HANSER

Leseprobe

Jonas Richartz

Spiele entwickeln mit Unreal Engine 4

// Programmierung mit Blueprints // Grundlagen & fortgeschrittene Techniken

ISBN (Buch): 978-3-446-44635-9

ISBN (E-Book): 978-3-446-44806-3

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-44635-9

sowie im Buchhandel.

© Carl Hanser Verlag, München

Inhalt

Vorwort XIII

1 Einleitung 1
1.1 Was brauche ich? 1
1.2 Was lerne ich? 2
1.3 Lizenzen 3
1.4 Weiterentwicklung der Engine 4

2 Erste Schritte 5
2.1 Wie fange ich an? 5
2.2 Motivation 6
2.3 Planung 7
2.4 Sicherheitskopien 7
2.5 Learning by Doing 8

3 Grundlagen 9
3.1 Installation 9
3.2 Epic Games Launcher 11
3.3 Erstellung eines Projekts 12
3.4 Oberfläche 14
3.4.1 Game/Editor View 15
3.4.1.1 Viewport 15
3.4.1.2 View Mode 17
3.4.1.3 Snapping 18
3.4.1.4 Navigation 18
3.4.2 Content Browser 19
3.4.3 World Outliner 21
3.4.4 Details 22
3.4.5 World Settings 23
3.4.6 Modes 24

3.4.6.1 Place 24
3.4.6.2 Paint 25
3.4.6.3 Landscape 26
3.4.6.4 Foliage 26
3.4.6.5 Geometry Editing 27
3.4.7 Play 27
3.5 Ausprobieren 28

4 Blueprints 29
4.1 Was sind Blueprints? 29
4.2 Das Actor-Blueprint 31
4.2.1 Der Hauptbereich 32
4.2.1.1 Viewport 33
4.2.1.2 Construction Script 34
4.2.1.3 Event Graph – Was ist ein Event? 34
4.2.2 Components 39
4.2.2.1 Viewport-Components 40
4.2.2.2 Blueprint-Components 41
4.2.3 Details 42
4.2.4 Debug-Bereich 43
4.3 Anwendungsbeispiele 45
4.3.1 Toggle 45
4.3.2 Sequenzer 46
4.3.3 Timeline 48
4.3.4 Spawn Actor 49
4.3.5 Reroute-Node 50
4.3.6 Is Valid? 50

5 Bausteine der Welt 53
5.1 Variablen 53
5.1.1 Boolean 54
5.1.2 Byte 56
5.1.3 Integer 56
5.1.4 Float 57
5.1.5 Name, String und Text 58
5.1.6 Vector 59
5.1.7 Rotator 60
5.1.8 Transform 60
5.2 Benutzen von Variablen 60
5.2.1 Variablen in Events 63
5.3 Arrays 66
5.4 Übung zu Arrays 68

6 Die Welt in 3D 71
6.1 World- und Relative-Transforms 71
6.2 Transforms in Blueprints 75
6.3 Meshes 79
6.3.1 Toolbar und Viewport 81
6.3.1.1 Sockets 83
6.3.1.2 Wireframe 85
6.3.1.3 Bounds 86
6.3.1.4 Pivot 87
6.3.1.5 Normals 88
6.3.1.6 Tangents und Binormals 88
6.3.1.7 Vertices 89
6.3.1.8 UV 90
6.3.2 Details 92
6.4 Collision 93
6.4.1 Kollisionstypen 97
6.5 Materials 99
6.5.1 Graph 100
6.5.1.1 Base Color 101
6.5.1.2 Metallic 102
6.5.1.3 Specular 103
6.5.1.4 Roughness 103
6.5.1.5 Emissive Color 104
6.5.1.6 Opacity 105
6.5.1.7 Opacity Mask 106
6.5.1.8 Normal 107
6.5.1.9 World Position Offset 108
6.5.1.10 World Displacement und Tessellation Multiplier 109
6.5.1.11 Subsurface Color 111
6.5.1.12 Clear Coat 112
6.5.1.13 Ambient Occlusion 112
6.5.1.14 Refraction 113
6.5.1.15 Pixel Depth Offset 114
6.5.2 Details 116
6.5.2.1 Physical Material 116
6.5.3 Palette 118

7 Licht und Schatten 119
7.1 Lichtarten 119
7.1.1 Directional Light 120
7.1.1.1 Light 121
7.1.1.2 Light Shaft 122
7.1.1.3 Distance Field Shadows 123
7.1.1.4 Lightmass 124

7.1.1.5 Cascaded Shadow Maps ... 124
7.1.1.6 Light Function ... 125
7.1.2 Point Light ... 126
7.1.2.1 Light ... 127
7.1.2.2 Light Profiles ... 128
7.1.3 Spot Light ... 129
7.1.4 Sky Light ... 130
7.1.4.1 Light ... 131
7.2 Lightmaps ... 132
7.2.1 Lightmass Importance Volume ... 135
7.2.2 Light Propagation Volumes ... 135
7.3 Global Illumination ... 137

8 Physik ... 139
8.1 Simulate Physics ... 139
8.1.1 Collisions ... 142
8.1.2 Physik in Blueprints ... 145
8.1.2.1 Physics Components ... 146
8.1.2.2 Physics Constraint ... 155
8.1.2.3 Physics Handle ... 159
8.1.2.4 Physics Thruster ... 162
8.1.2.5 Radial Force ... 163

9 Ein Level entsteht ... 165
9.1 BSP ... 166
9.1.1 Brush Settings ... 167
9.1.2 Surface Material ... 169
9.1.3 Geometry Editing ... 172

10 Landschaften ... 175
10.1 Landscape-Tool ... 175
10.1.1 Manage ... 176
10.1.1.1 Selection ... 180
10.1.1.2 Add ... 182
10.1.1.3 Delete ... 182
10.1.1.4 Move to Level ... 182
10.1.1.5 Change Component Size ... 183
10.1.1.6 Edit Splines ... 184
10.1.2 Sculpt ... 187
10.2 Landscape-Material ... 190
10.2.1 Layer Blend ... 192
10.2.2 Material Instance ... 194
10.2.3 Paint-Tool ... 195
10.2.4 Layer Weight ... 196

10.3 Foliage-Tool 198
10.4 Grass Output 201

11 Audio 205

11.1 Sound-Arten 205
11.1.1 Sound Cue 207
11.1.2 Sound Attenuation 211
11.1.3 Sound Class 213
11.1.4 Sound Mix 214
11.1.5 Dialogue Voice/Wave 215
11.1.6 Reverb Effect 217
11.1.7 Media Sound Wave 218

12 Partikel 219

12.1 Cascade 220
12.1.1 Emitter 221
12.1.2 Type Data 227
12.2 Ein Beispiel für Effekte 228

13 Der Character 233

13.1 Character Blueprint 233
13.1.1 Character Movement 238
13.1.2 Movement-Funktionen 240
13.1.3 Vorbereitungen für Interaktionen 242
13.1.4 Kameraeigenschaften 243
13.1.4.1 Post Process Volume 248

14 Kommunikation 251

14.1 Cast to Blueprint 251
14.2 Interface 253
14.2.1 Output 257
14.3 Reference 259
14.3.1 Alle Actors einer Klasse 262

15 User Interface 265

15.1 HUD-Klasse 265
15.2 Widgets 268
15.2.1 Canvas 269
15.2.2 Palette 272
15.2.2.1 Common 272
15.2.2.2 Input 282
15.2.2.3 Panel 285
15.2.2.4 Primitive 288
15.3 Benutzen von Widgets 289

16 Datenbanken **293**
16.1 Structs 293
16.2 Data Table 295
16.2.1 Datenbanken in Blueprints 297
16.2.2 Speichern und Laden von Daten 299

17 Animationen **303**
17.1 Skeletal Mesh 303
17.2 Skeleton 305
17.3 Animationen 307
17.3.1 Aim Offset 309
17.3.2 Blend Space 311
17.4 Animation Blueprint 312
17.4.1 Event Graph 312
17.4.2 Anim Graph 316
17.5 Retargeting 324

18 Netzwerk **329**
18.1 Grundwissen über Multiplayer 329
18.2 Replication 330
18.2.1 Events 332
18.2.2 Animationen 335
18.3 Sessions 337
18.4 OnlineSubsystem 340

19 KI **343**
19.1 Erste Schritte 343
19.2 Simple Patrouille 346
19.2.1 AIController 347
19.3 KI mit Angriff 351
19.4 Behaviour Tree/Query System 353

20 Debugging **355**
20.1 Fehlersuche 355
20.2 Optimierung 361

21 Spiel erstellen **367**

22 Ein eigenes Spiel **371**

23 Tipps und Tricks **415**
23.1 Features, die es in die vorherigen Kapitel nicht geschafft haben 415
23.1.1 Split Screen 415

23.1.2 Authority 416
23.1.3 Maus zur Welt 417
23.1.4 Enums 417
23.1.5 Audio stoppen 418
23.2 Schlusswort 419

Index 421

Vorwort

Der Traum, ein eigenes Computerspiel zu erstellen, ist mittlerweile keine Seltenheit. Aufgrund der heutigen Möglichkeiten ist dies auch leichter als je zuvor. Ein kleines Spiel kannst du in wenigen Schritten zusammenstellen, und oft ist nur deine eigene Fantasie dein Limit.

Ich möchte dir in diesem Buch die Unreal Engine 4 vorstellen und dir alle möglichen Grundlagen dazu beibringen. Außerdem verdeutliche ich dir die Gedankengänge, die während der Entwicklung erforderlich sind. Die Unreal Engine 4 ist die neueste und bis dato beste Engine von Epic Games, die dir vollkommen kostenlos zur Verfügung gestellt wird. Du brauchst keine komplizierte Programmiererfahrung, um Spiele mit dieser Engine zu entwickeln, und die Community steht dir jederzeit mit Rat und Tat beiseite!

