTwo() {
        // Execute some code
    }

    [MenuItem( "Example/DoFurther &c", false, 13 )]
    private static void DoFurther() {
        // Execute some code
    }

    [MenuItem( "CONTEXT/Camera/DoCameraThing" )]
    private static void DoCameraThing( MenuCommand cmd ) {
        // Execute some code
    }

    [ContextMenu( "ContextSomething" )]
    private void ContentSomething() {
        // Execute some code
    }

    [ContextMenuItem( "Reset", "ResetDate" )]
    [ContextMenuItem( "Set to Now", "SetDateToNow" )]
    public string Date = "";

    public void ResetDate() {
        Date = "";
    }

    public void SetDateToNow() {
        Date = DateTime.Now.ToString();
    }
}
```

Was sieht so aus



Gehen wir den grundlegenden Menüpunkt durch. Wie Sie unten sehen können, müssen Sie eine statische Funktion mit einem *MenuItem-* Attribut definieren, dem Sie eine Zeichenfolge als Titel für den Menüeintrag übergeben. Sie können Ihr Menüelement auf mehrere Ebenen vertiefen, indem Sie dem Namen ein / hinzufügen.

```
[MenuItem( "Example/DoSomething %#&d" )]
private static void DoSomething() {
    // Execute some code
}
```

> Sie können keinen Menüpunkt auf oberster Ebene haben. Ihre Menüpunkte müssen sich in einem Untermenü befinden!

Die Sonderzeichen am Ende des Namens des MenuItem sind für Tastenkombinationen, diese sind nicht erforderlich.

Es gibt Sonderzeichen, die Sie für Ihre Tastenkombinationen verwenden können. Diese sind:

- % - Strg unter Windows, Cmd unter OS X
- # - Verschiebung
- & - Alt

Das bedeutet, dass die Abkürzung *% # & d unter* Windows für Strg + Umschalt + Alt + D und unter OS X cmd + Umschalt + Alt + D steht.

Wenn Sie eine Tastenkombination ohne Sondertasten verwenden möchten, also beispielsweise nur die Taste 'D', können Sie das Zeichen _ (Unterstrich) der Tastenkombination voranstellen, die Sie verwenden möchten.

Es gibt einige andere spezielle Tasten, die unterstützt werden:

- LINKS, RECHTS, AUF, AB - für die Pfeiltasten
- F1..F12 - für die Funktionstasten
- HOME, END, PGUP, PGDN - für die Navigationstasten

> Tastenkombinationen müssen mit einem Leerzeichen von jedem anderen Text getrennt werden

Weiter sind die Menüpunkte des Validators. Überprüfungsmenüelemente ermöglichen das Deaktivieren von Menüelementen (ausgegraut, nicht anklickbar), wenn die Bedingung nicht erfüllt

ist. Ein Beispiel dafür könnte sein, dass Ihr Menüpunkt auf die aktuelle Auswahl von GameObjects einwirkt, die Sie im Validator-Menüpunkt überprüfen können.

```
[MenuItem( "Example/DoAnotherThing", true )]
private static bool DoAnotherThingValidator() {
    return Selection.gameObjects.Length > 0;
}

[MenuItem( "Example/DoAnotherThing _PGUP", false )]
private static void DoAnotherThing() {
    // Execute some code
}
```

Damit ein Prüfer-Menüelement funktioniert, müssen Sie zwei statische Funktionen erstellen, die beide das Attribut MenuItem und denselben Namen haben (die Tastenkombination ist nicht wichtig). Der Unterschied zwischen ihnen besteht darin, dass Sie sie als Prüferfunktion markieren oder nicht, indem Sie einen booleschen Parameter übergeben.

Sie können auch die Reihenfolge der Menüelemente festlegen, indem Sie eine Priorität hinzufügen. Die Priorität wird durch eine Ganzzahl definiert, die Sie als dritten Parameter übergeben. Je kleiner die Zahl, desto höher in der Liste, desto größer ist die Zahl in der Liste. Sie können ein Trennzeichen zwischen zwei Menüelementen hinzufügen, indem Sie sicherstellen, dass zwischen den Prioritäten der Menüelemente mindestens 10 Stellen stehen.

```
[MenuItem( "Example/DoOne %a", false, 1 )]
private static void DoOne() {
    // Execute some code
}

[MenuItem( "Example/DoTwo #b", false, 2 )]
private static void DoTwo() {
    // Execute some code
}

[MenuItem( "Example/DoFurther &c", false, 13 )]
private static void DoFurther() {
    // Execute some code
}
```

> Wenn Sie eine Menüliste mit einer Kombination aus priorisierten und nicht priorisierten Elementen haben, werden die nicht priorisierten Elemente von den priorisierten Elementen getrennt.

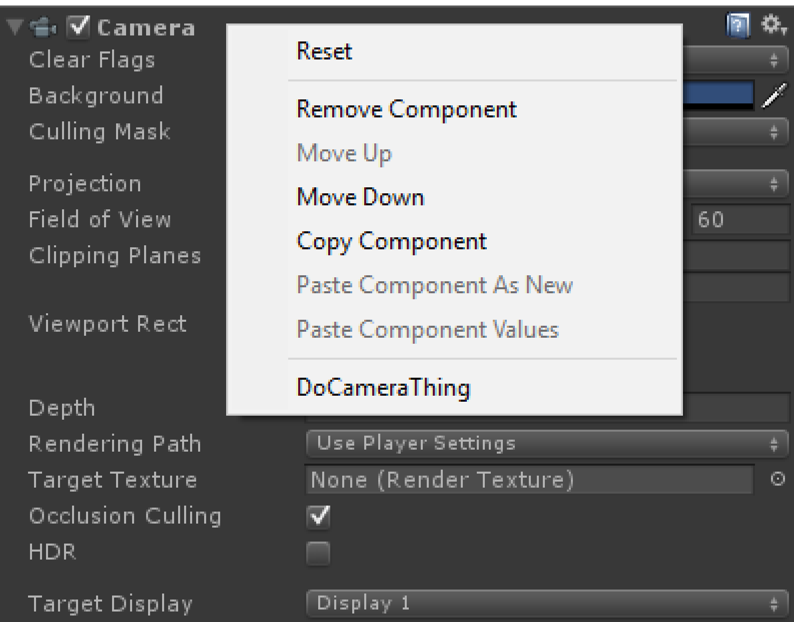
Als Nächstes fügen Sie dem Kontextmenü einer bereits vorhandenen Komponente ein Menüelement hinzu. Sie müssen den Namen des MenuItem mit CONTEXT beginnen (Groß- und Kleinschreibung beachten) und Ihre Funktion muss einen MenuCommand-Parameter übernehmen.

Das folgende Snippet fügt der Camera-Komponente ein Kontextmenüelement hinzu.

```
[MenuItem( "CONTEXT/Camera/DoCameraThing" )]
private static void DoCameraThing( MenuCommand cmd ) {
    // Execute some code
```

```
}
```

Was sieht so aus

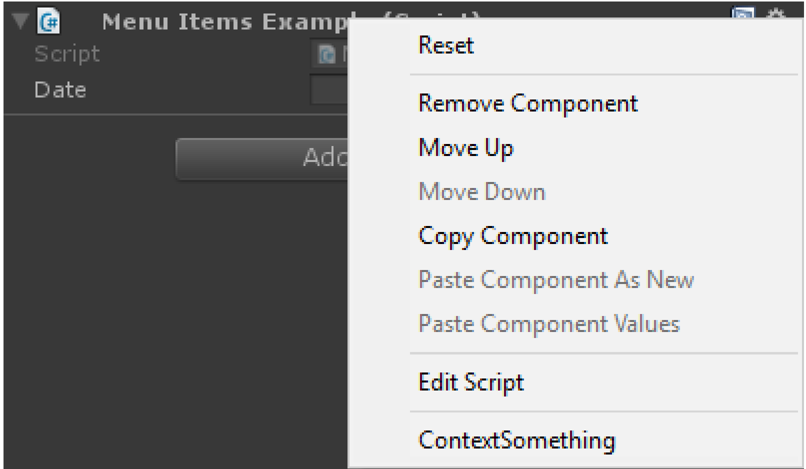


Mit dem Parameter MenuCommand haben Sie Zugriff auf den Komponentenwert und auf alle Nutzerdaten, die mit ihm gesendet werden.

Sie können Ihren eigenen Komponenten auch ein Kontextmenüelement hinzufügen, indem Sie das ContextMenu-Attribut verwenden. Dieses Attribut hat nur einen Namen, keine Validierung oder Priorität und muss Teil einer nicht statischen Methode sein.

```
[ContextMenu( "ContextSomething" )]
private void ContentSomething() {
    // Execute some code
}
```

Was sieht so aus

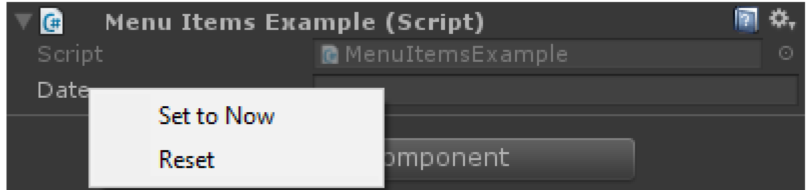


Sie können den Feldern in Ihrer eigenen Komponente auch Kontextmenüelemente hinzufügen. Diese Menüelemente werden angezeigt, wenn Sie mit der rechten Maustaste auf das Feld klicken, zu dem sie gehören, und sie können Methoden ausführen, die Sie in dieser Komponente definiert

haben. Auf diese Weise können Sie beispielsweise Standardwerte oder das aktuelle Datum hinzufügen (siehe unten).

```
[ContextMenuItem( "Reset", "ResetDate" )]
[ContextMenuItem( "Set to Now", "SetDateToNow" )]
public string Date = "";

public void ResetDate() {
    Date = "";
}

public void SetDateToNow() {
    Date = DateTime.Now.ToString();
}
```

Was sieht so aus



Gizmos

Gizmos werden zum Zeichnen von Formen in der Szenenansicht verwendet. Sie können diese Formen verwenden, um zusätzliche Informationen über Ihre GameObjects zu zeichnen, z. B. über das Frustum oder den Erkennungsbereich.

Im Folgenden finden Sie zwei Beispiele dafür

Beispiel eins

In diesem Beispiel werden die Methoden *OnDrawGizmos* und *OnDrawGizmosSelected* (Magie) verwendet.

```
public class GizmoExample : MonoBehaviour {

    public float GetDetectionRadius() {
        return 12.5f;
    }

    public float GetFOV() {
        return 25f;
    }

    public float GetMaxRange() {
        return 6.5f;
    }

    public float GetMinRange() {
        return 0;
    }
```

```
    public float GetAspect() {
        return 2.5f;
    }

    public void OnDrawGizmos() {
        var gizmoMatrix = Gizmos.matrix;
        var gizmoColor = Gizmos.color;

        Gizmos.matrix = Matrix4x4.TRS( transform.position, transform.rotation,
transform.lossyScale );
        Gizmos.color = Color.red;
        Gizmos.DrawFrustum( Vector3.zero, GetFOV(), GetMaxRange(), GetMinRange(), GetAspect()
);

        Gizmos.matrix = gizmoMatrix;
        Gizmos.color = gizmoColor;
    }

    public void OnDrawGizmosSelected() {
        Handles.DrawWireDisc( transform.position, Vector3.up, GetDetectionRadius() );
    }
}
```

In diesem Beispiel haben wir zwei Methoden zum Zeichnen von Gizmos, eine, die zeichnet, wenn das Objekt aktiv ist (OnDrawGizmos), und eine, wenn das Objekt in der Hierarchie ausgewählt wird (OnDrawGizmosSelected).

```
public void OnDrawGizmos() {
    var gizmoMatrix = Gizmos.matrix;
    var gizmoColor = Gizmos.color;

    Gizmos.matrix = Matrix4x4.TRS( transform.position, transform.rotation,
transform.lossyScale );
    Gizmos.color = Color.red;
    Gizmos.DrawFrustum( Vector3.zero, GetFOV(), GetMaxRange(), GetMinRange(), GetAspect() );

    Gizmos.matrix = gizmoMatrix;
    Gizmos.color = gizmoColor;
}
```

Zuerst speichern wir die Gizmo-Matrix und -Farbe, da wir sie ändern und nach dem Beenden zurückkehren möchten, um keine anderen Gizmo-Zeichnungen zu beeinflussen.

Als Nächstes wollen wir den Kegelstumpf zeichnen, den unser Objekt hat. Wir müssen jedoch die Gizmos-Matrix so ändern, dass sie mit der Position, Rotation und Skalierung übereinstimmt. Wir haben auch die Farbe des Gizmos auf Rot gesetzt, um den Stumpf zu betonen. Wenn dies erledigt ist, können Sie *Gizmos.DrawFrustum anrufen,* um das Frustum in der *Szenenansicht* zu zeichnen.

Wenn wir mit dem Zeichnen fertig sind, setzen wir die Matrix und Farbe des Gizmos zurück.

```
public void OnDrawGizmosSelected() {
    Handles.DrawWireDisc( transform.position, Vector3.up, GetDetectionRadius() );
}
```

Wir möchten auch einen Erkennungsbereich zeichnen, wenn wir unser GameObject auswählen. Dies geschieht über die Klasse *Handles* , da die Klasse *Gizmos* über keine Methoden für Datenträger verfügt.

Die Verwendung dieser Form des Zeichnens von Gizmos führt zu der unten gezeigten Ausgabe.

Beispiel zwei

In diesem Beispiel wird das *DrawGizmo*- Attribut verwendet.

```
public class GizmoDrawerExample {

    [DrawGizmo( GizmoType.Selected | GizmoType.NonSelected, typeof( GizmoExample ) )]
    public static void DrawGizmo( GizmoExample obj, GizmoType type ) {
        var gizmoMatrix = Gizmos.matrix;
        var gizmoColor = Gizmos.color;

        Gizmos.matrix = Matrix4x4.TRS( obj.transform.position, obj.transform.rotation,
obj.transform.lossyScale );
        Gizmos.color = Color.red;
        Gizmos.DrawFrustum( Vector3.zero, obj.GetFOV(), obj.GetMaxRange(), obj.GetMinRange(),
obj.GetAspect() );

        Gizmos.matrix = gizmoMatrix;
        Gizmos.color = gizmoColor;

        if ( ( type & GizmoType.Selected ) == GizmoType.Selected ) {
            Handles.DrawWireDisc( obj.transform.position, Vector3.up, obj.GetDetectionRadius()
);
        }
    }
}
```

Auf diese Weise können Sie die Gizmo-Aufrufe von Ihrem Skript trennen. Die meisten davon verwenden den gleichen Code wie das andere Beispiel, mit Ausnahme von zwei Dingen.

```
[DrawGizmo( GizmoType.Selected | GizmoType.NonSelected, typeof( GizmoExample ) )]
public static void DrawGizmo( GizmoExample obj, GizmoType type ) {
```

Sie müssen das DrawGizmo-Attribut verwenden, das die Aufzählung GizmoType als ersten Parameter und einen Typ als zweiten Parameter verwendet. Der Typ sollte der Typ sein, den Sie zum Zeichnen des Gizmos verwenden möchten.

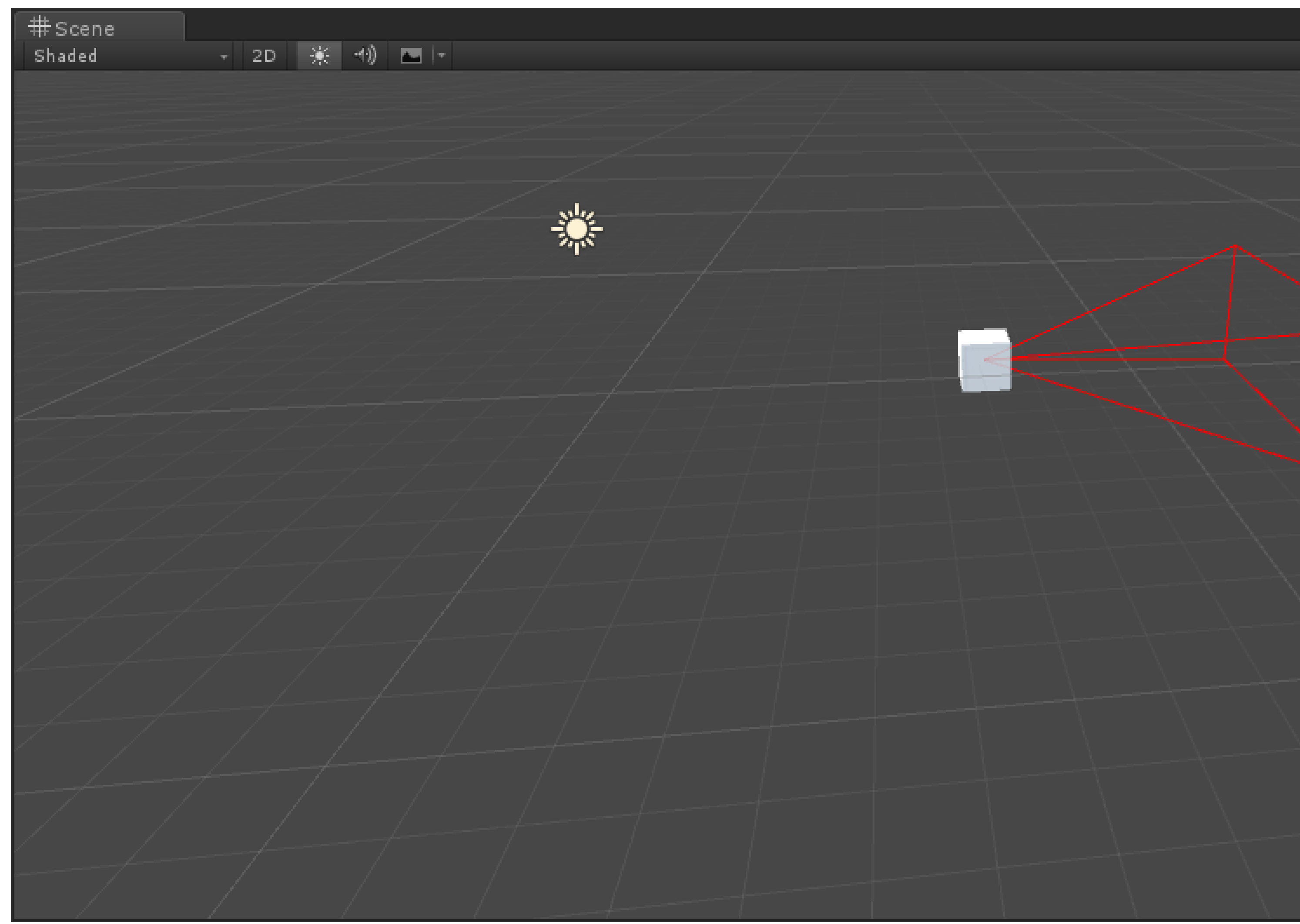
Die Methode zum Zeichnen des Gizmos muss statisch, öffentlich oder nicht öffentlich sein und kann beliebig benannt werden. Der erste Parameter ist der Typ, der mit dem Typ übereinstimmen sollte, der als zweiter Parameter im Attribut übergeben wird, und der zweite Parameter ist das Aufzählungs-GizmoType, das den aktuellen Status Ihres Objekts beschreibt.

```
if ( ( type & GizmoType.Selected ) == GizmoType.Selected ) {
    Handles.DrawWireDisc( obj.transform.position, Vector3.up, obj.GetDetectionRadius() );
}
```

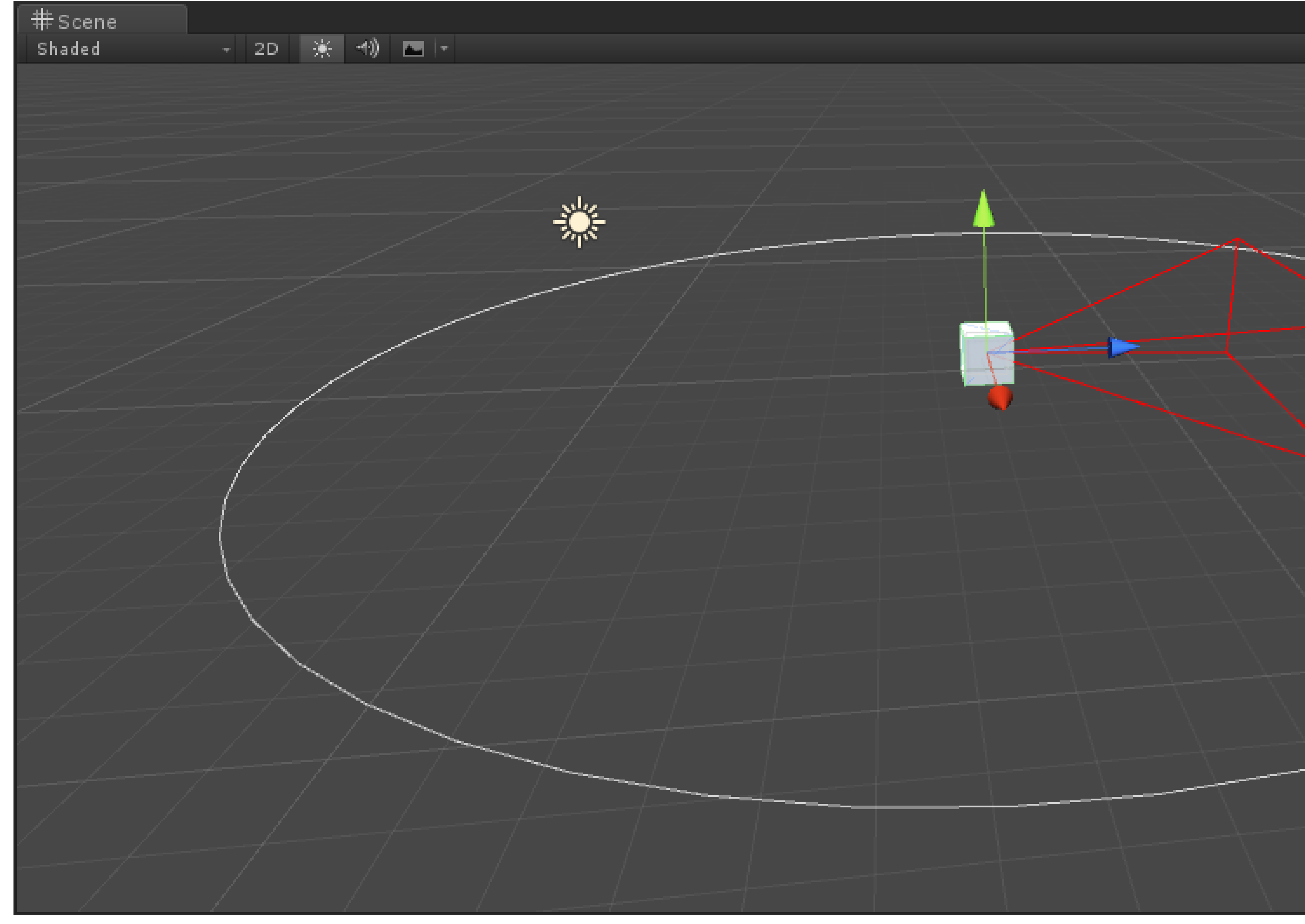
Der andere Unterschied besteht darin, dass Sie zur Überprüfung des GizmoType des Objekts eine UND-Prüfung des Parameters und des gewünschten Typs durchführen müssen.

Ergebnis

Nicht ausgewählt



Ausgewählt



Editor-Fenster

Warum ein Editorfenster?

Wie Sie vielleicht gesehen haben, können Sie in einem benutzerdefinierten Inspektor eine Menge tun. Wenn Sie nicht wissen, was ein benutzerdefinierter Inspektor ist, überprüfen Sie das Beispiel hier: http://www.riptutorial.com/unity3d/topic/2506 / Erweitern des Editors . An einem bestimmten Punkt möchten Sie jedoch möglicherweise ein Konfigurationsfenster oder eine angepasste Palette für Assets implementieren. In diesen Fällen verwenden Sie ein EditorWindow . Die Unity-Benutzeroberfläche besteht aus Editor-Windows, das Sie öffnen können (normalerweise durch die obere Leiste), Tabulatoren usw.

Erstellen Sie ein einfaches Editorfenster

Einfaches Beispiel

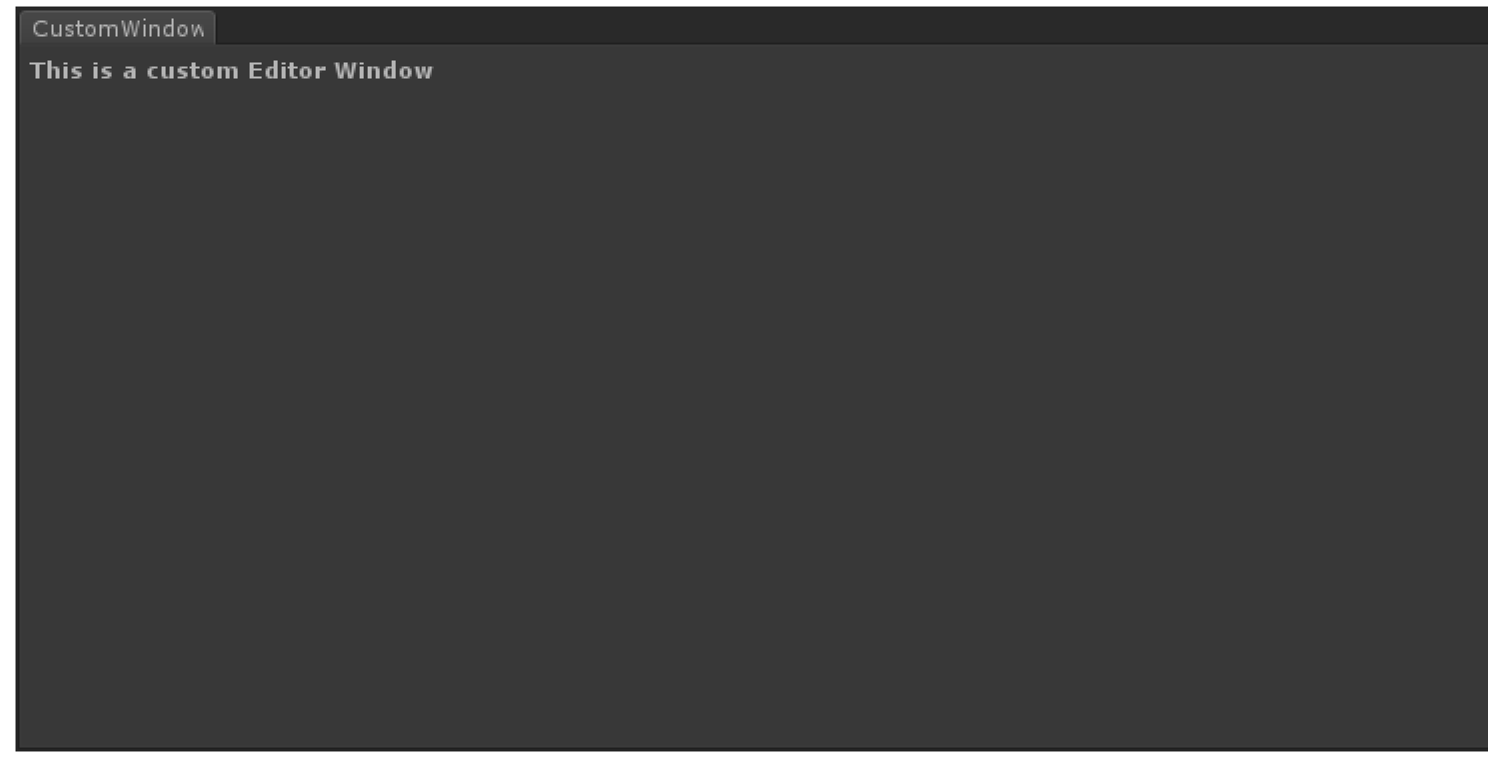
Das Erstellen eines benutzerdefinierten Editorfensters ist ziemlich einfach. Alles was Sie tun müssen, ist die EditorWindow-Klasse zu erweitern und die Init () - und die OnGUI () - Methode zu verwenden. Hier ist ein einfaches Beispiel:

```
using UnityEngine;
using UnityEditor;

public class CustomWindow : EditorWindow
{
    // Add menu named "Custom Window" to the Window menu
    [MenuItem("Window/Custom Window")]
    static void Init()
    {
        // Get existing open window or if none, make a new one:
        CustomWindow window = (CustomWindow) EditorWindow.GetWindow(typeof(CustomWindow));
        window.Show();
    }

    void OnGUI()
    {
        GUILayout.Label("This is a custom Editor Window", EditorStyles.boldLabel);
    }
}
```

Die 3 wichtigsten Punkte sind:

1. Vergessen Sie nicht, EditorWindow zu erweitern
2. Verwenden Sie das Init () wie im Beispiel angegeben. EditorWindow.GetWindow prüft, ob bereits ein CustomWindow erstellt wurde. Wenn nicht, wird eine neue Instanz erstellt. Damit stellen Sie sicher, dass Sie nicht mehrere Instanzen Ihres Fensters gleichzeitig haben
3. Verwenden Sie OnGUI () wie üblich, um Informationen in Ihrem Fenster anzuzeigen

Das Endergebnis wird so aussehen:



Tiefer gehen

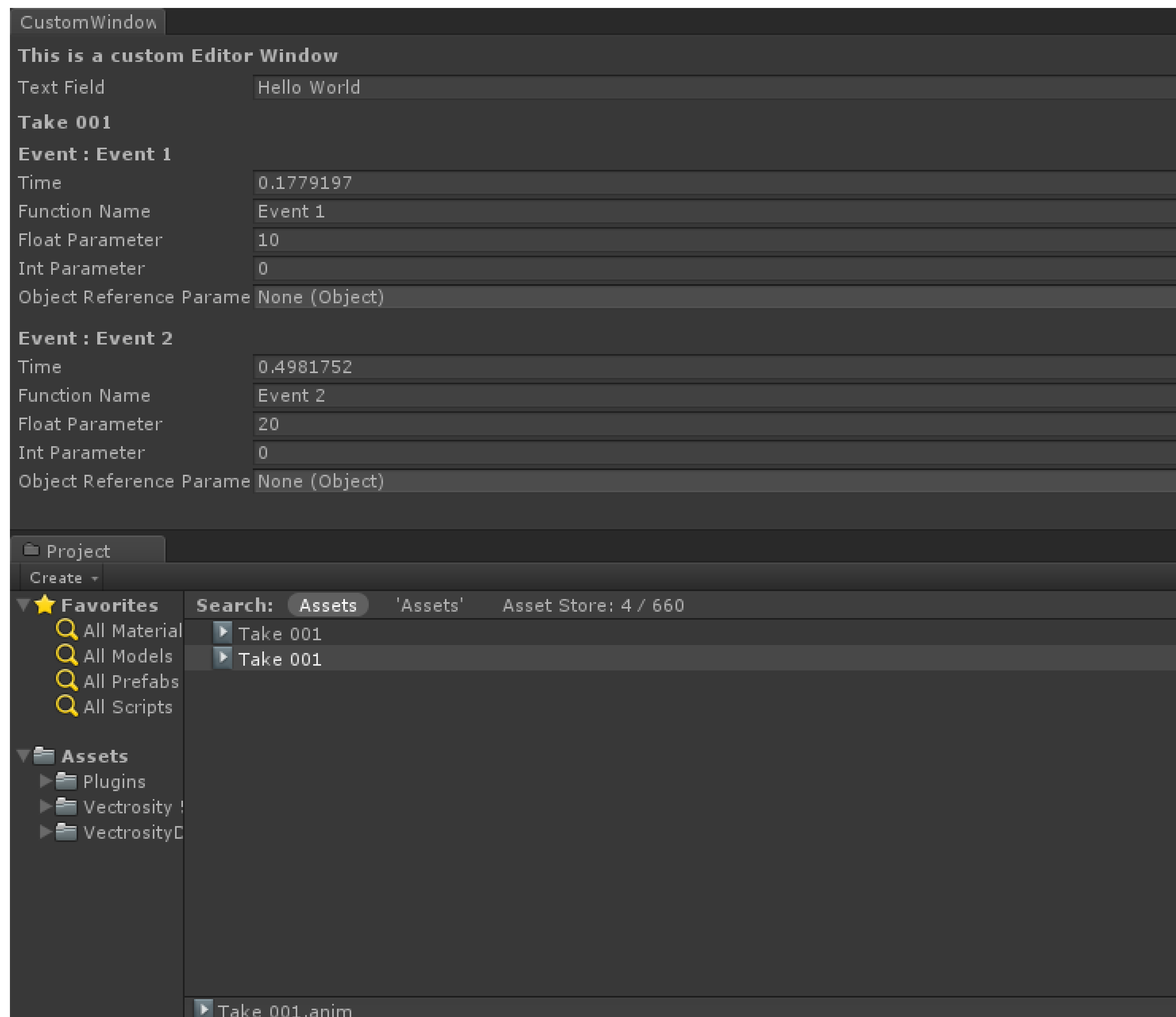
Natürlich werden Sie wahrscheinlich einige Assets mit diesem EditorWindow verwalten oder

ändern wollen. Hier ist ein Beispiel, in dem die Selection- Klasse verwendet wird (um die aktive Selection abzurufen) und die ausgewählten Asset-Eigenschaften über SerializedObject und SerializedProperty geändert werden .

