

# Spiele entwickeln mit Unreal Engine 4

Programmierung mit Blueprints: Grundlagen & fortgeschrittene Techniken. Mit einer Einführung in Virtual Reality

Bearbeitet von  
Von: Jonas Richartz

2., erweiterte Auflage 2017. Buch. 512 S. Gebunden  
ISBN 978 3 446 45290 9  
Format (B x L): 17.4 x 24.4 cm  
Gewicht: 1073 g

[Weitere Fachgebiete > EDV, Informatik > Programmiersprachen: Methoden > Spiele-  
Programmierung, Rendering, Animation](#)

schnell und portofrei erhältlich bei

  
DIE FACHBUCHHANDLUNG

Die Online-Fachbuchhandlung [beck-shop.de](http://beck-shop.de) ist spezialisiert auf Fachbücher, insbesondere Recht, Steuern und Wirtschaft. Im Sortiment finden Sie alle Medien (Bücher, Zeitschriften, CDs, eBooks, etc.) aller Verlage. Ergänzt wird das Programm durch Services wie Neuerscheinungsdienst oder Zusammenstellungen von Büchern zu Sonderpreisen. Der Shop führt mehr als 8 Millionen Produkte.

# HANSER



## Leseprobe

zu

## „Spiele entwickeln mit Unreal Engine 4“ (2. Auflage)

von Jonas Richartz

ISBN (Buch): 978-3-446-45290-9

ISBN (E-Book): 978-3-446-45369-2

Weitere Informationen und Bestellungen unter  
<http://www.hanser-fachbuch.de/9783446452909>

sowie im Buchhandel

© Carl Hanser Verlag München

# Inhalt

<b>Vorwort</b> .....	<b>XI</b>
<b>1 Einleitung</b> .....	<b>1</b>
1.1 Was brauche ich? .....	1
1.2 Was lerne ich? .....	2
1.3 Lizenzen .....	3
1.4 Weiterentwicklung der Engine .....	4
<b>2 Erste Schritte</b> .....	<b>5</b>
2.1 Wie fange ich an? .....	5
2.2 Motivation .....	6
2.3 Planung .....	6
2.4 Sicherheitskopien .....	7
2.5 Learning by Doing .....	8
<b>3 Grundlagen</b> .....	<b>9</b>
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
3.4.2 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
3.4.7 Play .....	28
3.5 Ausprobieren .....	29

<b>4</b>	<b>Blueprints</b>	<b>31</b>
4.1	Was sind Blueprints?	31
4.2	Das Actor-Blueprint	33
4.2.1	Der Hauptbereich	34
4.2.2	Components	41
4.2.3	Details	44
4.2.4	Debug-Bereich	45
4.3	Anwendungsbeispiele	47
4.3.1	Toggle	47
4.3.2	Sequenzler	48
4.3.3	Timeline	50
4.3.4	SpawnActor	51
4.3.5	Reroute-Node	52
4.3.6	IsValid?	53
4.3.7	Promote to variable	53
<b>5</b>	<b>Bausteine der Welt</b>	<b>55</b>
5.1	Variablen	55
5.1.1	Boolean	56
5.1.2	Byte	58
5.1.3	Integer	58
5.1.4	Float	59
5.1.5	Name, String und Text	60
5.1.6	Vector	61
5.1.7	Rotator	61
5.1.8	Transform	62
5.2	Benutzen von Variablen	62
5.2.1	Variablen in Events	65
5.3	Arrays	68
5.4	Übung zu Arrays	70
<b>6</b>	<b>Die Welt in 3D</b>	<b>73</b>
6.1	World- und Relative-Transforms	73
6.2	Transforms in Blueprints	77
6.3	Meshes	81
6.3.1	Toolbar und Viewport	84
6.3.2	Details	94
6.4	Collisions	97
6.4.1	Kollisionstypen	101
6.5	Materials	103
6.5.1	Graph	104
6.5.2	Details	119
6.5.3	Palette	122

<b>7</b>	<b>Licht und Schatten</b>	<b>123</b>
7.1	Lichtarten	123
7.1.1	Directional Light	124
7.1.2	Point Light	130
7.1.3	Spot Light	133
7.1.4	Sky Light	134
7.2	Lightmaps	136
7.2.1	Lightmass Importance Volume	139
7.2.2	Light Propagation Volumes	139
7.3	Global Illumination	141
<b>8</b>	<b>Physik</b>	<b>143</b>
8.1	Simulate Physics	143
8.1.1	Collisions	146
8.1.2	Physik in Blueprints	149
<b>9</b>	<b>Ein Level entsteht</b>	<b>169</b>
9.1	BSP	170
9.1.1	Brush Settings	171
9.1.2	Surface Material	173
9.1.3	Geometry Editing	176
<b>10</b>	<b>Landschaften</b>	<b>179</b>
10.1	Landscape-Tool	179
10.1.1	Manage	180
10.1.2	Sculpt	191
10.2	Landscape-Material	194
10.2.1	Layer Blend	195
10.2.2	Material Instance	197
10.2.3	Paint-Tool	198
10.2.4	Layer Weight	200
10.3	Foliage-Tool	202
10.4	Grass Output	205
<b>11</b>	<b>Audio</b>	<b>209</b>
11.1	Sound-Arten	209
11.1.1	Sound Cue	211
11.1.2	Sound Attenuation	215
11.1.3	Sound Class	217
11.1.4	Sound Mix	218
11.1.5	Dialogue Voice/Wave	219
11.1.6	Reverb Effect	221
11.1.7	Media Sound Wave	222

<b>12</b>	<b>Partikel</b>	<b>223</b>
12.1	Cascade	224
12.1.1	Emitter	225
12.1.2	Type Data	231
12.2	Ein Beispiel für Effekte	232
<b>13</b>	<b>Der Character</b>	<b>237</b>
13.1	Character Blueprint	237
13.1.1	Character Movement	242
13.1.2	Movement-Funktionen	245
13.1.3	Vorbereitungen für Interaktionen	247
13.1.4	Kameraeigenschaften	248
<b>14</b>	<b>Kommunikation</b>	<b>257</b>
14.1	Cast to Blueprint	257
14.2	Interface	260
14.2.1	Output	263
14.3	Reference	266
14.3.1	Alle Actors einer Klasse	268
<b>15</b>	<b>User Interface</b>	<b>271</b>
15.1	HUD-Klasse	271
15.2	Widgets	274
15.2.1	Canvas	275
15.2.2	Palette	278
15.3	Benutzen von Widgets	296
<b>16</b>	<b>Datenbanken</b>	<b>299</b>
16.1	Structs	299
16.2	Data Table	301
16.2.1	Datenbanken in Blueprints	303
16.2.2	Speichern und Laden von Daten	305
<b>17</b>	<b>Animationen</b>	<b>309</b>
17.1	Skeletal Mesh	309
17.2	Skeleton	311
17.3	Animationen	313
17.3.1	Aim Offset	315
17.3.2	Blend Space	317
17.4	Animation Blueprint	318
17.4.1	Event Graph	318
17.4.2	Anim Graph	322
17.5	Retargeting	330