Ich hätte nie gedacht, jemals in meinem Leben ein Buch zu schreiben, bin aber sehr froh, die Gelegenheit dafür erhalten zu haben. Ich möchte mich bei Sieglinde Schärl bedanken, die mir die Tür zu diesem Buch geöffnet und mich während des Schreibens gut begleitet hat. Weiter gilt mein Dank Sylvia Hasselbach für die weitere Betreuung, Jürgen Dubau für die Korrekturen sowie dem ganzen Team des Hanser Verlags, die alle geholfen haben, dieses Buch zustande zu bringen.

Sehr dankbar bin ich meiner Mutter, meinem Vater und meinem Bruder für ihre Unterstützung während der Monate des Schreibens. Ein besonderes Dankeschön geht an Christian Albrecht für seine Unterstützung und die Erstellung meiner Website.

Nicht zu vergessen der Dank an den Community Manager Chance Ivey sowie alle bei Epic Games für die fantastische Engine. Last, but not least danke ich meiner Community auf YouTube für die netten Kommentare, die mich überhaupt motiviert haben, mich an ein solches Buch zu wagen! Wenn mein Buch euch allen hilft, tolle Spiele zu erstellen, hat sich die Mühe gelohnt!

Leichlingen im Februar 2016
Jonas Richartz

4 Blueprints

In den vorangegangenen Kapiteln und bei deinen ersten eigenen Versuchen ist dir der Name *Blueprint* schon oft begegnet. In diesem Kapitel werden ich dir die Grundlagen und das Verständnis, was genau Blueprints sind, erklären. Aber dies ist nicht das einzige Kapitel über Blueprints, es ist nur der Anfang. Fast jedes Objekt ist auch ein Blueprint: dein Spieler, eine Tür, die man öffnen kann, oder die Steuerung deiner Animationen – alles wird mit Blueprints erledigt.

4.1 Was sind Blueprints?

Blueprints sind im Grunde nichts anderes als C++-Code-Klassen, nur dass man hierbei mit sogenannten **Nodes** arbeitet – anstatt sich durch viele Zeilen Code zu quälen, was besonders für Anfänger schnell unübersichtlich wird. Nodes kann man mit Bausteinen vergleichen. Man baut sich die gesamte Logik aus vielen verschiedenen Bausteinen zusammen. Will man einen Lichtschalter mit Licht einbauen, kann man sich die Logik so vorstellen:

Schalter wird betätigt → Ist das Licht an? → Wenn ja, ausschalten/Wenn nein, einschalten

Programmieren und Blueprints gehen nach genau diesem Prinzip vor: Einer Aktion folgt eine Reaktion. Aber im Gegensatz zum normalen Programmieren mit viel C/C++-Code setzt du die Blueprint-Bausteine ein, genau wie es die Logik vorgibt, mit einfachen Verbindungen zueinander. Ein Baustein im obigen Beispiel ist, dass der Schalter betätigt wird. Darauf folgt ein Baustein, der fragt, ob das Licht an oder aus ist, gefolgt von einem letzten Baustein, der das Licht dementsprechend verändert.

Klingt logisch, oder? Gehen wir noch einen Schritt zurück, und ich zeige dir ein konkretes Beispiel, wie man eine einfache Textausgabe macht. Dabei werde ich Blueprints mit einfachem C-Code vergleichen.

Wenn du dich schon ein wenig mit C auskennst, wird dir der kommende Code-Ausschnitt bekannt vorkommen. Wenn nicht, ist das aber nicht schlimm.

Listing 4.1 Einfaches C-Beispiel

```
#include <stdio.h>

int main(void) {

printf("Hallo Welt");

}
```

Hierbei handelt es sich um eine einfache Textausgabe. Auf dem Bildschirm wird der Text *Hallo Welt* angezeigt, und sonst passiert nichts. In der Unreal Engine 4 und Blueprints passiert das ganz ähnlich, aber hierbei braucht man, wie anfangs erwähnt, keinen Code.

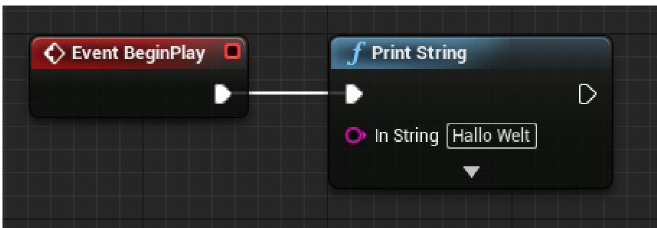

Bild 4.1 So sieht das Ganze in Blueprints aus

Wie man schnell sehen kann, gibt es ein sogenanntes *Event* namens *BeginPlay*. Existiert das jeweilige Blueprint in deinem Level, so wird anfangs immer das *Event* aufgerufen, ähnlich wie bei `int main` im C-Beispiel. Daraufhin wird mit *Print String* ein Text auf den Bildschirm ausgegeben, und genau wie im C-Beispiel wird die Welt mit „Hallo“ begrüßt.

Im direkten Vergleich kannst du sehen, wie man die Bausteine in Blueprints benutzt. Die Logik bleibt die gleiche. Mithilfe der Bausteine kannst du nahezu alles mit Blueprints programmieren, ohne dabei eine einzige Zeile Code zu schreiben. Aus eigener Erfahrung kann ich sagen, dass mir das Arbeiten mit Blueprints mehr Spaß macht, als Code zu schreiben, und ich kann auch deutlich schneller arbeiten. Das ist natürlich alles Geschmackssache, aber ich hoffe, dass ich dich im Verlauf des Buches von den Vorteilen von Blueprints überzeugen kann.

4.2 Das Actor-Blueprint

Bevor wir uns tiefer in die Logik von Blueprints und der Programmierung stürzen, schauen wir uns erst einmal an, wie so ein Blueprint standardmäßig aussieht.

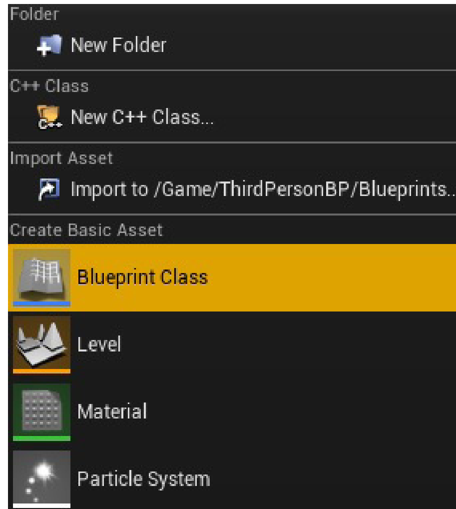

Bild 4.2
Der erste Schritt zum Erstellen eines Blueprints

Mit einem Rechtsklick im Content Browser in einem Ordner deiner Wahl kannst du eine Blueprint-Klasse erstellen: einfach auf *Blueprint Class* klicken, und schon öffnet sich ein neues Fenster.

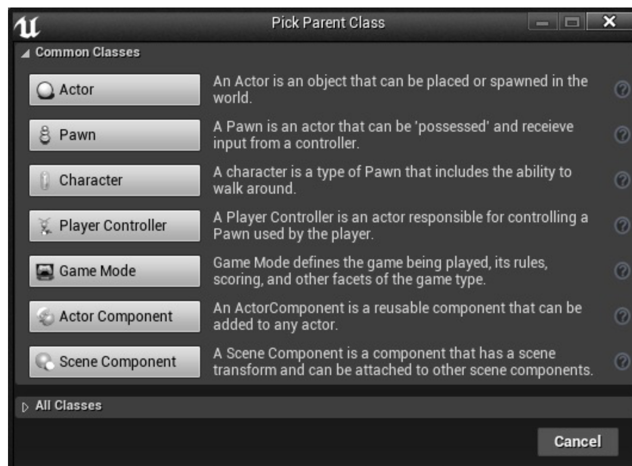

Bild 4.3
Die gängigsten Blueprints-Klassen

Es gibt unzählige verschiedene Blueprint-Klassen mit verschiedenen einmaligen Funktionen. Wir kümmern uns erstmal nur um das **Actor**-Blueprint. Diese Variante wirst du am meisten nutzen, um deinem Spiel Leben einzuhauchen. Deswegen erstellen wir auch erst einmal einen Actor.