```
    using System.Linq;
    using UnityEngine;
    using UnityEditor;

    public class CustomWindow : EditorWindow
    {
        private AnimationClip _animationClip;
        private SerializedObject _serializedClip;
        private SerializedProperty _events;

        private string _text = "Hello World";

        // Add menu named "Custom Window" to the Window menu
        [MenuItem("Window/Custom Window")]
        static void Init()
        {
            // Get existing open window or if none, make a new one:
            CustomWindow window = (CustomWindow) EditorWindow.GetWindow(typeof(CustomWindow));
            window.Show();
        }

        void OnGUI()
        {
            GUILayout.Label("This is a custom Editor Window", EditorStyles.boldLabel);

            // You can use EditorGUI, EditorGUILayout and GUILayout classes to display
anything you want
            // A TextField example
            _text = EditorGUILayout.TextField("Text Field", _text);

            // Note that you can modify an asset or a gameobject using an EditorWindow. Here
is a quick example with an AnimationClip asset
            // The _animationClip, _serializedClip and _events are set in OnSelectionChange()

            if (_animationClip == null || _serializedClip == null || _events == null) return;

            // We can modify our serializedClip like we would do in a Custom Inspector. For
example we can grab its events and display their information

            GUILayout.Label(_animationClip.name, EditorStyles.boldLabel);

            for (var i = 0; i < _events.arraySize; i++)
            {
                EditorGUILayout.BeginVertical();

                EditorGUILayout.LabelField(
                    "Event : " +
_events.GetArrayElementAtIndex(i).FindPropertyRelative("functionName").stringValue,
                    EditorStyles.boldLabel);

EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("time"),
true,
                    GUILayout.ExpandWidth(true));

EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("functionName"),
```

```
                    true, GUILayout.ExpandWidth(true));

EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("floatParameter"),

                    true, GUILayout.ExpandWidth(true));

EditorGUILayout.PropertyField(_events.GetArrayElementAtIndex(i).FindPropertyRelative("intParameter"),

                    true, GUILayout.ExpandWidth(true));
                EditorGUILayout.PropertyField(

_events.GetArrayElementAtIndex(i).FindPropertyRelative("objectReferenceParameter"), true,
                    GUILayout.ExpandWidth(true));

                EditorGUILayout.Separator();
                EditorGUILayout.EndVertical();
            }

            // Of course we need to Apply the modified properties. We don't our changes won't
be saved
            _serializedClip.ApplyModifiedProperties();
        }

        /// This Message is triggered when the user selection in the editor changes. That's
when we should tell our Window to Repaint() if the user selected another AnimationClip
        private void OnSelectionChange()
        {
            _animationClip =
                Selection.GetFiltered(typeof(AnimationClip),
SelectionMode.Assets).FirstOrDefault() as AnimationClip;
            if (_animationClip == null) return;

            _serializedClip = new SerializedObject(_animationClip);
            _events = _serializedClip.FindProperty("m_Events");
            Repaint();
        }
    }
```

Hier ist das Ergebnis:



Fortgeschrittene Themen

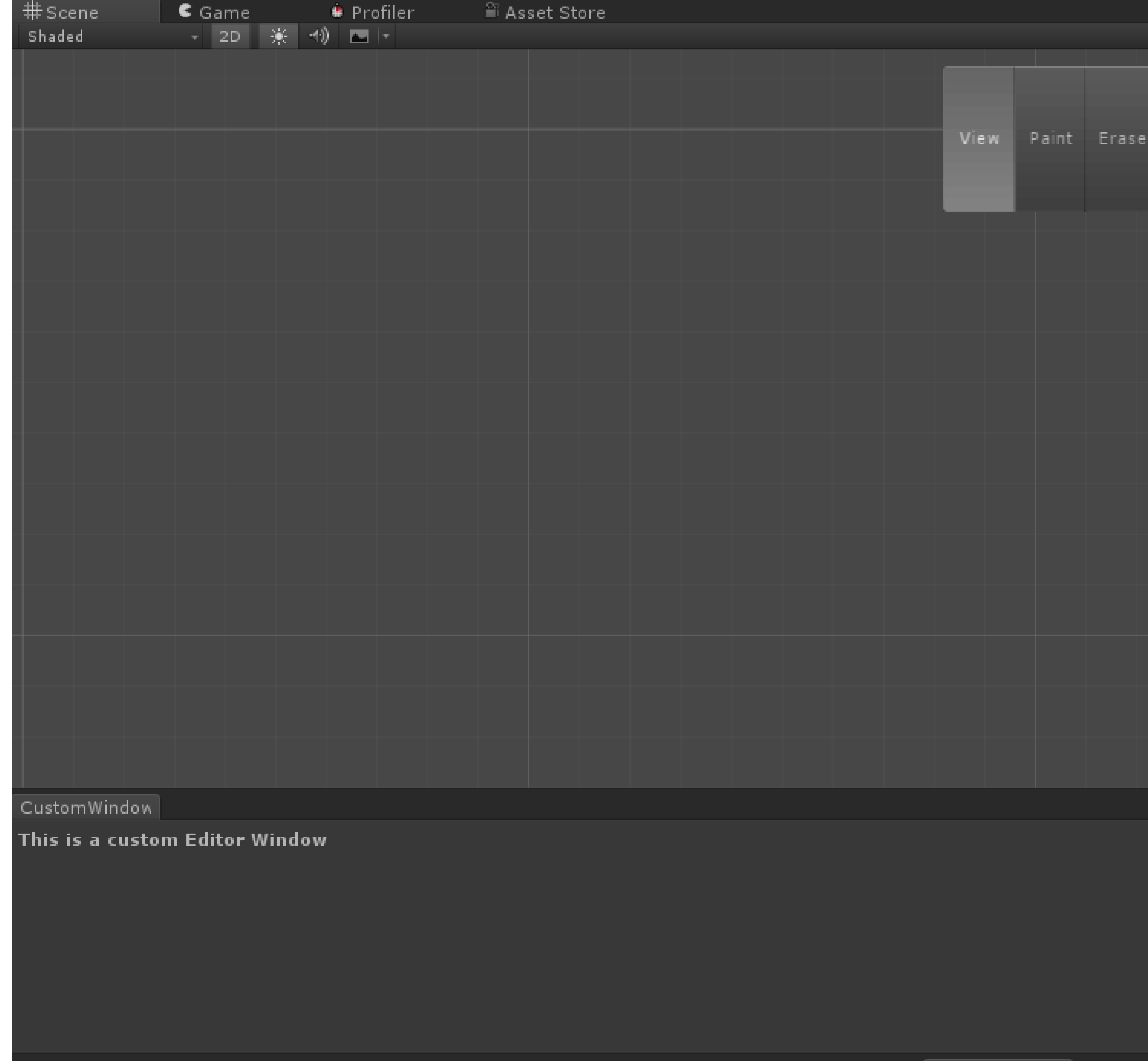
Im Editor können Sie einige wirklich fortgeschrittene Dinge erledigen, und die EditorWindow-Klasse eignet sich hervorragend für die Anzeige großer Informationsmengen. Die meisten fortschrittlichen Assets im Unity Asset Store (wie NodeCanvas oder PlayMaker) verwenden EditorWindow zum Anzeigen von benutzerdefinierten Ansichten.

In der SceneView zeichnen

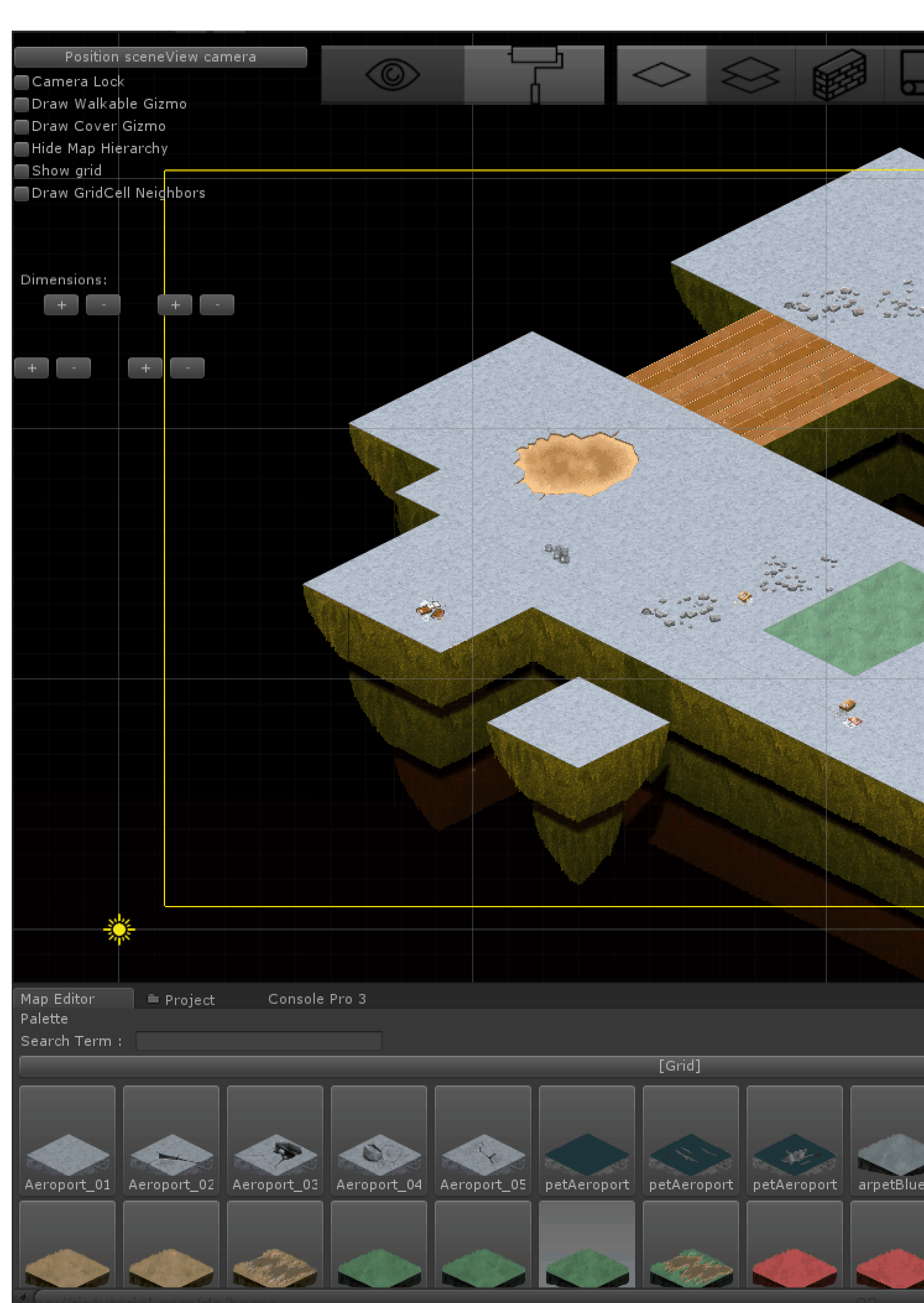
Eine interessante Sache, die Sie mit einem Editor-Fenster machen können, ist die Anzeige von Informationen direkt in SceneView. Auf diese Weise können Sie einen vollständig benutzerdefinierten Karten- / Welteditor erstellen, z. B. mit Ihrem benutzerdefinierten EditorWindow als Asset-Palette und zum Anhören von Klicks in der SceneView, um neue Objekte zu instanziieren. Hier ist ein Beispiel :

```
using UnityEngine;
using System;
using UnityEditor;

public class CustomWindow : EditorWindow {

    private enum Mode {
        View = 0,
        Paint = 1,
        Erase = 2
    }

    private Mode CurrentMode = Mode.View;

    [MenuItem ("Window/Custom Window")]
    static void Init () {
        // Get existing open window or if none, make a new one:
        CustomWindow window = (CustomWindow)EditorWindow.GetWindow (typeof (CustomWindow));
        window.Show();
    }

    void OnGUI () {
        GUILayout.Label ("This is a custom Editor Window", EditorStyles.boldLabel);
    }

    void OnEnable() {
        SceneView.onSceneGUIDelegate = SceneViewGUI;
        if (SceneView.lastActiveSceneView) SceneView.lastActiveSceneView.Repaint();
    }

    void SceneViewGUI(SceneView sceneView) {
        Handles.BeginGUI();
        // We define the toolbars' rects here
        var ToolBarRect = new Rect((SceneView.lastActiveSceneView.camera.pixelRect.width / 6),
10, (SceneView.lastActiveSceneView.camera.pixelRect.width * 4 / 6) ,
SceneView.lastActiveSceneView.camera.pixelRect.height / 5);
        GUILayout.BeginArea(ToolBarRect);
        GUILayout.BeginHorizontal();
        GUILayout.FlexibleSpace();
         CurrentMode = (Mode) GUILayout.Toolbar(
            (int) CurrentMode,
            Enum.GetNames(typeof(Mode)),
            GUILayout.Height(ToolBarRect.height));
        GUILayout.FlexibleSpace();
        GUILayout.EndHorizontal();
        GUILayout.EndArea();
        Handles.EndGUI();
    }
}
```

Dadurch wird eine Symbolleiste direkt in SceneView angezeigt



Hier ist ein kurzer Blick darauf, wie weit Sie gehen können:


Position sceneView camera
Camera Lock
Draw Walkable Gizmo
Draw Cover Gizmo
Hide Map Hierarchy
Show grid
Draw GridCell Neighbors
Dimensions:
Map Editor
Project
Console Pro 3
Palette
Search Term :
[Grid]
Aeroport_01
Aeroport_02
Aeroport_03
Aeroport_04
Aeroport_05
petAeroport
petAeroport
petAeroport
arpetBlue_

https://riptutorial.com/de/unity3d/topic/2506/erweitern-des-editors

Kapitel 14: GameObjects finden und sammeln

Syntax

- public static GameObject Find (Name der Zeichenfolge);
- public static GameObject FindGameObjectWithTag (String-Tag);
- public static GameObject [] FindGameObjectsWithTag (String-Tag);
- public static Object FindObjectOfType (Type-Typ);
- public static Object [] FindObjectsOfType (Type-Typ);

Bemerkungen

Welche Methode ist zu verwenden?

Seien Sie vorsichtig, wenn Sie zur Laufzeit nach GameObjects suchen, da dies Ressourcen verbrauchen kann. Insbesondere: Führen Sie FindObjectOfType oder Find in Update, FixedUpdate oder allgemeiner nicht in einer Methode aus, die einmal oder mehrmals pro Frame aufgerufen wird.

- Rufen Sie die Laufzeitmethoden `FindObjectOfType` und `Find` nur bei Bedarf auf
- `FindGameObjectWithTag` hat im Vergleich zu anderen string-basierten Methoden eine sehr gute Leistung. Unity führt getrennte Registerkarten für markierte Objekte und fragt diese anstelle der gesamten Szene ab.
- Für "statische" GameObjects (z. B. UI-Elemente und Prefabs), die im Editor erstellt wurden, verwenden Sie die serialisierbare GameObject-Referenz im Editor
- Behalten Sie Ihre Listen von GameObjects in Listen oder Arrays, die Sie selbst verwalten
- Wenn Sie viele GameObjects desselben Typs instanziieren, werfen Sie einen Blick auf Object Pooling
- Zwischenspeichern Sie Ihre Suchergebnisse, um zu vermeiden, dass die teuren Suchmethoden immer wieder ausgeführt werden.

Tiefer gehen

Abgesehen von den mit Unity gelieferten Methoden ist es relativ einfach, eigene Such- und Erfassungsmethoden zu erstellen.

- Im Falle von `FindObjectsOfType()` können Sie Ihre Skripte in einer `static` Auflistung eine Liste von sich selbst `FindObjectsOfType()` . Das Durchlaufen einer fertigen Objektliste ist viel schneller als das Durchsuchen und Untersuchen von Objekten in der Szene.

- Oder erstellen Sie ein Skript, das ihre Instanzen in einem auf Zeichenfolgen basierenden `Dictionary` speichert, und Sie verfügen über ein einfaches Tagging-System, das Sie erweitern können.

Examples

Suche nach dem Namen von GameObject

```
var go = GameObject.Find("NameOfTheObject");
```

Pros	Cons
Einfach zu verwenden	Die Leistung nimmt mit der Anzahl der Spielobjekte in der Szene ab
	Zeichenfolgen sind *schwache Referenzen* und verdächtigen Benutzerfehler

Suche nach den Tags von GameObject

```
var go = GameObject.FindGameObjectWithTag("Player");
```

Pros	Cons
Es können sowohl einzelne Objekte als auch ganze Gruppen durchsucht werden	Zeichenfolgen sind schwache Referenzen und verdächtigen Benutzerfehler.
Relativ schnell und effizient	Code ist nicht portierbar, da Tags in Skripts hart codiert sind.

Im Bearbeitungsmodus in Skripts eingefügt

```
[SerializeField]
GameObject[] gameObjects;
```

Pros	Cons
Gute Leistung	Die Objektauflistung ist statisch
Tragbarer Code	Kann sich nur auf GameObjects aus derselben Szene beziehen

Finden von GameObjects anhand von MonoBehaviour-Skripts

```
ExampleScript script = GameObject.FindObjectOfType<ExampleScript>();
GameObject go = script.gameObject;
```

> `FindObjectOfType()` gibt `null` wenn keiner gefunden wird.

Pros	Cons
Stark getippt	Die Leistung nimmt mit der Anzahl der für die Bewertung benötigten Spielobjekte ab
Es können sowohl einzelne Objekte als auch ganze Gruppen durchsucht werden	

Finden Sie GameObjects nach Namen aus untergeordneten Objekten

```
Transform tr = GetComponent<Transform>().Find("NameOfTheObject");
GameObject go = tr.gameObject;
```

`Find` gibt `null` wenn keiner gefunden wird

Pros	Cons
Begrenzter, klar definierter Suchumfang	Strings sind schwache Referenzen

GameObjects finden und sammeln online lesen: https://riptutorial.com/de/unity3d/topic/3793/gameobjects-finden-und-sammeln

Kapitel 15: Implementierung der MonoBehaviour-Klasse

Examples

Keine überschriebenen Methoden

Sie müssen `Awake` , `Start` , `Update` und andere Methoden nicht überschreiben, weil sie keine virtuellen Methoden sind, die in einer Basisklasse definiert sind.

Wenn Sie zum ersten Mal auf Ihr Skript zugreifen, durchsucht die Skriptlaufzeit das Skript, um festzustellen, ob einige Methoden definiert sind. Wenn dies der Fall ist, werden diese Informationen zwischengespeichert und die Methoden werden ihrer jeweiligen Liste hinzugefügt. Diese Listen werden dann zu verschiedenen Zeitpunkten einfach durchlaufen.

Der Grund, warum diese Methoden nicht virtuell sind, liegt in der Leistung. Wenn alle Skripts `Awake` , `Start` , `OnEnable` , `OnDisable` , `Update` , `LateUpdate` und `FixedUpdate` , werden diese alle zu ihren Listen hinzugefügt. Dies würde bedeuten, dass alle diese Methoden ausgeführt werden. Normalerweise wäre dies kein großes Problem, jedoch erfolgen alle diese Methodenaufrufe von der nativen Seite (C ++) zur verwalteten Seite (C #), was mit einem Leistungsaufwand verbunden ist.

Nun stellen Sie sich vor, alle diese Methoden stehen in ihren Listen und einige / die meisten von ihnen verfügen möglicherweise nicht einmal über einen eigentlichen Methodenkörper. Dies würde bedeuten, dass eine große Menge an Leistung verschwendet wird, wenn Methoden aufgerufen werden, die gar nichts tun. Um dies zu verhindern, hat Unity die Verwendung virtueller Methoden deaktiviert und ein Messagingsystem eingerichtet, das sicherstellt, dass diese Methoden nur dann aufgerufen werden, wenn sie tatsächlich definiert sind, wodurch unnötige Methodenaufrufe eingespart werden.

Weitere Informationen zu diesem Thema finden Sie hier in einem Unity-Blog: 10000 Update () Aufrufe und mehr zu IL2CPP hier: Eine Einführung in IL2CPP-Interna

Implementierung der MonoBehaviour-Klasse online lesen:
https://riptutorial.com/de/unity3d/topic/2304/implementierung-der-monobehaviour-klasse

Kapitel 16: Importeure und (Post-) Prozessoren

Syntax

- AssetPostprocessor.OnPreprocessTexture ()

Bemerkungen

Verwenden Sie `String.Contains()` , um nur Assets zu verarbeiten, die eine angegebene Zeichenfolge in ihren Asset-Pfaden haben.

```
if (assetPath.Contains("ProcessThisFolder"))
{
    // Process asset
}
```

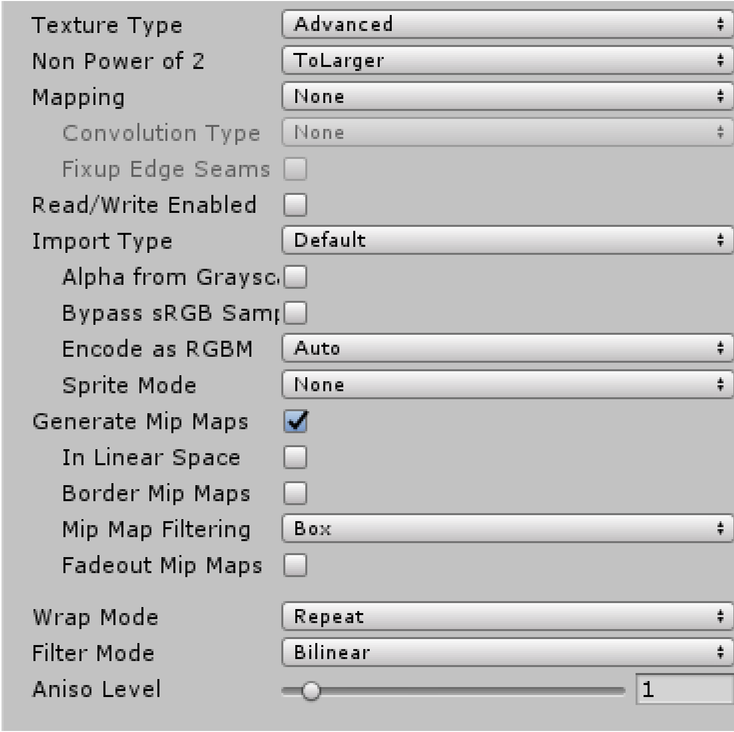
Examples

Textur-Postprozessor

Erstellen `TexturePostProcessor.cs` Datei `TexturePostProcessor.cs` beliebiger Stelle im Ordner **Assets** :

```
using UnityEngine;
using UnityEditor;

public class TexturePostProcessor : AssetPostprocessor
{
    void OnPostprocessTexture(Texture2D texture)
    {
        TextureImporter importer = assetImporter as TextureImporter;
        importer.anisoLevel = 1;
        importer.filterMode = FilterMode.Bilinear;
        importer.mipmapEnabled = true;
        importer.npotScale = TextureImporterNPOTScale.ToLarger;
        importer.textureType = TextureImporterType.Advanced;
    }
}
```

Jedes Mal, wenn Unity eine Textur importiert, hat es die folgenden Parameter:



Wenn Sie einen Postprozessor verwenden, können Sie die Texturparameter nicht ändern, indem Sie die **Importeinstellungen** im Editor **bearbeiten** .
Wenn Sie auf die Schaltfläche " **Übernehmen"** klicken, wird die Textur erneut importiert und der Postprozessor-Code wird erneut ausgeführt.

Ein einfacher Importeur

Angenommen, Sie haben eine benutzerdefinierte Datei, für die Sie einen Importer erstellen möchten. Es kann eine .xls-Datei sein oder was auch immer. In diesem Fall verwenden wir eine JSON-Datei, da es einfach ist, aber wir werden eine benutzerdefinierte Erweiterung auswählen, um zu erkennen, welche Dateien unsere sind.

Nehmen wir an, das Format der JSON-Datei ist

```
{
  "someValue": 123,
  "someOtherValue": 456.297,
  "someBoolValue": true,
  "someStringValue": "this is a string",
}
```

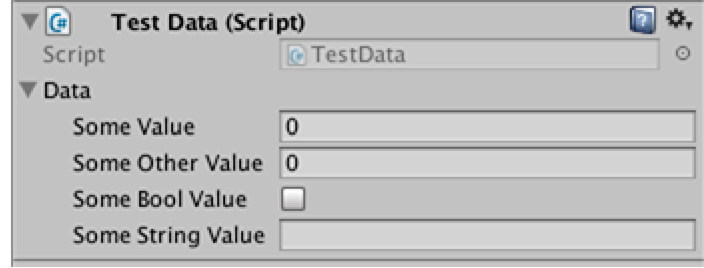
Speichern wir das jetzt als `Example.test` irgendwo *außerhalb* von Assets.

Als Nächstes `MonoBehaviour` ein `MonoBehaviour` mit einer benutzerdefinierten Klasse nur für die Daten. Die benutzerdefinierte Klasse dient ausschließlich zur Erleichterung der Deserialisierung des JSON. Sie müssen KEINE benutzerdefinierten Klasse verwenden, aber dieses Beispiel wird kürzer. Wir speichern dies in `TestData.cs`

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

```
public class TestData : MonoBehaviour {

    [System.Serializable]
    public class Data {
        public int someValue = 0;
        public float someOtherValue = 0.0f;
        public bool someBoolValue = false;
        public string someStringValue = "";
    }

    public Data data = new Data();
}
```

Wenn Sie dieses Skript manuell zu einem GameObject hinzufügen, sehen Sie so etwas wie



Als nächstes erstellen Sie einen `Editor` Ordner irgendwo unter `Assets` . Ich kann auf jeder Ebene sein. `TestDataAssetPostprocessor.cs` im Editor-Ordner eine `TestDataAssetPostprocessor.cs` Datei und `TestDataAssetPostprocessor.cs` diese ein.

```
using UnityEditor;
using UnityEngine;
using System.Collections;

public class TestDataAssetPostprocessor : AssetPostprocessor
{
    const string s_extension = ".test";

    // NOTE: Paths start with "Assets/"
    static bool IsFileWeCareAbout(string path)
    {
        return System.IO.Path.GetExtension(path).Equals(
           s_extension,
           System.StringComparison.Ordinal);
    }

    static void HandleAddedOrChangedFile(string path)
    {
        string text = System.IO.File.ReadAllText(path);
        // should we check for error if the file can't be parsed?
        TestData.Data newData = JsonUtility.FromJson<TestData.Data>(text);

        string prefabPath = path + ".prefab";
        // Get the existing prefab
        GameObject existingPrefab =
            AssetDatabase.LoadAssetAtPath(prefabPath, typeof(Object)) as GameObject;
        if (!existingPrefab)
        {
            // If no prefab exists make one
            GameObject newGameObject = new GameObject();
            newGameObject.AddComponent<TestData>();
```

```
            PrefabUtility.CreatePrefab(prefabPath,
                                       newGameObject,
                                       ReplacePrefabOptions.Default);
            GameObject.DestroyImmediate(newGameObject);
            existingPrefab =
                AssetDatabase.LoadAssetAtPath(prefabPath, typeof(Object)) as GameObject;
        }

        TestData testData = existingPrefab.GetComponent<TestData>();
        if (testData != null)
        {
            testData.data = newData;
            EditorUtility.SetDirty(existingPrefab);
        }
    }

    static void HandleRemovedFile(string path)
    {
        // Decide what you want to do here. If the source file is removed
        // do you want to delete the prefab? Maybe ask if you'd like to
        // remove the prefab?
        // NOTE: Because you might get many calls (like you deleted a
        // subfolder full of .test files you might want to get all the
        // filenames and ask all at once ("delete all these prefabs?").
    }

    static void OnPostprocessAllAssets (string[] importedAssets, string[] deletedAssets,
string[] movedAssets, string[] movedFromAssetPaths)
    {
        foreach (var path in importedAssets)
        {
            if (IsFileWeCareAbout(path))
            {
                HandleAddedOrChangedFile(path);
            }
        }

        foreach (var path in deletedAssets)
        {
            if (IsFileWeCareAbout(path))
            {
                HandleRemovedFile(path);
            }
        }

        for (var ii = 0; ii < movedAssets.Length; ++ii)
        {
            string srcStr = movedFromAssetPaths[ii];
            string dstStr = movedAssets[ii];

            // the source was moved, let's move the corresponding prefab
            // NOTE: We don't handle the case if there already being
            // a prefab of the same name at the destination
            string srcPrefabPath = srcStr + ".prefab";
            string dstPrefabPath = dstStr + ".prefab";

            AssetDatabase.MoveAsset(srcPrefabPath, dstPrefabPath);
        }
    }
}
```

`Example.test` dem `Example.test` sollten Sie die oben erstellte Datei `Example.test` in Ihren Unity Assets-Ordner ziehen und ablegen können. Das entsprechende Prefab sollte nun erstellt werden. Wenn Sie `Example.test` bearbeiten, werden die Daten im Prefab sofort aktualisiert. Wenn Sie das Prefab in die Szenenhierarchie ziehen, werden sowohl die Aktualisierung als auch die Änderungen in `Example.test` . Wenn Sie `Example.test` in einen anderen Ordner verschieben, wird das entsprechende Prefab mit verschoben. Wenn Sie ein Feld in einer Instanz ändern und dann die Datei `Example.test` ändern, werden nur die Felder aktualisiert, die Sie in der Instanz nicht `Example.test` .

Verbesserungen: Nachdem Sie in dem obigen Beispiel `Example.test` in Ihren `Assets` Ordner gezogen haben, werden Sie sehen, dass es sowohl `Example.test` als auch `Example.test.prefab` . Es wäre großartig zu wissen, dass es so funktioniert, als würden die `Example.test` funktionieren. Wir würden magisch nur `Example.test` und es ist ein `AssetBundle` oder `AssetBundle` . Wenn Sie wissen, geben Sie bitte dieses Beispiel an

Importeure und (Post-) Prozessoren online lesen: https://riptutorial.com/de/unity3d/topic/5279/importeure-und--post---prozessoren

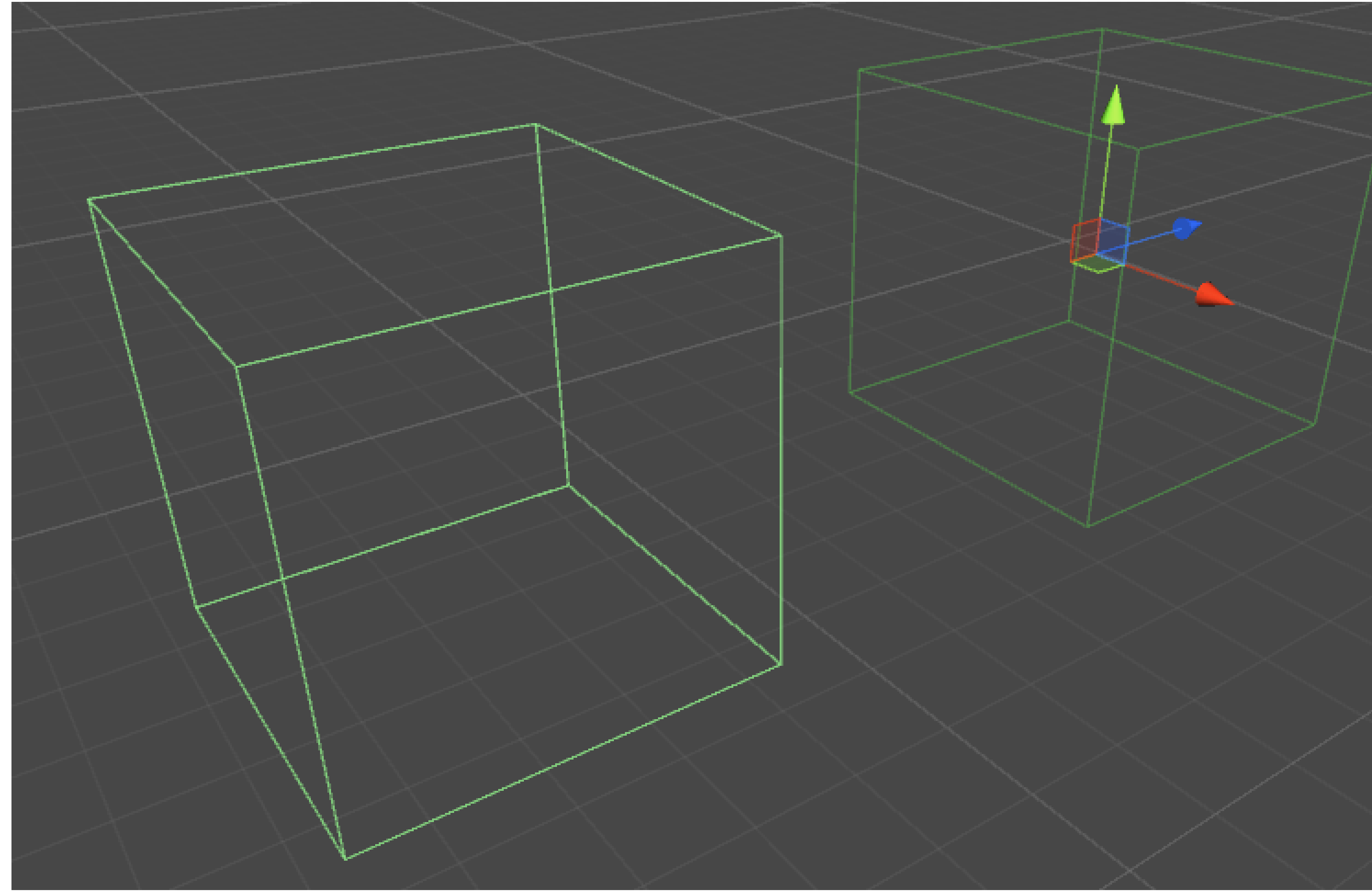
Kapitel 17: Kollision

Examples

Colliders

Box Collider

Ein primitiver Collider, der wie ein Quader geformt ist.



Eigenschaften

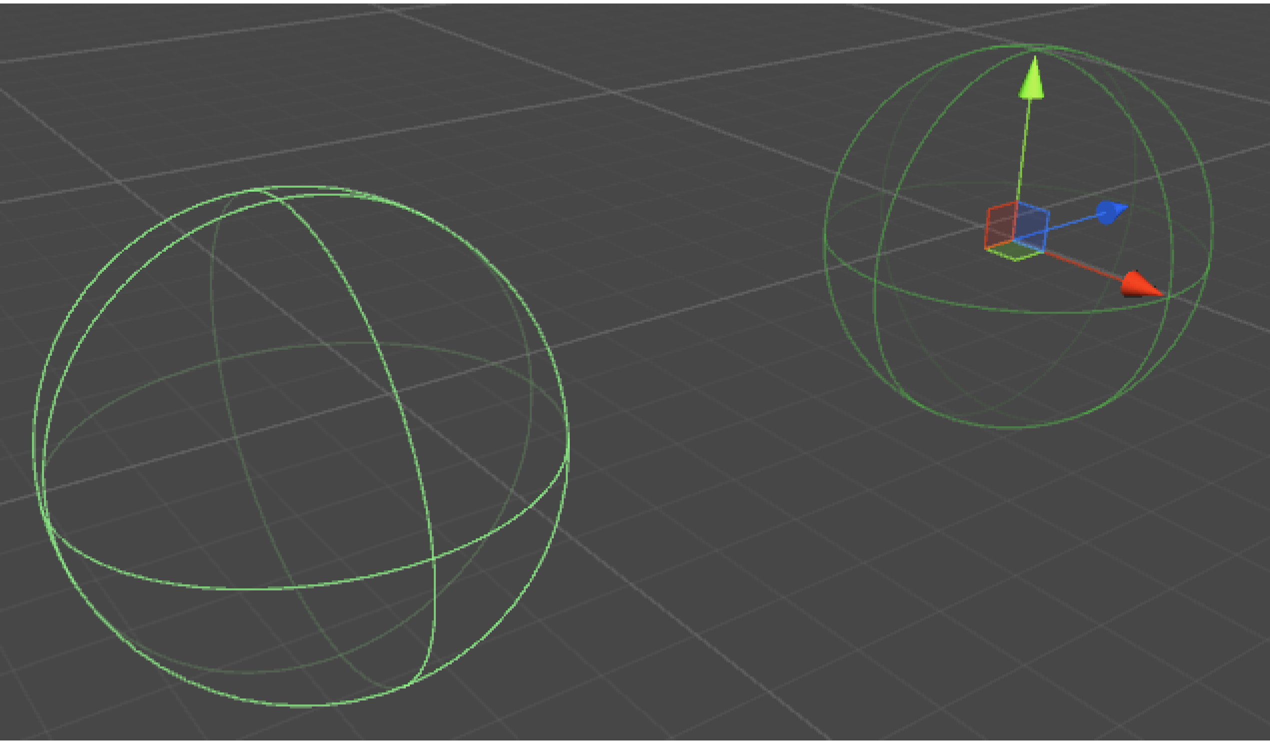
- **Ist Trigger** - Wenn diese Option aktiviert ist, ignoriert der Box Collider die Physik und wird zum Trigger Collider
- **Material** - Ein Verweis auf das Physikmaterial des Box Colliders, falls angegeben
- **Center** - Die zentrale Position des Box Colliders im lokalen Raum
- **Größe** - Die Größe des Box Colliders im lokalen Raum

Beispiel

```
// Add a Box Collider to the current GameObject.
BoxCollider myBC = BoxCollider)myGameObject.gameObject.AddComponent(typeof(BoxCollider));

// Make the Box Collider into a Trigger Collider.
myBC.isTrigger= true;

// Set the center of the Box Collider to the center of the GameObject.
myBC.center = Vector3.zero;

// Make the Box Collider twice as large.
myBC.size = 2;
```

Sphere Collider

Ein primitiver Collider, der wie eine Kugel geformt ist.



Eigenschaften

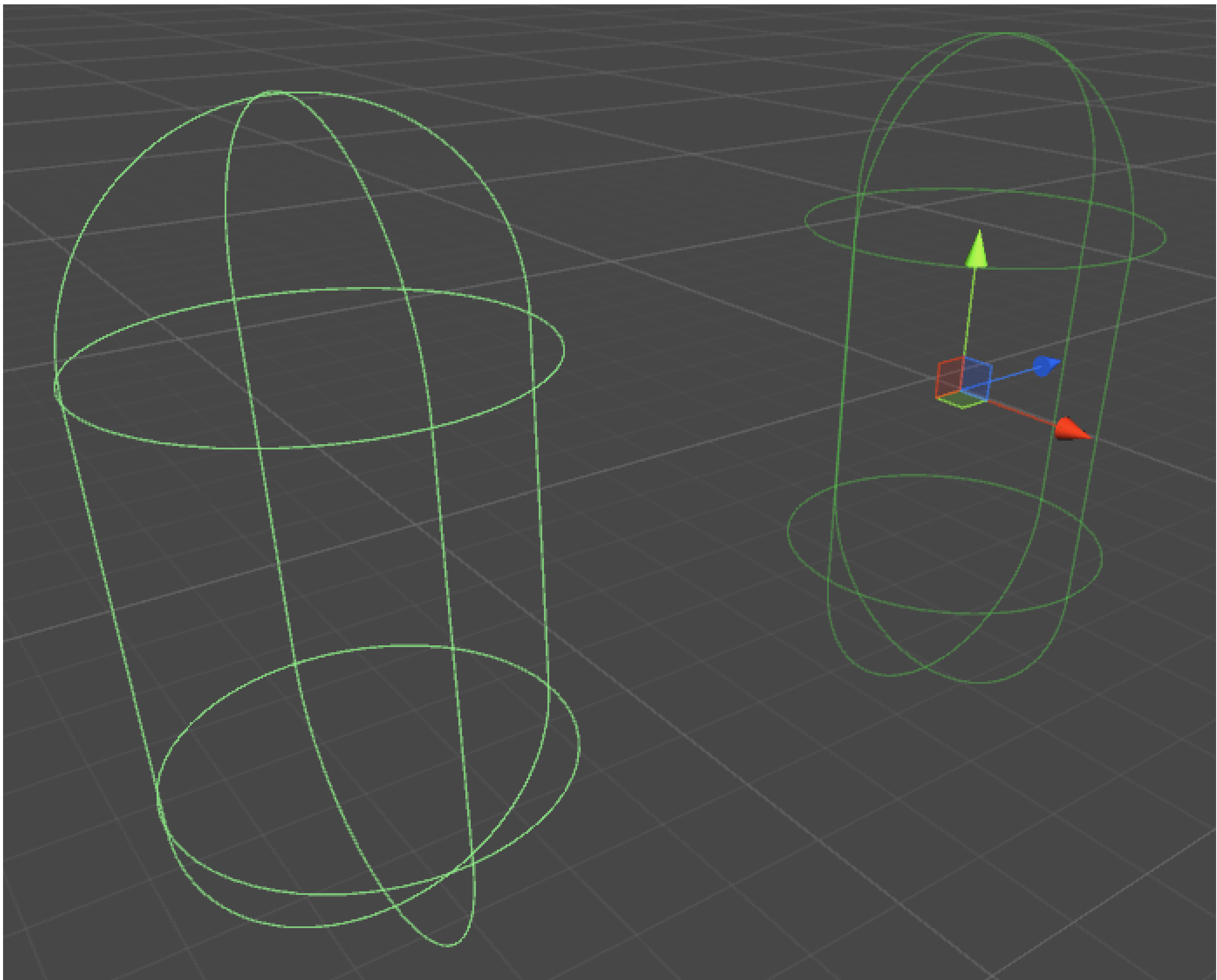
- **Ist Trigger** - Wenn diese Option aktiviert ist, ignoriert der Sphere Collider die Physik und wird zum Trigger Collider
- **Material** - Eine Referenz, falls angegeben, auf das Physikmaterial des Sphere Collider
- **Center** - Die zentrale Position des Sphere Colliders im lokalen Raum
- **Radius** - Der Radius des Colliders

Beispiel