<b>18</b>	<b>Netzwerk</b>	<b>335</b>
18.1	Grundwissen über Multiplayer	335
18.2	Replication	336
18.2.1	Events	338
18.2.2	Animationen	340
18.3	Sessions	343
18.4	OnlineSubsystem	346
18.5	Multiplayer-Beispiele und Probleme	348
18.5.1	Events werden nicht ausgeführt	348
18.5.2	Replication und deren Auswirkung	349
18.5.3	Replication kombinieren	353
18.5.4	Owner herausstechen lassen	354
<b>19</b>	<b>KI</b>	<b>357</b>
19.1	Erste Schritte	357
19.2	Simple Patrouille	360
19.2.1	AIController	361
19.3	KI mit Angriff	364
19.4	Behaviour Tree/Query System	367
<b>20</b>	<b>Debugging</b>	<b>369</b>
20.1	Fehlersuche	369
20.2	Optimierung	375
<b>21</b>	<b>Spiel erstellen</b>	<b>381</b>
21.1	Project Launcher	384
21.1.1	Project	386
21.1.2	Build	386
21.1.3	Cook	386
21.1.4	Package	390
21.1.5	Archive	390
21.1.6	Deploy	390
<b>22</b>	<b>Ein eigenes Spiel</b>	<b>393</b>
<b>23</b>	<b>Virtual Reality mit Unreal Engine 4</b>	<b>435</b>
23.1	Was ist „Virtual Reality“?	435
23.1.1	Mit allen Sinnen	436
23.1.2	Eine kurze Geschichte	436
23.2	Achtung, Virtual Reality!	437
23.2.1	Der Spieler	437
23.2.2	Die Technik	439
23.3	Das VR Template der Unreal Engine	441

23.3.1	Vorbereitung .....	441
23.3.2	Neues VR-Projekt erstellen .....	443
23.3.3	VR Template ausprobieren .....	446
23.4	Das VR Template erklärt .....	449
23.4.1	VR Blueprints .....	449
23.4.2	Der MotionControllerPawn .....	450
23.4.3	Das MotionController Blueprint .....	458
23.4.4	Der Pickup Cube .....	466
23.5	Die virtuelle Wurfbude .....	467
23.5.1	Game Design .....	467
23.5.2	Das Spielfeld .....	468
23.5.3	Der Wurfball .....	471
23.5.4	Der Punktezähler .....	473
23.5.5	Ringe als Ziele .....	475
23.5.6	Hebel zum Reset .....	482
23.6	Zusammenfassung .....	488
<b>24</b>	<b>Tipps und Tricks .....</b>	<b>489</b>
24.1	Features, die es in die vorherigen Kapitel nicht geschafft haben .....	489
24.1.1	Split Screen .....	489
24.1.2	Authority .....	491
24.1.3	Maus zur Welt .....	491
24.1.4	Enums .....	492
24.1.5	Audio stoppen .....	493
24.2	Schlusswort .....	493
<b>Index</b>	<b>.....</b>	<b>495</b>



# 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.

Besonderer Dank gilt Benedikt Engelhard für das VR-Kapitel sowie nochmals Sylvia Hasselbach für die Betreuung der zweiten Auflage.

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 August 2017

*Jonas Richartz*

# 4

## Blueprints

In den Kapiteln und bei deinen ersten eigenen Versuchen ist dir der Name *Blueprint* schon oft begegnet. In diesem Kapitel werde 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