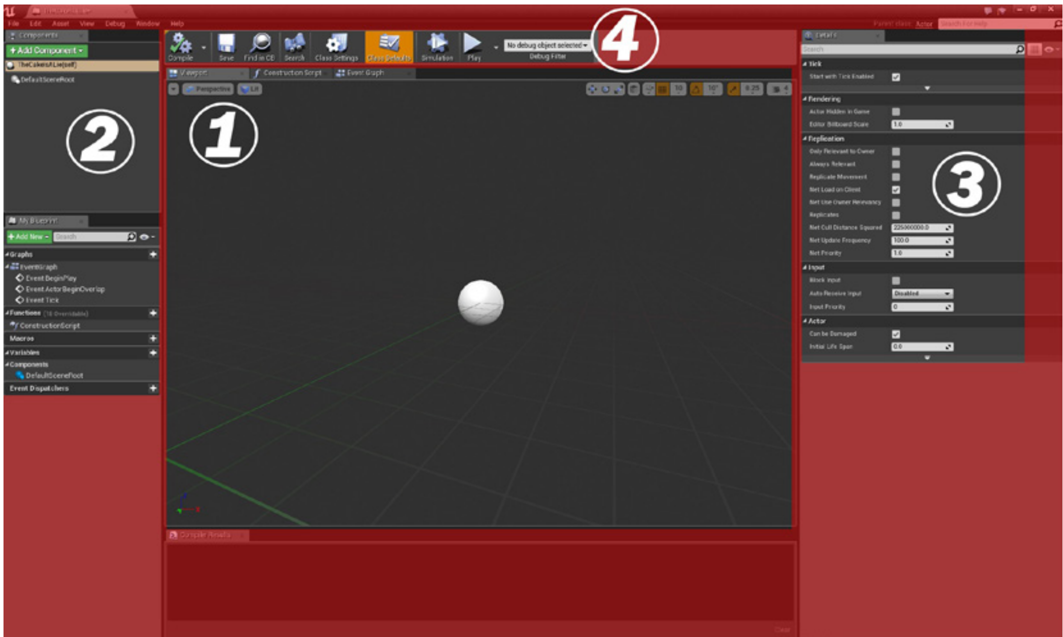

Bild 4.4 Übersicht eines Actor-Blueprints

- **Bereich 1: Viewport/ConstructionScript/EventGraph:** In diesem Bereich wirst du dich die meiste Zeit aufhalten.
- **Bereich 2: Components:** Hier werden alle Objekte, Variablen, Funktionen und vieles mehr aufgeführt, die sich momentan in deinem Blueprint befinden. Auf Add Component können auch neue Inhalte deinem Blueprint hinzugefügt werden.
- **Bereich 3: Details:** Wie gewohnt, bekommt man im *Details-Bereich* alle Informationen zum momentan ausgewählten Objekt bzw. Variable oder, falls ausgewählt, einige Standardeigenschaften des Blueprints.
- **Bereich 4: Debug:** Einige Optionen zum sogenannten Debuggen, Suche und zum direkten Start des Spiels aus dem Blueprint heraus.

4.2.1 Der Hauptbereich

Da wir uns die meiste Zeit im Hauptbereich (Bild 4.4, Bereich 1) aufhalten werden, schauen wir uns den als erstes an.

Bild 4.5 Die drei wichtigsten Bereiche

Es gibt drei Unterbereiche, die alle sehr wichtig für den täglichen Gebrauch sind. Wir werden uns alle nacheinander grob anschauen:

- **Viewport:** Besteht unser Blueprint aus ein oder mehreren Objekten, kannst du sie hier verändern und so positionieren, wie du es für richtig hältst.
- **Construction Script:** Hier wird es ganz interessant. *Construction Script* ist eine Funktion, die immer ausgeführt wird, sobald sich das Blueprint in deinem Level verändert, während du im Editor bist. Beim Spielen selbst wird das Script nicht ausgeführt.
- **Event Graph:** Alle anderen Funktionen und der Kern der Blueprint-Programmierung befinden sich im *Event Graph*.

Nach dieser kleinen Übersicht kehren wir nun zum Blueprint-Programmieren zurück.

4.2.1.1 Viewport

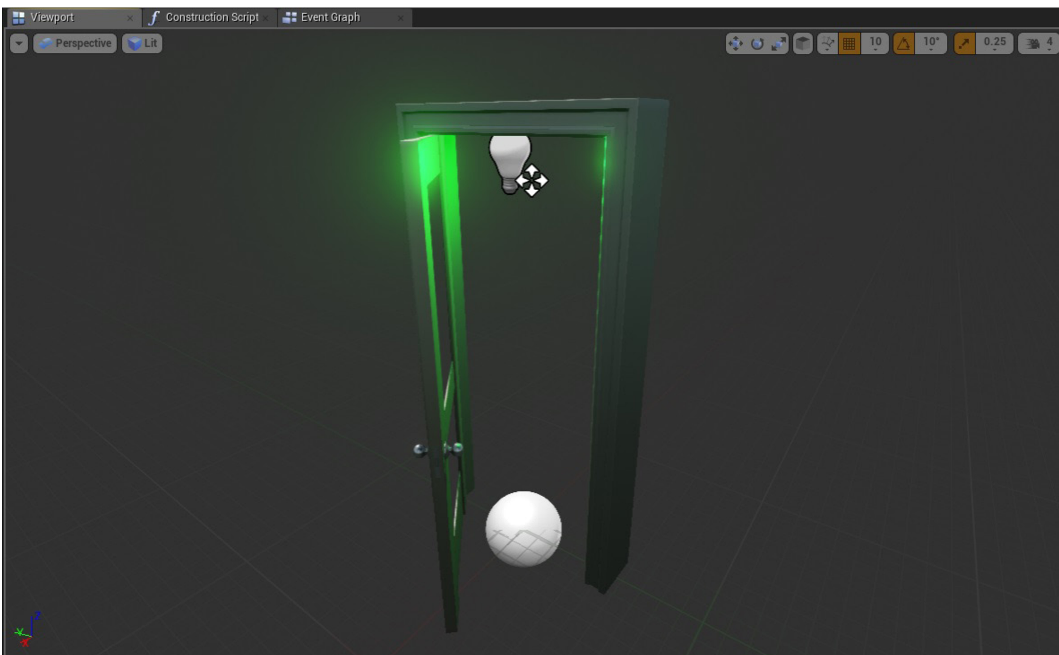

Bild 4.6 Beispiel-Viewport eines Blueprints

Im **Viewport** kannst du dir alle Inhalte des Blueprints ansehen und nach Belieben ausrichten. Hierbei handelt es sich natürlich nur um Inhalte die wir auch sehen können und die im Components-Bereich hinzugefügt wurden. Dazu später mehr.

Wie du sehen kannst, besteht dieses Viewport-Beispiel aus vier Komponenten bzw. Inhalten:

1. Scene Root
2. Türrahmen
3. Tür
4. Licht

Der **Scene Root** ist die große weiße Kugel und symbolisiert in der Regel den absoluten 0-Punkt im Blueprint, von dem der Rest der Komponenten ausgeht. Dieser existiert standardmäßig in jedem Actor-Blueprint, kann aber auch mit anderen Objekten ersetzt werden.

Türrahmen und Tür sind beides einfache statische Objekte, die ich im Blueprint platziert habe. Mit diesem Blueprint könnte ich jetzt die Funktion einbauen, diese Tür auf- und zumachen zu können. Das Tolle an Blueprints ist, dass du die Logik nur einmal machen musst und das Blueprint dann immer wieder verwenden kannst. Somit müsstest du die Tür nur einmal bauen und kannst sie verwenden, so oft du willst.

Das grüne Licht habe ich hinzugefügt, um zu zeigen, dass man wirklich alles in Blueprints einfügen kann. Wenn du möchtest, kannst du auch andere Blueprints hinzufügen oder Partikeleffekte, Sounds und vieles mehr.

4.2.1.2 Construction Script

Das **Construction Script** ist im Prinzip ähnlich wie der Event Graph, hat jedoch ein paar Restriktionen. Dieser Bereich wird immer ausgeführt, wenn sich das Blueprint im Editor verändert, sei es die Umstellung einer Variablen oder gar das Bewegen des Blueprints in deinem Level. Wichtig zu wissen ist, dass dieser Bereich nicht beim Spielen selber ausgeführt wird, sondern nur beim Erstellen, sprich beim Konstruieren des Levels. Es gibt einige nützliche Anwendungsbereiche für das **Construction Script**, aber dazu werden wir später in einem anderen Kapitel kommen.

4.2.1.3 Event Graph – Was ist ein Event?

Beim Aufruf des Event Graphs werden dir standardmäßig drei sogenannte Events vorgegeben. Ein **Event** könnte man aus der Sicht eines Programmierers auch als eine Funktion ohne Rückgabewert betrachten. Für Nicht-Programmierer ist ein Event ein Punkt, von dem aus die Logik anfängt. Diese kann entweder durch ein bestimmtes Ereignis aufgerufen werden oder man ruft das Event selbst auf.

Stellen wir uns ein Event als Knopf vor, so wird das Event mit allem, was dahinter steht, ausgeführt, wenn der Knopf gedrückt wird. Eine Funktion ist so ähnlich, wobei man im Gegensatz zu Events keinen automatischen Rückgabewert geben kann. Wenn ich also eine Funktion habe, in der ich überprüfen will, ob das Licht an oder aus ist, kann ich das Ergebnis als sogenannten Rückgabewert ausgeben. So könnte man in einem anderen Blueprint diese Funktion aufrufen und dort dann den Rückgabewert überprüfen, um zu wissen, ob das Licht an oder aus ist. Ein Event geht aber grundsätzlich nur in eine Richtung und kommuniziert nicht zurück von dem Ort, an dem es aufgerufen wurde. Zum Glück gibt es da aber dennoch Möglichkeiten, zurück zu kommunizieren, das ist aber ein wenig komplizierter. Mehr zu Funktionen später, kommen wir nun erst einmal wieder auf die Events zu sprechen.

Die Standard-Events

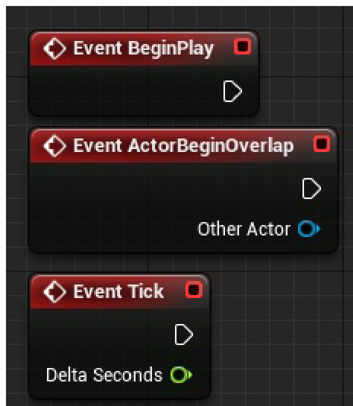

Bild 4.7
Die Standard-Events

- **Event BeginPlay:** Wird ausgeführt, sobald das Spiel mit dem Blueprint startet oder du das Blueprint während der Laufzeit erstellst.
- **Event ActorBeginOverlap:** Ist in deinem Blueprint ein Objekt mit Kollision, wird das Event ausgelöst, sobald es ein anderes Objekt mit Kollision überlappt. Zu dem Event gehört, wie auf den Bild 4.7 zu sehen, der **Other Actor**. **Other Actor** bezieht sich auf das Blueprint, welches das Event ausgelöst hat, also sprich: wer das aktuelle Blueprint überlappt hat.
- **Event-Tick:** Wird bei jedem **Tick** ausgeführt. In der Unreal Engine 4 bezieht sich ein **Tick** auf die momentane *FPS*. Je performanter dein Spiel läuft, desto öfter wird das Event ausgeführt. Zu dem Event gehört, wie auf den Bild 4.7 zu sehen, die **Delta Seconds**. Delta Seconds bezieht sich auf die Zeit von einem zum anderen **Tick**.