```
// Add a Sphere Collider to the current GameObject.
SphereCollider mySC =
SphereCollider)myGameObject.gameObject.AddComponent(typeof(SphereCollider));

// Make the Sphere Collider into a Trigger Collider.
mySC.isTrigger= true;

// Set the center of the Sphere Collider to the center of the GameObject.
mySC.center = Vector3.zero;

// Make the Sphere Collider twice as large.
mySC.radius = 2;
```

Capsule Collider

Zwei durch einen Zylinder verbundene Halbkugeln.



Eigenschaften

- **Ist Trigger** - Wenn diese Option aktiviert ist, ignoriert der Capsule Collider die Physik und wird zum Trigger Collider
- **Material** - Eine Referenz, falls angegeben, auf das Physikmaterial des Capsule Collider
- **Center** - Die zentrale Position des Capsule Collider im lokalen Raum
- **Radius** - Der Radius im lokalen Raum
- **Höhe** - Gesamthöhe des Colliders
- **Richtung** - Die Orientierungsachse im lokalen Raum

Beispiel

```
// Add a Capsule Collider to the current GameObject.
CapsuleCollider myCC =
CapsuleCollider)myGameObject.gameObject.AddComponent(typeof(CapsuleCollider));

// Make the Capsule Collider into a Trigger Collider.
myCC.isTrigger= true;

// Set the center of the Capsule Collider to the center of the GameObject.
myCC.center = Vector3.zero;

// Make the Sphere Collider twice as tall.
myCC.height= 2;

// Make the Sphere Collider twice as wide.
myCC.radius= 2;

// Set the axis of lengthwise orientation to the X axis.
myCC.direction = 0;

// Set the axis of lengthwise orientation to the Y axis.
myCC.direction = 1;

// Set the axis of lengthwise orientation to the Y axis.
myCC.direction = 2;
```

Wheel Collider

Eigenschaften

- **Masse** - Die Masse des Wheel Colliders
- **Radius** - Der Radius im lokalen Raum

- **Raddämpfungsrate** - Dämpfungswert für den Wheel Collider
- **Abhängungsdistanz** - Maximale Ausdehnung entlang der Y-Achse im lokalen Raum
- **Abstand zwischen Kraftpunkten** - Der Punkt, an dem Kräfte angewendet werden,
- **Center** - Zentrum des Wheel Colliders im lokalen Raum

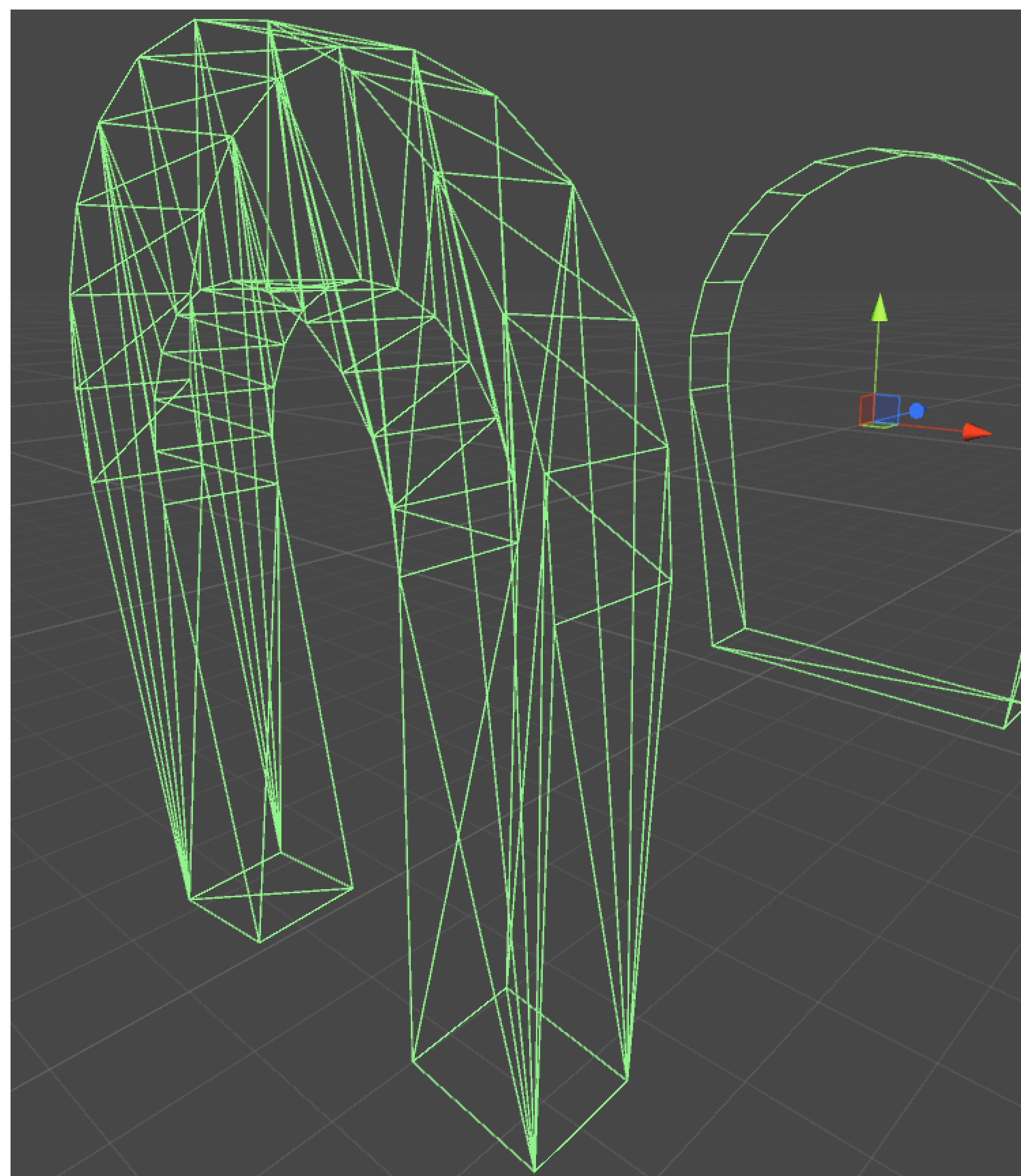
Federungsfeder

- **Frühling** - die Rate, mit der das Rad versucht, zur Zielposition zurückzukehren
- **Dämpfer** - Ein größerer Wert dämpft die Geschwindigkeit mehr und die Federung bewegt sich langsamer
- **Zielposition** - der Standardwert ist 0,5, bei 0 ist die Suspension erschöpft, bei 1 ist sie voll ausgefahren
- **Vorwärts- / seitwärts gerichtete Reibung** - wie sich der Reifen beim Vorwärts- oder seitwärtsrollen verhält

Beispiel

Mesh Collider

Ein Collider, der auf einem Mesh-Asset basiert.



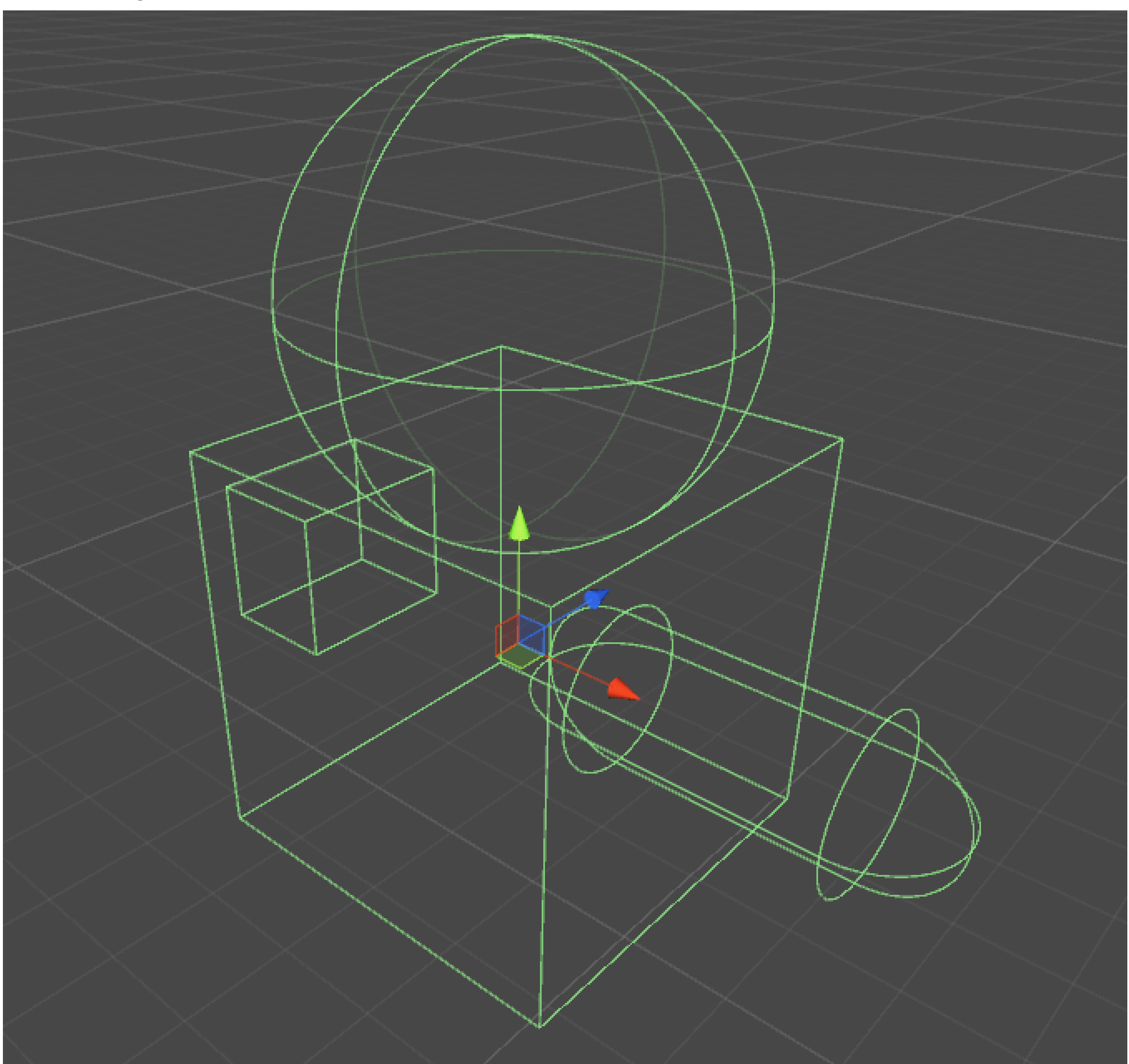
Eigenschaften

- **Ist Trigger** - Wenn diese Option aktiviert ist, ignoriert der Box Collider die Physik und wird zum Trigger Collider

- **Material** - Ein Verweis auf das Physikmaterial des Box Colliders, falls angegeben
- **Mesh** - Ein Verweis auf das Mesh, auf dem der Collider basiert
- **Convex** - Convex Mesh Colliders sind auf 255 Polygone beschränkt. Wenn diese Option aktiviert ist, kann dieser Collider mit anderen Mesh Colliders kollidieren

Beispiel

Wenn Sie mehr als einen Collider auf ein GameObject anwenden, nennen wir es einen zusammengesetzten Collider.



Wheel Collider

Der Wheel Collider in Unity ist auf dem PhysX Wheel Collider von Nvidia aufgebaut und hat daher viele ähnliche Eigenschaften. Technisch gesehen ist Einheit ein "unitless" Programm, aber um

alles sinnvoll zu machen, sind einige Standardeinheiten erforderlich.

Grundeigenschaften

- Masse - das Gewicht des Rades in Kilogramm, dies wird für den Radimpuls und den Moment des Interieurs beim Drehen verwendet.
- Radius - in Metern der Radius des Kolliders.
- Raddämpfungsrate - Stellt ein, wie "reaktionsfähig" die Räder auf das aufgebrachte Drehmoment reagieren.
- Aufhängungsentfernung - Gesamtentfernung in Metern, die das Rad zurücklegen kann
- Force App Point Distance (Kraftanwendungspunktabstand) - Wo ist die Kraft, die von der Federung auf den übergeordneten starren Körper ausgeübt wird
- Center - Die Mittelposition des Rades

Suspendierungseinstellungen

- Feder - Dies ist die Federkonstante K in Newton / Meter in der Gleichung:

  Kraft = Federkonstante * Abstand

Ein guter Ausgangspunkt für diesen Wert sollte die Gesamtmasse Ihres Fahrzeugs sein, dividiert durch die Anzahl der Räder, multipliziert mit einer Zahl zwischen 50 und 100. Wenn Sie beispielsweise ein Fahrzeug mit 2.000 kg und 4 Rädern haben, müsste jedes Rad Unterstützung 500kg. Multiplizieren Sie dies mit 75, und Ihre Federkonstante sollte 37.500 Newton / Meter betragen.

- Dämpfer - das Äquivalent eines Stoßdämpfers in einem Auto. Höhere Raten machen die Spannung "steifer" und niedrigere Raten machen sie "weicher" und schwingen eher.
  Ich kenne die Einheiten oder die Gleichung dafür nicht, ich denke es hat jedoch mit einer Frequenzgleichung in der Physik zu tun.

Seitliche Reibungseinstellungen

Die Reibungskurve in Einheit hat einen Schlupfwert, der dadurch bestimmt wird, wie stark das Rad (in m / s) von der gewünschten Position gegenüber der tatsächlichen Position rutscht.

- Extremum Slip - Dies ist die maximale Menge (in m / s), die ein Rad rutschen kann, bevor es die Traktion verliert

- Extremwert - Dies ist die maximale Reibung, die auf ein Rad ausgeübt werden sollte.

Die Werte für Exrtremum Slip sollten für die meisten realistischen Autos zwischen 0,2 und 2 m / s liegen. 2 m / s sind ungefähr 6 Fuß pro Sekunde oder 5 Meilen pro Stunde, was viel Schlupf ist. Wenn Sie der Meinung sind, dass Ihr Fahrzeug einen Wert von mehr als 2 m / s für Schlupf haben muss, sollten Sie die maximale Reibung (Extremum Value) erhöhen.

Max Fraction (Extremum Value) ist der Reibungskoeffizient in der Gleichung:

  Reibungskraft (in Newton) = Reibungskoeffizient * Abwärtskraft (in Newton)

Das bedeutet mit einem Koeffizienten von 1, dass Sie die gesamte Kraft des Fahrzeugs + der Aufhängung entgegen der Schlupfrichtung anwenden. In realen Anwendungen sind Werte über 1 selten, aber nicht unmöglich. Für einen Reifen auf trockenem Asphalt sind Werte zwischen 0,7 und 0 realistisch, daher ist der Standardwert 1,0 vorzuziehen.

Dieser Wert sollte realistisch nicht 2,5 überschreiten, da merkwürdiges Verhalten auftreten wird. Sie beginnen z. B. nach rechts zu drehen, aber da dieser Wert so hoch ist, wird eine große Kraft entgegen Ihrer Richtung ausgeübt und Sie beginnen in die Kurve zu gleiten anstatt wegzurutschen.

Wenn Sie beide Werte maximal eingestellt haben, sollten Sie den Asymptoten-Slip und den Wert erhöhen. Der Asymptoten-Schlupf sollte zwischen 0,5 und 2 m / s liegen und definiert den Reibungskoeffizienten für jeden Schlupfwert nach dem Asymptoten-Schlupf. Wenn Sie feststellen, dass sich Ihre Fahrzeuge gut verhalten, bis sie die Traktion brechen, und sich dann wie auf Eis verhält, sollten Sie den Asymptote-Wert erhöhen. Wenn Sie feststellen, dass Ihr Fahrzeug nicht driften kann, sollten Sie den Wert verringern.

Vorwärtsreibung

Die Vorwärtsreibung ist identisch mit der seitlichen Reibung, mit der Ausnahme, dass hiermit festgelegt ist, wie viel Traktion das Rad in Bewegungsrichtung hat. Wenn die Werte zu niedrig sind, werden Ihre Fahrzeuge durchgebrannt und drehen die Reifen, bevor sie langsam vorwärts fahren. Wenn es zu hoch ist, hat Ihr Fahrzeug möglicherweise die Tendenz, einen schlechten oder schlimmeren Schlag zu versuchen.

Zusätzliche Bemerkungen

Erwarten Sie nicht, dass Sie einen GTA-Klon oder einen anderen Rennklon erstellen können, indem Sie einfach diese Werte anpassen. In den meisten Fahrspielen werden diese Werte für verschiedene Geschwindigkeiten, Gelände und Wendewerte ständig im Skript geändert. Wenn Sie beim Drücken einer Taste lediglich ein konstantes Drehmoment auf die Radkollider ausüben, verhält sich Ihr Spiel nicht realistisch. In der Realität haben Autos Drehmomentkurven und Getriebe, um das auf die Räder aufgebrachte Drehmoment zu ändern.

Um optimale Ergebnisse zu erzielen, sollten Sie diese Werte so lange anpassen, bis Sie ein Auto bekommen, das einigermaßen gut reagiert, und dann das Raddrehmoment, den maximalen Drehwinkel und die Reibungswerte im Skript ändern.

Weitere Informationen zu den Radkollidern finden Sie in der Dokumentation zu Nvidia:
http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/Vehicles.html

Trigger-Colliders

Methoden

- `OnTriggerEnter()`
- `OnTriggerStay()`
- `OnTriggerExit()`

Sie können einen Collider in einen **Trigger** `OnTriggerEnter()` , um die `OnTriggerEnter()` , `OnTriggerStay()` und `OnTriggerExit()` verwenden. Ein Trigger Collider reagiert nicht physisch auf Kollisionen, andere GameObjects passieren sie einfach. Sie sind nützlich, um zu erkennen, wann sich ein anderes GameObject in einem bestimmten Bereich befindet oder nicht. Wenn Sie zum Beispiel ein Objekt sammeln, möchten wir es vielleicht einfach durchlaufen lassen, aber erkennen, wenn dies passiert.

Trigger Collider Scripting

Beispiel

Die folgende Methode ist ein Beispiel für einen Trigger-Listener, der erkennt, wenn ein anderer Collider in den Collider eines GameObject (z. B. eines Spielers) eintritt. Triggermethoden können zu jedem Skript hinzugefügt werden, das einem GameObject zugewiesen ist.

```
void OnTriggerEnter(Collider other)
{
    //Check collider for specific properties (Such as tag=item or has component=item)
}
```

Kollision online lesen: https://riptutorial.com/de/unity3d/topic/4405/kollision

Kapitel 18: Kommunikation mit dem Server

Examples

Erhalten

Get holt Daten vom Webserver. und `new WWW("https://urlexample.com");` mit einer URL aber ohne einen zweiten Parameter wird ein **Get durchgeführt** .

dh

```
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour
{
    public string url = "http://google.com";

    IEnumerator Start()
    {
        WWW www = new WWW(url); // One get.
        yield return www;
        Debug.Log(www.text); // The data of the url.
    }
}
```

Einfache Buchung (Post-Felder)

Jede Instanz von **WWW** mit einem zweiten Parameter ist ein *Beitrag* .

Hier ein Beispiel, um die *Benutzer-ID* und das *Kennwort* an den Server zu senden.

```
void Login(string id, string pwd)
{
    WWWForm dataParameters = new WWWForm();    // Create a new form.
    dataParameters.AddField("username", id);
    dataParameters.AddField("password", pwd);   // Add fields.
    WWW www = new WWW(url+"/account/login",dataParameters);
    StartCoroutine("PostdataEnumerator", www);
}

IEnumerator PostdataEnumerator(WWW www)
{
    yield return www;
    if (!string.IsNullOrEmpty(www.error))
    {
        Debug.Log(www.error);
    }
    else
    {
        Debug.Log("Data Submitted");
    }
}
```

Post (Datei hochladen)

Eine Datei auf den Server hochladen ist ebenfalls ein Beitrag. Sie können eine Datei ganz einfach über das **WWW** hochladen (**siehe** unten):

Laden Sie eine Zip-Datei auf den Server hoch

```
string mainUrl = "http://server/upload/";
string saveLocation;

void Start()
{
    saveLocation = "ftp:///home/xxx/x.zip"; // The file path.
    StartCoroutine(PrepareFile());
}

// Prepare The File.
IEnumerator PrepareFile()
{
    Debug.Log("saveLoacation = " + saveLocation);

    // Read the zip file.
    WWW loadTheZip = new WWW(saveLocation);

    yield return loadTheZip;

    PrepareStepTwo(loadTheZip);
}

void PrepareStepTwo(WWW post)
{
    StartCoroutine(UploadTheZip(post));
}

// Upload.
IEnumerator UploadTheZip(WWW post)
{
    // Create a form.
    WWWForm form = new WWWForm();

    // Add the file.
    form.AddBinaryData("myTestFile.zip",post.bytes,"myFile.zip","application/zip");

    // Send POST request.
    string url = mainUrl;
    WWW POSTZIP = new WWW(url,form);

    Debug.Log("Sending zip...");
    yield return POSTZIP;
    Debug.Log("Zip sent!");
}
```

In diesem Beispiel wird **Coroutine** zum Vorbereiten und Hochladen der Datei verwendet. Wenn Sie mehr über Unity-Coroutines erfahren möchten, besuchen Sie Coroutines .

Eine Anfrage an den Server senden

Es gibt viele Möglichkeiten, mit Servern über Unity als Client zu kommunizieren (je nach Zweck sind einige Methoden besser als andere). Zunächst muss die Notwendigkeit des Servers ermittelt werden, um Operationen effektiv vom und zum Server senden zu können. In diesem Beispiel senden wir einige Daten zur Validierung an unseren Server.

Höchstwahrscheinlich hat der Programmierer auf seinem Server eine Art Handler eingerichtet, um Ereignisse zu empfangen und dem Client entsprechend zu antworten - dies ist jedoch nicht in diesem Beispiel.

C #:

```
using System.Net;
using System.Text;

public class TestCommunicationWithServer
{
    public string SendDataToServer(string url, string username, string password)
    {
        WebClient client = new WebClient();

        // This specialized key-value pair will store the form data we're sending to the
server
        var loginData = new System.Collections.Specialized.NameValueCollection();
        loginData.Add("Username", username);
        loginData.Add("Password", password);

        // Upload client data and receive a response
        byte[] opBytes = client.UploadValues(ServerIpAddress, "POST", loginData);

        // Encode the response bytes into a proper string
        string opResponse = Encoding.UTF8.GetString(opBytes);

        return opResponse;
    }
```

Als erstes müssen Sie mit ihren using-Anweisungen werfen, um die Klassen WebClient und NameValueCollection verwenden zu können.

In diesem Beispiel nimmt die SendDataToServer-Funktion drei (optionale) Zeichenfolgeparameter an:

1. URL des Servers, mit dem wir kommunizieren
2. Erste Daten
3. Zweite Daten, die wir an den Server senden

Der Benutzername und das Kennwort sind die optionalen Daten, die ich an den Server sende. In diesem Beispiel verwenden wir es, um von einer Datenbank oder einem anderen externen Speicher aus weiter validiert zu werden.

Nachdem wir unsere Struktur eingerichtet haben, instanziieren wir einen neuen WebClient, der zum Senden unserer Daten verwendet wird. Nun müssen wir unsere Daten in unsere NameValueCollection laden und die Daten auf den Server hochladen.

Die UploadValues-Funktion berücksichtigt auch 3 notwendige Parameter:

1. IP-Adresse des Servers
2. HTTP-Methode
3. Daten, die Sie senden (in unserem Fall Benutzername und Passwort)

Diese Funktion gibt ein Byte-Array der Antwort vom Server zurück. Das zurückgegebene Byte-Array muss in eine richtige Zeichenfolge codiert werden, um die Antwort tatsächlich bearbeiten und analysieren zu können.

Man könnte so etwas machen:

```
if(opResponse.Equals(ReturnMessage.Success))
{
    Debug.Log("Unity client has successfully sent and validated data on server.");
}
```

Nun sind Sie vielleicht immer noch verwirrt, also werde ich kurz erklären, wie Sie mit einem Antwortserver umgehen.

In diesem Beispiel verwende ich PHP, um die Antwort des Clients zu bearbeiten. Ich würde die Verwendung von PHP als Back-End-Skriptsprache empfehlen, da sie äußerst vielseitig, einfach zu bedienen und vor allem schnell ist. Es gibt definitiv andere Möglichkeiten, mit einer Antwort auf einem Server umzugehen, aber PHP ist meiner Meinung nach bei weitem die einfachste und einfachste Implementierung in Unity.

PHP:

```
// Check to see if the unity client send the form data
if(!isset($_REQUEST['Username']) || !isset($_REQUEST['Password']))
{
    echo "Empty";
}
else
{
    // Unity sent us the data - its here so do whatever you want

    echo "Success";

}
```

Dies ist also der wichtigste Teil - das "Echo". Wenn unser Client die Daten auf den Server hochlädt, speichert der Client die Antwort (oder Ressource) in diesem Byte-Array. Sobald der Client die Antwort erhalten hat, wissen Sie, dass die Daten überprüft wurden, und Sie können im Client fortfahren, sobald das Ereignis eingetreten ist. Sie müssen auch darüber nachdenken, welche Art von Daten Sie (in einem gewissen Umfang) senden und wie Sie den tatsächlich gesendeten Betrag minimieren können.

Dies ist also nur eine Möglichkeit, Daten von Unity zu senden / zu empfangen - es gibt andere Möglichkeiten, die je nach Projekt für Sie wirksamer sind.

Kommunikation mit dem Server online lesen:
https://riptutorial.com/de/unity3d/topic/5578/kommunikation-mit-dem-server

Kapitel 19: Mobile Plattformen

Syntax

- public static int Input.touchCount
- public static Touch Input.GetTouch (int index)

Examples

Berührung erkennen

Um eine Berührung in Unity zu erkennen, ist es ziemlich einfach, dass Sie `Input.GetTouch()` und einen Index übergeben.

```
using UnityEngine;
using System.Collections;

public class TouchExample : MonoBehaviour {
    void Update() {
        if (Input.touchCount > 0 && Input.GetTouch(0).phase == TouchPhase.Began)
        {
            //Do Stuff
        }
    }
}
```

oder

```
using UnityEngine;
using System.Collections;

public class TouchExample : MonoBehaviour {
    void Update() {
        for(int i = 0; i < Input.touchCount; i++)
        {
            if (Input.GetTouch(i).phase == TouchPhase.Began)
            {
                //Do Stuff
            }
        }
    }
}
```

Diese Beispiele erhalten den letzten Rahmen des Spiels.

TouchPhase

Innerhalb der TouchPhase-Enumeration gibt es 5 verschiedene TouchPhase-Typen

- Begann - ein Finger berührte den Bildschirm
- Bewegt - ein Finger bewegt sich auf dem Bildschirm
- Stationär - ein Finger ist auf dem Bildschirm, bewegt sich jedoch nicht
- Beendet - ein Finger wurde vom Bildschirm gehoben
- Abgebrochen - Das System hat das Tracking für die Berührung abgebrochen

Zum Beispiel, um das Objekt zu verschieben, an das dieses Skript basierend auf Berührung über den Bildschirm angehängt wird.

```
public class TouchMoveExample : MonoBehaviour
{
    public float speed = 0.1f;

    void Update () {
        if(Input.touchCount > 0 && Input.GetTouch(0).phase == TouchPhase.Moved)
        {
            Vector2 touchDeltaPosition = Input.GetTouch(0).deltaPosition;
            transform.Translate(-touchDeltaPosition.x * speed, -touchDeltaPosition.y * speed,
0);
        }
    }
}
```

Mobile Plattformen online lesen: https://riptutorial.com/de/unity3d/topic/6285/mobile-plattformen

Kapitel 20: Multiplattform-Entwicklung

Examples

Compiler-Definitionen

Compiler-Definitionen führen plattformspezifischen Code aus. Mit ihnen können Sie kleine Unterschiede zwischen verschiedenen Plattformen machen.

- Trigger Game Center-Erfolge auf Apple-Geräten und Google Play-Erfolge auf Android-Geräten.
- Ändern Sie die Symbole in Menüs (Windows-Logo in Windows, Linux-Pinguin in Linux).
- Eventuell plattformspezifische Mechanik je nach Plattform.
- Und vieles mehr...

```
void Update(){

#if UNITY_IPHONE
    //code here is only called when running on iPhone
#endif

#if UNITY_STANDALONE_WIN && !UNITY_EDITOR
    //code here is only ran in a unity game running on windows outside of the editor
#endif

//other code that will be ran regardless of platform

}
```

Eine vollständige Liste der Unity-Compiler-Definitionen finden Sie hier

Plattformspezifische Methoden für Teilklassen organisieren

Teilklassen bieten eine saubere Möglichkeit, die Kernlogik Ihrer Skripts von plattformspezifischen Methoden zu trennen.

Partielle Klassen und Methoden werden mit dem Schlüsselwort `partial` markiert. Dies signalisiert dem Compiler, die Klasse "open" zu lassen und den Rest der Implementierung in anderen Dateien zu suchen.

```
// ExampleClass.cs
using UnityEngine;

public partial class ExampleClass : MonoBehaviour
{
    partial void PlatformSpecificMethod();

    void OnEnable()
    {
        PlatformSpecificMethod();
```

```
    }
}
```

Jetzt können wir Dateien für unsere plattformspezifischen Skripts erstellen, die die Teilmethode implementieren. Partielle Methoden können Parameter haben (auch `ref`), müssen aber `void` .

```
// ExampleClass.Iphone.cs

#if UNITY_IPHONE
using UnityEngine;

public partial class ExampleClass
{
    partial void PlatformSpecificMethod()
    {
        Debug.Log("I am an iPhone");
    }
}
#endif
```

```
// ExampleClass.Android.cs

#if UNITY_ANDROID
using UnityEngine;

public partial class ExampleClass
{
    partial void PlatformSpecificMethod()
    {
        Debug.Log("I am an Android");
    }
}
#endif
```

Wenn keine Teilmethode implementiert ist, lässt der Compiler den Aufruf aus.

> Tipp: Dieses Muster ist auch hilfreich, wenn Sie Editor-spezifische Methoden erstellen.

Multiplattform-Entwicklung online lesen: https://riptutorial.com/de/unity3d/topic/4816/multiplattform-entwicklung

Kapitel 21: Objekt-Pooling

Examples

Objektpool

Wenn Sie ein Spiel erstellen, müssen Sie manchmal immer wieder Objekte des gleichen Typs erstellen und zerstören. Sie können dies einfach tun, indem Sie ein Prefab erstellen und es instanziieren / zerstören, wann immer Sie dies benötigen. Dies ist jedoch ineffizient und kann Ihr Spiel verlangsamen.

Eine Möglichkeit, dieses Problem zu umgehen, ist das Pooling von Objekten. Im Grunde bedeutet dies, dass Sie einen Pool (mit oder ohne Begrenzung der Anzahl) von Objekten haben, die Sie wann immer möglich wiederverwenden können, um unnötige Instanziierung oder Zerstörung zu verhindern.

Unten sehen Sie ein Beispiel für einen einfachen Objektpool

```
public class ObjectPool : MonoBehaviour
{
    public GameObject prefab;
    public int amount = 0;
    public bool populateOnStart = true;
    public bool growOverAmount = true;

    private List<GameObject> pool = new List<GameObject>();

    void Start()
    {
        if (populateOnStart && prefab != null && amount > 0)
        {
            for (int i = 0; i < amount; i++)
            {
                var instance = Instantiate(Prefab);
                instance.SetActive(false);
                pool.Add(instance);
            }
        }
    }

    public GameObject Instantiate (Vector3 position, Quaternion rotation)
    {
        foreach (var item in pool)
        {
            if (!item.activeInHierarchy)
            {
                item.transform.position = position;
                item.transform.rotation = rotation;
                item.SetActive( true );
                return item;
            }
        }

        if (growOverAmount)
```

```
        {
            var instance = (GameObject)Instantiate(prefab, position, rotation);
            pool.Add(instance);
            return instance;
        }

        return null;
    }
}
```

Lassen Sie uns zuerst die Variablen durchgehen

```
public GameObject prefab;
public int amount = 0;
public bool populateOnStart = true;
public bool growOverAmount = true;

private List<GameObject> pool = new List<GameObject>();
```

- `GameObject prefab` : Dies ist der Prefab, den der Objektpool verwendet, um neue Objekte in den Pool zu instantiieren.
- `int amount` : Dies ist die maximale Anzahl von Elementen, die sich im Pool befinden können. Wenn Sie ein anderes Element instanziieren möchten und der Pool bereits sein Limit erreicht hat, wird ein anderes Element aus dem Pool verwendet.
- `bool populateOnStart` : Sie können den Pool beim Start `bool populateOnStart` oder nicht. Dadurch wird der Pool mit Instanzen des Prefab gefüllt, so dass Sie beim ersten Aufruf von `Instantiate` ein bereits vorhandenes Objekt erhalten
- `bool growOverAmount` : Wenn Sie diese `bool growOverAmount` auf true setzen, wächst der Pool immer dann, wenn in einem bestimmten Zeitraum mehr als der Betrag angefordert wird. Sie können die Anzahl der Elemente, die Sie in Ihren Pool aufnehmen möchten, nicht immer genau vorhersagen, sodass Sie bei Bedarf mehr Pool hinzufügen können.
- `List<GameObject> pool` : Dies ist der Pool, der Ort, an dem alle instanziierten / zerstörten Objekte gespeichert werden.

Schauen wir uns nun die `Start` Funktion an

```
void Start()
{
    if (populateOnStart && prefab != null && amount > 0)
    {
        for (int i = 0; i < amount; i++)
        {
            var instance = Instantiate(Prefab);
            instance.SetActive(false);
            pool.Add(instance);
        }
    }
}
```

In der `prefab` prüfen wir, ob die Liste beim Start `prefab` soll und ob das `prefab` festgelegt wurde und der Betrag größer als 0 ist (sonst würden wir unbegrenzt `prefab`).

Dies ist nur eine einfache Methode, um neue Objekte zu instantiieren und in den Pool zu setzen.

Zu beachten ist, dass wir alle Instanzen auf inaktiv setzen. Auf diese Weise sind sie noch nicht im Spiel sichtbar.

Als nächstes gibt es die `Instantiate` Funktion, bei der die meiste Magie stattfindet

```
public GameObject Instantiate (Vector3 position, Quaternion rotation)
{
    foreach (var item in pool)
    {
        if (!item.activeInHierarchy)
        {
            item.transform.position = position;
            item.transform.rotation = rotation;
            item.SetActive(true);
            return item;
        }
    }

    if (growOverAmount)
    {
        var instance = (GameObject)Instantiate(prefab, position, rotation);
        pool.Add(instance);
        return instance;
    }

    return null;
}
```

Die `Instantiate` Funktion sieht genauso aus wie Unitys eigene `Instantiate` Funktion, es sei denn, das Prefab wurde bereits als Klassenmitglied bereitgestellt.

Der erste Schritt der `Instantiate` Funktion besteht darin, zu prüfen, ob sich momentan ein inaktives Objekt im Pool befindet. Dies bedeutet, dass wir das Objekt wiederverwenden und dem Anforderer zurückgeben können. Wenn sich ein inaktives Objekt im Pool befindet, setzen wir die Position und die Rotation, setzen es auf aktiv (andernfalls könnte es versehentlich wiederverwendet werden, falls Sie es vergessen, es zu aktivieren) und an den Anforderer zurückgeben.

Der zweite Schritt erfolgt nur, wenn sich keine inaktiven Elemente im Pool befinden und der Pool den ursprünglichen Betrag überschreiten darf. Was passiert, ist einfach: Eine weitere Instanz des Prefab wird erstellt und dem Pool hinzugefügt. Wenn Sie das Wachstum des Pools zulassen, können Sie die richtige Anzahl von Objekten im Pool anzeigen.

Der dritte "Schritt" erfolgt nur, wenn sich keine inaktiven Elemente im Pool befinden und der Pool *nicht* vergrößert werden darf. In diesem Fall erhält der Anforderer ein ungültiges GameObject, was bedeutet, dass nichts verfügbar war und ordnungsgemäß behandelt werden sollte, um `NullReferenceExceptions` zu verhindern.

Wichtig!

Um sicherzustellen, dass Ihre Gegenstände wieder in den Pool gelangen, sollten Sie die Spielobjekte **nicht** zerstören. Das einzige, was Sie tun müssen, ist, sie inaktiv zu setzen, wodurch sie für die Wiederverwendung durch den Pool verfügbar sind.

Einfacher Objektpool

Nachfolgend finden Sie ein Beispiel für einen Objektpool, in dem ein bestimmter Objekttyp gemietet und zurückgegeben werden kann. Um den Objektpool zu erstellen, sind eine Funktion für die Erstellungsfunktion und eine Aktion zum Zerstören des Objekts erforderlich, um dem Benutzer Flexibilität zu geben. Wenn ein Objekt angefordert wird, wenn der Pool leer ist, wird ein neues Objekt erstellt. Wenn der Pool Objekte enthält, werden Objekte aus dem Pool entfernt und zurückgegeben.

Objektpool

```
public class ResourcePool<T> where T : class
{
    private readonly List<T> objectPool = new List<T>();
    private readonly Action<T> cleanUpAction;
    private readonly Func<T> createAction;

    public ResourcePool(Action<T> cleanUpAction, Func<T> createAction)
    {
        this.cleanUpAction = cleanUpAction;
        this.createAction = createAction;
    }

    public void Return(T resource)
    {
        this.objectPool.Add(resource);
    }

    private void PurgeSingleResource()
    {
        var resource = this.Rent();
        this.cleanUpAction(resource);
    }

    public void TrimResourcesBy(int count)
    {
        count = Math.Min(count, this.objectPool.Count);
        for (int i = 0; i < count; i++)
        {
            this.PurgeSingleResource();
        }
    }

    public T Rent()
    {
        int count = this.objectPool.Count;
        if (count == 0)
        {
            Debug.Log("Creating new object.");
            return this.createAction();
        }
        else
        {
            Debug.Log("Retrieving existing object.");
            T resource = this.objectPool[count-1];
            this.objectPool.RemoveAt(count-1);
            return resource;
        }
```

```
    }
}
```

Verwendungsbeispiel

```
public class Test : MonoBehaviour
{
    private ResourcePool<GameObject> objectPool;

    [SerializeField]
    private GameObject enemyPrefab;

    void Start()
    {
        this.objectPool = new ResourcePool<GameObject>(Destroy,() =>
Instantiate(this.enemyPrefab) );
    }

    void Update()
    {
        // To get existing object or create new from pool
        var newEnemy = this.objectPool.Rent();
        // To return object to pool
        this.objectPool.Return(newEnemy);
        // In this example the message 'Creating new object' should only be seen on the frame
call
        // after that the same object in the pool will be returned.
    }
}
```

Ein weiterer einfacher Objektpool

Ein anderes Beispiel: Eine Waffe, die Kugeln abschießt.

Die Waffe fungiert als Objektpool für die von ihr erstellten Aufzählungszeichen.

