

carsten SEIFERT

Komplett in  
**FARBE**

# SPIELE ENTWICKELN MIT **Unity**

3D-GAMES MIT **UNITY** UND **C#** FÜR  
**DESKTOP, WEB & MOBILE**



Auf DVD: komplettes Spiel  
plus Videotutorials

HANSER

Seifert

Spiele entwickeln mit Unity

### Hinweis:

Zu diesem Buch gehört eine DVD.  
Sollte diese DVD nicht beiliegen, können Sie  
sie unter [fachbuch@hanser.de](mailto:fachbuch@hanser.de) kostenlos  
anfordern.

### Bleiben Sie auf dem Laufenden!



Unser **Computerbuch-Newsletter** informiert  
Sie monatlich über neue Bücher und Termine.  
Profitieren Sie auch von Gewinnspielen und  
exklusiven Leseproben. Gleich anmelden unter



[www.hanser-fachbuch.de/newsletter](http://www.hanser-fachbuch.de/newsletter)



**Hanser Update** ist der IT-Blog des Hanser Verlags  
mit Beiträgen und Praxistipps von unseren Autoren  
rund um die Themen Online Marketing, Webent-  
wicklung, Programmierung, Softwareentwicklung  
sowie IT- und Projektmanagement. Lesen Sie mit  
und abonnieren Sie unsere News unter



[www.hanser-fachbuch.de/update](http://www.hanser-fachbuch.de/update)   

Lizenziert für gregwyss@hotmail.com.

© 2014 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

Carsten Seifert

# Spiele entwickeln mit Unity

3D-Games mit Unity und C#  
für Desktop, Web & Mobile

HANSER

**»Der Weltuntergang steht bevor,  
aber nicht so, wie Sie denken.  
Dieser Krieg jagt nicht alles in die Luft,  
sondern schaltet alles ab.«**



**Tom DeMarco  
Als auf der Welt das Licht ausging**

ca. 560 Seiten. Hardcover

ca. € 19,99 [D] / € 20,60 [A] / sFr 28,90

ISBN 978-3-446-43960-3

Erscheint im November 2014

Hier klicken zur  
Leseprobe

Sie möchten mehr über Tom DeMarco und seine Bücher erfahren.  
Einfach reinklicken unter [www.hanser-fachbuch.de/special/demarco](http://www.hanser-fachbuch.de/special/demarco)

Der Autor:

*Carsten Seifert*, Süderbrarup

[www.hummelwalker.de](http://www.hummelwalker.de)

Alle in diesem Buch enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht.

Ebenso übernehmen Autor und Verlag keine Gewähr dafür, dass beschriebene Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt deshalb auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen.



Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren) – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2014 Carl Hanser Verlag München, [www.hanser-fachbuch.de](http://www.hanser-fachbuch.de)

Lektorat: Sieglinde Schärl

Copy editing: Sandra Gottmann, Münster-Nienberge

Herstellung: Irene Weilhart

Erstellung der Dateien für das Beispiel-Game: Alexej Bodemer, Stuhr, [www.alexejbodemer.de](http://www.alexejbodemer.de)

Umschlagdesign: Marc Müller-Bremer, München, [www.rebranding.de](http://www.rebranding.de)

Umschlagrealisation: Stephan Rönigk

Gesamtherstellung: Kösel, Krugzell

Printed in Germany

Print-ISBN: 978-3-446-43939-9

E-Book-ISBN: 978-3-446-44129-3

# Inhalt

<b>Vorwort .....</b>	<b>XVII</b>
<b>1 Einleitung .....</b>	<b>1</b>
1.1 Multiplattform-Publishing .....	1
1.2 Das kann Unity (nicht) .....	2
1.3 Lizenzmodelle .....	2
1.4 Aufbau und Ziel des Buches .....	3
1.5 Weiterentwicklung von Unity .....	4
1.6 Online-Zusatzmaterial .....	4
<b>2 Grundlagen .....</b>	<b>5</b>
2.1 Installation .....	5
2.2 Oberfläche .....	5
2.2.1 Hauptmenü .....	7
2.2.2 Scene View .....	8
2.2.3 Game View .....	10
2.2.4 Toolbar .....	11
2.2.5 Hierarchy .....	13
2.2.6 Inspector .....	15
2.2.7 Project Browser .....	18
2.2.8 Console .....	20
2.3 Das Unity-Projekt .....	20
2.3.1 Neues Projekt anlegen .....	21
2.3.2 Bestehendes Projekt öffnen .....	22
2.3.3 Projektdateien .....	22
2.3.4 Szene .....	23
2.3.5 Game Objects .....	24
2.3.6 Components .....	24
2.3.7 Tags .....	25
2.3.8 Layer .....	26

2.3.9 Assets .....	26
2.3.10 Frames .....	30
2.4 Das erste Übungsprojekt .....	30
<b>3 C# und Unity .....</b>	<b>33</b>
3.1 Die Sprache C# .....	33
3.2 Syntax .....	34
3.3 Kommentare .....	35
3.4 Variablen .....	35
3.4.1 Namenskonventionen .....	35
3.4.2 Datentypen .....	36
3.4.3 Schlüsselwort var .....	37
3.4.4 Datenfelder/Array .....	37
3.5 Konstanten .....	39
3.5.1 Enumeration .....	39
3.6 Typkonvertierung .....	40
3.7 Rechnen .....	40
3.8 Verzweigungen .....	41
3.8.1 if-Anweisungen .....	41
3.8.2 switch-Anweisung .....	44
3.9 Schleifen .....	45
3.9.1 for-Schleife .....	45
3.9.2 Foreach-Schleife .....	46
3.9.3 while-Schleife .....	46
3.9.4 do-Schleife .....	46
3.10 Klassen .....	47
3.10.1 Komponenten per Code zuweisen .....	48
3.10.2 Instanziierung von Nichtkomponenten .....	48
3.11 Methoden/Funktionen .....	48
3.11.1 Werttypen und Referenztypen .....	50
3.11.2 Überladene Methoden .....	50
3.12 Lokale und globale Variablen .....	51
3.12.1 Namensverwechslung verhindern mit this .....	51
3.13 Zugriff und Sichtbarkeit .....	51
3.14 Statische Klassen und Klassenmember .....	52
3.15 Parametermodifizierer out/ref .....	53
3.16 Array-Übergabe mit params .....	54
3.17 Eigenschaften und Eigenschaftsmethoden .....	55
3.18 Vererbung .....	56
3.18.1 Basisklasse und abgeleitete Klassen .....	56
3.18.2 Vererbung und die Sichtbarkeit .....	57
3.18.3 Geerbte Methode überschreiben .....	57

3.18.4 Zugriff auf die Basisklasse .....	58
3.18.5 Klassen versiegeln .....	58
3.19 Polymorphie .....	59
3.20 Schnittstellen .....	59
3.20.1 Schnittstelle definieren .....	59
3.20.2 Schnittstellen implementieren .....	60
3.20.3 Zugriff über eine Schnittstellen .....	61
3.21 Namespaces .....	61
3.22 Generische Klassen und Methoden .....	62
3.22.1 List .....	63
3.22.2 Dictionary .....	63
<b>4 Skript-Programmierung .....</b>	<b>65</b>
4.1 MonoDevelop .....	65
4.1.1 Hilfe in MonoDevelop .....	66
4.1.2 Syntaxfehler .....	66
4.2 Nutzbare Programmiersprachen .....	67
4.2.1 Warum C#? .....	67
4.3 Unitys Vererbungsstruktur .....	68
4.3.1 Object .....	69
4.3.2 GameObject .....	69
4.3.3 ScriptableObject .....	69
4.3.4 Component .....	69
4.3.5 Transform .....	70
4.3.6 Behaviour .....	70
4.3.7 MonoBehaviour .....	70
4.4 Skripte erstellen .....	70
4.4.1 Skripte umbenennen .....	71
4.5 Das Skript-Grundgerüst .....	71
4.6 Unitys Event-Methoden .....	72
4.6.1 Update .....	72
4.6.2 FixedUpdate .....	73
4.6.3 Awake .....	73
4.6.4 Start .....	74
4.6.5 OnGUI .....	74
4.6.6 LateUpdate .....	75
4.7 Komponentenprogrammierung .....	75
4.7.1 Auf GameObjects zugreifen .....	76
4.7.2 GameObjects aktivieren und deaktivieren .....	77
4.7.3 GameObjects zerstören .....	78
4.7.4 GameObjects erstellen .....	78
4.7.5 Auf Components zugreifen .....	78
4.7.6 Components hinzufügen .....	80

4.7.7	Components entfernen .....	81
4.7.8	Components aktivieren und deaktivieren .....	81
4.8	Zufallswerte .....	81
4.9	Parallel Code ausführen .....	82
4.9.1	WaitForSeconds .....	83
4.10	Verzögerte und wiederholende Funktionsaufrufe mit Invoke .....	84
4.10.1	Invoke .....	84
4.10.2	InvokeRepeating, IsInvoking und CancelInvoke .....	84
4.11	Daten speichern und laden .....	85
4.11.1	PlayerPrefs-Voreinstellungen .....	85
4.11.2	Daten speichern .....	86
4.11.3	Daten laden .....	86
4.11.4	Key überprüfen .....	87
4.11.5	Löschen .....	87
4.11.6	Save .....	87
4.12	Szeneübergreifende Daten .....	88
4.12.1	Werteübergabe mit PlayerPrefs .....	88
4.12.2	Zerstörung unterbinden .....	89
4.13	Debug-Klasse .....	91
4.14	Kompilierungsreihenfolge .....	92
4.14.1	Programmsprachen mischen und der sprachübergreifende Zugriff .....	92
4.15	Ausführungsreihenfolge .....	93
<b>5</b>	<b>Objekte in der dritten Dimension .....</b>	<b>95</b>
5.1	Das 3D-Koordinatensystem .....	95
5.2	Vektoren .....	96
5.2.1	Ort, Winkel und Länge .....	97
5.2.2	Normalisieren .....	98
5.3	Das Mesh .....	98
5.3.1	Normalenvektor .....	99
5.3.2	MeshFilter und MeshRenderer .....	100
5.4	Transform .....	101
5.4.1	Kontextmenü der Transform-Komponente .....	101
5.4.2	Objekthierarchien .....	102
5.4.3	Scripting mit Transform .....	103
5.4.4	Quaternion .....	104
5.5	Shader und Materials .....	104
5.5.1	Material-Eigenschaften .....	105
5.5.2	Neues Material erstellen .....	108
5.5.3	Normalmaps erstellen .....	108
5.5.4	UV Mapping .....	110
5.6	3D-Modelle einer Szene zufügen .....	111

5.6.1	Primitives .....	111
5.6.2	3D-Modelle importieren .....	112
5.6.3	In Unity modellieren .....	113
5.6.4	Prozedurale Mesh-Generierung .....	114
<b>6</b>	<b>Kameras, die Augen des Spielers .....</b>	<b>115</b>
6.1	Die Kamera .....	115
6.1.1	Komponenten eines Kamera-Objektes .....	117
6.2	Kamerasteuerung .....	117
6.2.1	Statische Kamera .....	117
6.2.2	Parenting-Kamera .....	118
6.2.3	Kamera-Skripte .....	118
6.3	ScreenPointToRay .....	120
6.4	Mehrere Kameras .....	121
6.4.1	Kamerawechsel .....	121
6.4.2	Split-Screen .....	122
6.4.3	Einfache Minimap .....	123
6.4.4	Render Texture .....	125
6.5	Image Effects .....	125
6.6	Skybox .....	127
6.6.1	Skybox selber erstellen .....	127
<b>7</b>	<b>Licht und Schatten .....</b>	<b>129</b>
7.1	Ambient Light .....	129
7.2	Lichtarten .....	130
7.2.1	Directional Light .....	131
7.2.2	Point Light .....	132
7.2.3	Spot Light .....	132
7.2.4	Area Light .....	133
7.3	Schatten .....	134
7.3.1	Einfluss des MeshRenderers auf Schatten .....	135
7.4	Light Cookies .....	135
7.4.1	Import Settings eines Light Cookies .....	136
7.4.2	Light Cookies und Point Lights .....	136
7.5	Light Halos .....	138
7.5.1	Unabhängige Halos .....	138
7.6	Lens Flares .....	139
7.6.1	Eigene Flares .....	139
7.7	Projector .....	140
7.7.1	Standard Projectors .....	140
7.8	Lightmapping .....	141
7.9	Rendering Paths .....	144
7.9.1	Forward Rendering .....	144

7.9.2	Vertex Lit	145
7.9.3	Deferred Lighting	145
<b>8</b>	<b>Physik in Unity</b>	<b>147</b>
8.1	Physikberechnung	147
8.2	Rigidbody	148
8.2.1	Rigidbody kennenlernen	149
8.2.2	Masse Schwerpunkt	150
8.2.3	Kräfte und Drehmomente zufügen	151
8.3	Kollisionen	154
8.3.1	Collider	154
8.3.2	Trigger	158
8.3.3	Static Collider	159
8.3.4	Kollisionen mit schnellen Objekten	159
8.3.5	Terrain Collider	161
8.3.6	Layer-basierende Kollisionserkennung	161
8.3.7	Mit Layer-Masken arbeiten	161
8.4	Wheel Collider	163
8.4.1	Wheel Friction Curve	164
8.4.2	Entwicklung einer Fahrzeugsteuerung	166
8.4.3	Autokonfiguration	172
8.4.4	Fahrzeugstabilität	174
8.5	Physics Materials	175
8.6	Joints	176
8.6.1	Fixed Joint	176
8.6.2	Spring Joint	176
8.6.3	Hinge Joint	177
8.7	Raycasting	177
8.8	Character Controller	178
8.8.1	SimpleMove	179
8.8.2	Move	180
8.8.3	Kräfte zufügen	181
8.8.4	Einfacher First Person Controller	181
<b>9</b>	<b>Maus, Tastatur, Touch</b>	<b>185</b>
9.1	Virtuelle Achsen und Tasten	185
9.1.1	Der Input-Manager	185
9.1.2	Virtuelle Achsen	187
9.1.3	Virtuelle Tasten	187
9.1.4	Steuern mit Mauseingaben	188
9.1.5	Joystick-Inputs	188
9.1.6	Anlegen neuer Inputs	189
9.2	Achsen- und Tasteneingaben auswerten	189

9.2.1	GetAxis .....	189
9.2.2	GetButton .....	190
9.3	Tastatureingaben auswerten .....	190
9.3.1	GetKey .....	191
9.3.2	anyKey .....	191
9.4	Mauseingaben auswerten .....	192
9.4.1	GetMouseButton .....	192
9.4.2	mousePosition .....	192
9.4.3	Mauszeiger ändern .....	193
9.5	Touch-Eingaben auswerten .....	194
9.5.1	Der Touch-Typ .....	195
9.5.2	Input.touches .....	195
9.5.3	TouchCount .....	196
9.5.4	GetTouch .....	196
9.5.5	Touch-Controls .....	196
9.6	Beschleunigungssensor auswerten .....	197
9.6.1	Input.acceleration .....	198
9.6.2	Tiefpass-Filter .....	199
9.7	Steuerungen bei Mehrspieler-Games .....	199
9.7.1	Split-Screen-Steuerung .....	200
9.7.2	Netzwerkspiele .....	200
<b>10</b>	<b>Audio .....</b>	<b>203</b>
10.1	AudioListener .....	203
10.2	AudioSource .....	204
10.2.1	Durch Mauern hören verhindern .....	206
10.2.2	Sound starten und stoppen .....	207
10.2.3	Temporäre AudioSource .....	208
10.3	AudioClip .....	209
10.3.1	3D-Sound und Hintergrundmusik .....	209
10.3.2	Länge ermitteln .....	209
10.4	Reverb Zone .....	210
10.5	Filter .....	211
<b>11</b>	<b>Partikeleffekte mit Shuriken .....</b>	<b>213</b>
11.1	Editor-Fenster .....	214
11.2	Particle Effect Control .....	215
11.3	Numerische Parametervarianten .....	215
11.4	Farbparameter-Varianten .....	216
11.5	Default-Modul .....	216
11.6	Effekt-Module .....	217
11.6.1	Emission .....	218
11.6.2	Shape .....	218

11.6.3	Velocity over Lifetime .....	219
11.6.4	Limit Velocity over Lifetime .....	219
11.6.5	Force over Lifetime .....	220
11.6.6	Color over Lifetime .....	220
11.6.7	Color by Speed .....	220
11.6.8	Size over Lifetime .....	220
11.6.9	Size by Speed .....	221
11.6.10	Rotation over Lifetime .....	221
11.6.11	Rotation by Speed .....	221
11.6.12	External Forces .....	221
11.6.13	Collision .....	221
11.6.14	Sub Emitter .....	223
11.6.15	Texture-Sheet-Animation .....	223
11.6.16	Renderer .....	224
11.7	Partikelemission starten, stoppen und unterbrechen .....	226
11.7.1	Play .....	226
11.7.2	Stop .....	226
11.7.3	Pause .....	227
11.7.4	enableEmission .....	227
11.8	OnParticleCollision .....	227
11.8.1	GetCollisionEvents .....	227
11.9	Feuer erstellen .....	228
11.9.1	Materials erstellen .....	229
11.9.2	Feuer-Partikelsystem .....	229
11.9.3	Rauch-Partikelsystem .....	232
11.10	Wassertropfen erstellen .....	236
11.10.1	Tropfen-Material erstellen .....	236
11.10.2	Wassertropfen-Partikelsystem .....	236
11.10.3	Kollisionspartikelsystem .....	239
11.10.4	Kollisionssound .....	241
<b>12</b>	<b>Landschaften gestalten .....</b>	<b>243</b>
12.1	Was Terrains können und wo die Grenzen liegen .....	244
12.2	Terrainhöhe verändern .....	244
12.2.1	Pinsel .....	245
12.2.2	Oberflächen anheben und senken .....	245
12.2.3	Plateaus und Schluchten erstellen .....	246
12.2.4	Oberflächen weicher machen .....	247
12.2.5	Heightmaps .....	247
12.3	Terrain texturieren .....	249
12.3.1	Textur-Pinsel .....	250
12.3.2	Textures verwalten .....	250
12.4	Bäume und Sträucher .....	252
12.4.1	Bedienung des Place Tree-Tools .....	252

12.4.2	Wälder erstellen . . . . .	253
12.4.3	Mit Bäumen kollidieren . . . . .	253
12.5	Gräser und Details hinzufügen . . . . .	254
12.5.1	Detail-Meshs . . . . .	254
12.5.2	Gräser . . . . .	255
12.5.3	Quelldaten nachladen . . . . .	256
12.6	Terrain-Einstellungen . . . . .	256
12.6.1	Base Terrain . . . . .	257
12.6.2	Resolution . . . . .	257
12.6.3	Tree & Details Objects . . . . .	258
12.6.4	Wind Settings . . . . .	258
12.6.5	Zur Laufzeit Terrain-Eigenschaften verändern . . . . .	259
12.7	Der Weg zum perfekten Terrain . . . . .	260
12.8	Gewässer . . . . .	261
<b>13</b>	<b>Wind Zones . . . . .</b>	<b>263</b>
13.1	Spherical vs. Directional . . . . .	264
13.2	Wind Zone – Eigenschaften . . . . .	265
13.3	Frische Brise . . . . .	266
13.4	Turbine . . . . .	266
<b>14</b>	<b>GUI . . . . .</b>	<b>267</b>
14.1	GUIElements . . . . .	268
14.1.1	GUIElements ausrichten . . . . .	268
14.1.2	GUIText positionieren . . . . .	269
14.1.3	GUITexture skalieren und positionieren . . . . .	269
14.1.4	Interaktivität . . . . .	270
14.2	3DText . . . . .	271
14.3	OnGUI-Programmierung . . . . .	272
14.3.1	GUI . . . . .	273
14.3.2	GUILayout . . . . .	275
14.3.3	GUILayout und UISkin . . . . .	276
14.4	uGUI . . . . .	278
14.4.1	Canvas . . . . .	278
14.4.2	RectTransform . . . . .	281
14.4.3	uGUI-Sprite Import . . . . .	285
14.4.4	Grafische Controls . . . . .	286
14.4.5	Interaktive Controls . . . . .	289
14.4.6	Controls designen . . . . .	294
14.4.7	Animationen in uGUI . . . . .	295
14.4.8	Event Trigger . . . . .	296
14.5	Screen-Klasse . . . . .	297
14.5.1	Schriftgröße dem Bildschirm anpassen . . . . .	298

<b>15</b>	<b>Prefabs .....</b>	<b>299</b>
15.1	Prefabs erstellen und nutzen .....	299
15.2	Prefab-Instanzen erzeugen .....	299
15.2.1	Instanzen per Code erstellen .....	300
15.2.2	Instanzen weiter bearbeiten .....	300
15.3	Prefabs ersetzen und zurücksetzen .....	301
<b>16</b>	<b>Internet und Datenbanken .....</b>	<b>303</b>
16.1	Die WWW-Klasse .....	303
16.1.1	Rückgabewert-Formate .....	304
16.1.2	Parameter übergeben .....	305
16.2	Datenbank-Kommunikation .....	306
16.2.1	Daten in einer Datenbank speichern .....	306
16.2.2	Daten von einer Datenbank abfragen .....	307
16.2.3	Rückgabewerte parsen .....	309
16.2.4	Datenhaltung in eigenen Datentypen .....	310
16.2.5	HighscoreCommunication.cs .....	312
16.2.6	Datenbankverbindung in PHP .....	313
<b>17</b>	<b>Animationen .....</b>	<b>315</b>
17.1	Allgemeiner Animation-Workflow .....	316
17.2	Animationen erstellen .....	316
17.2.1	Animation View .....	317
17.2.2	Curves vs. Dope Sheet .....	318
17.2.3	Animationsaufnahme .....	318
17.2.4	Beispiel Fallgatter-Animation .....	322
17.3	Animationen importieren .....	323
17.3.1	Rig .....	323
17.3.2	Animationen .....	326
17.4	Animationen einbinden .....	329
17.4.1	Animator Controller .....	329
17.4.2	Animator-Komponente .....	334
17.4.3	Beispiel Fallgatter Animator Controller .....	335
17.5	Controller-Skripte .....	337
17.5.1	Parameter des Animator Controllers setzen .....	337
17.5.2	Animation States abfragen .....	338
17.5.3	Beispiel Fallgatter Controller-Skript .....	339
17.6	Animation Events .....	341
17.6.1	Beispiel Fallgatter-Bewegung .....	342
<b>18</b>	<b>Künstliche Intelligenz .....</b>	<b>345</b>
18.1	NavMeshAgent .....	346
18.1.1	Eigenschaften der Navigationskomponente .....	346
18.1.2	Zielpunkt zuweisen .....	347

18.2	NavigationMesh .....	347
18.2.1	Object Tab .....	349
18.2.2	Bake Tab .....	349
18.2.3	Layers Tab .....	350
18.3	NavMeshObstacle .....	350
18.4	Point & Click-Steuerung für Maus und Touch .....	351
<b>19</b>	<b>Fehlersuche und Performance .....</b>	<b>353</b>
19.1	Fehlersuche .....	353
19.1.1	Breakpoints .....	354
19.1.2	Variablen beobachten .....	355
19.1.3	Console Tab nutzen .....	355
19.2	Performance .....	356
19.2.1	Rendering-Statistik .....	356
19.2.2	Analyse mit dem Profiler .....	358
19.2.3	Echtzeit-Analyse auf Endgeräten .....	359
<b>20</b>	<b>Spiele erstellen und publizieren .....</b>	<b>361</b>
20.1	Der Build-Prozess .....	361
20.1.1	Szenen des Spiels .....	362
20.1.2	Plattformen .....	363
20.1.3	Notwendige SDKs .....	363
20.1.4	Plattformspezifische Optionen .....	364
20.1.5	Developer Builds .....	364
20.2	Publizieren .....	365
20.2.1	App .....	365
20.2.2	Browser-Game .....	366
20.2.3	Desktop-Anwendung .....	366
<b>21</b>	<b>Beispiel-Game .....</b>	<b>369</b>
21.1	Level-Design .....	370
21.1.1	Modellimport .....	371
21.1.2	Materials zuweisen .....	372
21.1.3	Prefabs erstellen .....	372
21.1.4	Dungeon erstellen .....	375
21.1.5	Dekoration erstellen .....	378
21.2	Inventarsystem erstellen .....	380
21.2.1	Verwaltungslogik .....	380
21.2.2	Oberfläche des Inventarsystems .....	386
21.2.3	Inventar-Items .....	388
21.3	Game Controller .....	393
21.4	Spieler erstellen .....	393
21.4.1	Lebensverwaltung .....	395
21.4.2	Spielersteuerung .....	403
21.4.3	Wurfstein entwickeln .....	411

21.5	Quest erstellen . . . . .	416
21.5.1	Erfahrungspunkte verwalten . . . . .	416
21.5.2	Questgeber erstellen . . . . .	418
21.5.3	Sub-Quest erstellen . . . . .	425
21.6	Gegner erstellen . . . . .	430
21.6.1	Model-, Rig- und Animationsimport . . . . .	430
21.6.2	Komponenten und Prefab konfigurieren . . . . .	431
21.6.3	Animator Controller erstellen . . . . .	432
21.6.4	NavMesh erstellen . . . . .	434
21.6.5	Umgebung und Feinde erkennen . . . . .	435
21.6.6	Gesundheitszustand verwalten . . . . .	438
21.6.7	Künstliche Intelligenz entwickeln . . . . .	442
21.7	Eröffnungsszene . . . . .	449
21.7.1	Startszene erstellen . . . . .	449
21.7.2	Startmenü erstellen . . . . .	450
21.8	Web-Player-Anpassungen . . . . .	454
21.8.1	Web-Player-Template ändern . . . . .	454
21.8.2	Quit-Methode im Web-Player abfangen . . . . .	455
21.9	Umstellung auf uGUI . . . . .	455
21.9.1	Skriptanpassungen . . . . .	456
21.9.2	Controls erstellen . . . . .	459
21.10	So könnte es weitergehen . . . . .	466
<b>22</b>	<b>Der Produktionsprozess in der Spieleentwicklung . . . . .</b>	<b>469</b>
22.1	Die Produktionsphasen . . . . .	469
22.1.1	Ideen- und Konzeptionsphase . . . . .	470
22.1.2	Planungsphase . . . . .	470
22.1.3	Entwicklungsphase . . . . .	470
22.1.4	Testphase . . . . .	471
22.1.5	Veröffentlichung und Postproduktion . . . . .	471
22.2	Das Game-Design-Dokument . . . . .	471
<b>Schlusswort . . . . .</b>	<b>473</b>	
<b>Index . . . . .</b>	<b>475</b>	

# Vorwort

Für viele von uns sind Computerspiele heutzutage allgegenwärtige Wegbegleiter. Egal wo, auf dem Smartphone, dem Tablet oder dem heimischen PC sind sie installiert und täglich in Benutzung. Manchmal dienen sie als Zeitvertreib, bis der nächste Bus kommt, manchmal sind sie aber auch Bestandteil eines intensiven Hobbys.

Aber nicht nur das Spielen kann Spaß machen, auch das Entwickeln dieser Games kann begeistern. Sowohl im Freizeitbereich als auch in der Arbeitswelt wird der Beruf des Spieleentwicklers immer beliebter. Es ist also kein Wunder, dass sich in den letzten Jahren bereits ganze Studiengänge dem Entwickeln von Computerspielen gewidmet haben.

In diesem Buch möchte ich Ihnen Unity, eine weit verbreitete Entwicklungsumgebung für Computerspiele, näherbringen und erläutern, wie Sie mit diesem Werkzeug Spiele selber entwickeln können. Dabei richtet sich das Buch sowohl an Einsteiger, Umsteiger und auch an Spieleentwickler, die mit Unity nun richtig durchstarten möchten.

An dieser Stelle möchte ich mich ganz besonders bei meiner Frau Cornelia bedanken, die mich während des Schreibens so geduldig unterstützt hat und mir jederzeit beim Formulieren und Korrigieren hilfsbereit zur Seite stand.

Auch danke ich ganz herzlich Alexej Bodemer, der für das Beispiel-Game dieses Buches alle 3D-Modelle, Texturen und Musikdateien entworfen und zur Verfügung gestellt hat.

Zudem gilt mein Dank Will Goldstone und Unity Technologies, die mir die bis dato aktuellsten Beta-Versionen zur Verfügung gestellt haben.

Weiter möchte ich Frau Sieglinde Schärl, Kristin Rothe und dem gesamten Hanser-Verlag-Team danken, die mir nicht nur das Schreiben dieses Buches ermöglicht haben, sondern auch jederzeit mit Rat und Tat zur Seite standen.

Nicht zuletzt danke ich auch meiner gesamten Community, die mich während des Schreibens auf meinem Blog und meinen sozialen Kanälen so konstruktiv begleitet hat.

Süderbrarup, August 2014

*Carsten Seifert*

Lizenziert für gregwyss@hotmail.com.

© 2014 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

# 1

# Einleitung

Computerspiele gehören heutzutage zu den beliebtesten Freizeitgestaltungen unserer Zeit. Mit Zunahme der Popularität ist aber auch der Anspruch an diese Spiele gestiegen. Während früher ein einzelner Programmierer ausreichte, um alle notwendigen Aufgaben zu erledigen, werden heutzutage anspruchsvolle Computerspiele meist von Teams, bestehend aus Modellierern, Grafikern, Sounddesignern, Level-Designern und natürlich auch Programmierern, umgesetzt.

Um den stetig wachsenden Ansprüchen zu genügen, sind aber auch die Werkzeuge der Entwickler ständig mitgewachsen. Eines dieser Werkzeuge ist Unity. Unity ist eine Spieleentwicklungsumgebung für Windows- und Mac OS X-Systeme und wird von der aus Dänemark stammenden Firma Unity Technologies entwickelt. Mit ihr können Sie sowohl interaktive 3D- als auch 2D-Inhalte erstellen. Ich spreche deshalb von Inhalten und nicht nur von Spielen, weil Unity zwar eigentlich für die Entwicklung von 3D-Spielen gedacht war, mittlerweile aber auch immer häufiger Anwendung in anderen Bereichen findet. So wird es beispielsweise für Architekturvisualisierungen genutzt, im E-Learning-Bereich eingesetzt oder in der Digital-Signage-Branche für das Erstellen digitaler Werbe- und Informationssysteme genommen.

Da Unity ursprünglich für die Entwicklung von 3D-Spielen konzipiert wurde, lautet die Internet-Adresse der Firma *unity3d.com*. Dies ist der Grund, weshalb die Entwicklungssoftware auch gerne mal „Unity3D“ genannt wird, was aber eben nicht ganz korrekt ist.

## ■ 1.1 Multiplattform-Publishing

Eine besondere Stärke von Unity ist die Unterstützung von Multiplattform-Publishing. Das bedeutet, dass Sie in Unity ein Spiel einmal entwickeln können, das Sie dann aber für mehrere Plattformen exportieren können. Aktuell werden standardmäßig Windows Desktop-Programme, OS X, Linux, Windows Store Apps, iOS, Android, Windows Phone 8, Blackberry 10, Google Native Client sowie ein eigener Webplayer als Publishing-Format unterstützt.

Der erwähnte Webplayer, auch Unity-PlugIn genannt, ähnelt dabei dem Flash Player und wird im Browser installiert. Mit ihm können Spiele, die mit Unity entwickelt wurden, direkt im Browser ohne große funktionale Einbußen gespielt werden.

Sie können mit Unity auch für die Konsolen XBox, Wii und PlayStation entwickeln. Dafür müssen Sie allerdings extra Lizenzen erwerben, die zum einen nicht günstig und zum anderen nur für Firmen verfügbar sind, die von den jeweiligen Konsolenherstellern auch als Entwickler akzeptiert wurden. Deshalb werde ich die Konsolenentwicklung in diesem Buch auch außen vor lassen und nicht näher beleuchten.

## ■ 1.2 Das kann Unity (nicht)

Unity bringt eine ganze Reihe an nützlichen Werkzeugen mit, um Spiele und andere 3D-Anwendungen zu entwickeln. So gibt es neben einer ausgeklügelten Physik-Engine auch Tools für Partikeleffekte, zur Landschaftsgestaltung oder auch für Animationen. Außerdem wird Unity mit einer extra angepassten Version der Softwareentwicklungsumgebung Mono-Develop ausgeliefert, in der die Programmierung umgesetzt und das Debugging vorgenommen werden kann.

Eines ist Unity allerdings nicht: Es ist keine 3D-Modellierungssoftware. Unity bietet zwar von Haus aus einige 3D-Grundobjekte an, sogenannte Primitives, die für kleinere Aufgaben genutzt werden können, aber für richtige Modellierungsaufgaben sollten Sie auf die dafür entwickelten Spezialtools wie 3ds Max oder das kostenlose Blender zurückgreifen.

## ■ 1.3 Lizenzmodelle

Die Spieleentwicklungsumgebung gibt es in einer kostenlosen Ausführung (auch Free, Indie oder Unity Basic genannt) und in einer kostenpflichtigen Edition. Die kostenlose Version unterstützt bereits alle oben genannten Zielplattformen und erlaubt eine komplette Spieleentwicklung bis hin zum fertigen Spiel. Die kostenpflichtige Version, die auch Unity Pro genannt wird, beinhaltet dann noch weitere Funktionen, um das Spiel grafisch weiter aufzubereiten, die Performance zu optimieren oder Zusatzfunktionen für Animationen und künstliche Intelligenz. Allerdings sind für bestimmte Plattformen wie iOS, Android und Blackberry 10 noch weitere Lizenzen notwendig, um die zusätzlichen Pro-Features auch dort zu erhalten.

Da sowohl Unity Pro als auch die Indie-Version bzw. Unity Basic kommerziell genutzt werden dürfen, gibt es beim Einsatz der kostenlosen Version die Vorgabe, dass nur Firmen bzw. Entwickler diese einsetzen dürfen, die nicht mehr als 100 000 US-Dollar Umsatz in einem Geschäftsjahr machen.

## ■ 1.4 Aufbau und Ziel des Buches

Mit diesem Buch werden Sie lernen, auf Basis von Unity eigene 3D-Spiele zu entwickeln. Sie werden sich mit den unterschiedlichen Tools der Game Engine vertraut machen und die Skript-Programmierung erlernen. Dabei steht aber nicht das Ziel im Fokus, wirklich alle Funktionen und Möglichkeiten zu beleuchten, die Unity dem Entwickler anbietet. Vielmehr zeige ich Ihnen, wie die unterschiedlichen Bereiche von Unity funktionieren und miteinander zusammenarbeiten. Denn der Funktionsumfang dieser Spieleentwicklungsumgebung ist mittlerweile so umfangreich geworden, dass es gar nicht mehr möglich ist, alle Tools und deren Möglichkeiten bis ins letzte Detail in einem einzigen Buch ausführlich zu behandeln. Daher liegt der Schwerpunkt auf den Kernfunktionen, die in der kostenlosen Unity-Version für die 3D-Spieleentwicklung bereitgestellt werden, ergänzt um Anwendungsbeispiele und Praxistipps.

Möchten Sie weiter in die Tiefe eines speziellen Tools gehen oder mehr Informationen zu bestimmten Scripting-Klassen erhalten, empfehle ich Ihnen die mit Unity mitgelieferten Hilfe-Dokumente, die Sie über das Help-Menü erreichen. Dort finden Sie sowohl ein ausführliches Manual über die in Unity integrierten Tools sowie eine *Scripting-Referenz* über alle Unity-Klassen und deren Möglichkeiten. Letztere finden Sie auch über die Hilfe von MonoDevelop, der mitgelieferten Programmierumgebung.

Spieleentwicklung wird häufig auch als Spieleprogrammierung bezeichnet, auch wenn viele Aufgaben in der Spieleentwicklung heutzutage nicht mehr programmiert, sondern mithilfe von Tools erledigt werden. Nichtsdestotrotz ist der Programmieranteil bei der Entwicklung eines Spiels doch immer noch sehr hoch. Deshalb wird auch das Buch zunächst mit zwei größeren programmierbezogenen Kapiteln beginnen, wo es unter anderem auch um die Grundlagen der Programmierung geht. Erst danach werden wir in die 3D-Welt von Unity eintauchen und die verschiedenen Werkzeuge behandeln. Der Aufbau ist deshalb so gewählt, weil ich in den folgenden Kapiteln immer wieder kleinere Skript-Beispiele zeige, die Einsatzmöglichkeiten in der Praxis demonstrieren.

Am Ende des Buches werden wir schließlich in einem etwas größeren Kapitel ein Beispiel-Game entwickeln, das viele der behandelten Themen noch einmal aufgreift und die gesamten Zusammenhänge in der Praxis zeigt.



### Die DVD zum Buch

Dem Buch liegt eine DVD bei, auf der sich Folgendes befindet:

- Beispiel-Game (Dungeon Crawler) mit allen dazugehörigen Ressourcen
- Anwendungsbeispiel für eine Auto-Steuerung inklusive eines 3D-Modells mit zusätzlichen Skripten
- Beispiel für eine Sprite-Animation
- Vorlage für ein einfaches Übungsprojekt
- ergänzende Video-Tutorials, in denen die Verwendung spezieller Werkzeuge und Verfahren in der Praxis demonstriert wird.

Im Buch werde ich Hinweiskästen, wie den obigen, einsetzen, um Hinweise und Tipps zu geben. Je nach Typ des Hinweises werden diese mit unterschiedlichen Icons ausgezeichnet.



#### Hinweise zu Inhalten auf der DVD



#### Hinweise zu weiterführenden Inhalten im Internet



#### Praxistipps



#### Allgemeine Hinweise

## ■ 1.5 Weiterentwicklung von Unity

Die Spieleindustrie gehört zu den Branchen, die sich aktuell am schnellsten verändern. Kein Wunder also, dass auch Unity ständig weiterentwickelt wird und neue Funktionen erhält. Sollten Sie Unterschiede zwischen dem Buch und Ihrer Unity-Version erkennen, wird dies sicher der ständigen Weiterentwicklung von Unity geschuldet sein.

Aber nicht nur der Funktionsumfang wird ständig weiterentwickelt, auch das Lizenzmodell von Unity ist nicht von Veränderungen ausgeschlossen. Vielleicht können Sie schon Funktionen mit Unity Basic nutzen, die in diesem Buch noch als Pro-Feature gelten. Ein gutes Beispiel hierfür sind die Berechtigungen für die Mobile-Entwicklung. Noch vor Kurzem hätten Sie bereits für die Basic-Mobile-Entwicklung extra Lizenzen kaufen müssen. Diese Berechtigungen sind jetzt bereits in der kostenlosen Version integriert. Und auch das Add-On für das Pathfinding (siehe Kapitel „Künstliche Intelligenz“) war anfangs ein Pro-Feature, heutzutage ist es fast komplett in der Indie-Version enthalten.

## ■ 1.6 Online-Zusatzmaterial

Aufgrund der kontinuierlichen Weiterentwicklung von Unity werde ich Ihnen in einem passwortgeschützten Bereich meiner Website Zusatzmaterialien bereitstellen, die den Inhalt dieses Buches ergänzen und auch aktuell halten sollen.



#### Passwortgeschützter Bereich für Zusatzmaterialien

URL: <http://www.hummelwalker.de/buch-zusatzmaterial/>

Passwort: uN1TyZuS4TzMaT3RiAl

# 2

# Grundlagen

In diesem Kapitel werden Sie die Oberfläche von Unity sowie deren grundlegende Bedienung kennenlernen. Wir werden auf die Struktur eines Unity-Projektes eingehen und die grundlegenden Prinzipien von Unity behandeln.

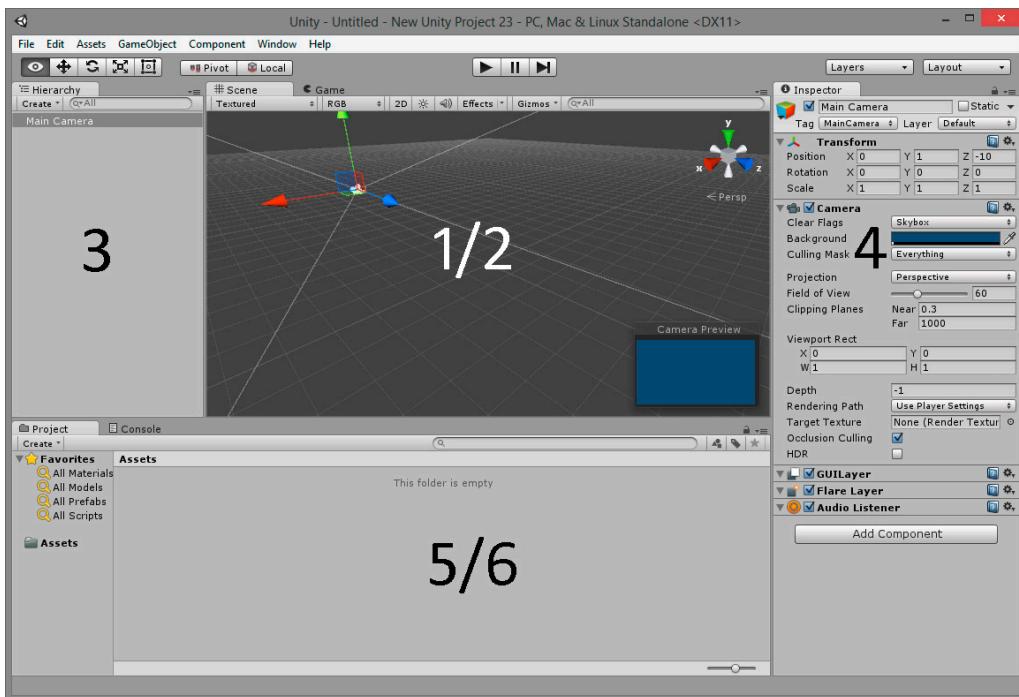
## ■ 2.1 Installation

Sollten Sie noch keine Installationsdatei haben, besuchen Sie zunächst die Seite <http://www.unity3d.com> und laden sich dort die aktuelle Unity-Version herunter. Nach dem Download können Sie Unity installieren. Da die Installation eigentlich selbsterklärend ist, sollten Sie lediglich darauf achten, dass Sie eine Komplettinstallation machen und keine Teile davon ausnehmen. Profis können natürlich selber bewerten, was sie benötigen, Anfängern würde ich aber immer eine Komplettinstallation empfehlen.