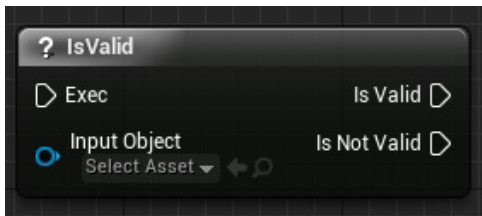
```
#include <stdio.h>

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**

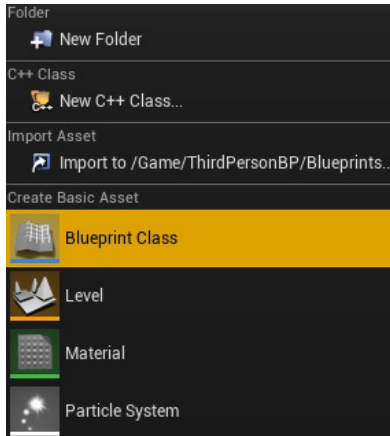
So sieht das Ganze in Blueprints aus

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

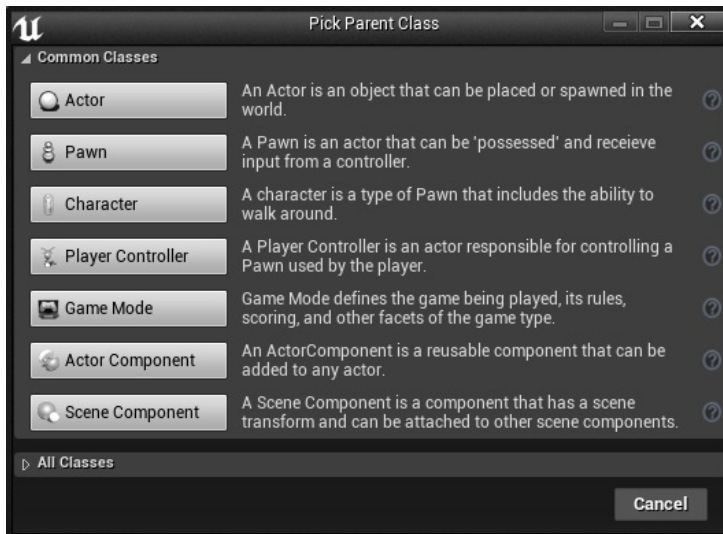
Bevor wir uns tiefer in die Logik von Blueprints und der Programmierung stürzen, schauen wir uns erst einmal an, wie so ein Blueprint standardmäßig aussieht.



**Bild 4.2**

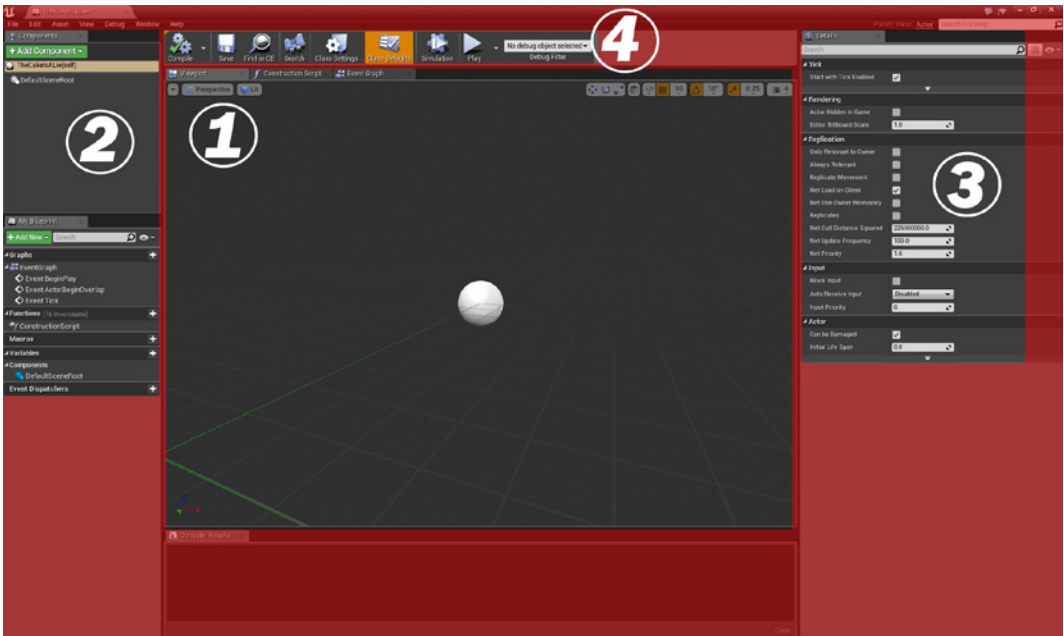
Der erste Schritt zum Erstellen eines Blueprints

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 erst einmal 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, zur Suche und zum direkten Start des Spiels aus dem Blueprint heraus.

### 4.2.1 Der Hauptbereich

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



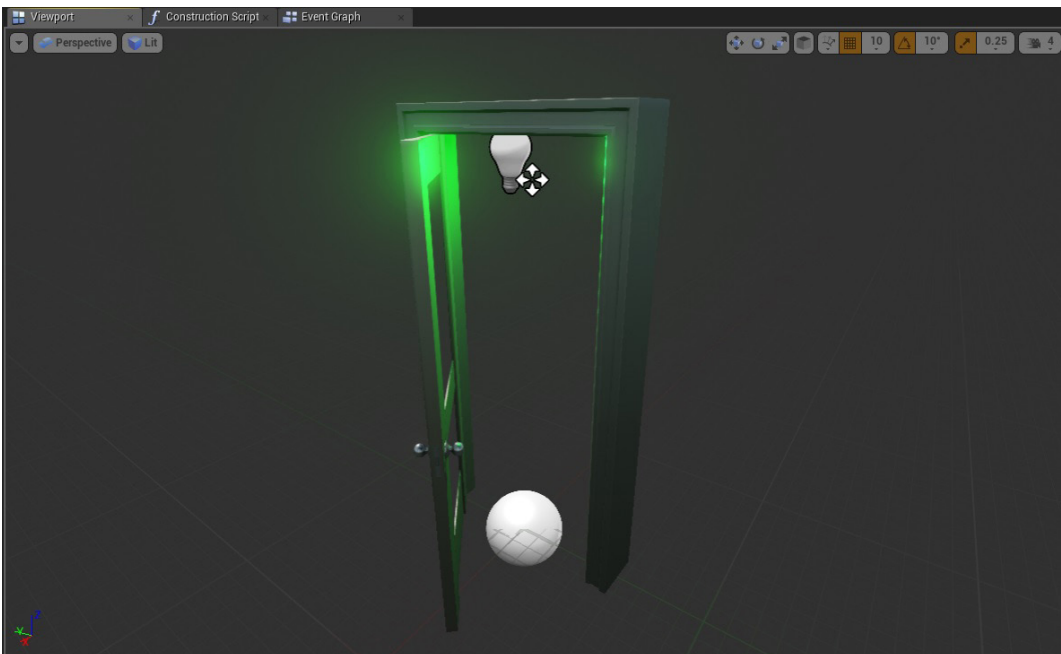
**Bild 4.5** Die drei wichtigsten Bereiche

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 Null-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 herstellen 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

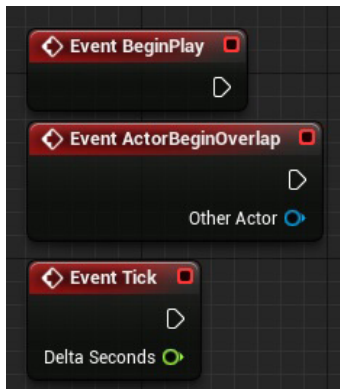
Das **Construction Script** ist im Prinzip ähnlich wie der Event Graph, hat jedoch ein paar Restriktionen. Dieser Bereich wird immer ausgeführt, wenn sich das Blueprint im Editor verändert, sei es die Umstellung einer Variablen oder gar das Bewegen des Blueprints in deinem Level. Wichtig zu wissen ist, dass dieser Bereich nicht beim Spielen selbst 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 dahintersteht, 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 diesen 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:** Befindet sich in deinem Blueprint ein Objekt mit Kollision, wird das Event ausgelöst, sobald es ein anderes Objekt mit Kollision überlappt. Zu dem Event gehört, wie in 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 in 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!

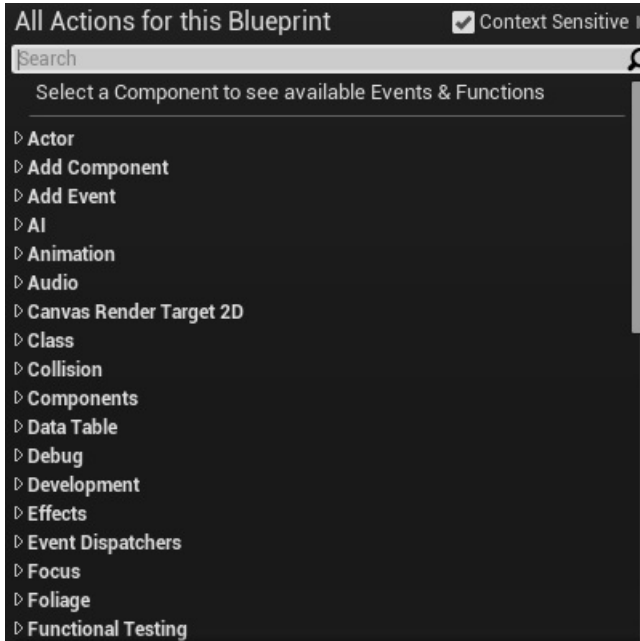
Es gibt noch eine Vielzahl an unterschiedlichen Events mit den unterschiedlichsten Funktionen, dies wäre aber definitiv zu viel für den Anfang. Wir werden uns in den kommenden Kapiteln nach und nach weitere Events anschauen, aber für den Anfang sollte das erst einmal reichen.

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

### Ein eigenes Event erstellen

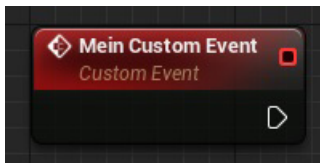
Mit Rechtsklick im Event Graph öffnen wir das sogenannte *Context-Menü*, in dem alle verfügbaren Optionen von Nodes und Events, die man platzieren kann, dargestellt werden.





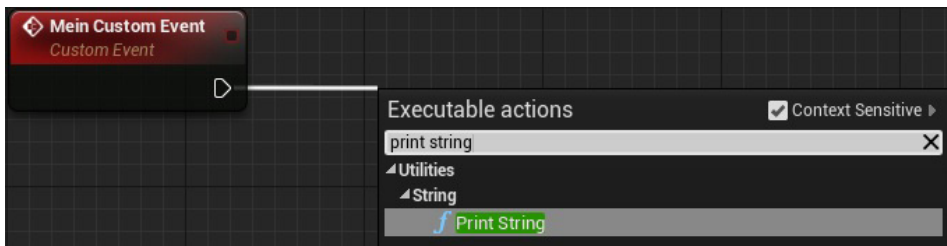
**Bild 4.8**  
Das Context-Menü im Event Graph

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.



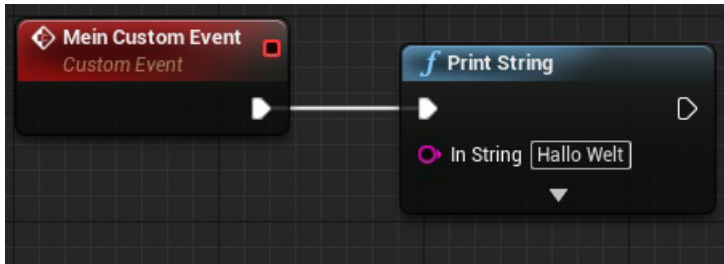
**Bild 4.9**  
Ein wunderschönes Event

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



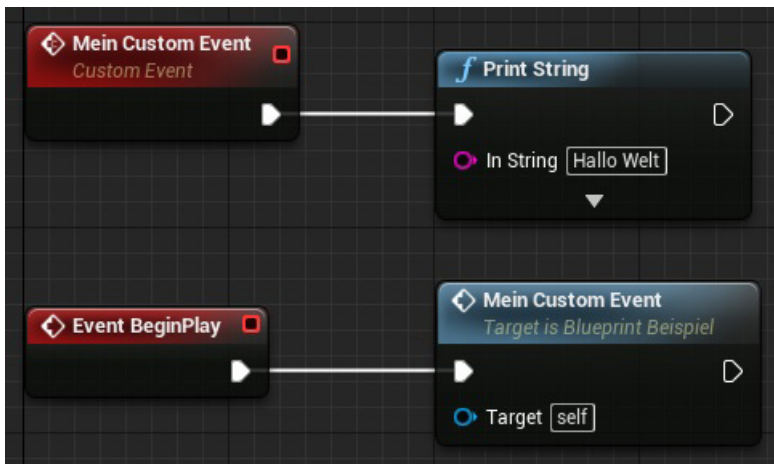
**Bild 4.10** So sollte es aussehen

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, das 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, das 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, das 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 dem C-Code in Listing 4.2.

**Listing 4.2** Das gleiche Beispiel ähnlich in C