```
public class Weapon : MonoBehaviour {

    // The Bullet prefab that the Weapon will create
    public Bullet bulletPrefab;

    // This List is our object pool, which starts out empty
    private List<Bullet> availableBullets = new List<Bullet>();

    // The Transform that will act as the Bullet starting position
    public Transform bulletInstantiationPoint;

    // To spawn a new Bullet, this method either grabs an available Bullet from the pool,
    // otherwise Instantiates a new Bullet
    public Bullet CreateBullet () {
        Bullet newBullet = null;

        // If a Bullet is available in the pool, take the first one and make it active
        if (availableBullets.Count > 0) {
            newBullet = availableBullets[availableBullets.Count - 1];

            // Remove the Bullet from the pool
```

```
            availableBullets.RemoveAt(availableBullets.Count - 1);

            // Set the Bullet's position and make its GameObject active
            newBullet.transform.position = bulletInstantiationPoint.position;
            newBullet.gameObject.SetActive(true);
        }
        // If no Bullets are available in the pool, Instantiate a new Bullet
        else {
            newBullet newObject = Instantiate(bulletPrefab, bulletInstantiationPoint.position,
Quaternion.identity);

            // Set the Bullet's Weapon so we know which pool to return to later on
            newBullet.weapon = this;
        }

        return newBullet;
    }

}

public class Bullet : MonoBehaviour {

    public Weapon weapon;

    // When Bullet collides with something, rather than Destroying it, we return it to the
pool
    public void ReturnToPool () {
        // Add Bullet to the pool
        weapon.availableBullets.Add(this);

        // Disable the Bullet's GameObject so it's hidden from view
        gameObject.SetActive(false);
    }

}
```

Objekt-Pooling online lesen: https://riptutorial.com/de/unity3d/topic/2276/objekt-pooling

Kapitel 22: Optimierung

Bemerkungen

1. Deaktivieren Sie nach Möglichkeit Skripts für Objekte, wenn diese nicht benötigt werden. Wenn Sie beispielsweise ein Skript für ein feindliches Objekt haben, das der Spieler sucht und auf den Spieler schießt, sollten Sie dieses Skript deaktivieren, wenn der Feind beispielsweise zu weit vom Spieler entfernt ist.

Examples

Schnelle und effiziente Prüfungen

Vermeiden Sie unnötige Operationen und Methodenaufrufe, wo immer Sie können, insbesondere bei einer Methode, die oftmals pro Sekunde aufgerufen wird, z. B. `Update` .

Entfernungs- / Entfernungsprüfungen

Verwenden Sie `sqrMagnitude` anstelle des `magnitude` wenn Sie Entfernungen vergleichen. Dies vermeidet unnötige `sqrt` . Beachten Sie, dass bei der Verwendung von `sqrMagnitude` auch die rechte Seite ein Quadrat sein muss.

```
if ((target.position - transform.position).sqrMagnitude < minDistance * minDistance))
```

Bounds Checks

Objektschnittstellen können grob geprüft werden, indem geprüft wird, ob sich deren Grenzen zwischen `Collider` / `Renderer` schneiden. Die `Bounds` Struktur verfügt auch über eine praktische `Intersects` Methode, mit deren Hilfe festgestellt werden kann, ob sich zwei Grenzen schneiden.

`Bounds` helfen uns auch, eine `Bounds.SqrDistance` Annäherung an den *tatsächlichen* Abstand zwischen Objekten zu erhalten (siehe `Bounds.SqrDistance`).

Vorsichtsmaßnahmen

Die Begrenzungsprüfung funktioniert für konvexe Objekte sehr gut, aber Begrenzungsprüfungen bei konkaven Objekten können abhängig von der Form des Objekts zu viel höheren Ungenauigkeiten führen.

Die Verwendung von `Mesh.bounds` wird nicht empfohlen, da sie lokale `Mesh.bounds` zurückgibt. Verwenden `MeshRenderer.bounds` stattdessen `MeshRenderer.bounds` .

Verwendungszweck

Wenn Sie einen lang andauernden Vorgang ausführen, der auf der nicht-Thread-sicheren Unity-API beruht , können Sie Coroutines verwenden , um ihn auf mehrere Frames aufzuteilen und Ihre Anwendung reaktionsfähig zu halten.

Coroutines helfen auch, teure Aktionen in jedem n-ten Frame auszuführen , anstatt diese Aktion in jedem Frame auszuführen .

Aufteilen langlaufender Routinen auf mehrere Frames

Coroutines helfen dabei, lange laufende Vorgänge auf mehrere Frames zu verteilen, um die Framerate Ihrer Anwendung zu erhalten.

Routinen, die prozedural malen oder Terrain erzeugen oder Lärm erzeugen, sind Beispiele, für die eine Coroutine-Behandlung erforderlich ist.

```
for (int y = 0; y < heightmap.Height; y++)
{
    for (int x = 0; x < heightmap.Width; x++)
    {
        // Generate pixel at (x, y)
        // Assign pixel at (x, y)

        // Process only 32768 pixels each frame
        if ((y * heightmap.Height + x) % 32 * 1024) == 0)
            yield return null; // Wait for next frame
    }
}
```

> Der obige Code ist ein leicht verständliches Beispiel. In Produktionscode ist es besser , die pro Pixel Prüfung zu vermeiden , die überprüft , wenn sie Nutzen `yield return` wird (vielleicht tun es alle 2-3 Zeilen) und vorab berechnen `for` Schleifenlänge im Voraus.

Kostspielige Aktionen seltener durchführen

Mit Coroutinen können Sie seltener teure Aktionen ausführen, sodass die Leistung nicht so groß ist, wie dies bei jedem Frame der Fall wäre.

Nehmen Sie das folgende Beispiel direkt aus dem Handbuch :

```
private void ProximityCheck()
```

```
{
    for (int i = 0; i < enemies.Length; i++)
    {
        if (Vector3.Distance(transform.position, enemies[i].transform.position) <
dangerDistance)
                return true;
    }
    return false;
}

private IEnumerator ProximityCheckCoroutine()
{
    while(true)
    {
        ProximityCheck();
        yield return new WaitForSeconds(.1f);
    }
}
```

Durch die Verwendung der CullingGroup-API können Annäherungstests noch weiter optimiert werden.

Häufige Fehler

Ein häufiger Fehler, den Entwickler begehen, ist der Zugriff auf Ergebnisse oder Nebenwirkungen von Coroutinen *außerhalb* der Coroutine. Coroutines geben die Kontrolle an den Aufrufer zurück, sobald eine `yield return` Anweisung gefunden wird und das Ergebnis oder der Nebeneffekt noch nicht ausgeführt wird. Zur Umgehung Probleme , bei denen Sie das Ergebnis / Nebenwirkung außerhalb des Koroutine verwenden *müssen,* überprüfen Sie diese Antwort .

Zeichenketten

Man könnte argumentieren, dass es in Unity größere Ressourcenfresser gibt als die demütige Saite, aber dies ist einer der einfacheren Aspekte, die frühzeitig behoben werden können.

String-Operationen bauen Müll auf

Die meisten String-Vorgänge erzeugen winzige Mengen an Müll, aber wenn diese Vorgänge im Verlauf einer einzelnen Aktualisierung mehrmals aufgerufen werden, stapeln sie sich. Im Laufe der Zeit wird die automatische Speicherbereinigung ausgelöst, was zu sichtbaren CPU-Spitzen führen kann.

Zwischenspeichern Sie Ihre Stringoperationen

Betrachten Sie das folgende Beispiel.

```
string[] StringKeys = new string[] {
    "Key0",
```

```
    "Key1",
    "Key2"
};

void Update()
{
    for (var i = 0; i < 3; i++)
    {
        // Cached, no garbage generated
        Debug.Log(StringKeys[i]);
    }

    for (var i = 0; i < 3; i++)
    {
        // Not cached, garbage every cycle
        Debug.Log("Key" + i);
    }

    // The most memory-efficient way is to not create a cache at all and use literals or
constants.
    // However, it is not necessarily the most readable or beautiful way.
    Debug.Log("Key0");
    Debug.Log("Key1");
    Debug.Log("Key2");
}
```

Es kann dumm und überflüssig wirken, aber wenn Sie mit Shadern arbeiten, können Sie auf Situationen wie diese stoßen. Das Zwischenspeichern der Schlüssel macht einen Unterschied.

Bitte beachten Sie, dass Zeichenketten und *Konstanten* erzeugen keinen Müll, da sie statisch in den Programmstapel Raum injiziert werden. Wenn Sie zur Laufzeit Strings *generieren* und *garantiert* jedes Mal **dieselben** Strings wie im obigen Beispiel generieren, ist das Zwischenspeichern definitiv hilfreich.

Für andere Fälle, in denen der erzeugte String nicht jedes Mal gleich ist, gibt es keine andere Alternative zum Generieren dieser Strings. Daher ist die Gedächtnisspitze, bei der jedes Mal manuell Strings erzeugt werden, normalerweise vernachlässigbar, sofern nicht Zehntausende von Strings gleichzeitig erzeugt werden.

Die meisten Stringoperationen sind Debug-Nachrichten

Stringoperationen für Debug-Nachrichten ausführen, z. `Debug.Log("Object Name: " + obj.name)` ist in Ordnung und kann während der Entwicklung nicht vermieden werden. Es ist jedoch wichtig sicherzustellen, dass irrelevante Debug-Nachrichten nicht im freigegebenen Produkt landen.

Eine Möglichkeit besteht darin, das Conditional-Attribut in Ihren Debug-Aufrufen zu verwenden. Dadurch werden nicht nur die Methodenaufrufe entfernt, sondern auch alle String-Vorgänge, die darin ausgeführt werden.

```
using UnityEngine;
using System.Collections;

public class ConditionalDebugExample: MonoBehaviour
```

```
{
    IEnumerator Start()
    {
        while(true)
        {
            // This message will pop up in Editor but not in builds
            Log("Elapsed: " + Time.timeSinceLevelLoad);
            yield return new WaitForSeconds(1f);
        }
    }

    [System.Diagnostics.Conditional("UNITY_EDITOR")]
    void Log(string Message)
    {
        Debug.Log(Message);
    }
}
```

Dies ist ein vereinfachtes Beispiel. Vielleicht möchten Sie etwas Zeit investieren, um eine vollständigere Protokollierungsroutine zu entwerfen.

Stringvergleich

Dies ist eine geringfügige Optimierung, die jedoch erwähnenswert ist. Der Vergleich von Strings ist etwas komplizierter als man denkt. Das System versucht standardmäßig, kulturelle Unterschiede zu berücksichtigen. Sie können stattdessen einen einfachen binären Vergleich verwenden, der schneller abschneidet.

```
// Faster string comparison
if (strA.Equals(strB, System.StringComparison.Ordinal)) {...}
// Compared to
if (strA == strB) {...}

// Less overhead
if (!string.IsNullOrEmpty(strA)) {...}
// Compared to
if (strA == "") {...}

// Faster lookups
Dictionary<string, int> myDic = new Dictionary<string, int>(System.StringComparer.Ordinal);
// Compared to
Dictionary<string, int> myDictionary = new Dictionary<string, int>();
```

Referenzen zwischenspeichern

Zwischenspeichern von Referenzen, um die teuren Aufrufe insbesondere in der Update-Funktion zu vermeiden. Dies kann durch Cachen dieser Referenzen beim Start (falls verfügbar oder falls verfügbar) und Überprüfen auf null / bool flat erfolgen, damit die Referenz nicht erneut abgerufen wird.

Beispiele:

Komponentenverweise zwischenspeichern

Veränderung

```
void Update()
{
    var renderer = GetComponent<Renderer>();
    renderer.material.SetColor("_Color", Color.green);
}
```

zu

```
private Renderer myRenderer;
void Start()
{
    myRenderer = GetComponent<Renderer>();
}

void Update()
{
    myRenderer.material.SetColor("_Color", Color.green);
}
```

Objektreferenzen zwischenspeichern

Veränderung

```
void Update()
{
    var enemy = GameObject.Find("enemy");
    enemy.transform.LookAt(new Vector3(0,0,0));
}
```

zu

```
private Transform enemy;

void Start()
{
    this.enemy = GameObject.Find("enemy").transform;
}

void Update()
{
    enemy.LookAt(new Vector3(0, 0, 0));
}
```

Zusätzlich können Sie kostspielige Anrufe wie Mathfax-Anrufe zwischenspeichern.

Vermeiden Sie den Aufruf von Methoden, die Strings verwenden

Vermeiden Sie den Aufruf von Methoden, die Strings verwenden, die Methoden akzeptieren. Bei diesem Ansatz werden Reflexionen verwendet, die das Spiel verlangsamen können, insbesondere wenn sie in der Aktualisierungsfunktion verwendet werden.

Beispiele:

```
    //Avoid StartCoroutine with method name
    this.StartCoroutine("SampleCoroutine");

    //Instead use the method directly
    this.StartCoroutine(this.SampleCoroutine());

    //Avoid send message
    var enemy = GameObject.Find("enemy");
    enemy.SendMessage("Die");

    //Instead make direct call
    var enemy = GameObject.Find("enemy") as Enemy;
    enemy.Die();
```

Vermeiden Sie leere Einheitsmethoden

Vermeiden Sie leere Einheitsmethoden. Abgesehen von einem schlechten Programmierstil ist das Runtime-Scripting mit einem geringen Aufwand verbunden. In vielen Fällen kann sich dies auf die Leistung auswirken.

```
void Update
{
}

void FixedUpdate
{
}
```

Optimierung online lesen: https://riptutorial.com/de/unity3d/topic/3433/optimierung

Kapitel 23: Physik

Examples

Starre Körper

Überblick

Die Rigidbody-Komponente verleiht einem GameObject eine *physische Präsenz* in der Szene, indem es auf Kräfte reagieren kann. Sie können Kräfte direkt auf das GameObject anwenden oder auf äußere Kräfte wie die Schwerkraft oder einen anderen starren Körper reagieren lassen.

Eine Rigidbody-Komponente hinzufügen

Sie können einen Rigidbody hinzufügen, indem Sie auf **Component> Physics> Rigidbody klicken**

Verschieben eines Rigidbody-Objekts

Wenn Sie einen Rigidbody auf ein GameObject anwenden, sollten Sie Kräfte oder Drehmoment verwenden, um es zu bewegen, anstatt seine Transformation zu verändern. Verwenden `AddForce()` Methoden `AddForce()` oder `AddTorque()` :

```
// Add a force to the order of myForce in the forward direction of the Transform.
GetComponent<Rigidbody>().AddForce(transform.forward * myForce);

// Add torque about the Y axis to the order of myTurn.
GetComponent<Rigidbody>().AddTorque(transform.up * torque * myTurn);
```

Masse

Sie können die Masse eines Rigidbody-GameObjects ändern, um die Reaktion auf andere Rigidbodies und Kräfte zu beeinflussen. Eine höhere Masse bedeutet, dass das GameObject mehr Einfluss auf andere physikbasierte GameObjects hat und eine größere Kraft erfordert, um sich selbst zu bewegen. Objekte mit unterschiedlicher Masse fallen in derselben Geschwindigkeit ab, wenn sie die gleichen Widerstandswerte haben. Masse im Code ändern:

```
GetComponent<Rigidbody>().mass = 1000;
```

Ziehen

Je höher der Ziehwert, desto langsamer wird ein Objekt beim Bewegen. Betrachten Sie es als eine gegnerische Kraft. Ziehen Sie den Code in den Code:

```
GetComponent<Rigidbody>().drag = 10;
```

---

isKinematic

Wenn Sie einen Rigidbody als **Kinematisch** markieren, kann er nicht von anderen Truppen beeinflusst werden, er kann jedoch andere GameObjects beeinflussen. Um den Code zu ändern:

```
GetComponent<Rigidbody>().isKinematic = true;
```

Einschränkungen

Es ist auch möglich, Einschränkungen für jede Achse hinzuzufügen, um die Position oder Drehung des Rigidbody im lokalen Raum einzufrieren. Der Standard ist `RigidbodyConstraints.None` wie hier gezeigt:

Ein Beispiel für Einschränkungen im Code:

```
// Freeze rotation on all axes.
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezeRotation

// Freeze position on all axes.
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezePosition

// Freeze rotation and motion an all axes.
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezeAll
```

Sie können den bitweisen OR-Operator `|` mehrere Einschränkungen wie folgt kombinieren:

```
// Allow rotation on X and Y axes and motion on Y and Z axes.
GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezePositionZ |
    RigidbodyConstraints.FreezeRotationX;
```

Kollisionen

Wenn Sie möchten, dass ein GameObject mit einem Rigidbody auf Kollisionen reagiert, müssen Sie auch einen Collider hinzufügen. Arten von Collider sind:

- Box-Collider
- Kugel-Collider
- Capsule Collider
- Radkollidator
- Mesh Collider

Wenn Sie mehr als einen Collider auf ein GameObject anwenden, nennen wir es einen zusammengesetzten Collider.

Sie können einen Collider in einen **Trigger** `OnTriggerEnter()` , um die `OnTriggerEnter()` , `OnTriggerStay()` und `OnTriggerExit()` verwenden. Ein Trigger-Collider reagiert nicht physisch auf Kollisionen, andere GameObjects passieren sie einfach. Sie sind nützlich, um zu erkennen, wann sich ein anderes GameObject in einem bestimmten Bereich befindet oder nicht. Wenn Sie zum Beispiel ein Objekt sammeln, möchten wir es vielleicht einfach durchlaufen lassen, aber erkennen, wenn dies passiert.

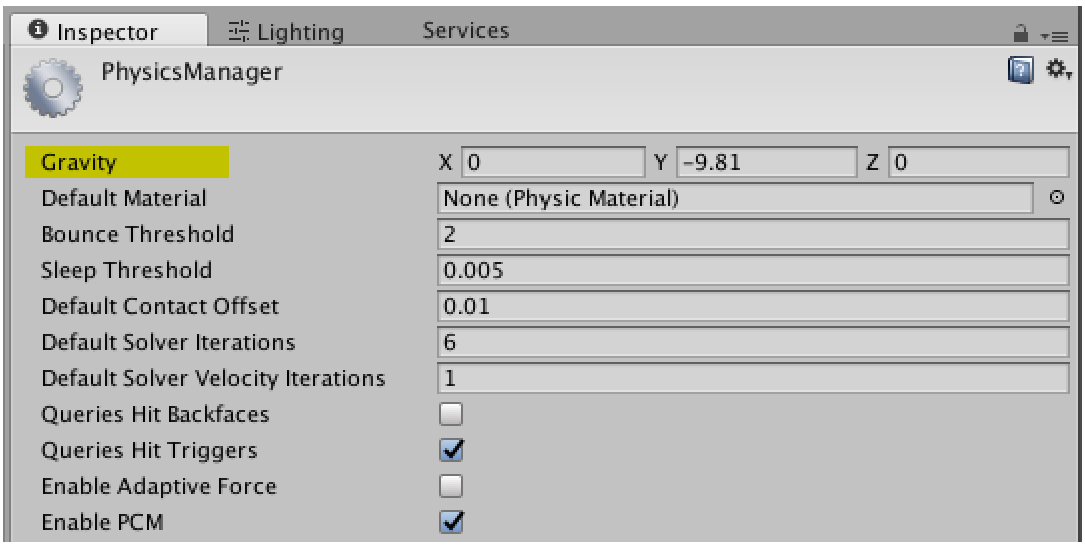
Schwerkraft im starren Körper

Die `useGravity` Eigenschaft eines `RigidBody` , ob die Schwerkraft sie beeinflusst oder nicht. Wenn der `RigidBody` auf `false` `RigidBody` ist, verhält sich der `RigidBody` wie im Weltraum (ohne dass eine konstante Kraft in irgendeiner Richtung auf ihn `RigidBody` wird).

```
GetComponent<RigidBody>().useGravity = false;
```

`RigidBody` ist sehr nützlich in Situationen, in denen Sie alle anderen Eigenschaften von `RigidBody` außer der durch die Schwerkraft kontrollierten Bewegung benötigen.

Wenn diese Funktion aktiviert, die `RigidBody` wird von einer Gravitationskraft, die im Rahmen betroffen sein `Physics Settings` - `Physics Settings` :



Die Schwerkraft wird in Welteinheiten pro Sekunde im Quadrat definiert und hier als

dreidimensionaler Vektor eingegeben: Das bedeutet, dass bei allen Einstellungen im Beispielbild alle `RigidBodies` mit der Eigenschaft `useGravity` auf `True` eine Kraft von 9,81 `useGravity` pro Sekunde erfahren *pro Sekunde* in Abwärtsrichtung (als negative Y-Werte im Unity-Koordinatensystem zeigen nach unten).

Physik online lesen: https://riptutorial.com/de/unity3d/topic/3680/physik

Kapitel 24: Prefabs

Syntax

- public static Object PrefabUtility.InstantiatePrefab (Objektziel);
- public static Object AssetDatabase.LoadAssetAtPath (Zeichenfolge assetPath, Typ type);
- public static Object Object.Instantiate (Objektvorlage);
- public static Object Resources.Load (Zeichenfolgenpfad);

Examples

Einführung

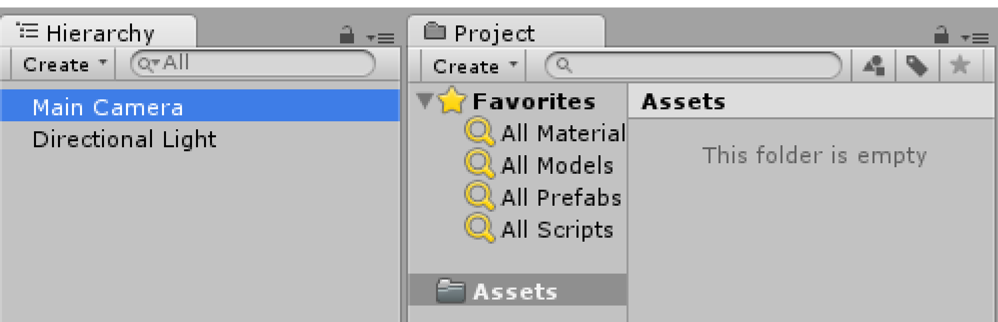
Prefabs sind ein Asset-Typ, der die Speicherung eines kompletten GameObjects mit seinen Komponenten, Eigenschaften, angehängten Komponenten und serialisierten Eigenschaftswerten ermöglicht. Es gibt viele Szenarien, in denen dies nützlich ist, darunter:

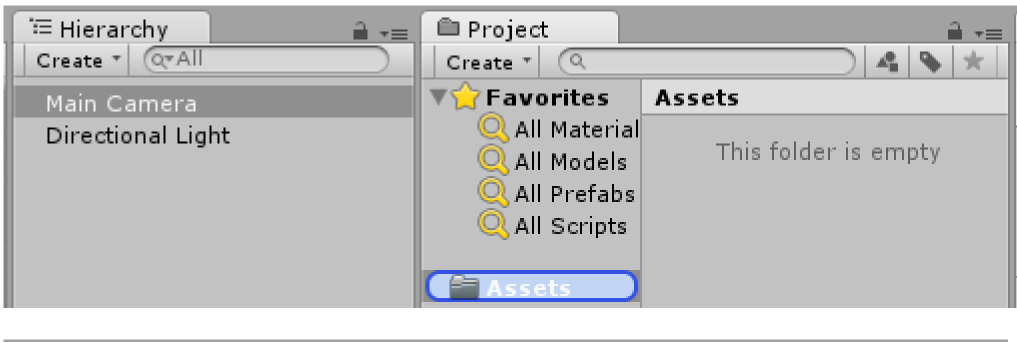
- Objekte in einer Szene duplizieren
- Gemeinsames Objekt für mehrere Szenen freigeben
- Einmaliges Ändern eines Fertigteils und Änderung der Änderungen auf mehrere Objekte / Szenen
- Erstellen Sie doppelte Objekte mit geringfügigen Änderungen, während die allgemeinen Elemente von einem Basis-Prefab bearbeitet werden können
- GameObjects zur Laufzeit instanziieren

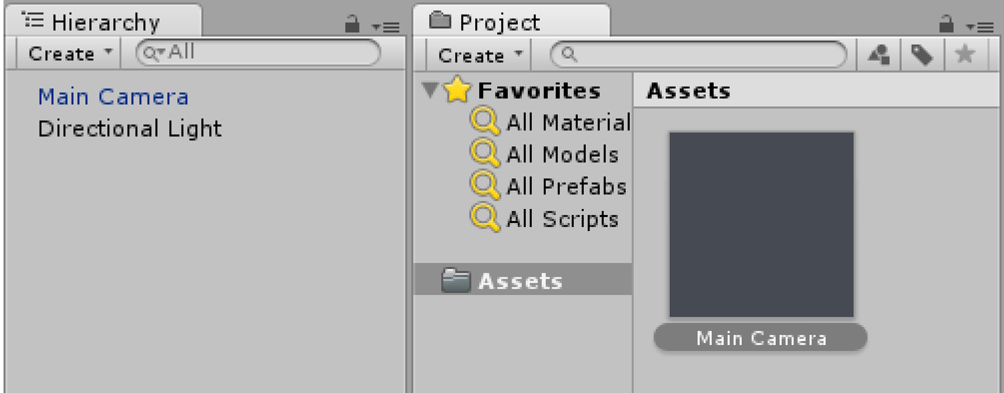
In Unity gibt es eine Art Faustregel: "Alles sollte Prefabs sein". Während dies wahrscheinlich übertrieben ist, fördert dies die Wiederverwendung von Code und das Erstellen von GameObjects auf wiederverwendbare Weise, was sowohl speichereffizient als auch gut gestaltet ist.

Prefabs erstellen

Um ein Prefab zu erstellen, ziehen Sie ein Spielobjekt aus der Szenenhierarchie in den Ordner oder Unterordner **Assets** :





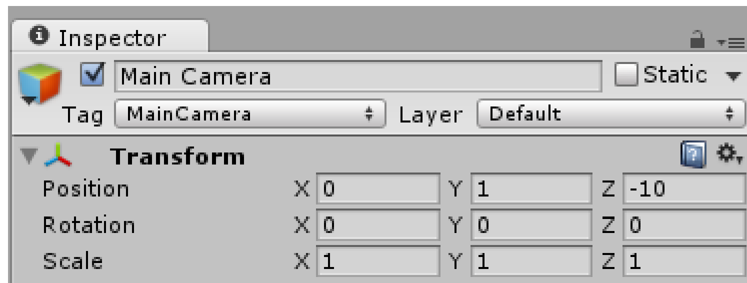


Der Spielobjektname wird blau und zeigt damit an, dass er **mit einem Prefab verbunden ist** . Dieses Objekt ist jetzt eine **Prefab-Instanz** , genau wie eine Objektinstanz einer Klasse.

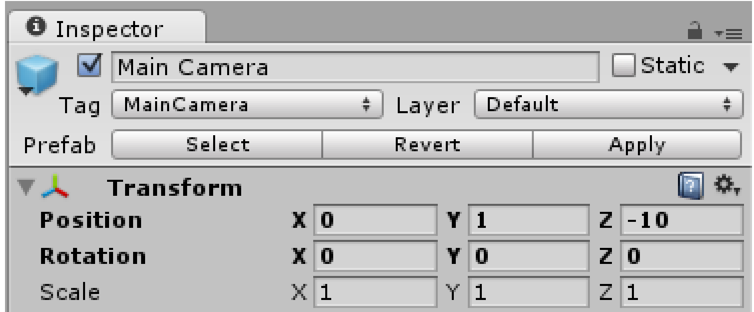
Ein Prefab kann nach der Instantiierung gelöscht werden. In diesem Fall wird der Name des zuvor verbundenen Spielobjekts rot angezeigt:

Prefab-Inspektor

Wenn Sie in der Hierarchieansicht ein Prefab auswählen, werden Sie feststellen, dass sich der Inspektor leicht von einem normalen Spielobjekt unterscheidet:



vs



Fettgedruckte Eigenschaften bedeuten, dass ihre Werte von den Fertigwerten abweichen. Sie können jede Eigenschaft eines instantiierten Prefab ändern, ohne die ursprünglichen Prefab-Werte zu beeinflussen. Wenn ein Wert in einer Prefab-Instanz geändert wird, wird er fett dargestellt, und alle nachfolgenden Änderungen desselben Werts im Prefab werden nicht in der geänderten Instanz angezeigt.

Sie können die ursprünglichen Werte wiederherstellen, indem **Sie auf die** Schaltfläche Wiederherstellen klicken. In diesem Fall werden auch Wertänderungen in der Instanz angezeigt. Um einen einzelnen Wert zurückzusetzen, können Sie außerdem mit der rechten **Maustaste auf den Wert** klicken und die **Option Wert auf Voreinstellung zurücksetzen** drücken. Um eine Komponente **rückgängig zu machen** , klicken Sie mit der rechten Maustaste darauf und drücken **Sie Auf Voreinstellung wiederherstellen** .

Wenn **Sie auf die** Schaltfläche Übernehmen klicken, werden die vorgefertigten Eigenschaftswerte mit den aktuellen Eigenschaften der Spielobjekteigenschaft überschrieben. Es gibt keine Schaltfläche "Rückgängig" oder Bestätigungsdialogfeld, behandeln Sie diese Schaltfläche daher sorgfältig.

Die Schaltfläche **Auswählen** markiert das verbundene Prefab in der Ordnerstruktur des Projekts.

Prefabs instanziieren

Es gibt zwei Möglichkeiten, Prefabs zu instanziieren: während der **Entwurfszeit** oder der **Laufzeit** .

Design-Zeitinstanziierung

Das Instantiieren von Prefabs zur Entwurfszeit ist nützlich, um mehrere Instanzen desselben Objekts visuell zu platzieren (z. B. *Platzieren von Bäumen beim Entwerfen einer Spielebene*).

- Um ein Prefab visuell zu instanziieren, ziehen Sie es aus der Projektansicht in die Szenenhierarchie.

- Wenn Sie eine schreiben Editor Erweiterung , können Sie instanziiert auch eine Fertig programmatisch Aufruf `PrefabUtility.InstantiatePrefab()` Methode:

```
GameObject gameObject =
(GameObject)PrefabUtility.InstantiatePrefab(AssetDatabase.LoadAssetAtPath("Assets/MainCamera.pref
typeof(GameObject)));
```

Laufzeit-Instantiierung

Das Instantiieren von Prefabs zur Laufzeit ist nützlich, um Instanzen eines Objekts gemäß einer bestimmten Logik zu erstellen (z. B. *alle 5 Sekunden einen Feind auftauchen*).

Um ein Prefab zu instanziieren, benötigen Sie einen Verweis auf das Prefab-Objekt. Dazu können Sie ein `public GameObject` Feld in Ihrem `MonoBehaviour` Skript verwenden (und dessen Wert mithilfe des Inspektors im Unity-Editor `MonoBehaviour`):

```
public class SomeScript : MonoBehaviour {
   public GameObject prefab;
}
```

Oder indem Sie das Prefab im Ressourcenordner ablegen und `Resources.Load` :

```
GameObject prefab = Resources.Load("Assets/Resources/MainCamera");
```

Sobald Sie einen Verweis auf das Fertigobjekt haben, können Sie es mit der Funktion `Instantiate` beliebiger Stelle in Ihrem Code instanziieren (z. B. *innerhalb einer Schleife, um mehrere Objekte zu erstellen*):

```
GameObject gameObject = Instantiate<GameObject>(prefab, new Vector3(0,0,0),
Quaternion.identity);
```

> Hinweis: Der *vorgefertigte* Begriff ist zur Laufzeit nicht vorhanden.

Verschachtelte Prefabs

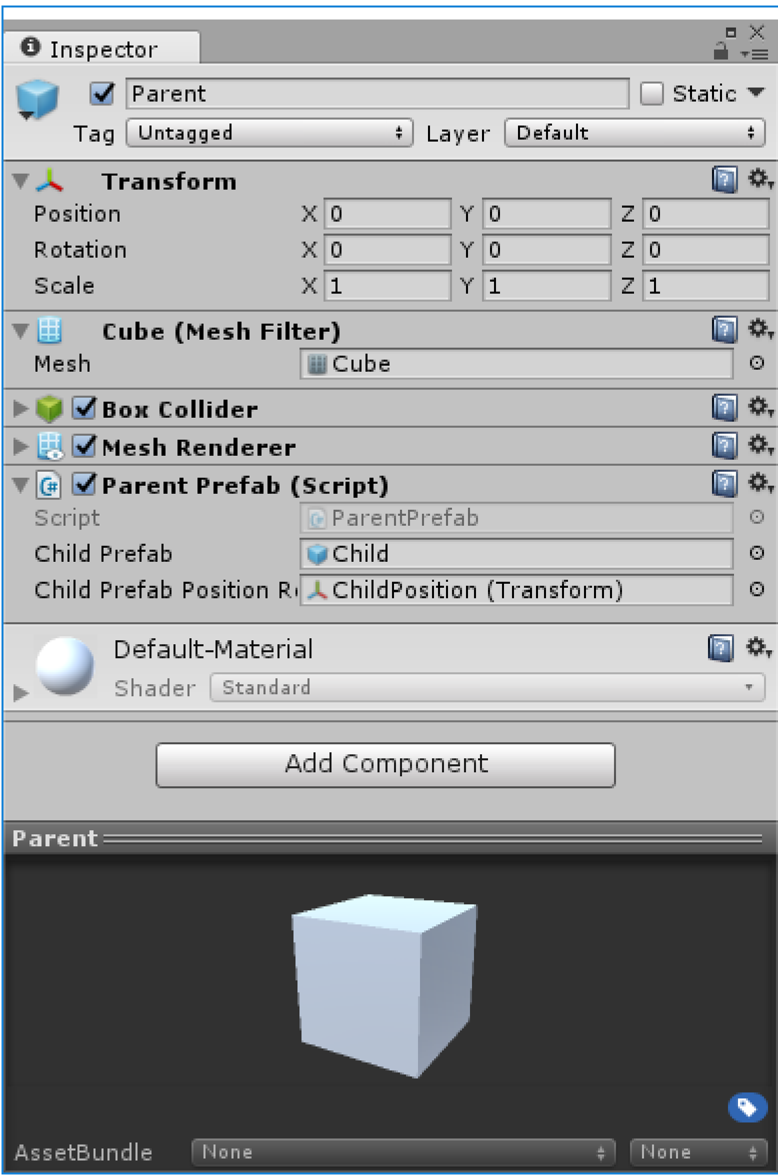
Geschachtelte Prefabs sind derzeit in Unity nicht verfügbar. Sie können ein Prefab in ein anderes

ziehen und anwenden, aber Änderungen an dem untergeordneten Prefab werden nicht auf das verschachtelte angewendet.

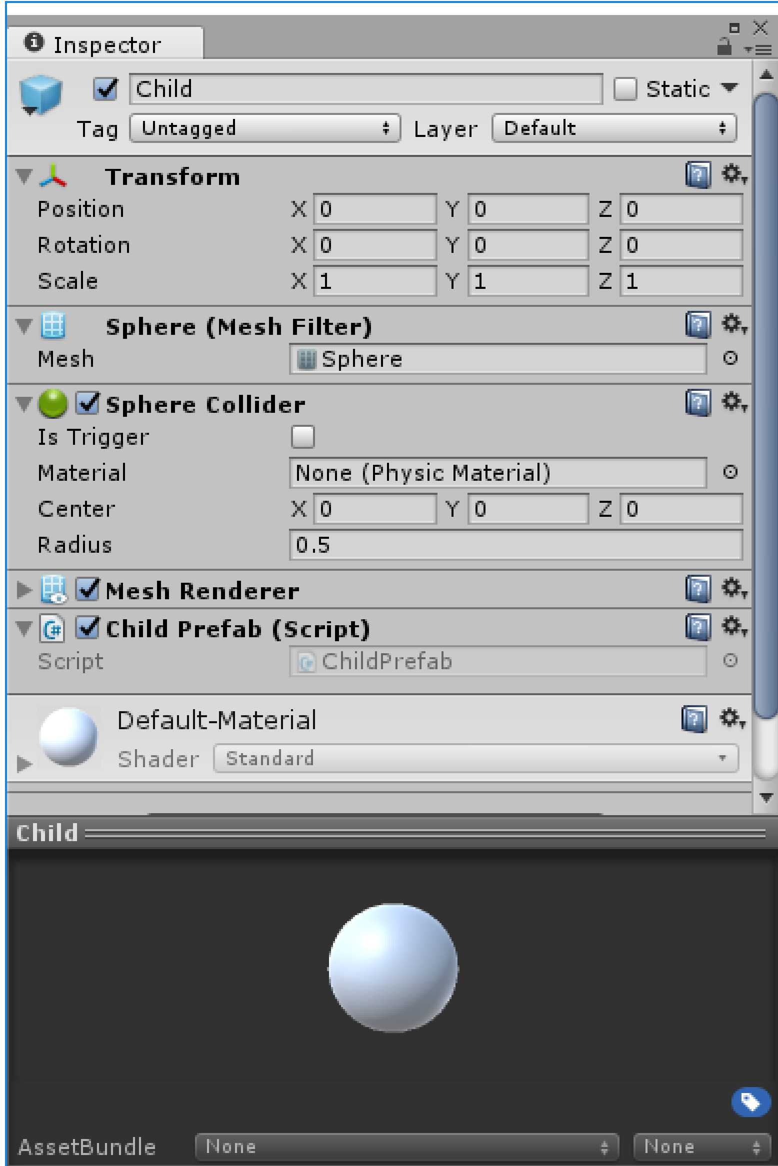
Es gibt jedoch eine einfache Problemumgehung: **Sie müssen dem übergeordneten Prefab ein einfaches Skript hinzufügen, das ein untergeordnetes Skript instanziiert.**