Wenn Sie die Installation abgeschlossen haben und Unity das erste Mal starten, wird sich der *Project Wizard* von Unity zeigen. Über diesen können Sie ein existierendes Projekt öffnen oder ein neues Projekt erstellen. Mehr zum Öffnen und Erstellen eines Projektes erfahren Sie im Abschnitt „Das Unity-Projekt“. Für den ersten Start können Sie einfach die Standardeinstellungen übernehmen und über **Create** ein neues Projekt erstellen.

## ■ 2.2 Oberfläche

Die Oberfläche von Unity besteht aus mehreren frei anpassbaren Fenstern (auch Tabs genannt), die sich innerhalb von Unity übereinander und nebeneinander andocken sowie auch außerhalb des Hauptfensters verschieben lassen.



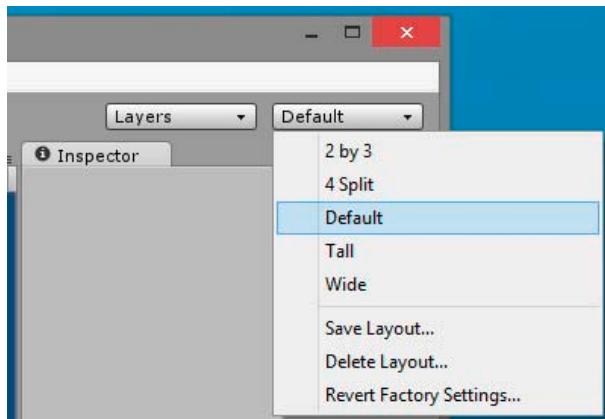
**Bild 2.1** Default-Ansicht von Unity

Die Fenster (Tabs) besitzen zwar unterschiedliche Aufgaben, gehören funktional aber alle zusammen und werden deshalb auch kontinuierlich untereinander synchronisiert.

1. **Scene View** dient dem interaktiven Gestalten von Szenen und 3D-Welten.
2. **Game View** dient als Vorschau des fertigen Spiels. Dieses Fenster wird in Bild 2.1 von der *Scene View* verdeckt, wird aber von Unity automatisch nach vorne gebracht, wenn das Spiel gestartet wird.
3. **Hierarchy** zeigt alle in der Szene existierenden Objekte (*GameObjects*) in deren Hierarchiestruktur an.
4. **Inspector** zeigt alle Komponenten und öffentlichen Parameter des aktuell selektierten *GameObjects* an.
5. **Project Browser** dient dem Anzeigen und Verwalten aller digitalen Inhalte (auch *Assets* genannt), die zu dem Projekt gehören.
6. **Console** dient dem Anzeigen von Fehler- und Hinweismeldungen. Dieses Fenster befindet sich in der Default-Ansicht hinter dem *Project Browser*. Um die Meldungen zu sehen, müssen Sie auf den Reiter des Fensters klicken.

Abgesehen von der *Game View* und dem *Console-Tab* können Sie den Tabs nicht nur Informationen entnehmen, sondern auch die Objekte verändern.

Sie können die Tab-Anordnungen in Unity jederzeit speichern und auch zurücksetzen. Hierfür stellt Unity im oberen Hauptmenü über **Window/Layouts** entsprechende Funktionen und Standardanordnungen zur Verfügung. Als Schnellzugriff dient hier das *Layouts-Dropdown-Menü*, das Sie ganz rechts im oberen Bereich von Unity finden.

**Bild 2.2**

Schnellzugriff auf die Layout-Funktionen

Wenn Sie mit **Save Layout** eigene Fenster-Anordnungen speichern, stehen diese Layouts nicht nur in diesem Projekt zur Verfügung, sondern auch in allen anderen Unity-Projekten. Mit **Delete Layout** löschen Sie diese natürlich ebenfalls in allen Projekten. Sollten Sie dabei eine Standardsicht aus Versehen gelöscht haben, können Sie über **Revert Factory Settings** alle Layout-Einstellungen wieder auf die Werkseinstellungen zurücksetzen.

## 2.2.1 Hauptmenü

Oberhalb aller Subfenster und Toolbars befindet sich das Hauptmenü von Unity. Viele Teile dieses Menüs finden Sie zusätzlich noch einmal als Schnellzugriff-Menüs in anderen Bereichen von Unity wieder, wie z.B. das *Layouts-Drop-down-Menü* (siehe oben).

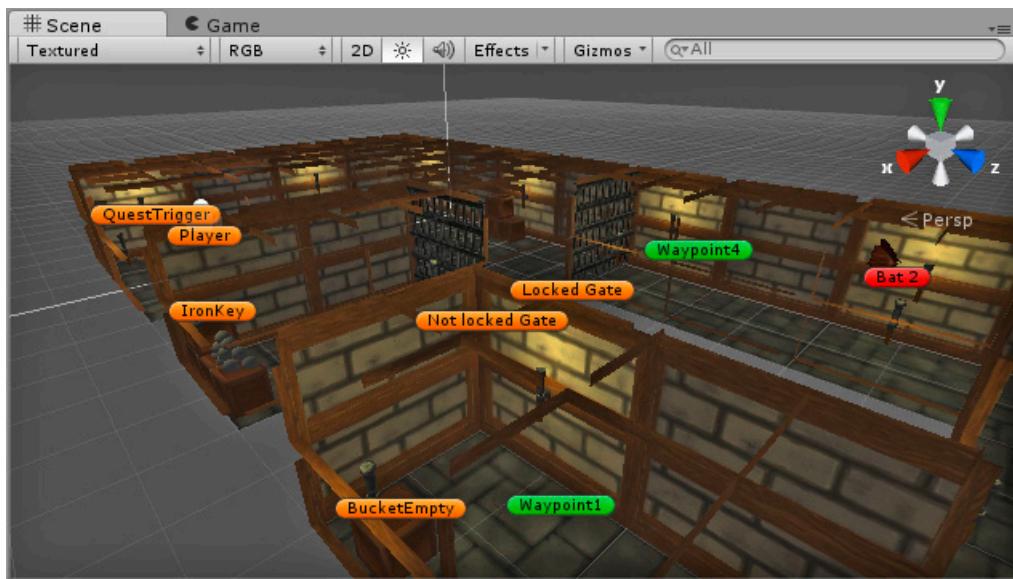
**Bild 2.3** Hauptmenü

Hinter diesen Hauptmenüpunkten verbergen sich folgende Funktionen:

- **File** beinhaltet alle Funktionen, die sich mit dem Erstellen und Speichern von Dateien auf Projekt- und Szene-Ebene beschäftigen.
- **Edit** besitzt vor allem Einstellungen zum Verändern von Daten.
- **Asset** beschäftigt sich mit dem Verwalten und Erstellen neuer *Assets*, z.B. von Skripten und Materialien.
- **GameObject** beschäftigt sich vor allem mit dem Erstellen verschiedener, vorkonfigurierter *GameObjects*.
- **Component** bietet alle Standardkomponenten von Unity an.
- **Window** bietet vor allem weitere Fenster/Tabs an, die Sie anzeigen können.
- **Help** besitzt hauptsächlich Quellen und Links zu weiterführenden Informationen für Unity. Hier gelangen Sie auch zum „Unity Manual“ und zu der „Scripting Reference“, wo Sie viele weiterführende Informationen finden.

## 2.2.2 Scene View

Das Fenster mit der Überschrift „Scene“ wird als *Scene View* bezeichnet und dient dem interaktiven Positionieren und Verändern von Objekten innerhalb einer Szene bzw. eines Levels. Sie können hier neue Objekte hinzufügen, diese verschieben, rotieren und auch skalieren. Ein wichtiges Werkzeug hierfür sind die *Transform-Tools*, die Sie in der Toolbar direkt unter dem Hauptmenü oben links finden. Über Maus und Tastatur können Sie zudem durch die Szene navigieren (siehe Abschnitt 2.2.2.1, „Navigieren in der Scene View“).



**Bild 2.4** Scene View

Möchten Sie ein neues *GameObject* in der *Scene View* platzieren, können Sie einfach ein *GameObject* aus dem *Project Browser* in die *Scene View* hineinziehen und fallen lassen. Anschließend können Sie es noch verschieben und mit den erwähnten *Transform-Tools* modifizieren (siehe Abschnitt 2.2.4, „Toolbar“).

Am oberen Rand der *Scene View* befindet sich eine schmale *Control Bar*, die Ihnen über mehrere Untermenüs verschiedene Einstellmöglichkeiten bietet (siehe Bild 2.4).

- **Mesh Rendering-Menü** wechselt die Ansicht zwischen verschiedenen Darstellungsarten (Texturiert, Drahtgitter usw.).
- **Texture Rendering-Menü** stellt verschiedene Darstellungsformen der Texturen zur Verfügung (RGB, Alpha ...).
- **2D-Button** wechselt zwischen der normalen 3D-Ansicht und einer speziellen Darstellungsform, die besonders für 2D-Games und das GUI-Design geeignet ist. Mehr zu dieser Ansicht erfahren Sie im Abschnitt 2.2.2.2.
- **Beleuchtungs-Button** schaltet die Darstellung der Lichtquellen bzw. deren Auswirkungen in der *Scene View* ein/aus.

- **Sound-Button** schaltet den Ton ein/aus.
- **Effects-Menü** ermöglicht, bestimmte Effekte in der Szene zu aktivieren/deaktivieren (Skybox, Flares ...).
- **Gizmos-Menü** schaltet typbezogen Hilfsgrafiken in der *Scene View* ein/aus (Gizmos sind grafische Hilfsmittel, um nicht sichtbare Objekte wie Kameras, Soundquellen etc. darzustellen).
- **Suchmaske**, um Objekte in der Szene zu suchen (dabei wird alles ausgegraut, außer die gefundenen Objekte)

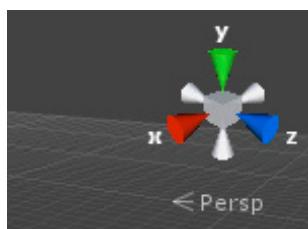
### 2.2.2.1 Navigieren in der Scene View

Unity bietet unterschiedliche Möglichkeiten, durch die *Scene View* zu navigieren. Manche Funktionen sind hierbei auch auf unterschiedliche Weise zu erreichen. Im Folgenden möchte ich Ihnen die wichtigsten vorstellen.

- **Objekte selektieren** Sie mit der linken Maustaste.
- **Ansicht in der Höhe und Seite verschieben** Sie mit der mittleren Maustaste bzw. durch Drücken auf das Mausrad. Sie können auch über die Pfeiltasten navigieren oder alternativ auch das Verschiebewerkzeug der *Transform-Tools* aktivieren (Shortcut: `Q`) und dann die linke Maustaste nutzen (siehe *Transform-Tools*). Wenn Sie große Szenen besitzen, können Sie die Geschwindigkeit des Verschiebens über das zusätzliche Drücken der `[Umsch]-Taste` (`Shift`) noch erhöhen.
- **Ansicht in der Tiefe und Seite verschieben** Sie über die Pfeiltasten. Hierbei bewegen Sie sich den Pfeiltasten entsprechend durch die Szene.
- **Ansicht drehen** Sie über die rechte Maustaste. Alternativ können Sie auch die Kombination aus `[Alt]-Taste` und der linken Maustaste nutzen.
- **Ansicht zoomen** Sie über das Drehen des Mausrads. Alternativ können Sie auch die `[Alt]-Taste` und die rechte Maustaste drücken, während Sie den Mauszeiger bewegen.
- **Objekte fokussieren** Sie, indem Sie ein Objekt in der *Scene View* oder in der *Hierarchy* selektieren und dann die Taste `F` drücken. Die Ansicht wird dann so gesetzt, dass das Objekt im Mittelpunkt steht.

### 2.2.2.2 Scene Gizmo

Oben rechts innerhalb der *Scene View* finden Sie das *Scene Gizmo*, das die Orientierung des Koordinatensystems zur aktuellen Sicht der *Scene View* darstellt. Das *Scene Gizmo* ist dabei wie ein Kompass und zeigt Ihnen, von welcher Seite Sie aktuell die Szene betrachten.

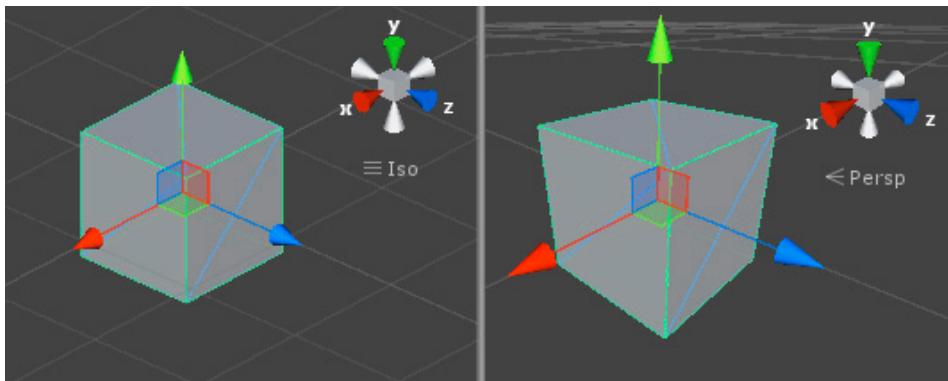


**Bild 2.5**  
Scene Gizmo

Wenn Sie auf die unterschiedlichen Achsen des *Scene Gizmos* klicken, wechselt die Ansicht dabei in die jeweilige Seiten-, Drauf- oder Frontansicht. Ein Klick auf den mittleren Würfel ändert zudem die Ansicht zwischen orthogonal und perspektivisch.

- **Orthogonal** ignoriert Tiefenverhältnisse. So werden beispielsweise zwei gleich große Objekte immer gleich groß dargestellt, egal wie weit diese vom Betrachter entfernt sind.
- **Perspektivisch** stellt Objekte „natürlich“ dar und berücksichtigt die Entfernung vom Betrachter.

Unter dem *Scene Gizmo* wird Ihnen in Textform zusätzlich angezeigt, welche Ansicht Sie aktuell gewählt haben. Die drei parallel verlaufenden Linien bedeuten hierbei orthogonal, die auseinandergehenden Linien bedeuten perspektivisch. Ein Klick auf diesen Text wechselt ebenfalls zwischen diesen beiden Ansichtsformen.



**Bild 2.6** Vergleich: Orthogonal (links) vs. Perspektivisch (rechts)

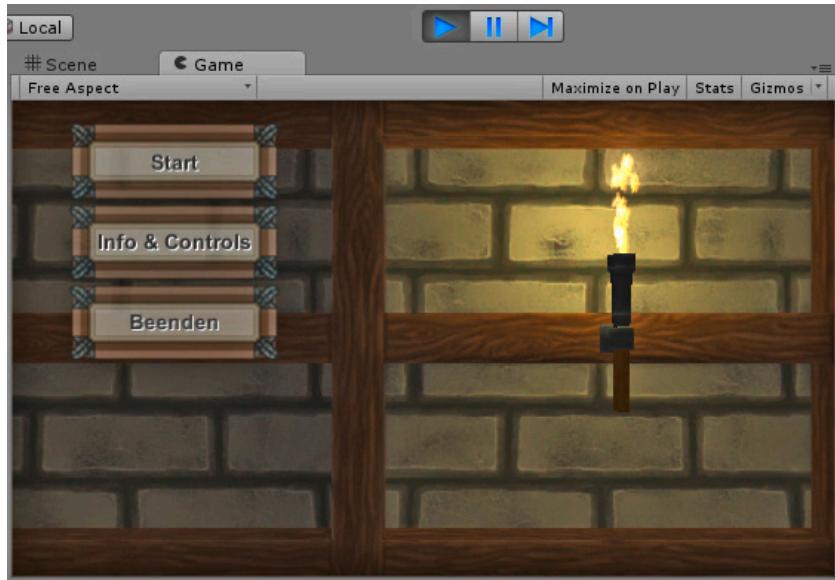


### 2D-Button

Der *2D-Button*, den Sie in der *Control Bar* der *Scene View* finden, ändert lediglich die Ansicht auf „Orthogonal“ und dreht die räumliche Betrachtung so, dass der Nutzer in Richtung der positiven Z-Achse schaut, während die Y-Achse nach oben zeigt. Er ruft also nur eine Standardansicht auf, die sich für 2D-Spiele als praktisch erwiesen hat. Ansonsten nimmt dieser Button keinen Einfluss auf das Spiel.

## 2.2.3 Game View

Die *Game View* dient dem Testen Ihres Projektes. Sie zeigt das Spiel aus Sicht der Kamera und ermöglicht Ihnen, das Spiel so zu betrachten, wie es später nach der Erstellung aussehen wird. Wenn Sie das Spiel über die Play-Taste starten, können Sie über dieses Fenster Ihr Spiel testen.



**Bild 2.7** Game View

Wie die *Scene View* besitzt auch dieses Fenster am oberen Rand eine schmale *Control Bar* mit verschiedenen Funktionen.

- **Size-Menü** gibt Ihnen die Möglichkeit, zum Testen das Bildschirmformat vorzugeben. Es können Seitenverhältnisse sowie feste Pixelwerte vorgegeben werden, wie das Bild in der *Game View* dargestellt werden soll.
- **Maximize on Play-Button** wechselt bei Aktivierung beim Testen automatisch in einen Fullscreen-Modus.
- **Stats-Button** blendet eine kleine Statistik in der *Game View* ein, die verschiedene Performance-Daten darstellt.
- **Gizmos-Menü** schaltet typbezogen die Hilfsgrafiken in der *Game View* dazu.

## 2.2.4 Toolbar

Direkt unter dem eigentlichen Hauptmenü finden Sie die Toolbar mit ihren verschiedenen Werkzeugen.

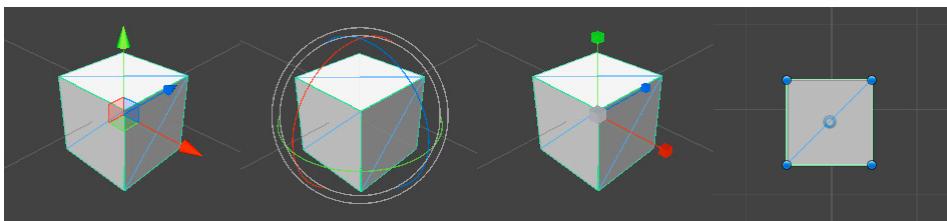


**Bild 2.8** Toolbar

## Transform-Tools

Ganz links liegen die **Transform-Tools**, mit denen Sie das aktuell selektierte Objekt in der *Scene View* verändern können. Je nach Tool wechselt dabei das Handle des selektierten *GameObjects*, mit dem Sie dann das Objekt dementsprechend verändern können.

- **Hand-Tool** (Hand) dient keiner direkten Transform-Modifikation. Es erlaubt Ihnen das Verschieben der *Scene View*-Perspektive per Maus und gedrückter linker Maustaste.
- **Translate-Tool** (Kreuz) verschiebt das selektierte Objekt in der Szene.
- **Rotate-Tool** (drehende Pfeile) dreht das Objekt.
- **Scale-Tool** skaliert das Objekt bzw. Mesh in die verschiedenen Achsen. Selektieren Sie hierfür eins der drei Enden und ziehen Sie dieses in die jeweilige Richtung. Über den grauen Cube in der Mitte des Objektes skalieren Sie das Objekt in alle Richtungen gleichzeitig.
- **Rect-Tool** ermöglicht, das Objekt an den Rechteck-Handles zu verändern. Dieses Tool eignet sich vor allem zum Modifizieren von Texturen des Typs „Sprite“ und anderen 2D-Objekten. Durch das Verschieben einer Ecke verändern Sie die Größe des Objektes. Hier können Sie durch das zusätzliche Halten der **[Umsch]**-Taste auch alle Seiten gleichzeitig skalieren. Bewegen Sie Ihren Mauszeiger (ohne gedrückte Maustaste) bei einem Handle etwas nach außen, erscheint ein Rotations-Symbol. Wenn Sie nun die linke Maustaste drücken, können Sie das Objekt um den *Pivot-Punkt* drehen.



**Bild 2.9** Handles der unterschiedlichen Transform-Tools (Translate-, Rotate-, Scale- und Rect-Tool)

Ein besonderes Feature des *Translate-Tools* ist das **Vertex-Snapping**. Mit diesem können Sie auf einfache Weise Modelle nebeneinander platzieren. Selektieren Sie hierfür ein Objekt in der Szene und drücken Sie anschließend die Taste **[V]**. Nun können Sie durch Bewegen der Maus einen Vertex des Objektes auswählen (halten Sie hierbei keine Maustaste gedrückt!). Wenn Sie einen passenden gefunden haben, drücken Sie die linke Maustaste und bewegen die Maus zu einem Vertex eines anderen Objektes, wo das Objekt schließlich positioniert werden soll. Das erste Objekt wird nun Vertex an Vertex platziert.

Zu den *Transform-Tools* gehören auch die beiden daneben liegenden Funktionsknöpfe, die sogenannten **Gizmo Display Toggles**.

Mit diesen Toggle-Buttons können Sie folgende Funktionen ausführen:

- **Pivot/Center** definiert das Transformationszentrum. *Pivot* ist hierbei der Original-Nullpunkt des Modells, *Center* nimmt die Mitte des gerenderten Objektes.
- **Local/Global** legt fest, ob sich die Transformationsachsen auf das lokale oder das globale Koordinatensystem beziehen sollen.

 **Transform-Tools**

Eine Demonstration aller *Transform-Tools* und des „Vertex-Snappings“ finden Sie in Video-Form auf der DVD. In diesem Zusammenhang werden auch *Sprites* näher erläutert.

## Play Controls

Der nächste Abschnitt sind die **Play Controls**. Mit diesen können Sie Ihr Spiel in der *Game View* testen.

Ein erstes Drücken auf die Play-Taste startet das Spiel (Play-Modus), ein zweiter Klick beendet es. Der zweite Button der *Play Controls* pausiert das Spiel und der dritte spielt es in kleinen Schritten ab. Letzteres eignet sich beispielweise zum Testen von Kollisionen, Animationen oder Partikeleffekten.

 **Änderungen im Play-Modus**

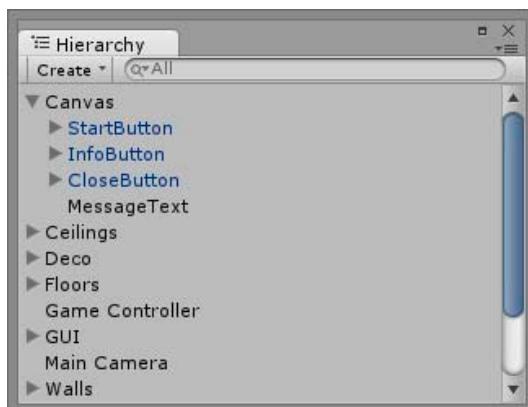
Alle Änderungen, die Sie im Play-Modus in der Szene machen, werden nach dem Stoppen wieder rückgängig gemacht.

Als Nächstes folgt das **Layers Drop-down-Menü**. Über dieses können Sie die *GameObjects* dieser Layer in der *Scene View* sichtbar und unsichtbar schalten.

Ganz am Ende finden Sie schließlich das *Layouts-Drop-down-Menü*, über das Sie den Aufbau der Subfenster von Unity speichern, laden und verwalten können.

## 2.2.5 Hierarchy

Die *Hierarchy* zeigt alle *GameObjects* der aktuellen Szene an. Unity bei der Darstellung zwischen normalen *GameObjects* und *Prefab*-Instanzen, die blau dargestellt werden. *Prefabs* sind Kopien einer Vorlage und besitzen zusätzliche Funktionalitäten (siehe „*Prefabs*“).



**Bild 2.10**  
Hierarchy-Tab

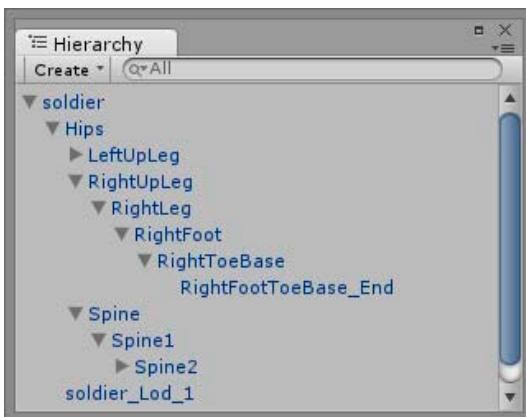
Das *Hierarchy*-Fenster und die *Scene View* werden ständig synchronisiert, sodass ein selektiertes Objekt in der *Scene View* auch gleichzeitig in der *Hierarchy* markiert ist. Sie können auch ein *GameObject* aus dem *Project Browser* direkt in die *Hierarchy* ziehen, um es in einer Szene zu platzieren. Allerdings wird es selten dort erscheinen, dass es auch in der *Scene View* zu sehen ist. Über das anschließende Drücken der Taste **F** springt der Fokus zu dem neuen Objekt (siehe „Navigieren in der Scene View“).

Das *Hierarchy*-Fenster besitzt oben links ein Drop-down-Menü, das einen Schnellzugriff auf **GameObject/Create General** erlaubt und Ihnen die Möglichkeit bietet, schnell neue *GameObjects* Ihrem Projekt zuzufügen. Rechts neben dem Menü finden Sie eine Suchmaske, um nach *GameObjects* zu filtern.

Außerdem können Sie über das Kontextmenü der rechten Maustaste *GameObjects* löschen, umbenennen, kopieren und noch einiges mehr. An dieser Stelle möchte ich noch kurz darauf hinweisen, dass das Duplizieren eines Objektes oder eines Assets in Unity generell über das Tastenkürzel **Strg**+**D** funktioniert, nicht Windows-typisch über **Strg**+**C** und **Strg**+**V**.

### 2.2.5.1 Parenting

Ein wichtiges Konzept von Unity ist das sogenannte *Parenting* bzw. Eltern-Kind-Konzept. Bei diesem kann ein beliebiges *GameObject* unter einem anderen Objekt hängen und erbt von diesem die relativen Transform-Eigenschaften wie Position und die Rotation. Besitzt ein *GameObject* ein oder mehrere Unterobjekte, wird dies durch ein kleines Dreieck am linken Rand dargestellt. Wenn Sie auf dieses klicken, werden die Unterobjekte wie in einer Ordnerstruktur auf- und zugeklappt.



**Bild 2.11**

Struktur eines Soldaten mit mehreren Kind-Objekten

Die im *Inspector* angezeigten Werte der *Transform*-Komponente sind bei einem Kind-Objekt immer lokale Werte, sie stellen die Position, Rotation, Skalierung also immer relativ zum Eltern-Objekt dar. Beträgt die Position eines Kind-Objektes nun (0, 0, 0), so befindet es sich im Zentrum des Eltern-Objektes. Verschieben Sie dieses Eltern-Objekt, wird das Kind-Objekt zwar mitverschoben, die Position bleibt aber im *Inspector* bei (0, 0, 0).

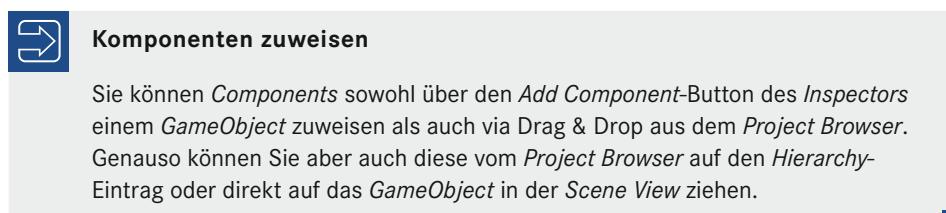
## 2.2.6 Inspector

Im *Inspector* werden alle Informationen und Komponenten des aktuell selektierten *GameObjects* angezeigt. Hier können Sie Komponenten hinzufügen, entfernen und Einstellungen an diesen vornehmen.



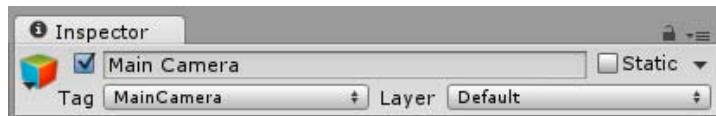
**Bild 2.12**  
Inspector-Beispiel

Am Ende der Komponenten finden Sie schließlich noch den Button **Add Component**, über den Sie Komponenten suchen und diesem *GameObject* zuweisen können. Alternativ können Sie Standardkomponenten von Unity auch über das Hauptmenü im Bereich **Component** finden und zufügen.



### 2.2.6.1 Kopfinformationen im Inspector

Im Kopf des *Inspectors* finden Sie neben dem Namen des *GameObjects* viele weitere Informationen und Einstellungsmöglichkeiten.



**Bild 2.13**  
Kopfinformationen  
im Inspector

Ganz links können Sie ein **Icon** auswählen, das in der *Scene View* an der Position des *GameObjects* dargestellt werden soll. Klicken Sie hierfür auf das angezeigte Symbol und wählen Sie ein Icon aus. Das Zeichen wechselt dann auf das ausgewählte **Icon**. Ein Würfel-Symbol, wie in Bild 2.13 zu sehen, bedeutet, dass kein spezielles **Icon** angezeigt werden soll.

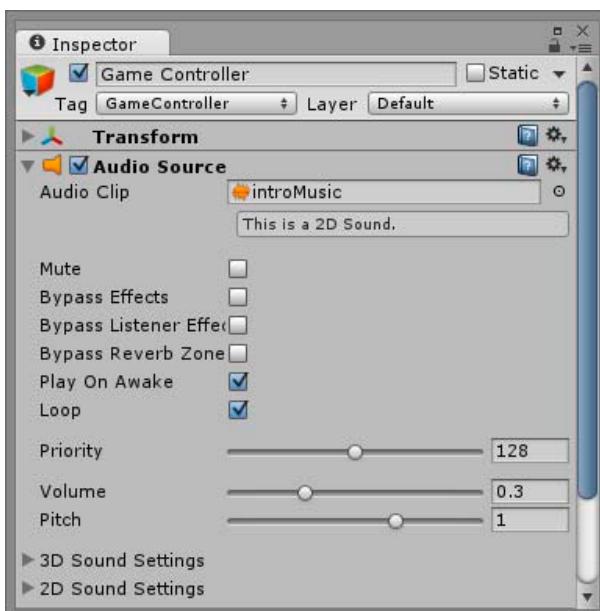
Daneben finden Sie eine Checkbox, über die Sie das gesamte Objekt mit all seinen Komponenten aktivieren und deaktivieren können. Deaktivierte Objekte werden in der *Hierarchy* grau dargestellt und in der *Scene View* nicht mehr angezeigt.

Rechts neben dem Namen finden Sie die **Static**-Checkbox. Über diese können Sie definieren, ob ein Objekt statisch ist, also ob es sich in seiner örtlichen Lage verändert bzw. verändert kann. Diese Einstellung ist lediglich für einige Spezialfunktionen wichtig, die sich auf nicht bewegende Objekte beziehen. Für welche Funktionen dieser Kenner genau gelten soll, können Sie zudem über das rechte Drop-down-Menü zusätzlich einschränken (dargestellt durch den kleinen, nach unten zeigenden Pfeil).

In der zweiten Zeile können Sie dem *GameObject* zudem noch einen **Tag** und einen **Layer** zuweisen, auf die wir später noch zu sprechen kommen.

### 2.2.6.2 Öffentlichen Variablen Werte zuweisen

Alle Parameter, die Sie im *Inspector* verändern können, sind sogenannte *öffentliche Variablen*. Solchen „public“ Parametern können Sie aber nicht nur Werte in Form von Zahlen und Texten zuweisen, Sie können diesen auch Elemente aus dem *Project Browser* oder aus der aktuellen Szene zuweisen – je nach Typ der Variablen eben. So besitzt z. B. jede  *AudioSource* einen Parameter namens  *AudioClip*, dem die abzuspielende Audiodatei zugewiesen wird. Nehmen Sie einfach eine beliebige Audiodatei aus dem *Project Browser* und ziehen Sie diese auf die  *AudioClip*-Variable.



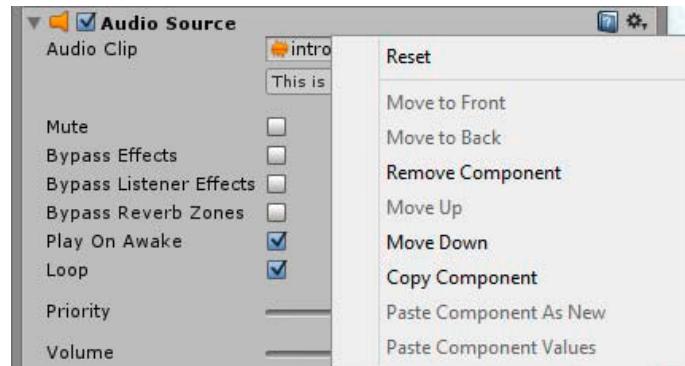
**Bild 2.14**  
Darstellung öffentlicher Variablen im Inspector

Anstatt solche Objekte per Drag & Drop einer Variablen zuzuweisen, können Sie dies auch über ein Menü machen, das Sie über ein Kreis-Symbol neben dem jeweiligen Parameter erreichen (siehe Bild 2.14). Dieses öffnet einen Dialog, aus dem Sie dann auch das passende Objekt auswählen können. Möchten Sie hierbei der Variablen keinen Inhalt zuweisen, wählen Sie einfach den Eintrag *None* aus, der jedes Mal mit angeboten wird.

### 2.2.6.3 Komponenten-Menüs

Jede Komponente (z.B. AudioSource, Transform) eines *GameObjects* besitzt ein Menü mit einigen Grundsatzfunktionen zum Anordnen, Zurücksetzen und Löschen der jeweiligen Komponente. Mit dem Befehl **Reset** setzen Sie z.B. alle Werte auf Default-Einstellungen zurück.

Dieses Menü erreichen Sie über das Zahnrad-Symbol im oberen rechten Bereich einer Komponente. Parallel dazu stehen die gleichen Funktionen auch über das Kontextmenü der rechten Maustaste zur Verfügung.

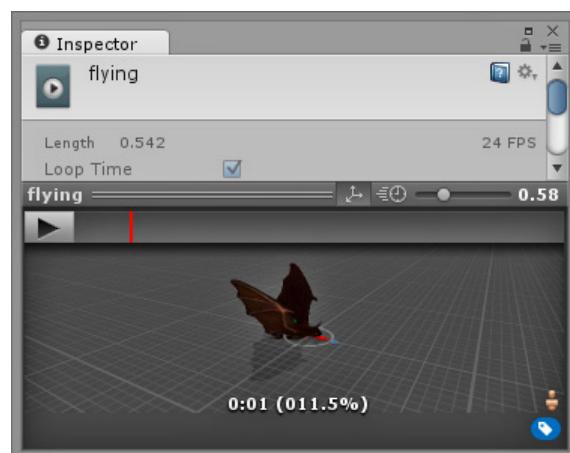


**Bild 2.15**

Komponenten-Menü  
im Inspector

### 2.2.6.4 Preview-Fenster

Wenn Sie im Project Browser ein Asset selektieren, erscheint im *Inspector* noch ein zusätzliches *Preview-Fenster*, in dem eine Vorschau des markierten Assets angezeigt wird. Je nach Asset-Typ bietet das Fenster noch zusätzliche Funktionen an. So wird z.B. bei einer Animation ein Start-Button angeboten, um diese zu starten, und ein Geschwindigkeitsregler, der die Schnelligkeit des Abspielens steuert.



**Bild 2.16**

Preview-Fenster des Inspectors

Über das kleine blaue Schild-Symbol können Sie zudem jedem Asset ein oder mehrere Label zufügen, über die Sie diese später filtern können.

## 2.2.7 Project Browser

Im *Project Browser* befinden sich alle *Assets* des Projektes. Die linke Seite dieses Fensters zeigt dabei die Ordnerstruktur Ihrer Projektdateien an, die rechte Seite stellt den Inhalt des aktuell selektierten Ordners dar. Den Inhalt können Sie dabei über die Suchmaske im oberen Bereich durchsuchen und mithilfe des Type- und des Label-Filters aussortieren (siehe „Assets suchen und finden“).

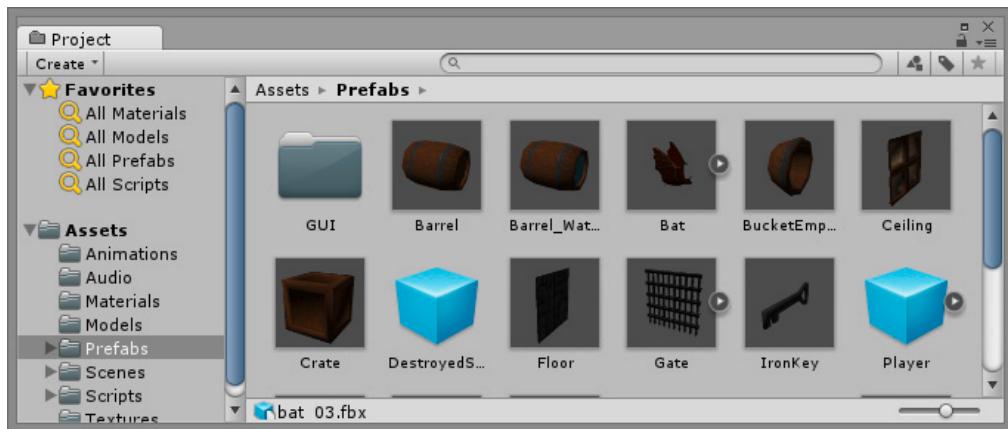


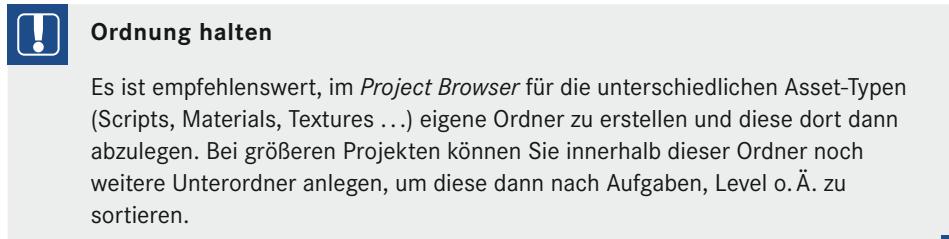
Bild 2.17 Project Browser

Mit dem Schieberegler unten rechts können Sie die Darstellung der Assets verändern. Dabei können Sie die Inhalte in einer Listendarstellung oder als Symbole anzeigen, deren Größe Sie mit dem Regler stufenlos festlegen können.

Über einen Rechtsklick in den *Project Browser* erhalten Sie eine große Auswahl an Funktionen. Der Bereich *Create* ist dabei ein Schnellzugriff auf das Hauptmenü **Assets/Create**, der auch über das Drop-down-Menü oben links im *Project Browser* angeboten wird. Beide Menüs bieten Ihnen somit die Möglichkeit, neue *Assets* wie Skripte, Materialien, *Shader* etc. zu erstellen. Aber auch Unterordner können Sie hier erstellen, die Sie für das Gliedern der Assets nutzen sollten.

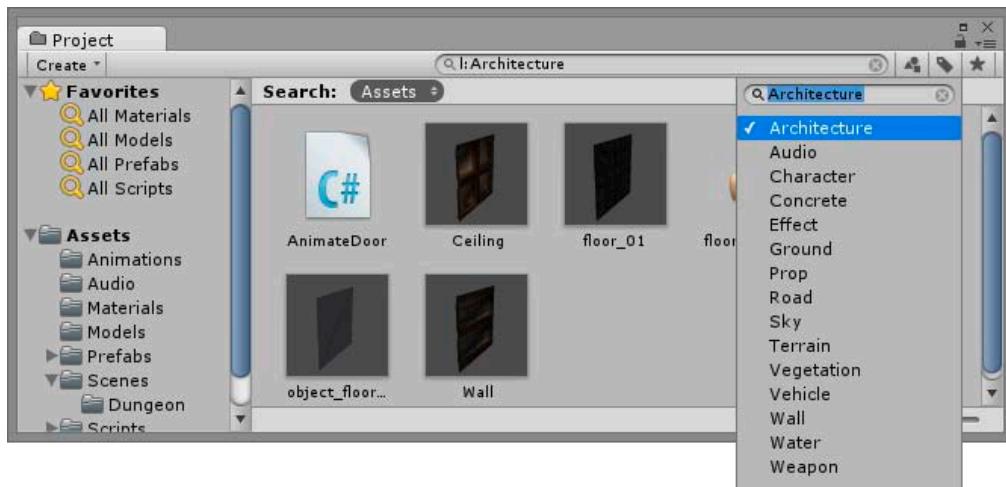
### 2.2.7.1 Assets suchen und finden

Da sich im *Project Browser* alle Dateien befinden, die Sie in Ihrem Projekt nutzen können, wird die Menge der Dateien schnell ansteigen. Deshalb ist es sehr wichtig, von vornherein eine gute Struktur aufzubauen, nach der die Assets im *Project Browser* abgelegt werden.