HINWEIS: Event-Tick kann sehr nützlich sein, aber man muss sehr aufpassen, wann und wie man das Event benutzt. Ein paar dieser Events sind gleichzeitig verkraftbar, aber wenn du es mit den Event-Ticks übertreibst, kann die Performance in deinem Spiel schnell in den Keller wandern. Also immer schön aufpassen und wenn möglich, etwas anderes benutzen!

Es gibt noch eine Vielzahl an unterschiedlichen Events mit den unterschiedlichsten Funktionen, dies wäre aber definitiv zu viel für den Anfang. Wir werden uns in den kommenden Kapiteln nach und nach weitere Events anschauen, aber für den Anfang sollte das erst einmal reichen.

Aber anstelle nur von diesen Pool an Events zu abhängig zu sein, kannst du dir auch ein eigenes Event erstellen.

Ein eigenes Event erstellen

Mit Rechtsklick im Event Graph öffnen wir das sogenannte *Context-Menü*, in dem alle verfügbaren Optionen von Nodes und Events, die man platzieren kann, dargestellt werden.

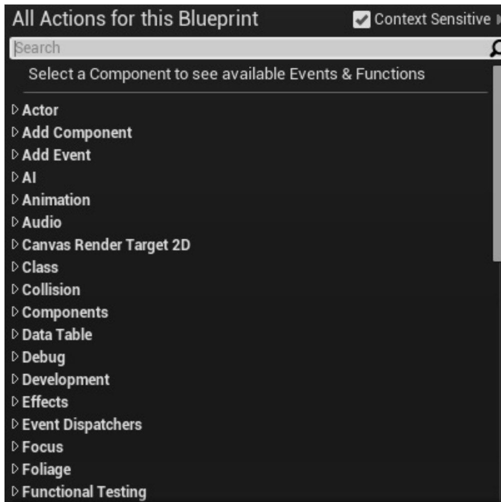

Bild 4.8
Das Context-Menü

Mit einer Suche nach *Add Custom Event* und anschließendem **Enter** oder mit einem Mausklick erstellst du dein eigenes *Custom Event*, dem du auch prompt einen Namen geben kannst.

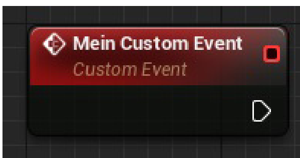

Bild 4.9
Ein wunderschönes Event

Mit unserem erstellten Event kannst du ein *Print String* hinzufügen, wie im eingangs erwähnten Beispiel beim Bild 4.1 Einfach mit einem Mausklick auf den Pfeil bei deinem Custom Event klicken, ziehen und loslassen. Damit kannst du eine neue **Node** erstellen, und beide werden sich automatisch nach der Erstellung verbinden.

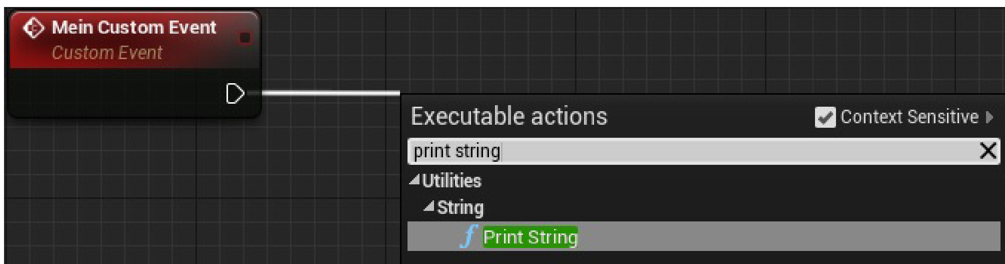

Bild 4.10 So sollte es aussehen

Nachdem du nach *Print String* gesucht und es hinzugefügt hast, sieht der ganze Spaß dann wie folgt aus:

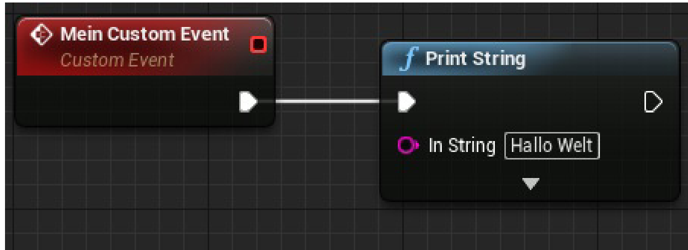

Bild 4.11 Das erste funktionsfähige Custom Event

Sehr gut, nun haben wir ein eigenes Event, welches beim Aufruf den Text *Hallo Welt* ausgibt. Aber das ist nun genau die Frage: Wie rufe ich mein eigenes Event auf? Ganz einfach, wir nehmen oder erstellen das **Event BeginPlay** und rufen unser Event in einer neuen **Node** auf. Dafür müssen wir von **Event BeginPlay** aus nach den Namen unseren erstellten Events suchen. Nur noch per Klick bestätigen, und schon haben wir ein Event, welches ein anderes Event aufruft.

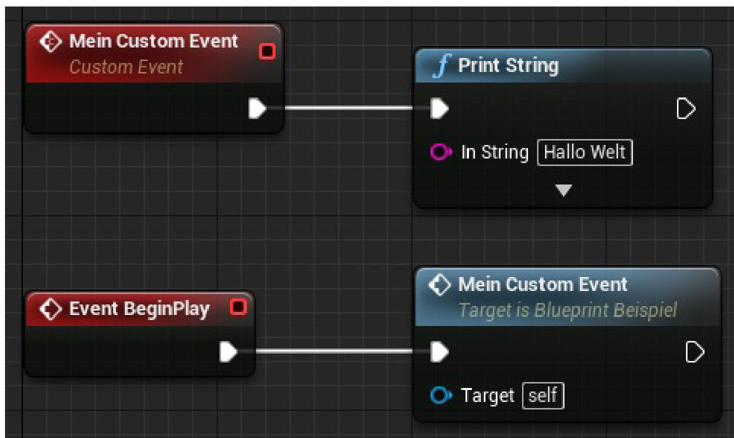

Bild 4.12 Mein Custom Event wird beim Start ausgeführt

Als Erstes wird **Event BeginPlay** ausgeführt und geht direkt zur nächsten Node namens *Mein Custom Event*. Das **Target** (Ziel) ist, wie man sehen kann, das Blueprint selbst, zu erkennen am *self*. Man kann dementsprechend Events in anderen Blueprints ausführen, wenn einem das Ziel bekannt ist. In *Mein Custom Event* wird direkt **Print String** ausgeführt, welches den Wert *Hallo Welt* ausgibt.

Platzieren wir unser Blueprint in das Spiel und drücken auf *Play,* wird also auf dem Bildschirm *Hallo Welt* ausgegeben. Vergleichbar wäre das in etwa mit folgendem C-Code.

Listing 4.2 Das gleiche Beispiel ähnlich in C

```
#include <stdio.h>

int main(void) {

MeineCustomFunktion ();

}
void MeineCustomFunktion(){

printf("Hallo Welt");

}
```

HINWEIS: Die gezeigten Codes dienen nur der Veranschaulichung und sind nicht eins zu eins mit Blueprints vergleichbar. Es geht mir nur um die Logik, die dahinter steht.

Auch im C-Beispiel wird die `int main`-Funktion aufgerufen, in der die `MeineCustomFunktion` aufgerufen wird. Es gibt in der Unreal-Engine ein paar Unterschiede zwischen Events und Funktionen, dies wird in *Kapitel 5* weiter erläutert.

Das waren einige der Standard-Events, doch es gibt noch einige weitere Events, die wir uns vorknöpfen werden, wenn es soweit ist.

4.2.2 Components

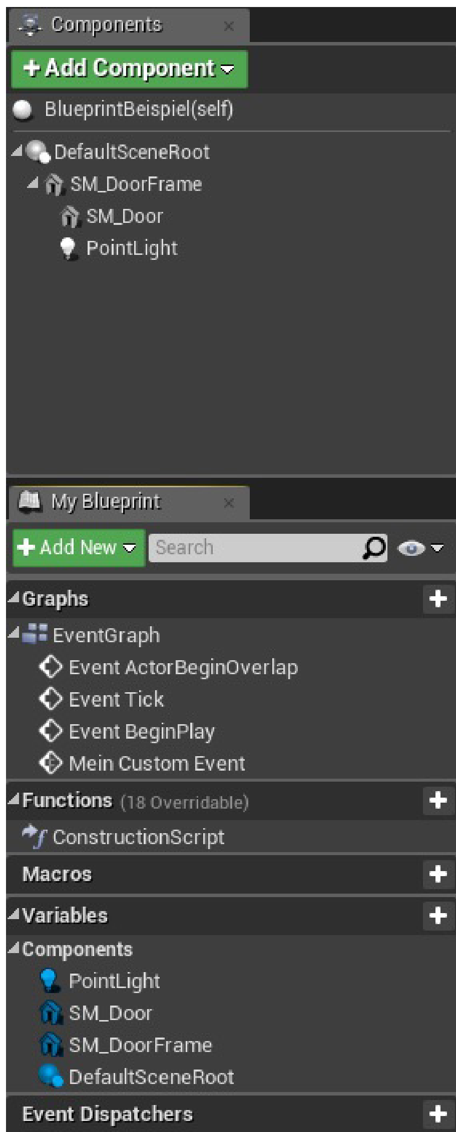

Bild 4.13
Der Components-Bereich

Der Components-Bereich ist in zwei Sektionen geteilt. Die obere Sektion beinhaltet alle Components, die es auch im Viewport zu sehen gibt und die du interaktiv dort verschieben kannst. Die untere Sektion enthält alle Informationen, Events, Funktionen, Components und Variablen des gesamten Blueprints. Wie du sehen kannst, sind alle Components aus der oberen Sektion auch in der unteren enthalten. Aber schauen wir uns einmal diese Bereiche etwas genauer an.