```
using UnityEngine;

public class ParentPrefab : MonoBehaviour {

    [SerializeField] GameObject childPrefab;
    [SerializeField] Transform childPrefabPositionReference;

    // Use this for initialization
    void Start () {
        print("Hello, I'm a parent prefab!");
        Instantiate(
            childPrefab,
            childPrefabPositionReference.position,
            childPrefabPositionReference.rotation,
            gameObject.transform
        );
    }
}
```

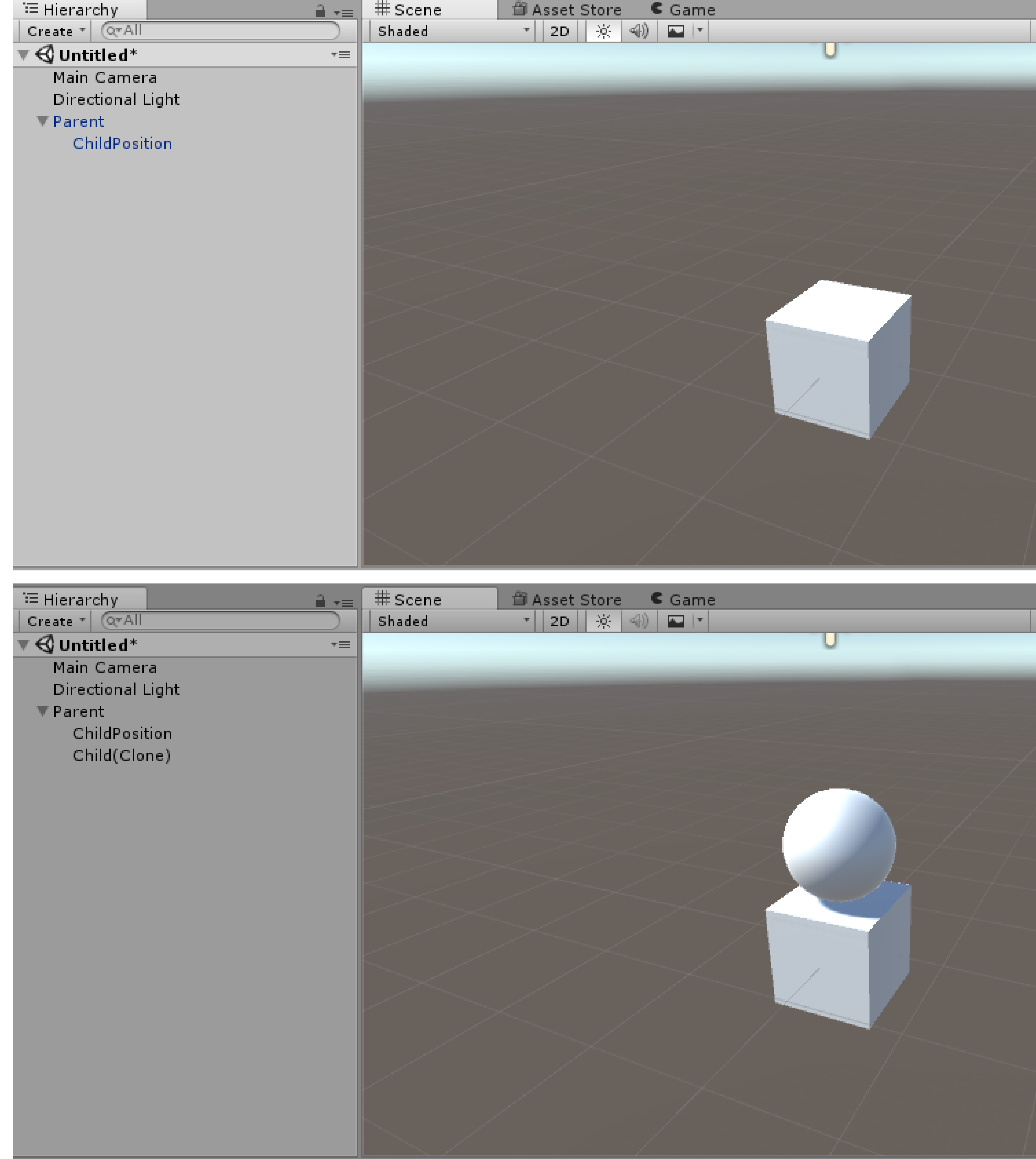
Elternteil vorgefertigt:



Kind Fertigteil:



Szene vor und nach dem Start:



Prefabs online lesen: https://riptutorial.com/de/unity3d/topic/2133/prefabs

Kapitel 25: Quaternionen

Syntax

- Quaternion.LookRotation (Vector3 weiterleiten [, Vector3 aufwärts]);
- Quaternion.AngleAxis (Float-Winkel, Vector3-AchseOfRotation);
- float angleBetween = Quaternion.Angle (Quaternion-Drehung1, Quaternion-Drehung2);

Examples

Einführung zu Quaternion vs Euler

Eulerwinkel sind "Gradwinkel" wie 90, 180, 45, 30 Grad. Quaternionen unterscheiden sich von Eulerwinkeln dadurch, dass sie einen Punkt auf einer Einheitskugel darstellen (der Radius beträgt 1 Einheit). Sie können sich diese Kugel als eine 3D-Version des Einheitskreises vorstellen, den Sie in der Trigonometrie lernen. Quaternionen unterscheiden sich von Eulerwinkeln dadurch, dass sie imaginäre Zahlen verwenden, um eine 3D-Drehung zu definieren.

Auch wenn dies kompliziert klingt (und ist es zweifellos auch), verfügt Unity über hervorragende integrierte Funktionen, mit denen Sie zwischen Eulerwinkeln und Viertelbereichen wechseln können, sowie Funktionen zum Ändern von Quaternionen, ohne dass Sie etwas über die dahinterstehende Mathematik wissen.

Konvertierung zwischen Euler und Quaternion

```
// Create a quaternion that represents 30 degrees about X, 10 degrees about Y
Quaternion rotation = Quaternion.Euler(30, 10, 0);

// Using a Vector
Vector3 EulerRotation = new Vector3(30, 10, 0);
Quaternion rotation = Quaternion.Euler(EulerRotation);

// Convert a transfroms Quaternion angles to Euler angles
Quaternion quaternionAngles = transform.rotation;
Vector3 eulerAngles = quaternionAngles.eulerAngles;
```

Warum eine Quaternion verwenden?

Quaternionen lösen ein Problem, das als Kardanverriegelung bekannt ist. Dies geschieht, wenn die primäre Rotationsachse mit der Tertiärachse kollinear wird. Hier ist ein visuelles Beispiel @ 2:09

Quaternion Look Rotation

`Quaternion.LookRotation(Vector3 forward [, Vector3 up])` erstellt eine Quaternion-Rotation, die den Vorwärtsvektor nach unten zeigt und die Y-Achse mit dem Up-Vektor ausgerichtet ist. Wenn der Aufwärtsvektor nicht angegeben ist, wird Vector3.up verwendet.

Drehen Sie dieses Spielobjekt, um ein Zielobjekt anzusehen

```
// Find a game object in the scene named Target
public Transform target = GameObject.Find("Target").GetComponent<Transform>();

// We subtract our position from the target position to create a
// Vector that points from our position to the target position
// If we reverse the order, our rotation would be 180 degrees off.
Vector3 lookVector = target.position - transform.position;
Quaternion rotation = Quaternion.LookRotation(lookVector);
transform.rotation = rotation;
```

Quaternionen online lesen: https://riptutorial.com/de/unity3d/topic/1782/quaternionen

Kapitel 26: Raycast

Parameter

Parameter	Einzelheiten
Ursprung	Der Startpunkt des Strahls in Weltkoordinaten
Richtung	Die Richtung des Strahls
maximale Entfernung	Die maximale Entfernung, die der Strahl auf Kollisionen überprüfen soll
Ebenenmaske	Eine Ebenenmaske, die zum selektiven Ignorieren von Kollisionen verwendet wird, wenn ein Strahl geworfen wird.
queryTriggerInteraction	Gibt an, wo diese Abfrage auf Auslöser treffen soll.

Examples

Physik-Raycast

Diese Funktion wirft einen Strahl von Punkt `origin` in Richtung `direction` der Länge `maxDistance` gegen alle Collider in der Szene.

Die Funktion erfolgt in der `origin` `direction` `maxDistance` und berechnen , ob es eine collider vor dem Gameobject ist.

```
Physics.Raycast(origin, direction, maxDistance);
```

Diese Funktion druckt beispielsweise `Hello World` auf die Konsole, wenn innerhalb von 10 Einheiten des `GameObject` angehängt ist:

```
using UnityEngine;

public class TestPhysicsRaycast: MonoBehaviour
{
    void FixedUpdate()
    {
        Vector3 fwd = transform.TransformDirection(Vector3.forward);

        if (Physics.Raycast(transform.position, fwd, 10))
            print("Hello World");
    }
}
```

Physics2D Raycast2D

Sie können Raycasts verwenden, um zu prüfen, ob ein AI laufen kann, ohne vom Rand eines Levels zu fallen.

```
using UnityEngine;

public class Physics2dRaycast: MonoBehaviour
    {
        public LayerMask LineOfSightMask;
        void FixedUpdate()
        {
            RaycastHit2D hit = Physics2D.Raycast(raycastRightPart, Vector2.down, 0.6f *
heightCharacter, LineOfSightMask);
            if(hit.collider != null)
            {
                //code when the ai can walk
            }
            else
            {
                //code when the ai cannot walk
            }
    }
}
```

In diesem Beispiel stimmt die Richtung. Die Variable raycastRightPart ist der rechte Teil des Charakters, sodass der Raycast im rechten Teil des Charakters stattfindet. Die Entfernung beträgt das 0,6 fache des Charakters, so dass der Raycast keinen Treffer erzielt, wenn er auf den Boden trifft, der viel niedriger ist als der Boden, auf dem er gerade steht. Stellen Sie sicher, dass die Layermask nur auf Masse eingestellt ist, andernfalls werden auch andere Arten von Objekten erkannt.

> RaycastHit2D selbst ist eine Struktur und keine Klasse, sodass der Treffer nicht null sein kann. Das bedeutet, dass Sie nach dem Collider einer RaycastHit2D-Variablen suchen müssen.

Raycast-Anrufe kapseln

Wenn Sie Ihre Skripts direkt mit `Raycast` aufrufen, kann dies zu Problemen führen, wenn Sie die Kollisionsmatrizen in der Zukunft ändern müssen, da Sie jedes `LayerMask` Feld `LayerMask` , um die Änderungen zu `LayerMask` . Je nach Größe Ihres Projekts kann dies zu einem großen Unternehmen werden.

Das Einkapseln von `Raycast` Anrufen kann Ihr Leben auf der `Raycast` Linie erleichtern.

Wenn man es von einem SoC- Prinzip aus betrachtet, sollte ein Spielobjekt LayerMasks wirklich nicht kennen oder interessieren. Es braucht nur eine Methode, um die Umgebung zu scannen. Ob das Raycast-Ergebnis dieses oder jenes zurückgibt, sollte für das Spielobjekt keine Rolle spielen. Es sollte nur auf die erhaltenen Informationen einwirken und keine Annahmen über die Umgebung treffen, in der es sich befindet.

Um dies zu erreichen, können Sie den LayerMask-Wert in ScriptableObject- Instanzen verschieben und diese als eine Form von Raycast-Services verwenden, die Sie in Ihre Skripts einfügen.

```
// RaycastService.cs
using UnityEngine;

[CreateAssetMenu(menuName = "StackOverflow")]
public class RaycastService : ScriptableObject
{
    [SerializeField]
    LayerMask layerMask;

    public RaycastHit2D Raycast2D(Vector2 origin, Vector2 direction, float distance)
    {
        return Physics2D.Raycast(origin, direction, distance, layerMask.value);
    }

    // Add more methods as needed

}
```

```
// MyScript.cs
using UnityEngine;

public class MyScript : MonoBehaviour
{
    [SerializeField]
    RaycastService raycastService;

    void FixedUpdate()
    {
        RaycastHit2D hit = raycastService.Raycast2D(Vector2.zero, Vector2.down, 1f);
    }
}
```

Auf diese Weise können Sie eine Reihe von Raycast-Diensten erstellen, die alle unterschiedliche LayerMask-Kombinationen für unterschiedliche Situationen aufweisen. Sie könnten eine haben, die nur Bodenkollisionen trifft, und eine andere, die Bodenkollisionen und Einbahnplattformen trifft.

Wenn Sie einmal drastische Änderungen an Ihren LayerMask-Setups vornehmen müssen, müssen Sie nur diese RaycastService-Assets aktualisieren.

Lesen Sie weiter

- Inversion der Kontrolle
- Abhängigkeitsspritze

Raycast online lesen: https://riptutorial.com/de/unity3d/topic/2826/raycast

Kapitel 27: Ressourcen

Examples

Einführung

Mit der Ressourcenklasse können Assets, die nicht Teil der Szene sind, dynamisch geladen werden. Dies ist sehr nützlich, wenn Sie On-Demand-Assets verwenden müssen, z. B. mehrsprachige Audios, Texte usw.

Assets müssen in einem Ordner namens **Resources** abgelegt werden. Es ist möglich, mehrere Ressourcenordner über die Projekthierarchie zu verteilen. `Resources` prüft alle Ressourcenordner, die Sie möglicherweise haben.

Jedes in Resources platzierte Asset wird in den Build aufgenommen, auch wenn es nicht in Ihrem Code angegeben ist. Fügen Sie daher Ressourcen nicht wahllos in Resources ein.

```
//Example of how to load language specific audio from Resources

[RequireComponent(typeof(AudioSource))]
public class loadIntroAudio : MonoBehaviour {
    void Start () {
        string language = Application.systemLanguage.ToString();
        AudioClip ac = Resources.Load(language + "/intro") as AudioClip; //loading intro.mp3
specific for user's language (note the file file extension should not be used)
        if (ac==null)
        {
            ac = Resources.Load("English/intro") as AudioClip; //fallback to the english
version for any unsupported language
        }
        transform.GetComponent<AudioSource>().clip = ac;
        transform.GetComponent<AudioSource>().Play();
    }
}
```

Ressourcen 101

Einführung

Unity verfügt über einige speziell benannte Ordner, die eine Vielzahl von Anwendungen ermöglichen. Einer dieser Ordner heißt "Ressourcen".

Der Ordner "Resources" ist eine von nur zwei Möglichkeiten, Assets zur Laufzeit in Unity zu laden (die andere ist AssetBundles (Unity Docs).

Der Ordner "Resources" kann sich an beliebiger Stelle innerhalb des Assets-Ordners befinden, und Sie können mehrere Ordner mit dem Namen Resources haben. Der Inhalt aller 'Resources'-Ordner wird während der Kompilierzeit zusammengeführt.

Die primäre Methode zum Laden eines Assets aus einem Resources-Ordner besteht in der Verwendung der Resources.Load- Funktion. Diese Funktion benötigt einen String-Parameter, mit dem Sie den Pfad der Datei **relativ** zum Ordner Resources angeben können. Beachten Sie, dass Sie beim Laden eines Assets KEINE Dateierweiterungen angeben müssen

```
public class ResourcesSample : MonoBehaviour {

    void Start () {
        //The following line will load a TextAsset named 'foobar' which was previously place
under 'Assets/Resources/Stackoverflow/foobar.txt'
        //Note the absence of the '.txt' extension! This is important!

        var text = Resources.Load<TextAsset>("Stackoverflow/foobar").text;
        Debug.Log(string.Format("The text file had this in it :: {0}", text));
    }
}
```

Objekte, die aus mehreren Objekten bestehen, können auch aus Ressourcen geladen werden. Beispiele hierfür sind 3D-Modelle mit eingebetteten Texturen oder mehrere Sprites.

```
//This example will load a multiple sprite texture from Resources named "A_Multiple_Sprite"
var sprites = Resources.LoadAll("A_Multiple_Sprite") as Sprite[];
```

Alles zusammen setzen

Hier ist eine meiner Helfer-Klassen, mit denen ich alle Sounds für jedes Spiel laden kann. Sie können dies an jedes GameObject in einer Szene anhängen und die angegebenen Audiodateien werden aus dem Ordner 'Resources / Sounds' geladen

```
public class SoundManager : MonoBehaviour {

    void Start () {

        //An array of all sounds you want to load
        var filesToLoad = new string[] { "Foo", "Bar" };

        //Loop over the array, attach an Audio source for each sound clip and assign the
        //clip property.
        foreach(var file in filesToLoad) {
            var soundClip = Resources.Load<AudioClip>("Sounds/" + file);
            var audioSource = gameObject.AddComponent<AudioSource>();
            audioSource.clip = soundClip;
        }
    }
}
```

Abschließende Anmerkungen

1. Unity ist intelligent, wenn es darum geht, Assets in Ihren Build einzubinden. Alle Assets, die nicht serialisiert sind (dh in einer Szene verwendet werden, die in einem Build enthalten ist) werden von einem Build ausgeschlossen. Dies gilt jedoch NICHT für alle Ressourcen im Ordner Ressourcen. Gehen Sie daher beim Hinzufügen von Assets zu diesem Ordner nicht über Bord

2. Assets, die mit Resources.Load oder Resources.LoadAll geladen werden, können zukünftig mit Resources.UnloadUnusedAssets oder Resources.UnloadAsset entladen werden

Ressourcen online lesen: https://riptutorial.com/de/unity3d/topic/4070/ressourcen

Kapitel 28: Schichten

Examples

Layer-Nutzung

Unity-Layer ähneln Tags darin, dass sie zum Definieren von Objekten verwendet werden können, mit denen interagiert werden soll oder sich auf bestimmte Weise verhalten soll. Layer werden jedoch hauptsächlich mit Funktionen in der Klasse `Physics` : Unity-Dokumentation - Physik

Layer werden durch eine ganze Zahl dargestellt und können auf diese Weise an die Funktionen übergeben werden:

```
using UnityEngine;
class LayerExample {

    public int layer;

    void Start()
    {
        Collider[] colliders = Physics.OverlapSphere(transform.position, 5f, layer);
    }
}
```

Die Verwendung einer Ebene auf diese Weise schließt nur Collider ein, deren GameObjects die in den Berechnungen angegebene Ebene angegeben hat. Dies macht weitere Logik einfacher und verbessert die Leistung.

LayerMask-Struktur

Die `LayerMask` Struktur ist eine Schnittstelle, die fast genauso funktioniert wie das Übergeben einer Ganzzahl an die betreffende Funktion. Der größte Vorteil besteht jedoch darin, dass der Benutzer die betreffende Ebene aus einem Dropdown-Menü im Inspektor auswählen kann.

```
using UnityEngine;
class LayerMaskExample{

    public LayerMask mask;
    public Vector3 direction;

    void Start()
    {
        if(Physics.Raycast(transform.position, direction, 35f, mask))
        {
            Debug.Log("Raycast hit");
        }
    {
}
```

Es verfügt auch über mehrere statische Funktionen, mit denen Ebenennamen in Indizes oder

Indizes in Ebenennamen umgewandelt werden können.

```
using UnityEngine;
class NameToLayerExample{

    void Start()
    {
        int layerindex = LayerMask.NameToLayer("Obstacle");
    {
}
```

Um die Ebenenüberprüfung zu vereinfachen, definieren Sie die folgende Erweiterungsmethode.

```
public static bool IsInLayerMask(this GameObject @object, LayerMask layerMask)
{
    bool result = (1 << @object.layer & layerMask) == 0;

    return result;
}
```

Mit dieser Methode können Sie überprüfen, ob sich ein Spielobjekt in einer Layermask befindet (im Editor ausgewählt) oder nicht.

Schichten online lesen: https://riptutorial.com/de/unity3d/topic/4762/schichten

Kapitel 29: ScriptableObject

Bemerkungen

ScriptableObjects mit AssetBundles

Achten Sie beim Hinzufügen von Prefabs zu AssetBundles, wenn diese Verweise auf ScriptableObjects enthalten. Da ScriptableObjects im Wesentlichen Assets sind, erstellt Unity Duplikate davon, bevor diese zu AssetBundles hinzugefügt werden. Dies kann zu unerwünschtem Verhalten während der Laufzeit führen.

Wenn Sie ein solches GameObject von einem AssetBundle laden, müssen Sie möglicherweise die ScriptableObject-Assets erneut in die geladenen Skripts einfügen, um die gebündelten zu ersetzen. Siehe Abhängigkeitseinspritzung

Examples

Einführung

ScriptableObjects sind serialisierte Objekte, die nicht an Szenen oder Spielobjekte gebunden sind, wie dies bei MonoBehaviours der Fall ist. Eine Möglichkeit: Es handelt sich um Daten und Methoden, die an Asset-Dateien in Ihrem Projekt gebunden sind. Diese ScriptableObject-Assets können an MonoBehaviours oder andere ScriptableObjects übergeben werden, wo auf ihre öffentlichen Methoden zugegriffen werden kann.

Aufgrund ihrer Beschaffenheit als serialisierte Assets eignen sie sich hervorragend für Manager-Klassen und Datenquellen.

ScriptableObject-Assets erstellen

Nachfolgend finden Sie eine einfache ScriptableObject-Implementierung.

```
using UnityEngine;

[CreateAssetMenu(menuName = "StackOverflow/Examples/MyScriptableObject")]
public class MyScriptableObject : ScriptableObject
{
    [SerializeField]
    int mySerializedNumber;

    int helloWorldCount = 0;

    public void HelloWorld()
    {
        helloWorldCount++;
        Debug.LogFormat("Hello! My number is {0}.", mySerializedNumber);
```

```
        Debug.LogFormat("I have been called {0} times.", helloWorldCount);
    }
}
```

Indem `CreateAssetMenu` Attribut `CreateAssetMenu` zur Klasse hinzufügt, listet es das Untermenü **Assets / Create auf** . In diesem Fall ist es unter **Assets / Create / StackOverflow / Beispiele** .

Einmal erstellt, können ScriptableObject-Instanzen über den Inspector an andere Scripts und ScriptableObjects übergeben werden.

```
using UnityEngine;

public class SampleScript : MonoBehaviour {

    [SerializeField]
    MyScriptableObject myScriptableObject;

    void OnEnable()
    {
        myScriptableObject.HelloWorld();
    }
}
```

Erstellen Sie ScriptableObject-Instanzen durch Code

Sie erstellen neue ScriptableObject-Instanzen über `ScriptableObject.CreateInstance<T>()`

```
T obj = ScriptableObject.CreateInstance<T>();
```

Wo `T` `ScriptableObject` .

> Erstellen Sie keine ScriptableObjects, indem Sie ihre Konstruktoren aufrufen. `new ScriptableObject()` .

Das Erstellen von ScriptableObjects mithilfe von Code zur Laufzeit ist selten erforderlich, da der Hauptzweck die Datenserialisierung ist. Sie können an dieser Stelle genauso gut Standardklassen verwenden. Dies ist häufiger, wenn Sie Editorerweiterungen mit Skripts erstellen.

ScriptableObjects werden auch im PlayMode im Editor serialisiert

Beim Zugriff auf serialisierte Felder in einer ScriptableObject-Instanz ist besondere Vorsicht geboten.

Wenn ein Feld mit `SerializeField` als `public` gekennzeichnet oder serialisiert ist, ist die Änderung des Werts dauerhaft. Sie werden beim Beenden des Playmode nicht wie bei MonoBehaviours zurückgesetzt. Dies kann manchmal nützlich sein, aber es kann auch ein Durcheinander verursachen.

Aus diesem Grund ist es am besten, serialisierte Felder schreibgeschützt zu machen und öffentliche Felder vollständig zu vermeiden.

```
public class MyScriptableObject : ScriptableObject
{
    [SerializeField]
    int mySerializedValue;

    public int MySerializedValue
    {
        get { return mySerializedValue; }
    }
}
```

Wenn Sie öffentliche Werte in einem ScriptableObject speichern möchten, die zwischen Wiedergabesitzungen zurückgesetzt werden, sollten Sie das folgende Muster in Betracht ziehen.

```
public class MyScriptableObject : ScriptableObject
{
    // Private fields are not serialized and will reset to default on reset
    private int mySerializedValue;

    public int MySerializedValue
    {
        get { return mySerializedValue; }
        set { mySerializedValue = value; }
    }
}
```

Vorhandene ScriptableObjects zur Laufzeit finden

Um *aktive* ScriptableObjects zur Laufzeit zu finden, können Sie `Resources.FindObjectsOfTypeAll()` .

```
T[] instances = Resources.FindObjectsOfTypeAll<T>();
```

Dabei ist `T` der Typ der von Ihnen durchsuchten ScriptableObject-Instanz. *Aktiv* bedeutet, dass es zuvor in irgendeiner Form in den Speicher geladen wurde.

Diese Methode ist sehr langsam. Denken Sie daran, den Rückgabewert zwischenzuspeichern und den Aufruf nicht zu häufig zu machen. Ein direkter Verweis auf ScriptableObjects in Ihren Skripts sollte Ihre bevorzugte Option sein.

> *Tipp:* Sie können Ihre eigenen Instanzsammlungen verwalten, um schneller nachschlagen zu können. `OnEnable()` Sie Ihre ScriptableObjects sich während `OnEnable()` in einer gemeinsam genutzten Sammlung `OnEnable()` .

ScriptableObject online lesen: https://riptutorial.com/de/unity3d/topic/3434/scriptableobject

Kapitel 30: Singletons in der Einheit

Bemerkungen

Zwar gibt es Denkschulen, die überzeugende Argumente dafür liefern , warum der uneingeschränkte Einsatz von Singletons eine schlechte Idee ist, z. B. Singleton auf gameprogrammingpatterns.com . Es gibt jedoch Situationen , in denen Sie ein GameObject in Unity über mehrere Szenen hinweg beibehalten möchten (z. B. für nahtlose Hintergrundmusik). wobei sichergestellt ist, dass nicht mehr als eine Instanz existieren kann; ein perfekter Anwendungsfall für ein Singleton.

Durch das Hinzufügen dieses Skripts zu einem GameObject bleibt es nach der Instanziierung (z. B. durch Einfügen an einer beliebigen Stelle in einer Szene) in allen Szenen aktiv, und es wird nur eine Instanz vorhanden sein.

ScriptableObject (UnityDoc) -Instanzen bieten für einige Anwendungsfälle eine gültige Alternative zu Singletons. Obwohl sie die Einzelinstanzregel nicht implizit erzwingen, behalten sie ihren Status zwischen den Szenen bei und spielen gut mit dem Unity-Serialisierungsprozess. Sie fördern auch die Inversion der Kontrolle, da Abhängigkeiten durch den Editor eingefügt werden .

```
// MyAudioManager.cs
using UnityEngine;

[CreateAssetMenu] // Remember to create the instance in editor
public class MyAudioManager : ScriptableObject {
    public void PlaySound() {}
}
```

```
// MyGameObject.cs
using UnityEngine;

public class MyGameObject : MonoBehaviour
{
    [SerializeField]
    MyAudioManager audioManager; //Insert through Inspector

    void OnEnable()
    {
        audioManager.PlaySound();
    }
}
```

Lesen Sie weiter

- Singleton-Implementierung in C #

Examples

Implementierung mit RuntimeInitializeOnLoadMethodAttribute

Seit **Unity 5.2.5 ist** es möglich, RuntimeInitializeOnLoadMethodAttribute zum Ausführen der Initialisierungslogik unter Umgehung der MonoBehaviour-Reihenfolge der Ausführung zu verwenden . Es bietet eine Möglichkeit, eine sauberere und robustere Implementierung zu erstellen:

```
using UnityEngine;

sealed class GameDirector : MonoBehaviour
{
    // Because of using RuntimeInitializeOnLoadMethod attribute to find/create and
    // initialize the instance, this property is accessible and
    // usable even in Awake() methods.
    public static GameDirector Instance
    {
        get; private set;
    }

    // Thanks to the attribute, this method is executed before any other MonoBehaviour
    // logic in the game.
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
    static void OnRuntimeMethodLoad()
    {
        var instance = FindObjectOfType<GameDirector>();

        if (instance == null)
            instance = new GameObject("Game Director").AddComponent<GameDirector>();

        DontDestroyOnLoad(instance);

        Instance = instance;
    }

    // This Awake() will be called immediately after AddComponent() execution
    // in the OnRuntimeMethodLoad(). In other words, before any other MonoBehaviour's
    // in the scene will begin to initialize.
    private void Awake()
    {
        // Initialize non-MonoBehaviour logic, etc.
        Debug.Log("GameDirector.Awake()", this);
    }
}
```

Die resultierende Reihenfolge der Ausführung:

1. `GameDirector.OnRuntimeMethodLoad()` gestartet ...
2. `GameDirector.Awake()`
3. `GameDirector.OnRuntimeMethodLoad()` abgeschlossen.
4. `OtherMonoBehaviour1.Awake()`
5. `OtherMonoBehaviour2.Awake()` usw.

Ein einfaches Singleton MonoBehaviour in Unity C

In diesem Beispiel wird eine private statische Instanz der Klasse zu Beginn deklariert.

Der Wert eines statischen Feldes wird von Instanzen gemeinsam genutzt. Wenn eine neue Instanz dieser Klasse erstellt wird, findet das `if` eine Referenz auf das erste Singleton-Objekt, wodurch die neue Instanz (oder ihr Spielobjekt) zerstört wird.

```
using UnityEngine;

public class SingletonExample : MonoBehaviour {

    private static SingletonExample _instance;

    void Awake(){

        if (_instance == null){

            _instance = this;
            DontDestroyOnLoad(this.gameObject);

            //Rest of your Awake code

        } else {
            Destroy(this);
        }
    }

    //Rest of your class code

}
```

Advanced Unity Singleton

In diesem Beispiel werden mehrere Varianten der im Internet gefundenen MonoBehaviour-Singletons zu einer einzigen kombiniert, und Sie können ihr Verhalten in Abhängigkeit von globalen statischen Feldern ändern.

Dieses Beispiel wurde mit Unity 5 getestet. Um dieses Singleton zu verwenden, müssen Sie es nur wie folgt erweitern: `public class MySingleton : Singleton<MySingleton> {}` . Sie können auch außer Kraft setzen müssen `AwakeSingleton` es üblich , anstatt zu verwenden `Awake` . Ändern Sie für weitere Einstellungen die Standardwerte für statische Felder wie unten beschrieben.

---

1. Diese Implementierung verwendet das DisallowMultipleComponent- Attribut, um eine Instanz pro GameObject beizubehalten.
2. Diese Klasse ist abstrakt und kann nur erweitert werden. Es enthält auch eine virtuelle Methode `AwakeSingleton` , die überschrieben werden muss, anstatt normales `Awake` implementieren.
3. Diese Implementierung ist threadsicher.
4. Dieses Singleton ist optimiert. Durch die Verwendung eines `instantiated` Flags anstelle einer Instanznullprüfung vermeiden wir den Overhead, der mit der Implementierung des Operators `==` von Unity `==` . (Lesen Sie mehr)
5. Diese Implementierung erlaubt keine Aufrufe an die Singleton-Instanz, wenn sie von Unity

zerstört wird.

6. Dieses Singleton bietet folgende Optionen:

- `FindInactive` : `FindInactive` ob nach Instanzen von Komponenten desselben Typs gesucht werden soll, die mit dem inaktiven GameObject verbunden sind.
- `Persist` : Ob Komponente zwischen Szenen am Leben erhalten `Persist` .
- `DestroyOthers` : ob andere Komponenten desselben Typs zerstört werden sollen und nur eine behalten.
- `Lazy` : Ob Singleton-Instanz "on the fly" (in `Awake`) oder nur "on demand" (wenn Getter aufgerufen wird) gesetzt wird.

```
using UnityEngine;

[DisallowMultipleComponent]
public abstract class Singleton<T> : MonoBehaviour where T : Singleton<T>
{
    private static volatile T instance;
    // thread safety
    private static object _lock = new object();
    public static bool FindInactive = true;
    // Whether or not this object should persist when loading new scenes. Should be set in
Init().
    public static bool Persist;
    // Whether or not destory other singleton instances if any. Should be set in Init().
    public static bool DestroyOthers = true;
    // instead of heavy comparision (instance != null)
    // http://blogs.unity3d.com/2014/05/16/custom-operator-should-we-keep-it/
    private static bool instantiated;

    private static bool applicationIsQuitting;

    public static bool Lazy;

    public static T Instance
    {
        get
        {
            if (applicationIsQuitting)
            {
                Debug.LogWarningFormat("[Singleton] Instance '{0}' already destroyed on
application quit. Won't create again - returning null.", typeof(T));
                return null;
            }
            lock (_lock)
            {
                if (!instantiated)
                {
                    Object[] objects;
                    if (FindInactive) { objects = Resources.FindObjectsOfTypeAll(typeof(T)); }
                    else { objects = FindObjectsOfType(typeof(T)); }
                    if (objects == null || objects.Length < 1)
                    {
                        GameObject singleton = new GameObject();
                        singleton.name = string.Format("{0} [Singleton]", typeof(T));
                        Instance = singleton.AddComponent<T>();
                        Debug.LogWarningFormat("[Singleton] An Instance of '{0}' is needed in
the scene, so '{1}' was created{2}", typeof(T), singleton.name, Persist ? " with
DontDestoryOnLoad." : ".");
```

```
                    }
                    else if (objects.Length >= 1)
                    {
                        Instance = objects[0] as T;
                        if (objects.Length > 1)
                        {
                            Debug.LogWarningFormat("[Singleton] {0} instances of '{1}'!",
objects.Length, typeof(T));
                            if (DestroyOthers)
                            {
                                for (int i = 1; i < objects.Length; i++)
                                {
                                    Debug.LogWarningFormat("[Singleton] Deleting extra '{0}'
instance attached to '{1}'", typeof(T), objects[i].name);
                                    Destroy(objects[i]);
                                }
                            }
                        }
                        return instance;
                    }
                }
                return instance;
            }
        }
        protected set
        {
            instance = value;
            instantiated = true;
            instance.AwakeSingleton();
            if (Persist) { DontDestroyOnLoad(instance.gameObject); }
        }
    }

    // if Lazy = false and gameObject is active this will set instance
    // unless instance was called by another Awake method
    private void Awake()
    {
        if (Lazy) { return; }
        lock (_lock)
        {
            if (!instantiated)
            {
                Instance = this as T;
            }
            else if (DestroyOthers && Instance.GetInstanceID() != GetInstanceID())
            {
                Debug.LogWarningFormat("[Singleton] Deleting extra '{0}' instance attached to
'{1}'", typeof(T), name);
                Destroy(this);
            }
        }
    }

    // this might be called for inactive singletons before Awake if FindInactive = true
    protected virtual void AwakeSingleton() {}

    protected virtual void OnDestroy()
    {
        applicationIsQuitting = true;
        instantiated = false;
    }
```

```
}
```

Singleton Implementierung durch Basisklasse

In Projekten mit mehreren Singleton-Klassen (wie dies häufig der Fall ist), kann es sauber und bequem sein, das Singleton-Verhalten in eine Basisklasse zu abstrahieren:

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

public abstract class MonoBehaviourSingleton<T> : MonoBehaviour {

    private static Dictionary<Type, object> _singletons
        = new Dictionary<Type, object>();

    public static T Instance {
        get {
            return (T)_singletons[typeof(T)];
        }
    }

    void OnEnable() {
        if (_singletons.ContainsKey(GetType())) {
            Destroy(this);
        } else {
            _singletons.Add(GetType(), this);
            DontDestroyOnLoad(this);
        }
    }
}
```

Ein MonoBehaviour kann dann das Singleton-Muster durch Erweitern von MonoBehaviourSingleton implementieren. Dieser Ansatz ermöglicht die Verwendung des Musters mit einem minimalen Footprint auf dem Singleton selbst:

```
using UnityEngine;
using System.Collections;

public class SingletonImplementation : MonoBehaviourSingleton<SingletonImplementation> {

    public string Text= "String Instance";

    // Use this for initialisation
    IEnumerator Start () {
        var demonstration = "SingletonImplementation.Start()\n" +
                            "Note that the this text logs only once and\n"
                            "only one class instance is allowed to exist.";
        Debug.Log(demonstration);
        yield return new WaitForSeconds(2f);
        var secondInstance = new GameObject();
        secondInstance.AddComponent<SingletonImplementation>();
    }

}
```

Beachten Sie, dass einer der Vorteile des Singleton-Musters darin besteht, dass auf einen Verweis auf die Instanz statisch zugegriffen werden kann:

```
// Logs: String Instance
Debug.Log(SingletonImplementation.Instance.Text);
```

Beachten Sie jedoch, dass diese Vorgehensweise minimiert werden sollte, um die Kopplung zu reduzieren. Dieser Ansatz bringt aufgrund der Verwendung von Dictionary auch einen geringen Performance-Preis mit sich. Da diese Auflistung jedoch nur eine Instanz jeder Einzelklasse enthalten kann, muss dies im Hinblick auf das DRY-Prinzip (Don't Repeat Yourself), Readability und Bequemlichkeit ist klein.