Möchten Sie Ihre Assets nun noch zusätzlich gruppieren, z. B. nach Zusammengehörigkeit oder einer bestimmten Eigenschaft, können Sie dies mit sogenannten Labels machen, die Sie im *Preview-Fenster* den Assets zufügen können.

Über den *Project Browser* können Sie nun über das Label-Symbol, das Sie neben dem Suchfeld finden, danach filtern. Der Vorteil hierbei ist, dass sich die Filterung nun über alle Ordner erstreckt, deren Treffer dann auch alle im Asset-Fenster angezeigt werden.



**Bild 2.18** Filterung nach dem Label „Architecture“

Den *Asset-Type* können Sie ebenfalls in die Filterung mit einfließen lassen. Nutzen Sie hierfür das links neben dem Label-Filter liegende Symbol.

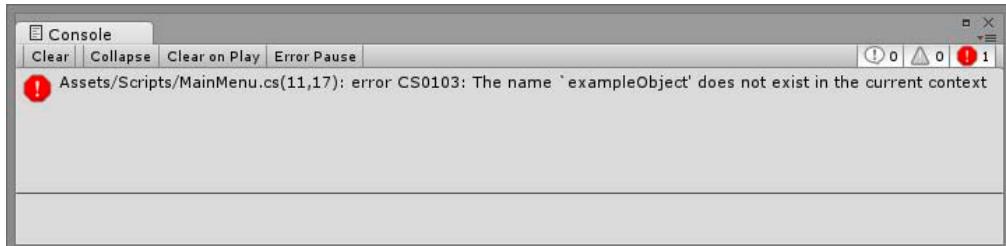
Um schließlich häufig genutzte Filter wiederzuverwenden, können Sie sie unter *Favorites* abspeichern. Nutzen Sie hierfür das Stern-Symbol, das Sie rechts neben dem Label-Symbol finden. Wollen Sie einen abgespeicherten Favoriten wieder löschen, können Sie dies über das Kontextmenü des jeweiligen Favoriten machen.

### 2.2.7.2 Assets importieren

Wenn Sie externe Assets (z. B. Texturen, Musikdateien, 3D-Modelle etc.) in Ihr Programm importieren wollen, können Sie diese einfach aus einem beliebigen Ordner Ihres Computers in den *Project Browser* hineinziehen und fallen lassen. Unity erstellt eine Kopie der Datei und importiert diese in das Projekt.

## 2.2.8 Console

Der *Console*-Tab, zu Deutsch „Konsole“, dient dem Anzeigen von Fehler- und Warnmeldungen. Sollten Sie beispielsweise Programmierfehler gemacht haben, zeigt Unity diese hier an. Aber auch Sie selber als Entwickler können hierüber Meldungen ausgeben.



**Bild 2.19** Console-Tab mit einer Fehlermeldung

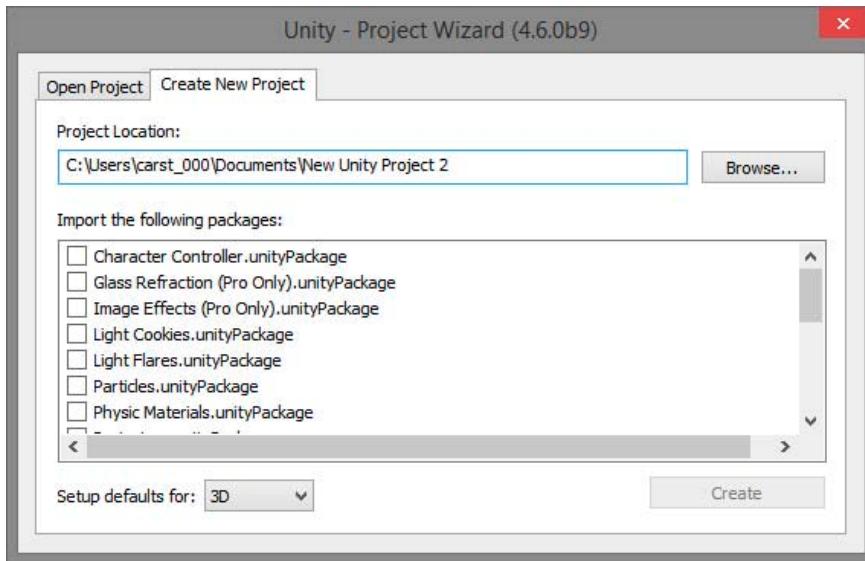
Meldungen zu Fehlern, die zwingend behoben werden müssen (Errors), werden in dem Fenster rot dargestellt. Warnungen, also nicht kritische Fehler, erscheinen demgegenüber gelb. Normale Hinweismeldungen werden weiß angezeigt. Über die drei Symbole an der rechten Seite können Sie die Meldungen auch nach diesen Einstufungen filtern.

Neben den drei Filtermöglichkeiten besitzt die obere Konsolenleiste noch einige Buttons. Diese haben folgende Bedeutungen:

- **Clear** löscht alle Einträge.
- **Collapse** unterdrückt doppelt erscheinende Meldungen.
- **Clear on Play** löscht alle Einträge beim Start des Spiels über den Play-Button.
- **Error Pause** unterbricht das Spiel, sobald ein Eintrag mit `Debug.LogError("Fehlertext")` ausgeführt wird. Mehr dazu im Kapitel „Skript-Programmierung“.

## ■ 2.3 Das Unity-Projekt

Das Gesamtgerüst eines Spiels bzw. einer Anwendung wird in Unity als Projekt bezeichnet. Es speichert alle allgemeinen Einstellungen und Daten wie beispielsweise den Namen des Spiels, die Tastenbelegungen oder auch Grafikeinstellungen. Das Spiel an sich findet aber nicht in einem Projekt statt, sondern in Szenen (Scenes), die ebenfalls in dem Projekt gespeichert werden. Jedes spielbare Projekt besitzt also mindestens eine Szene. Zum Anlegen und Öffnen von Projekten nutzt Unity den *Project Wizard*, der sich übrigens auch öffnet, wenn Sie zum ersten Mal Unity starten (siehe „Einleitung“).



**Bild 2.20** Project Wizard

### 2.3.1 Neues Projekt anlegen

Über das Menü **File/New Project** öffnen Sie den Project Wizard, um ein neues Projekt zu erstellen. Auf dem Tab *Create New Project* wird Ihnen gleich ein neuer Name vorgeschlagen, den Sie natürlich auch ändern können. Oder Sie wählen gleich einen komplett neuen Pfad über den Button **Browse** aus. Navigieren Sie zu einem Ordner, in dem Sie Ihr Projekt erstellen möchten, und bestätigen Sie die Auswahl.

Über **Create** wird nun in diesem Ordner ein neues Projekt erstellt. Dabei trägt das Projekt immer den Namen des letzten Ordners.

Wenn Sie möchten, können Sie noch vor dem Erstellen im Bereich *Import the following packages* verschiedene *Unity Packages* auswählen, die Sie von vornherein in Ihr Projekt importieren wollen. Sie können aber auch alle Packages im Nachhinein importieren, weshalb diese Auswahl nicht so essenziell ist.

Als Letztes können Sie noch im unteren Abschnitt entscheiden, ob Sie die Standardeinstellungen für ein 3D- oder ein 2D-Spiel nutzen wollen. Diese Auswahl hat keinen Einfluss auf die tatsächliche Spieleentwicklung, sondern lediglich auf einige Standardeinstellungen. Aber auch diesen Parameter können Sie später noch über **Edit/Project Settings/Editor** mit dem Parameter *Default Behaviour Mode* ändern. Die Hauptunterschiede dieser beiden Möglichkeiten sind:

- **2D** setzt die Kamera wie auch die *Scene View*-Darstellung auf *Orthographic* (siehe Abschnitt 2.2.2 und Kapitel „Kameras, die Augen des Spielers“) und den *Texture Type* beim Texturen-Import auf den Default-Wert *Sprite*.
- **3D** setzt die Kamera standardmäßig auf *Perspective* und den *Texture Type* beim Texturen-Import auf den Default-Wert *Texture*.

Haben Sie alle Einstellungen getätigt, erzeugen Sie mit dem Button **Create** Ihr neues Projekt.

Nach dem Erstellen des Projektes öffnet Unity dieses. Sollten Sie vorher Packages ausgewählt haben, werden diese natürlich im *Project Browser* angezeigt.

### 2.3.2 Bestehendes Projekt öffnen

Möchten Sie bereits erstellte Projekte öffnen, starten Sie über das Menü **File/Open Project** den *Project Wizard*. Der *Project Wizard* zeigt Ihnen im Tab *Open Project* die zuletzt geöffneten Projekte an. Über **Open Other ...** können Sie natürlich auch ältere Projekte auswählen. Navigieren Sie hier zum Hauptordner des jeweiligen Projektes und bestätigen Sie die Auswahl.

Beachten Sie, dass Unity nur eine Instanz unterstützt. Das bedeutet, dass Sie immer nur ein Projekt zurzeit bearbeiten können. Sie können Unity nicht ein zweites Mal starten, um ein anderes Projekt parallel zu öffnen.

#### 2.3.2.1 Ältere Projekte öffnen

Sollte das Projekt mit einer älteren Unity-Version erstellt worden sein, möchte Unity das Projekt „upgraden“. Dabei werden Dateien auf das Format der neuen Version konvertiert. Es ist ratsam, jedes Projekt einmal zu sichern, bevor Sie diesen Upgrading-Prozess starten. Sollten Sie dies noch nicht getan haben, brechen Sie mit **Quit** den Vorgang ab und machen Sie eine Sicherungskopie des Projektes.

Danach können Sie das Projekt noch einmal starten und dann den Upgrading-Prozess mit **Continue** durchführen lassen. Dieser Vorgang kann je nach Projektgröße auch schon mal etwas länger dauern. In manchen Fällen erscheint nach diesem Prozess noch eine Extra-abfrage zum Konvertieren einzelner Dateien. Wenn nichts Besonderes dagegen spricht, sollten Sie dies ebenfalls bestätigen.

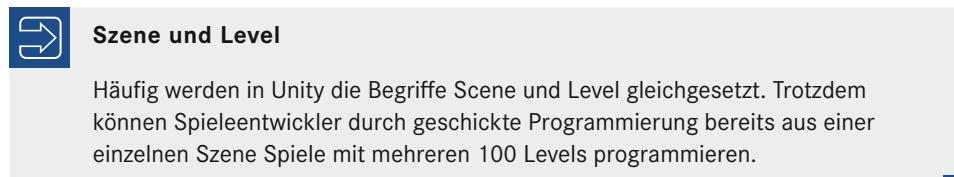
### 2.3.3 Projektdateien

Jedes Unity-Projekt besteht aus mehreren Hauptordnern, die ich im Folgenden kurz vorstellen möchte:

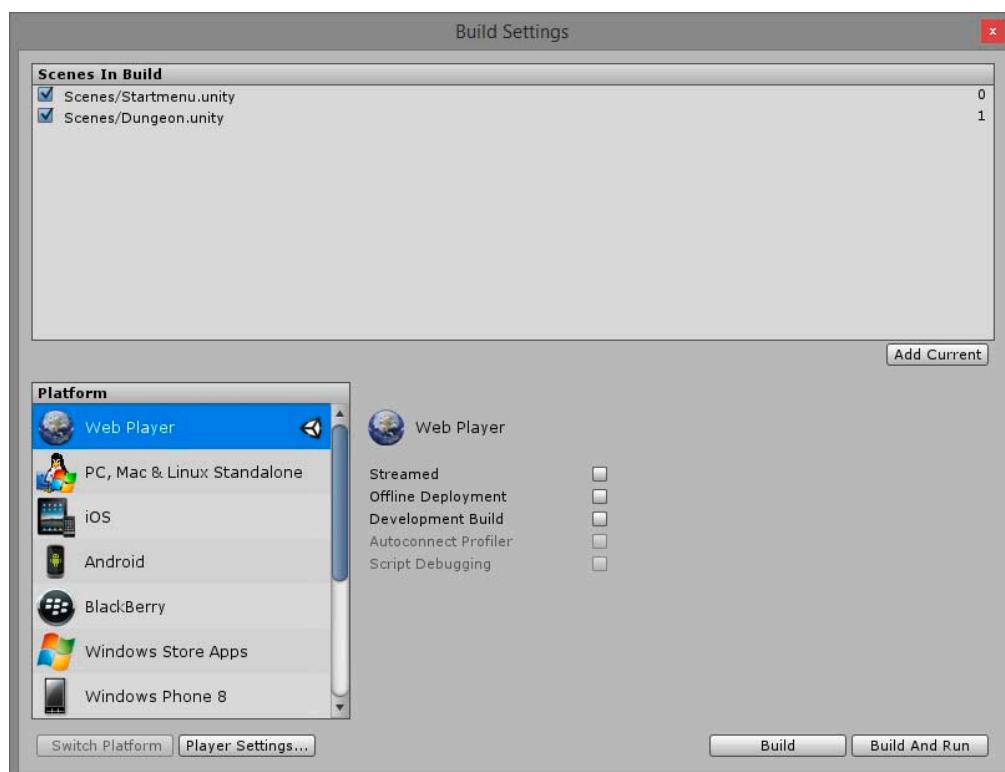
- **Assets** speichert alle Ressourcen des Projektes. Alle Dateien und Ordner, die sich in diesem befinden, werden in dem *Project Browser* von Unity angezeigt.
- **ProjectSettings** speichert alle Projekteinstellungen wie *Tags*, *Player Settings* usw.
- **Library** dient als Speicher für Metadaten von importierten **Assets**.
- **Temp** dient zum Ablegen von temporären Dateien, die von Unity zum Beispiel beim Build-Vorgang erstellt werden.

## 2.3.4 Szene

Eine Szene können Sie sich wie einen Level oder wie eine Welt eines Spiels vorstellen, in der der Aufbau des Spiels mit seinen 3D-Modellen, Grafiken, Musikdateien etc. (allgemein auch Assets genannt) festgelegt wird.



Eine neue Szene erzeugen Sie über das Menü **File/New Scene**. Eine bestehende Szene öffnen Sie über **File/Open Scene**. Jede Szene, die später im Spiel auch eingebunden werden soll, muss zuvor über **File/Build Settings** dem Spiel zugefügt werden, wo es dann auch einen eindeutigen Index erhält, über den man die Szene auch dann identifizieren und starten kann.



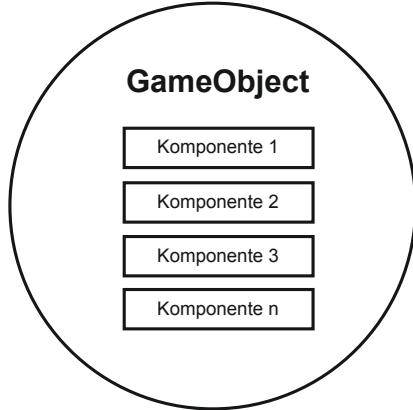
**Bild 2.21** Build Settings mit den Scene Indices

### 2.3.5 Game Objects

Der wichtigste Baustein in einer Unity-Szene ist das *GameObject*. Jedes Objekt, das sich in einer Szene befindet, ist ein solches.

Trotzdem ist ein *GameObject* an sich zunächst einmal nichts anderes als ein leerer Container, der im Grunde nichts kann. Seine eigentlichen Eigenschaften und Fähigkeiten erhält ein *GameObject* erst durch sogenannte Komponenten bzw. *Components*, die man einem *GameObject* zufügen kann.

Dies ist ein ganz elementares Prinzip von Unity, das stets bedacht werden muss. Anstatt Eigenschaften eines *GameObjects* zu verändern, werden Sie normalerweise immer Parameter einer Komponente ändern, die natürlich zu dem jeweiligen *GameObject* gehört.



**Bild 2.22**

Schematische Darstellung eines *GameObjects* mit Komponenten

Das Spannende an diesem Verfahren ist nun, dass jedes *GameObject* alles sein kann. Um z.B. aus einer Kugel eine Lampe zu machen, brauchen Sie dieser lediglich eine Beleuchtungs-Komponente zu geben und die Komponenten entsprechend zu parametrisieren. Das war es schon.

Möchten Sie jetzt ein einfaches *GameObject* Ihrer Szene zufügen, können Sie dies über das Hauptmenü **GameObject/Create Empty** machen. Diese *GameObjects* werden auch „Empty GameObjects“ genannt, obwohl sie gar nicht wirklich „empty“ sind. Denn auch diese besitzen bereits eine Komponente, und zwar die *Transform*-Komponente. Diese ermöglicht Ihnen, das *GameObject* in der Szene zu platzieren – was ja auch durchaus Sinn macht, wenn man im Editor arbeitet. Ein komplett leeres *GameObject* können Sie nur via Code erzeugen, worauf wir aber noch im Kapitel „Skript-Programmierung“ zu sprechen kommen.

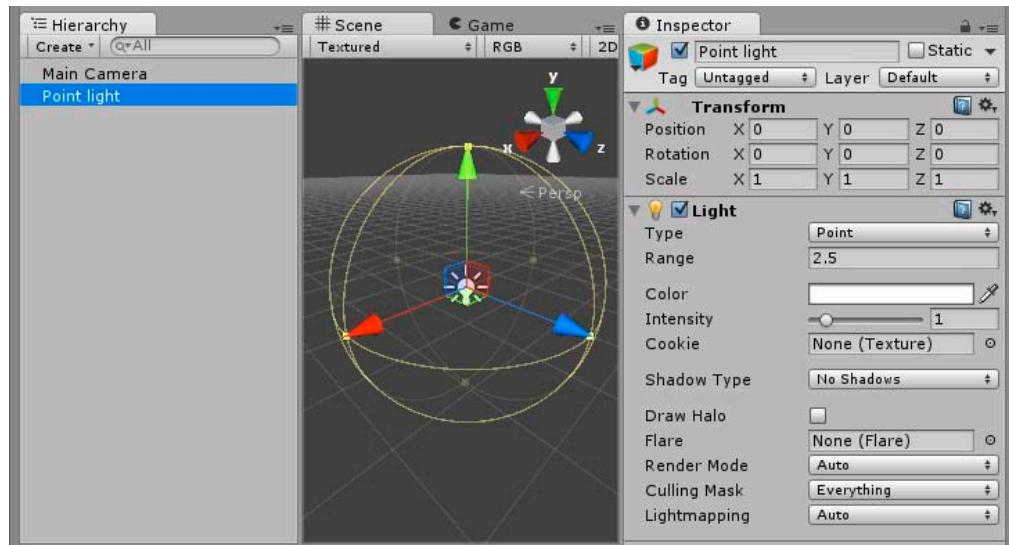
### 2.3.6 Components

*Components* (Komponenten) werden *GameObjects* zugefügt und verleihen dem *GameObject* seine eigentlichen Fähigkeiten und bestimmen deren Verhalten.

Unity liefert bereits eine ganze Reihe fertiger *Components* mit, die beispielsweise der Kollisionserkennung dienen, Sound abspielen können oder Partikeleffekte erzeugen. Auch

selbst programmierte Skripte gelten in Unity als Komponenten. Auf die genauen Zusammenhänge werden wir aber im bereits erwähnten Kapitel „Skript-Programmierung“ noch genauer eingehen.

Wenn Sie ein *GameObject* in der *Hierarchy* oder direkt in der Szene selektieren, werden Ihnen im *Inspector* alle Komponenten dieses *GameObjects* angezeigt. Ein schematisch dargestelltes *GameObject* wie in Bild 2.22 könnte dann in der Praxis wie in Bild 2.23 aussehen. Das dort selektierte *GameObject* besitzt eine *Transform*-Komponente (dargestellt durch den dreiachsigem Handle) und eine *Light*-Komponente (dargestellt durch das Glühlampen-Gizmo und die gelbe Kugel, die die Beleuchtungsreichweite zeigt).



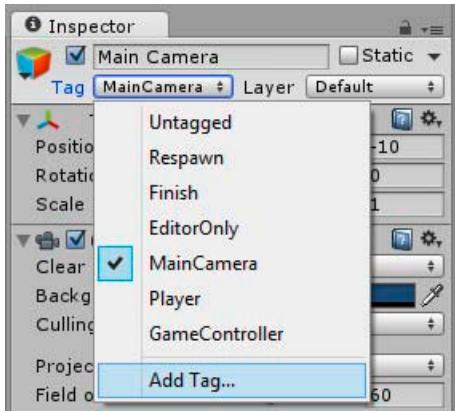
**Bild 2.23** GameObject mit Komponenten

Ein *Component* ist also ein ganz wichtiger Bestandteil eines *GameObjects*, weshalb ein *Component* aber auch nicht alleine in einer Szene existieren kann, sondern immer einem *GameObject* zugefügt sein muss.

### 2.3.7 Tags

Das englische Wort „Tag“ bedeutet nichts anderes als Etikett, und genau das ist ein *Tag* auch. Es ist ein Typenschild für *GameObjects*, um diese beim Programmieren besser identifizieren zu können.

Sie weisen den Tag im *Inspector* zu, wo Sie im oberen Bereich ein Drop-down-Menü mit allen *Tags* finden. Jedes *GameObject* kann nur mit einem einzigen Tag gekennzeichnet werden. Standardmäßig stellt Unity bereits einige wichtige Tags wie z.B. „Player“, „MainCamera“ oder „GameController“ bereit. Über den Menüpunkt **Add new tag** können Sie aber auch weitere *Tags* definieren und anschließend zuweisen.



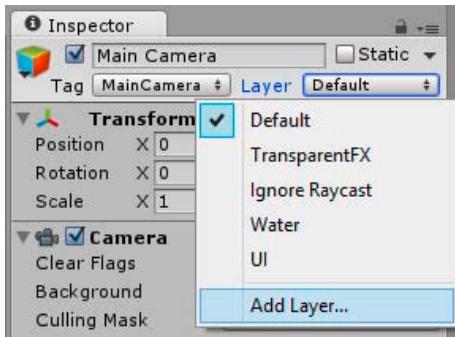
**Bild 2.24**  
Tag zuweisen

### 2.3.8 Layer

Eine weitere Kategorisierung neben *Tags* sind *Layer*. Über diese werden meist funktional zusammengehörige Objekte markiert, wie z.B. UI-Objekte oder Objekte, die von bestimmten Funktionalitäten betroffen sind.

So gibt es zum Beispiel bei Kameras die Möglichkeit festzulegen, welche *Layer* von der Kamera gerendert werden sollen. Alle anderen Objekte werden ignoriert und nicht von dieser dargestellt.

*Layer* weisen Sie ebenfalls im *Inspector* zu. Auch dort können Sie vordefinierte wählen und auch eigene erstellen.



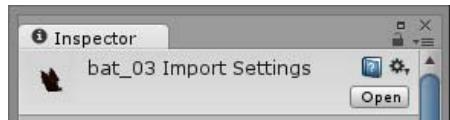
**Bild 2.25**  
Layer zuweisen

### 2.3.9 Assets

Als *Asset* werden alle digitalen Inhalte bezeichnet, die sich im Unity-Projekt und damit im Ordner „Assets“ des jeweiligen Projektes befinden. Das können beispielsweise Audiodateien, Texturen, Materialien, Skripte oder auch 3D-Modelle sein.

Den Ordner „Assets“ müssen Sie nicht lange suchen. Er wird mit allen seinen Unterordnern und enthaltenen Dateien im *Project Browser* angezeigt. Jeder Asset-Typ besitzt eigene Import-Settings, anhand derer Sie bestimmen, wie diese in Unity importiert werden sollen.

Auch wenn ich in diesem Buch einige *Import Settings* vorstellen werde, so würde das Erläutern aller verfügbaren Parameter den Rahmen sprengen. Deshalb möchte ich an dieser Stelle auf die Hilfe-Funktionen hinweisen, die Ihnen bei allen *Import Settings* in Form eines Fragezeichens zur Verfügung gestellt werden.

**Bild 2.26**

Hilfe-Funktion in den Import Settings



Auf der DVD finden Sie eine Video-Demonstration, die Ihnen noch einmal den Umgang mit Assets zeigt.

### 2.3.9.1 UnityPackage

*UnityPackage* ist ein Container-Format, welches für den Austausch von Assets zwischen unterschiedlichen Unity-Projekten gedacht ist. Dabei bleiben die Verknüpfungen und Strukturen der einzelnen Assets untereinander erhalten.

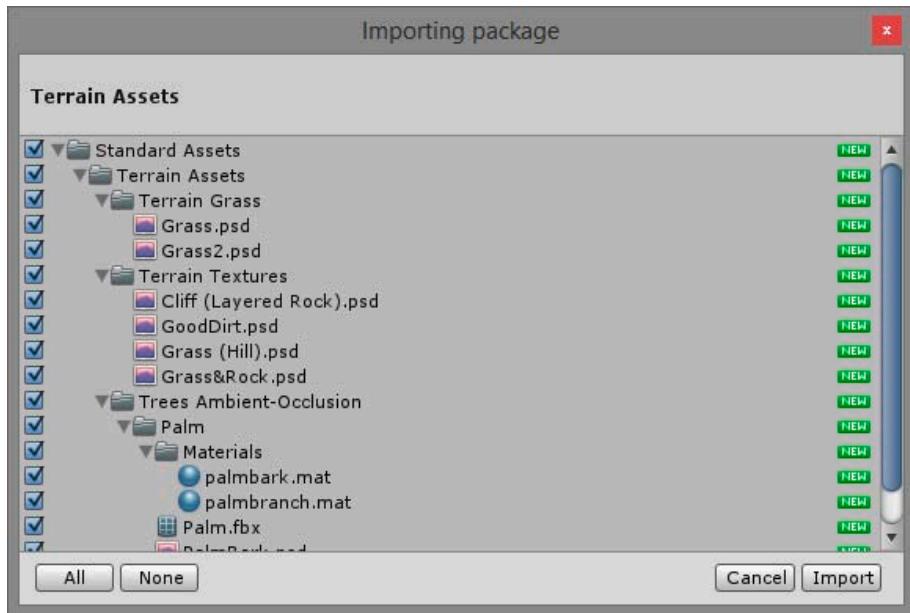
Auf diese Weise kann beispielsweise eine fertig konfigurierte Spielerfigur ohne Probleme von einem Projekt in ein anderes geladen werden, ohne dass Komponenten und Assets neu zugewiesen werden müssen. Ein *UnityPackage* besitzt die Dateiendung *.unitypackage*.

### 2.3.9.2 Asset Import

Um Assets im *UnityPackage*-Format zu importieren, gehen Sie im Hauptmenü auf **Assets / Import Package**. Alternativ können Sie die gleichen Menüpunkte aber auch im *Project Browser* über die rechte Maustaste erreichen.

Dort können Sie über **Custom Package** ein beliebiges *UnityPackage* von Ihrer Festplatte importieren. Zusätzlich stehen Ihnen dort aber auch noch von Unity mitgelieferte Asset-Pakete zur Verfügung, die sogenannten *Standard Assets*. Das sind thematisch zusammenhängende Assets, die Unity im *UnityPackage*-Format standardmäßig mitliefert und direkt im Menüzweig einbindet.

Egal ob Sie nun *Standard Assets* oder eine eigene *UnityPackage*-Datei auswählen, in beiden Fällen öffnet sich nach der Wahl des Paketes ein Auswahldialog, über den Sie noch einmal bestimmen können, welche Dateien genau aus dem Paket importiert werden sollen.

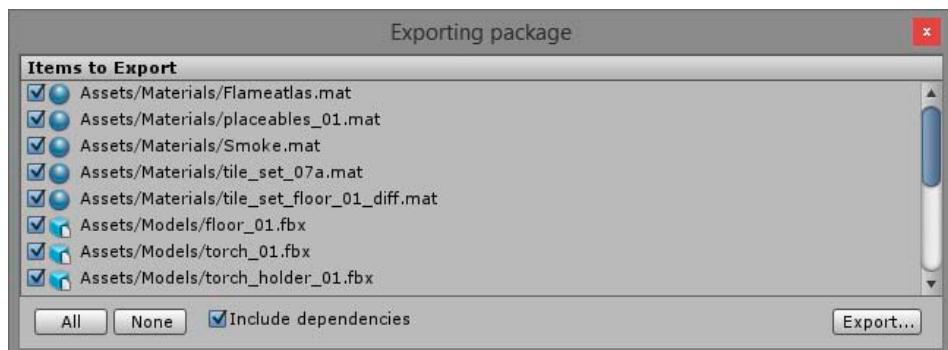


**Bild 2.27** Asset Import-Dialog

### 2.3.9.3 Asset Export

Den Export als *UnityPackage* erreichen Sie sowohl über das Kontextmenü der rechten Maustaste im *Project Browser* als auch über das Hauptmenü im Bereich **Assets**.

Selektieren Sie hierfür beliebig viele *Assets* und Ordner im *Project Browser*, die Sie exportieren möchten, und wählen dann die Funktion **Export Package**. Es öffnet sich ein Fenster, in dem noch einmal alle ausgewählten Dateien angezeigt werden, um hier ggf. noch einige wieder auszuschließen. Drücken Sie dann **Export**, und es wird ein *UnityPackage* mit den jeweiligen Dateien erstellt.



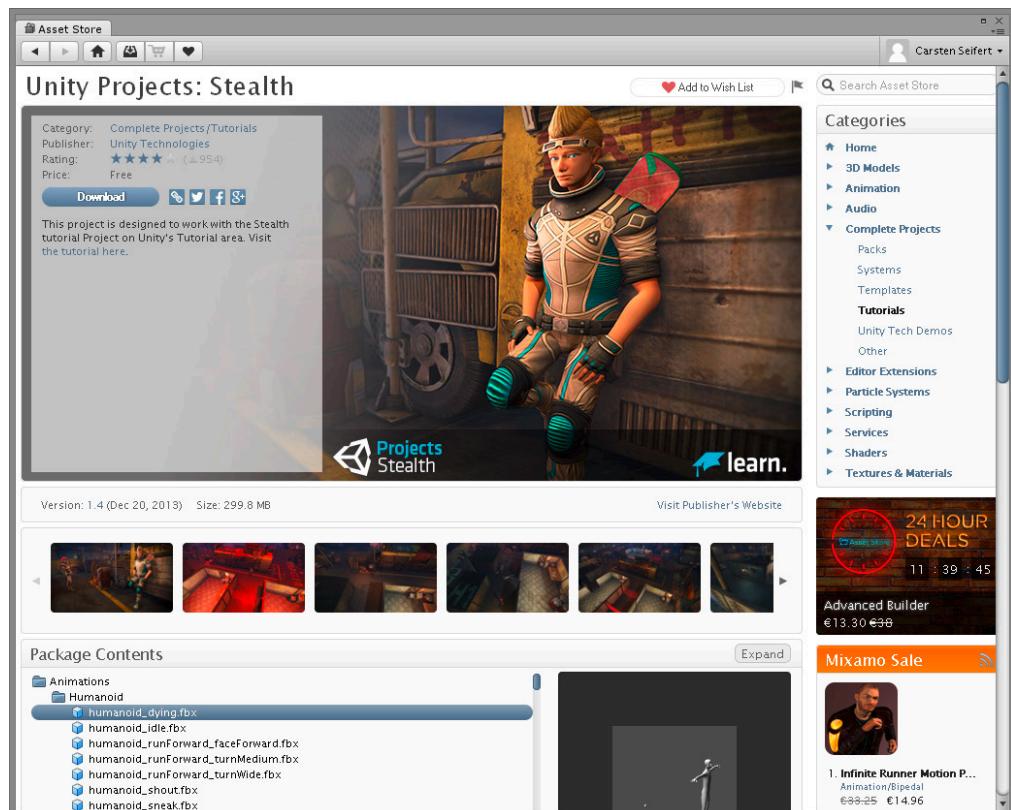
**Bild 2.28** Asset Export-Dialog

### 2.3.9.4 Asset Store

Unity besitzt einen integrierten Zubehör-Shop, den *Asset Store*. In diesem finden Sie eine große Auswahl an kostenlosen und kostenpflichtigen *Assets* aller Art. Dabei reicht das Spektrum von kleinen 3D-Modellen und einfachen Skripten über komplexe Wettersimulationen bis hin zu *Editor Extensions*, die Unity um ganz neue Funktionen erweitern. Solche *Editor Extensions* können z. B. Visual Scripting Tools sein, mit denen Sie nicht mehr Code programmieren müssen, sondern diesen über das Anordnen von Grafiken und Symbolen automatisch erzeugen können.

Aber nicht nur einzelne Assets finden Sie in dem Store, dort gibt es auch ganze Templates für Projekte und Anwendungsbeispiele für bestimmte Techniken. Auch Unity Technologies selbst stellt dort mittlerweile seine kostenlosen Beispiele und Tutorial-Projekte hinein, sodass Sie auch dort viele kostenlose Assets finden, die, wie die *Standard Assets* auch, vom Hersteller selbst stammen.

Sie öffnen den *Asset Store* direkt aus Unity heraus über das Menü **Window/Asset Store**. In einem Extra-Tab öffnet Unity dann den Store.



**Bild 2.29** Asset Store

### 2.3.10 Frames

Als *Frame* wird ein einzelnes auf dem Monitor dargestelltes Bild bezeichnet. Vergleichen kann man dies am besten mit einem Daumenkino, bei dem durch das Abblättern mehrerer Einzelbilder einer fortlaufenden Bildfolge eine Bewegung vorgetäuscht werden kann. Ein einzelnes Bild stellt hierbei ein *Frame* dar. Und auch in Unity werden keine echten Bewegungen von den Objekten gemacht, sondern es werden von der Game Engine ganz schnell Bilder auf dem Monitor gezeichnet, auch *Rendern* genannt, die rasch hintereinander dargestellt werden.

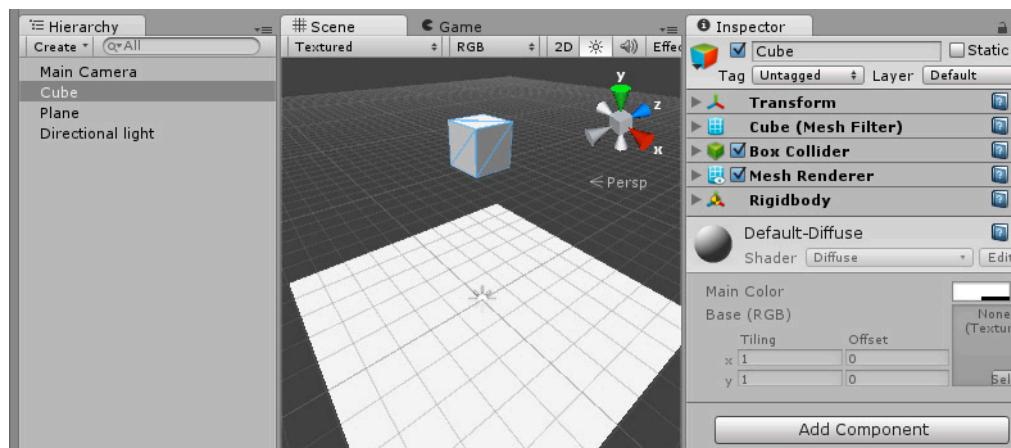
## ■ 2.4 Das erste Übungsprojekt

Um sich mit den vorgestellten Funktionalitäten vertraut zu machen, ist es am besten, diese in der Praxis anzuwenden. Im Folgenden erläutere ich deshalb ein mögliches Vorgehen, wie Sie ein kleines Übungsprojekt erstellen können, um die vorgestellten Funktionen kennenzulernen. In den folgenden Kapiteln werden wir uns ab und zu auch auf dieses Projekt beziehen, um Dinge zu testen.

1. Erstellen Sie als Erstes über **File/New Project** ein neues Projekt mit dem *Setup-Default „3D“*. Sie können den von Unity vorgeschlagenen Projektnamen einfach übernehmen oder natürlich auch selber einen wählen.
2. Nach dem Start des neuen Projektes sollten Sie in dem *Hierarchy*-Fenster nun ein Objekt namens „Main Camera“ finden. Dies ist Ihr Kamera-Objekt, aus dessen Sicht der Spieler das Spiel betrachtet. Überprüfen Sie, dass dieses Objekt auf die Koordinaten (0, 1, -10) positioniert ist und eine Rotation von (0, 0, 0) besitzt.
3. Klicken Sie im *Project Browser* den Oberordner „Assets“ an und legen Sie über die rechte Maustaste **Create/Folder** einen neuen Ordner mit dem Namen „Scenes“ an.
4. Speichern Sie die neue Szene über das Hauptmenü **File/Save Scene as**. In dem *Save Scene*-Dialog wird Ihnen nun der neue Ordner „Scenes“ angeboten. Wählen Sie diesen aus, geben Sie Ihrer Szene den Dateinamen „Test“ und drücken Sie auf **Speichern**.
5. Öffnen Sie nun im *Project Browser* den Ordner „Scenes“. Sie werden dort nun einen Eintrag finden, der die Szene „Test“ symbolisiert. Wenn Sie später weitere Szenen erzeugen, sollten Sie diese nun ebenfalls in diesem Ordner abspeichern.
6. Fügen Sie jetzt Ihrer Szene einen Würfel hinzu. Dies machen Sie über **GameObject/Create General/Cube** (je nach Version kann der Pfad auch **GameObject/Create Other/Cube** lauten).
7. Markieren Sie den Würfel „Cube“ im *Hierarchy*-Fenster und drücken Sie **F**, um in der *Scene View* auf den Würfel zu fokussieren.
8. Verschieben Sie nun den Würfel auf die Position (0, 5, 0). Versuchen Sie dies sowohl mit den *Transform-Tools* wie auch direkt über die *Inspector*-Eigenschaften des Würfels. Nutzen Sie hierbei zum Drehen und Verschieben der Ansicht der *Scene View* auch die bereits vorgestellten Navigationsmöglichkeiten dieses Fensters.

9. Erzeugen Sie nun über **GameObject/Create General/Plane** eine Fläche und positionieren Sie diese auf der Position (0, 0, 0).
10. Fügen Sie der Szene noch ein Lichtobjekt hinzu. Dies machen Sie über **GameObject/Create General/Directional Light**.
11. Starten Sie das Spiel über die Play-Taste. Ihre beiden Objekte sollten nun in der *Game View* zu sehen sein.
12. Wechseln Sie in die *Scene View* und verschieben Sie dort den „Cube“ auf Position (3, 5, 0).
13. Wechseln Sie wieder in die *Game View* und Sie werden sehen, dass der Würfel zur Seite verschoben wurde.
14. Stoppen Sie nun das Spiel. Der Würfel wird wieder auf (0, 5, 0) zurückgesetzt.
15. Als Nächstes fügen Sie dem Würfel eine Physik-Komponente zu, damit dieser von einer virtuellen Erdanziehungskraft angezogen wird. Markieren Sie hierfür den Würfel in der Hierarchy und klicken Sie im Hauptmenü auf **Component/Physics/Rigidbody**.
16. Starten Sie das Spiel und beobachten Sie, wie der Würfel nach unten fällt und auf der Fläche liegen bleibt.
17. Speichern Sie die Szene über **File/Save Scene**.
18. Beenden Sie das Spiel und fügen Sie nun die Fläche dem Würfel als Kind-Objekt hinzu.
19. Wenn Sie den Würfel verschieben, drehen oder auch skalieren, werden Sie sehen, wie sich die Fläche mitverändert.
20. Setzen Sie die Transform-Werte des Würfels wieder auf die Originalwerte zurück und starten noch einmal das Spiel. Da die Fläche ein Kind-Objekt vom Würfel ist, fällt dieser nun gemeinsam mit dem Würfel runter. Da die Objekte auf kein anderes Objekt mehr fallen können, schließlich gibt es keine anderen in der Szene, können Sie das Spiel nun wieder stoppen.

Da wir dieses Projekt im Laufe des Buches noch häufiger nutzen wollen, speichern Sie die Szene bitte nicht noch ein weiteres Mal, damit der Szenenzustand vom Schritt 16 bzw. 17



**Bild 2.30** Erstes Übungsprojekt

erhalten bleibt. Ansonsten können Sie jetzt noch weiter mit diesem Projekt die vorgestellten Funktionen testen.



Dieses erste kleine Übungsprojekt finden Sie auch auf der DVD.

# 3

## C# und Unity

Auch wenn Unity bereits für viele Standardaufgaben fertige Komponenten mitliefert (zum Beispiel für die Bereiche Audio, Physik, Partikeleffekte oder auch das Rendering), muss die eigentliche, individuelle Spiellogik immer noch von Ihnen selber programmiert werden. In Unity wird dies mit Skripten gemacht, einzelne Textdateien, die beim Erstellen des Spiels von einem Programm (genannt Compiler) in für Computer verständliche Befehle übersetzt werden. Skripte verhalten sich in Unity für gewöhnlich wie Komponenten und werden auch dementsprechend an *GameObjects* angehängt.

Da wir in diesem Buch mit der Programmiersprache C# arbeiten, möchte ich Ihnen als Erstes diese Sprache und wie sie in Unity eingesetzt wird, etwas näherbringen. Sollten Sie sich bereits mit C# auseinandergesetzt haben, möchte ich Ihnen trotzdem empfehlen, dieses Kapitel durchzulesen, da hier auch einige Besonderheiten erläutert werden, die speziell in Unity gelten.

### ■ 3.1 Die Sprache C#

C# wurde von der Firma Microsoft entwickelt und gehört zu den sogenannten *Objektorientierten Programmiersprachen*. Sie wurde entwickelt, um Anwendungen mit dem .NET Framework zu entwickeln, einer Plattform, die eine große Palette an Klassenbibliotheken, Programmierschnittstellen und Dienstprogrammen zur Verfügung stellt. Unity nutzt die Sprache aber in Kombination mit dem Mono-Framework, einer Open-Source-Variante des .NET Frameworks, wodurch C#-Anwendungen auch auf Nicht-Microsoft-Systemen betrieben werden können (bekannt auch als Mono-Projekt).