```
#include <stdio.h>

int main(void) {

MeineCustomFunktion ();

}
void MeineCustomFunktion(){

printf("Hallo Welt");

}
```

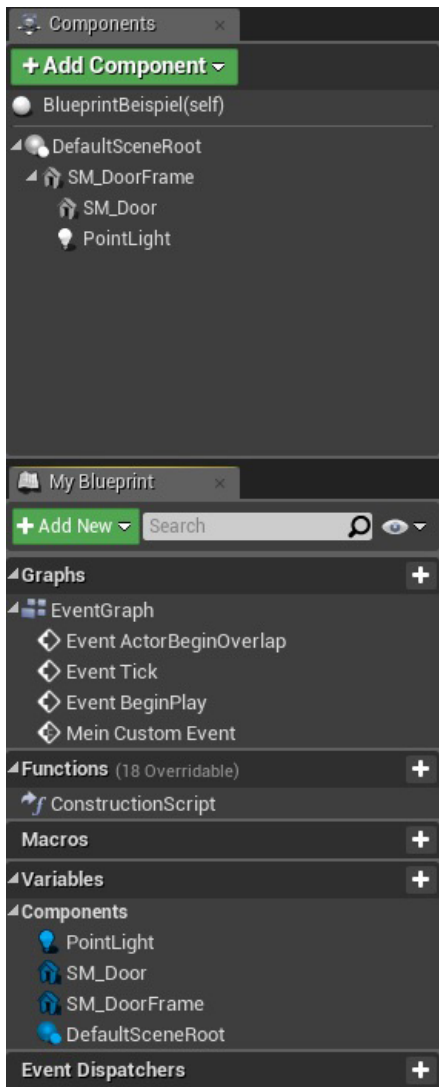


**HINWEIS:** Die gezeigten Codes dienen nur der Veranschaulichung und sind nicht eins zu eins mit Blueprints vergleichbar. Es geht mir nur um die Logik, die dahinter steht.

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, was in *Kapitel 5* weiter erläutert wird.

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

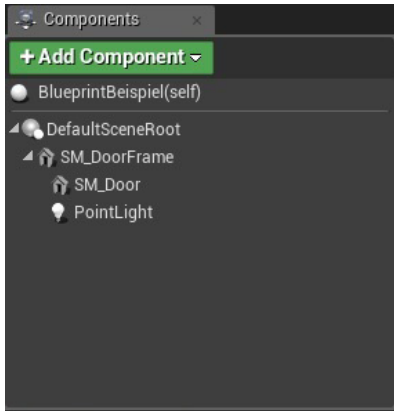
## 4.2.2 Components



**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.

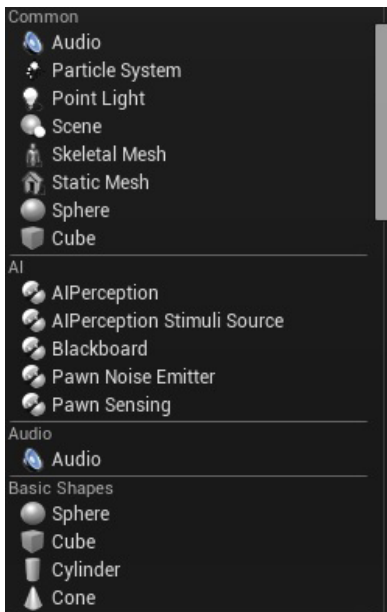
### 4.2.2.1 Viewport-Components



**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.



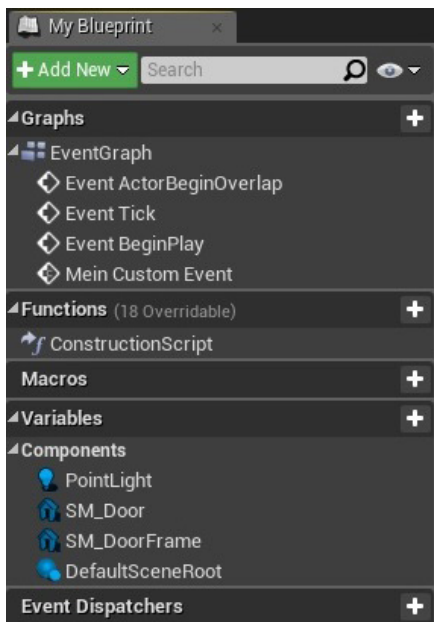
**Bild 4.15**  
Ein Bruchteil der Auswahl

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 verschiedenen 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.

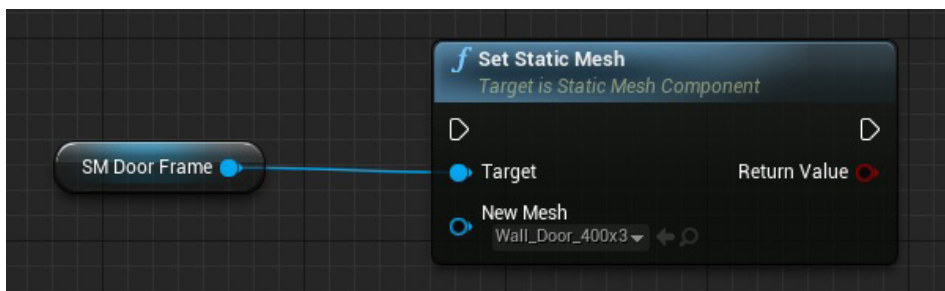
### 4.2.2.2 Blueprint Components



**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.

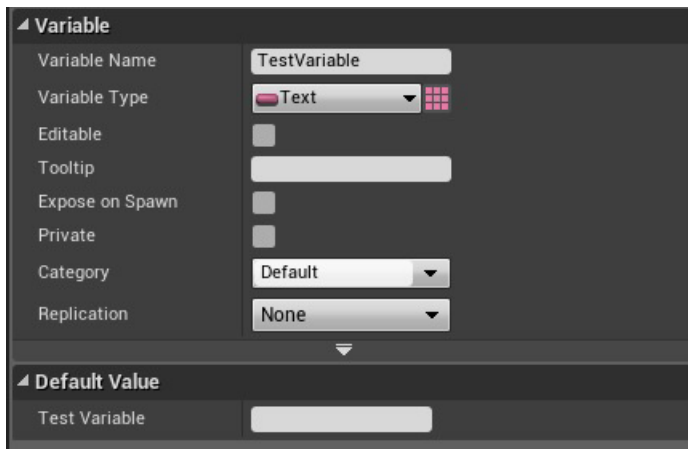


**Bild 4.17** Kleines Beispiel für die Components im Viewport

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. ausgewählter Variablen 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*.

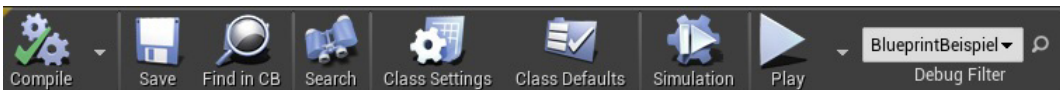
*Editable* steht dafür, dass die Variable editierbar ist. Im Prinzip sind alle Variablen editierbar, jedoch ist hierbei gemeint, dass die Variable von außerhalb des Blueprints verändert werden kann. Das heißt also, wenn du eine Text-Variablen hast, die für den Namen eines Charakters stehen soll, so platzierst 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 direkt diejenigen, wo der Fehler passiert ist, um einzugreifen und genau nachzuschauen, was passiert.



**Bild 4.19** Der Debug-Bereich

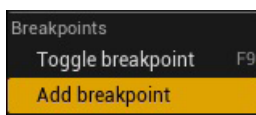
Aber wie funktioniert das genau in Blueprints? Zuerst einmal beinhaltet der von mir genannte Debug-Bereich einiges mehr als nur das reine Debugging. Genauer gesagt ist für das Debugging am Anfang nur der **Debug Filter** nötig. Es gibt neben den ganzen anderen Buttons nur zwei, die nicht unbedingt selbsterklärend 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

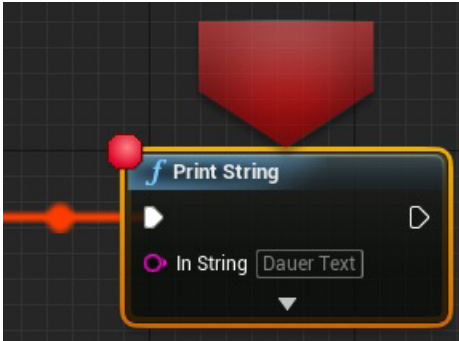
Alle momentan benutzen Stränge werden als rote und sich damit bewegende Punkte dargestellt. Somit kannst du genau nachverfolgen, woran und wo dein Blueprint aktuell arbeitet – ein ziemlich cooles Feature. Aber wir wollen vielleicht auch einfach nur ab einer bestimmten **Node** anfangen, zu debuggen.



**Bild 4.21**  
Breakpoint einfügen

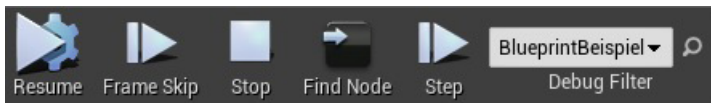


Einfach mit rechts auf den gewünschten Node klicken, von dem du mit dem Debuggen anfangen willst, und **Add breakpoint** auswählen. Das sieht danach wie folgt aus.



**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.

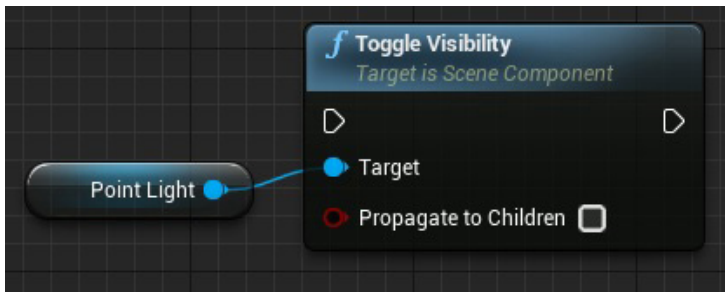
In *Kapitel 20* geht es nochmal ausführlicher um Debugging.

Jetzt haben wir erst einmal 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, was sich hinter der Logik verbirgt und welche Nodes besonders häufig verwendet werden.