4.2.2.1 Viewport-Components

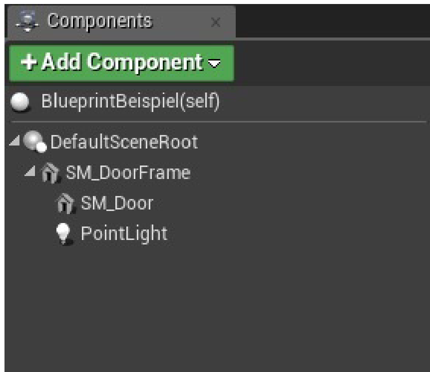

Bild 4.14
Alle Viewport-Components

Hier siehst du ein Beispiel des **Viewport-Components** von Bild 4.6. Der **Default Scene Root** ist in der Hierarchie an oberster Stelle. Im Viewport kann man den Scene Root aber nicht verschieben, egal ob es sich um den Default Scene Root oder irgendein anderes Component handelt, welches anstelle des Roots an erster Stelle steht. Wenn man also das Blueprint verschiebt, verschiebt man eigentlich den Default Scene Root bzw. die oberste Stelle in der Hierarchie.

Alle anderen Components bewegen sich demzufolge abhängig vom Root, und alle in der Hierarchie untergeordneten Components bewegen sich mit dem übergeordneten Component. Wenn wir also in diesem Beispiel das 3D-Objekt *SM_DoorFrame* verschieben, bewegen sich und rotieren *SM_Door* und *PointLight* mit dem Objekt mit, nicht aber der Scene Root oder andere Objekte, die nur dem Scene Root untergeordnet sind.

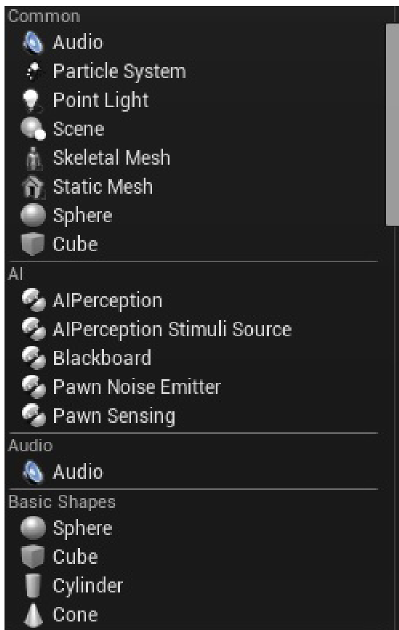

Bild 4.15
Ein Bruchteil der Auswahl

Will man somit bestimmte Teile eines Blueprints immer beisammen haben, so ist es sinnvoll, sie gut in die Hierarchiestruktur einzufügen, sodass man am Ende möglichst wenig Arbeit hat, wenn man sie modifizieren muss.

Wenn du nun auf **+ Add Component** klickst, hast du eine große Auswahl an verschiedensten Components, die du hinzufügen kannst.

Es gibt zu viele Components, um sie alle in diesem Buch durchgehen zu können. Im Laufe der kommenden Kapitel werde ich jedoch nach und nach die wichtigsten Components verwenden.

4.2.2.2 Blueprint-Components

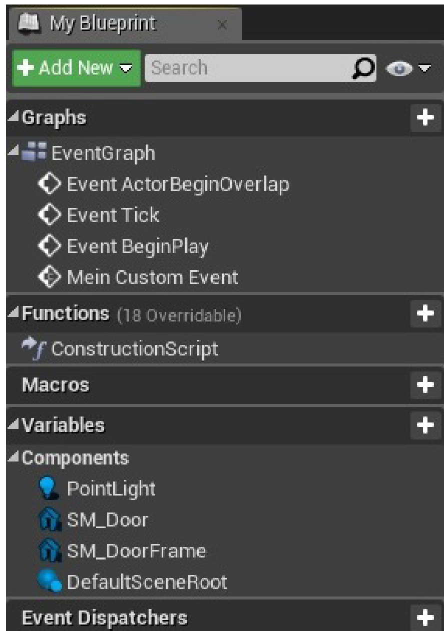

Bild 4.16
Alle Components des Blueprints im Überblick

In den Blueprint-Components werden alle Components des Blueprints sowie alle *Events, Funktionen*, *Makros* und *Variablen* aufgelistet. Von hier aus kann man einfach neue Variablen erstellen, in den *Event Graph* hineinziehen und benutzen. Auch kann man die Components aus dem Viewport hier direkt im *Event Graph* hineinziehen und benutzen.

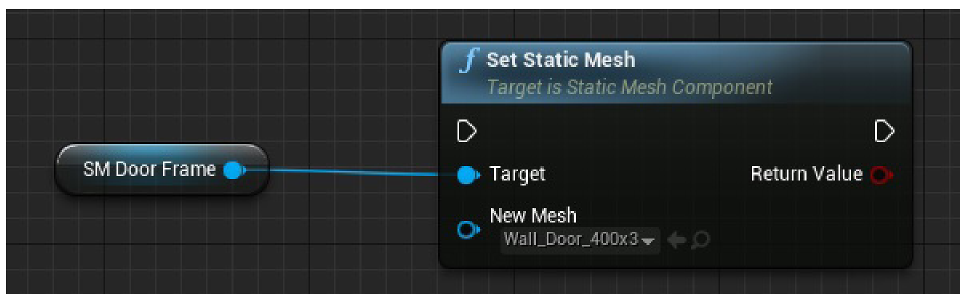

Bild 4.17 Kleines Beispiel für die Components im Viewport

Hier siehst du noch ein kleines Beispiel, was man im *Event Graph* mit einem der Viewport-Components machen kann. Ich benutze das 3D-Objekt **SM_DoorFrame** und verwandle es mit einer Funktion namens **Set Static Mesh** in ein anderes 3D-Objekt. Ein weiteres Beispiel wäre, dass man eine Wand hat und sie bei einer Explosion in eine ähnliche Wand mit einem Loch ändert. Wie genau das nun alles zu handhaben ist, wirst du im Laufe der kommenden Kapitel noch lernen.

4.2.3 Details

Der Details-Bereich ist der gleiche wie der, den ich schon in *Kapitel 3* erklärt habe. Hier gibt es eigentlich nur minimale Veränderungen, die je nach ausgewähltem Objekt bzw. Variable variieren.

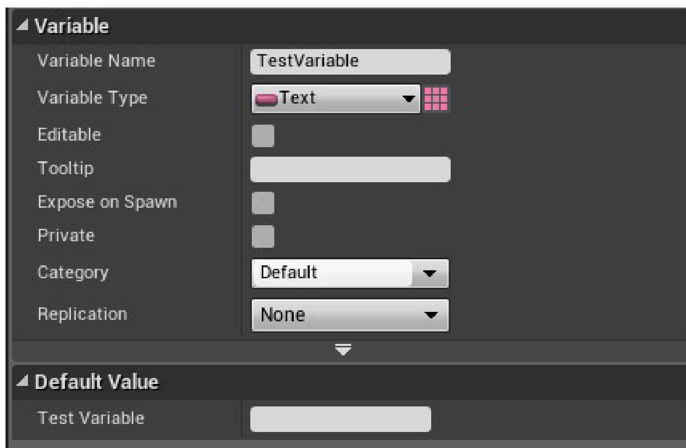

Bild 4.18 Der Details-Bereich einer Variablen

Hier nur ein kleines Beispiel einer *Variablen* des Typs Text. Eine der wichtigsten Optionen sind hier *Editable* und *Default Value*.

Editable steht dafür, dass die Variable editierbar ist. Im Prinzip sind alle Variablen editierbar, jedoch ist hierbei gemeint, dass die Variable von außerhalb des Blueprints verändert werden kann. Das heißt also, wenn du eine Text-Variable hast, die für den Namen eines Charakter stehen soll. So platzierst du mehrere Blueprints in deinem Level, die für Charaktere stehen sollen, und kannst für jedes gesetzte Blueprint andere Namen vergeben. Du setzt also ein einziges Blueprint mehrmals in deinem Level, aber gibst jeder sogenannten *Instanz* deines Blueprints einen anderen Namen. Der eine Charakter heißt somit *Ludwig*, während ein anderer *Peter* heißt, sie stammen aber vom exakt gleichen Blueprint.

Default Value ist der Standardwert einer Variablen. Beziehen wir das nun wieder auf Charaktere, wäre der *Default Value* eventuell einfach nur *Bürger*. So würde jedes platzierte Blueprint standardmäßig Bürger heißen – es sei denn, dieser würde manuell verändert werden.

4.2.4 Debug-Bereich

Kommen wir nun zu einem der interessantesten Bereiche eines Blueprints. Dieser hat weniger mit der eigentlichen Funktionalität deines Blueprints zu tun und mehr mit der Überprüfung und Suche von Fehlern.

Eventuell wirst du schon mal den Begriff **Debugging** gehört oder gelesen haben. Beim Programmieren geht man beim Debugging Zeile für Zeile durch, um genau feststellen zu können, wie und ob alles funktioniert, man kann aber auch gezielt erst an bestimmten Punkten einschreiten. Hast du zum Beispiel eine große Kette an Funktionen und bei einer passiert ein Fehler, lohnt es sich dementsprechend oft, nicht die fehlerfreien Funktionen durchzugehen, sondern man will direkt da, wo der Fehler passiert, eingreifen und genau nachschauen, was passiert.