Da die Sprache sehr umfangreich ist, werde ich Ihnen im Folgenden aber nicht alle Möglichkeiten von C# vorstellen und den Fokus vor allem auf die Themen legen, die für Sie als Unity-Programmierer interessant sind. Sollten Sie weitere Informationen zur Sprache C# haben wollen, sollten Sie über die bekannten Suchmaschinen fündig werden. Außerdem gibt es viele gute Bücher, wie z.B. die Visual C#-Bücher von Walter Doberenz und Thomas Gewinnus, die die Sprache in allen ihren Facetten behandeln.

Sollten Sie einige Themen nicht gleich verstehen, verzweifeln Sie nicht. Sie werden sicher den einen oder anderen Aha-Effekt haben, wenn Sie auch die anderen Buchkapitel durchgearbeitet haben und diese Stellen später erneut durchlesen.

Als Einstieg möchte ich Ihnen noch ein kleines Skript zeigen, das Sie am Ende dieses Buch für das Beispiel-Game programmieren werden. Das Skript dient dem Verwalten der Lebensstärke und wird sowohl vom Spieler wie auch von den Gegnern genutzt.

#### **Listing 3.1** HealthController-Skript

```
using UnityEngine;
using System.Collections;
public class HealthController : MonoBehaviour {
    public float health = 5;
    private bool isDead = false;
    void ApplyDamage(float damage) {
        health -= damage;
        if(health <= 0 && !isDead) {
            isDead = true;
            Dying();
        }
        else {
            Damaging();
        }
    }
    public virtual void Damaging()
    {
    }
    public virtual void Dying ()
    {
    }
}
```

## ■ 3.2 Syntax

Zunächst einmal einige Worte zum grundsätzlichen Programmieren: Programmcode (auch Quelltext oder Source-Code genannt) besteht aus Code-Zeilen, die nacheinander abgearbeitet werden. Damit der Computer diese Zeilen auch versteht, werden diese mit einem Programm namens Compiler in Maschinensprache übersetzt, das sogenannte Kompilieren. Damit der Compiler auch weiß, wann das Ende einer einzelnen Codezeile erreicht ist, wird in C# am Ende ein Semikolon ; geschrieben. Weitere wichtige Zeichen sind in C# die geschwungenen Klammern {}. Diese werden dafür genutzt, um zusammenhängende Codeblöcke zu kennzeichnen.

## ■ 3.3 Kommentare

Neben dem eigentlichen Code wird es auch vorkommen, dass Sie Anmerkungen zu Ihrem Code hinzufügen möchten, die eben nicht vom Computer ausgeführt werden sollen. Solche nicht auszuführenden Zeilen werden auch Kommentare genannt. Einen Kommentar markieren Sie mit einem Doppelslash //. Alles, was dahinter bis zum nächsten Zeilenumbruch steht, gilt als nicht auszuführender Code. Was in dieser Zeile vor dem Doppelslash steht, gilt aber als ausführbar!

### **Listing 3.2 Einzeiliger Kommentar**

```
int zahl1; //Dies ist eine Befehlszeile, die auch kompiliert wird.  
zahl1 = 2;
```

Wenn Sie mehrere Zeilen als Kommentare markieren möchten, können Sie dies durch das Schreiben von /\* am Anfang und \*/ am Ende erreichen:

### **Listing 3.3 Mehrzeiliger Kommentar**

```
/*  
Dies ist alter Code  
int lifePoints;  
lifePoints= 2;  
*/  
int lifePoints = 2; //Das ist der neue Code
```

Normalerweise werden Kommentare von der Entwicklungsumgebung farblich hervorgehoben, z.B. in Grün oder Rot.

## ■ 3.4 Variablen

Variablen sind zunächst einmal nichts anderes als Platzhalter, um Werte zu speichern. Der Name einer Variablen muss immer eindeutig sein und wird von Ihnen vergeben. Auch wenn dieser zunächst einmal keine Rolle spielt, müssen Sie darauf achten, dass er keine Sonderzeichen, Umlaute oder Leerzeichen besitzt. Außerdem dürfen Variablennamen nicht mit einer Zahl beginnen.

### **3.4.1 Namenskonventionen**

Um die Lesbarkeit von Programmcode zu vereinfachen, gibt es einige Konventionen, an die Sie sich halten sollten. Zum einen sollten die Variablennamen den Inhalt der Variablen beschreiben. Hierbei nutzen Sie am besten englische Begriffe, z.B. speed. Außerdem sollten Variablenbezeichnungen immer mit einem Kleinbuchstaben beginnen. Bei Namen, die aus mehreren Wörtern bestehen, sollten diese durch Großbuchstaben voneinander abgehoben werden, z.B. enemySpeed. Diese Schreibweise wird übrigens auch *Camel Case* genannt.

## 3.4.2 Datentypen

Damit der Compiler auch weiß, was für Werte in einer Variablen gespeichert werden dürfen (Zahlen, Buchstaben ...), müssen Sie der Variablen noch sagen, von was für einem Datentyp sie ist. Dies wird als Variablen Deklaration bezeichnet. Häufig genutzte Datentypen sind in der Unity-Spieleentwicklung:

- **string** für Text
- **int** für Ganzzahl
- **float** für Fließkommazahl
- **bool** für Boolean
- **enum** für Enumeration

In C# beginnt die Deklaration einer Variablen mit dem Datentyp gefolgt von dem Namen der Variablen. Am Ende folgt natürlich das Semikolon:

### **Listing 3.4** Variablen Deklaration

```
int lifePoints;
float height;
string name;
```

Um diesen Variablen Werte zuzuweisen, wird in C # das Gleichzeichen genutzt:

### **Listing 3.5** Werte zuweisen

```
lifePoints = 2;
name = "Carsten";
height = 1.5F;
```

Wie Sie sehen, benutze ich bei Texten zusätzlich zu dem Text an sich noch Anführungsstriche, um dem Compiler zu sagen, wo der zugewiesene Text beginnt und wo er endet. Wenn ich Kommazahlen zuweisen (hierfür benötige ich den Datentyp *float*), schreibe ich statt des Kommas einen Punkt. Zudem schreibe ich noch bei *float*-Variablen ein F hinter dem Wert, damit der Compiler weiß, dass es sich bei dem Wert auch tatsächlich um eine Fließkommazahl vom Typ *float* handelt. Alternativ gibt es nämlich in C# noch den Typ *double*, welcher aber einen größeren Zahlenbereich abdeckt als *float* und deshalb nicht in eine *float*-Variable „hineinpasst“:

### **Listing 3.6** Unterschied float und double

```
float gravity;
gravity = 9.81F;
double gravity;
double= 9.81;
```

In Unity ist aber *float* geläufiger als *double*, weshalb ich in Zukunft nur von *float*-Werten sprechen werde. Um den obigen Code etwas zu kürzen, können Sie gleich beim Deklarieren der Variablen diese auch mit einem Wert vorbelegen (auch Initialisierung genannt):

### **Listing 3.7** Variablen Deklaration mit Initialisierung

```
float gravity = 9.81F;
```

Ein weiterer Typ, der sehr viel in der Spieleprogrammierung Verwendung findet, ist der Datentyp Boolean (*bool*). Dieser Typ kann nur zwei Zustände annehmen: *TRUE* und *FALSE*, also wahr oder falsch. Ein typisches Beispiel hierfür ist eine Checkbox, also ein Haken in der GUI, um beispielsweise eine Funktion zu aktivieren oder zu deaktivieren. Je nach Zustand dieser Checkbox hat nun die boolesche Variable, die den Wert im Programmcode speichert, den Wert *TRUE* oder *FALSE*.

**Listing 3.8** Boolesche Variable

```
bool isAttacking = false;
```

### 3.4.3 Schlüsselwort var

Neben dem Festlegen eines Datentyps mit *int*, *string* usw. gibt es in C# auch die Möglichkeit, eine Variable mit *var* zu definieren. In diesem Fall wird der Datentyp dieser Variablen erst mit dem Zuweisen des ersten Wertes bestimmt. Dann entscheidet der Compiler selber, welcher Datentyp hier der richtige ist, und legt diesen fest. Jede zukünftige Zuweisung, die einen anderen Typ erfordert würde, führt dann aber zu einem Fehler.

**Listing 3.9** Variablendeclaration mit var

```
var lifePoint;  
lifePoint = 5;
```

### 3.4.4 Datenfelder/Array

Wenn Sie mehrere Variablen eines Datentyps benötigen, können Sie diese über eine Array-Definition erstellen. Ein Array ist also kein Datentyp an sich, sondern eher eine Variation eines Datentyps.

#### 3.4.4.1 Arrays erstellen

Ein Array definieren Sie durch eckige Klammern hinter dem eigentlichen Datentyp:

**Listing 3.10** Array-Deklaration

```
int[] speedLimits;
```

Im Gegensatz zu einer normalen Variablen existiert das Array aber noch nicht durch die alleinige Definition der Variablen. Dies müssen Sie noch einmal extra machen, und zwar mit dem Schlüsselwort *new*. Diesen Vorgang nennt man auch Instanziieren. Zudem müssen Sie beim Instanziieren dem Array noch mitgeben, wie groß es sein soll, also aus wie vielen Integer-Variablen das Array bestehen soll. Das sieht dann wie folgt aus:

**Listing 3.11** Array-Instanziierung

```
speedLimits= new int[5];
```

Oder etwas kürzer:

**Listing 3.12** Array-Deklaration mit Instanziierung

```
int[] speedLimits = new int[5];
```

Sie können beim Instanziieren auch gleich jedem Element (Item) des Arrays auch einen Startwert mitgeben. Dies kann dann so aussehen:

**Listing 3.13** Array-Initialisierung mit verschiedenen Werten

```
int[] speedLimits= new int[5]{30,50,80,100,120};
```

Wichtig ist hierbei, dass Sie auch tatsächlich so viele Werte in den geschweiften Klammern übergeben, wie das Array Elemente besitzt.

#### 3.4.4.2 Zugriff auf ein Array-Element

Wenn Sie nun auf ein bestimmtes Element dieses Arrays zugreifen möchten, müssen Sie über eine Index-Zahl das richtige Item auswählen. Für Unerfahrene wird dies erst einmal eine Umstellung bedeuten, denn das erste Element eines Arrays hat den Index 0, nicht 1! Damit hat ein Array mit fünf Elementen den höchsten Index 4!

**Listing 3.14** Array-Elemente mit Index ansprechen

```
int[] speedLimits= new int[5]{30,50,80,100,120};
int[0] = 20; // Wert betrug vorher 30
int[4] = 140; // Wert betrug vorher 120
```

#### 3.4.4.3 Anzahl aller Array-Items ermitteln

Es kann sein, dass Arrays auch automatisch erstellt werden. Wenn Sie nun wissen möchten, aus wie vielen Items denn nun das Array besteht, können Sie dies über die Eigenschaft Length ermitteln:

**Listing 3.15** Array-Größe ermitteln

```
int len = speedLimits.Length;
```

#### 3.4.4.4 Mehrdimensionale Arrays

Sie haben auch die Möglichkeit, ein Array mit mehreren Dimensionen zu erstellen.

Stellen Sie sich vor, Sie möchten mit einem Array ein quadratisches Rasterspielfeld darstellen (ein Schachbrett zum Beispiel). Dann ist es sehr unübersichtlich, ein Array mit 64 Elementen zu erstellen. Einfacher ist es, ein Array mit zwei Dimensionen zu erstellen, wo der erste Index die X-Achse (beim Schach wären es die Buchstaben A bis H) darstellt und der zweite die Y-Achse. Eine solche Definition sieht in C# dann so aus:

**Listing 3.16** Mehrdimensionales Array

```
int[,] tile= new int[8,8];
```

Bedenken Sie auch hier, dass jeder Index des Arrays mit einer 0 beginnt. Wenn Sie nun die Spielfiguren mit Zahlen verschlüsseln (0 bedeutet keine Figur, 1 stellt einen Bauer dar, 2 ist ein Turm), können Sie so kinderleicht die Positionen der Figuren in diesem Array speichern:

**Listing 3.17** Mehrdimensionales Array als Schachbrett-Speicher

```
tile[0,0] = 2; //Auf A1 befindet sich ein Turm.  
tile[0,1] = 1; //Auf A2 befindet sich ein Bauer.  
tile[1,1] = 1; //Auf B2 befindet sich ein Bauer.  
tile[0,2] = 0; //Auf A3 befindet sich keine Figur.
```

## ■ 3.5 Konstanten

Es gibt immer wieder Werte, die nicht verändert werden sollen bzw. dürfen. Diese können Sie als Konstanten definieren, wodurch diese nicht mehr verändert werden können. Hierfür müssen Sie vor dem Datentyp das Schlüsselwort `const` schreiben. Zudem ist es notwendig, Konstanten bereits bei der Deklaration zu initialisieren:

**Listing 3.18** Konstanten-Deklaration

```
const float gravity = 9.81F;
```

### 3.5.1 Enumeration

Mit *Enumerationen* können Sie lesbare Auflistungen von Konstanten erstellen. Was etwas kryptisch klingt, ist aber eine ganz praktische Sache. Stellen Sie sich vor, Sie wollen den Zustand einer Figur speichern. Dieser hat drei Zustände: Stehen, Gehen, Springen. Um abzufragen, in welchem Zustand sich die Figur gerade befindet, möchten Sie diesen nun in einer Variablen speichern. Hierfür definieren Sie zunächst einmal eine Enumeration (kurz *Enum*), die diese drei Werte besitzt. Wir wollen diese Enumeration einfach mal *State* nennen:

**Listing 3.19** Enum-Deklaration

```
enum State {Idle,Walk,Jump}
```

Danach können Sie nun eine Variable vom Typ dieser Enumeration erstellen und dieser einen der drei Werte zuweisen.

**Listing 3.20** Enumeration nutzen

```
myState State;  
myState = State.Idle;
```

## ■ 3.6 Typkonvertierung

Es ist auch möglich, eine Variable eines Typs in einen anderen Typ umzuwandeln. Voraussetzung ist natürlich, dass es auch vom Inhalt her passt. Eine *Integer*-Variable kann immer in eine *Float*-Variable umgewandelt werden. Der umgekehrte Weg funktioniert nur dann, wenn die *Float*-Zahl keinen Kommawert besitzt. Bei der Typkonvertierung wird vor dem zu konvertierenden Wert einfach in runden Klammern der Zieltyp angegeben:

**Listing 3.21** Typkonvertierung

```
int lifePoints;
string lifePointsText = "2";
lifePoints = (int) lifePointsText;
```

Zahlen in ein *String* umzuwandeln, geht sogar noch einfacher. Diese besitzen von Haus aus eine Methode/Funktion (was dies ist, erkläre ich gleich noch), die diese Aufgabe übernimmt. Hierfür geben Sie hinter der Zahlenvariablen einen Punkt an und danach den Text *ToString()*. Bereits beim Tippen des Punktes sollte eine Auswahl in MonoDevelop erscheinen, die diesen Befehl zur Verfügung stellt.

**Listing 3.22** *ToString*-Beispiel

```
lifePointsText = lifePoints.ToString();
```

Diese Funktion ist sehr nützlich, da auf diese Weise die *Integer*-Variable wie ein *String* behandelt werden kann:

**Listing 3.23** String-Verkettung

```
lifePointsText = "Sie besitzen " + lifePoints.ToString() + " Lebenspunkte.;"
```

Wie Sie sehen, können Sie in C# mithilfe des *+Zeichens* Strings aneinanderhängen. Dieses Verfahren wird auch als String-Verkettung bezeichnet und wird sehr häufig bei Textausgaben genutzt.

## ■ 3.7 Rechnen

Wenn Sie mit Zahlen rechnen möchten, dann sollten Sie die folgenden Rechenbefehle kennen, die in C# genutzt werden:

**Tabelle 3.1** Rechenoperatoren

Operator	Funktion
+	Addieren
-	Subtrahieren

Operator	Funktion
*	Multiplizieren
/	Dividieren
%	Modulo (Division mit Rest)
++	Wert um 1 erhöhen
-	Wert um 1 vermindern

Das Ergebnis der Rechnung weisen Sie mit einem einfachen Gleichzeichen einer Variablen zu.

#### **Listing 3.24** Beispielrechnung

```
int health = 5;
int damage = 1;
health = health - damage;
```

Sie können Rechnungen auch verkürzen. Nehmen wir an, Sie wollen einen Wert um eins erhöhen, dann können Sie auch die folgenden Schreibweisen nutzen:

#### **Listing 3.25** Verkürzte Schreibweise einer Addition

```
lifePoints += 1;
```

Beim Sonderfall des Hochzählens bzw. Runterzählens um 1 geht es sogar noch kürzer:

#### **Listing 3.26** Werte um eins erhöhen und reduzieren

```
lifePoints++;
lifepoint --;
```

## ■ 3.8 Verzweigungen

Nachdem Sie nun alle wichtigen Vorkenntnisse für das Programmieren kennengelernt haben, können wir endlich mit dem eigentlichen Programmieren beginnen. Denn hier geht es ja nicht darum, einfach nur Variablen zu definieren und mit Werten zu befüllen, vielmehr geht es darum, abhängig von diesen Werten Entscheidungen zu treffen.

Hier kommen sogenannte Verzweigungen ins Spiel. Sie entscheiden aufgrund von definierten Bedingungen, welche Codeblöcke als Nächstes ausgeführt werden sollen.

### 3.8.1 if-Anweisungen

Die einfachste Verzweigung ist die sogenannte *if-Anweisung*. Sie arbeitet wie eine Weiche, die abhängig von einer Bedingung einen Code ausführen lässt. Zudem kann ein Alternativcode angegeben werden, der ausgeführt werden soll, wenn die Bedingung nicht erfüllt ist,

sodass sich der gesamte Code dann wie folgt verhält: „Wenn die Bedingung zutrifft, dann mache dies, ansonsten mache das.“

Eine if-Anweisung wird mit dem Signalwort `if` eingeleitet, gefolgt von der Bedingung, die in runden Klammern eingeschlossen wird. In geschwungenen Klammern folgt dann der eigentliche Code. Der optionale Alternativcode wird dann mit dem Wort `else` eingeleitet.

#### **Listing 3.27** Einfache if-Anweisung

```
if(lifePoints == 2) {
    message = "Du hast 2 Leben";
}
else {
    message = "Keine Ahnung, wie viele Leben Du hast." +
    "Aber es sind auf jeden Fall keine 2.";
}
```

Wenn der Codeblock bzw. der `else`-Zweig nur aus einer einzigen Zeile besteht, kann auch auf die geschwungenen Klammern verzichtet werden.

#### **Listing 3.28** if-Anweisung ohne Klammern

```
if(lifePoints == 2)
    message = "Du hast 2 Leben";
```

Die Bedingungen der if-Anweisung werden dabei mit Vergleichsoperatoren angegeben, die Sie der Tabelle 3.2 entnehmen können.

**Tabelle 3.2** Vergleichsoperatoren

Operator	Erläuterung
<code>a == b</code>	Vergleicht, ob <code>a</code> gleich <code>b</code> ist
<code>a != b</code>	Vergleicht, ob <code>a</code> ungleich <code>b</code> ist
<code>a &gt; b</code>	Vergleicht, ob <code>a</code> größer <code>b</code> ist
<code>a &lt; b</code>	Vergleicht, ob <code>a</code> kleiner <code>b</code> ist
<code>a &lt;= b</code>	Vergleicht, ob <code>a</code> kleiner oder gleich <code>b</code> ist
<code>a &gt;= b</code>	Vergleicht, ob <code>a</code> größer oder gleich <code>b</code> ist

if-Anweisungen können aber nicht nur auf einer Ebene existieren. Sie können auch ineinander verschachtelt sein, sodass weitere if-Bedingungen abgefragt werden können, wenn etwas zutrifft oder eben nicht zutrifft.

#### **Listing 3.29** Verschachtelte if-Anweisung

```
if(lifePoints == 2) {
    message = "Du hast 2 Leben.";
}
else {
    if (lifePoints != 3) {
        message = "Du hast keine 2 und keine 3 Leben.";
    }
    else {
```

```

        message ="Du hast 3 Leben.";
    }
}

```

Bei booleschen Werten können Sie auch anstatt `if (isAlive == true){}` eine Kurzschreibweise `if (isAlive) {}` nutzen. Möchten Sie nun auf FALSE anstatt auf TRUE prüfen, können Sie das Ergebnis einfach mit einem Ausrufezeichen invertieren.

#### **Listing 3.30** if-Anweisung mit negativem Bool-Wert

```

if (!isAlive) {
    Destroy(gameObject);
}

```

### 3.8.1.1 Komplexere if-Anweisungen

Ist die Code-Ausführung von mehreren Bedingungen abhängig, können Sie die Bedingungen mit Operatoren miteinander logisch verknüpfen. Die am meisten genutzten Verknüpfungen sind die sogenannten bedingten Operatoren.

**Tabelle 3.3** Bedingte Operatoren

Operator	Funktion	Erläuterung
<code>&amp;&amp;</code>	UND	Wenn beide Operanden <i>TRUE</i> sind, ist auch das Ergebnis <i>TRUE</i>
<code>  </code>	ODER	Wenn mindestens ein Operand <i>TRUE</i> ist, ist auch das Ergebnis <i>TRUE</i>

In der folgenden *if-Anweisung* müssen beide Bedingungen *TRUE* sein, damit wir in den Ausführungsblock gelangen:

#### **Listing 3.31** if-Anweisung mit zwei Bedingungen

```

if (health > 0 && lifePoints > 0) {
    //Code...
}

```

Beachten Sie, dass Sie auch hier ein Ausrufezeichen zum Negieren nutzen können. So wird der Code dann ausgeführt, wenn entweder `lifePoints` größer als 0 ist oder die boolesche Variable `isDetected` den Wert *FALSE* besitzt.

#### **Listing 3.32** if-Anweisung mit zwei ODER-verknüpften Bedingungen

```

if (lifePoints > 0 || !isDetected) {
    //Code...
}

```

Wie in der normalen Algebra können Sie auch hier mithilfe von Klammern Operationen beliebig komplex gestalten. Das folgende Beispiel verlangt, dass entweder beide linken Werte (`health` und `lifePoints`) größer 0 sind oder dass lediglich die Variable `delay` größer 0 ist.

**Listing 3.33** if-Anweisungen mit kombinierten Bedingungen

```
if ((health > 0 && lifePoints > 0) || (delay > 0)){
    //Code...
}
```

**3.8.2 switch-Anweisung**

Bei einer if-Anweisung haben Sie immer nur maximal zwei mögliche Code-Abschnitte, die ausgeführt werden können. Zwar können Sie durch Verschachtelungen der *if-Anweisung* auch mehrere Zustände darstellen, allerdings wird dies irgendwann sehr unübersichtlich.

Abhilfe schafft hier die *switch-Anweisung*. Mit dieser können Sie beliebig viele Zustände überprüfen und dann den Code ausführen, der für den jeweiligen Fall (*case*) gilt.

**Listing 3.34** Beispiel switch-Anweisung

```
int damage = 1;
string title;
switch (damage)
{
    case 0:
        message="Daneben!";
    case 1:
        message="Ach, nur ein Kratzer!";
        break;
    case 2:
        message="Verdamm, das tut weh!";
        break;
    default:
        message="Arrrgh!";
        break;
}
```

Wichtig bei der *switch-Anweisung* ist der Abschluss eines jeden *case*-Abschnitts mit dem Befehl *break*. Dieser sorgt dafür, dass nach dem Ausführen des Codes die *switch-Anweisung* verlassen wird.

Sollte keiner der definierten *case*-Bedingungen zutreffen, können Sie schließlich mit dem Signalwort *default* einen Standardfall definieren, der gewählt wird, sobald keiner der vorherigen Fälle passte. In dem Listing 3.34 würde dies immer dann zutreffen, wenn der Schaden höher als zwei wäre (einen negativen damage-Wert ignorieren wir einfach mal).

## ■ 3.9 Schleifen

Neben den Verzweigungen sind ein weiteres wichtiges Element in der Programmierung die Schleifen. Diese sorgen dafür, dass bestimmte Code-Abschnitte so lange wiederholt werden, bis eine vorgegebene Abbruchbedingung erfüllt ist.

### 3.9.1 for-Schleife

Der Klassiker unter den Schleifen ist die *for-Schleife*. Sie wird eingesetzt, wenn bekannt ist, wie häufig ein Codebereich wiederholt werden soll. Bei diesem Schleifentyp wird eine Zählervariable (Schleifenzähler) mit einem Startwert belegt und so lange hochgezählt, wie eine angegebene Laufbedingung erfüllt ist. In C# ist der erste Ausdruck der Schleifenzähler mit dem Startwert, der zweite stellt die Laufbedingung dar, der dritte Ausdruck beschreibt die Zählererhöhung.

**Listing 3.35** Beispiel for-Schleife

```
int counter = 0;
for(counter = 0; counter < 10; counter++)
{
    //Code...
}
```

Bei dem obigen Beispiel wurde der Zähler `counter` auf den Startwert 0 gesetzt und durchläuft den folgenden Code zehnmal. Dies liegt an der Zählererhöhung `++`, die den Wert von `counter` nach jedem Durchlauf um eins erhöht.

#### 3.9.1.1 Negative Schrittweite

Das nächste Beispiel zeigt eine Schrittweite von `-2` und durchläuft den Code, solange der Zähler größer 0 ist. Außerdem wird hier der Schleifenzähler erst in der *for-Schleife* definiert, was auch recht häufig in der Praxis gemacht wird:

**Listing 3.36** Negative Schrittweite

```
for(int counter = 10; counter > 0; counter-=2)
{
    //Code...
}
```

#### 3.9.1.2 break

Eine *for-Schleife* muss nicht zwangsläufig immer bis zum Ende durchlaufen werden. Sie kann auch mit dem Schlüsselwort `break` abgebrochen werden:

**Listing 3.37** for-Schleife mit break

```
int[] speedLimits= new int[5] {30,50,80,100,120};
int currentSpeed = 100;
```

```

int currentGear;
for(int counter = 0; counter < 5; Counter++)
{
    //Code...
    if(speedLimits[counter] >= currentSpeed)
    {
        currentGear = counter +1;
        break;
    }
}

```

## 3.9.2 Foreach-Schleife

Die *foreach-Schleife* dient dem Durchlaufen von Arrays. Hierbei wird eine zusätzliche Laufvariable genutzt, die den Wert des aktuellen Elements des Arrays übernimmt. Da es sich hierbei also um eine Kopie des Originals handelt, kann der Inhalt des Arrays auf diese Weise nicht verändert werden. Wichtig ist hierbei noch zu wissen, dass Sie die Laufvariable erst in der Schleife definieren.

**Listing 3.38** Beispiel foreach

```

foreach(int currentLimit in speedLimits) {
    if(currentLimit == 80) {
        //Code...
    }
}

```

## 3.9.3 while-Schleife

Die *while-Schleife* läuft so lange, bis eine *Abbruchbedingung* erfüllt ist. Die Abbruchbedingung wird am Anfang der Schleife abgefragt, sodass die Schleife unter Umständen auch gar nicht ausgeführt werden könnte, nämlich dann, wenn die Abbruchbedingung von Anfang an erfüllt ist.

**Listing 3.39** Beispiel while-Schleife

```

int enemyIndex = 0;
while(enemyIndex < 5)
{
    if(enemyIndex== 3)
        break;
    //Code...
    enemyIndex++;
}

```

Auch diese Schleife kann durch den Befehl `break` einfach beendet werden, ohne die eigentliche Abbruchbedingung zu erfüllen. Zusätzlich gibt es aber noch den Befehl `continue`. Dieser sorgt dafür, dass sich die Schleife beim Erreichen des `continue`-Befehls so verhält, als wäre sie bereits am Ende des Schleifencodes angekommen, und beginnt deshalb mit dem nächsten Durchlauf der Schleife.

### 3.9.4 do-Schleife

Im Gegensatz zur *while-Schleife*, wo die Abbruchbedingung am Anfang überprüft wird, wird bei der *do-Schleife* die Bedingung erst am Ende überprüft. Die Folge ist die, dass die Schleife mindestens einen Durchlauf macht, egal ob die Bedingung am Ende der Schleife von Anfang an erfüllt war oder nicht.

**Listing 3.40** Beispiel do-Schleife

```
int enemyIndex= 0;
do {
    //Code...
    enemyIndex++;
} while(enemyIndex > 5);
```

Auch bei der *do-Schleife* ist es möglich, mit dem Befehl `break` die Schleife zu verlassen.

## ■ 3.10 Klassen

Wie bereits eingangs erwähnt, ist C# eine sogenannte **objektorientierte Programmierung**. Hierbei geht es darum, zusammenhängende Funktionalitäten und Werte in sogenannten Objekten (was nichts anderes als Variablen sind) zu bündeln und zu kapseln.

Das bedeutet, dass sich z. B. alles, was den Spieler betrifft, in einem Objekt namens `player` befindet. Alles, was den Gegner betrifft, speichert ein Objekt namens `enemy`. Nur was machen wir, wenn wir nicht nur einen Gegner, sondern zwei oder noch mehr Gegner haben wollen? Müssen wir für jeden Gegner ein komplett neues Objekt programmieren? Nein, natürlich nicht, denn hier kommen nun Klassen ins Spiel.

Eine Klasse ist eine Vorlage für Objekte. Sie beschreibt, wie Objekte dieser Klasse „aussehen“ und wie sie funktionieren sollen. Eine Klasse beginnt mit dem Wort `class`, gefolgt vom Namen der Klasse. Dieser beginnt in C# zur besseren Unterscheidung zu normalen Variablen mit einem Großbuchstaben. Danach folgt der Code der Klasse, der in geschweiften Klammern eingeschlossen wird:

**Listing 3.41** Einfache Klasse

```
class Enemy{
    int health;
}
```

In Unity stellt ein Skript genau eine Klasse dar. Der Name des Skriptes muss den gleichen Namen tragen wie die Klasse in dem Skript. Wenn Sie nun ein Objekt von einem Skript bzw. einer Klasse erstellen möchten, ist dies in Unity ganz einfach. Da normalerweise Skripte als Komponenten gelten (mehr dazu im Kapitel „Skript-Programmierung“), können Sie diese ganz einfach auf ein jeweiliges *GameObject* ziehen, wodurch automatisch ein Objekt dieser Klasse erstellt wird. Diesen Vorgang nennt man auch Instanziieren. Das Ergebnis, also das Objekt, bezeichnet man deshalb auch häufig als Instanz dieser Klasse.

### 3.10.1 Komponenten per Code zuweisen

Anstatt ein solches Komponenten-Skript per Drag & Drop auf ein *GameObject* zu ziehen, können Sie alternativ auch per Programmcode einem *GameObject* ein Skript bzw. ein Objekt Ihrer Klasse zuweisen. Hierfür gibt es den Befehl `AddComponent`.

**Listing 3.42** Komponente einem GameObject per Code zufügen

```
gameObject.AddComponent<MyScriptName>();
```

### 3.10.2 Instanziierung von Nichtkomponenten

Instanzen von Nichtkomponenten, wie z.B. *GameObjects* oder Skript-Klassen, die nicht von der Klasse `MonoBehaviour` erben (mehr dazu im Kapitel „Skript-Programmierung“), werden für gewöhnlich auf dem typischen C#-Weg erzeugt. Hierfür deklarieren Sie zunächst eine Variable vom Typ der Klasse. Als Nächstes kommt die Instanziierung des Objektes. Hierbei wird das neue Objekt erzeugt, was Sie mit dem Schlüsselwort `new` und dem Klassennamen machen.

**Listing 3.43** Instanziierung eines Objektes

```
Enemy enemy;  
enemy = new Enemy();
```

Die beiden obigen Zeilen können aber auch kürzer gefasst werden, und zwar so:

**Listing 3.44** Kurzform einer Instanziierung

```
Enemy enemy = new Enemy();
```

Die beiden Klammern hinter `new` und dem Klassennamen sind wichtig und müssen immer geschrieben werden, da es sich hier nicht um die Klasse an sich handelt, sondern um den *Konstruktor* dieser Klasse, eine Methode, die beim Erstellen des Objektes ausgeführt wird. Mehr zum Thema *Konstruktor* erfahren Sie z.B. in den bereits erwähnten Büchern von Walter Doberenz und Thomas Gewinnus.

## ■ 3.11 Methoden/Funktionen

Algorithmen, also Programmcodezeilen, die keine Variablendeklarationen o.Ä. sind, können innerhalb einer Klasse nur in Methoden geschrieben werden. Methoden sind Codeeinheiten, die über den Aufruf ihres Methodennamens angesprochen werden und sowohl Übergabeparameter als auch Rückgabewerte besitzen können.

Beachten Sie, dass in JavaScript normalerweise nicht der Begriff „Methode“ genutzt wird, stattdessen findet dort die Bezeichnung „Funktion“ Anwendung. In beiden Fällen wird aber

das Gleiche gemeint. Und weil Unity, wie bereits erwähnt, nicht nur C# als Sprache unterstützt, nutzen Unity-Entwickler häufig beide Begriffe.

Für Methoden/Funktionen gibt es eine ähnliche Namenskonvention wie für Variablen, nur dass diese mit einem Großbuchstaben beginnen, z.B. `SetPosition`. Eine Methodendefinition besitzt zwei wichtige Teile:

- **Übergabeparameter**, sie werden hinter dem Methodennamen innerhalb von runden Klammern mit dem Variablentyp definiert. Besitzt die Methode keine Übergabeparameter, werden trotzdem die Klammern geschrieben. Beachten Sie hierbei, dass bei der Übergabe einer Variablen eine Kopie erstellt wird, die dann in der Methode genutzt wird. Es ist also nicht die gleiche Variable, die der Methode übergeben und die in der Methode an sich verwendet wird.
- **Rückgabewert**, er wird vor dem Methodennamen in Form einer Typdefinition festgelegt. Der Rückgabewert wird innerhalb der Methode mit dem Schlüsselwort `return` zurückgegeben. Soll die Methode keinen Rückgabewert besitzen, wird vor die Methode das Schlüsselwort `void` geschrieben.

Die folgende Methode überprüft, ob zwei übergebene String-Variablen inhaltlich gleich sind.

#### **Listing 3.45** Beispiel-Methode

```
bool IsEqual(string stringA, string stringB)
{
    bool result = false;
    if (stringA.ToLower() == stringB.ToLower())
        result = true;
    return result;
}
```

Beachten Sie, dass das Beispiel ebenfalls auf Methoden zugreift, und zwar `ToLower()`, die jede String-Variablen bereitstellt. Diese gibt den Inhalt der Variablen `stringB` in Kleinbuchstaben zurück, sodass aus „Test“ ein „test“ wird.

Die nachfolgende Zeile vergleicht nun, ob die beiden Strings die gleichen Buchstaben besitzen, und gibt dann entsprechend ein TRUE oder ein FALSE über die `result`-Variable zurück.

Der Aufruf der obigen Methode kann dann wie in Listing 3.46 aussehen.

#### **Listing 3.46** Aufruf der Beispiel-Methode

```
//Code...
bool equal;
string myName = "Test";
string colliderName = "TEST"
equal = IsEqual(myName, colliderName); //Gibt ein TRUE zurueck
//Code...
```



### Don't Repeat Yourself

Eine wichtige Aufgabe von Methoden besteht darin, wiederholende Code-Abschnitte zu vermeiden. Mehrfach genutzter Code wird dabei in eigene Methoden ausgliedert, wodurch der Code nur noch an einer Stelle gewartet werden muss. Müssen Sie z.B. in einer Methode zwei Positionen vergleichen, die aber zuvor noch auf die gleiche Weise berechnet werden müssen, könnten Sie die Logik der Berechnung in eine Methode ausgliedern. Über die Übergabeparameter der Methode geben Sie ihr dann die Initialwerte für die Berechnung mit.

## 3.11.1 Werttypen und Referenztypen

Auch wenn ich jetzt nicht zu tief in die Materie einsteigen möchte, so sollten Sie unbedingt noch wissen, dass es zwei unterschiedliche Datentypen von Variablen gibt, deren Unterschiede sich gerade bei der Übergabe an Methoden bemerkbar machen.

- **Werttypen** sind Datentypen, die den Wert enthalten. Beispiele hierfür sind die einfachen Typen *int*, *bool* oder auch *Enumerationen*. Übergeben Sie einen Werttyp einer Methode, wird also auch der Wert kopiert. Sie können in diesem Fall innerhalb der Methode die Variable beliebig verändern, ohne dass dies Einfluss auf die von außen übergebene Variable hat.
- **Referenztypen** sind Datentypen, die nicht den Wert, sondern nur einen *Zeiger* auf einen anderen Speicherplatz enthalten (also die Speicheradresse). Erst in diesem anderen Speicherort ist der eigentliche Inhalt der Variablen abgelegt. Übergeben Sie nun einen Referenztyp einer Methode, wird hier also nicht der Inhalt, sondern nur der Zeiger kopiert. Das Ändern der Variabldaten innerhalb der Methode ändert damit auch die Daten des Objektes, das der Methode von außen übergeben wurde. Ein typisches Beispiel für Referenztypen sind Instanzen von Klassen.

Um Sie nun noch endgültig zu verwirren, will/muss ich Ihnen jetzt noch sagen, dass auch String-Variablen Referenztypen sind. Aber, und das ist die gute Nachricht, Strings haben eine Wertsemantik. Das bedeutet, dass sich *Strings* mehr oder weniger wie Werttypen verhalten. Von daher können Sie *Strings* meistens genauso verwenden wie *int*-Variablen oder *Booleans*. Für die vorherige Beispiel-Methode bedeutet das, dass Sie den Wert von `stringA` innerhalb der Methode verändern können, ohne dass dies Auswirkung auf den übergebenen Wert der Variablen `myName` hat.

## 3.11.2 Überladene Methoden

In C# gibt es auch die Möglichkeit, zwei Methoden in einer Klasse mit dem gleichen Namen zu definieren. Und zwar funktioniert dies dann, wenn sich die Übergabeparameter unterscheiden. Diese Technik bezeichnet man auch als das Überladen einer Methode.

**Listing 3.47** Überladung von Methoden

```
static bool IsEqual(string stringA, string stringB)
{ //Code...
static bool IsEqual(string stringA, string stringB, bool ignoreCase)
{ //Code...}
```

## ■ 3.12 Lokale und globale Variablen

Je nachdem, wo Sie in Klassen eigene Variablen deklarieren, existieren diese nur in einer Funktion oder in der gesamten Klasse.

- **Lokale Variablen** werden innerhalb einer Methode deklariert und existieren nur in dieser einen Methode.
- **Globale Variablen** werden außerhalb von Methoden deklariert und gelten in der gesamten Klasse.

### 3.12.1 Namensverwechslung verhindern mit this

Durch die unterschiedlichen Gültigkeitsbereiche kann es vorkommen, dass eine lokale Variable und eine globale Variable den gleichen Namen tragen. Um hier Missverständnissen vorzubeugen, bietet C# das Schlüsselwort **this** an, das auf die eigene Klasseninstanz verweist. Möchten Sie in diesem Fall also auf die lokale Variable zugreifen, schreiben Sie einfach den Namen der Variablen. Möchten Sie auf die globale Variable zugreifen, schreiben Sie stattdessen erst **this**, dann einen Punkt und danach den Namen.

**Listing 3.48** Zugriff mit this

```
int quantity = 0;
void Add(int quantity) {
    this.quantity += quantity;
}
```

## ■ 3.13 Zugriff und Sichtbarkeit

*Globale Variablen* wie auch alle Methoden können mit zusätzlichen *Zugriffsmodifizierern* erweitert werden. Diese legen fest, ob diese nur innerhalb der Klasse sichtbar und damit auch nutzbar sind oder ob auch von einer anderen Klasse bzw. Instanz auf diese zugegriffen werden kann.

- **public** ermöglicht den Zugriff von überall. *Public*-Variablen werden zudem in Unity im *Inspector* angezeigt.

- **private** grenzt den Zugriff auf die eigene Klasse ein.

- **protected** grenzt den Zugriff auf die eigene Klasse und alle anderen Klassen ein, die von dieser erben.

- **internal** grenzt den Zugriff auf alle Klassen der eigenen Assembly ein (wichtig bei DLLs).

Wenn Sie keinen dieser Schlüsselwörter angeben, gelten Variablen und Methoden in C# automatisch als *private*. Klassen können übrigens nur mit **public** und **internal** ausgezeichnet werden. Fehlt das Schlüsselwort, gilt diese als *internal*.

**Listing 3.49** Beispiel für Sichtbarkeit

```
public class Test {
    private int health;
    public bool IsEqual(string stringA, string stringB) {
        //Code...
    }
}
```

## ■ 3.14 Statische Klassen und Klassenmember

Es gibt Situationen, wo Sie z. B. Methoden oder Variablen programmieren, die unabhängig einer erstellten Instanz funktionieren sollen, z. B. einen Zähler aller bisher erstellten Instanzen dieser Klasse. In diesem Fall darf natürlich nicht jede Instanz eigene Zählmechanismen besitzen, deren Ergebnisse sich dann vielleicht sogar noch unterscheiden. Für solche Anwendungszwecke gibt es die sogenannten *statischen* Methoden/Variablen/... oder auch allgemein *Klassenmember* genannt. (Im Gegensatz dazu werden die herkömmlichen Methoden/Variablen/... als *Instanzmember* bezeichnet.) Sie werden mit dem Schlüsselwort **static** deklariert und sind nicht über eine Instanz, sondern direkt über den Klassennamen erreichbar. Unity nutzt dies z. B. in seiner Mathematik-Klasse *Mathf*.