Singleton-Pattern mit Unitys Entity-Component-System

Die Kernidee besteht darin, GameObjects zur Darstellung von Singletons zu verwenden, was mehrere Vorteile bietet:

- Reduziert die Komplexität auf ein Minimum, unterstützt jedoch Konzepte wie Abhängigkeitseinspritzung
- Singletons haben als Teil des Entity-Component-Systems einen normalen Unity-Lebenszyklus
- Singletons können lokal geladen werden, wo sie regelmäßig benötigt werden (z. B. in Update-Schleifen)
- Keine statischen Felder erforderlich
- Bestehende MonoBehaviours / -Komponenten müssen nicht modifiziert werden, um sie als Singletons zu verwenden
- Einfach zurückzusetzen (einfach das Singletons GameObject zerstören), wird bei der nächsten Verwendung wieder faul geladen
- Einfach zu injizierende Mocks (initialisieren Sie es einfach mit dem Mock, bevor Sie es verwenden)
- Überprüfung und Konfiguration mit dem normalen Unity-Editor und kann bereits zur Bearbeitungszeit erfolgen (Screenshot eines Singleton im Unity-Editor abrufbar)

Test.cs (verwendet das Beispiel-Singleton):

```
using UnityEngine;
using UnityEngine.Assertions;

public class Test : MonoBehaviour {
    void Start() {
        ExampleSingleton singleton = ExampleSingleton.instance;
        Assert.IsNotNull(singleton); // automatic initialization on first usage
        Assert.AreEqual("abc", singleton.myVar1);
        singleton.myVar1 = "123";
        // multiple calls to instance() return the same object:
        Assert.AreEqual(singleton, ExampleSingleton.instance);
        Assert.AreEqual("123", ExampleSingleton.instance.myVar1);
    }
}
```

ExampleSingleton.cs (enthält ein Beispiel und die tatsächliche Singleton-Klasse):

```
using UnityEngine;
using UnityEngine.Assertions;

public class ExampleSingleton : MonoBehaviour {
    public static ExampleSingleton instance { get { return Singleton.get<ExampleSingleton>();
} }
    public string myVar1 = "abc";
    public void Start() { Assert.AreEqual(this, instance, "Singleton more than once in
scene"); }
}

/// <summary> Helper that turns any MonBehaviour or other Component into a Singleton
</summary>
public static class Singleton {
    public static T get<T>() where T : Component {
        return GetOrAddGo("Singletons").GetOrAddChild("" + typeof(T)).GetOrAddComponent<T>();
    }
    private static GameObject GetOrAddGo(string goName) {
        var go = GameObject.Find(goName);
        if (go == null) { return new GameObject(goName); }
        return go;
    }
}

public static class GameObjectExtensionMethods {
    public static GameObject GetOrAddChild(this GameObject parentGo, string childName) {
        var childGo = parentGo.transform.FindChild(childName);
        if (childGo != null) { return childGo.gameObject; } // child found, return it
        var newChild = new GameObject(childName);        // no child found, create it
        newChild.transform.SetParent(parentGo.transform, false); // add it to parent
        return newChild;
    }

    public static T GetOrAddComponent<T>(this GameObject parentGo) where T : Component {
        var comp = parentGo.GetComponent<T>();
        if (comp == null) { return parentGo.AddComponent<T>(); }
        return comp;
    }
}
```

Die beiden Erweiterungsmethoden für GameObject sind auch in anderen Situationen hilfreich. Wenn Sie sie nicht brauchen, verschieben Sie sie in die Singleton-Klasse und machen sie privat.

MonoBehaviour & ScriptableObject-basierte Singleton-Klasse

Die meisten Singleton-Beispiele verwenden MonoBehaviour als Basisklasse. Der Hauptnachteil ist, dass diese Singleton-Klasse nur zur Laufzeit lebt. Dies hat einige Nachteile:

- Sie können die Singleton-Felder nicht direkt bearbeiten, sondern nur den Code ändern.
- Keine Möglichkeit, einen Verweis auf andere Assets im Singleton zu speichern.
- Keine Möglichkeit, das Singleton als Ziel eines Unity-UI-Ereignisses festzulegen. Am Ende verwende ich das, was ich als "Proxy-Komponenten" bezeichne. Sein einziger Vorschlag besteht darin, 1-Zeilen-Methoden zu haben, die "GameManager.Instance.SomeGlobalMethod ()" nennen.

Wie in den Anmerkungen angemerkt, gibt es Implementierungen, die versuchen, dieses Problem mithilfe von ScriptableObjects als Basisklasse zu lösen, ohne jedoch die Laufzeitvorteile des MonoBehaviour zu verlieren. Diese Implementierung löst diese Probleme, indem ein ScriptableObject als Basisklasse und ein zugeordnetes MonoBehavior zur Laufzeit verwendet werden:

- Es ist ein Asset, sodass seine Eigenschaften wie jeder andere Unity-Asset im Editor aktualisiert werden können.
- Es spielt sich gut mit dem Unity-Serialisierungsprozess ab.
- Es ist möglich, anderen Assets Referenzen aus dem Editor zuzuweisen (Abhängigkeiten werden über den Editor eingefügt).
- Unity-Ereignisse können Methoden direkt am Singleton aufrufen.
- Kann von überall in der Codebase mit "SingletonClassName.Instance" aufgerufen werden
- Hat Zugriff auf Laufzeit-MonoBehaviour-Ereignisse und -Methoden wie: Update, Awake, Start, FixedUpdate, StartCoroutine usw.

```
/************************************************************
 * Better Singleton by David Darias
 * Use as you like - credit where due would be appreciated :D
 * Licence: WTFPL V2, Dec 2014
 * Tested on Unity v5.6.0 (should work on earlier versions)
 * 03/02/2017 - v1.1
 * **********************************************************/

using System;
using UnityEngine;
using SingletonScriptableObjectNamespace;

public class SingletonScriptableObject<T> :
SingletonScriptableObjectNamespace.BehaviourScriptableObject where T :
SingletonScriptableObjectNamespace.BehaviourScriptableObject
{
    //Private reference to the scriptable object
    private static T _instance;
    private static bool _instantiated;
    public static T Instance
    {
        get
        {
            if (_instantiated) return _instance;
            var singletonName = typeof(T).Name;
            //Look for the singleton on the resources folder
            var assets = Resources.LoadAll<T>("");
            if (assets.Length > 1) Debug.LogError("Found multiple " + singletonName + "s on
the resources folder. It is a Singleton ScriptableObject, there should only be one.");
            if (assets.Length == 0)
            {
                _instance = CreateInstance<T>();
                Debug.LogError("Could not find a " + singletonName + " on the resources
folder. It was created at runtime, therefore it will not be visible on the assets folder and
it will not persist.");
            }
            else _instance = assets[0];
            _instantiated = true;
            //Create a new game object to use as proxy for all the MonoBehaviour methods
            var baseObject = new GameObject(singletonName);
```

```
            //Deactivate it before adding the proxy component. This avoids the execution of
the Awake method when the the proxy component is added.
            baseObject.SetActive(false);
            //Add the proxy, set the instance as the parent and move to DontDestroyOnLoad
scene
            SingletonScriptableObjectNamespace.BehaviourProxy proxy =
baseObject.AddComponent<SingletonScriptableObjectNamespace.BehaviourProxy>();
            proxy.Parent = _instance;
            Behaviour = proxy;
            DontDestroyOnLoad(Behaviour.gameObject);
            //Activate the proxy. This will trigger the MonoBehaviourAwake.
            proxy.gameObject.SetActive(true);
            return _instance;
        }
    }
    //Use this reference to call MonoBehaviour specific methods (for example StartCoroutine)
    protected static MonoBehaviour Behaviour;
    public static void BuildSingletonInstance() {
SingletonScriptableObjectNamespace.BehaviourScriptableObject i = Instance; }
    private void OnDestroy(){ _instantiated = false; }
}

// Helper classes for the SingletonScriptableObject
namespace SingletonScriptableObjectNamespace
{
    #if UNITY_EDITOR
    //Empty custom editor to have cleaner UI on the editor.
    using UnityEditor;
    [CustomEditor(typeof(BehaviourProxy))]
    public class BehaviourProxyEditor : Editor
    {
        public override void OnInspectorGUI(){}
    }

    #endif

    public class BehaviourProxy : MonoBehaviour
    {
        public IBehaviour Parent;

        public void Awake() { if (Parent != null) Parent.MonoBehaviourAwake(); }
        public void Start() { if (Parent != null) Parent.Start(); }
        public void Update() { if (Parent != null) Parent.Update(); }
        public void FixedUpdate() { if (Parent != null) Parent.FixedUpdate(); }
    }

    public interface IBehaviour
    {
        void MonoBehaviourAwake();
        void Start();
        void Update();
        void FixedUpdate();
    }

    public class BehaviourScriptableObject : ScriptableObject, IBehaviour
    {
        public void Awake() { ScriptableObjectAwake(); }
        public virtual void ScriptableObjectAwake() { }
        public virtual void MonoBehaviourAwake() { }
        public virtual void Start() { }
        public virtual void Update() { }
```

```
        public virtual void FixedUpdate() { }
    }
}
```

Hier gibt es eine Beispiel-GameManager-Singleton-Klasse, die das SingletonScriptableObject (mit vielen Kommentaren) verwendet:

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

//this attribute is optional but recommended. It will allow the creation of the singleton via
the asset menu.
//the singleton asset should be on the Resources folder.
[CreateAssetMenu(fileName = "GameManager", menuName = "Game Manager", order = 0)]
public class GameManager : SingletonScriptableObject<GameManager> {

    //any properties as usual
    public int Lives;
    public int Points;

    //optional (but recommended)
    //this method will run before the first scene is loaded. Initializing the singleton here
    //will allow it to be ready before any other GameObjects on every scene and will
    //will prevent the "initialization on first usage".
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
    public static void BeforeSceneLoad() { BuildSingletonInstance(); }

    //optional,
    //will run when the Singleton Scriptable Object is first created on the assets.
    //Usually this happens on edit mode, not runtime. (the override keyword is mandatory for
this to work)
    public override void ScriptableObjectAwake(){
        Debug.Log(GetType().Name + " created." );
    }

    //optional,
    //will run when the associated MonoBehavioir awakes. (the override keyword is mandatory
for this to work)
    public override void MonoBehaviourAwake(){
        Debug.Log(GetType().Name + " behaviour awake." );

        //A coroutine example:
        //Singleton Objects do not have coroutines.
        //if you need to use coroutines use the atached MonoBehaviour
        Behaviour.StartCoroutine(SimpleCoroutine());
    }

    //any methods as usual
    private IEnumerator SimpleCoroutine(){
        while(true){
            Debug.Log(GetType().Name + " coroutine step." );
            yield return new WaitForSeconds(3);
        }
    }

    //optional,
    //Classic runtime Update method (the override keyword is mandatory for this to work).
    public override void Update(){
```

```
    }

    //optional,
    //Classic runtime FixedUpdate method (the override keyword is mandatory for this to work).
    public override void FixedUpdate(){

    }
}

/*
*   Notes:
*   - Remember that you have to create the singleton asset on edit mode before using it. You
have to put it on the Resources folder and of course it should be only one.
*   - Like other Unity Singleton this one is accessible anywhere in your code using the
"Instance" property i.e: GameManager.Instance
*/
```

Singletons in der Einheit online lesen: https://riptutorial.com/de/unity3d/topic/2137/singletons-in-der-einheit

Kapitel 31: So verwenden Sie Asset-Pakete

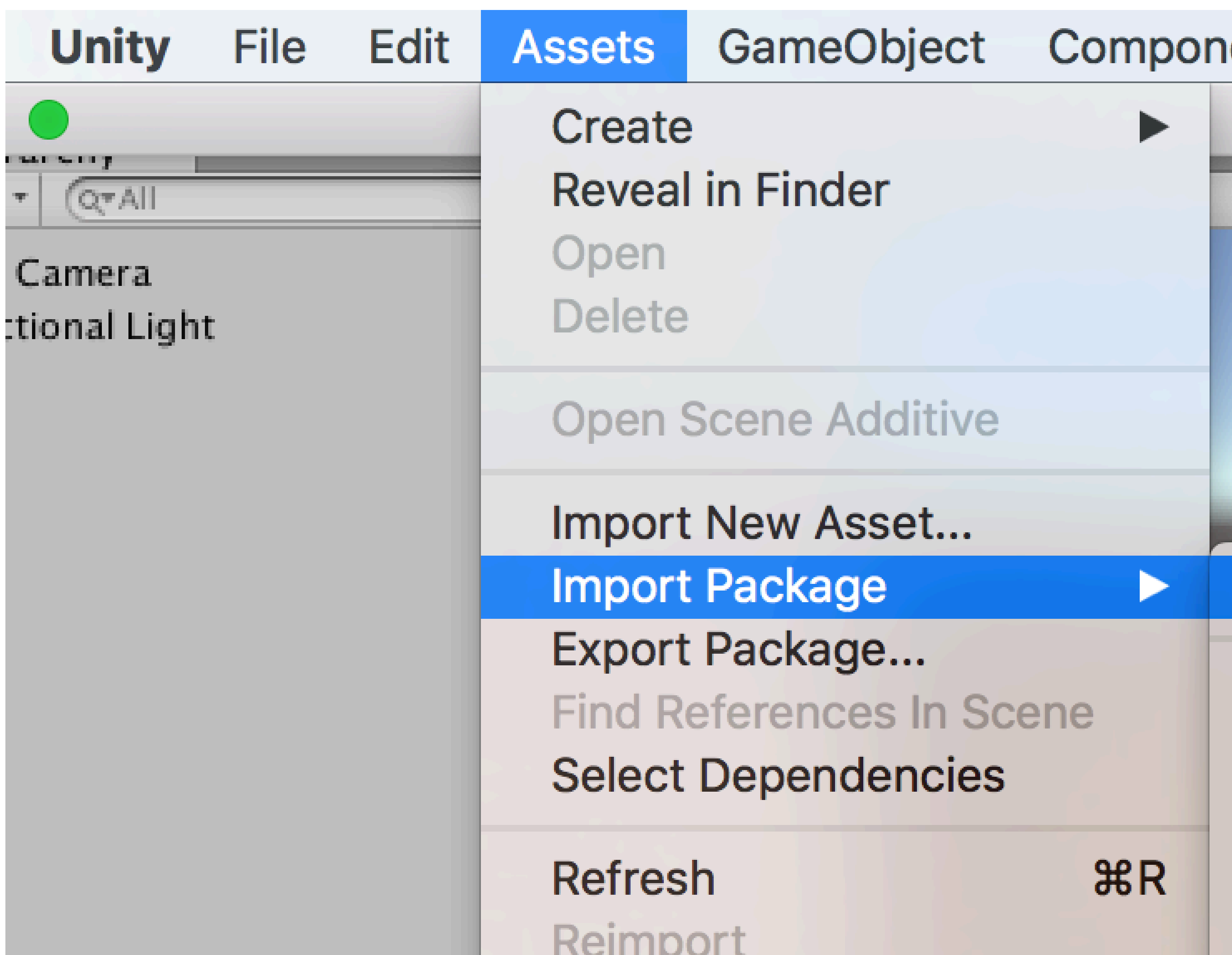
Examples

Asset-Pakete

Asset-Pakete (mit dem Dateiformat `.unitypackage`) werden häufig verwendet, um Unity-Projekte an andere Benutzer zu verteilen. Wenn Sie mit Peripheriegeräten mit eigenen SDKs (z. B. Oculus) arbeiten, werden Sie möglicherweise aufgefordert, eines dieser Pakete herunterzuladen und zu importieren.

.Unitypackage importieren

Um ein Paket zu importieren, `.unitypackage` die Unity-Menüleiste und klicken Sie auf `Assets > Import Package > Custom Package...` Navigieren Sie dann im `.unitypackage` Dateibrowser zu der `.unitypackage` Datei.



So verwenden Sie Asset-Pakete online lesen: https://riptutorial.com/de/unity3d/topic/4491/so-verwenden-sie-asset-pakete

Kapitel 32: Sofortiges grafisches Benutzeroberflächensystem (IMGUI)

Syntax

- public static void GUILayout.Label (Zeichenfolgentext, params GUILayoutOption [] Optionen)
- public static bool GUILayout.Button (Zeichenfolgentext, params GUILayoutOption [] Optionen)
- Öffentlicher statischer String GUILayout.TextArea (String-Text, params GUILayoutOption [] - Optionen)

Examples

GUILayout

Altes UI-System-Tool, das jetzt für schnelles und einfaches Prototyping oder Debugging im Spiel verwendet wird.

```
void OnGUI ()
{
    GUILayout.Label ("I'm a simple label text displayed in game.");

    if ( GUILayout.Button("CLICK ME") )
    {
        GUILayout.TextArea ("This is a \n
                            multiline comment.")
    }
}
```

Die GUILayout- Funktion arbeitet innerhalb der **OnGUI-** Funktion.

Sofortiges grafisches Benutzeroberflächensystem (IMGUI) online lesen: https://riptutorial.com/de/unity3d/topic/6947/sofortiges-grafisches-benutzeroberflachensystem--imgui-

Kapitel 33: Stichworte

Einführung

Ein Tag ist eine Zeichenfolge, die zum Markieren von `GameObject` Typen verwendet werden kann. Auf diese Weise können bestimmte `GameObject` Objekte einfacher über Code identifiziert werden.

Ein Tag kann auf ein oder mehrere Spielobjekte angewendet werden, ein Spielobjekt hat jedoch immer nur ein Tag. Standardmäßig wird das Tag *"Untagged"* verwendet, um ein `GameObject` , das nicht absichtlich markiert wurde.

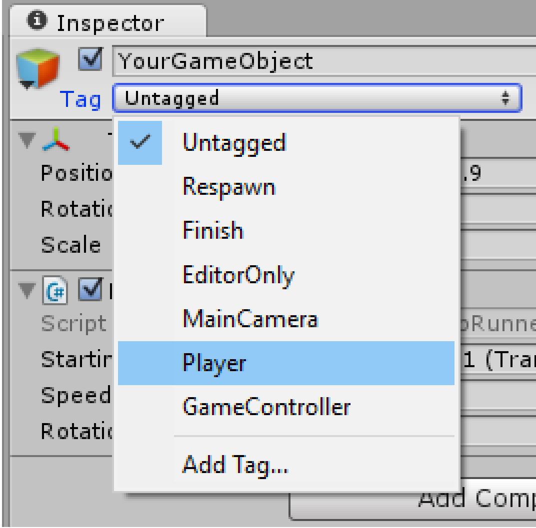
Examples

Tags erstellen und anwenden

Tags werden normalerweise über den Editor angelegt. Sie können Tags jedoch auch über ein Skript anwenden. Jedes benutzerdefinierte Tag muss über das Fenster " *Tags & Layer" erstellt werden* , bevor es auf ein Spielobjekt angewendet wird.

Tags im Editor einstellen

Wenn Sie ein oder mehrere Spielobjekte ausgewählt haben, können Sie im Inspector ein Tag auswählen. Spielobjekte tragen immer ein einzelnes Tag. Standardmäßig werden Spielobjekte als *"Nicht getagged" gekennzeichnet* . Sie können auch zum Fenster *" Tags & Layer "* wechseln, indem Sie *"Add Tag ..."* auswählen. Es ist jedoch wichtig zu beachten, dass Sie nur zum Fenster *Tags & Layers gelangen* . Alle von Ihnen erstellten Tags werden *nicht* automatisch auf das Spielobjekt angewendet.



Tags über Skript einstellen

Sie können ein Spielobjekt-Tag direkt über den Code ändern. Es ist wichtig zu beachten , dass Sie einen Tag aus der Liste der aktuellen Tags *muss;* Wenn Sie ein Tag angeben, das noch nicht erstellt wurde, führt dies zu einem Fehler.

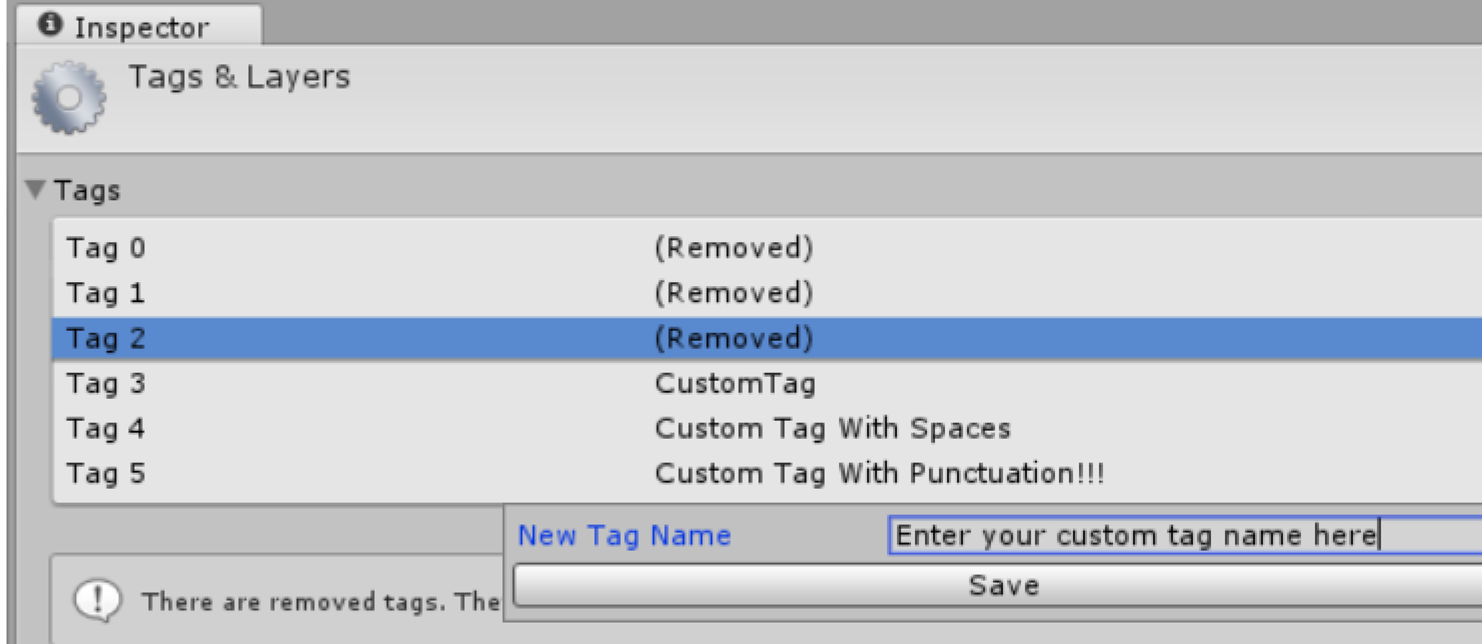
Wie in anderen Beispielen beschrieben, kann die Verwendung einer Reihe `static string` Variablen im Gegensatz zum manuellen Schreiben jedes Tags die Konsistenz und Zuverlässigkeit gewährleisten.

Das folgende Skript veranschaulicht, wie wir eine Reihe von Spielobjekt-Tags ändern können, indem `static string` Referenzen verwendet werden, um die Konsistenz zu gewährleisten. Beachten Sie die Annahme, dass jede `static string` ein Tag darstellt, das bereits im Fenster *Tags & Layers erstellt wurde* .

```
using UnityEngine;

public class Tagging : MonoBehaviour
{
    static string tagUntagged = "Untagged";
    static string tagPlayer = "Player";
    static string tagEnemy = "Enemy";

    /// <summary>Represents the player character. This game object should
    /// be linked up via the inspector.</summary>
    public GameObject player;
    /// <summary>Represents all the enemy characters. All enemies should
    /// be added to the array via the inspector.</summary>
    public GameObject[] enemy;

    void Start ()
    {
        // We ensure that the game object this script is attached to
        // is left untagged by using the default "Untagged" tag.
        gameObject.tag = tagUntagged;

        // We ensure the player has the player tag.
        player.tag = tagUntagged;

        // We loop through the enemy array to ensure they are all tagged.
        for(int i = 0; i < enemy.Length; i++)
        {
            enemy[i].tag = tagEnemy;
        }
    }
}
```

Benutzerdefinierte Tags erstellen

Unabhängig davon, ob Sie Tags über den Inspector oder über Skript festlegen, *müssen Sie Tags* vor der Verwendung über das Fenster *Tags & Layers* deklarieren. Sie können auf dieses Fenster zugreifen, indem Sie in einem Dropdown-Menü für Spielobjekte-Tags *"Tags hinzufügen"* auswählen. Alternativ finden Sie das Fenster unter **Bearbeiten> Projekteinstellungen> Tags und Ebenen** .



Wählen Sie einfach die Schaltfläche `+` , geben Sie den gewünschten Namen ein und wählen Sie `Speichern` , um ein Tag zu erstellen. Durch Auswahl der Schaltfläche `-` wird das aktuell hervorgehobene Tag entfernt. Beachten Sie, dass das Tag auf diese Weise sofort als *"(entfernt)"* angezeigt wird und beim nächsten Laden des Projekts vollständig entfernt wird.

Durch Auswählen des Zahnrads / Zahnrads oben rechts im Fenster können Sie alle benutzerdefinierten Optionen zurücksetzen. Dadurch werden alle benutzerdefinierten Tags sowie alle unter *"Sortierebenen"* und *"Ebenen"* vorhandenen benutzerdefinierten *Ebenen* sofort entfernt.

Suche nach GameObjects nach Tag:

Tags machen es besonders einfach, bestimmte Spielobjekte zu finden. Wir können nach einem einzelnen Spielobjekt oder nach mehreren Objekten suchen.

Ein einzelnes `GameObject`

Wir können die statische Funktion `GameObject.FindGameObjectWithTag(string tag)` , um nach einzelnen Spielobjekten zu suchen. Es ist wichtig anzumerken, dass auf diese Weise Spielobjekte nicht in einer bestimmten Reihenfolge abgefragt werden. Wenn Sie nach einem Tag suchen, das für mehrere Spielobjekte in der Szene verwendet wird, kann diese Funktion nicht garantieren, *welches* Spielobjekt zurückgegeben wird. Daher ist es sinnvoller, wenn wir wissen, dass nur *ein* Spielobjekt einen solchen Tag verwendet, oder wenn wir uns nicht um die genaue Instanz von `GameObject` , die zurückgegeben wird.

```
///<summary>We create a static string to allow us consistency.</summary>
string playerTag = "Player"

///<summary>We can now use the tag to reference our player GameObject.</summary>
GameObject player = GameObject.FindGameObjectWithTag(playerTag);
```

Suchen eines Arrays von `GameObject` Instanzen

Wir können die statische Funktion `GameObject.FindGameObjectsWithTag(string tag)` , um nach *allen* Spielobjekten zu suchen, die ein bestimmtes Tag verwenden. Dies ist nützlich, wenn eine Gruppe bestimmter Spielobjekte durchlaufen werden soll. Dies kann auch nützlich sein, wenn wir ein *einzelnes* Spielobjekt suchen möchten, aber möglicherweise *mehrere* Spielobjekte haben, die dasselbe Tag verwenden. Da wir nicht garantieren können, welche Instanz von `GameObject.FindGameObjectWithTag(string tag)` , müssen Sie stattdessen ein Array aller potenziellen `GameObject` Instanzen mit `GameObject.FindGameObjectsWithTag(string tag)` und das resultierende Array weiter analysieren, um die Instanz zu finden, die wir sind Auf der Suche nach.

```
///<summary>We create a static string to allow us consistency.</summary>
string enemyTag = "Enemy";

///<summary>We can now use the tag to create an array of all enemy GameObjects.</summary>
GameObject[] enemies = GameObject.FindGameObjectsWithTag(enemyTag );

// We can now freely iterate through our array of enemies
foreach(GameObject enemy in enemies)
{
    // Do something to each enemy (link up a reference, check for damage, etc.)
}
```

Tags vergleichen

Beim Vergleich zweier GameObjects nach Tags sollte Folgendes beachtet werden: Garbage Collector verursacht Mehraufwand, da jedes Mal eine Zeichenfolge erstellt wird:

```
if (go.Tag == "myTag")
{
    //Stuff
}
```

Wenn Sie diese Vergleiche in Update () und einem anderen normalen Callback (oder einer Schleife) von Unity durchführen, sollten Sie diese Methode ohne Heap-Zuweisung verwenden:

```
if (go.CompareTag("myTag")
{
    //Stuff
}
```

Außerdem ist es einfacher, Ihre Tags in einer statischen Klasse zu halten.

```
public static class Tags
{
    public const string Player = "Player";
    public const string MyCustomTag = "MyCustomTag";
}
```

Dann können Sie sicher vergleichen

```
if (go.CompareTag(Tags.MyCustomTag)
{
    //Stuff
}
```

Auf diese Weise werden Ihre Tag-Zeichenfolgen zur Kompilierzeit generiert und Sie begrenzen die Auswirkungen von Rechtschreibfehlern.

Genauso wie Tags in einer statischen Klasse gehalten werden, ist es auch möglich, sie in einer Aufzählung zu speichern:

```
public enum Tags
{
    Player, Ennemies, MyCustomTag;
}
```

und dann können Sie es mit der Methode enum `toString()` :

```
if (go.CompareTag(Tags.MyCustomTag.toString())
{
    //Stuff
}
```

Stichworte online lesen: https://riptutorial.com/de/unity3d/topic/5534/stichworte

Kapitel 34: Unity Profiler

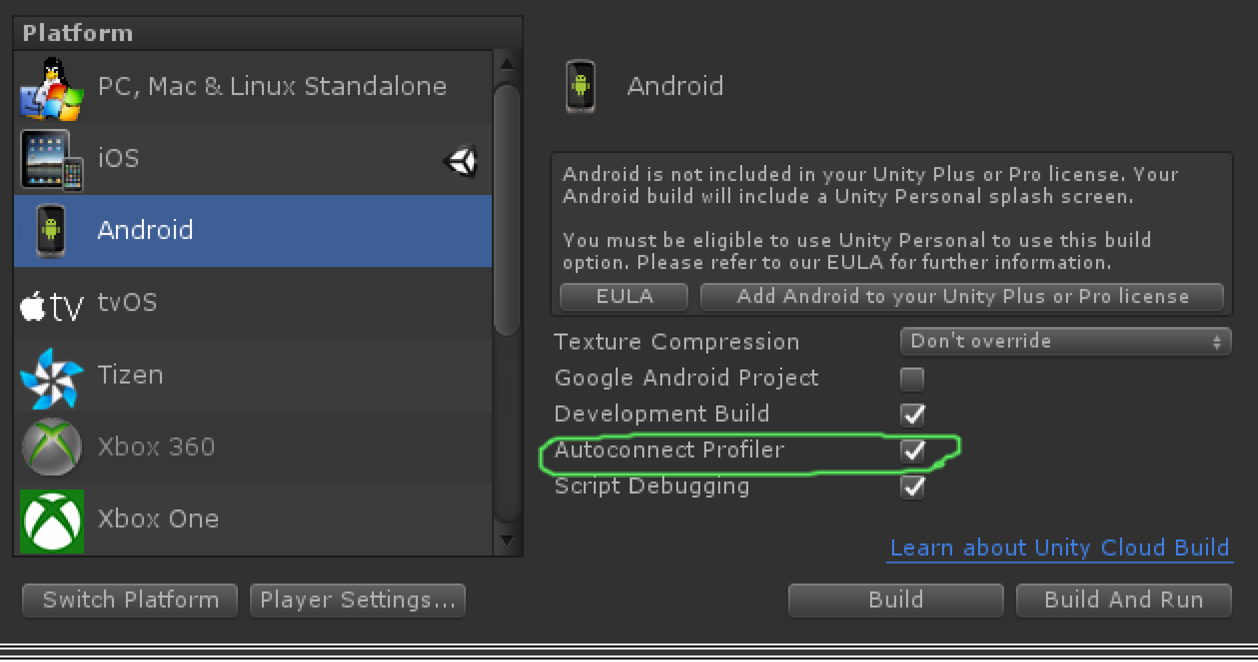
Bemerkungen

Verwenden des Profilers auf einem anderen Gerät

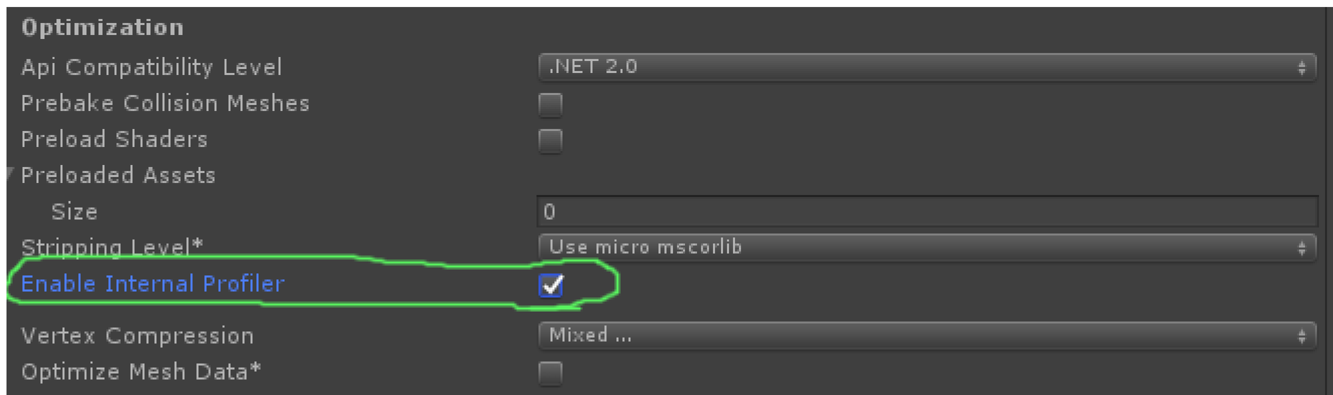
Es gibt ein paar wichtige Informationen, die Sie benötigen, um den Profiler auf verschiedenen Plattformen ordnungsgemäß anzuschließen.

Android

Um das Profil richtig **anzubinden** , muss die Schaltfläche "Erstellen und Ausführen" im Fenster "Build-Einstellungen" mit der **aktivierten** Option " **Autoconnect Profiler"** verwendet werden.



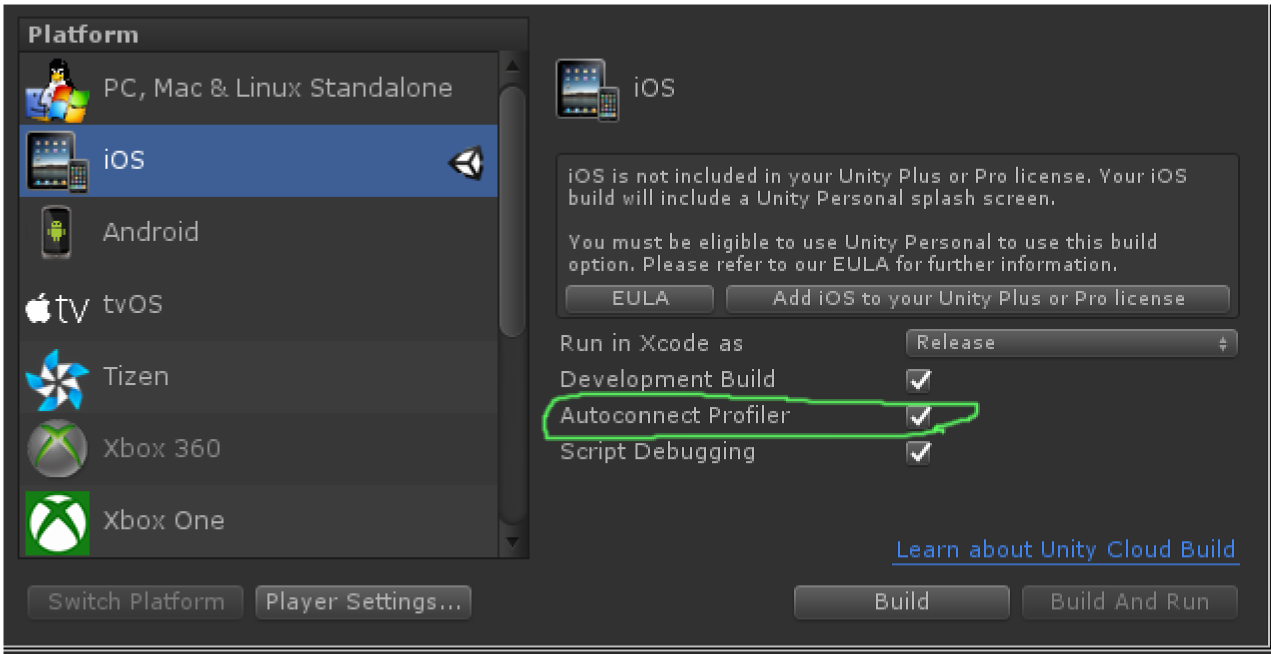
Eine weitere obligatorische Option: Im Inspector von Android Player-Einstellungen auf der Registerkarte Weitere Einstellungen gibt es ein Kontrollkästchen **Internen Profiler** aktivieren, der aktiviert werden muss, damit LogCat Profiler-Informationen ausgibt.



Wenn Sie nur "Build" verwenden, kann der Profiler keine Verbindung zu einem Android-Gerät herstellen, da "Build and Run" bestimmte Befehlszeilenargumente verwendet, um ihn mit LogCat zu starten.

iOS

Um das Profil ordnungsgemäß **anzufügen** , muss die Schaltfläche "Erstellen und Ausführen" im Fenster "Build-Einstellungen" mit der Option " **Autoconnect Profiler** " aktiviert werden, die beim ersten Durchlauf verwendet werden muss.



Unter iOS gibt es keine Option in den Player-Einstellungen, die festgelegt werden muss, damit der Profiler aktiviert werden kann. Es sollte sofort funktionieren.

Examples

Profiler Markup

Verwenden der Profiler- Klasse

Eine sehr gute Vorgehensweise ist die Verwendung von Profiler.BeginSample und Profiler.EndSample, da im Profiler-Fenster ein eigener Eintrag vorhanden ist.

Außerdem werden diese Tags bei der Erstellung ohne Entwicklung mit ConditionalAttribute entfernt, sodass Sie sie nicht aus Ihrem Code entfernen müssen.