Bild 4.19 Der „Debug"-Bereich

Aber wie funktioniert das genau in Blueprints? Zuerst einmal beinhaltet der von mir genannte Debug-Bereich einiges mehr als nur das reine Debugging. Genauer gesagt ist für das Debugging am Anfang nur der **Debug Filter** nötig. Es gibt neben den ganzen anderen Buttons nur zwei, die nicht unbedingt selbsterklärend sind. Das sind *Class Settings* und *Class Defaults*. In *Kapitel 13* werden wir uns das genauer ansehen.

Kommen wir nun zum **Debug Filter.** Sobald du ein Objekt von deinem Blueprint in dein Level platziert hast, kannst du dort eins davon auswählen. Sobald du dann auf *Play* drückst kannst du in Echtzeit sehen, was passiert.

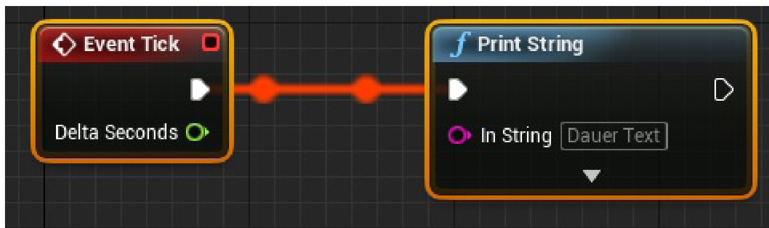

Bild 4.20 So sehen aktive Debug-Stränge aus

Alle momentan benutzen Stränge werden als rote und sich damit bewegende Punkte dargestellt. Somit kannst du genau nachverfolgen, was und wo dein Blueprint aktuell dran arbeitet, ein ziemlich cooles Feature. Aber wir wollen vielleicht auch einfach nur ab einer bestimmten **Node** anfangen zu debuggen.

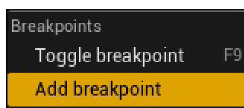

Bild 4.21 Breakpoint einfügen

Einfach mit rechts auf den gewünschten Node klicken, von welcher du mit dem Debuggen anfangen willst, und **Add breakpoint** auswählen. Das sieht danach wie folgt aus.

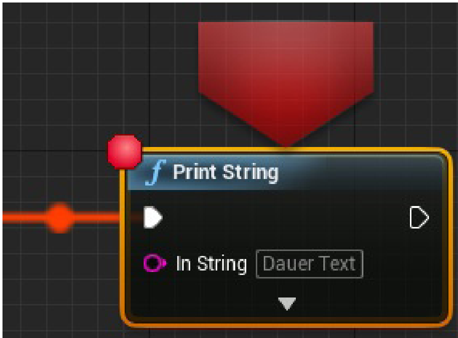

Bild 4.22
Debug Breakpoint

Nachdem wir einen **Breakpoint** hinzugefügt haben und nochmals auf *Play* drücken, zentriert sich das Sichtfeld direkt auf den *Breakpoint Node,* sobald das Blueprint dort angelangt ist. Von dort aus geht es auch nicht automatisch weiter, und die *Debug-Leiste* hat sich auch verändert.

Bild 4.23 Die erweiterte Debug-Leiste

Im erweiterten Debug-Bereich hast du nun also einige weitere nützliche Debug-Funktionen.

Resume erlaubt es Dir, einfach ganz normal weiterzumachen, bis ein weiterer Breakpoint erreicht wird.

Frame Skip springt einen Frame weiter. Da ist es ganz unterschiedlich, wo er dann herauskommt. Das kann an einigen Stellen nützlich sein, wird aber meist eher selten benutzt.

Stop beendet das laufende Spiel.

Find Node wird dich, falls du dich in deinem Blueprint oder einem anderen verirrt hast, zum aktuellen Breakpoint zurückleiten.

Step geht einen Schritt weiter. Wie man es vom Programmieren eventuell kennt, geht man damit Zeile für Zeile (oder wie in diesem Fall Baustein für Baustein) durch.

In *Kapitel 20* geht es nochmal ausführlicher um Debugging.

Jetzt haben wir erstmal einige der Grundlagen von Blueprints abgehandelt. Im nächsten Kapitel kümmern wir uns mehr um die verschiedenen Variablen.

4.3 Anwendungsbeispiele

Blueprints werden immer und überall verwendet. Fast alle besonderen Objekte sind im Prinzip auch eine Ableitung von Blueprints, und es gibt eine Unmenge an Bausteinen, die zur Verfügung stehen. Ich werde dir nun einige Beispiele zeigen, wie verschiedene Funktionen durchgeführt werden können, um dir einen kleinen Einblick zu verschaffen, wie der Aufbau der Logik dahinter ist und welche Nodes besonders häufig verwendet werden.

4.3.1 Toggle

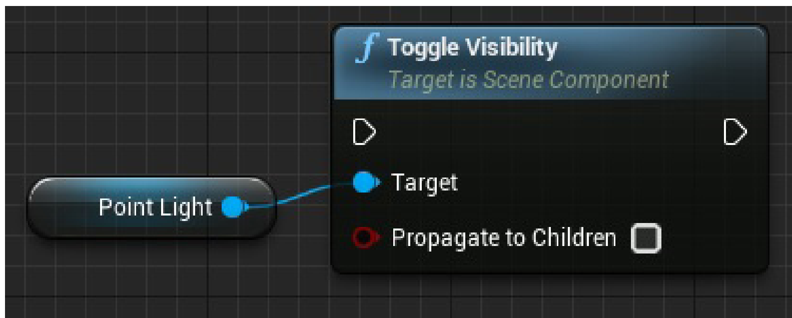

Bild 4.24 Toggle Visibility

Toggle ist eine einfache Funktion, um die Status eines Elements in das jeweilige Gegenteil zu verändern. Toggle Visibility sorgt beispielsweise dafür, dass in diesem Fall ein Licht ein- und ausgeschaltet wird. Ist das Licht sichtbar und diese Node wird ausgeführt, wird das Licht automatisch unsichtbar. Sollte das Licht schon unsichtbar sein, wird es automatisch wieder sichtbar. Es kommt ja in einigen Häusern vor, dass es mehrere Lichtschalter für die gleichen Lichter geben kann, und da würde eine Toggle-Node ungemein helfen. Jeder Lichtschalter führt einfach bei Betätigung eine Toggle-Node im Licht aus, und das Licht wird schon „wissen“, wie es sich verändern muss, sodass man sich selbst keine großen Gedanken darüber machen muss.

4.3.2 Sequenzer

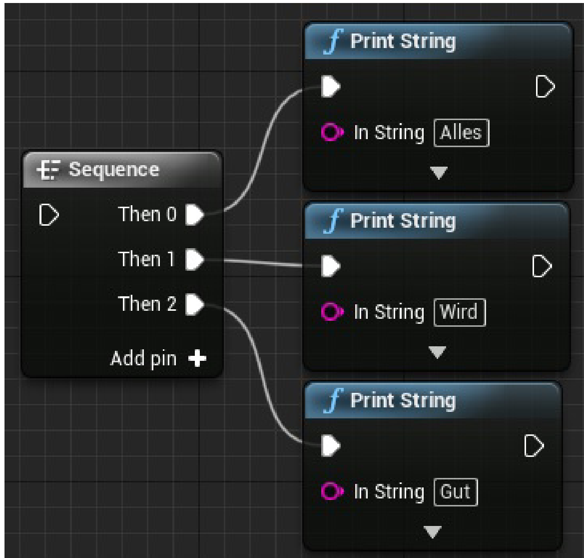

Bild 4.25
Sequence-Node

Normalerweise läuft die Logik vom Blueprint an einem Strang ab, und man hat keine Möglichkeiten, innerhalb eines Blueprints mehrere Aufgaben gleichzeitig auszuführen. Aber da kommt die Sequence-Node ins Spiel. Mit dieser Node kannst du mehrere Funktionen und Logiken anbinden, ohne alles nacheinander hineinstopfen zu müssen, was dein Blueprint immer unübersichtlicher macht. Sobald eine Sequence-Node ausgeführt wird, werden alle Ausgänge nacheinander ausgeführt, was der Übersichtlichkeit ungemein dient. Es gibt aber auch einige Funktionen, die keinen Ausgang haben, aber du willst noch mehr machen, wie beispielsweise bei einem Gate.

Gate

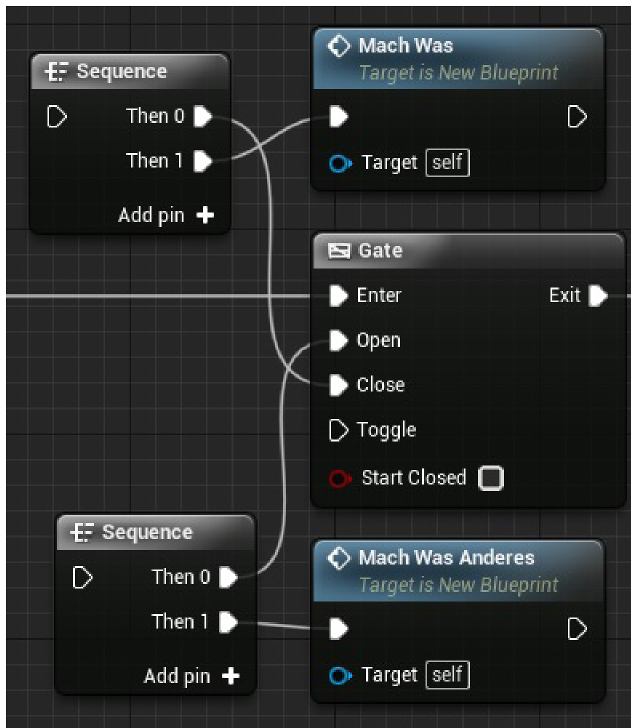

Bild 4.26
Sequence mit Gate

Ein Gate ist ein Logik-Tor, welches offen oder zu sein kann. Wenn wir also nur zu bestimmten Zeiten die Logik nach dem Gate ausführen wollen, müssen wir das Gate öffnen und schließen können. Aber dies ist auch oft nicht das Einzige, was man damit bewirken will. Dafür gibt es dann die Sequence-Node, die einem dabei hilft. Zuerst wird mit der Sequence-Node das Gate entweder geschlossen oder geöffnet und anschließend ein Event ausgeführt. Ohne die Sequence-Node müsste man also erst das Event oder die Funktionen ausführen, und am Ende soll dann das Gate verändert werden. Du kannst dir eventuell schon vorstellen, wie die ganzen Verbindungen durch das Blueprint drunter und drüber verteilt sind, was wir mit der Sequence-Node deutlich übersichtlicher gestalten können.