**Listing 3.50** Aufruf der statischen RoundToInt-Methode

```
int myInteger = Mathf.RoundToInt(1.5f);
```

Die Deklaration einer statischen Methode sieht dann so aus:

**Listing 3.51** Deklaration einer statischen Methode

```
public static void SetLevelSettings(int levelIndex) {
    //Code...
}
```

Die Klasse *Mathf* geht sogar noch weiter. Hier wurde die ganze Klasse als statisch deklariert, sodass der Compiler ausschließlich *Klassenmember* zulässt. Member ist der Oberbegriff für alle „Dinge“, die sich in einer Klasse befinden können, z. B. Methoden, Variablen usw. Eine statische Klasse wird dann wie folgt definiert:

**Listing 3.52** Statische Klasse

```
public static class GameSettings
{
    public static string playerName = "";
    public static void SetLevelSettings(int levelIndex)
    {
        //Code...
    }
}
```

## ■ 3.15 Parametermodifizierer out/ref

Normalerweise wird eine Variable, die einer Funktion übergeben wird, wie bereits oben erwähnt, kopiert. Das bedeutet, dass eine Veränderung von Werttyp-Variablen (oder mit einer Werttyp-Semantik) innerhalb einer Funktion keinen Einfluss auf die von außen übergebene Variable hat. Dies kann aber in bestimmten Fällen vielleicht gewünscht sein.

Hierfür gibt es die Parametermodifizierer `ref` und `out`, die Variablen (egal ob Werttyp oder Verweistyp) per Referenz an eine Methode übergeben und nicht durch die Kopie des Inhalts. Hierbei müssen sowohl der Übergabeparameter in der Methode mit dem Modifizierer gekennzeichnet werden als auch die Variable, die der Methode übergeben wird.

**Listing 3.53** Beispieldmethode mit ref-Modifizierer

```
public bool IsEqual(ref myName, colliderName) {
    bool result = false;
    if (stringA.ToLower() == stringB.ToLower())
    {
        result = true;
        stringA = stringB;
    }
    return result;
}
```

**Listing 3.54** Aufruf einer Methode mit dem ref-Modifizierer

```
string myName = "Player";
string colliderName = "player";
equal = IsEqual(ref myName, colliderName);
```

In dem obigen Fall hat `myName` am Anfang den Wert „Player“ und `colliderName` den Wert „player“. Nach dem Aufruf der Methode haben beide Variablen den Wert „player“.

`out` wird im Übrigen genauso verwendet wie `ref`. Der Unterschied dieser beiden Parameter ist folgender:

- **ref** verlangt eine bereits initialisierte Variable. Die übergebene Variable muss also bereits vor der Übergabe an die Methode einen Wert besitzen.
- **out** benötigt keine initialisierte Variable. Dafür muss aber in der Methode selbst sichergestellt werden, dass in dieser auf jeden Fall ein gültiger Wert zugewiesen wird.

Ein großer Vorteil ist, dass Sie hierüber nun die Möglichkeit haben, mehrere Werte aus einer Methode zurückzugeben. Die Unity-Klasse `Physics` nutzt beispielsweise bei der Methode `Raycast` einen `out`-Parameter. Die Methode sendet von einem Startpunkt einen Teststrahl aus, um zu überprüfen, ob sich in der Richtung andere Objekte befinden. Der eigentliche Rückgabewert der Funktion ist hier nur ein Boolean und sagt lediglich aus, ob der virtuelle Teststrahl ein Objekt getroffen hat. Wurde etwas getroffen, können Sie über den `out`-Parameter weitere Informationen über das getroffene Objekt erhalten.

**Listing 3.55** Raycast-Methode der Physics-Klasse

```
RaycastHit hit;
if (Physics.Raycast(transform.position, Vector3.down, out hit))
    float distanceToGround = hit.distance;
```

## ■ 3.16 Array-Übergabe mit params

Es gibt neben den obigen Parametermodifizierern auch noch den Modifizierer `params`. Dieser vereinfacht die Übergabe von Arrays an eine Methode, sodass Sie bei der Übergabe von einem oder mehreren Werten nicht unbedingt erst ein Array eines Typs definieren müssen. Sie können einfach durch Komma getrennte Variablen-Aufzählungen übergeben.

**Listing 3.56** Methodendefinition mit params

```
public int DamageAddition(params int[] damageValues) {
    int val = 0;
    foreach(int current in damageValues)
    {
        val += current;
    }
    return val;
}
```

In der Methode wird der Übergabeparameter `damageValues` wie ein ganz normales Array des Typs `int` behandelt. Dies ist zunächst nichts Besonderes. Das Besondere tritt erst beim Übergeben des Arrays an die Methode zutage.

**Listing 3.57** Parameterübergabe mit params

```
int sum;
int damage1 = 5;
int damage2 = 3;
sum = DamageAddition(damage1, damage2);
int[] damageArray = new int[2];
damageArray[0] = 5;
damageArray[1] = 3;
sum = DamageAddition(damageArray);
```

## ■ 3.17 Eigenschaften und Eigenschaftsmethoden

Stellen Sie sich vor, Sie haben eine Klasse NPC, die eine *public*-Variable health besitzt. Objekten dieser Klasse können Sie nun natürlich einen beliebigen Wert zuweisen, auch wenn es vielleicht gar nicht erlaubt ist, dass Objekte einen negativen health-Wert besitzen. Anstatt dies nun überall dort zu berücksichtigen, wo Sie diesen Wert weiterverarbeiten, wäre es da doch sinnvoll, von vornherein dafür zu sorgen, dass dies nicht geschehen kann. Hierfür gibt es in C# sogenannte Eigenschaftsmethoden, die bereits beim Übertragen des Wertes diesen überprüfen und auch verändern können. Eine *Eigenschaftsmethode*, kurz *Eigenschaft* bzw. *Property* genannt, besteht aus zwei Submethoden, die sich *Get-* und *Set-Accessoren* nennen.

- Get dient dem Abfragen des Wertes.
- Set dient dem Zuweisen des Wertes.

Die eigentliche Variable, in unserem Fall ist es health, wird als *private* deklariert und ist damit nur noch über die Eigenschaftsmethoden zugänglich. Dieses Vorgehen wird auch Datenkapselung genannt. Die Klasse NPC sähe dann mit diesen *Accessoren* wie folgt aus:

**Listing 3.58** Beispiel einer Eigenschaft

```
public class NPC {
    private int health;
    public int Health
    {
        get {
            return health;
        }
        set {
            health= value;
            if(health<0)
                health = 0;
        }
    }
}
```

Beachten Sie hierbei die Groß- und Kleinschreibung: Die Eigenschaftsmethoden werden großgeschrieben und sind *public*. Die Variablen werden kleingeschrieben und sind *private*.

**Listing 3.59** Zugriff auf eine Eigenschaft

```
NPC npc = new NPC();
int h = 3;
npc.Health = 3; //setzt die private-Variable health auf 3
npc.Health -= 5; //zieht 5 vom aktuellen Wert ab, sodass health eigentlich -2 wäre
h = npc.Health; //Health gibt den Wert 0 zurück
```

Weil in Unity der *Inspector* nur *public*-Variablen anzeigt, aber keine Eigenschaftsmethoden, wird in Unity häufig auf Eigenschaftsmethoden verzichtet.

## ■ 3.18 Vererbung

Ein wichtiges Merkmal von *objektorientierter Programmierung* ist die Möglichkeit der Vererbung. Das bedeutet nichts anderes, als dass Sie als Programmierer eine neue Klasse entwickeln können, die eine andere Klasse mit ihren ganzen Methoden, Variablen etc. als Basis nutzen kann, ohne den Code noch einmal zu schreiben. Diese neue Klasse bezeichnet man auch als abgeleitete Klasse. Die Klasse, von der sie erbt, wird als Basisklasse bezeichnet. Wenn eine Klasse nun von einer anderen erben soll, wird hinter dem Klassennamen ein Doppelpunkt geschrieben, gefolgt von dem Namen der Basisklasse.

**Listing 3.60** Beispiel Vererbung

```
public class MyScriptName : MonoBehaviour {
    //Code
}
```

Auch wenn Sie selber vielleicht nicht viel Gebrauch von Vererbung machen, ist dies ein sehr wichtiges Thema für Sie als Unity-Programmierer. Jede Klasse, die in Unity erstellt wird und als Komponente genutzt werden soll, muss von der Unity-Klasse `MonoBehaviour` erben. Diese beinhaltet alles Notwendige, was eine Klasse benötigt, um eben als Komponente genutzt werden zu können. Mehr zu diesem Thema erfahren Sie in Kapitel 4, „Skript-Programmierung“.

### 3.18.1 Basisklasse und abgeleitete Klassen

Wenn in C# eine Klasse von einer anderen erben soll, muss hinter dem Klassennamen ein Doppelpunkt und dahinter folgend der Name der Basisklasse geschrieben werden. Nehmen wir an, wir haben eine Klasse `Person` mit einer öffentlichen Variablen `name` und einer privaten Variablen `points`.

**Listing 3.61** Basisklasse Person

```
public class Person {
    public string name = "";
    private int points = 0;
}
```

Die abgeleitete Klasse `NPC`, die eine zusätzliche Variable `health` besitzen soll, würde dann wie folgt aussehen:

**Listing 3.62** Abgeleitete Klasse NPC

```
public class NPC : Person {
    public int health;
}
```

Jetzt können Sie ein Objekt der Klasse `NPC` erstellen und beide öffentlichen Variablen nutzen:

**Listing 3.63** Nutzen der Klasse NPC mit Zugriff auf geerbte Variablen

```
NPC npc = new NPC();
npc.name = "Legolas";
npc.health = 10;
```

**3.18.2 Vererbung und die Sichtbarkeit**

Auch wenn im obigen Beispiel die Klasse NPC von der Basisklasse Person erbt, so gelten trotzdem noch die Regeln der Sichtbarkeit, die ich bereits weiter oben erklärt hatte. Sie können also in einer abgeleiteten Klasse nicht auf eine *private*-Variable oder eine *private*-Methode der Basisklasse zugreifen. Wenn Sie trotzdem den Zugriff erlauben möchten, dann gibt es die Möglichkeit, diese Member statt mit *private* mit dem Schlüsselwort *protected* zu definieren.

**Listing 3.64** Nutzen von protected

```
public class Person {
    public string name = "";
    protected int points = 0;
}
```

Dadurch ist die Variable *points* zwar immer noch nicht von außen zu sehen, aber in der abgeleiteten Klasse NPC kann auf diese zugegriffen werden.

**Listing 3.65** Zugriff auf eine protected-Variable in der abgeleiteten Klasse

```
public class NPC : Person {
    public int health;
    public void DoSomething() {
        points = 100;
    }
}
```

**3.18.3 Geerbte Methode überschreiben**

Stellen Sie sich vor, Sie haben eine Basisklasse Person, die neben verschiedenen Variablen und Methoden unter anderem eine Methode namens Shoot besitzt. Zum Darstellen dieser Funktionalität soll hier zunächst einfach ein kurzer Text in der Konsole (*Console-Tab*) ausgegeben werden. Dies können Sie mit dem Befehl `Debug.Log` machen. Während die Klasse Person mit einem Standardtext „Peng!“ ausgestattet wird, soll die Klasse NPC nun etwas anderes von sich geben. In C# gibt es für diesen Zweck die Möglichkeit, Methoden zu überschreiben. Hierfür muss zunächst einmal die zu überschreibende Methode in der Basisklasse als überschreibbar deklariert werden. Dies wird mit dem Schlüsselwort *virtual* gemacht.

**Listing 3.66** Überschreibbare Methode

```
public class Person {
    public virtual void Shoot() {
        Debug.Log("Peng!");
    }
}
```

Die abgeleitete Klasse kann nun (muss aber nicht) diese Methode überschreiben. Das Schlüsselwort hierfür lautet `override`.

**Listing 3.67** Geerbte Methode überschreiben

```
public class NPC : Person {
    public override void Shoot() {
        Debug.Log("Krawumm!");
    }
}
```

**3.18.4 Zugriff auf die Basisklasse**

Wenn Sie nun in der abgeleiteten Klasse eine Methode der Basisklasse überschrieben haben, aber nun auf die Methode der Basisklasse zugreifen möchten, stellt sich natürlich die Frage, wie man das denn nun macht. Dies wird über das Schlüsselwort `base` ermöglicht. Über `base` greifen Sie auf Methoden, Variablen etc. der Basisklasse zu.

**Listing 3.68** Zugriff auf die Methode der Basisklasse

```
public class NPC : Person {
    public override void Shoot() {
        base.Shoot();
        Debug.Log("Krawumm!");
    }
}
```

**3.18.5 Klassen versiegeln**

Eine weitere Möglichkeit der Vererbung ist das Versiegeln. Hierunter wird das Unterbinden der Weitervererbungsfähigkeit einer Klasse verstanden. Mit anderen Worten: Wenn Sie bei der Klassendefinition das Schlüsselwort `sealed` voranstellen, kann keine andere Klasse mehr von dieser erben.

**Listing 3.69** Versiegelte Klasse

```
public sealed class NPC : Person { //Code... }
```

## ■ 3.19 Polymorphie

*Objektorientierte Programmiersprachen* bieten eine weitere nützliche Fähigkeit, die sich *Polymorphie* nennt. Hierbei können Sie alle Objekte/Instanzen, die eine gemeinsame Basisklasse haben, auch gleich behandeln. Die folgende Methode löst von einem übergebenen Objekt die Shoot-Funktion aus. Dabei ist es dem Skript völlig egal, ob das Objekt nun tatsächlich vom Typ Person ist oder eine Klasse, die von Person abgeleitet wurde (z.B. NPC aus dem vorherigen Beispiel).

**Listing 3.70** Methode LetsShoot mit Person-Parameter

```
void LetsShoot(Person person) {
    person.Shoot();
}
```

**Listing 3.71** Übergabe verschiedener Objekte an LetsShoot

```
Person person = new Person();
NPC npc = new NPC();
LetsShoot(person);
LetsShoot(npc);
```

## ■ 3.20 Schnittstellen

Ein weiteres Feature der *Objektorientierung* von C# sind *Schnittstellen* (engl. *Interfaces*). Mit diesen definieren Sie Gemeinsamkeiten unterschiedlicher Klassen, über die Sie diese wieder ansprechen können. Auf diese Weise können Sie unterschiedliche Klassen gleich behandeln, was in bestimmten Situationen sehr sinnvoll sein kann. Es gibt allgemeine Schnittstellen, die bereits von Microsoft vordefiniert wurden, die auch für bestimmte Funktionalitäten wichtig sind, und Sie können Ihre eigenen Schnittstellen definieren.

### 3.20.1 Schnittstelle definieren

Zum Definieren einer Schnittstelle wird das Schlüsselwort `interface` genutzt. Der Name einer Schnittstelle beginnt dann konventionsgemäß immer mit einem I, in dem obigen Beispiel wäre es also `IShootable`. Der Rest der Schnittstelle ähnelt dann einer Klasse, mit dem Unterschied, dass die Methoden nicht ausprogrammiert werden. Denn dies geschieht ja erst in den Klassen, die diese Schnittstelle implementieren. Die `IShootable`-Schnittstelle könnte dann wie folgt aussehen:

**Listing 3.72** Beispiel-Interface

```
public interface IShootable {
    public void Shoot();
}
```

Ein wichtiger Unterschied zwischen Klassen und Schnittstellen betrifft die Zugriffsmodifizierer. Während bei Klassen alles als *private* angesehen wird, was keinen Zugriffsmodifizierer hat, gelten bei Schnittstellen diese als *internal*. Schließlich dient eine Schnittstelle ja auch dem Beschreiben der Außenwirkung einer Klasse. Alle anderen Zugriffsmodifizierer, abgesehen von *public* und eben *internal*, sind bei Schnittstellen sogar verboten.

### 3.20.2 Schnittstellen implementieren

Soll eine Klasse eine Schnittstelle implementieren, wird dies genauso gemacht wie das Erben von einer Klasse, also mit einem Doppelpunkt. Wird sowohl von einer Klasse geerbt wie auch eine (oder mehrere) Schnittstellen implementiert, werden diese nach der Nennung der Basisklasse kommagetrennt aufgezählt. Die Implementierung der *IShootable*-Schnittstelle könnte dann wie folgt aussehen. Beachten Sie hierbei vor allem, dass das Skript die Methode *Shoot* besitzt, was durch die Implementierung der Schnittstelle erzwungen wird.

**Listing 3.73** Skript mit implementiertem Interface

```
using UnityEngine;
using System.Collections;
public class Gun : MonoBehaviour, IShootable {
    private WeaponController weaponController;
    void Awake () {
        weaponController = gameObject.GetComponent<WeaponController>();
    }
    public void UseMe() {
        weaponController.Switch (this);
    }
    public void Shoot () {
        Debug.Log ("Peng!");
    }
}
```

Die einzelnen Bereiche des Skripts sind momentan nicht so wichtig und werden noch in den anderen Kapiteln genauer behandelt. Trotzdem möchte ich schon einmal vorwegnehmen, was hier eigentlich passiert. Sobald die Methode *Awake* aufgerufen wird, wird die Skript-Komponente *WeaponController* (deren Code gleich noch behandelt wird) gesucht und einer *private*-Variablen zugewiesen. Wird nun die Methode *UseMe* aufgerufen, wird die eigene Skript-Instanz von *Gun* an die *Switch*-Methode des *WeaponControllers* übergeben. Die Methode *Shoot* schreibt schließlich nur noch zur Demonstration den Text „Peng!“ in die Konsole von Unity. Später würde hier der Code programmiert, der dann z.B. ein Projektil verschießt.

### 3.20.2.1 Unterstützung durch MonoDevelop

Bei der Implementierung einer Schnittstelle hilft Ihnen die Programmierumgebung MonoDevelop. Wenn Sie im obigen Beispiel nun hinter MonoBehaviour die Implementierung IShootable geschrieben haben, gehen Sie mit Ihrem Mauszeiger auf IShootable und drücken die rechte Maustaste. In dem nun erscheinenden Kontextmenü gehen Sie auf **Refactor/Implement implicit**. Danach werden alle Eigenschaften, Methoden etc. in Ihre Klasse automatisch eingefügt. Diese müssen Sie nun nur noch mit Leben füllen. Die *Exception* (Fehlermeldung), die per Default eingefügt wird, können Sie dann natürlich entfernen.

### 3.20.3 Zugriff über eine Schnittstellen

Um auf eine Schnittstelle zuzugreifen, können Sie ein *Interface* nutzen wie einen Variablen-typ. In dem folgenden Skript wird deshalb sowohl bei der Parameterdefinition von der Methode Switch wie auch bei der *public*-Variablendefinition die Schnittstelle genommen. Hierdurch kann Switch jede beliebige Klasse übergeben werden, solange diese die Schnittstelle IShootable besitzt und damit auch die Shoot-Funktion.

**Listing 3.74** Zugriff über Interface

```
using UnityEngine;
using System.Collections;
public class WeaponController : MonoBehaviour {
    public IShootable weapon;
    public void Switch(IShootable newWeapon) {
        weapon = newWeapon;
    }
    public void ShootCurrentWeapon() {
        weapon.Shoot();
    }
}
```

## ■ 3.21 Namespaces

Wenn wir am Ende dieses Buches das Demo-Spiel entwickeln, werden Sie feststellen, dass während eines Unity-Projektes doch recht viele Klassen entstehen. Hinzu kommt noch, dass auch das Framework, das Unity im Hintergrund für die Programmierung nutzt, ebenfalls viele Klassen bereitstellt. Damit man hier nicht die Übersicht verliert und es zu Verwechslungen kommt, gibt es sogenannte *Namespaces*.

*Namespaces* sind Organisationsstrukturen, die zusammenhängende Typen (Klassen usw.) gliedern und in logische Einheiten zusammenfassen. Sie können sich dies wie einen Ordner bei einem Dateisystem vorstellen, in dem verschiedene Dateien enthalten sind. Wenn Sie nun in Unity einen Typen nutzen wollen, muss der *Namespace* in dem Projekt referenziert und in die Klasse eingebunden sein. Normalerweise kümmert sich Unity um dies alles. Nur wenn Sie spezielle Typen nutzen möchten, müssen Sie selber Hand anlegen.

Das Einbinden eines *Namespace* machen Sie am Anfang einer Skript-Datei mit dem Signalwort `using`. Die folgende Beispielklasse bindet den Namespace `System.Collections.Generic` ein, um den Typ `List` nutzen zu können.

**Listing 3.75** Einbinden eines Namespace

```
using System.Collections.Generic;
public class TestClass {
    private List<string> myList;
}
```

Wenn Sie sich jetzt noch einmal das Schnittstellen-Beispiel ansehen, werden Sie bemerken, dass auch dort bereits *Namespaces* auf diese Weise eingebunden wurden, und zwar die *Namespaces* `UnityEngine` und `System.Collections`. Mehr zu diesen beiden *Namespaces* erfahren Sie im Kapitel „Skript-Programmierung“.

## ■ 3.22 Generische Klassen und Methoden

Der Begriff *Generisch* bedeutet in der *objektorientierten Programmierung* eigentlich nur, dass z.B. Methoden oder Klassen allgemein, das heißt typunabhängig entworfen werden. Welcher Typ am Ende dann genommen werden soll, wird dann der Klasse (oder Methode) über spitze Klammern übergeben. Im Falle von C# wird dieser zuzuweisende Typ in der generischen Klasse (bzw. Methode) meist durch den Buchstaben *T* symbolisiert.

**Listing 3.76** Einfaches Beispiel einer generischen Klasse

```
using UnityEngine;
using System.Collections;
public class MyGeneric<T> {
    public T current;
}
```

Überall dort, wo der Parameter *T* in der generischen Klasse eingesetzt wurde, wird später der übergebene Typ eingesetzt. Die Variable *current* wird also den Datentyp haben, den Sie beim Deklarieren der Instanz mitgeben.

**Listing 3.77** Nutzen einer generischen Klasse

```
MyGeneric<string> test = new MyGeneric<string>();
test.current = "ddd";
MyGeneric<int> test2 = new MyGeneric<int>();
test2.current = 2;
```

### 3.22.1 List

Ein generischer Typ, der in Unity-Projekten gerne genutzt wird, ist der Typ `List<T>`. Dieser arbeitet ähnlich wie ein Array, nur dass Objekte dieses Typs wesentlich mehr Funktionalitäten als normale Arrays anbieten. So können Sie diesen Objekten über die Methode `Add` beliebig viele neue Elemente anhängen, diese mit verschiedenen Befehlen durchsuchen oder mithilfe von `Remove` bestimmte Elemente gezielt löschen. Sie müssen lediglich beim Erstellen eines `List`-Objektes den Typ mitgeben, den das Objekt aufnehmen soll. Beachten Sie deshalb im folgenden Beispiel vor allem die Variablendeclaration von `myNumbers`, aber auch das Einbinden des Namespaces `System.Collections.Generic`, zu dem die `List`-Klasse gehört.

**Listing 3.78** Nutzen der List-Klasse

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
public class GenericTest : MonoBehaviour{
    private List<int> myNumbers = new List<int>();
    void Start () {
        myNumbers.Add(2);
        myNumbers.Add(33);
        myNumbers.Add (17);
        if (myNumbers.IndexOf(33)>=0)
            myNumbers.Remove(33);
        Debug.Log(myNumbers.Count.ToString()); //Gibt eine 2 aus
    }
}
```



#### List-Objekte sortieren

Eine weitere Stärke der `List`-Klasse ist das Sortieren der verschiedenen Elemente mithilfe der Methode `Sort`. Hierbei sollten Sie aber beachten, dass nicht jeder Typ sortiert werden kann. Nur wenn dieser die Schnittstelle `IComparable` aus dem Namespace `System` implementiert hat, funktioniert dies. Mehr zu diesem Thema erfahren Sie im Internet (Stichwort „`IComparable`-Schnittstelle“).

### 3.22.2 Dictionary

Eine weitere Klasse aus dem `Generic`-Namespace, die Sie kennen sollten, ist die `Dictionary< TKey, TValue >`-Klasse, kurz `Dictionary`. Diese besitzt sogar gleich zwei generische Parameter. Ein `Dictionary` speichert sogenannte *Schlüssel-Wert-Paare*. Dabei stellt der erste Wert einen eindeutigen Identifizierer (*Schlüssel*) und der zweite den Wert dieses Schlüssels dar. Jeder Schlüssel darf dabei nur einmal im `Dictionary` vorkommen. Das nächste Beispiel zeigt anhand einer einfachen Inventarverwaltung einige Basisfunktionen dieser Klasse. So wird gezeigt, wie Sie mit `ContainsKey` feststellen, ob es diesen Schlüssel bereits im `Dictionary` gibt, wie Sie mit der `Add`-Methode ein neues *Schlüssel-Wert-Paar* erzeugen und wie Sie über den *Schlüssel* auf den dazugehörigen Wert zugreifen, um diesen zu verändern.

**Listing 3.79** Einfaches Inventarsystem mit einem Dictionary-Objekt

```
using System.Collections;
using System.Collections.Generic;
public static class SimpleInventory {
    private static Dictionary<string,int> items = new Dictionary<string, int>();
    public static void AddItems (string key, int val) {
        if (items.ContainsKey(key))
            items[key] += val;
        else
            items.Add(key, val);
    }

    public static bool RemoveItems (string key, int val) {
        if (items.ContainsKey(key)) {
            if (items[key] >= val) {
                items[key] -= val;
                return true;
            }
            else
                return false;
        }
        else
            return false;
    }
}
```

# 4

# Skript-Programmierung

Nachdem wir im vorherigen Kapitel die wichtigsten C#-Grundlagen für Unity kennengelernt haben, kommen wir nun zum eigentlichen Programmieren der Skripte. Dabei entspricht jedes Skript normalerweise genau einer *Klasse*, weshalb in Unity die Begriffe *Skript* und *Klasse* nicht selten auch synonym für einander benutzt werden.

## ■ 4.1 MonoDevelop

Zum Programmieren der Skripte liefert Unity die Programmierumgebung MonoDevelop mit. MonoDevelop ist eine Open-Source-Entwicklungsumgebung für Softwareentwickler, die im Rahmen des Mono-Projektes entwickelt wurde, einer Open-Source-Alternative zum .NET Framework von Microsoft. Unity liefert eine speziell angepasste Version aus, die unter anderem ein leichtes *Debuggen* im Zusammenspiel mit Unity ermöglicht.

Durch einen Doppelklick auf ein Skript in Unity wird automatisch MonoDevelop mit dem jeweiligen Skript geöffnet. Sollte MonoDevelop bereits mit einem anderen Skript geöffnet sein, wird ein zusätzlicher Reiter mit dem jeweiligen Skript angelegt. Das andere Skript bleibt in MonoDevelop offen, sodass ein Wechseln zwischen den Skripten möglich ist.

Für Sie als Unity-Entwickler ist vor allem das Hauptfenster von MonoDevelop wichtig, in dessen Reitern der Code der verschiedenen Skripte angezeigt wird. Aber auch der Solution-Explorer, den Sie auf der linken Seite finden, ist hilfreich. Dieser zeigt Ihnen alle Code-Dateien des Unity-Projektes nach Programmiersprachen sortiert an und ermöglicht das Öffnen weiterer Skript-Dateien.

Ganz elementar ist vor allem noch die *Speicher*-Funktion von MonoDevelop, die Sie über **File/Save** bzw. die Tastenkombination **[Strg]+[S]** erreichen. Anpassungen, die in Unity zur Verfügung stehen sollen, müssen immer zuerst in MonoDevelop gespeichert werden. Auf die Debugging-Funktionalitäten, die natürlich ebenfalls sehr wichtig sind, werden wir noch im Kapitel „Fehlersuche und Performance“ zu sprechen kommen.

Sollten Sie bereits Erfahrungen mit anderen Entwicklungsumgebungen haben und eine besondere bevorzugen (z.B. Visual Studio), können Sie diese auch anstelle von MonoDevelop nutzen. Für diesen Fall können Sie unter **Edit/Preferences** eine alternative IDE im Bereich *External Tools* als *External Script Editor* hinterlegen.

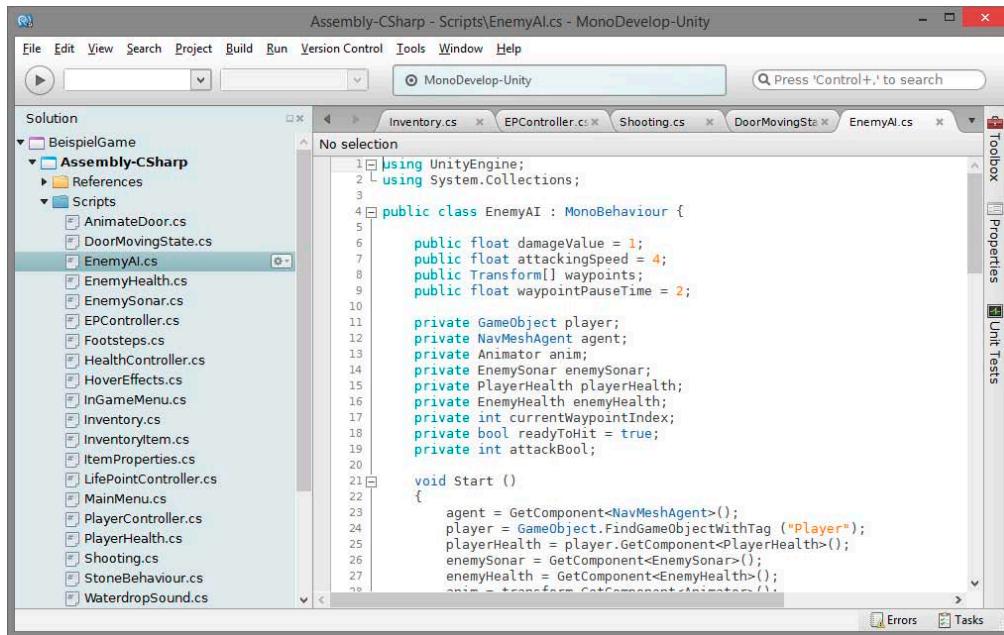


Bild 4.1 MonoDevelop

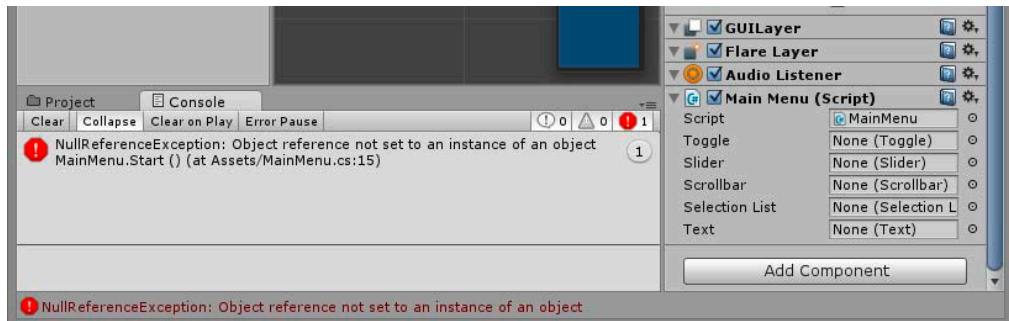
### 4.1.1 Hilfe in MonoDevelop

Die mit Unity mitgelieferte MonoDevelop-Version besitzt einige Sonderanpassungen, wozu auch eine zusätzliche Hilfe-Funktion gehört, die den Entwickler direkt zur *Scripting Reference* von Unity führt.

Selektieren Sie hierfür den Befehl, zu dem Sie weitere Informationen wünschen, und drücken Sie auf der deutschen PC-Tastatur [Strg]+[Umsch]+[?]. Alternativ können Sie dieses auch über das MonoDevelop-Menü Help/Unity API Reference erreichen. Wenn Sie in MonoDevelop nur einen Klassennamen wie z.B. Application eingeben und die Hilfe aufrufen, erhalten Sie auf diese Weise alle Informationen über Methoden und Variablen, die diese Klasse zur Verfügung stellt.

### 4.1.2 Syntaxfehler

Sollten Sie in einem Skript einen Syntaxfehler haben, z.B. ein Semikolon vergessen haben („Parsing Error“) oder eine nicht instanzierte Variable nutzen, bringt Unity nach dem Speichern dieses fehlerhaften Skriptes eine Fehlermeldung. Diese erscheint sowohl ganz unten im Unity-Fenster als auch in der Konsole (*Console-Tab*). Wenn Sie auf diese Meldung klicken, wird MonoDevelop geöffnet und Sie gelangen dorthin, wo der Fehler auftritt.



**Bild 4.2** Darstellung einer Fehlermeldung in Unity



### Weiterleitung von Fehlermeldungen

Nicht immer leitet ein Klick auf die Fehlermeldung in der Konsole tatsächlich zu der Stelle, wo der Entwickler den Programmierfehler gemacht hat. Sollten Sie z. B. ein Semikolon am Ende einer Zeile vergessen, kann es vorkommen, dass der Compiler erst die nächste Zeile als fehlerhaft erkennt, da die (vorherige) Codezeile noch nicht abgeschlossen ist und dieser neue Befehl an dieser Stelle nicht stehen darf. Gerade bei Einsteigern führt das manchmal zu langen Fehler sucharien, weil der Fehler ganz woanders zu finden ist, als er angezeigt wird.

## ■ 4.2 Nutzbare Programmiersprachen

Beim Skript-Programmieren können Sie zwischen den Programmiersprachen JavaScript, Boo und C# wählen. Auch wenn Sie die Sprachen in einem Unity-Projekt beliebig viel mischen dürfen (nicht innerhalb eines Skriptes!), ist es empfehlenswert, soweit es geht darauf zu verzichten.

An sich ist das Mixen der Sprache innerhalb eines Projektes kein Problem. Aber sobald Sie von einem Skript auf ein anderes zugreifen möchten, kann es problematisch werden. Auf die Details komme ich noch im Abschnitt 4.14, „Kompilierungsreihenfolge“, zu sprechen. Deshalb empfehle ich soweit es geht, sich innerhalb eines Projektes auf eine Sprache zu beschränken.

### 4.2.1 Warum C#?

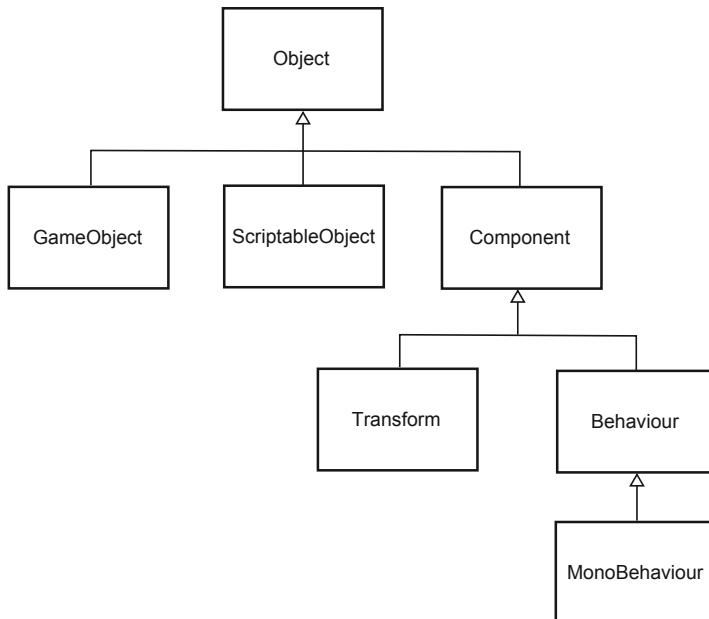
Wie Sie schon mitbekommen haben werden, nutze ich in diesem Buch die Sprache C#. Anfangs stand bei Unity die Sprache JavaScript im Fokus. Im Laufe der Zeit hat sich dies aber stark gewandelt, sodass mittlerweile C# im Mittelpunkt steht. C# bietet mehr Funk-

tionalitäten und ist zudem weiter verbreitet als der für Unity extra angepasste JavaScript-Ableger, auch häufig *UnityScript* genannt.

## ■ 4.3 Unitys Vererbungsstruktur

Wie ich bereits im vorherigen Kapitel „C# und Unity“ erwähnte, müssen alle Skripte, die als Komponenten genutzt werden sollen, von der Klasse *MonoBehaviour* erben. Diese Klasse erbt allerdings wieder von einer anderen Klasse. Am Ende haben Komponenten und *GameObjects* sogar den gleichen Ursprung, und zwar die Klasse *Object* aus dem Namespace *UnityEngine*.

Das Bild 4.3 stellt die Zusammenhänge der wichtigsten Klassen vom Namespace *UnityEngine* in einem Klassendiagramm dar. Die Pfeile zeigen dabei immer von der abgeleiteten Klasse zu der jeweiligen Basisklasse.



**Bild 4.3** Klassendiagramm vom Namespace *UnityEngine*

Die Bedeutungen dieser Klassen möchte ich im Folgenden kurz erläutern.

### 4.3.1 Object

*Object* ist die übergeordnete Basisklasse aller Objekte in Unity. Sie stellt einige fundamentale Eigenschaften und Methoden zur Verfügung, wie z.B. die statische Methode `Destroy`, um ein *Object* zu zerstören, oder auch die Methode `DontDestroyOnLoad`, die ich noch in diesem Kapitel näher erläutern werde. Von dieser Klasse werden nun die Klassen *GameObject*, *Component* und *ScriptableObject* abgeleitet.

Für erfahrenere C#-Entwickler sei an dieser Stelle noch mal darauf hingewiesen, dass es sich hierbei um eine Unity-eigene *Object*-Klasse handelt, nicht die vom Framework bereitgestellte .NET-Klasse.

### 4.3.2 GameObject

*GameObject* erbt direkt von *Object* und ist die Basisklasse aller Objekte, die sich in einer Unity-Szene befinden.

Meistens werden *GameObjects* über das Hauptmenü **GameObject** im Editor erstellt. Wenn Sie *GameObject*-Instanzen per Code erzeugen wollen, werden diese wie normale Objekte einer Klasse mit dem Signalwort `new` instanziiert.

### 4.3.3 ScriptableObject

*ScriptableObject* erbt ebenfalls direkt von *Object*. Zunächst einmal sind *ScriptableObjects* gewöhnliche Skripte, die von *ScriptableObject* erben und normalerweise dem Speichern größerer Datenmengen dienen. Instanzen von diesen Klassen werden dabei mit der statischen Methode `CreateInstance` der *ScriptableObject*-Klasse erzeugt und können zur Laufzeit wie auch zur Entwicklungszeit erstellt werden.

*ScriptableObjects* finden eher in speziellen Situationen Anwendung, z.B. um Datencontainer zu erstellen, die außerhalb einer Szene abgelegt, konfiguriert und bei Bedarf eingelesen werden können.

### 4.3.4 Component

Die *Component*-Klasse erbt ebenfalls von *Object* und ist die Basis aller Klassen, die an ein *GameObject* angehängt werden können. Deshalb werden sie auch dadurch erzeugt, indem sie mithilfe der *GameObject*-Methode `AddComponent` der jeweiligen Instanz zugefügt werden.

Von *Component* erben weitere wichtige Klassen, von denen ich die Klassen *Transform* und *Behaviour* exemplarisch nennen möchte.

Da in Unity häufig alle von *Component* abgeleiteten Klassen als *Components* bezeichnet werden, wird auch im Deutschen oft von „Komponenten“ gesprochen.

### 4.3.5 Transform

Die Klasse *Transform* erbt von *Component* und wird standardmäßig jedem *GameObject* in einer Szene zugefügt. Nur wenn Sie per Code ein *GameObject* erzeugen, besitzt dieses nicht automatisch eine *Transform*-Komponente.

Von *Transform* erbt wiederum die Klasse *RectTransform*, die vom *uGUI*-System bei der GUI-Gestaltung eingesetzt wird.

### 4.3.6 Behaviour

Die Klasse *Behaviour* erbt von der Klasse *Component*. Klassen, die von *Behaviour* erben, werden auch gerne *Behaviours* oder auch allgemein als *Components*/Komponenten bezeichnet und können über die Eigenschaft *enabled* ein- und ausgeschaltet werden.

Beispiele für Klassen, die von *Behaviours* abgeleitet werden, sind *Light* oder auch *NavMeshAgent*, auf die ich noch in den weiteren Kapiteln zu sprechen komme. Von *Behaviour* erbt schließlich auch die Klasse *MonoBehaviour*.

### 4.3.7 MonoBehaviour

Die Klasse *MonoBehaviour* ist die Basisklasse aller Skript-Klassen in Unity, die als Komponenten genutzt werden sollen und einem *GameObject* angehängt werden können.

Auch wenn *MonoBehaviour* von *Behaviour* erbt, müssen Sie beachten, dass durch eine deaktivierte *enabled*-Eigenschaft lediglich das Ausführen der Methoden *Start*, *Awake*, *Update* und *OnGUI* unterbunden wird. Andere Methoden können trotzdem aufgerufen und ausgeführt werden, also auch z. B. die Physik-Methoden *OnTriggerEnter* oder *OnCollisionEnter*.

## ■ 4.4 Skripte erstellen

In Unity beinhaltet jedes Skript genau eine Klasse. Dabei ist der Name des Skriptes immer der gleiche wie der der Klasse. Ansonsten gibt es später einen Fehler, wenn Sie dieses nutzen wollen. Deshalb gilt auch bei der Namenswahl des Skriptes: keine Leer- und Sonderzeichen nutzen. Um ein Skript zu erstellen, wählen Sie eine der folgenden Vorgehensweisen:

- Sie nutzen das **Hauptmenü Assets/Create** und wählen dort den gewünschten Skript-Typ (in unserem Fall **C# Script**).
- Sie nutzen das **Create**-Menü des *Project Browsers* (befindet sich oben), wo Sie ebenfalls die Skript-Typen aufgelistet vorfinden.
- Sie nutzen das **Kontextmenü** des *Project Browsers*, das Sie über die rechte Maustaste erreichen. Auch dort finden Sie ebenfalls über den Menüweg **Create** die Skript-Typen.

- Sie nutzen die *New Script*-Option des *Add Component*-Buttons im **Inspector**. Sobald Sie ein *GameObject* in der *Hierarchy* selektieren, finden Sie diesen unter allen zugefügten *Components* des *GameObjects*. Das Skript wird per Default in der Root des *Project Browsers* erstellt und gleich dem selektierten *GameObject* angehängt.

Durch einen Doppelklick auf das erstellte Skript öffnet sich dieses in der standardmäßig mitgelieferten Entwicklungsumgebung MonoDevelop.

#### 4.4.1 Skripte umbenennen

Sollten Sie nachträglich den Namen eines Skriptes im *Projekt Browser* oder in dem Skript selbst ändern, muss parallel immer das Gegenstück mitgeändert werden. Sollten sich diese unterscheiden, und sei es nur in der Groß- und Kleinschreibung, gibt es später einen Fehler oder Sie können es einfach nicht einem *GameObject* zufügen. Bedienen Sie sich deshalb beim Umbenennen bestenfalls der *Rename*-Funktionalität in MonoDevelop. Diese nennt nicht nur die Klasse, sondern auch das Skript und alle Einbindungen des Skriptes im Projekt um. Sie erreichen diese Funktion über die rechte Maustaste **Refactor/Rename**, sobald Sie den Cursor in den Klassennamen positioniert haben.

Die *Rename*-Funktion macht übrigens auch Sinn, wenn Sie bereits verwandte Variablen umbenennen wollen. Auch in diesem Fall werden alle Stellen, wo Sie diese bereits genutzt haben, umbenannt.



##### Skripte umbenennen

Sollten Sie ein Skript umbenennen, das bereits in einer Szene einem *GameObject* angehängt wurde, dann kann es vorkommen, dass Sie dort nach dem Umbenennen anstatt des Skriptes einen Eintrag „Missing (MonoBehaviour)“ vorfinden. Löschen Sie in so einer Situation diesen Eintrag und fügen Sie Ihr umbenanntes Skript erneut zu. Am einfachsten löschen Sie diesen Eintrag über die Funktion **Remove Component** im Kontextmenü der Komponente (symbolisiert durch das Zahnrad im *Inspector*).

## ■ 4.5 Das Skript-Grundgerüst

Wenn Sie ein neues C#-Skript wie weiter oben beschrieben erstellt haben und nun per Doppelklick öffnen, werden Sie das folgende Grundgerüst sehen:

##### **Listing 4.1** Skript-Grundgerüst