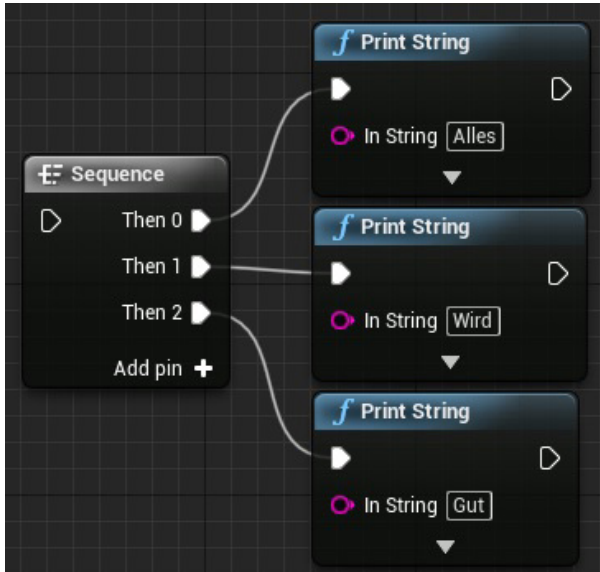
### 4.3.1 Toggle



**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 Lichtquellen 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.

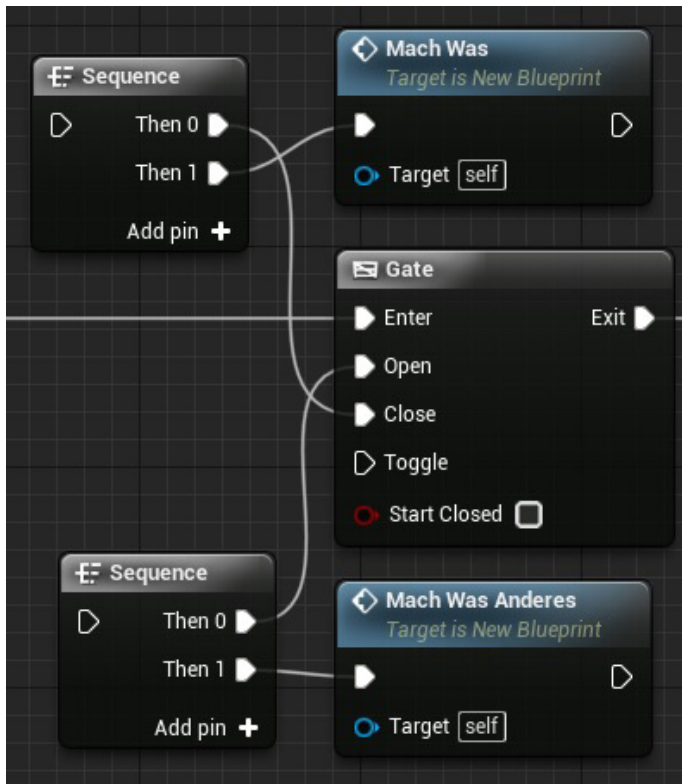
### 4.3.2 Sequenzer



**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.

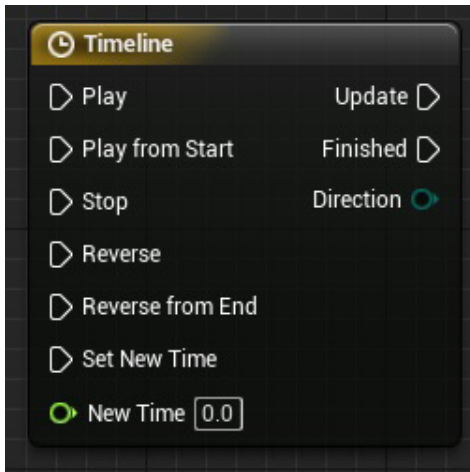
## Gate



**Bild 4.26** Sequence mit Gate

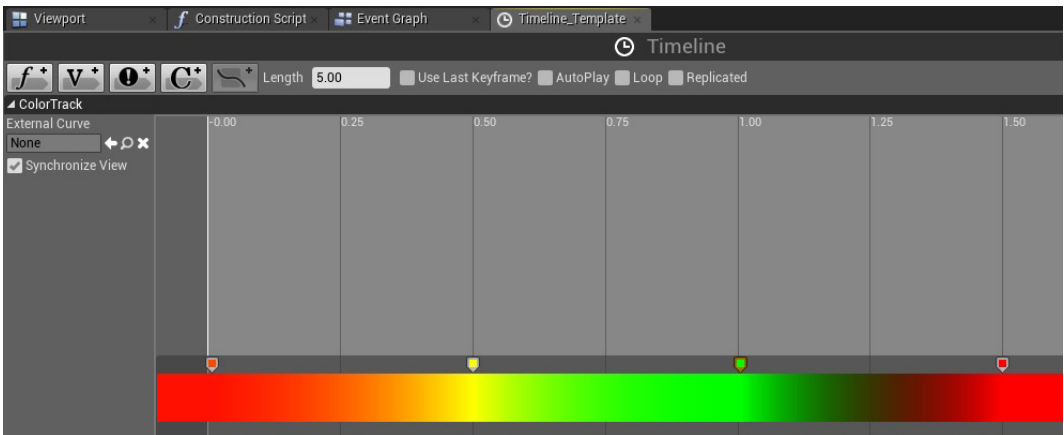
Ein Gate ist ein Logik-Tor, das offen ist oder sein kann. Wenn wir also nur zu bestimmten Zeiten die Logik nach dem Gate ausführen wollen, müssen wir das Gate öffnen und schließen können. Aber dies ist auch oft nicht das Einzige, was man damit bewirken will. Dafür gibt es dann die Sequence-Node, die 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



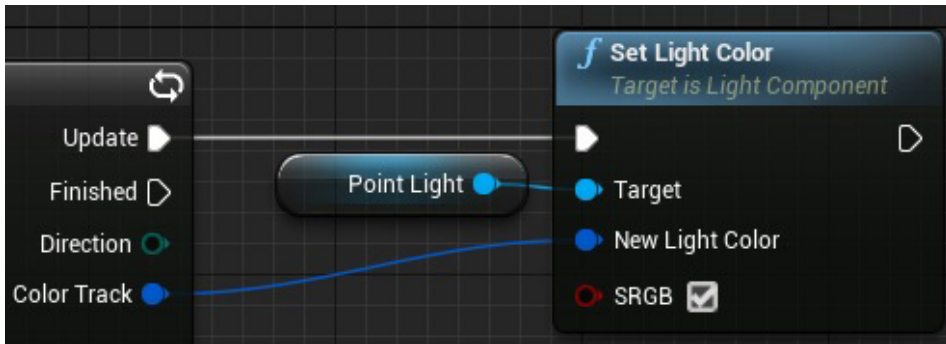
**Bild 4.27**  
Timeline-Node

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



**Bild 4.28** Farbänderungen in einer Timeline

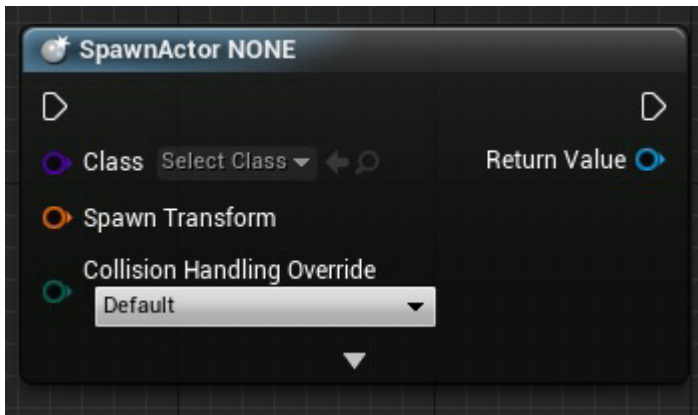
Wie du in Bild 4.28 sehen kannst, könnte man beispielsweise eine Farbänderung in eine Timeline einbauen und diese dann mithilfe des Update-Outputs einem Licht zuweisen. Update wird permanent ausgeführt, während die Timeline läuft.



**Bild 4.29** Benutzen des Update-Ausgangs

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

#### 4.3.4 SpawnActor

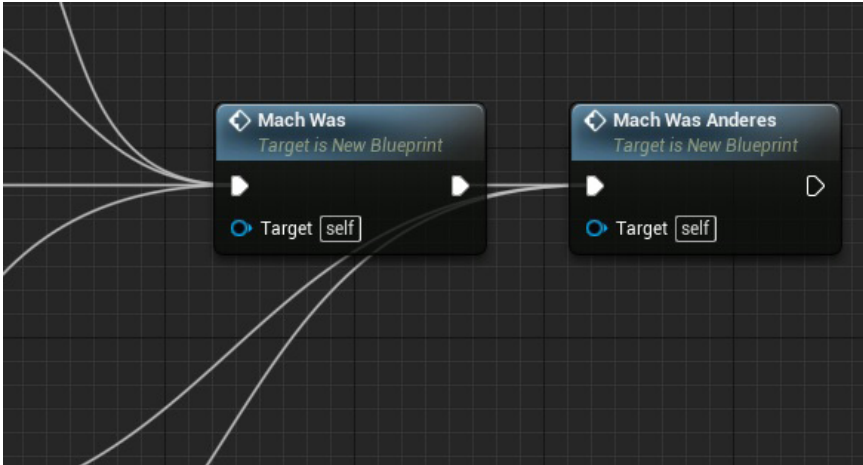


**Bild 4.30**  
SpawnActor-Node

Mit SpawnActor 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 in Bild 4.30.

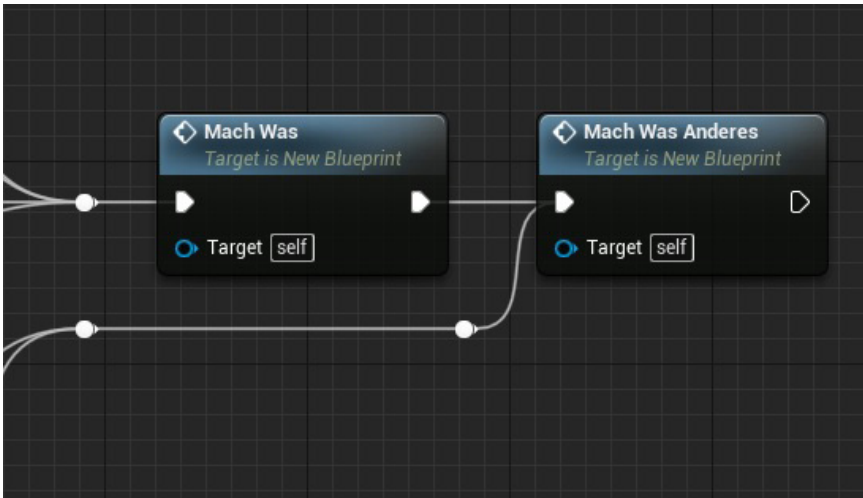
### 4.3.5 Reroute-Node

Blueprints und Verbindungen innerhalb von Blueprints können sehr schnell unübersichtlich werden und durcheinandergeraten.



**Bild 4.31** Ohne Reroute-Node

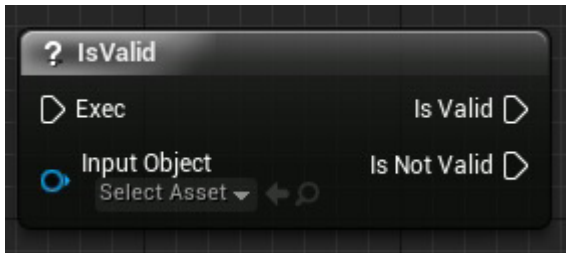
Wir wollen verhindern, dass unsere Blueprints unübersichtlich werden, und da kommen Reroute-Nodes zum Einsatz, um die Verbindungen zu bündeln und umzuleiten.



**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 IsValid?



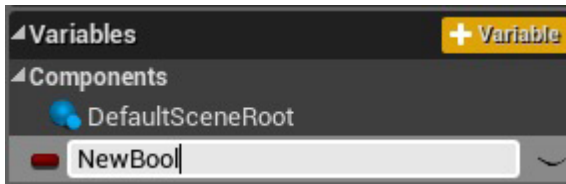
**Bild 4.33**  
IsValid-Node

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, das/die du in das Input-Object einsetzt, noch existiert bzw. ob es noch gültig ist oder nicht.

Nehmen wir einmal 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.

### 4.3.7 Promote to variable

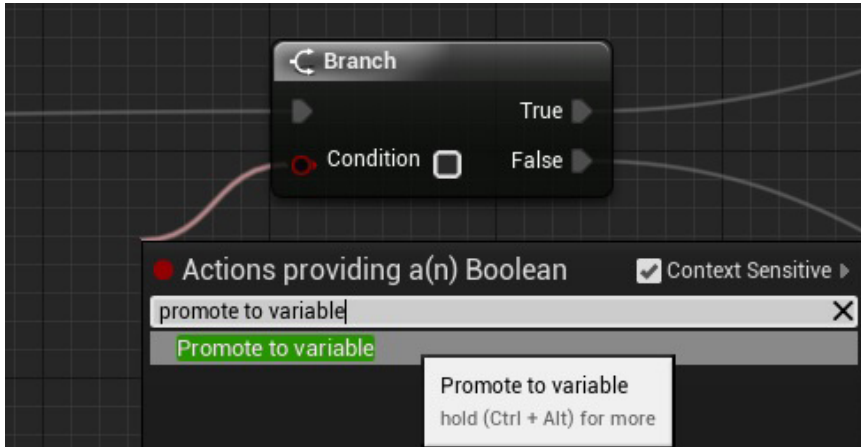
Das Arbeiten mit Blueprints soll schnell gehen, damit man sich nicht lange mit Kleinigkeiten aufhalten muss. Dabei geht es auch um Sachen, wie die Erstellung einer Variablen. Im Normalfall würde man eine Variable im Blueprint-Component-Bereich (vgl. Abschnitt 4.2.2.2) erstellen, und zwar mit einem Linksklick auf das + bei Variablen.



**Bild 4.34**  
Standardvariante zur Variablen-  
Erstellung

Dies kann unter Umständen störend sein, wenn du gerade vollkommen in deine Logik vertieft bist. Doch dafür gibt es eine Lösung, wie du schnell innerhalb der Logik eine neue Variable erstellen kannst und deinen Fokus beibehältst. Dies machst du mit *Promote to variable*.