```
public class SomeClass : MonoBehaviour
{
    void SomeFunction()
    {
        Profiler.BeginSample("SomeClass.SomeFunction");
        // Various call made here
```

```
        Profiler.EndSample();
    }
}
```

Dadurch wird im Profiler-Fenster ein Eintrag "SomeClass.SomeFunction" erstellt, der das Debugging und die Identifizierung des Flaschenhalses erleichtert.

Unity Profiler online lesen: https://riptutorial.com/de/unity3d/topic/6974/unity-profiler

Kapitel 35: User Interface System (UI)

Examples

Ereignis im Code abonnieren

Standardmäßig sollte man event mithilfe von inspector abonnieren. Manchmal ist es jedoch besser, dies im Code zu tun. In diesem Beispiel abonnieren wir ein Klickereignis einer Schaltfläche, um damit umzugehen.

```
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(Button))]
public class AutomaticClickHandler : MonoBehaviour
{
    private void Awake()
    {
        var button = this.GetComponent<Button>();
        button.onClick.AddListener(HandleClick);
    }

    private void HandleClick()
    {
        Debug.Log("AutomaticClickHandler.HandleClick()", this);
    }
}
```

Die UI-Komponenten stellen ihren Hauptlistener normalerweise einfach bereit:

- Schaltfläche : onClick
- Dropdown: onValueChanged
- InputField: onEndEdit , onValidateInput , onValueChanged
- Bildlaufleiste: onValueChanged
- ScrollRect: onValueChanged
- Slider: onValueChanged
- Toggle: onValueChanged

Hinzufügen von Maus-Listenern

Manchmal möchten Sie Listener für bestimmte Ereignisse hinzufügen, die nicht von den Komponenten bereitgestellt werden, insbesondere Mausereignisse. Dazu müssen Sie sie mithilfe einer `EventTrigger` Komponente selbst `EventTrigger` :

```
using UnityEngine;
using UnityEngine.EventSystems;

[RequireComponent(typeof( EventTrigger ))]
public class CustomListenersExample : MonoBehaviour
{
```

```
    void Start( )
    {
        EventTrigger eventTrigger = GetComponent<EventTrigger>( );
        EventTrigger.Entry entry = new EventTrigger.Entry( );
        entry.eventID = EventTriggerType.PointerDown;
        entry.callback.AddListener( ( data ) => { OnPointerDownDelegate(
(PointerEventData)data ); } );
        eventTrigger.triggers.Add( entry );
    }

    public void OnPointerDownDelegate( PointerEventData data )
    {
        Debug.Log( "OnPointerDownDelegate called." );
    }
}
```

Verschiedene EventID sind möglich:

- PointerEnter
- PointerExit
- PointerDown
- PointerUp
- PointerClick
- Ziehen
- Fallen
- Scrollen
- UpdateSelected
- Wählen
- Abwählen
- Bewegung
- InitializePotentialDrag
- BeginnenDrag
- EndDrag
- einreichen
- Stornieren

User Interface System (UI) online lesen: https://riptutorial.com/de/unity3d/topic/2296/user-interface-system--ui-

Kapitel 36: Vector3

Einführung

Die `Vector3` Struktur stellt eine 3D-Koordinate dar und ist eine der `UnityEngine` der `UnityEngine` Bibliothek. Die `Vector3` Struktur wird am häufigsten in der `Transform` Komponente der meisten Spielobjekte gefunden, wo sie zum Halten von *Position* und *Skalierung verwendet wird* . `Vector3` bietet eine gute Funktionalität zum Ausführen allgemeiner Vektoroperationen. Weitere `Vector3` zur `Vector3` Struktur finden Sie in der Unity-API.

Syntax

- public Vector3 ();
- public Vector3 (Float x, Float y);
- public Vector3 (float x, float y, float z);
- Vector3.Lerp (Vector3 startPosition, Vector3 targetPosition, Float-BewegungFraktion);
- Vector3.LerpUnclamped (Vector3 startPosition, Vector3 targetPosition, Float-BewegungFraction);
- Vector3.MoveTowards (Vector3 startPosition, Vector3 targetPosition, Schwimmdistanz);

Examples

Statische Werte

Die `Vector3` Struktur enthält einige statische Variablen, die häufig verwendete `Vector3` Werte `Vector3` . Die meisten stellen eine *Richtung dar* , aber sie können immer noch kreativ verwendet werden, um zusätzliche Funktionen bereitzustellen.

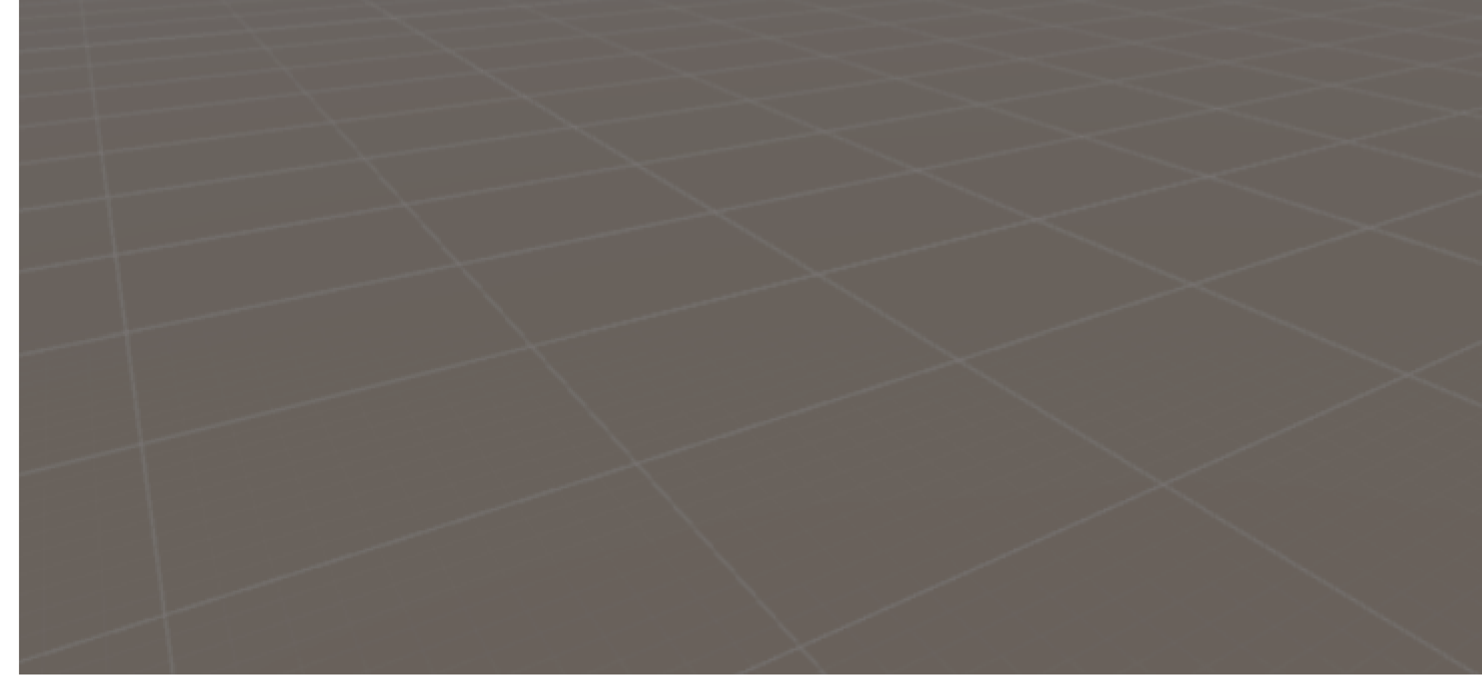
`Vector3.zero` und `Vector3.one` werden normalerweise in Verbindung mit einem *normalisierten* `Vector3` . ein `Vector3` bei dem die `x` , `y` und `z` Werte eine Stärke von 1 haben. `Vector3.zero` stellt `Vector3.zero` den niedrigsten Wert dar, während `Vector3.one` den größten Wert darstellt.

`Vector3.zero` wird auch häufig verwendet, um die Standardposition bei Objekttransformationen `Vector3.zero` .

Die folgende Klasse verwendet `Vector3.zero` und `Vector3.one` zum Aufblasen und `Vector3.one` einer Kugel.

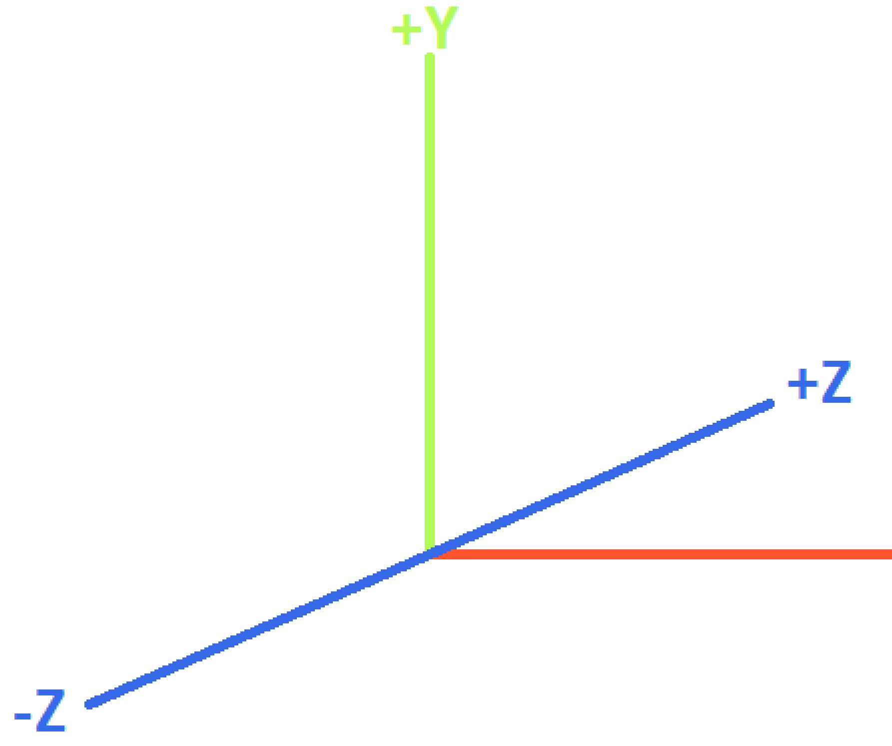
```
using UnityEngine;
```

```
public class Inflater : MonoBehaviour
{
    <summary>A sphere set up to inflate and deflate between two values.</summary>
    public ScaleBetween sphere;

    ///<summary>On start, set the sphere GameObject up to inflate
    /// and deflate to the corresponding values.</summary>
    void Start()
    {
        // Vector3.zero = Vector3(0, 0, 0); Vector3.one = Vector3(1, 1, 1);
        sphere.SetScale(Vector3.zero, Vector3.one);
    }
}
```



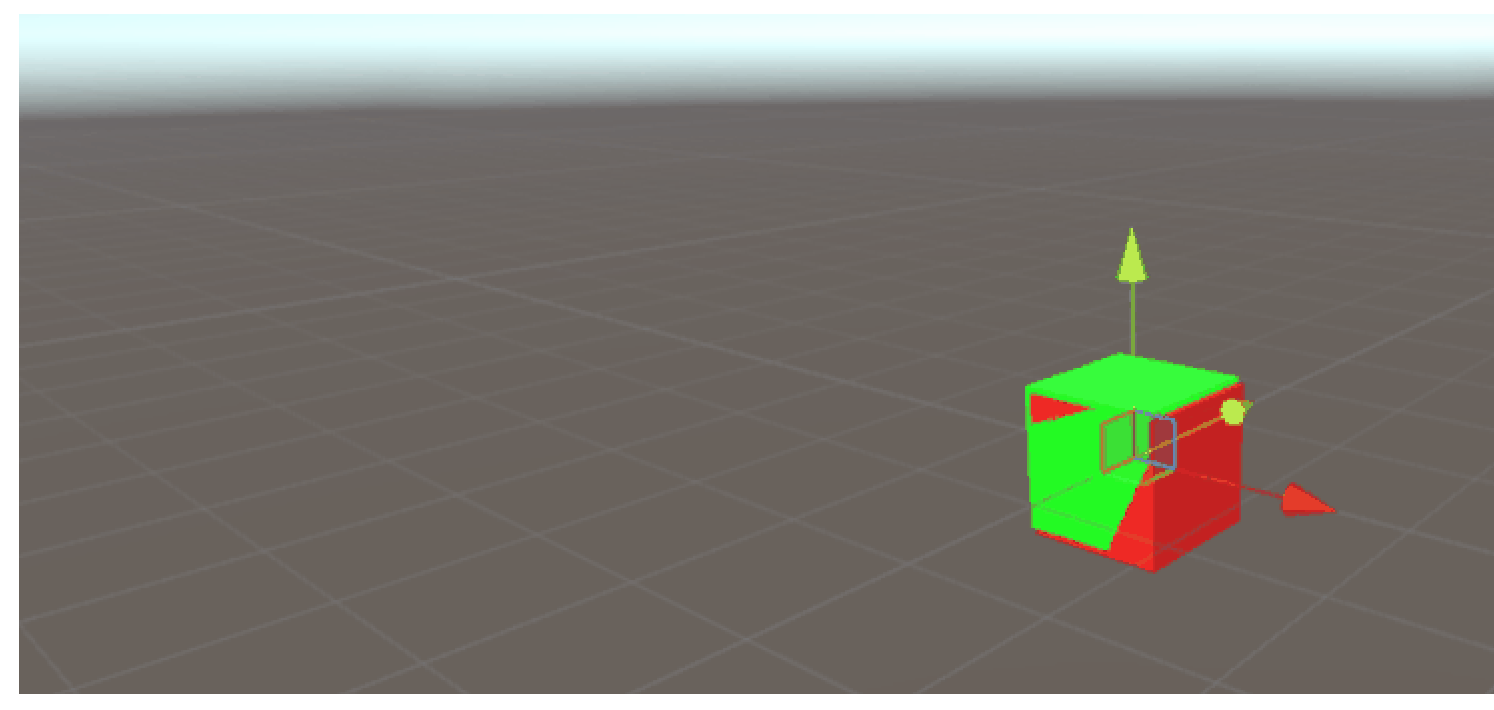
Statische Anweisungen

Die statischen Richtungen können in einer Reihe von Anwendungen nützlich sein, wobei die Richtung entlang der positiven und negativen Richtung aller drei Achsen liegt. Es ist wichtig zu wissen, dass Unity ein linkshändiges Koordinatensystem verwendet, das die Richtung beeinflusst.



Die folgende Klasse verwendet die statischen `Vector3` Richtungen, um Objekte entlang der drei Achsen zu verschieben.

```
using UnityEngine;

public class StaticMover : MonoBehaviour
{
    <summary>GameObjects set up to move back and forth between two directions.</summary>
    public MoveBetween xMovement, yMovement, zMovement;

    ///<summary>On start, set each MoveBetween GameObject up to move
    /// in the corresponding direction(s).</summary>
    void Start()
    {
        // Vector3.left = Vector3(-1, 0, 0); Vector3.right = Vector3(1, 0, 0);
        xMovement.SetDirections(Vector3.left, Vector3.right);

        // Vector3.down = Vector3(0, -1, 0); Vector3.up = Vector3(0, 0, 1);
        yMovement.SetDirections(Vector3.down, Vector3.up);

        // Vector3.back = Vector3(0, 0, -1); Vector3.forward = Vector3(0, 0, 1);
        zMovement.SetDirections(Vector3.back, Vector3.forward);
    }
}
```



Index

Wert	x	y	z	Äquivalente `new Vector3()` Methode
`Vector3.zero`	0	0	0	`new Vector3(0, 0, 0)`
`Vector3.one`	1	1	1	`new Vector3(1, 1, 1)`
`Vector3.left`	-1	0	0	`new Vector3(-1, 0, 0)`
`Vector3.right`	1	0	0	`new Vector3(1, 0, 0)`
`Vector3.down`	0	-1	0	`new Vector3(0, -1, 0)`
`Vector3.up`	0	1	0	`new Vector3(0, 1, 0)`
`Vector3.back`	0	0	-1	`new Vector3(0, 0, -1)`
`Vector3.forward`	0	0	1	`new Vector3(0, 0, 1)`

Einen Vector3 erstellen

Eine `Vector3` Struktur kann auf verschiedene Arten erstellt werden. `Vector3` ist eine Struktur und muss daher normalerweise vor der Verwendung instanziiert werden.

Konstrukteure

Es gibt drei eingebaute Konstruktoren zum Instantiieren eines `Vector3` .

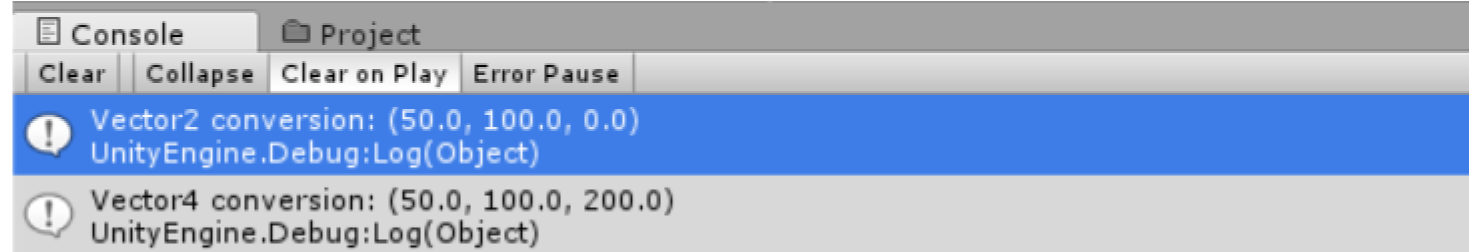
Konstrukteur	Ergebnis
`new Vector3()`	Erzeugt eine `Vector3` Struktur mit Koordinaten von (0, 0, 0).
`new Vector3(float x, float y)`	Erzeugt eine `Vector3` Struktur mit den angegebenen `x` und `y` -Koordinaten. `z` wird auf 0 gesetzt.
`new Vector3(float x, float y, float z)`	Erzeugt eine `Vector3` Struktur mit den angegebenen `x` , `y` und `z` -Koordinaten.

Konvertieren von einem `Vector2` oder `Vector4`

In seltenen `Vector2` können Sie auf Situationen `Vector2` in denen Sie die Koordinaten einer `Vector2` oder `Vector4` Struktur als `Vector3` . In solchen Fällen können Sie den `Vector2` oder `Vector4` einfach direkt in den `Vector3` , ohne ihn vorher zu instantiieren. Wie sollte davon ausgegangen werden, ein `Vector2` nur struct passieren `x` und `y` - Werte, während eine `Vector4` Klasse seine auslassen wird `w` .

Wir können die direkte Konvertierung im folgenden Skript sehen.

```
void VectorConversionTest()
{
    Vector2 vector2 = new Vector2(50, 100);
    Vector4 vector4 = new Vector4(50, 100, 200, 400);

    Vector3 fromVector2 = vector2;
    Vector3 fromVector4 = vector4;

    Debug.Log("Vector2 conversion: " + fromVector2);
    Debug.Log("Vector4 conversion: " + fromVector4);
}
```



Bewegung anwenden

Die `Vector3` Struktur enthält einige statische Funktionen, die `Vector3` sind, wenn Sie eine Bewegung auf den `Vector3` anwenden `Vector3` .

`Lerp`

und `LerpUnclamped`

Die Lerp-Funktionen sorgen für eine Bewegung zwischen zwei Koordinaten basierend auf einem bereitgestellten Bruchteil. `LerpUnclamped` `Lerp` nur eine Bewegung zwischen den beiden Koordinaten `LerpUnclamped` erlaubt `LerpUnclamped` Brüche, die sich außerhalb der Grenzen zwischen den beiden Koordinaten bewegen.

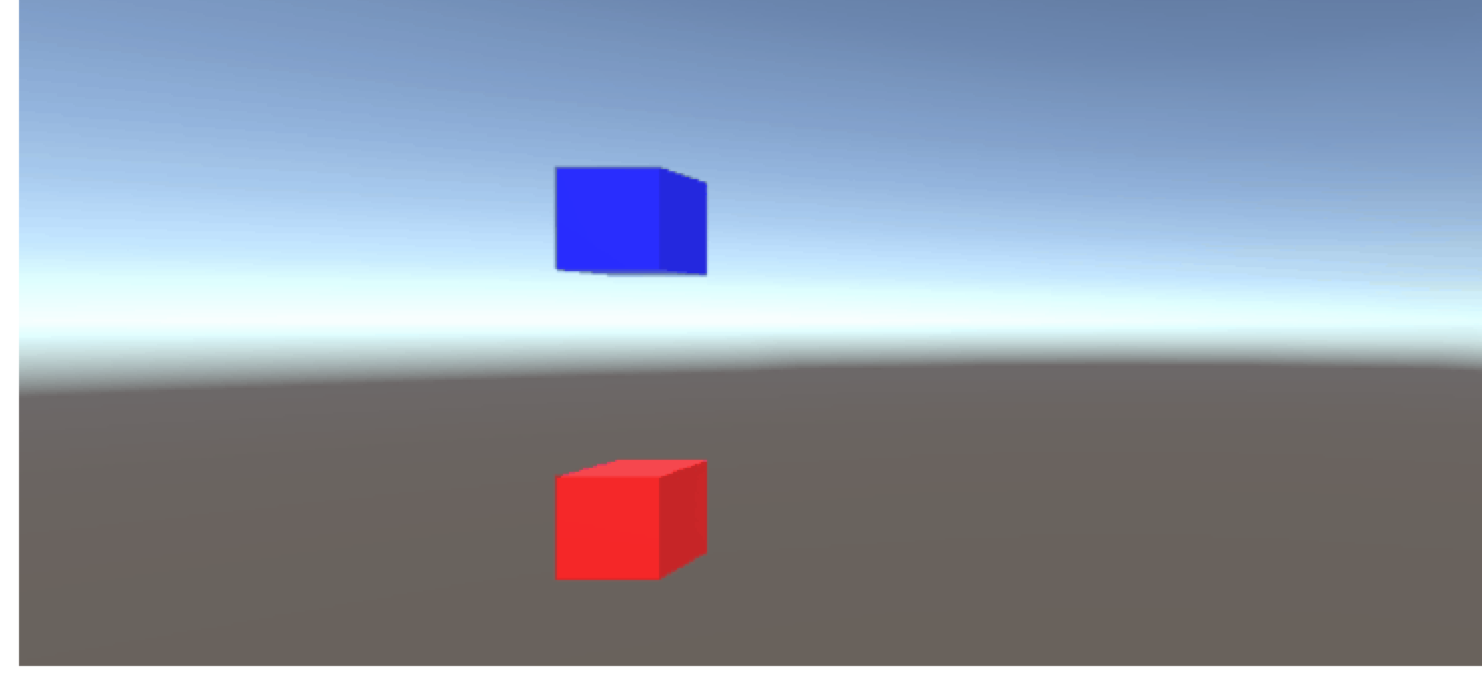
Den Bewegungsanteil stellen wir als `float` bereit. Mit einem Wert von `0.5` finden wir den Mittelpunkt zwischen den beiden `Vector3` -Koordinaten. Ein Wert von `0` oder `1` gibt den ersten oder zweiten `Vector3` zurück, da diese Werte entweder mit keiner Bewegung korrelieren (und somit den ersten `Vector3`) oder eine abgeschlossene Bewegung (die den zweiten `Vector3`). Es ist wichtig zu beachten, dass keine der Funktionen für die Änderung des Bewegungsanteils geeignet ist. Dies müssen wir manuell berücksichtigen.

Bei `Lerp` werden alle Werte zwischen `0` und `1` geklemmt. Dies ist nützlich, wenn wir uns in eine Richtung bewegen möchten und das Ziel nicht überschreiten möchten. `LerpUnclamped` kann einen beliebigen Wert annehmen und kann verwendet werden, um *sich* vom Ziel *weg* oder *am* Ziel *vorbei zu* `LerpUnclamped` .

Das folgende Skript verwendet `Lerp` und `LerpUnclamped` , um ein Objekt in einem gleichmäßigen Tempo zu verschieben.

```
using UnityEngine;

public class Lerping : MonoBehaviour
{
    /// <summary>The red box will use Lerp to move. We will link
    /// this object in via the inspector.</summary>
    public GameObject lerpObject;
    /// <summary>The starting position for our red box.</summary>
    public Vector3 lerpStart = new Vector3(0, 0, 0);
    /// <summary>The end position for our red box.</summary>
    public Vector3 lerpTarget = new Vector3(5, 0, 0);

    /// <summary>The blue box will use LerpUnclamped to move. We will
    /// link this object in via the inspector.</summary>
    public GameObject lerpUnclampedObject;
    /// <summary>The starting position for our blue box.</summary>
    public Vector3 lerpUnclampedStart = new Vector3(0, 3, 0);
    /// <summary>The end position for our blue box.</summary>
    public Vector3 lerpUnclampedTarget = new Vector3(5, 3, 0);

    /// <summary>The current fraction to increment our lerp functions by.</summary>
    public float lerpFraction = 0;

    private void Update()
    {
        // First, I increment the lerp fraction.
        // delaTime * 0.25 should give me a value of +1 every second.
        lerpFraction += (Time.deltaTime * 0.25f);

        // Next, we apply the new lerp values to the target transform position.
        lerpObject.transform.position
```

```
            = Vector3.Lerp(lerpStart, lerpTarget, lerpFraction);
        lerpUnclampedObject.transform.position
            = Vector3.LerpUnclamped(lerpUnclampedStart, lerpUnclampedTarget, lerpFraction);
    }
}
```



MoveTowards

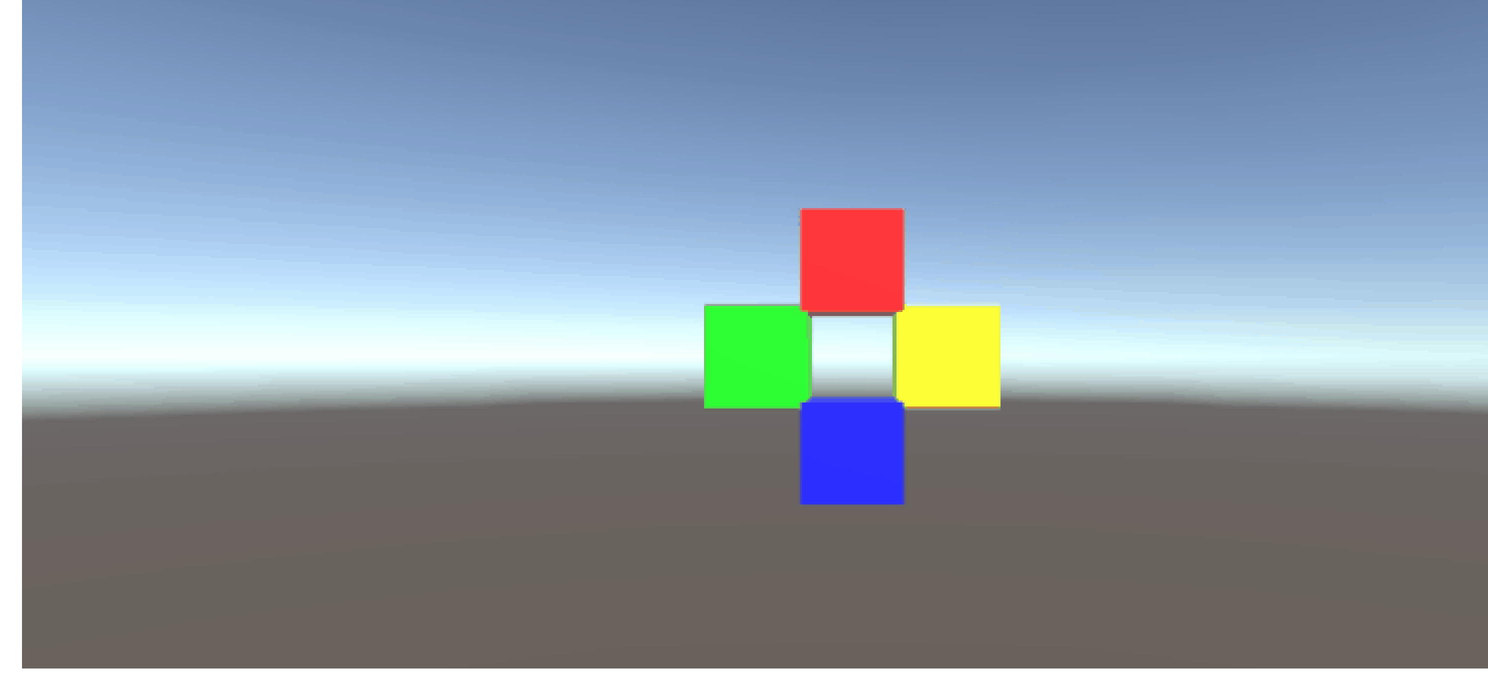
`MoveTowards` verhält sich *ähnlich* wie `Lerp` ; Der Hauptunterschied besteht darin, dass wir eine tatsächliche *Entfernung* angeben, anstatt einen *Bruch* zwischen zwei Punkten. Es ist wichtig zu `MoveTowards` , dass `MoveTowards` sich nicht über das Ziel `Vector3` .

Ähnlich wie bei `LerpUnclamped` können wir einen *negativen* Abstandswert `LerpUnclamped` , um *sich* vom Ziel `Vector3` . In solchen Fällen bewegen wir uns niemals am Ziel `Vector3` und somit ist die Bewegung unbestimmt. In diesen Fällen können wir das Ziel `Vector3` als "entgegengesetzte Richtung" behandeln. Solange der `Vector3` in die gleiche Richtung `Vector3` , sollte sich die negative Bewegung relativ zum Start `Vector3` wie normal verhalten.

Das folgende Skript verwendet `MoveTowards` , um eine Gruppe von Objekten mit einem geglätteten Abstand zu einer Gruppe von Positionen zu verschieben.

```
using UnityEngine;

public class MoveTowardsExample : MonoBehaviour
{
    /// <summary>The red cube will move up, the blue cube will move down,
    /// the green cube will move left and the yellow cube will move right.
    /// These objects will be linked via the inspector.</summary>
    public GameObject upCube, downCube, leftCube, rightCube;
    /// <summary>The cubes should move at 1 unit per second.</summary>
    float speed = 1f;
```

```
    void Update()
    {
        // We determine our distance by applying a deltaTime scale to our speed.
        float distance = speed * Time.deltaTime;

        // The up cube will move upwards, until it reaches the
        //position of (Vector3.up * 2), or (0, 2, 0).
        upCube.transform.position
            = Vector3.MoveTowards(upCube.transform.position, (Vector3.up * 2f), distance);

        // The down cube will move downwards, as it enforces a negative distance..
        downCube.transform.position
            = Vector3.MoveTowards(downCube.transform.position, Vector3.up * 2f, -distance);

        // The right cube will move to the right, indefinetly, as it is constantly updating
        // its target position with a direction based off the current position.
        rightCube.transform.position = Vector3.MoveTowards(rightCube.transform.position,
            rightCube.transform.position + Vector3.right, distance);

        // The left cube does not need to account for updating its target position,
        // as it is moving away from the target position, and will never reach it.
        leftCube.transform.position
            = Vector3.MoveTowards(leftCube.transform.position, Vector3.right, -distance);
    }
}
```



SmoothDamp

`SmoothDamp` Sie sich `SmoothDamp` als eine Variante von `MoveTowards` mit integrierter Glättung vor. Laut offizieller Dokumentation wird diese Funktion am häufigsten verwendet, um eine reibungslose Kameraführung zu ermöglichen.

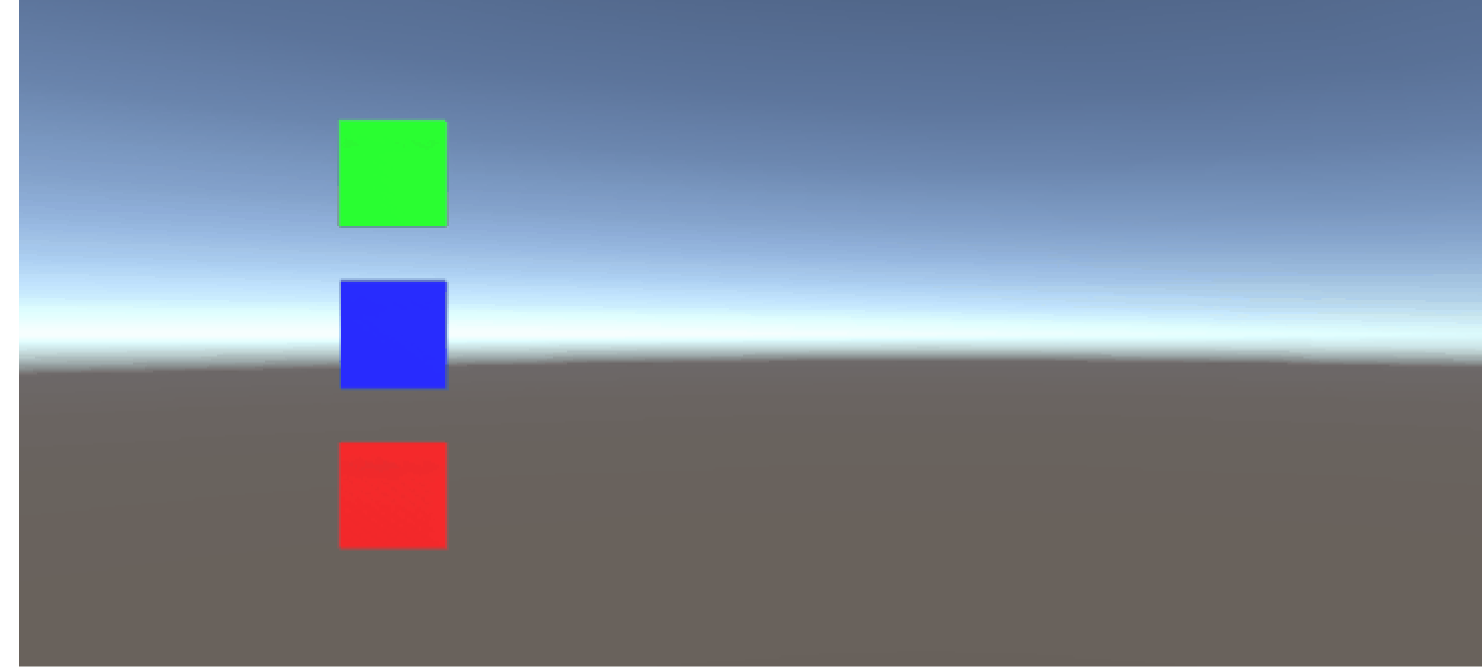
Neben den Start- und Ziel- `Vector3` Koordinaten müssen wir auch einen `Vector3` , um die

Geschwindigkeit darzustellen, und einen `float` der die *ungefähre* Zeit darstellt, die die Bewegung benötigt. Im Gegensatz zu vorherigen Beispielen geben wir die Geschwindigkeit als *Referenz an* , um sie intern zu erhöhen. Es ist wichtig, dies zu beachten, da das Ändern der Geschwindigkeit außerhalb der Funktion, während wir die Funktion noch ausführen, zu unerwünschten Ergebnissen führen kann.

Zusätzlich zu den *erforderlichen* Variablen, können wir auch einen `float` die maximale Geschwindigkeit unseres Objektes zu repräsentieren, und einen `float` den Zeitabstand seit dem letzten darstellen `SmoothDamp` Aufruf an das Objekt. Wir müssen diese Werte nicht *angeben* . Standardmäßig gibt es keine Höchstgeschwindigkeit, und die Zeitlücke wird als `Time.deltaTime` interpretiert. Noch wichtiger ist, wenn Sie die Funktion eins pro Objekt innerhalb einer `MonoBehaviour.Update()` Funktion `MonoBehaviour.Update()` , müssen Sie *keine* `MonoBehaviour.Update()` .