4.3.3 Timeline

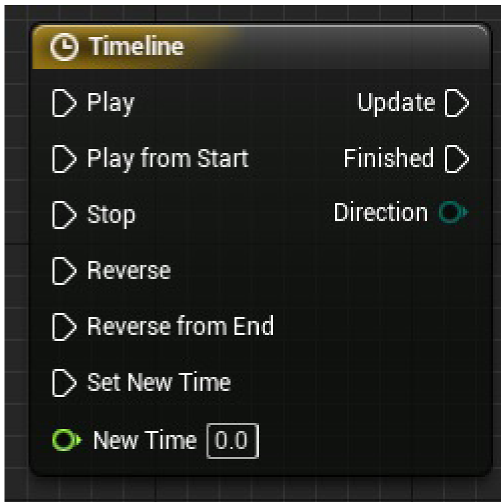

Bild 4.27 Timeline-Node

Eine weitere nützliche Node ist die Timeline-Node, welche du in deinem Blueprints erstellen kannst. Mit einem Doppelklick auf der Timeline kannst du zeitliche Angaben erstellen und dort verschiedene Tracks erstellen. Ein Track kann für Zahlenwerte, Positionsangaben, Farben und mehr stehen, welche man nach Belieben anpassen kann.

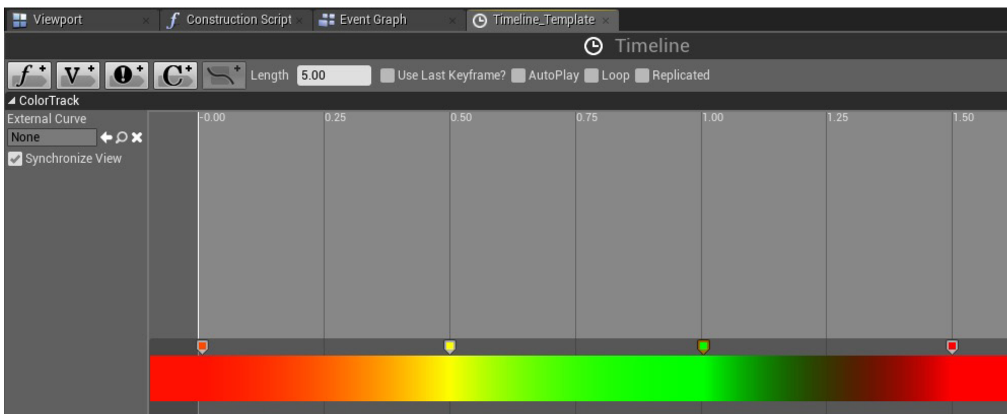

Bild 4.28 Farbänderungen in einer Timeline

Wie du sehen kannst, könnte man beispielsweise eine Farbänderung in einer Timeline einbauen und diese dann mithilfe des Update-Outputs einem Licht zuweisen. Update wird permanent ausgeführt, während die Timeline läuft.

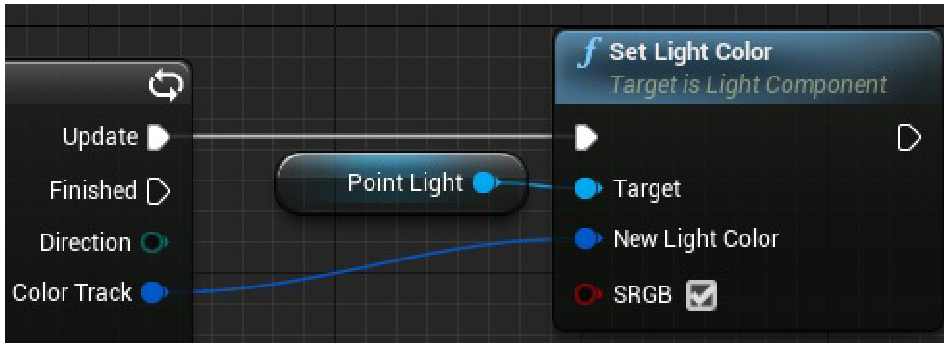

Bild 4.29 Benutzen des Update-Ausgangs

Während die Timeline aktiv ist, wird also in diesem Falle die Farbe des Lichtes geändert, bis die Timeline abgeschlossen wurde, wo dann einmalig der Finished-Output ausgeführt wird. Damit könntest du beispielsweise auch flackerndes Licht erstellen, aber es ist nicht nur bei Lichtern hilfreich. Du kannst die Werte, die herauskommen, natürlich benutzen, wie du willst.

4.3.4 Spawn Actor

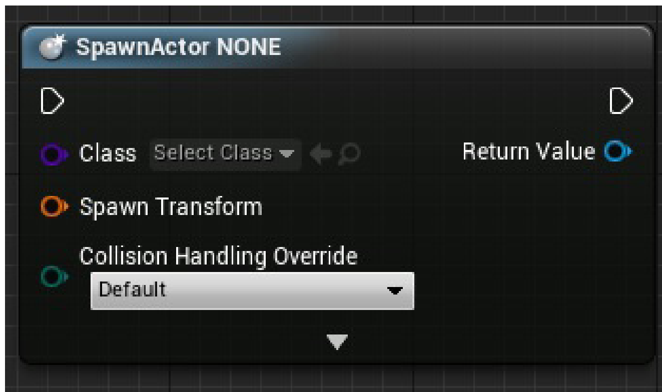

Bild 4.30 SpawnActor-Node

Mit Spawn Actor können andere Blueprints gespawnt werden. Diese Node wird sehr häufig verwendet, wenn du während des Spielens Blueprints erscheinen lassen willst. So könnte der zu spawnende Actor eine Goldmünze sein, und jedes Mal, wenn du einen Schalter betätigst, soll eine neue Goldmünze erscheinen. Wenn du also etwas erscheinen lassen willst, denk immer an die SpawnActor-Node.

4.3.5 Reroute-Node

Blueprints und Verbindungen innerhalb von Blueprints können sehr schnell unübersichtlich werden und durcheinander geraten.

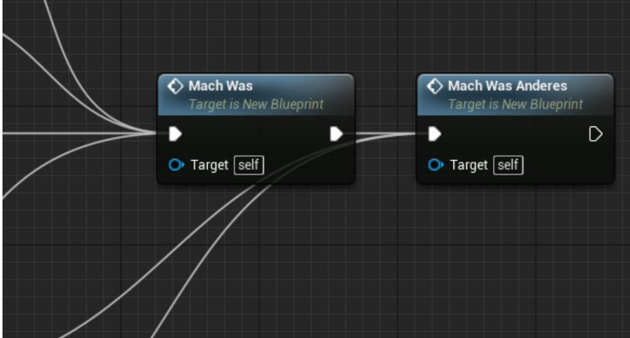

Bild 4.31
Ohne Reroute-Node

Wir wollen immer verhindern, dass unsere Blueprints unübersichtlich werden, und da kommen Reroute-Nodes zum Einsatz, um die Verbindungen zu bündeln und umzuleiten.

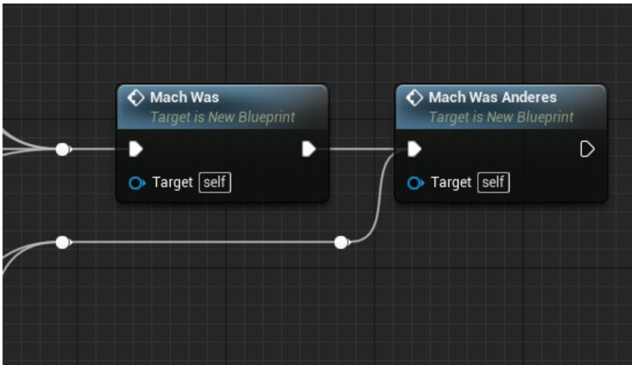

Bild 4.32
Mit Reroute-Node

Eine Reroute-Node kannst du entweder erstellen, indem du innerhalb des Context-Menüs danach suchst, oder du klickst einfach doppelt auf die jeweiligen Verbindungen. Damit lassen sich deine Blueprints schön ordentlich halten.

4.3.6 Is Valid?

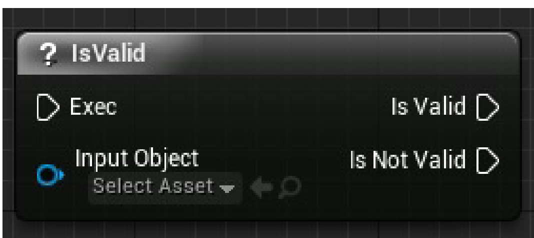

Bild 4.33
IsValid-Node

Zustände und Blueprints können sich im Laufe der Zeit verändern und gelöscht/zerstört werden. Mit der IsValid-Node kannst du überprüfen, ob das Objekt/Variable, welches du in das Input-Object einsetzt, noch existiert bzw. ob es noch gültig ist oder nicht.