**Bild 4.35** Promote to variable

Wenn du innerhalb des Event-Graphs, Construction-Script oder einer Funktion bist, kannst du von Bausteinen, die eine Variable benötigen oder ausgeben, einen Strang erzeugen und mithilfe von *Promote to variable* automatisch eine Variable des gleichen Typs erstellen. In Bild 4.35 wird beispielsweise eine Bool variable erstellt. Was genau ein Bool macht, erfährst du in Kapitel 5. Mir hilft diese Variante immer sehr, wenn ich im Fokus bin und schnell eine neue Variable erzeugen will. Die erzeugten Variablen tauchen wie gewohnt auch im Blueprint-Components-Bereich auf, wo du gegebenenfalls auch einen Standardwert eintragen kannst.

# Index

## Symbole

3D 73

## A

Actors 33  
Add 68  
Add Controller Yaw Input 241  
Add Movement Input 242  
AddOptions 290  
AddToViewport 429  
AI 357  
AIController 358  
Aim Offset 315  
Alignment 174  
Always Relevant 336  
Ambient Cubemap 251  
Ambient Occlusion 117, 251  
Animation Blueprint 318  
Animationen 309  
Animation Notifies 314, 327  
Animations 313  
Anim Graph 322  
Anim Trail 231  
Anti-Aliasing (AA) 252  
APEX 150  
Append 290  
Apply Radial Damage 159  
Array 68  
Arrow Component 238  
Artificial Intelligence (AI) 357  
AttachActorToComponent 320  
Attenuation Radius 131, 134  
Audio 209  
Audio Component 493  
Auto Exposure 251

## B

Baked Lighting 136  
Base Color (Diffuse) 105  
Beam Data 231  
Behaviour Tree 367  
Billboard 265, 360  
Binormals 90  
Blendables 253  
Blend Space 317, 323  
Bloom 250  
Blueprint Components 43  
Blueprint-Interface 260, 397  
Blueprint-Kommunikation 257  
Blueprints 31  
– Physik 149  
Bool 56  
Boolean 56  
Border 279  
Bounds 88  
Branch 57  
Breakpoint 369  
Brush Settings 171  
BSP 170  
Build Lighting Only 136  
Button 279  
Byte 58

## C

Camera Settings 248  
Canvas 275  
Canvas Panel 293  
CapsuleComponent 238  
Cascade 223  
Cascaded Shadow Maps 128  
Cast to 257  
Change Component Size 187

Chaperone 447  
 Character Blueprint 237, 318  
 Character Movement 238, 242  
 Check Box 281  
 Chunk Parameters 158  
 Circular Throbber 295  
 Clear Coat 116  
 Collision 97, 146  
 Collision Presets 101  
 Color Grading 249  
 ColorOverLife 422  
 Combo Box 288  
 Comment Box 377  
 Components 41  
 Conduit 324  
 Constraints 144, 159  
 ConstructionScript 424  
 Content Browser 20  
 CreatePlayer 489  
 Create Session 344  
 Create Widget 429  
 Cubemaps 135  
 Custom Event 37, 65, 304

## D

Damage 154  
 Data Table 301  
 Datenbanken 299  
   – in Blueprints 303  
 Debug Filter 372  
 Debugging 45, 369  
 Density 121  
 Depth of Field 251  
 Desaturation 253  
 Destroy Session 344  
 Destructible 150  
 Destructible Flags 155  
 Dialogue Voice 219  
 Dialogue Wave 219  
 Diffuse 103  
 Directional Light 124  
 Distance Field Shadows 127  
 DPI 276  
 Draw Texture 271, 425

## E

Ecute Console Command 290  
 Editable Text 295  
 eigenes Spiel 393

Eingabe-Events 454  
 Emissive Color 108, 129  
 Emitter 225  
 Enum 492  
 Epilepsie 438  
 Event ActorBeginOverlap 37  
 Event BeginPlay 37, 451, 461  
 EventBlueprintUpdateAnimation 318  
 Event Construct 430  
 Event Drop 466  
 Event Graph 36  
 Event Hit 146, 418  