```
using UnityEngine;
using System.Collections;
public class MyScriptName : MonoBehaviour {
```

```
// Use this for initialization
void Start () {

}

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

}
```

In den ersten beiden Zeilen werden mit dem Befehl `using` zwei *Namespaces* eingebunden. Besonders der Namespace *UnityEngine* ist für ein *MonoBehaviour*-Skript zwingend erforderlich und darf nicht entfernt werden. Zusätzliche *Namespaces* können bei Bedarf natürlich oben ergänzt werden.

Nach den *Namespaces* folgt dann die Klassendefinition mit der *MonoBehaviour*-Erbung. Jedes Skript erbt per Default erst einmal von dieser Klasse, die unter anderem auch die `Start`- und `Update`-Methoden zur Verfügung stellt. Sie können diese Methoden bei Bedarf auch löschen. Und auch die Vererbung können Sie ändern bzw. komplett entfernen. Nur sind diese Klassen dann logischerweise auch nicht mehr als Komponenten nutzbar.

## ■ 4.6 Unitys Event-Methoden

Sie wissen nun, wie Sie in Unity ein Skript erstellen. Jetzt bleibt noch die Frage, wann der Code in diesen Skripten eigentlich ausgeführt wird. Die Vorgehensweise von Unity ist hierbei die folgende: *MonoBehaviour* stellt mehrere Methoden zur Verfügung, die durch bestimmte Ereignisse ausgelöst oder auch in regelmäßigen Abständen von Unity aufgerufen werden. Hierbei durchläuft Unity alle *GameObjects* und deren Skript-Instanzen und führt dort die entsprechenden Methoden aus.

Einige wichtige dieser Methoden möchte ich im Folgenden kurz vorstellen. Bedenken Sie hierbei, dass jedes Skript diese Methoden besitzen darf/kann, sie müssen aber nicht implementiert sein. Im Gegenteil, löschen Sie lieber Methoden, wenn diese nicht mit Code gefüllt sind. Denn jede der folgenden Methoden wird entsprechend der Definitionen aufgerufen und kostet dementsprechend Performance, auch wenn sie leer sind.

### 4.6.1 Update

Die wohl wichtigste Methode in Unity ist die Methode `Update`. Hier wird der meiste Code programmiert. `Update` wird das ganze Spiel hindurch aufgerufen, und zwar jedes Mal bevor ein Frame gerendert wird, also bevor ein neues Einzelbild auf dem Monitor gezeichnet wird.

Beachten Sie, dass die Rendering-Zeiten der einzelnen Frames unterschiedlich lang sein können. Wenn Sie in `Update` Code programmieren wollen, der von Frames unabhängig sein soll, sollten die Werte nochmals mit dem Zeitwert `deltaTime` der Klasse `Time` multipliziert

werden. Dieser Wert stellt die Zeit zwischen dem letzten und dem aktuellen Frame dar. Dieses Vorgehen wird z.B. genutzt, wenn Sie einem *GameObject* Bewegungen direkt über die *Transform*-Komponente zufügen. Hierbei wird in jedem Aufruf von *Update* das Objekt ein kleines bisschen weiter vorwärts bewegt. Mehr zum Thema *Transform* erfahren Sie in Kapitel 5, „Objekte in der dritten Dimension“.

**Listing 4.2** Frameunabhängige Bewegung einer Transform-Komponente

```
void Update()
{
    float speed = 20.0f;
    float distance = speed * Time.deltaTime;
    transform.Translate(Vector3.forward * distance);
}
```

## 4.6.2 FixedUpdate

Die *FixedUpdate*-Methode wird in fest definierten Zeitintervallen aufgerufen und sollte genutzt werden, wenn Sie auf *Rigidbody*s zugreifen wollen (*Rigidbody*s sind für das Simulieren von physikalischen Verhalten zuständig). Der Hintergrund ist der, dass die Physik-Engine von Unity nicht framebasiert, sondern in diskreten Zeitintervallen arbeitet, also in festen Zeitabschnitten. Deshalb werden vor jeder Berechnung der Physik in Unity alle *FixedUpdate*-Methoden in den *MonoBehaviour*-Skripten ausgeführt. Mehr zum Thema Physik erfahren Sie im Kapitel „Physik in Unity“.

Das folgende Beispiel fügt dem *Rigidbody* des Objektes in jedem *FixedUpdate*-Aufruf ein Drehmoment zu, damit sich dieses um die Y-Achse dreht.

**Listing 4.3** Zufügen eines Drehmoments

```
void FixedUpdate()
{
    rigidbody.AddTorque(0, 2, 0);
}
```



### Zeitintervall ändern

Das Zeitintervall, in dem *FixedUpdate* aufgerufen wird, beträgt standardmäßig 0,2 Sekunden. Über *Edit/Project Settings/Time/Fixed Timestep* können Sie dies aber ändern.

## 4.6.3 Awake

Diese Methode wird einmalig ausgeführt, und zwar wenn die Skript-Instanz geladen wird. Beim Laden einer Szene werden zunächst alle Objekte in der Szene initialisiert. Danach folgen in einer zufälligen Reihenfolge die Aufrufe der *Awake*-Methoden der einzel-

nen *GameObjects*. Da zunächst alle *GameObjects* initialisiert wurden, ist es bereits möglich, auch andere *GameObjects* zu finden und Variablen zuzuweisen.

**Listing 4.4** Zuweisung eines *GameObjects* in der *Awake*-Methode

```
private GameObject cam;
void Awake()
{
    cam = GameObject.FindGameObjectWithTag("MainCamera");
}
```

#### 4.6.4 Start

Die Methode *Start* verhält sich ähnlich wie *Awake*. Allerdings wird diese Methode im Gegensatz zu *Awake* nur bei aktiven Instanzen ausgeführt. Ist also eine Skript-Instanz zu Anfang deaktiviert und wird erst später auf *enabled* gesetzt, wird auch *Start* erst in dem Moment aufgerufen. Außerdem werden die *Start*-Methoden erst nach dem Ausführen aller *Awake*-Methoden aufgerufen, wodurch es auf einfache Weise möglich ist, die Initialisierungsreihenfolgen zu steuern und bestimmte Ausführungen zu verzögern.

Das folgende Beispiel sucht mit der *FindWithTag*-Methode die Spielerfigur, um deren Skript-Instanz der Klasse *HealthController* einer temporären Variablen zuzuweisen. Hiervon wird im nächsten Schritt der aktuelle Wert von *health* abgefragt und einer eigenen *private*-Variablen zugewiesen.

**Listing 4.5** Ermittlung des Spielers *HealthControllers* in der *Start*-Methode

```
private int playerHealth;
void Start()
{
    HealthController hc = GameObject.FindGameObjectWithTag("Player").
        GetComponent<HealthController>();
    playerHealth = hc.health;
}
```

#### 4.6.5 OnGUI

Die *OnGUI*-Methode gehört zum Unity-eigenen programmierorientierten GUI-System. In dieser Methode werden die Steuerelemente für dieses System programmiert. Da das System aber nicht framebasiert, sondern eventbasiert arbeitet, wobei die Events durch Nutzereingaben oder Unity-interne Rendering-Vorgänge ausgelöst werden, wird auch diese Methode dementsprechend aufgerufen. Aus diesem Grund kann es vorkommen, dass *OnGUI* auch mehrmals pro Frame aufgerufen wird. Mehr zum Thema GUI erfahren Sie im gleichnamigen Kapitel.

**Listing 4.6** Label-Darstellung in der OnGUI-Methode

```
void OnGUI() {  
    GUI.Label(new Rect(0,0,100,30),"Hello World!");  
}
```

### 4.6.6 LateUpdate

Eine weitere Methode, die zwar nicht ganz so häufig verwendet wird, aber dennoch wichtig ist, ist die `LateUpdate`-Methode. Diese wird ausgeführt, nachdem alle `Update`-Methoden aufgerufen wurden, aber noch bevor gerendert wird. Vor allem bei Kamera-Skripten wird diese Methode deshalb gerne genutzt. Wenn eine Kamera ein Zielobjekt verfolgt, das in der `Update`-Methode bewegt wird, kann auch die Kamera in `LateUpdate` neu positioniert werden, bevor schließlich gerendert wird.

Das folgende Beispiel könnte einer Kamera zugefügt werden, damit diese immer in die Richtung eines anderen Objektes gerichtet ist, genauer gesagt in die Richtung dessen *Transform-Component*. Dies könnte beispielsweise für die Steuerung einer Überwachungskamera sein, die zwar an einer Stelle fest installiert ist, aber den Spieler die ganze Zeit im Raum verfolgt.

**Listing 4.7** Kamera-Skript für eine Überwachungskamera

```
public Transform target;  
void LateUpdate () {  
    transform.LookAt(target.position);  
}
```

## ■ 4.7 Komponentenprogrammierung

Ein wichtiger Grundsatz von Unity ist, dass jedes *GameObject* seine eigenen *Components* besitzt. Sie programmieren also keine Lebensverwaltung, die die Lebensstärke aller Gegner verwaltet, sondern ein Skript, das lediglich die Gesundheit eines einzelnen Gegners verwaltet. Dieses Skript wird dann aber wieder jedem Gegner zugefügt, sodass jeder Gegner seine eigene Verwaltung hat.

Dieser Grundsatz kann (und sollte) in bestimmten Situation auch manchmal nicht eingehalten werden. Aber zunächst einmal sollten Sie darauf achten, dass jedes *GameObject* seine eigenen Components besitzt. Dies macht im Übrigen auch im Zugriff auf die Komponenten einen großen Unterschied, da Unity für Zugriffe auf eigene Komponenten häufig Vereinfachungen anbietet, was Sie im Folgenden auch noch sehen werden.

## 4.7.1 Auf GameObjects zugreifen

Das Zugreifen auf das eigene *GameObject* einer Komponente ist sehr einfach. Benutzen Sie hierfür einfach die kleingeschriebene Variable `gameObject`. Hierüber ist es sehr einfach, auf Komponenten des eigenen *GameObjects* zuzugreifen, worauf ich gleich noch zu sprechen komme.

### **Listing 4.8** Zugriff auf eine Komponente des eigenen GameObjects

```
OtherScript otherScript = gameObject.GetComponent<OtherScript>();
```

Um auf andere *GameObjects* zuzugreifen, gibt es mehrere Methoden, die Sie sowohl zur Entwicklungszeit als auch zur Laufzeit nutzen können. Die erste und auch gleichzeitig performance-schonendste Methode ist die *Inspector*-Zuweisung, bei der Sie eine *public*-Variable erstellen, auf die Sie bereits während der Entwicklungszeit das andere *GameObject* per Drag & Drop ziehen können.

### **Listing 4.9** public-Variable für die Inspector-Zuweisung

```
public GameObject player;
void Update()
{
    //Code...
    if(player != null) {
        //Code...
    }
}
```

Aber auch zur Laufzeit können Sie auf andere *GameObjects* zugreifen. Achten Sie hierbei darauf, wenn Sie häufiger auf diese zugreifen wollen, dass Sie diese in Variablen speichern. Dies bringt einen enormen Performancevorteil, da Unity für jeden Zugriff im schlimmsten Fall einmal durch alle *GameObjects* einer Szene laufen muss, um das passende zu finden. Wenn Sie dann auch noch diesen Zugriff in einer *Update*-Methode vornehmen, käme es in jedem Frame zu diesem Vorgang.

### 4.7.1.1 FindWithTag

Sie können ein *GameObject* anhand seines *Tags* finden. Hierfür stellt die Klasse *GameObject* die Methode *FindWithTag* zur Verfügung. Sollten sich hierbei mehrere *GameObjects* mit diesem *Tag* in der Szene befinden, wird zufällig eines ausgewählt.

### **Listing 4.10** GameObject-Zuweisung über den Tag

```
GameObject cam;
void Awake()
{
    cam = GameObject.FindWithTag("MainCamera");
}
```

### 4.7.1.2 FindGameObjectsWithTag

Gibt es mehrere Objekte mit dem gleichen Tag, die Sie aber auch alle in einem Array sammeln möchten, sollten Sie die Methode `FindGameObjectsWithTag` nutzen. Als Rückgabewert erhalten Sie ein `GameObject`-Array aller gefundenen Objekte mit diesem *Tag*.

**Listing 4.11** Alle GameObjects eines Tags einem Array zuweisen

```
GameObject[] enemies;
void Awake()
{
    enemies = GameObject.FindGameObjectsWithTag("Enemy");
}
```

### 4.7.1.3 Find

Sie können ein *GameObject* anhand seines Namens finden. Hierfür bietet sich die statische Methode `Find` der Klasse *GameObject* an.

**Listing 4.12** Suche eines GameObjects anhand des Namens

```
GameObject leftArm;
leftArm = GameObject.Find("Left Arm");
```

Sie können die Suchabfrage auch weiter spezifizieren, indem Sie bei der Übergabe auch die Eltern-Objekte des gesuchten *GameObjects* mit eintragen. Dabei trennen Sie die unterschiedlichen *GameObjects* mit einem Slash-Zeichen `/`. Um darzustellen, dass ein Objekt das letzte ist und selber kein Eltern-Objekt besitzt (Root-Objekt), können Sie dies mit einem vorangestellten Slash darstellen.

**Listing 4.13** Detailliertere GameObject-Suche anhand des Namens

```
leftArm = GameObject.Find("Bob/Left Arm"); //Left Arm hat ein Eltern-Objekt Bob
leftArm = GameObject.Find("/Bob/Left Arm"); //Bob ist das Root-Objekt
```

## 4.7.2 GameObjects aktivieren und deaktivieren

Jedes *GameObject* bietet die Methode `SetActive` an. Über diese können Sie ein komplettes *GameObject* aktivieren und auch deaktivieren. Übergeben Sie der Methode den Zustand, den Sie erreichen möchten.

**Listing 4.14** Deaktivierung eines GameObjects mit `SetActive`

```
void Start() {
    gameObject.SetActive(false);
}
```

Beachten Sie, dass Sie ein deaktiviertes Objekt nur aktivieren können, wenn Sie bereits den Zugriff auf dieses haben (z.B. über eine *Inspector*-Zuweisung). Über die oben vorgestellten *Find*-Methoden haben Sie keine Chance, ein deaktiviertes Objekt zu finden.

### 4.7.3 GameObjects zerstören

Um ein *GameObject* komplett zu zerstören, müssen Sie das Objekt der Methode *Destroy* übergeben.

**Listing 4.15** Spieler anhand des Tags finden und zerstören

```
void DestroyPlayer() {
    GameObject player;
    player = GameObject.FindGameObjectWithTag("Player");
    Destroy(player);
}
```

Sie können der *Destroy*-Methode auch einen *float*-Parameter mitgeben, der die Zerstörung um die jeweiligen Sekunden verzögert.

**Listing 4.16** Verzögerte GameObject-Zerstörung

```
Destroy(player, 2.5f);
```

### 4.7.4 GameObjects erstellen

Meistens werden Sie *GameObjects* über das Hauptmenü oder mit *Prefabs* erstellen (siehe Kapitel „Prefabs“). Aber Sie können natürlich auch zur Laufzeit ganz neue *GameObjects* erstellen. Dies machen Sie, indem Sie auf dem normalen C#-Weg eine neue Instanz der *GameObject*-Klasse erstellen. Hierbei spielt es keine Rolle, ob die Instanz eine lokale oder globale Variable ist oder welchen Zugriffsmodifizierer dieser besitzt. Da jedes *GameObject* per Default erst einmal den Namen „New GameObject“ trägt, können Sie Ihrem *GameObject* zudem auch gleich beim Instanziieren einen Namen mitgeben.

**Listing 4.17** Erzeugung eines neuen GameObjects

```
GameObject go = new GameObject();
GameObject go2 = new GameObject("My Name");
```

### 4.7.5 Auf Components zugreifen

Um auf *Components* zuzugreifen, gibt es mehrere Möglichkeiten. Der einfachste Weg, auf Komponenten anderer *GameObjects* zuzugreifen, ist das Deklarieren einer *public*-Variablen vom Typ des *Components*, auf die Sie das *GameObject* des *Components* während der Entwicklungszeit ziehen. Die Variable referenziert dann nicht auf das *GameObject*, sondern auf die Komponente des *GameObjects*. Besitzt das *GameObject* nicht diese Komponente, so ist der Inhalt der Variablen *null*, also leer.

**Listing 4.18 Öffentliche Komponente verwenden**

```
public Transform player;
void Start()
{
    if (player != null)
        player.position = Vector3.zero;
}
```

**4.7.5.1 GetComponent**

Jedes *GameObject* besitzt die Methode *GetComponent*, mit der Sie auf alle Komponenten des jeweiligen *GameObjects* zugreifen können. Schreiben Sie diesen Aufruf ohne ein vorgestelltes *GameObject*, sucht die Methode beim eigenen *GameObject* nach dieser Komponente.

Das folgende Beispiel sucht sowohl den *HealthController* des Players wie auch den eigenen und weist beiden verschiedenen Variablen zu, um diese dann später zu nutzen.

**Listing 4.19 Komponenten Variablen zuweisen**

```
HealthController playerHc;
HealthController myHc;
void Awake()
{
    GameObject player;
    player = GameObject.FindGameObjectWithTag("Player");
    playerHc = player.GetComponent<HealthController>();
    myHc = GetComponent<HealthController>();
}
void Update ()
{
    if (playerHc.health > 0)
    {
        //Code...
    }
}
```

**Zugriffe reduzieren**

Achten Sie darauf, die *GetComponent*-Zugriffe so weit wie möglich zu reduzieren.

Bei häufigeren Zugriffen speichern Sie die Komponenten stattdessen in Variablen.

**4.7.5.2 SendMessage**

Eine sehr interessante Möglichkeit, um eine Methode aufzurufen, ist die Methode *SendMessage*. *SendMessage* wird von der *GameObject*-Klasse zur Verfügung gestellt und durchläuft alle *MonoBehaviour*-Skripte des *GameObjects* und ruft dabei jede Methode mit diesem Namen auf. Sie brauchen also nicht einmal zu wissen, in welchem Skript sich diese befindet. Optional können Sie *SendMessage* auch einen Übergabeparameter mitgeben sowie einen Optionsparameter vom Typ *SendMessageOptions*. Letzterer legt fest, ob eine Empfänger-Methode zwingend notwendig ist oder nicht. Sollte eine notwendig sein, aber es wird keine gefunden, wird ein Fehler ausgelöst.

Das folgende Beispiel-Skript könnte einer beliebigen Waffe oder Falle mit einem *Collider* angehängt werden, um bei Kontakt Schaden zu verursachen. Dabei versucht das Skript, eine Methode mit dem Namen *ApplyDamage* im anderen *GameObject* aufzurufen und dieser den Schadenswert 1 zu übertragen. Hierfür wird auf das *GameObject* des übergebenen *Collision*-Parameters zugegriffen und *SendMessage* aufgerufen.

#### **Listing 4.20** Schaden zufügen mittels *SendMessage*

```
using UnityEngine;
using System.Collections;
public class Damage : MonoBehaviour {
    public float amount = 1;
    void OnCollisionEnter(Collision collision) {
        collision.gameObject.SendMessage (
            "ApplyDamage", amount, SendMessageOptions.DontRequireReceiver);
    }
}
```

Der Parameter *DontRequireReceiver* besagt, dass kein Empfänger notwendig ist. Es wird also kein Fehler ausgelöst, wenn das Zielobjekt kein Skript mit dieser Methode besitzt. Mehr zum Thema Kollisionen erfahren Sie im Kapitel „Physik in Unity“.

#### **4.7.5.3 Variablenzugriff**

Möchten Sie auf Komponenten des eigenen *GameObjects* zugreifen, bietet Unity für einige den Zugriff über Variablen an, was den Programmieraufwand erheblich vereinfacht. Hierzu gehört beispielsweise die  *AudioSource*-Komponente, die *Rigidbody*-Komponente oder auch die  *Transform*-Komponente.

#### **Listing 4.21** Komponentenzugriff über *GameObject*-Variablen

```
audio.Play();
rigidbody.AddForce(0, 1, 0);
transform.Translate(0, 10, 0);
```



#### **API-Änderungen in Unity5**

Unity Technologies plant für Unity5 einige Änderungen im Scripting-Bereich vorzunehmen. So soll u.a. der Variablenzugriff eingeschränkt werden. Diese Zugriffe müssen dann durch  *GetComponent*-Befehle ersetzt werden. Unity plant hierzu eine automatische Skript-Konvertierung, die durchgeführt werden kann, wenn ein älteres Projekt erstmalig mit Unity5 geöffnet wird.

#### **4.7.6 Components hinzufügen**

Möchten Sie zur Laufzeit einem *GameObject* eine neue Komponente hinzufügen, können Sie hierfür die Methode *AddComponent* nutzen, die jedes *GameObject* anbietet. Das folgende Beispiel fügt dem eigenen *GameObject* eine Skript-Komponente *HealthController* sowie ein *Rigidbody* zu.

**Listing 4.22** Komponenten zufügen

```
gameObject.AddComponent<HealthController>();
gameObject.AddComponent<Rigidbody>();
```

**4.7.7 Components entfernen**

Um eine Komponente von einem *GameObject* zu entfernen, nutzen Sie wie beim Zerstören eines *GameObjects* die Methode *Destroy*. Wenn Sie das eigene Skript zerstören wollen, nutzen Sie das Schlüsselwort *this*. Auch hier können Sie über einen zweiten Parameter zeitliche Verzögerungen einbauen.

**Listing 4.23** Komponenten entfernen

```
Destroy(rigidbody); //das eigene Rigidbody zerstoeren
Destroy(this, 2); // die eigene Skript-Instanz mit 2 sec. Verzoegegerung zerstoeren
```

**4.7.8 Components aktivieren und deaktivieren**

Anstatt eine Komponente komplett gleich zu entfernen, kann es auch sinnvoll sein, sie nur zu deaktivieren. Der Vorteil ist der, dass Sie sie später auch wieder aktivieren können. Ein klassisches Beispiel ist hierfür das Licht. Das folgende Beispiel greift auf die *Light*-Komponente des eigenen *GameObjects* zu und nutzt den logischen Negationsoperator (das Ausrufezeichen), um den Zustand von *enabled* umzukehren. Der Effekt ist der, dass eingeschaltetes Licht ausgeschaltet und ausgeschaltetes Licht eingeschaltet wird.

**Listing 4.24** Lichtschalter

```
void Switch() {
    light.enabled = !light.enabled;
}
```

**■ 4.8 Zufallswerte**

Zufallswerte sind ein wichtiges Element, um Spiele interessant zu machen. Die Anwendungsmöglichkeiten reichen dabei vom Zuweisen eines zufälligen Parameterwertes wie der Stärke über das Generieren und Platzieren von *GameObjects* bis hin zum Erstellen ganzer Levels. Hierfür bietet Unity die Klasse *Random* an, die mehrere interessante Methoden zum Generieren von Zufallswerten anbietet. Die wohl gängigste ist die Methode *Range*. Beachten Sie hierbei, dass es diese in zwei unterschiedlichen Varianten gibt.

- **Range für float-Werte** generiert einen Zufallswert, der zwischen dem Start- und dem Endwert liegt. Dabei können auch die Start- und die Endwerte als Ergebnisse zurückgegeben werden.
- **Range für int-Werte** generiert einen Zufallswert, der vom Startwert reicht, aber kleiner (!) als der Endwert sein muss.

**Listing 4.25** Erzeugen einer zufälligen Position auf der X-Achse

```
using UnityEngine;
using System.Collections;

public class RandomXPosition : MonoBehaviour {

    void Start () {
        float xPos = Random.Range(0.0F, 5.0F);
        Vector3 pos = new Vector3(xPos, 10,0);
        transform.position = pos;
    }
}
```

Die Klasse bietet noch einige weitere Varianten der Zufallsberechnung. So erreichen Sie über die statische Variable `rotation` eine zufällige Drehung vom Typ `Quaternion` oder mit Hilfe von `insideUnitSphere` bekommen Sie einen Zufallswert im `Vector3`-Format, bei dem x, y und z jeweils einen zufälligen Wert zwischen -1 und 1 besitzen.

**Listing 4.26** Generierung von Zufallsposition und -rotation

```
using UnityEngine;
using System.Collections;
public class Randomize : MonoBehaviour {
    public float radius = 3;
    void Start () {
        transform.position = Random.insideUnitSphere * radius;
        transform.rotation = Random.rotation;
    }
}
```

## ■ 4.9 Parallel Code ausführen

Mit *Coroutines* haben Sie die Möglichkeit, Methoden auszuführen, die parallel zu den eigentlichen `Update`- und `LateUpdate`-Aufrufen ausgeführt werden. Während normaler Code, der sich in der `Update`-Methode befindet, in jedem Frame ausgeführt wird, können Sie mit *Coroutines* Codeblöcke programmieren, die sich über beliebig viele Frames erstrecken.

**Listing 4.27** Ein Coroutine

```
string message = "";
IEnumerator Countdown()
{
    yield return new WaitForSeconds(1);
```

```

    message ="3";
    yield return new WaitForSeconds(1);
    message ="2";
    yield return new WaitForSeconds(1);
    message ="1";
    yield return new WaitForSeconds(1);
    message ="Go!";
}

```

Das Beispiel in Listing 4.27 stellt einen Countdown dar, der vier Mal eine Sekunde wartet und jeweils danach der Variablen `message` einen neuen Inhalt zuweist (die dann z.B. in der GUI visualisiert wird). Um eine solche *Coroutine* nun zu starten, nutzen Sie den Befehl `StartCoroutine`.

#### **Listing 4.28** Start einer Coroutine

```

void Start () {
    StartCoroutine(Countdown ());
}

```

Die wichtigen Schlüsselwörter einer *Coroutine* sind zum einen die Schnittstellendefinition `IEnumerator`, die das Durchlaufen der Methode über mehrere Frames ermöglicht, sowie der Befehl `yield return` (siehe Listing 4.27). Letzterer speichert den aktuellen Stand der *Routine*, um beim nächsten Frame an der gleichen Stelle weitermachen zu können. Hierfür geben Sie `yield return null` an.

Die folgende *Coroutine* durchläuft zehn Frames und schreibt in jedem Frame die aktuelle Spielzeit in die Konsole.

#### **Listing 4.29** Coroutine über zehn Frames

```

IEnumerator FrameTest()
{
    for (int i = 0; i < 10; i++)
    {
        Debug.Log (Time.time);
        yield return null;
    }
}

```

Die oben eingesetzte Methode `WaitForSeconds` verlängert hierbei das Aussetzen des Codes, sodass nicht gleich im nächsten Frame weitergemacht wird, sondern erst nach einer Sekunde.

### 4.9.1 WaitForSeconds

Mithilfe von `yield return WaitForSeconds(2.5f)` können Sie die Ausführung einer *Coroutine* für eine bestimmte Zeit unterbrechen. Der übergebene `float`-Parameter gibt hierbei die Wartezeit an. Beachten Sie dabei, dass eine Unterbrechung mit `WaitForSeconds` nur so genau sein kann, wie die Frames-Dauer ist. Wenn also jeder Frame 0,02 Sekunden dauert, kann auch die Genauigkeit von `WaitForSeconds` nicht besser sein.

## ■ 4.10 Verzögerte und wiederholende Funktionsaufrufe mit Invoke

Unity stellt Ihnen Möglichkeiten zur Verfügung, damit Sie innerhalb einer *MonoBehaviour*-Klasse andere Methoden zeitverzögert und auch wiederholend aufrufen können. Hierfür bietet Unity die sogenannten *Invoke*-Befehle an.

### 4.10.1 Invoke

Die Methode *Invoke* erwartet den Namen einer Methode, die gestartet werden soll, sowie einen Sekundenwert, der die Verzögerung angibt. Der Code läuft nach dem Ausführen des Befehls ganz normal weiter. Erst nach Erreichen der Zeitangabe wird die angegebene Methode ausgelöst. Die angegebene Methode muss sich dabei im selben Skript befinden. Ob eine Methode bereits via *Invoke* verzögert aufgerufen wird, können Sie über *IsInvoking* erfahren (mehr dazu gleich).

Das folgende Beispiel führt die Methode *DelayedMessage* mit 2,5 Sekunden Verzögerung nach Ausführung der *Start*-Methode aus. Diese weist einer Variablen einen Begrüßungstext zu, dessen Inhalt wiederum in der GUI angezeigt werden könnte.

**Listing 4.30** Verzögter Aufruf einer Methode

```
public string welcomeMessage = "";
void Start () {
    Invoke ("DelayedMessage",2.5F);
}
void DelayedMessage()
{
    welcomeMessage = "Seid willkommen, Meister!";
}
```

### 4.10.2 InvokeRepeating, IsInvoking und CancelInvoke

*InvokeRepeat* arbeitet ähnlich wie *Invoke*, nur dass diese Methode zusätzlich die angegebene Methode nicht nur einmal aufruft, sondern beliebig häufig. Das Intervall der Wiederholung übergeben Sie als dritten Parameter. Der zweite bestimmt weiterhin die Verzögerung der ersten Ausführung.

Mit der Methode *IsInvoking* können Sie zusätzlich überprüfen, ob eine Methode bereits über einen *Invoke*-Befehl initiiert wurde. Diese Ausführungen können Sie dann mit *CancelInvoke* abbrechen. Dabei übergeben Sie den Namen der Methode, die Sie abbrechen möchten. Übergeben Sie keine, werden alle Methoden abgebrochen, die Sie mit einem *Invoke*-Befehl gestartet haben.

Das folgende Beispiel könnte von einem *Spawner*-Skript stammen, das drei Sekunden nach dem Spielstart jede Sekunde ein neues Objekt an einer zufälligen Position erzeugt.

**Listing 4.31** Spawner-Skript

```

public GameObject go;
public bool isGameOver = false;
void Start ()
{
    //Startet die verzögerte Wiederholung von RandomInstantiation
    InvokeRepeating("RandomInstantiation",3,1);
}

void Update ()
{
    //Überprüft, ob RandomInstantiation noch ausgeführt wird
    if (IsInvoking ("RandomInstantiation"))
    {
        //Wenn das Spiel zu Ende ist,
        //soll RandomInstantiation abgebrochen werden
        if (isGameOver)
            CancelInvoke("RandomInstantiation");
    }
}

void RandomInstantiation()
{
    //Position, wo das Prefab instanziert werden soll
    Vector3 pos = new Vector3();
    pos = Random.insideUnitSphere * 10; //Zufallswerte für alle Achsen
    pos.y = 0; // Y wird auf 0 gesetzt. X und Z liegen zwischen -10 und 10
    //Prefab instanzieren
    Instantiate (go,pos,Quaternion.identity);
}

```

## ■ 4.11 Daten speichern und laden

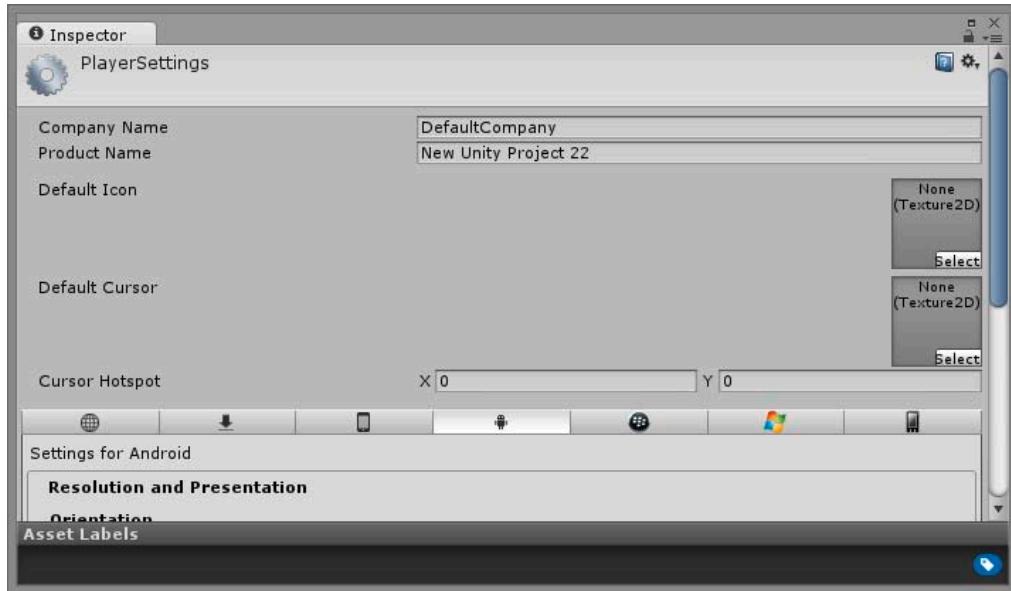
Unity liefert Ihnen mit der Klasse `PlayerPrefs` eine tolle Möglichkeit, Werte unabhängig vom System zu speichern. Egal ob Windows, Mac OSX, der Webplayer oder Android, Sie übergeben einfach den `PlayerPrefs`-Methoden die zu speichernden Werte. Wo diese am Ende tatsächlich landen, darum kümmert sich Unity. In dem folgenden Abschnitt kommt hierzu noch ein kleines Anwendungsbeispiel.

### 4.11.1 PlayerPrefs-Voreinstellungen

Auch wenn sich die Pfade auf allen Plattformen unterscheiden, so nutzen sie doch alle die in Unity hinterlegten Bezeichnungen für *Company Name* und *Product Name*.

Auf Windows-Systemen wird beispielsweise bei Stand-alone-Anwendungen der Registry-Pfad `HKCU\Software\[company name]\[product name]` genommen, Webplayer-Anwendungen schreiben auf Windows-Systemen in das Verzeichnis `%APPDATA%\Unity\WebPlayerPrefs`.

*Company Name* und *Product Name* ändern Sie in den *Player Settings* (*Edit/Project Settings/Player*). Standardmäßig steht *Company Name* auf „DefaultCompany“ und *Product Name* auf dem Namen Ihres Projektes.



**Bild 4.4** Company Name und Product Name in den Player Settings

### 4.11.2 Daten speichern

PlayerPrefs stellt Ihnen drei statische Methoden zur Verfügung, um Werte als *int*, *float* oder *string* zu speichern. Alle drei Methoden erwarten zwei Parameter. Dem ersten Parameter übergeben Sie einen String, der einen eindeutigen Identifizierer darstellt, also den Key (die ID bzw. der Name des Wertes). Der zweite Parameter ist dann der tatsächliche Wert in dem jeweiligen Datentyp.

**Listing 4.32** Daten speichern mit PlayerPrefs

```
PlayerPrefs.SetInt("Lifepoints", 4);
PlayerPrefs.SetFloat("Speed", 2.5F);
string playerName = "Carsten";
PlayerPrefs.SetString ("Name", playerName);
```

### 4.11.3 Daten laden

Zum Laden der Daten stehen Ihnen ebenfalls drei statische Methoden zur Verfügung. Auch hier müssen Sie den Key übergeben.

**Listing 4.33** Daten laden mit PlayerPrefs

```
int myLifepoints = PlayerPrefs.GetInt("Lifepoints");
float speed = PlayerPrefs.GetFloat("Speed");
string playerName = PlayerPrefs.GetString ("Name");
```

**4.11.4 Key überprüfen**

Bevor Sie Daten laden, also auf einen Key zugreifen, kann es häufig sinnvoll sein, zunächst zu überprüfen, ob dieser überhaupt existiert. Auch wenn Unity hier zwar keinen Fehler verursacht, kann es ja trotzdem sinnvoll sein, in dem Fall die Variable mit Standardwerten zu belegen. Diese Überprüfung machen Sie mit dem Befehl HasKey.

**Listing 4.34** HasKey der PlayerPrefs-Klasse

```
if (!PlayerPrefs.HasKey("Lifepoints"))
    PlayerPrefs.SetInt("Lifepoints",5);
```

**4.11.5 Löschen**

Sie können aber natürlich nicht nur neue Einträge erzeugen, Sie können sie auch wieder löschen. Hierbei können Sie die statische Methode DeleteKey nutzen. Um alle gespeicherten Werte auf einmal zu löschen, können Sie DeleteAll nutzen.

**Listing 4.35** Lösch-Methoden von PlayerPrefs

```
PlayerPrefs.DeleteKey ("Lifepoints");
PlayerPrefs.DeleteAll();
```

**4.11.6 Save**

Bei einigen Plattformen, wie zum Beispiel dem Webplayer, schreibt Unity die Werte nicht gleich beim Aufrufen der Methoden SetInt, SetFloat und SetString auf die Festplatte. Stattdessen speichert Unity diese zunächst einmal temporär zwischen. Erst beim Beenden der Anwendung, beim Webplayer wäre dies das ordnungsgemäße Schließen des Tabs oder des Browsers, werden diese Werte erst auf die Festplatte geschrieben.

Da sich Unity im Hintergrund um alles kümmert, macht dieses Verhalten für Sie zunächst keinen Unterschied. Nur in einem Fehlerfall, wenn z. B. der Browser abstürzt, kann es eben sein, dass die Werte nicht mehr rechtzeitig zurückgeschrieben werden können. Hierfür stellt Unity nun den Befehl Save zur Verfügung, mit dem Sie das tatsächliche Speichern erzwingen können.

**Listing 4.36** Speichern der PlayerPrefs erzwingen

```
PlayerPrefs.Save ();
```

## ■ 4.12 Szeneübergreifende Daten

*GameObjects* existieren eigentlich nur innerhalb einer Szene. Wird eine neue Szene gestartet, werden alle *GameObjects* der vorherigen Szene zerstört.

Das Laden einer neuen Szene machen Sie dabei mit der Methode `LoadLevel` der `Application`-Klasse. Dieser übergeben Sie entweder den Namen oder den *Level Index* der jeweiligen Szene. Diese definieren Sie in den *Build Settings*, die Sie über `File/Build Settings` erreichen. Mehr zu diesem Thema erfahren Sie im Kapitel „Spiele erstellen und publizieren“.

**Listing 4.37** Szene mit dem Level Index 1 starten

```
Application.LoadLevel(1);
```

Wenn nun aber alle *GameObjects* zerstört werden, hat das natürlich den Nachteil, dass Daten wie z.B. Erfahrungspunkte, Lebensstärke usw. bei einem Szenewechsel verloren gehen. Schließlich werden diese ja in Skripten gespeichert, die wiederum an einem *GameObject* hängen. Für dieses Dilemma gibt es aber natürlich Lösungen, sogar mehrere. Von diesen möchte ich Ihnen zwei Ansätze vorstellen.

### 4.12.1 Werteübergabe mit PlayerPrefs

Den ersten Ansatz haben Sie bereits kennengelernt: `PlayerPrefs`. Sie können einfach alle Werte, die Sie übergeben möchten, beim Beenden einer Szene mit `PlayerPrefs` abspeichern und bei Beginn einer neuen Szene diese einfach laden und den Variablen wieder zuweisen.

Hierfür wäre eine Möglichkeit das Abspeichern der Werte in der `OnDestroy`-Methode, die jedes `MonoBehaviour`-Skript bereitstellt. Wenn dann die Objekte und Komponenten beim Wechsel in eine neue Szene zerstört werden, werden hier die Daten eines Skriptes weggeschrieben. In der `Start`- oder `Awake`-Methode können sie dann wieder eingelesen werden.

Hierzu ein kleines Beispiel: Das folgende Skript besitzt eine Variable `health`, deren aktueller Wert beim Beenden mit den `PlayerPrefs`-Methoden gespeichert wird. Wenn danach eine neue Szene gestartet wird, liest das Skript diesen Wert in der `Start`-Methode wieder aus den `PlayerPrefs` in die Variable zurück. Weil der `Key` beim allerersten Aufruf des Skriptes noch nicht existieren wird, wird zur Fehlervermeidung vor dem Einlesen noch überprüft, ob dieser `Key` überhaupt existiert.

**Listing 4.38** Methodik zum Übergeben einer Variablen an eine neue Szene