Nehmen wir mal an, dass es einen Schalter gibt, der eine Kiste spawnen lässt. Die Kiste kann jedoch auch kaputtgehen und somit zerstört werden. Wenn der Schalter also betätigt wird, kann überprüft werden, ob die Kiste noch existiert oder nicht. Falls sie existieren sollte, kann die Position der Kiste zurück zum Ausgangspunkt gebracht werden. Sollte diese nicht mehr existieren, muss eine neue Kiste gespawnt werden. Wenn du dir also unsicher bist, ob etwas valide ist, dann solltest du es überprüfen.

Index

Symbole

3D 71

A

Actors 31
Add 66
AddControllerYawInput 237
AddMovementInput 237
AddOptions 283
AddToViewport 409
AI 343
AIController 344
Aim Offset 309
Alignment 170
AlwaysRelevant 330
Ambient Cubemap 246
Ambient Occlusion 112, 246
Animation Blueprint 312
Animationen 303
Animation Notifies 308, 322
Animations 307
Anim Graph 316
Anim Trail 227
Anti-Aliasing (AA) 247
APEX 147
Append 283
Apply Radial Damage 155
Array 66
ArrowComponent 234
Artificial Intelligence (AI) 343
AttachActorToComponent 314
Attenuation Radius 127, 130
Audio 205
Audio Component 418
Auto Exposure 246

B

Baked Lighting 132
Base Color (Diffuse) 101
Beam Data 227
Behaviour Tree 353
Billboard 258, 346
Binormals 88
Blendables 248
Blend Space 311, 317
Bloom 245
Blueprint-Components 41
Blueprint-Interface 253, 375
Blueprint-Kommunikation 251
Blueprints 29
– Physik 145
Bool 54
Boolean 54
Border 273
Bounds 86
Branch 55
Breakpoint 355
Brush Settings 167
BSP 166
Build Lighting Only 132
Button 273
Byte 56

C

Camera Settings 243
Canvas 269
Canvas Panel 286
CapsuleComponent 234
Cascade 219
Cascaded Shadow Maps 124
Cast to 251
Change Component Size 183
Character Blueprint 233, 312
Character Movement 234, 238
Check Box 274
Chunk Parameters 154
Circular Throbber 288
Clear Coat 112
Collision 93, 142
Collision Presets 97
Color Grading 244
ColorOverLife 400
Combo Box 282
Comment Box 363
Components 39
Conduit 318
Constraints 140, 155
ConstructionScript 402
Content Browser 20
CreatePlayer 415
Create Session 338
Create Widget 409
Cubemaps 131
Custom Event 35, 63, 298

D

Damage 150
Data Table 295
Datenbanken 293
– in Blueprints 297
Debug Filter 358
Debugging 43, 355
Density 117
Depth of Field 246
Desaturation 248
Destroy Session 338
Destructible 146
Destructible Flags 151
Dialogue Voice 215
Dialogue Wave 215
Diffuse 99
Directional Light 120
Distance Field Shadows 123
DPI 270
Draw Texture 265, 403

E

Editable Text 288
Eigenes Spiel 371
Emissive Color 104, 125
Emitter 221
Enum 417
Event ActorBeginOverlap 35
Event BeginPlay 35, 406
EventBlueprintUpdateAnimation 312
Event Construct 410
Event Graph 34
Event Hit 142, 397
Event-Tick 35, 58, 78, 161, 265, 312, 348, 381
EXE 367, 414
Execute Console Command 284, 409

F

FBX 79
Film 244
Find Session 339
Fit 170
Float 57
Foliage 26, 198
ForEachLoop 67, 263, 283, 297
Forward Vector 238
Fracture 148
Frensel 113
Friction 117

G

Game/Editor View 15
GameInstance 405, 411
Gate 47
GenerateWakeEvents 383
Geometry Editing 27, 172
Get 61
Get All Actors Of Class 262
GetAnimInstance 315
GetClass 262
GetDataTableRow 297

GetDataTableRowNames 297
GetMoveStatus 348
GetSelectedOption 283
Git 7
Global Illumination 137
GPU Sprites 227
Grab Component 161
Grid Panel 286

H

HDR 131
Heightmap 176
Hierarchy Depth 152
Hit Results 160
Horizontal Box 287
HUD 265

I

IES 128
Image 275
Impulse 163
Input 282
InputAxisLookUp 237
Input Mapping 235
Input Mode 291
Installation 9
Integer 56
Interface Event 256
Interface Output 257
Is Valid 50

J

Join Session 339

K

Künstliche Intelligenz (KI) 343
– Beispiel 346

L

LAN 340
Landscape 175
– Add 182
– Components 176
– Delete 182
– Grass Output 201
– Heightmap 176
– Manage 176
– Material 190
– Paint-Tool 195
– Sculpt 187
– Selection 181
– Splines 184
Launcher 11
Layer Blend 192
LayerBlendPerBone 323
Layer Weight 197
Lens Flare 246
Lerp 229
Level of Detail (LOD) 179
Licht 119
Light 64, 121
Light Function 125
Lightmaps 132
Lightmass 124
Lightmass Importance Volume 135
Light Profiles 128
Light Propagation Volume 135
Light Shaft 122
Limits 157
Lizenzen 2 f.
– Gebühren 3
Location 71
LOD 92, 364

M

Macros 361
Maps & Modes 338
Mass 140
Mass Scale 140
Material-Graph 100
Material Instance 194, 365, 392
Materials 99
Math Expression 63
Media Sound Wave 218
Menu Anchor 288
Mesh 79
Mesh Data 228
Metallic 102
Microsoft Visual Studio 12
Modes 172

Motion Blur 247
Motor 158
Movement Functions 240
MoveToActor 348
Move to Level 182
Multicast 332
Multiplayer 329
Multiplikation 62

N

Name 58
Named Slot 276
Native Widget Host 289
Navigation 18
Nav-Mesh 344
Netzwerk 329
Normal 99, 107
Normals 88

O

OBJ 79
OnComponentBeginOverlap 351
OnlineSubsystem 340
OnlyRelevantToOwner 330
OnMouseCapture 279
OnValueChanged 279
Opacity 105
Opacity Mask 102, 106
Optimierung 361
Order 168
Overlapping Actors 380
Overlapping Components 380
Overlay 287

P

Package Project 367, 414
Paint 25
Palette 118
Panner 391
Parallax Occlusion Mapping 114
Particle System 219
Partikel, Beispiel 228
Password 2
Pawn 233
Physical Material 116, 180
Physical Surface 117
Physics Asset 303
Physics Component 146
Physics Handle 159
Physics Thruster 162
Physik 139
– Beispiel 158
– in Blueprints 145
Pivot 87
Pixel Depth Offset 114
Play 27
PlaySoundAtLocation 397
Point Light 127
Polygone 85
Post Process 243
Post Process Volume 136, 248
Print String 30, 357
Progress Bar 278
Projection 157
Projektdateien 2
Projekt erstellen 12

R

Radial Force 163
ReceiveDrawHUD 265
Reference 259
Refraction 113
Release Component 161
Remove From Parent 291
Replicate Animations 335
ReplicateMovement 331
Replication 330
RepNotify 331
Reroute-Node 50
Restitution 117
Retargeting 324
Reverb Effect 218
Ribbon Data 228
Rig 325
Right Vector 238
Rotation 73
Rotator 60
Roughness 103
Row Editor 296
RunOnOwningClient 332
RunOnServer 332

S

SaveGame 299
Scale 74
Scale Box 287
Scene Color 245
Screen Percentage 247
Screen Size 270
Screen Space Reflection 248
Scroll Box 287
Sequenzer 46
Session 337
Session Result 339
Set 61
Set Actor Tick Enabled 78
SetCollisionEnabled 146
SetSimulatePhysics 145
ShowMouseCursor 409
Simulate Physics 140, 381, 395
Size Box 287
Skeletal Mesh 303
Skeleton 305
Sky Light 131
Slider 279
SLS 131
Snapping 18
Socket 83, 306
Solidity 168
Sound Attenuation 211
Sound Class 213
Sound Cue 207
Sound Delay 210
Sound Mix 214
Source-Code 3
Spacer 289
Spawn Actor 49
Specular 103
Speichern/Laden 299
Spin Box 284
Split Screen 415
Spot Light 129
State Machine 318
Static-Mesh-Editor 81
Static/Stationary/Movable Light 133
Steam 340
String 58
Struct 293
Subsurface Color 111
SubUV 400
Supported Platforms 368
Surface Material 169
SVN 7
SwitchHasAuthority 416
Systemvoraussetzungen 1

T

Tags 144
Tangents 88
Tessellation 109
Text 58
Text Block 280
Text Box 281
– Multi-Line 285
Texture Sample 101
Throbber 288 f.
Timeline 48
Toggle 45
Toolbar 81
Trace 99, 160, 242, 417
Transform 60, 75
Transition Rule 320
Translucent 105
Trello 7
Triangles 85
Trigger 144
TryGetPawnOwner 312
Two Sided 106
Type Data 227

U

Unfuddle 7
Uniform Grid Panel 287
UsePawnControlRotation 237
UV-Map 90

V

Variablen 53
Vector 59
VectorLength 313
Vertical Box 287
Vertices 85, 89
View Mode 17
Viewport 15, 33, 81
Viewport-Components 40
VInterp 77
VInterpTo 379

W

WatchThisValue 356
WAV 207
Webseite 2, 420
White Balance 244
Whiteboxing 165
Widget 265, 268, 289, 407
Widget Switcher 287
Wireframe 85
World Displacement 109
World Outliner 21
World Position Offset 108
World Settings 23
Wrap Box 288
Wrap Text 280

Y

YouTube 4, 420