 Event Pickup 466  
 Event-Tick 37, 59, 80, 165, 271, 318, 362, 403,  
   453, 461  
 EXE 381, 434  
 Execute Console Command 429  
 Execute Teleportation 457

## F

FBX 81  
 Film 249  
 Find Session 345  
 Fit 175  
 Float 59  
 Foliage 27, 202  
 ForEachLoop 69, 269, 290, 303, 345  
 Forward Vector 242  
 Fracture 152  
 Framerate 440  
 Frensel 117  
 Friction 121

## G

Game/Editor View 15  
 GamelInstance 427, 431  
 Gate 49  
 GenerateWakeEvents 404  
 Geometry Editing 28, 176  
 Get 63  
 Get Actor Near Hand 465  
 Get All Actors Of Class 268  
 GetAnimInstance 321  
 GetClass 269  
 GetDataTableRow 303  
 GetDataTableRowNames 303  
 Get HMDDevice Name 451  
 GetMoveStatus 362  
 GetSelectedOption 290

Git 7  
 Global Illumination 141  
 GPU Sprites 231  
 Grab Actor 464  
 Grab Component 165  
 GrabSphere 460  
 Grid Panel 293

## H

HandMesh 459  
 HDR 135  
 Heightmap 180  
 Hierarchy Depth 156  
 Hit Results 164  
 Horizontal Box 294  
 HTC Vive 443  
 HUD 271

## I

IES 132  
 Image 282  
 Impulse 168  
 Input 288  
 InputAxis LookUp 241  
 Input Mapping 240  
 Input Mode 297  
 Installation 9  
 Integer 58  
 Interface Event 263  
 Interface Output 263  
 Is Valid 53

## J

Join Session 345  
 Jumpscare 438

## K

Künstliche Intelligenz (KI) 357
 

- Beispiel 360

## L

LAN 346  
 Landscape 179
 

- Add 186
- Components 180
- Delete 186

- Grass Output 205
- Heightmap 180
- Manage 180
- Material 194
- Paint-Tool 198
- Sculpt 192
- Selection 185
- Splines 188

Latenz 439  
 Launcher 11  
 Layer Blend 195  
 LayerBlendPerBone 329  
 Layer Weight 200  
 Lens Flare 251  
 Lerp 233  
 Level of Detail (LOD) 183  
 Licht 123  
 Light 66, 125  
 Light Function 129  
 Lightmaps 136  
 Lightmass 128  
 Lightmass Importance Volume 139  
 Light Profiles 132  
 Light Propagation Volume 139  
 Light Shaft 126  
 Limits 161  
 Lizenzen 2f.
 

- Gebühren 3

 Location 73  
 LOD 94, 378  
 Luis Cataldi 441

## M

Macros 375  
 Maps & Modes 344  
 Mass 144  
 Mass Scale 144  
 Material-Graph 104  
 Material Instance 197, 378, 412  
 Materials 103  
 Math Expression 65  
 Media Sound Wave 222  
 Menu Anchor 295  
 Mesh 81  
 Mesh Data 232  
 Metallic 106  
 Microsoft Visual Studio 12  
 Modes 176  
 Motion Blur 252  
 MotionController 458

Motion Controller Component 458  
 MotionControllerPawn 450  
 MotionControllerPawn Event Graph 451  
 Motion Sickness 438  
 Motion to Photon 440  
 Motor 162  
 Movement Functions 245  
 MoveToActor 362  
 Move to Level 186  
 Multicast 338  
 Multiplayer 335  
 Multiplikation 64

## N

Name 60  
 Named Slot 282  
 Native Widget Host 295  
 Navigation 18  
 Nav-Mesh 358  
 Netzwerk 335  
 Normal 103, 111  
 Normals 90

## O

OBJ 81  
 Oculus Rift 442  
 OnComponentBeginOverlap 365  
 OnlineSubsystem 346  
 Only Relevant to Owner 336  
 OnMouseCapture 286  
 OnValueChanged 286  
 Opacity 109  
 Opacity Mask 106, 110  
 Optimierung 375  
 Order 172  
 Overlapping Actors 401  
 Overlapping Components 401  
 Overlay 294

## P

Package Project 381, 434  
 Paint 26  
 Palette 122  
 Panner 412  
 Parallax Occlusion Mapping 118  
 Particle System 223  
 Partikel, Beispiel 232  