```
using UnityEngine;
using System.Collections;
public class HealthValue : MonoBehaviour {
    public int health = 2;
    void Start () {
        if (PlayerPrefs.HasKey("Health"))
            health = PlayerPrefs.GetInt("Health");
    }
}
```

```

void OnDestroy () {
    PlayerPrefs.SetInt("Health", health);
}
}

```

### 4.12.1.1 Startmenüs zur Initialisierung nutzen

Beim Einsatz des obigen Skriptes sind einige Punkte zu beachten. Wenn das Spiel zum Beispiel beendet und neu gestartet wird, wird der Variablen immer der Wert des letzten Spiels zugewiesen. Beim Fortführen eines alten Spiels kann das vielleicht erwünscht sein, bei einem Neustart des Spiels aber sicher nicht.

Damit dies nicht geschieht, gibt es eine einfache Lösung: Sie können in einer vorgelagerten Szene, z.B. einer Startmenü-Szene, die *Keys* mit den Initialwerten vorbelegen. Dadurch können Sie das obige Skript sogar ohne die *HasKey*-Abfrage nutzen. Außerdem haben Sie darüber die Möglichkeit, im Startmenü den Spieler zu fragen, ob dieser ein altes Spiel fortführen möchte oder ein neues starten will. Ein solches Skript könnte dann wie folgt aussehen:

**Listing 4.39** Einfaches Startmenü

```

using UnityEngine;
using System.Collections;
public class InitValues : MonoBehaviour {
    public int health = 2;
    void OnGUI()
    {
        if(GUILayout.Button ("New"))
        {
            PlayerPrefs.SetInt("Health", health);
            Application.LoadLevel(1);
        }

        if (PlayerPrefs.HasKey("Health"))
        {
            if(GUILayout.Button("Continue"))
            {
                Application.LoadLevel(1);
            }
        }
    }
}

```

Das obige Skript könnte beispielsweise in der Szene mit dem Index 0 genutzt werden, wobei das vorherige *HealthValue*-Skript dementsprechend in einer Szene mit dem Index 1 eingesetzt wird.

### 4.12.2 Zerstörung unterbinden

Die obige Vorgehensweise hat einige Nachteile. So kann sie recht aufwendig werden, wenn man alle Parameter einer anpassbaren GUI, eines Spielers oder eines Inventars abspeichern möchte. Aber spätestens wenn Sie Texturen und Materialien von einer zur anderen Szene

übergeben wollen, haben Sie hier ein Problem. Denn Texturen können Sie nicht mit den *PlayerPrefs* abspeichern.

Hierfür besitzt die übergeordnete *Object*-Klasse die statische Methode *DontDestroyOnLoad*, die das Zerstören eines beliebigen Objektes beim Ladevorgang unterbindet. Da der Übergabeparameter ebenfalls vom Typ *Object* ist, können Sie dieser ein beliebiges Objekt übergeben (siehe „Unitys Vererbungsstruktur“), auch wenn für gewöhnlich hier ein *GameObject* übergeben wird.

Das folgende Beispiel-Skript verhindert das Zerstören des *GameObjects*, dem das folgende Skript angehängt wird, und all seiner Komponenten. Wird die Szene beendet und eine neue gestartet, wird das gesamte *GameObject* in die nächste Szene übernommen. Dadurch bleibt natürlich auch der Wert der Variablen *lifePoints* erhalten.

#### **Listing 4.40** DontDestroyOnLoad-Beispiel

```
using UnityEngine;
using System.Collections;
public class LifePointController : MonoBehaviour {
    public int lifePoints = 0;
    void Awake() {
        DontDestroyOnLoad(gameObject);
    }
    //Code...
}
```

Beachten Sie hierbei, dass beim Start der nächsten Szene die *Awake*- wie auch die *Start*-Methode nicht erneut ausgeführt werden. Sollten Sie diese nutzen, können Sie auch die Methode *OnLevelWasLoaded* anwenden, die auch den aktuellen *Level Index* übergeben bekommt.

#### **Listing 4.41** OnLevelWasLoaded

```
void OnLevelWasLoaded (int level){
    //Code...
}
```

### **4.12.2.1 DontDestroyOnLoad als Singleton**

Mit *DontDestroyOnLoad* verhindern Sie, dass das übergebene *Object* beim Szenenwechsel zerstört wird. Wenn Sie allerdings gar nicht in eine neue Szene wechseln, sondern vielleicht einfach die gleiche Szene erneut starten (weil z.B. der Spieler gestorben ist), existiert das *Object* auf einmal doppelt.

Um dieses Problem zu lösen, kann das sogenannte *Singleton*-Entwurfsmuster genutzt werden. Dies sorgt dafür, dass es nur eine Instanz einer Klasse geben darf. Hierfür wird eine statische Variable genutzt, die die erste Instanz, die von der Klasse erzeugt wird, speichert. Ist die Variable bereits mit einer anderen Instanz belegt, wird diese einfach zerstört (in unserem Fall wollen wir dafür sorgen, dass gleich das gesamte *GameObject* zerstört wird).

Zusätzlich wird in Unity häufig entweder die Variable selbst oder eine Eigenschaft öffentlich bereitgestellt, um von außen einfacher auf diese Instanz zuzugreifen. Das obige *MonoBehaviour*-Skript könnte als *Singleton* dann wie folgt aussehen:

**Listing 4.42** DontDestroyOnLoad-Singleton

```
using UnityEngine;
using System.Collections;
public class LifePointController : MonoBehaviour {
    public int lifePoints = 0;
    //Statische Variable vom Typ der eigenen Klasse
    private static LifePointController instance;
    //Statische Eigenschaft fuer den einfachen Zugriff auf die Instanzen.
    public static LifePointController Instance {
        get{
            return instance;
        }
    }
    void Awake() {
        //Ist die statische Variable noch nicht gefuellt?
        if (instance == null) {
            //Weise diese Instanz der Variablen instance zu.
            instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else {
            //Zerstoere dieses GameObject
            Destroy(gameObject);
        }
    }
}
```

## ■ 4.13 Debug-Klasse

Unity besitzt das Fenster *Console* (Konsole), in dem Fehlermeldung und Warnungen von Unity angezeigt werden. Unity bietet hier die Möglichkeit, selber Hinweistexte und Ähnliches dort auszugeben. Die Klasse Debug bietet hierfür einige hilfreiche Funktionen an, wovon die Hauptfunktion die Methode Log ist. Diese gibt einen beliebigen Text in der Konsole aus.

**Listing 4.43** Ausgabe mit Debug.Log

```
void Start() {
    Debug.Log("Testausgabe");
}
```

Neben der normalen Log-Methode gibt es noch die Varianten LogWarning und.LogError, die die Meldungen als Warn- und als Fehlermeldungen ausgeben.

Aber die Debug-Klasse bietet noch weitere Funktionen für das Debuggen an. So können Sie mithilfe der Klasse auch Linien in der *Scene View* zeichnen (DrawLine und DrawRay) oder mithilfe der Methode Break den Editor anhalten. Mehr zu dieser Klasse erfahren Sie in der Scripting Reference, die mit Unity mitgeliefert wird.

**Listing 4.44** Zeichnen einer Linie mithilfe der Debug-Klasse

```
void Update() {
    Debug.DrawLine(Vector3.zero, new Vector3(2, 0, 0), Color.red);
}
```

## ■ 4.14 Kompilierungsreihenfolge

Eine Besonderheit von Unity ist die Skript-Kompilierungsreihenfolge, die sich in vier unterschiedliche Phasen aufteilt. Das ist deshalb wichtig, weil Sie von einem Skript nur auf andere zugreifen können, die sich in der gleichen oder in einer früheren Phase befinden.

Bleibt die Frage, wie festgelegt wird, welches Skript in welcher Phase kompiliert wird. Dies wird in Unity über die Ordnernamen gesteuert, in denen Sie Ihre Skripte im *Project Browser* ablegen. Eigentlich spielen die Namen keine Rolle, einige davon sind aber für besondere Zwecke reserviert.

Ein Beispiel für solche reservierten Namen lautet *Editor*. In Ordnern mit diesem Namen sollten nur Skripte gespeichert werden, die auch von der Klasse *Editor*, nicht von , erben. Mit diesen können Sie Unity beispielsweise um eigene Funktionalitäten erweitern.

Unity geht bei der Kompilierung in vier verschiedenen Schritten vor.

1. Es werden alle Skripte kompiliert, die sich in Ordnern mit den Namen *Standard Assets*, *Pro Standard Assets* und *Plugins* befinden.
2. Es werden alle *Editor*-Skripte, die sich in Ordnern namens *Editor* und in den obigen Ordnern *Standard Assets*, *Pro Standard Assets* und *Plugins* befinden, kompiliert.
3. Alle anderen Skripte, die sich außerhalb von Ordnern mit dem Namen *Editor* befinden, werden kompiliert.
4. Zum Schluss werden alle übrig gebliebenen Skripte (in Ordnern namens *Editor*) kompiliert.

Weiter gibt es noch den reservierten Ordnernamen *WebPlayerTemplates*, dessen enthaltene Skripte gar nicht kompiliert werden.

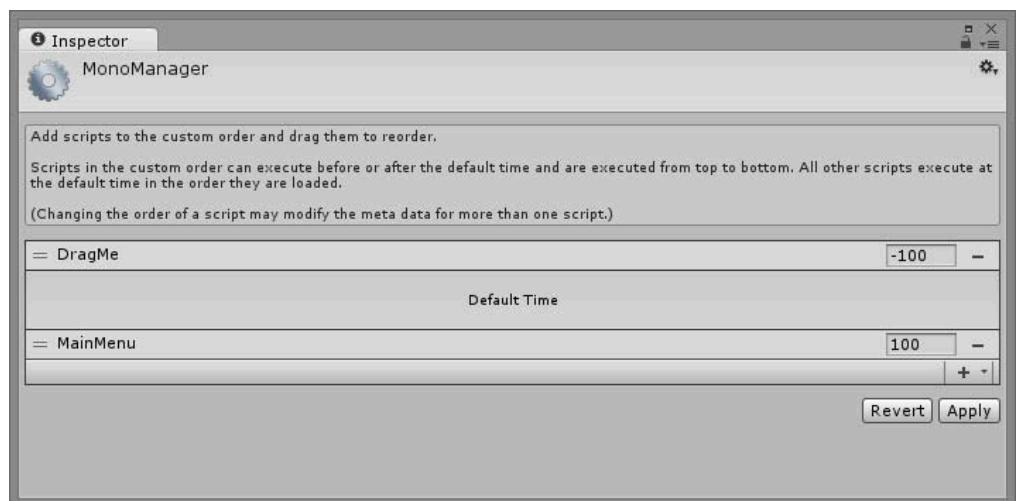
### 4.14.1 Programmsprachen mischen und der sprachübergreifende Zugriff

Möchten Sie mit Ihrem C#-Skript auf eine Boo- oder eine JavaScript-Klasse zugreifen, muss das Skript, auf das Sie zugreifen wollen, in einer früheren Phase kompiliert worden sein als es selbst. In diesem Fall könnten Sie das JS-Skript in den *Plugins*-Ordner verschieben und schon ist es von Ihrem C#-Skript aus zugänglich.

Hierdurch ergibt sich aber ein Problem: Sie können immer nur in eine Richtung zugreifen, es ist nicht möglich, in beide Richtungen zuzugreifen. Dies sollten Sie bedenken, wenn Sie mit mehreren Programmiersprachen arbeiten.

## ■ 4.15 Ausführungsreihenfolge

Manchmal ist es wichtig sicherzustellen, dass ein Skript vor dem anderen ausgeführt wird. Bei zwei Skripten können Sie sich noch häufig mit Update und LateUpdate behelfen, bei drei Skripten geht das aber schon nicht mehr. Hierfür gibt es die *Script Execution Order Settings*, die Sie unter **Edit/Project Settings/Script Execution Order** finden.



**Bild 4.5** Script Execution Order

Durch einen Klick auf das Pluszeichen können Sie ein Skript hinzufügen. Danach können Sie das Skript dort belassen oder nach oben ziehen und den Zeitwert ändern. Die Skripte werden dann von oben nach unten abgearbeitet. Alle hier nicht definierten Skripte werden im Bereich *Default Time* in einer zufälligen Reihenfolge aufgerufen.

Lizenziert für gregwyss@hotmail.com.

© 2014 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

# 5

# Objekte in der dritten Dimension

In der *Scene View* von Unity können Sie dreidimensionale Objekte im 3D-Raum platzieren. Sie können die Objekte dort verschieben, drehen und auch skalieren. Auf diese Weise können Sie Ihre eigenen 3D-Welten gestalten, die der Spieler dann später erkunden kann. In diesem Kapitel werden wir auf einige Grundlagen des dreidimensionalen Raums und der Darstellung von 3D-Objekten eingehen.

## ■ 5.1 Das 3D-Koordinatensystem

Zum Darstellen von Objekten im dreidimensionalen Raum nutzt Unity ein sogenanntes linkshändiges Koordinatensystem mit den Achsen X, Y und Z. Der Begriff kommt von einer Merkregel, die helfen soll, sich die Richtungen der Achsen besser zu merken. Bei dieser stellen drei Finger der linken Hand jeweils eine Achse dar:

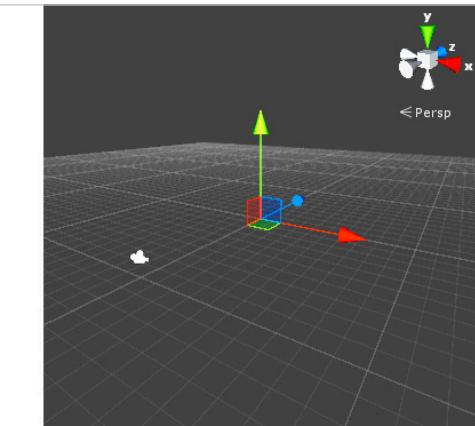
- **Daumen** stellt die X-Achse dar
- **Zeigefinger** stellt die Y-Achse dar
- **Mittelfinger** stellt die Z-Achse dar

Zeigen Sie nun mit dem Daumen nach rechts (x) und dem Zeigefinger nach oben (y). Wenn Sie nun den Mittelfinger abspreizen, zeigt dieser in den Raum, was der Richtung der Z-Achse in Unity entspricht (siehe Bild 5.1).



### Koordinatensystem bei 2D-Spielen in Unity

Auch wenn Sie mit Unity 2D-Spiele erstellen, arbeiten Sie immer in einem 3D-Koordinatensystem, das neben der X- und Y-Achse auch eine Z-Achse für die Tiefe besitzt.



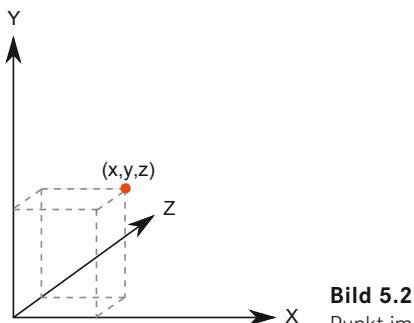
**Bild 5.1** Merkregel zum linkshändischen Koordinatensystem

## ■ 5.2 Vektoren

Die kleinste Einheit in einem Raum ist der Punkt. Um einen Punkt im dreidimensionalen Raum zu beschreiben, werden Vektoren genutzt. Unity stellt hierfür den Typ *Vector3* zur Verfügung, der für die einzelnen Koordinaten die Parameter x, y und z besitzt. Im Folgenden werde ich für Vektoren die Schreibweise (x,y,z) nutzen.

**Listing 5.1** Zwei Möglichkeiten, Vektoren Koordinaten zuzuweisen

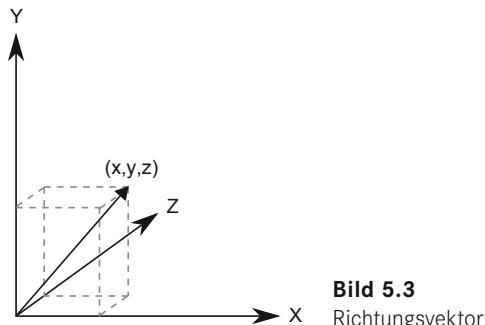
```
//Vektor 1
Vector3 point1 = new Vector3();
point1.x = 2;
point1.y = 4;
point1.z = 0;
//Vektor 2
Vector3 point2 = new Vector3(2,4,0);
```



**Bild 5.2**  
Punkt im dreidimensionalen Raum

## 5.2.1 Ort, Winkel und Länge

Vektoren können, wie bereits erwähnt, dazu genutzt werden, Ortsangaben zu beschreiben (**Ortsvektoren** genannt). So wird mit der Angabe (2, 4, 0) die Position eines Punktes beschrieben. Genauso können sie aber auch genutzt werden, um eine Richtung mit einem Betrag darzustellen (**Richtungsvektor**). Als Betrag bezeichnet man hierbei die Länge des Vektors, sprich die Entfernung vom Ursprung (0, 0, 0) bis zur Koordinate des Vektors.



**Bild 5.3**

Richtungsvektor

Während Sie den Winkel über Winkelfunktionen bestimmen können, wird der Betrag über den Satz des Pythagoras berechnet. Sie brauchen aber glücklicherweise den Betrag nicht selber zu berechnen, hierfür stellt der *Vector3*-Typ die Eigenschaft `magnitude` bereit. Zudem gibt es noch die Variable `sqrMagnitude`, die den quadrierten Betrag zurückgibt.

Da ein kontinuierliches Berechnen der Beträge viel zu viel Rechenaufwand bedeuten würde, berechnet Unity diese Werte erst in dem Moment, in dem der Entwickler diese abfragt. Sollten Sie dabei häufiger auf den Betrag zugreifen, ist es empfehlenswert, nicht mit `magnitude` zu arbeiten, sondern mit dem Quadratwert (`sqrMagnitude`). Dies hat den Hintergrund, dass beim Berechnen des Betrages das rechenintensive „Wurzelziehen“ zum Einsatz kommt, was beim Abfragen des Quadratwertes nicht der Fall ist.

### Listing 5.2 Mit `sqrMagnitude` arbeiten

```
public Transform target;
public float minDistance = 3;
private float minSqrDistance;

void Start () {
    minSqrDistance = minDistance * minDistance;
}

void Update () {
    Vector3 distance = new Vector3();
    distance = target.position - transform.position;
    if (distance.sqrMagnitude < minSqrDistance) {
        //Code...
        Debug.Log ("Du bist aber nah!");
    }
}
```

## 5.2.2 Normalisieren

Unter dem Normalisieren eines Vektors versteht man das Kürzen der Vektorlänge auf die Länge 1. Die Vektorrichtung wird dabei beibehalten, sodass hier einzig und alleine die Richtung des Vektors interessant ist. Ein Vektor (0,1,0) bedeutet in Unity beispielsweise oben, der Vektor (-1,0,0) bedeutet links.

Zur Vereinfachung stellt Ihnen der *Vector3*-Typ bereits acht normalisierte Richtungsvektoren als statische Variablen zur Verfügung: *forward*, *back*, *up*, *down*, *left*, *right*, *one*, *zero*. Während die meisten selbsterklärend sein dürften, stellt *one* den *Vector3*-Wert (1, 1, 1) und *zero* den Wert (0, 0, 0) dar. Diese beiden sind allerdings keine normalisierten Vektoren.

Um den normalisierten Vektor von einer *Vector3*-Variablen zu erhalten, können Sie den Parameter *normalized* nutzen. Das folgende Beispiel ist ein Teil eines Skriptes, das die Normalisierung dafür nutzt, um zu beschreiben, in welche Richtung ein Abtaststrahl (*Raycast*) geschickt werden soll.

**Listing 5.3** Raycast überprüft die freie Sicht zum Spieler

```
public bool isInSight = false;
void OnTriggerEnter (Collider other){
    isInSight = false;
    Vector3 direction = other.transform.position - transform.position;
    RaycastHit hit;
    if(Physics.Raycast(transform.position, direction.normalized, out hit, 10)){
        if(hit.collider.CompareTag("Player"))
            isInSight = true;
    }
}
```

## ■ 5.3 Das Mesh

Nachdem Sie nun einzelne Punkte in dreidimensionalen Räumen beschreiben können, kommt nun das eigentliche 3D-Modell. Das kann ein Haus, eine Waffe oder auch eine Spielerfigur sein. Ein 3D-Modell besteht aus vielen einzelnen Vektoren, auch *Vertices* genannt, die die Form des Modells beschreiben. Hierbei werden immer drei nahe liegende Vertices zusammengefasst, die gemeinsam eine kleine Dreiecksfläche beschreiben (auch *Triangle* oder *Polygon* genannt). Hierdurch entsteht bei komplexeren Objekten ein großes Gebilde, bestehend aus vielen kleinen Dreiecksflächen. So eine Struktur wird auch *Polygonnetz* oder *Mesh* genannt. Letzterer ist der, den Sie in Unity vorfinden.



### Performance-Tipps

Umso detaillierterer 3D-Modelle bzw. *Meshes* sind, desto mehr Performance kosten diese. Vermeiden Sie deshalb unnötige *Triangles* in Ihren Modellen.

Auch ist es performancefreundlicher, ein großes *Mesh* mit vielen *Triangles* zu haben, als diese *Triangles* auf mehrere *Meshes* zu verteilen.

Beispiel: Anstatt eine Autokarosserie aus vielen kleinen einzelnen *Meshes* bestehen zu lassen (ein *Mesh* für die linke Tür, eins für die rechte Tür, eins für die Motorhaube usw.), ist es performanter, die Karosserie in einem großen *Mesh* darzustellen.

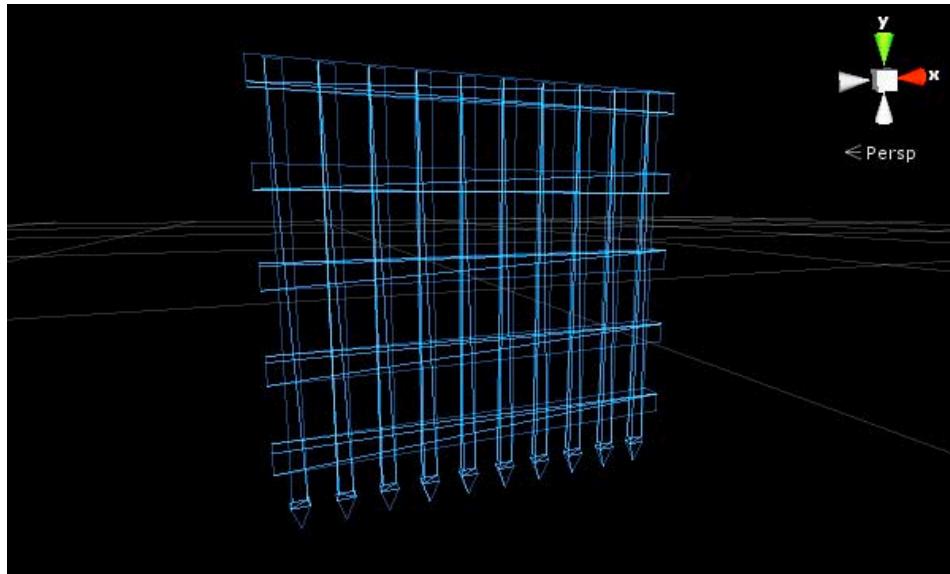


Bild 5.4 Mesh eines Fallgatters

### 5.3.1 Normalenvektor

Neben den bereits bekannten *Mesh*-Bestandteilen sind auch die sogenannten *Normalenvektoren* wichtig, kurz *Normalen* genannt.

*Normalen* sind für die Berechnung des Lichteinfalls, dessen Brechung und die daraus resultierenden Reflexionen zuständig und stehen für gewöhnlich rechtwinklig auf jeder Dreiecksfläche des *Meshs*. Durch deren Aufgabe beschreiben sie auch, von welcher Seite eine Fläche gesehen werden kann. Denn im Gegensatz zur realen Welt ist eine Fläche in Unity nur von einer Seite sichtbar, und zwar von der Richtung, in die der *Normalenvektor* zeigt. Von der anderen Seite aus wird die Fläche nicht dargestellt.



#### Verdrehte Normalen

Sollten Sie mal ein 3D-Modell in Unity importieren, wo einzelne Flächen nicht zu sehen sind, dann ist es sehr wahrscheinlich, dass die *Normalen* in die falsche Richtung zeigen. Am einfachsten ist es dann, die falsche Normale in der Modeling-Software zu drehen und das *Mesh* neu zu importieren.

### 5.3.2 MeshFilter und MeshRenderer

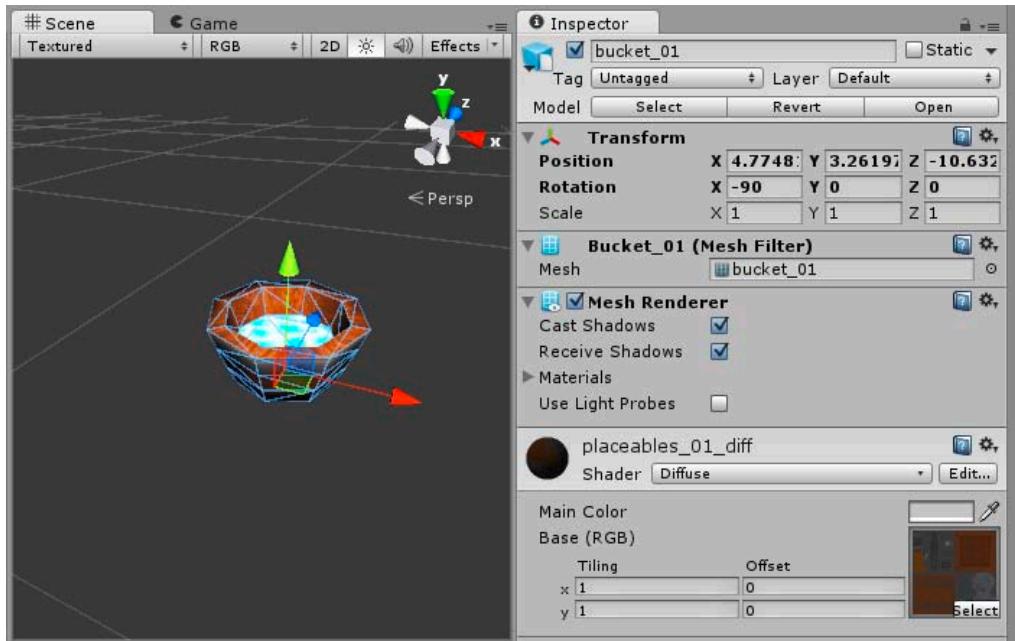
Da jedes Objekt, das sich in einer Unity-Szene befindet, immer ein *GameObject* ist, benötigen wir zum Darstellen eines 3D-Modells bzw. eines *Meshs* natürlich ebenfalls ein *GameObject*. Ein *GameObject* besitzt aber nicht die Fähigkeit, ein *Mesh* darzustellen. Hierfür benötigt es Komponenten.

Als Erstes benötigt ein *GameObject* einen **MeshFilter**. Diese Komponente sorgt dafür, dass das *GameObject* ein *Mesh* zunächst einmal überhaupt aufnehmen kann. Um das *Mesh* nun auch rendern zu können, braucht das *GameObject* noch eine weitere Komponente, den **MeshRenderer**. Von diesem gibt es wiederum zwei wichtige Typen:

- **MeshRenderer** wird bei herkömmlichen, starren Objekten eingesetzt
- **SkinnedMeshRenderer** wird bei animierten Objekten genutzt, dessen *Mesh* durch die Animationen verformt wird (siehe Kapitel „Animationen“).

Die letzte wichtige Komponente, die das *GameObject* noch benötigt, ist eine *Transform*-Komponente, die ich gleich noch etwas genauer erläutern werde.

Sobald Sie nun ein 3D-Modell aus dem *Project Browser* in die Szene hineinziehen, erzeugt Unity automatisch ein *GameObject* mit allen notwendigen Komponenten, mit dem Sie dann auch weiterarbeiten können.



**Bild 5.5** GameObject mit Komponenten zum Anzeigen eines 3D-Modells

Auf jede der einzelnen Komponenten können Sie per Code zugreifen. Sie könnten sogar während des Spiels das komplette *Mesh* modifizieren, um z.B. ein Modell zu zerteilen oder Einschlaglöcher von Granaten zu erzeugen. Aber auch einfachere Effekte wie ein Flackern des Spielers, wenn dieser verletzt wird, sind hierüber sehr leicht umzusetzen.

**Listing 5.4** Flacker-Effekt nach Schadenzuführung

```
public float currentHealth = 5;
void ApplyDamage(float damage)
{
    currentHealth -= damage;
    StartCoroutine(DamageEffect());
}

IEnumerator DamageEffect()
{
    renderer.enabled = false;
    yield return new WaitForSeconds(0.2F);
    renderer.enabled = true;
    yield return new WaitForSeconds(0.2F);
    renderer.enabled = false;
    yield return new WaitForSeconds(0.2F);
    renderer.enabled = true;
    yield return new WaitForSeconds(0.2F);
    renderer.enabled = true;
}
```

## ■ 5.4 Transform

Die wichtigste Komponente eines *GameObjects* ist die *Transform*-Komponente. Sie beschreibt die Position, die Drehung und die Skalierung des *GameObjects* im dreidimensionalen Raum. In der Entwicklungsphase können Sie diese Parameter sowohl über den *Inspector* als auch direkt in der *Scene View* mit den *Transform-Tools* ändern.

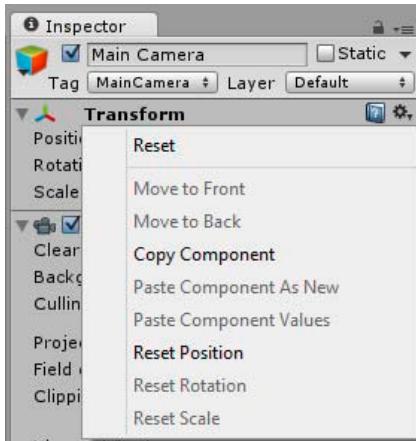
Da die *Transform*-Komponente so grundlegend ist, besitzt bereits jedes *Empty GameObject*, das Sie über **GameObject/Create Empty** erstellen, eine *Transform*-Komponente. So ganz „empty“ sind diese *Empty GameObjects* also dann doch nicht.

**Wem gehört das Handle?**

Da die Position eines *GameObjects* in der *Transform*-Komponente festgelegt wird, kann ein *GameObject* ohne *Transform* auch an keinem bestimmten Ort in einer Szene platziert werden. Der *Handle*, mit dem Sie ein *GameObject* in der Szene verschieben, gehört dementsprechend eigentlich auch nicht zum *GameObject*, sondern zur *Transform*-Komponente.

### 5.4.1 Kontextmenü der Transform-Komponente

Wie auch jede andere Komponente besitzt auch die *Transform*-Komponente im *Inspector* ein kleines Kontextmenü, symbolisiert durch ein Zahnrad.

**Bild 5.6**

Kontextmenü der Transform-Komponente

Hierbei ist vor allem die **Reset**-Funktion interessant, die alle Werte der Komponente auf Default-Werte zurücksetzt, also die Position und die Rotation auf (0,0,0) und die Skalierung auf (1,1,1). Mit **Reset Position**, **Reset Rotation** und **Reset Scale** können Sie die Parameter aber auch separat zurücksetzen.

## 5.4.2 Objekthierarchien

Häufig befinden sich *GameObjects* in hierarchischen Baumstrukturen. Wenn Sie beispielsweise einen Spielcharakter betrachten, dann besteht dieser normalerweise aus einem Hauptobjekt (auch *Root* genannt), das weitere Unterobjekte besitzt. Dieses Konzept nennt sich Parenting und ist ein wesentliches bei Unity. Wenn Sie nun die Parameter der Transform-Komponente des Eltern-Objekts verändern, z.B. Sie verschieben es, dann werden auch alle Kind-Objekte mit verschoben.

Allerdings bringt dieses Verfahren etwas Neues mit sich. Und zwar müssen Sie nun zwischen lokalen und globalen Werten unterscheiden. Im *Inspector* werden dabei immer die lokalen Werte einer *Transform*-Komponente angezeigt. Bei lokalen Werten beziehen sich diese immer auf das eigene Eltern-Objekt, bei globalen bezieht sich dies auf das Weltkoordinatensystem.

Wenn wir das obige Beispiel des Verschiebens betrachten, dann verursacht das Verschieben des Eltern-Objektes für die Kind-Objekte keine lokalen Änderungen, obwohl sich diese global natürlich sehr wohl ändern. Auch ein Skalieren des Eltern-Objektes wirkt sich direkt auf die Kind-Objekte aus, sie werden mit skaliert. Trotzdem wird im *Inspector* der Kind-Objekte keine Änderung angezeigt. Erst wenn Sie das Kind-Objekt in der *Hierarchy* aus dem Verbund mit seinem *Root*-Objekt heraustrennen und es dort als eigenständiges Objekt platzieren, ändern sich auf einmal alle Werte im *Inspector*: Die globalen Werte bleiben erhalten, nur die lokalen Werte verändern sich auf einmal, da sich die *Root* des Objektes ändert.



### Lokale und globale Angaben

Nur bei *Root*-Objekten, also Objekten, die keinem anderen *GameObject* untergeordnet wurden, sind lokale und globale Angaben identisch.

### 5.4.3 Scripting mit Transform

Die *Transform*-Klasse stellt für Sie als Spieleprogrammierer eine ganze Menge nützlicher Eigenschaften und Methoden bereit. Hier eine kleine Auswahl, wobei die kleingeschriebenen Variablen sind, denen Sie sowohl Werte zuweisen und die Sie auch abfragen können:

- **eulerAngles** Globale Drehungen in Grad.
- **localEulerAngles** Lokale Drehung in Grad relativ zum Eltern-Objekt
- **position** Globale Position
- **localPosition** Lokale Position relativ zum Eltern-Objekt
- **rotation** Die globalen Drehungen gespeichert als *Quaternion*
- **localRotation** Die lokalen Drehungen relativ zur Eltern-Objekt-Rotation als *Quaternion*
- **parent** Zugriff auf die *Transform*-Komponente des Eltern-Objekts
- **LookAt** Diese Methode dreht das Objekt, sodass die eigene positive Z-Achse in Richtung eines Zielobjektes zeigt.
- **Rotate** Die Methode fügt dem Objekte eine Drehung zu. Über einen Extraparameter kann angegeben werden, ob sich die Angabe auf das lokale oder globale Koordinatensystem bezieht.
- **Translate** Die Methode bewegt das Objekt in eine Richtung. Über einen Extraparameter kann angegeben werden, ob sich die Angabe auf das lokale oder globale Koordinatensystem bezieht.

Das folgende Beispiel bewegt ein *GameObject* über die Methode *Translate* nach vorne und rotiert es gleichzeitig über die Methode *Rotate* rechts herum um die Z-Achse.

#### Listing 5.5 GameObject verschieben und drehen

```
float speedFactor = 2;
float rotationFactor = 100;
void Update ()
{
    transform.Translate(Vector3.forward * Time.deltaTime * speedFactor);
    transform.Rotate(Vector3.back * Time.deltaTime * rotationFactor);
}
```

#### 5.4.4 Quaternion

*Quaternion* ist ein spezieller Datentyp für Rotationen. Im Gegensatz zu einem *Vector3*-Wert beinhaltet ein *Quaternion* vier Werte (*w*, *x*, *y*, *z*), die Sie aber am besten nie direkt verändern sollten. Stattdessen sollten Sie die Methoden und Eigenschaften nutzen, die dieser zur Verfügung stellt. So besitzt ein *Quaternion* zum Beispiel die Variable *eulerAngles*, der Sie die Drehungen im *Vector3*-Format übergeben oder die Sie auch erfragen können.

**Listing 5.6** Verwendung von *eulerAngles* einer *Quaternion*-Variablen