```
using UnityEngine;

public class SmoothDampMovement : MonoBehaviour
{
    /// <summary>The red cube will imitate the default SmoothDamp function.
    /// The blue cube will move faster by manipulating the "time gap", while
    /// the green cube will have an enforced maximum speed. Note that these
    /// objects have been linked via the inspector.</summary>
    public GameObject smoothObject, fastSmoothObject, cappedSmoothObject;

    /// <summary>We must instantiate the velocities, externally, so they may
    /// be manipulated from within the function. Note that by making these
    /// vectors public, they will be automatically instantiated as Vector3.Zero
    /// through the inspector. This also allows us to view the velocities,
    /// from the inspector, to observe how they change.</summary>
    public Vector3 regularVelocity, fastVelocity, cappedVelocity;

    /// <summary>Each object should move 10 units along the X-axis.</summary>
    Vector3 regularTarget = new Vector3(10f, 0f);
    Vector3 fastTarget = new Vector3(10f, 1.5f);
    Vector3 cappedTarget = new Vector3(10f, 3f);

    /// <summary>We will give a target time of 5 seconds.</summary>
    float targetTime = 5f;

    void Update()
    {
        // The default SmoothDamp function will give us a general smooth movement.
        smoothObject.transform.position = Vector3.SmoothDamp(smoothObject.transform.position,
            regularTarget, ref regularVelocity, targetTime);

        // Note that a "maxSpeed" outside of reasonable limitations should not have any
        // effect, while providing a "deltaTime" of 0 tells the function that no time has
        // passed since the last SmoothDamp call, resulting in no movement, the second time.
        smoothObject.transform.position = Vector3.SmoothDamp(smoothObject.transform.position,
            regularTarget, ref regularVelocity, targetTime, 10f, 0f);

        // Note that "deltaTime" defaults to Time.deltaTime due to an assumption that this
        // function will be called once per update function. We can call the function
        // multiple times during an update function, but the function will assume that enough
        // time has passed to continue the same approximate movement. As a result,
        // this object should reach the target, quicker.
```

```
        fastSmoothObject.transform.position = Vector3.SmoothDamp(
            fastSmoothObject.transform.position, fastTarget, ref fastVelocity, targetTime);
        fastSmoothObject.transform.position = Vector3.SmoothDamp(
            fastSmoothObject.transform.position, fastTarget, ref fastVelocity, targetTime);

        // Lastly, note that a "maxSpeed" becomes irrelevant, if the object does not
        // realistically reach such speeds. Linear speed can be determined as
        // (Distance / Time), but given the simple fact that we start and end slow, we can
        // infer that speed will actually be higher, during the middle. As such, we can
        // infer that a value of (Distance / Time) or (10/5) will affect the
        // function. We will half the "maxSpeed", again, to make it more noticeable.
        cappedSmoothObject.transform.position = Vector3.SmoothDamp(
            cappedSmoothObject.transform.position,
            cappedTarget, ref cappedVelocity, targetTime, 1f);
    }
}
```



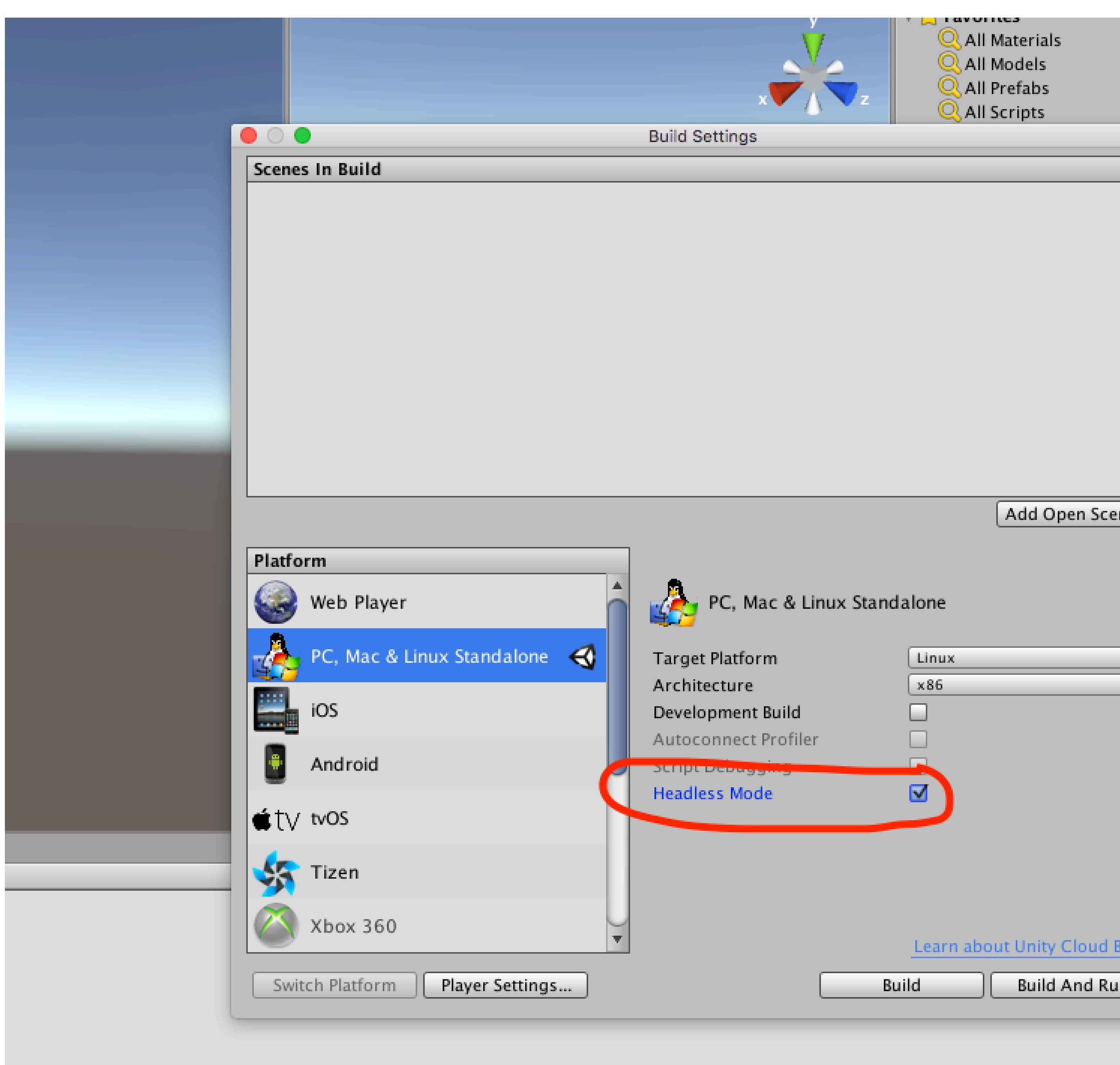
Vector3 online lesen: https://riptutorial.com/de/unity3d/topic/7827/vector3

Kapitel 37: Vernetzung

Bemerkungen

Headless-Modus in Unity

Wenn Sie einen Server für die Bereitstellung unter Linux erstellen, haben die Build-Einstellungen die Option "Headless-Modus". Eine mit dieser Option erstellte Anwendung zeigt nichts an und liest keine Benutzereingaben. Dies ist normalerweise das, was wir für einen Server wollen.



Examples

Erstellen eines Servers, eines Clients und Senden einer Nachricht

Unity-Netzwerke bieten die High-Level-API (HLA), um Netzwerkkommunikation von Low-Level-Implementierungen abzugrenzen.

In diesem Beispiel erfahren Sie, wie Sie einen Server erstellen, der mit einem oder mehreren Clients kommunizieren kann.

Mit dem HLA können wir eine Klasse problemlos serialisieren und Objekte dieser Klasse über das Netzwerk senden.

Die Klasse, die wir zur Serialisierung verwenden

Diese Klasse muss von MessageBase übernommen werden. In diesem Beispiel wird nur eine Zeichenfolge innerhalb dieser Klasse gesendet.

```
using System;
using UnityEngine.Networking;

public class MyNetworkMessage : MessageBase
{
    public string message;
}
```

Server erstellen

Wir erstellen einen Server, der den Port 9999 überwacht, maximal 10 Verbindungen zulässt und Objekte aus dem Netzwerk unserer benutzerdefinierten Klasse liest.

Der HLA ordnet einer ID verschiedene Nachrichtentypen zu. In der MsgType-Klasse ist ein Standardnachrichtentyp von Unity Networking definiert. Beispielsweise hat der Verbindungstyp die ID 32 und er wird im Server aufgerufen, wenn ein Client eine Verbindung herstellt, oder im Client, wenn er sich mit einem Server verbindet. Sie können Handler registrieren, um die verschiedenen Nachrichtentypen zu verwalten.

Wenn Sie, wie in unserem Fall, eine benutzerdefinierte Klasse senden, definieren wir einen Handler mit einer neuen ID, die der Klasse zugeordnet ist, die wir über das Netzwerk senden.

```
using UnityEngine;
using System.Collections;
using UnityEngine.Networking;

public class Server : MonoBehaviour {

    int port = 9999;
```

```
int maxConnections = 10;

// The id we use to identify our messages and register the handler
short messageID = 1000;

// Use this for initialization
void Start () {
    // Usually the server doesn't need to draw anything on the screen
    Application.runInBackground = true;
    CreateServer();
}

void CreateServer() {
    // Register handlers for the types of messages we can receive
    RegisterHandlers ();

    var config = new ConnectionConfig ();
    // There are different types of channels you can use, check the official documentation
    config.AddChannel (QosType.ReliableFragmented);
    config.AddChannel (QosType.UnreliableFragmented);

    var ht = new HostTopology (config, maxConnections);

    if (!NetworkServer.Configure (ht)) {
        Debug.Log ("No server created, error on the configuration definition");
        return;
    } else {
        // Start listening on the defined port
        if(NetworkServer.Listen (port))
            Debug.Log ("Server created, listening on port: " + port);
        else
            Debug.Log ("No server created, could not listen to the port: " + port);
    }
}

void OnApplicationQuit() {
    NetworkServer.Shutdown ();
}

private void RegisterHandlers () {
    // Unity have different Messages types defined in MsgType
    NetworkServer.RegisterHandler (MsgType.Connect, OnClientConnected);
    NetworkServer.RegisterHandler (MsgType.Disconnect, OnClientDisconnected);

    // Our message use his own message type.
    NetworkServer.RegisterHandler (messageID, OnMessageReceived);
}

private void RegisterHandler(short t, NetworkMessageDelegate handler) {
    NetworkServer.RegisterHandler (t, handler);
}

void OnClientConnected(NetworkMessage netMessage)
{
    // Do stuff when a client connects to this server

    // Send a thank you message to the client that just connected
    MyNetworkMessage messageContainer = new MyNetworkMessage();
    messageContainer.message = "Thanks for joining!";

    // This sends a message to a specific client, using the connectionId
```

```
        NetworkServer.SendToClient(netMessage.conn.connectionId,messageID,messageContainer);

        // Send a message to all the clients connected
        messageContainer = new MyNetworkMessage();
        messageContainer.message = "A new player has conencted to the server";

        // Broadcast a message a to everyone connected
        NetworkServer.SendToAll(messageID,messageContainer);
    }

    void OnClientDisconnected(NetworkMessage netMessage)
    {
        // Do stuff when a client dissconnects
    }

    void OnMessageReceived(NetworkMessage netMessage)
    {
        // You can send any object that inherence from MessageBase
        // The client and server can be on different projects, as long as the MyNetworkMessage
or the class you are using have the same implementation on both projects
        // The first thing we do is deserialize the message to our custom type
        var objectMessage = netMessage.ReadMessage<MyNetworkMessage>();
        Debug.Log("Message received: " + objectMessage.message);

    }
}
```

Der Kunde

Nun erstellen wir einen Client

```
using System;
using UnityEngine;
using UnityEngine.Networking;


public class Client : MonoBehaviour
{
    int port = 9999;
    string ip = "localhost";

    // The id we use to identify our messages and register the handler
    short messageID = 1000;

    // The network client
    NetworkClient client;

    public Client ()
    {
        CreateClient();
    }

    void CreateClient()
    {
        var config = new ConnectionConfig ();

        // Config the Channels we will use
        config.AddChannel (QosType.ReliableFragmented);
```

```
        config.AddChannel (QosType.UnreliableFragmented);

        // Create the client ant attach the configuration
        client = new NetworkClient ();
        client.Configure (config,1);

        // Register the handlers for the different network messages
        RegisterHandlers();

        // Connect to the server
        client.Connect (ip, port);
    }

    // Register the handlers for the different message types
    void RegisterHandlers () {

        // Unity have different Messages types defined in MsgType
        client.RegisterHandler (messageID, OnMessageReceived);
        client.RegisterHandler(MsgType.Connect, OnConnected);
        client.RegisterHandler(MsgType.Disconnect, OnDisconnected);
    }

    void OnConnected(NetworkMessage message) {
        // Do stuff when connected to the server

        MyNetworkMessage messageContainer = new MyNetworkMessage();
        messageContainer.message = "Hello server!";

        // Say hi to the server when connected
        client.Send(messageID,messageContainer);
    }

    void OnDisconnected(NetworkMessage message) {
        // Do stuff when disconnected to the server
    }

    // Message received from the server
    void OnMessageReceived(NetworkMessage netMessage)
    {
        // You can send any object that inherence from MessageBase
        // The client and server can be on different projects, as long as the MyNetworkMessage
or the class you are using have the same implementation on both projects
        // The first thing we do is deserialize the message to our custom type
        var objectMessage = netMessage.ReadMessage<MyNetworkMessage>();

        Debug.Log("Message received: " + objectMessage.message);
    }
}
```

Vernetzung online lesen: https://riptutorial.com/de/unity3d/topic/5671/vernetzung

Kapitel 38: Verwandelt sich

Syntax

- void Transform.Translate (Vector3-Übersetzung, Space relativeTo = Space.Self)
- void Transform.Translate (float x, float y, float z, Space relativeTo = Space.Self)
- void Transform.Rotate (Vector3 eulerAngles, Space relativeTo = Space.Self)
- void Transform.Rotate (Float xAngle, Float yAngle, Float zAngle, Space relativeTo = Space.Self)
- void Transform.Rotate (Vector3-Achse, Float-Winkel, Space relativTo = Space.Self)
- void Transform.RotateAround (Vector3-Punkt, Vector3-Achse, Schwimmwinkel)
- void Transform.LookAt (Transformationsziel, Vector3 worldUp = Vector3.up)
- void Transform.LookAt (Vector3 worldPosition, Vector3 worldUp = Vector3.up)

Examples

Überblick

Transformationen enthalten die Mehrheit der Daten eines Objekts in der Einheit, einschließlich der übergeordneten Elemente, der untergeordneten Elemente, der Position, der Rotation und der Skalierung. Es hat auch Funktionen, um jede dieser Eigenschaften zu ändern. Jedes GameObject hat eine Transformation.

Objekt verschieben (verschieben)

```
// Move an object 10 units in the positive x direction
transform.Translate(10, 0, 0);

// translating with a vector3
vector3 distanceToMove = new Vector3(5, 2, 0);
transform.Translate(distanceToMove);
```

Objekt drehen

```
// Rotate an object 45 degrees about the Y axis
transform.Rotate(0, 45, 0);

// Rotates an object about the axis passing through point (in world coordinates) by angle in
degrees
transform.RotateAround(point, axis, angle);
// Rotates on it's place, on the Y axis, with 90 degrees per second
transform.RotateAround(Vector3.zero, Vector3.up, 90 * Time.deltaTime);

// Rotates an object to make it's forward vector point towards the other object
transform.LookAt(otherTransform);
// Rotates an object to make it's forward vector point towards the given position (in world
coordinates)
transform.LookAt(new Vector3(10, 5, 0));
```

Weitere Informationen und Beispiele finden Sie in der Unity-Dokumentation .

Beachten Sie auch, dass, wenn das Spiel starre Körper verwendet, die Transformation nicht direkt interagiert werden darf (es sei denn, der starre Körper hat `isKinematic == true`). In diesem Fall verwenden Sie AddForce oder andere ähnliche Methoden, um direkt auf den starren Körper einzuwirken .

Erziehung und Kinder

Unity arbeitet mit Hierarchien, um Ihr Projekt organisiert zu halten. Sie können Objekte mit dem Editor einer Stelle in der Hierarchie zuweisen, dies kann jedoch auch über Code erfolgen.

Erziehung

Sie können das übergeordnete Objekt eines Objekts mit den folgenden Methoden festlegen

```
var other = GetOtherGameObject();
other.transform.SetParent( transform );
other.transform.SetParent( transform, worldPositionStays );
```

Wenn Sie ein übergeordnetes Transformationsobjekt festlegen, wird die Position des Objekts als Weltposition beibehalten. Sie können wählen, diese Position relativ zu machen, indem Sie für den Parameter *worldPositionStays* den *Wert false angeben* .

Sie können auch mit der folgenden Methode überprüfen, ob das Objekt einer anderen Transformation untergeordnet ist

```
other.transform.IsChildOf( transform );
```

Ein Kind bekommen

Da Objekte einander übergeordnet werden können, können Sie auch Kinder in der Hierarchie suchen. Am einfachsten geht dies mit der folgenden Methode

```
transform.Find( "other" );
transform.FindChild( "other" );
```

Hinweis: FindChild ruft Find unter der Haube auf

Sie können auch nach untergeordneten Elementen in der Hierarchie suchen. Dazu fügen Sie ein "/" hinzu, um festzulegen, dass Sie tiefer gehen sollen.

```
transform.Find( "other/another" );
transform.FindChild( "other/another" );
```

Eine andere Methode zum Abrufen eines Kindes ist die Verwendung von GetChild

```
transform.GetChild( index );
```

GetChild erfordert eine ganze Zahl als Index, die kleiner als die Gesamtzahl der untergeordneten Elemente sein muss

```
int count = transform.childCount;
```

Geschwisterindex ändern

Sie können die Reihenfolge der Kinder eines GameObjects ändern. Sie können dies tun, um die Zeichenreihenfolge der Kinder zu definieren (vorausgesetzt, sie befinden sich auf derselben Z-Ebene und derselben Sortierreihenfolge).

```
other.transform.SetSiblingIndex( index );
```

Sie können den Schwesterindex auch mit den folgenden Methoden schnell auf den ersten oder letzten Index setzen

```
other.transform.SetAsFirstSibling();
other.transform.SetAsLastSibling();
```

Alle Kinder trennen

Wenn Sie alle untergeordneten Elemente einer Transformation freigeben möchten, können Sie Folgendes tun:

```
foreach(Transform child in transform)
{
    child.parent = null;
}
```

Außerdem bietet Unity eine Methode für diesen Zweck:

```
transform.DetachChildren();
```

Grundsätzlich setzen sowohl Loop als auch `DetachChildren()` die Eltern von Kindern der ersten `DetachChildren()` auf Null - was bedeutet, dass sie keine Eltern haben werden.

(Kinder der ersten Vertiefung: die Transformationen, die der Transformation direkt untergeordnet sind)

Verwandelt sich online lesen: https://riptutorial.com/de/unity3d/topic/2190/verwandelt-sich

Kapitel 39: Verwenden der Git-Quellcodeverwaltung mit Unity

Examples

Verwenden von Git Large File Storage (LFS) mit Unity

Vorwort

Git kann mit der Entwicklung von Videospielen sofort arbeiten. Der Hauptvorteil ist jedoch, dass die Versionsverwaltung großer (> 5 MB) Mediendateien langfristig ein Problem darstellen kann, da Ihr Commit-Verlauf aufgebläht wird - Git wurde ursprünglich nicht für die Versionsverwaltung von Binärdateien entwickelt.

Die gute Nachricht ist, dass GitHub seit Mitte 2015 ein Plug-In für Git namens LFS veröffentlicht , das sich direkt mit diesem Problem befasst. Sie können jetzt große Binärdateien einfach und effizient versionieren!

Schließlich konzentriert sich diese Dokumentation auf die spezifischen Anforderungen und Informationen, die erforderlich sind, um sicherzustellen, dass Ihr Git-Leben mit der Entwicklung von Videospielen gut funktioniert. Dieses Handbuch behandelt nicht die Verwendung von Git selbst.

Git & Git-LFS installieren

Ihnen als Entwickler stehen eine Reihe von Optionen zur Verfügung. Als erste Wahl können Sie die Kern-Git-Befehlszeile installieren oder eine der gängigen Git-GUI-Anwendungen für Sie erledigen lassen.

Option 1: Verwenden Sie eine Git-GUI-Anwendung

Dies ist wirklich eine persönliche Präferenz, da es viele Optionen gibt, was Git-GUI angeht oder ob überhaupt eine GUI verwendet wird. Sie haben eine Reihe von Anwendungen zur Auswahl, hier sind 3 der beliebtesten:

- Sourcetree (kostenlos)
- Github Desktop (kostenlos)
- SmartGit (Commerical)

Nachdem Sie Ihre Anwendung installiert haben, gehen Sie bitte zu google und befolgen Sie die Anweisungen, um sicherzustellen, dass sie für Git-LFS eingerichtet ist. Dieser Schritt wird in diesem Handbuch übersprungen, da er anwendungsspezifisch ist.

Option 2: Installieren Sie Git & Git-LFS

Das ist ziemlich einfach - Installieren Sie Git . Dann. Installieren Sie Git LFS .

Git Large File Storage für Ihr Projekt konfigurieren

Wenn Sie das Git LFS-Plugin verwenden, um Binärdateien besser zu unterstützen, müssen Sie einige Dateitypen festlegen, die von Git LFS verwaltet werden sollen. Fügen Sie Ihrer `.gitattributes` Datei im Stammverzeichnis Ihres Repositorys Folgendes `.gitattributes` , um gängige Binärdateien zu unterstützen, die in Unity-Projekten verwendet werden:

```
# Image formats:
*.tga filter=lfs diff=lfs merge=lfs -text
*.png filter=lfs diff=lfs merge=lfs -text
*.tif filter=lfs diff=lfs merge=lfs -text
*.jpg filter=lfs diff=lfs merge=lfs -text
*.gif filter=lfs diff=lfs merge=lfs -text
*.psd filter=lfs diff=lfs merge=lfs -text

# Audio formats:
*.mp3 filter=lfs diff=lfs merge=lfs -text
*.wav filter=lfs diff=lfs merge=lfs -text
*.aiff filter=lfs diff=lfs merge=lfs -text

# 3D model formats:
*.fbx filter=lfs diff=lfs merge=lfs -text
*.obj filter=lfs diff=lfs merge=lfs -text

# Unity formats:
*.sbsar filter=lfs diff=lfs merge=lfs -text
*.unity filter=lfs diff=lfs merge=lfs -text

# Other binary formats
*.dll filter=lfs diff=lfs merge=lfs -text
```

Einrichten eines Git-Repositorys für Unity

Beim Initialisieren eines Git-Repositorys für die Unity-Entwicklung müssen einige Dinge getan werden.

Ordner ignorieren

Im Repository sollte nicht alles versioniert werden. Sie können die untenstehende Vorlage zu Ihrer `.gitignore` Datei im Stammverzeichnis Ihres Repositorys hinzufügen. Alternativ können Sie die Open Source Unity .gitignore auf GitHub überprüfen und alternativ mit gitignore.io eine Unity generieren .

```
# Unity Generated
[Tt]emp/
[Ll]ibrary/
[Oo]bj/

# Unity3D Generated File On Crash Reports
sysinfo.txt

# Visual Studio / MonoDevelop Generated
ExportedObj/
obj/
*.csproj
*.unityproj
*.sln
*.suo
*.tmp
*.user
*.userprefs
*.pidb
*.booproj
*.svd

# OS Generated
desktop.ini
.DS_Store
.DS_Store?
.Spotlight-V100
.Trashes
ehthumbs.db
Thumbs.db
```

Weitere Informationen zum Einrichten einer .gitignore-Datei finden Sie hier .

Unity-Projekteinstellungen

Standardmäßig werden Unity-Projekte nicht für die korrekte Unterstützung der Versionsverwaltung eingerichtet.

1. (Überspringen Sie diesen Schritt in v4.5 und höher.) Aktivieren Sie die Option `External` in `Unity → Preferences → Packages → Repository` .
2. Wechseln Sie zu `Visible Meta Files` `Edit → Project Settings → Editor → Version Control Mode` .
3. Wechseln Sie unter `Edit → Project Settings → Editor → Asset Serialization Mode` zu `Force Text` `Edit → Project Settings → Editor → Asset Serialization Mode` .
4. Speichern Sie die Szene und das Projekt im Menü `File` .

Zusätzliche Konfiguration

Eine der wenigen Ärgernisse, die man bei der Verwendung von Git mit Unity-Projekten hat, ist, dass Git keine Verzeichnisse interessiert und leere Verzeichnisse nach dem Entfernen von Dateien glücklich hinterlassen wird. Unity erstellt `*.meta` Dateien für diese Verzeichnisse und kann

zu `*.meta` zwischen den Teammitgliedern führen, wenn Git-Commits diese Metadateien hinzufügen und entfernen.

Fügen Sie diesen Git-Post-Merge-Hook für Repositorys mit Unity-Projekten in den Ordner `/.git/hooks/` . Nach jedem Git-Pull / Merge-Vorgang wird geprüft, welche Dateien entfernt wurden, ob das Verzeichnis, in dem es sich befand, leer ist und ob es gelöscht wird.

Szenen und Prefabs zusammenführen

Ein häufiges Problem bei der Arbeit mit Unity ist, wenn zwei oder mehr Entwickler eine Unity-Szene oder ein Prefab (* .unity-Dateien) ändern. Git weiß nicht, wie sie korrekt aus der Box zusammengefügt werden sollen. Glücklicherweise hat das Unity-Team ein Tool namens SmartMerge bereitgestellt, das die einfache Zusammenführung automatisch macht. Das erste , was zu tun ist , die folgenden Zeilen zu Ihrer hinzufügen `.git` oder `.gitconfig` - Datei: (Windows: `%USERPROFILE%\.gitconfig` , Linux / Mac OS X: `~/.gitconfig`)

```
[merge]
tool = unityyamlmerge

[mergetool "unityyamlmerge"]
trustExitCode = false
cmd = '<path to UnityYAMLMerge>' merge -p "$BASE" "$REMOTE" "$LOCAL" "$MERGED"
```

Unter **Windows** lautet der Pfad zu UnityYAMLMerge:

```
C:\Program Files\Unity\Editor\Data\Tools\UnityYAMLMerge.exe
```

oder

```
C:\Program Files (x86)\Unity\Editor\Data\Tools\UnityYAMLMerge.exe
```

und unter **MacOSX** :

```
/Applications/Unity/Unity.app/Contents/Tools/UnityYAMLMerge
```

Sobald dies erledigt ist, steht das Mergetool zur Verfügung, wenn Konflikte während der Zusammenführung auftreten. Vergessen Sie nicht, `git mergetool` manuell `git mergetool` , um UnityYAMLMerge auszulösen.

Verwenden der Git-Quellcodeverwaltung mit Unity online lesen:
https://riptutorial.com/de/unity3d/topic/2195/verwenden-der-git-quellcodeverwaltung-mit-unity

Kapitel 40: Virtuelle Realität (VR)

Examples

VR-Plattformen

Es gibt zwei Hauptplattformen in VR, eine mobile Plattform, wie **Google Cardboard** , **Samsung GearVR** , die andere ist eine PC-Plattform, wie **HTC Vive, Oculus, PS VR** ...

Unity unterstützt offiziell den **Oculus Rift** , **Google Carboard** , die **Steam VR** , die **Playstation VR** , die **Gear VR** und die **Microsoft Hololens** .

Die meisten Plattformen haben ihre eigene Unterstützung und Sdk. Normalerweise müssen Sie die SDK zunächst als Erweiterung für Unity herunterladen.

SDKs:

- Google Cardboard
- Tagtraum-Plattform
- Samsung GearVR (integriert seit Unity 5.3)
- Oculus Rift
- HTC Vive / Open VR
- Microsoft Hololens

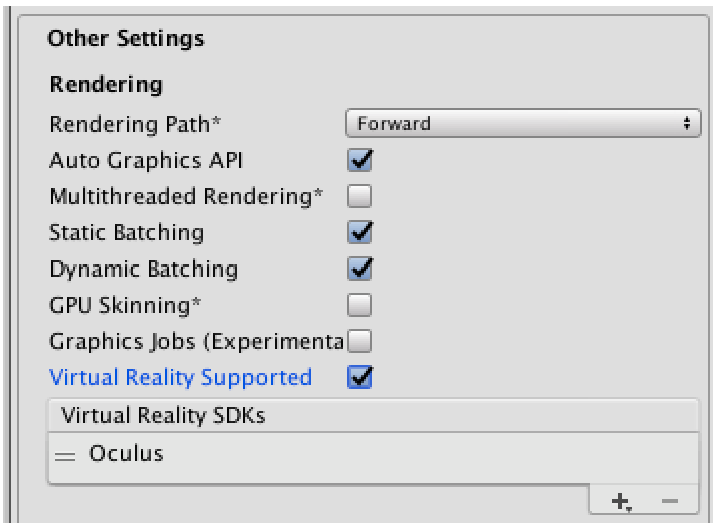
Dokumentation:

- Google Cardboard / Daydream
- Samsung GearVR
- Oculus Rift
- HTC Vive
- Microsoft Hololens

Aktivieren der VR-Unterstützung

Öffnen Sie im Unity Editor die **Player-Einstellungen** (Bearbeiten> Projekteinstellungen> Player).

Aktivieren Sie unter **Andere Einstellungen die** Option *Virtual Reality Supported* .



Fügen Sie VR-Geräte für jedes Build-Ziel in der Liste der *Virtual Reality-SDKs* unter dem Kontrollkästchen hinzu oder entfernen Sie sie.

Hardware

Es gibt eine notwendige Hardwareabhängigkeit für eine VR-Anwendung, die normalerweise von der Plattform abhängt, für die Sie eine Plattform erstellen. Es gibt zwei große Kategorien für Hardwaregeräte, basierend auf ihren Bewegungsfunktionen:

1. 3 DOF (Freiheitsgrade)
2. 6 DOF (Freiheitsgrade)

3 DOF bedeutet, dass die Bewegung des Head-Mounted-Displays (HMD) in drei Dimensionen betrieben werden muss, die sich um die drei orthogonalen Achsen drehen, die um den Schwerpunkt des HMDs zentriert sind - die Längsachse, die Vertikalachse und die Horizontalachse. Bewegung um die Längsachse wird als Rollen bezeichnet, Bewegung um die Querachse wird als Nickbewegung bezeichnet und Bewegung um die senkrechte Achse wird als Gieren bezeichnet, ähnliche Prinzipien, die die Bewegung eines sich bewegenden Objekts wie eines Flugzeugs oder eines Autos bestimmen, was bedeutet, dass Sie dies auch tun werden Sie können in allen X-, Y- und Z-Richtungen durch die Bewegung Ihrer HMD in der virtuellen Umgebung sehen, Sie können jedoch nichts bewegen oder berühren (die Bewegung durch einen zusätzlichen Bluetooth-Controller ist nicht gleich).

6 DOF ermöglicht jedoch eine raumnahe Erfahrung, bei der Sie sich auch um die X-, Y- und Z-Achse abseits der Roll-, Nick- und Gierbewegungen um den Schwerpunkt bewegen können, also den Freiheitsgrad von 6.

Gegenwärtig erfordert eine VR-Raumwaage für 6 DOF eine hohe Rechenleistung mit einer High-End-Grafikkarte und Arbeitsspeicher, die Sie wahrscheinlich nicht von Ihren Standard-Laptops erhalten, und erfordert einen Desktop-Computer mit optimaler Leistung und mindestens 6 x 6 Fuß Freier Speicherplatz, während ein 3-DOF-Erlebnis nur durch ein Standard-Smartphone mit integriertem Kreisel erreicht werden kann (das bei den meisten modernen Smartphones eingebaut ist, die etwa 200 USD oder mehr kosten).

Einige gängige Geräte, die heute auf dem Markt erhältlich sind, sind:

- Oculus Rift (6 DOF)
- HTC Vive (6 DOF)
- Tagtraum (3 DOF)
- Gear VR Angetrieben von Oculus (3 DOF)
- Google Cardboard (3 DOF)

Virtuelle Realität (VR) online lesen: https://riptutorial.com/de/unity3d/topic/5787/virtuelle-realitat--vr-

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit unity3d	Alexey Shimansky, Chris McFarland, Community, Desutoroiya, driconmax, Ḟḹáḿíṅḡ ỏḿƀíé, James Radvan, josephsw, Linus Juhlin, Luís Fonseca, Maarten Bicknese, martinhodler, matiaslauriti, Mike B, Minzkraut, PlanetVaster, R.K123, S. Tarık Çetin, Skyblade, SourabhV, SP., tenpn, tim, user3071284
2	Android Plugins 101 - Eine Einführung	Venkat at Axiom Studios
3	Animation der Einheit	4444, Fiery Raccoon, Guglie
4	Anzeigenintegration	ʇolɐəz əɥʇ qoq
5	Asset Store	JakeD, Trent, zwcloud
6	Attribute	4444, Thundernerd
7	Audiosystem	R4mbi, ʇolɐəz əɥʇ qoq
8	Coroutinen	agiro, Fattie, Fehr, Giuseppe De Francesco, Problematic, Skyblade, Thulani Chivandikwa, Thundernerd, ʇolɐəz əɥʇ qoq, volvis
9	CullingGroup-API	volvis
10	Designmuster	Ian Newland
11	Eingabesystem	Programmer, Skyblade, ʇolɐəz əɥʇ qoq
12	Einheitsbeleuchtung	Ḟḹáḿíṅḡ ỏḿƀíé
13	Erweitern des Editors	Pierrick Bignet, Skyblade, Thundernerd, ʇolɐəz əɥʇ qoq, volvis
14	GameObjects finden und sammeln	Pierrick Bignet, S. Tarık Çetin, volvis
15	Implementierung der MonoBehaviour-Klasse	matiaslauriti, Skyblade, Thundernerd, user3797758
16	Importeure und (Post-)	gman, Skyblade, volvis

	Prozessoren	
17	Kollision	Ḟḷáḿíṅḡ ȯḿƀíé, jjhavokk, Xander Luciano
18	Kommunikation mit dem Server	David Martinez, devon t, Ḟḷáḿíṅḡ ȯḿƀíé, Maxim Kamalov , tim
19	Mobile Plattformen	Airwarfare, Skyblade
20	Multiplattform-Entwicklung	user3797758, volvis
21	Objekt-Pooling	Chris McFarland, Ed Marty, lase, matiaslauriti, S. Tarık Çetin, Thulani Chivandikwa, Thundernerd, ʇolɐəz əɥʇ qoq, volvis
22	Optimierung	Ed Marty, EvilTak, Ḟḷáḿíṅḡ ȯḿƀíé, Grigory, JohnTube, Skyblade, Thulani Chivandikwa, volvis
23	Physik	eunoia, Ḟḷáḿíṅḡ ȯḿƀíé, jack jay
24	Prefabs	Brandon Mintern, Dávid Florek, Ḟḷáḿíṅḡ ȯḿƀíé, gman, Gnemlock, Guglie, James Radvan, Jean Vitor, josephsw, Lich, matiaslauriti, Skyblade, Thulani Chivandikwa, ʇolɐəz əɥʇ qoq, Woltus, yummypasta
25	Quaternionen	matiaslauriti, Tiziano Coroneo, Xander Luciano, yummypasta
26	Raycast	driconmax, Meinkraft, Skyblade, user3570542, volvis, wouterrobot
27	Ressourcen	glaubergft, MadJlzz, Skyblade, Venkat at Axiom Studios
28	Schichten	Arijoon, dreadnought, Light Drake, RamenChef, Skyblade
29	ScriptableObject	volvis
30	Singletons in der Einheit	David Darias, Fehr, James Radvan, JohnTube, matiaslauriti, Maxim Kamalov, Simon Heinen, SP., Tiziano Coroneo, Umair M, volvis, Zze,
31	So verwenden Sie Asset-Pakete	Ḟḷáḿíṅḡ ȯḿƀíé
32	Sofortiges grafisches Benutzeroberflächensystem (IMGUI)	Skyblade, Soaring Code
33	Stichworte	Arijoon, Augure, glaubergft, Gnemlock, MadJlzz, Skyblade, Trent

34	Unity Profiler	Amitayu Chakraborty, ForceMagic, RamenChef, Skyblade
35	User Interface System (UI)	Hellium, matiaslauriti, Maxim Kamalov, Programmer, RamenChef, Skyblade, Umair M
36	Vector3	driconmax, Ḟļáḿíṅḡ ỏḿƀíé, Gnemlock
37	Vernetzung	David Martinez, driconmax, Rafiwui, RamenChef
38	Verwandelt sich	ADB, Jean Vitor, matiaslauriti, S. Tarık Çetin, Skyblade, Thundernerd, Xander Luciano
39	Verwenden der Git-Quellcodeverwaltung mit Unity	Commodore Yournero, Hacky, James Radvan, matiaslauriti, Max Yankov, Maxim Kamalov, Pierrick Bignet, Ricardo Amores, S. Tarık Çetin, S.Richmond, Skyblade, Thulani Chivandikwa, YsenGrimm, yummypasta
40	Virtuelle Realität (VR)	4444, Airwarfare, Guglie, pew., Pratham Sehgal, tim