 Password 2

Pawn 237  
 Phobien 438  
 Physical Material 120, 184  
 Physical Surface 122  
 Physics Asset 309  
 Physics Component 150  
 Physics Handle 163  
 Physics Thruster 166  
 Physik 143
 

- Beispiel 162
- in Blueprints 149

 Pickup Cube 466  
 Pivot 89  
 Pixel Depth Offset 118  
 Play 28  
 PlaySoundAtLocation 418  
 Point Light 130  
 Polygone 87  
 Post Process 248  
 Post Process Volume 140, 256  
 Print String 32, 370  
 Progress Bar 284  
 Projection 161  
 Projektdateien 2  
 Projekt erstellen 12  
 Promote to-variable 53

## R

Radial Force 167  
 ReceiveDrawHUD 271  
 Reference 266  
 Refraction 117  
 Release Actor 465  
 Release Component 165  
 Remove From Parent 298  
 Replicate Animations 340  
 Replicate Movement 337  
 Replication 336  
 RepNotify 337  
 Reroute-Node 52  
 Restitution 121  
 Retargeting 330  
 Reverb Effect 221  
 Ribbon Data 232  
 Rig 331  
 Right Vector 242  
 Rotation 75  
 Rotator 61  
 Roughness 107  
 Row Editor 302

Run on owning Client 338  
Run on Server 338

## S

SaveGame 305  
Scale 76  
Scale Box 294  
Scene Color 250  
Screen Percentage 252  
Screen Size 276  
Screen Space Reflection 252  
Scroll Box 294  
Sequenzler 48  
Session 343  
Session Result 345  
Set 63  
Set Actor Tick Enabled 80  
SetCollisionEnabled 150  
SetSimulatePhysics 149  
Set Tracking Origin 451  
ShowMouseCursor 429  
Simulate Physics 144, 403, 416  
Size Box 294  
Skeletal Mesh 309  
Skeleton 311  
Sky Light 134  
Slider 285  
SLS 135  
Snapping 18  
Socket 85, 312  
Solidity 172  
Sound Attenuation 215  
Sound Class 217  
Sound Cue 211  
Sound Delay 214  
Sound Mix 218  
Source-Code 3  
Spacer 296  
SpawnActor 51  
Specular 107  
Speichern/Laden 305  
Spin Box 291  
Split Screen 489  
Spot Light 133  
State Machine 324  
Static-Mesh-Editor 83  
Static/Stationary/Movable Light 137  
Steam 346  
SteamVRChaperone 460  
SteamVR-Einrichtungsassistent 443

Stereoskopie 436  
String 60  
Struct 299  
Subsurface Color 115  
SubUV 421  
Supported Platforms 382  
Surface Material 173  
SVN 7  
SwitchHasAuthority 491  
Systemvoraussetzungen 1

## T

Tags 148  
Tangents 90  
Tessellation 113  
Text 60  
Text Block 286  
Text Box 287  
– Multi-Line 291  
Texture Sample 105  
Throbber 295 f.  
Timeline 50  
Toggle 47  
Toolbar 84  
Touch Controller 443  
Trace 103, 164, 247, 491  
Transform 62, 77  
Transition Rule 325  
Translucent 109  
Trello 7  
Triangles 87  
Trigger 148  
TryGetPawnOwner 318  
Two Sided 110  
Type Data 231

## U

Unfuddle 7  
Uniform Grid Panel 294  
Use Pawn Control Rotation 241  
UV-Map 92

## V

Variablen 55  
Vector 61  
VectorLength 319  
Vertical Box 294  
Vertices 88, 91

View Mode 17  
Viewport 15, 35, 84  
Viewport-Components 42  
VInterp 79  
VInterpTo 401  
Virtual Reality 435  
virtuelle Wurfbede 467  
Vive Controller 443  
VR Template 441

## W

WatchThisValue 370  
WAV 211  
Webseite 2, 494

White Balance 248  
Whiteboxing 169  
Widget 271, 274, 296, 427  
Widget Switcher 294  
Wireframe 87  
World Displacement 113  
World Outliner 21  
World Position Offset 112  
World Settings 24  
Wrap Box 294  
Wrap Text 287

## Y

YouTube 4, 494