```
Quaternion rotation = Quaternion.identity;
rotation.eulerAngles = new Vector3(0, 45, 0);
```

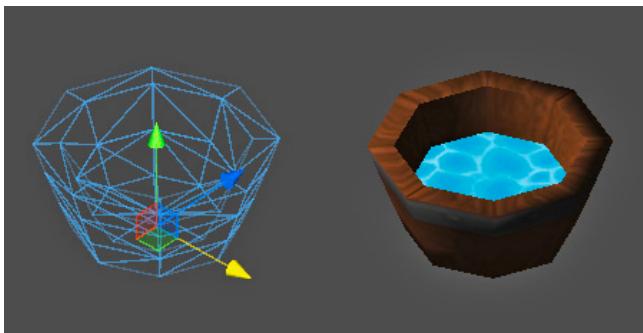
Die statische *ReadOnly*-Variable *identity* bedeutete wiederum so viel wie „keine Drehung“. Ein typisches Beispiel ist die Verwendung beim Erstellen einer Prefab-Instanz mit der Methode *Instantiate*, bei der das *GameObject* nicht noch zusätzlich gedreht werden soll. Mehr hierzu lesen Sie im Kapitel „Prefabs“.

Quaternions bieten noch eine Menge mehr. Über den bereits erwähnten Shortcut **[Strg] + [Umsch] + [1]** (siehe Kapitel „Skript-Programmierung“) erfahren Sie in der Hilfe von MonoDevelop mehr über diesen Datentyp.

## ■ 5.5 Shader und Materials

Da ein *Mesh* lediglich aus einer großen Menge von Vektoren besteht, die wiederum Dreiecksflächen bilden, beschreibt ein solches Drahtgitter lediglich die Form eines 3D-Modells. Was hier noch fehlt, ist die Beschreibung des Oberflächenaussehens.

Hier kommen *Materials* (Materialien) ins Spiel. Sie legen fest, welche Texturen und welche Farben die Oberflächen darstellen sollen und wie sich diese visuell auf verschiedene Einflussfaktoren verhalten. Zur Berechnung des Aussehens kommen dabei sogenannte *Shader* zum Einsatz. Sie beschreiben in einer speziellen Programmiersprache, wie die Materialien auf den Oberflächen gerendert werden sollen.



**Bild 5.7**

Mesh ohne und mit zugewiesenen Material

Unity bringt eine ganze Palette fertiger *Shader* von Haus aus mit, die sich aber in ihren Eigenschaften und den zur Verfügung stehenden Parametern stark unterscheiden können. So reichen diese sogenannten *Build-In-Shader* von einfachsten matten *Shadern* für das Zuweisen einer einfachen Textur über komplexe *Reflexionsshader* mit Tiefeneffekten bis hin zu *Skybox-Shadern*, die für die Darstellung des Himmels zuständig sind.

Wenn Sie nun einem *Material* einen *Shader* und ggf. Farben und Texturen zugewiesen haben, können Sie dies anschließend dem *Mesh-Renderer* eines *GameObjects* zuweisen, das dann das *Material* auf dem *Mesh* darstellt.



### So kommt die Textur auf das Mesh

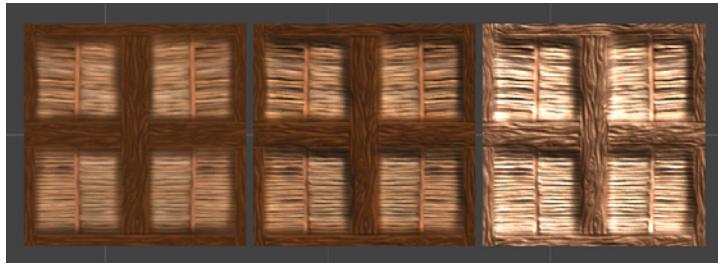
Um einem Modell eine Textur zuzuweisen, benötigen Sie neben der Textur auch immer ein *Material*. Diesem weisen Sie die Grafikdatei zu. Der *Shader* des *Materials* bestimmt dann, wie die Textur dargestellt werden und sich verhalten soll.

#### 5.5.1 Material-Eigenschaften

*Materials* reichen, wie bereits gesagt, die Variablen der *Shader* nach außen, damit wir als Entwickler die Farben, Texturen und andere Parameter unseren Vorstellungen entsprechend wählen und zuweisen können. Hierbei gibt es einige Parameter, die sehr häufig Verwendung finden und ich deshalb kurz vorstellen möchte:

- **Main Color** legt die Hauptfarbe fest, die noch einmal über alle anderen Texturen gelegt wird.
- **Base (RGB)** ist die Standardtextur des Materials. *Tiling* bestimmt die Wiederholungen der Textur in die Achsrichtung und *Offset* verschiebt die Textur in die Achsrichtungen.
- **Specular Color** legt die Glanzfarbe des *Materials* fest.
- **Base** ist die Basistextur des *Materials*.
- **Normalmap** ist eine spezielle Textur, deren Farbwerte die Ausrichtungen von *Normalen* repräsentieren. *Shader* mit diesen Texturen entnehmen die *Normaleninformationen* nicht vom *Mesh* (siehe *Normalenvektor*), sondern von diesen Texturen. Dadurch können 3D-Eindrücke entstehen, obwohl das *Mesh* selber ganz flach ist.
- **Reflection Cubemap** nimmt eine *Cubemap* auf. Eine *Cubemap* ist ein *Asset-Typ*, der aus sechs verschiedenen Grafiken besteht. Je nach Betrachtungswinkel werden diese Texturen dann als Reflexionen auf dem Objekt dargestellt.
- **Heightmap** ist eine Grauton-Textur (engl. Grayscale), die Höhen und Tiefen beschreibt. Umso dunkler der Pixel, desto weiter entfernt liegt der Punkt vom Betrachter. Hiermit simulieren *Shader* Tiefeneffekte auf einem 3D-Modell, die eigentlich nicht da sind.

Dank dieser und weiterer Parameter lassen sich mit *Shadern* die unterschiedlichsten Darstellungseffekte erzielen. Das Bild 5.8 zeigt zum Beispiel drei unterschiedliche *Shader*, die auf einer einfachen quadratischen Fläche eine Textur darstellen. Während bei der ersten der *Shader* des *Materials* nur die Textur an sich zeigt, nutzen die anderen beiden zusätzlich



**Bild 5.8** Shader-Vergleich: Diffuse, Bumped Diffuse und Bumped Specular

noch eine *Normalmap*, um der Oberfläche etwas mehr Tiefe zu verleihen. Der dritte *Shader* unterstützt zudem noch einen Glanzeffekt, wodurch Lichtreflexionen entstehen.

### 5.5.1.1 Import von Texturen

Beim Importieren von Texturen bietet Ihnen Unity eine sehr umfangreiche Palette an Parametern an. Da das Beschreiben alle Parameter mehrere Seiten füllen würde, möchte ich an dieser Stelle nur einige wichtige vorstellen. Eine ausführliche Beschreibung aller Parameter finden Sie im Manual, das Sie über die Hilfe-Funktion beim *Texture Import* erreichen.

Der wichtigste Parameter der *Import Settings* einer Textur ist der **Texture Type**, der wichtige Basic-Einstellungen vornimmt. Abhängig von diesem, werden Ihnen dann weitere Parameter angeboten. Sie können folgende *Texture Types* wählen:

- **Texture** wird bei herkömmlichen Texturen benutzt.
- **Normal Map** wird bei *Normalmaps* genutzt.
- **Sprite** wird häufig bei 2D Games genutzt und wandelt das Bild in den Assettyp *Sprite* um.
- **Cursor** wird bei Grafiken genutzt, die den Mauszeiger ersetzen sollen.
- **GUI** wird bei Grafiken von Benutzeroberflächen eingesetzt (z.B. Buttons).
- **Reflection** wird bei *Cube Maps* eingesetzt, die zum Vortäuschen von Reflexionen genommen werden.
- **Cookie** wird bei *Light-Cookies* eingesetzt (siehe Kapitel „Licht und Schatten“).
- **Advanced** eignet sich bei Grafiken, die spezielle Einstellungen erfordern, z.B. bei Texturen, deren Größe nicht auf Zweierpotenzen aufbaut.

Wenn Sie Texturen in Unity nutzen, ist es optimal, wenn Sie Texturen nutzen, deren Seitenlängen auf Zweierpotenzen aufbauen, z.B.  $16 \times 16$  oder  $256 \times 32$ . Dem Parameter **Max Size** weisen Sie dann die größere Seitenlänge zu. Sie können aber natürlich auch andere Formate wählen, die nicht auf der Zweipotenz basieren. Nutzen Sie hier den *Texture Type Advanced* mit dem Parameter *None Power of 2*.

Ein weiterer wichtiger Parameter ist **Format**. Dieser hat Einfluss auf die Qualität und den Speicherbedarf der Textur. Zur Verfügung stehen:

- **Compressed** wandelt die Grafik in eine komprimierte RGB-Textur. *Compressed* ist das Standardformat, das auch am häufigsten verwendet wird.
- **16 bits** erzeugt eine Low-Quality-Truecolor-Textur, bestehend aus 16 Rot-, Grün-, Blau- und Alpha-Stufen.

- **Truecolor** erzeugt High-Quality-Auflösung der Textur, die auch den meisten Speicher benötigt.

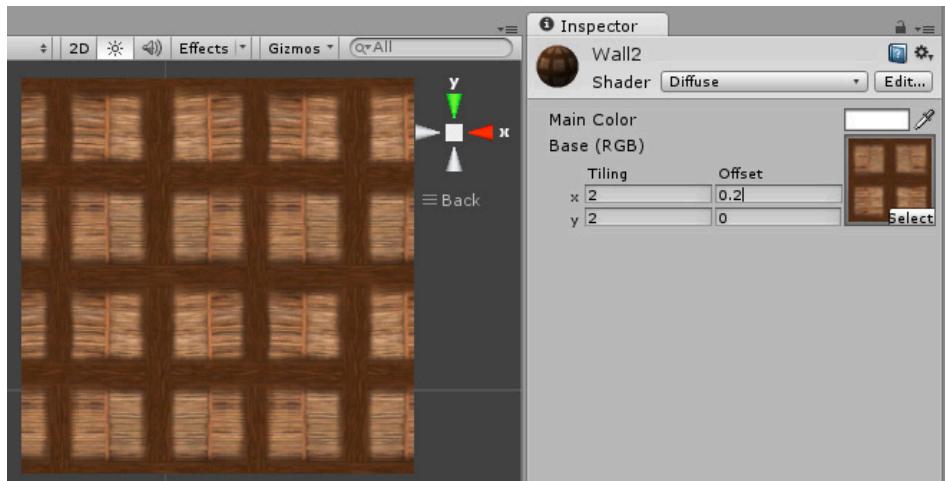
Ein weiterer wichtiger Parameter ist der **Wrap Mode**, der abhängig von der Wiederholung des Bildes gesetzt werden sollte.

- **Repeat** sollte gewählt werden, wenn das Motiv über den *Material*-Parameter *Tiling* wiederholt wird.
- **Clamp** sollte gewählt werden, wenn das Motiv nur einmal gezeichnet wird.

### 5.5.1.2 Effekte von Tiling und Offset

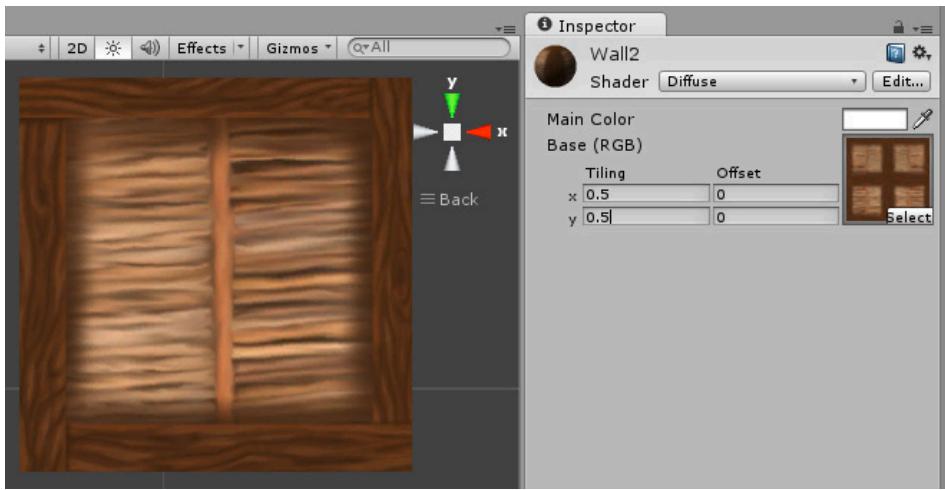
Jede Textur besitzt dabei die beiden Parameter-Pärchen *Tiling* und *Offset*, die mehr Möglichkeiten bieten, als man am Anfang denken mag. Denn je nachdem, ob Sie den *Tiling*-Parametern Werte größer oder kleiner als 1 zuweisen, erhalten Sie völlig verschiedene Effekte und Möglichkeiten.

Beim Zuweisen von Werten größer als 1 verursacht *Tiling* ein Wiederholen der Textur in die jeweiligen X- und die Y-Richtungen in Höhe der angegebenen Werte. Weisen Sie X und Y jeweils den Wert 2 zu, wird die Textur in beide Richtungen doppelt angezeigt. Beachten Sie hierbei, dass Sie die Texturen strecken bzw. stauchen, wenn X und Y unterschiedliche Werte erhalten. Ein Offset zwischen 0 und 1 verschiebt zudem den Anfang der ersten Textur am Rand, sodass der Rest dann auf der gegenüberliegenden Seite angezeigt wird. Beachten Sie hierbei, dass diesen Texturen in den *Import Settings* der *Wrap Mode* *Repeat* zugewiesen wird.



**Bild 5.9** Material mit wiederholender Textur und einem horizontalen Offset

Beim Zuweisen niedriger Werte als 1 verursacht *Tiling* einen völlig anderen Effekt. Hierdurch wird die Textur nicht mehr mehrfach, sondern nur noch ausschnittsweise angezeigt. Weisen Sie X und Y jeweils 0.25 zu, wird nur noch ein Viertel der Textur angezeigt. Über die *Offset*-Parameter können Sie jetzt noch bestimmen, welches Viertel angezeigt wird. Ein Offset von (0,0) zeigt das Viertel unten links an. Ein Offset von (0.5,0.5) zeigt das Viertel oben rechts an. Zum Darstellen sauberer Kanten sollten Sie hier den *Wrap Mode* auf *Clamp* setzen.



**Bild 5.10** Material mit Texturausschnitt

## 5.5.2 Neues Material erstellen

Beim Erstellen eines *Materials* können Sie wie folgt vorgehen:

1. Erstellen Sie ein neues *Material-Asset* über das Hauptmenü **Asset/Create/Material** oder über **Create/Material** im *Project Browser*.
2. Wählen Sie einen geeigneten *Shader*. Die *Shader*-Auswahl eines *Materials* finden Sie im oberen Bereich des *Inspectors* (siehe Bild 5.10).
3. Weisen Sie den verschiedenen Parametern Farbwerte und Texturen zu und legen Sie bei den Texturen die *Tiling*- und *Offset*-Werte fest.
4. Anschließend können Sie das Material der Material-Eigenschaft eines *MeshRenderers* zufügen oder direkt auf ein *Mesh* in der *Scene View* per Drag & Drop fallen lassen. Unity fügt dann automatisch das Material dem *MeshRenderer* zu.

## 5.5.3 Normalmaps erstellen

Unity bietet Ihnen die Möglichkeit, in Unity eine *Normalmap* zu erstellen. So können Sie aus einer *Heightmap* eine *Normalmap* erstellen. Bei einer *Heightmap* handelt es sich um eine Grauton-Textur, bei der die Farbe die Höhenlage beschreibt. Weiß bedeutet hierbei hoch, Schwarz bedeutet tief.

Stellen Sie für diese Konvertierung den *Texture Type* auf *Normalmap* und aktivieren Sie den dazugehörigen Parameter **Create from Grayscale**. Der Parameter *Bumpiness* gibt Ihnen dabei die Kontrolle darüber, wie stark der Effekt zur Geltung kommen soll. Über den *Filering*-Parameter definieren Sie zusätzlich, ob die Übergänge zwischen den Höhenunterschieden abrupt oder eher weich abfallen/aufsteigend sein sollen.



Bild 5.11  
Heightmap



Bild 5.12  
Zwei Normalmaps mit unterschiedlichen Bumpiness-Werten

Einige *Shader* benötigen sowohl eine *Heightmap* als auch eine *Normalmap*. In diesem Fall sollten Sie natürlich vor der Konvertierung eine Kopie der *Heightmap* machen (z.B. über **Strg + D**). Wie sich nun der Einfluss dieser *Normalmap* auf die Darstellung einer gewöhnlichen Textur auswirkt, können Sie dem Bild 5.13 entnehmen. Links sehen Sie die Textur ohne weitere Effekte, rechts sehen Sie diese mit einer zusätzlichen *Normalmap* dargestellt.

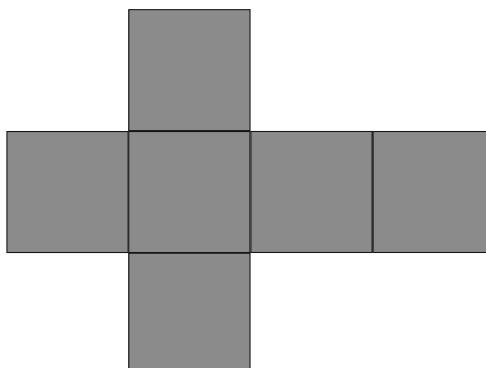


Bild 5.13  
Einfluss einer Normalmap auf die Darstellung einer Textur

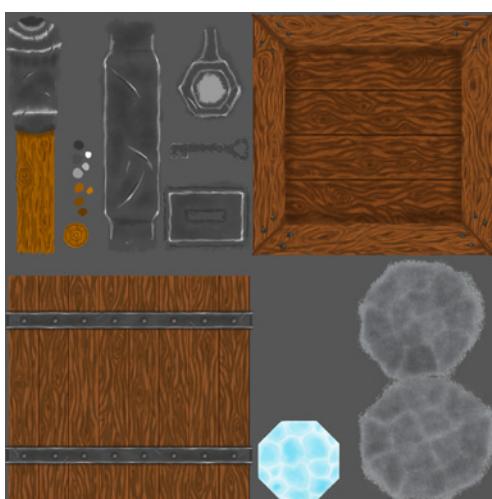
### 5.5.4 UV Mapping

Wenn Sie sich mit *Meshes*, *Materials* und *Textures* beschäftigen, werden Sie immer wieder über die Bezeichnung *UV* stolpern. Die Buchstaben *U* und *V* stehen hierbei für die Achsenbezeichnungen des Koordinatensystems, auf dem die zweidimensionale Textur abgebildet wird. Das dreidimensionale *Mesh* wird, wie bereits bekannt, mit den Koordinaten *X*, *Y* und *Z* beschrieben. Beim *UV-Mapping* wird nun festgelegt, welcher Teil der Textur auf welchem *Triangle* des dreidimensionalen *Mesh* liegen soll.

Als Erstes wird hierbei das *Mesh* entfaltet. Das Entfalten können Sie sich am Beispiel eines Pappkartons vorstellen. Dieser wird an den Kanten zerschnitten, um ihn schließlich flach auf den Boden legen zu können. Beim *UV-Mapping* wird nun so ein virtuell entfalteter Karton auf eine Textur gelegt, um dort festzulegen, welche Stelle der Textur wo auf dem Karton später angezeigt werden soll. Dieses Vorgehen funktioniert bei jedem dreidimensionalen Körper, auch bei Kugeln oder auch komplexen Körpern wie dem Modell einer Autokarosserie oder eines Drachens.



**Bild 5.14**  
Entfalteter Würfel



**Bild 5.15**  
Eine Textur für mehrere Meshes

Da Sie nun mithilfe des *UV-Mappings* bestimmen, welcher Teil der Textur wo auf dem *Mesh* angezeigt werden soll, können Sie auf einer Textur auch die Grafiken weiterer *Meshe*s platzieren. Hierdurch können Sie mit einem einzigen *Material* und dessen Textur beliebig viele Objekte texturieren, auch wenn diese völlig unterschiedlich aussehen. In dem Beispiel-Game nutzen beispielsweise die Fässer, Kisten und Gefäße ein und das gleiche *Material*.

## ■ 5.6 3D-Modelle einer Szene zufügen

Auch wenn die vorherigen Unterkapitel vielleicht etwas kompliziert klangen, so ist die eigentliche Praxis recht einfach. Im Grunde brauchen Sie sich um die obigen Teile kaum zu kümmern, da Unity im Hintergrund bereits das meiste für Sie erledigt.

### 5.6.1 Primitives

Unity stellt Ihnen über das Hauptmenü **GameObject/Create General/Cube/Sphere/...** einige 3D-Grundobjekte zur Verfügung, in Unity *Primitives* genannt, die Sie in Ihren Projekten nutzen können. Jedes *Primitive* besitzt hierbei nicht nur die typischen Komponenten, die ich bereits oben beschrieben hatte, sie besitzen zusätzlich auch eine *Collider*-Komponente, um mit anderen Objekten zu interagieren. Mehr zu diesem Thema erfahren Sie im Kapitel „Physik in Unity“.

Auch wenn Sie wohl eher selten Spiele erstellen werden, die nur aus *Primitives* bestehen, werden diese doch sehr häufig in Unity-Projekten genutzt.

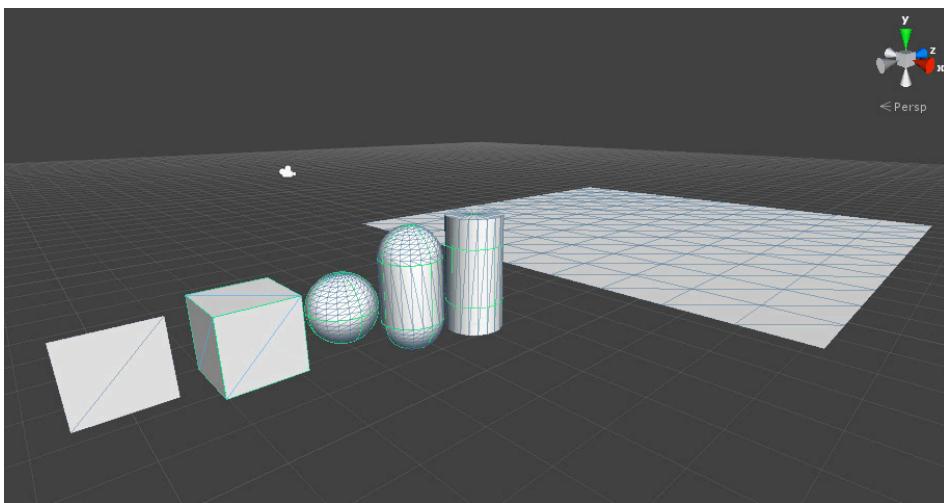
- **Cube** stellt einen sechsseitigen Würfel dar. Häufig werden Cubes beim *Prototyping* genutzt, wo die endgültigen Modelle noch nicht fertig sind. Auch werden sie bei grafisch nicht zu anspruchsvollen Spielen zum Darstellen von Mauern oder einfachen Boxen genutzt. Die aber wohl häufigste Anwendung finden sie als *Trigger*- und *Collider*-Objekte, wobei der *Renderer* deaktiviert und lediglich dessen *Collider* genutzt wird. Mehr hierzu erfahren Sie im Kapitel „Physik in Unity“.
- **Sphere** erzeugt eine Kugel. *Spheres* werden gerne zum Darstellen von Kugeln, Planeten und anderen runden Objekten genutzt.
- **Capsule** ist ein kapselförmiges Objekt. Da es in der realen Welt wenige Objekte mit dieser Form gibt, werden diese meistens zum *Prototyping* genutzt. Viele Unity-Entwickler nutzen diese hierbei, um Figuren symbolisch darzustellen.
- **Cylinder** stellt einen Zylinder dar. Mit diesen können Sie einfache Räder, Achsen oder andere Stangen darstellen.
- **Quad** stellt eine quadratische Fläche dar, dessen Fläche in die lokalen X- und Y-Achsen aufgespannt ist. Die Ausrichtung entspricht damit der Default-Kameraausrichtung und der 2D-Kamera-Sicht von Unity. Ein *Quad* besteht aus zwei *Triangles* und wird vor allem zum Darstellen von Bildern und Informationen wie z.B. Partikeln oder einfachen GUI-Elementen genutzt.

## 11.2 5 Objekte in der dritten Dimension

- **Plane** stellt eine zweidimensionale Platte dar, deren Fläche in die lokalen X- und Z-Achsen aufgespannt ist. Die Normalen der Plane sind nach oben gerichtet, weshalb sie auch gerne zum Darstellen von Untergründen genommen werden. Sie können natürlich auch gedreht und skaliert werden, sodass sie auch häufig als Wände und Ähnliches Verwendung finden. Eine Plane ist zehnmal so groß wie ein *Quad* und besitzt dementsprechend 200 *Triangles*.

 **Einheiten und Messinstrumente**

Eine Einheit in Unity entspricht einem Meter in der realen Welt. Da ein Cube bei einer Skalierung von 1 eine Kantenlänge von einer Einheit besitzt, eignen sich Cubes gut als Messinstrument.



**Bild 5.16** Primitives: Quad, Cube, Sphere, Capsule, Cylinder, Plane

### 5.6.2 3D-Modelle importieren

In den meisten Fällen werden Sie sicher nicht nur mit *Primitives* arbeiten, sondern eigene 3D-Modelle verwenden, die Sie vorher in einer 3D-Modelling-Software erstellt haben. Unity unterstützt hierbei die Formate .FBX, .DAE, .3DS, .DXF und .OBJ, wobei das bevorzugte Format FBX ist. Neben diesen Formaten können auch ganze Projektdateien von gängigen 3D-Modellierungstools importiert werden. Dafür muss allerdings auch das dementsprechende Werkzeug auf dem gleichen Rechner installiert sein. Hierzu gehören aktuell die Tools Max, Maya, Blender, Cinema4D, Modo, Lightwave und Cheetah3D.

Sobald Sie nun eine Datei in den *Project Browser* ziehen, wird das 3D-Modell in Unity importiert. Beim Import wird für das 3D-Modell ein *Model-Prefab* im *Project Browser* abgelegt, für das, sobald es in die Szene gezogen wird, ein *GameObject* mit allen notwendigen Komponenten angelegt wird. Die restlichen Inhalte des 3D-Modells, wie Texturen, *Materials*, Animatio-



**Bild 5.17** Importierte Dateien eines 3D-Modells mit mehreren Sub-Meshes

nen etc., werden in untergeordneten Ordnern als separate *Assets* abgelegt, die dann dem Hauptobjekt wieder zugewiesen werden.

Wie bei jedem Automatismus kann es auch hier vorkommen, dass dieser Automatismus nicht ganz korrekt funktioniert. In dem Fall müssen Sie diese Zuweisungen nach dem Import noch einmal manuell nachholen. Zudem können Sie in den *Import Settings* des Modells noch weitere Details zum Import definieren.

### 5.6.2.1 Model Import Settings

Die *Import Settings* eines 3D-Modells bestehen insgesamt aus drei Reitern. Dabei ist der Reiter **Model** für den eigentlichen *Mesh*-Import wichtig. Die anderen Reiter werden noch in dem Kapitel „Animationen“ behandelt, da diese für das Animieren der mitgelieferten Animationen des Modells zuständig sind. Im Folgenden werde ich nun die essenziellsten Parameter vorstellen, die für den Import des 3D-Modells wichtig sind.

- **Scale Factor** bestimmt die Skalierung eines Objektes beim Import. Unitys Physik-System interpretiert hierbei eine Einheit als einen Meter.
- **Generate Colliders** fügt jedem *Mesh* einen *MeshCollider* zur Kollisionserkennung zu (siehe „Physik in Unity“).
- **Swap UVs** wechselt den ersten und zweiten UV-Kanal.
- **Generate Lightmap UVs** erzeugt einen zweiten UV-Kanal zum Darstellen von Lightmap-Texturen (siehe „Licht und Schatten“).
- **Import Materials** importiert die inkludierten Materialien des 3D-Modells.

### 5.6.3 In Unity modellieren

Mithilfe einiger *Editor Extensions* haben Sie die Möglichkeit, direkt in Unity eigene 3D-Modelle zu erstellen. So gibt es Tools, mit denen Sie z.B. Straßen direkt in Unity modellieren können. Bei diesen kümmern sich die Tools für gewöhnlich um das Anlegen der ver-

schiedenen *GameObjects*, Materialien und der benötigten Komponenten. *Editor Extensions* erhalten Sie z.B. im *Asset Store* von Unity ([Window/Asset Store](#)). Mit etwas Erfahrung können Sie natürlich auch selber solche Tools programmieren. Allerdings soll das kein Thema dieses Buches sein, da hierfür etwas mehr Erfahrung sowohl in Unity als auch im Programmieren benötigt wird.

#### 5.6.4 Prozedurale Mesh-Generierung

Eine weitere Möglichkeit, ein 3D-Modell in Unity zu erzeugen, ist die prozedurale *Mesh*-Generierung. Hierbei erzeugen Sie zur Laufzeit des Spiels ein *Mesh* per Programmcode. Bei diesem Verfahren müssen Sie sich natürlich um alles selber kümmern, was Unity beim Modelliimport im Hintergrund automatisch macht. Sie müssen sich um das *Mesh*, die Normalen, die Texturen und Materialien, die Komponenten und natürlich um das *GameObject* selber kümmern.

Dieses Vorgehen ist recht aufwendig und wird auch nur in Spezialfällen genutzt, z.B. zum Generieren zufällig gestalteter Gegenstände. Ähnlich wie das Entwickeln eigener *Editor Extensions* gehört auch dieser Bereich eher zu den anspruchsvolleren Tätigkeitsfeldern von Unity.

# 6

# Kameras, die Augen des Spielers

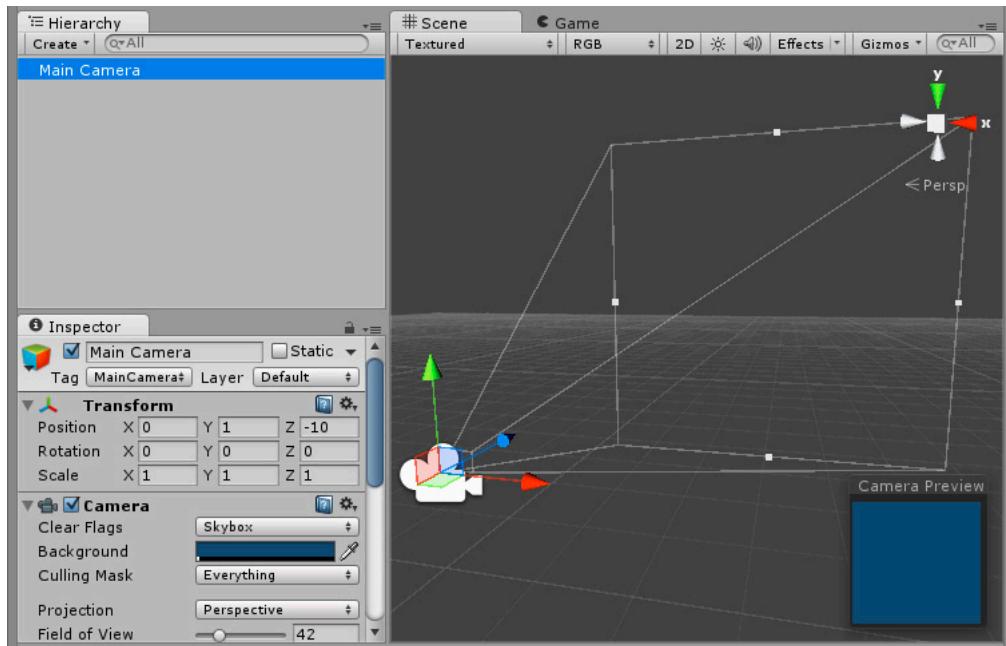
In diesem Kapitel geht es um die Sicht des Spielers auf das Spiel. Dies klingt recht trivial, ist aber ein ganz wichtiger Faktor eines Computerspiels. Ganze Spielgenres wie *First-Person-Shooter* oder die klassischen *Sidescroller-Games* identifizieren sich hierüber und haben sogar ihren Namen von dieser. Aber auch andere Spiele, wie z.B. Rollenspiele oder auch Autorennspiele, haben ganz typische Sichten, aus der der Spieler das Game steuert.

## ■ 6.1 Die Kamera

Sobald Sie in Unity eine neue Szene oder gar ein neues Projekt anlegen, wird Ihnen zunächst eine Default-Szene angelegt, die bereits ein Kamera-Objekt besitzt. Dieses *GameObject* heißt „Main Camera“ und besitzt zur Identifikation den Tag „MainCamera“.

Ein Kamera-*GameObject* symbolisiert nichts anderes als das Auge des Spielers. Möglich macht dies das *Camera-Component*, das dem *GameObject* angefügt wurde. Diese besitzt eine ganze Reihe an Parametern, die die Kamera sehr vielschichtig einsetzbar macht.

- **Clear Flags** legt fest, wie die Kamera ihre vorher gezeichneten Bilder überzeichnen soll. Normalerweise wird hier das alte Bild mit einer Art „Grundierung“ erst einmal überzeichnet. Und erst danach wird auf dieser der neue Frame gezeichnet. Da Unity aber nur die Objekte einer Szene rendert, bleibt überall dort, wo kein Objekt zu sehen ist, diese Grundierung zu sehen, z.B. in einer Outdoor-Szene der Himmel. Per Default ist deshalb auch der Parameter „Skybox“ gewählt, wodurch eine Art Himmelstextur gezeichnet wird (siehe Abschnitt „Skybox“). Es kann aber auch z.B. mit „Solid Color“ definiert werden, dass diese Grundierung lediglich eine Farbe ist. Ansonsten ist dieser Parameter aber vor allem dann wichtig, wenn Sie mit mehreren Kameras arbeiten, die unterschiedliche *GameObjects* zeichnen sollen (siehe Parameter *Culling Mask*).
- **Background** definiert eine Hintergrundfarbe, die dargestellt wird, wenn in *Clear Flags* die Option „Solid Color“ gewählt wurde.
- **Culling Mask** bestimmt, welche *GameObjects* von dieser Kamera gezeichnet werden. Hierbei wird der bei den *GameObjects* zu hinterlegende *Layer* genutzt. Objekte mit einem *Layer*, der bei diesem Parameter deaktiviert wurde, werden von der Kamera im Spiel auch nicht gezeichnet.



**Bild 6.1** Die „Main Camera“

- **Projection** legt fest, ob das Kamerabild eine dreidimensionale, perspektivische Ansicht haben soll (*Perspective*) oder eine orthogonale Ansicht (*Orthographic*), die keine perspektivische Sicht zulässt. Letztere wird vor allem bei 2D-Spielen genutzt. Im Kapitel „Grundlagen“ hatten wir diese unterschiedlichen Darstellungsformen bereits in der *Scene View* kennengelernt.
- **Field of View** bestimmt den Winkel der Kamera entlang der Y-Achse, angegeben in Grad (gibt es nur, wenn *Projection* auf *Perspective* gestellt ist).
- **Size** legt die Größe des Darstellungsausschnitts fest (gibt es nur, wenn *Projection* auf *Orthographic* gestellt ist).
- **Clipping Planes** bestimmt den Abstand, in dem Objekte im Bild der Kamera dargestellt werden. Objekte, die näher zum Spieler liegen als in *Near* definiert, werden genauso beim Rendern ignoriert wie die, die zu weit weg sind (*Far*).
- **Normalized View Port Rect** beschreibt, welchen Raum das Kamerabild vom gesamten Bildschirm einnehmen soll. X beschreibt den horizontalen Startpunkt des Kamerabildes auf dem Bildschirm. 0 bedeutet links, 1 ganz rechts. Y beschreibt den vertikalen Startpunkt des Kamerabildes auf dem Bildschirm. 0 bedeutet unten, 1 ganz oben. W beschreibt, wie viel Raum in der Breite das Bild einnehmen soll. 0 bedeutet keine, 1 bedeutet die volle Breite. H beschreibt, wie viel Raum in der Höhe das Bild einnehmen soll. 0 bedeutet keine, 1 bedeutet die volle Höhe.
- **Depth** bestimmt die Zeichnungsfolge dieser Kamera. Wenn Sie mehrere Kameras besitzen, überzeichnen die Kameras mit höheren *Depth*-Werten die Kamerabilder der niedrigeren Werte.

- **Rendering Path** definiert die von der Kamera zu nutzende Rendering-Methode (siehe „Rendering Paths“).
- **Target Texture** ist ein Pro-Feature und legt eine Zieltextur fest, auf der das Kamerabild projiziert werden soll.
- **HDR** aktiviert das *High Dynamic Range-Rendering* für diese Kamera.

Sobald Sie ein *Kamera-GameObject* selektieren, wird Ihnen in einem kleinen Vorschaubild unten links in der *Scene View* das aufgenommene Bild der Kamera angezeigt (siehe Bild 6.1). Wenn Sie nun das Spiel starten, werden Sie in der *Game View* genau das Gleiche sehen wie in der *Camera Preview*.

Unity ist aber nicht auf eine Kamera beschränkt, Sie können über das Menü **GameObject/Create General/Camera** weitere Kamera-Objekte in der Szene platzieren. Einige Beispiele für das Arbeiten mit mehreren Kameras werden Sie im Abschnitt „Mehrere Kameras“ behandeln.

### 6.1.1 Komponenten eines Kamera-Objektes

Ein *Kamera-GameObject* besitzt per Default bereits mehrere Komponenten, die in den weiteren Kapiteln aber noch näher behandelt werden.

- **Camera** ist für die eigentliche Bildaufnahme zuständig.
- **GUI Layer** ist für die Darstellung *GUI Text* und *GUI Texture*-Elemente zuständig.
- **Flare Layer** ist für das Darstellen von Lens Flares notwendig.
- **Audio Listener** ist zum Hören von Sound wichtig.

## ■ 6.2 Kamerasteuerung

Wie eingangs erwähnt, gibt es mehrere Möglichkeiten, eine Kamera in einem Spiel zu positionieren und zu nutzen. Eine wichtige Frage ist nun, wie man dafür sorgt, dass der Spieler nun auch in dem Bild zu sehen ist.

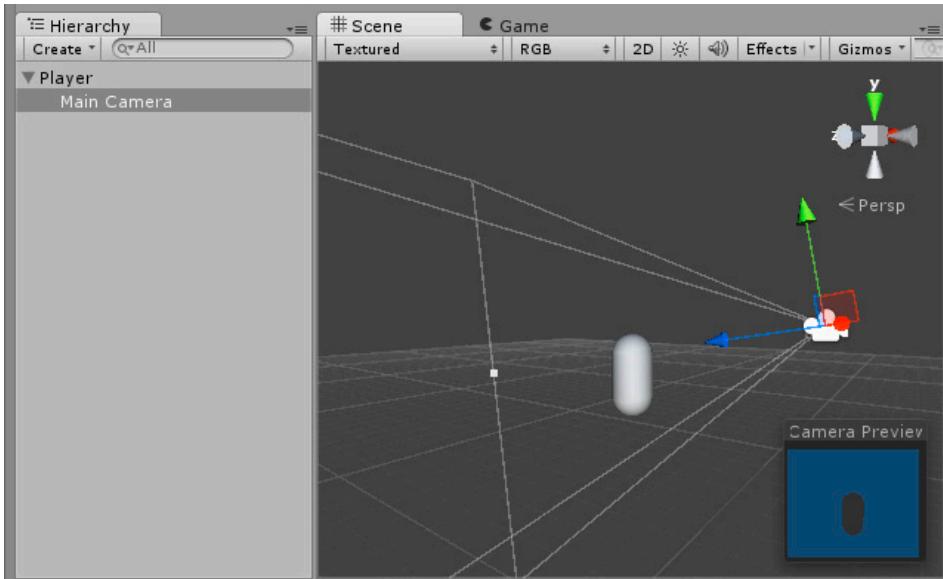
### 6.2.1 Statische Kamera

Die einfachste Möglichkeit ist die statische Positionierung der Kamera. Dies funktioniert natürlich nur, wenn das Spielgeschehen in einem übersichtlichen, fest definierten Bereich stattfindet.

Gerade im Casual-Game-Bereich, wo es um Schnelligkeit und Geschicklichkeit geht, werden gerne statische Kameras eingesetzt. Denken Sie nur an den Klassiker „Tetris“, wo das gesamte Spiel starr von der Seite betrachtet wird. Aber auch bei Denkspielen wie *Puzzle* etc. ist diese „Kameraführung“ recht beliebt.

## 6.2.2 Parenting-Kamera

Sobald Ihr Game eine Spielerfigur besitzt, die sich durch eine Szene bewegt, ist es meistens notwendig, dass das Kamerabild bzw. die Kamera dem Spieler in irgendeiner Form folgt. Die einfachste Möglichkeit hierbei ist es, die Kamera dem Spieler als Kind-Objekt hinzuzufügen.



**Bild 6.2** Eine als Kind-Objekt zugefügte Kamera

Solange die Bewegungen des Spielers sowie die Level-Gestaltung dieses Vorgehen erlauben, ist es eine sehr einfache und effektive Art der Kameraführung. Für dieses Vorgehen gibt es ein ganz prominentes Beispiel: den First-Person-Shooter. Hier kommt es sogar vor, dass es überhaupt keinen sichtbaren Spielcharakter gibt, sondern nur die Kamera und die Waffen, die in das Bild hineinragen.

## 6.2.3 Kamera-Skripte

Das obige *Parenting-Verfahren* hat natürlich auch seine Grenzen. Wenn sich der Spieler überschlägt, würde sich die Kamera mitüberschlagen, was aber in den meisten Spielen eher nicht erwünscht sein dürfte. Am häufigsten werden deshalb Skripte eingesetzt, um die Kamera zu steuern. Hierdurch können lebendigere Kamerabewegungen erreicht werden, die auch wesentlich weichere Kameraschwenks ermöglichen.

So werden bei Autospielen die Kameras eigentlich immer über Skripte an die Autos gebunden, statt sie einfach als starres Kind-Objekt anzufügen. Drehungen und Sprünge würden sonst einfach zu heftig auf dem Bildschirm erscheinen. Skripte können die Kamera abfedern, um so für eine geschmeidigere Spielsicht zu sorgen.

Skripte können zudem auch kleine Intelligenzen implementiert bekommen, die die Kamera immer so ausrichten, dass der Spieler im Kamerabild zu sehen ist, auch wenn sich dieser in einer kleinen Ecke quetscht. Nicht zuletzt gibt es aber auch Spiele, wo der Nutzer selber die Kamera separat zum Spielcharakter steuern kann, um beispielsweise das Gelände besser erkunden zu können. Sie sehen, man kann eine ganze Menge in die Steuerung einer Kamera implementieren.

Unity liefert in den *Standard Assets „Scripts“* gleich mehrere Skripte mit, die Sie für die Kamerasteuerung nutzen können. Aber auch auf der Wiki-Seite von Unity (<http://wiki.unity3d.com/>) finden Sie eine größere Auswahl an Skript-Vorlagen speziell für Kameras, die von der Unity-Community kostenlos zur Verfügung gestellt wurden.

### 6.2.3.1 Kamera-Skripte programmieren

Im Normalfall werden Kamera-Skripte der Kamera zugefügt. Damit das Skript aber weiß, wer das Ziel bzw. der Spieler ist, wird dieser einer Variablen zugewiesen, die häufig *target* oder *player* genannt wird. Die eigentliche Positionierung und Rotation der Kamera wird meistens dann in der *LateUpdate*-Methode berechnet, da der Spieler selber bereits in der *Update*-Methode bewegt wurde.

Das folgende Beispiel-Skript merkt sich am Anfang des Spiels den relativen Abstand zum Spieler und hält diesen während des gesamten Spieles aufrecht. Im Gegensatz zur *Parenting*-Variante dreht sich die Kamera aber nicht mit dem Spieler mit, sondern betrachtet das Spiel immer aus der gleichen Richtung. Diese Kamerasteuerung wird bei vielen klassischen Rollenspielen eingesetzt.

**Listing 6.1** Klassisches RPG-Kamera-Skript

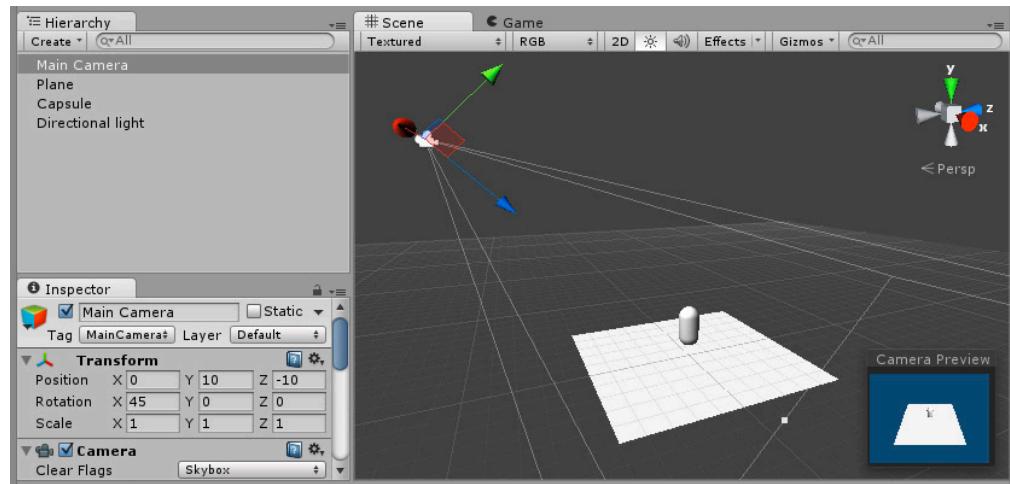
```
using UnityEngine;
using System.Collections;
public class CameraController : MonoBehaviour
{
    public Transform target;
    private Vector3 offset;
    void Start () {
        offset = target.position - transform.position;
    }
    void LateUpdate () {
        transform.position = target.position - offset;
    }
}
```

Um mit diesem Skript eine typische Rollenspielansicht zu erhalten, positionieren Sie einfach den Spieler beispielsweise auf der Koordinate (0, 0, 0) und die Kamera auf (0, 10, -10). Nun drehen Sie die Kamera noch 45° um die X-Achse nach unten, damit diese auf den Spieler hinabsieht. Der Spieler sollte sich jetzt in der Mitte des Kamerabildes befinden.



#### Vereinfachter Zugriff auf die Main Camera

Über die statische Variable *main* der *Camera*-Klasse greifen Sie direkt auf die Kamera zu, die den Tag „*MainCamera*“ besitzt. Haben Sie mehrere Kameras mit diesem Tag in der Szene, erhalten Sie eine zufällig ausgewählte.



**Bild 6.3** Traditionelle RPG-Ansicht im 45°-Winkel

## ■ 6.3 ScreenPointToRay

Die *Camera*-Klasse bietet einige sehr hilfreiche Methoden an, wovon ich die Methode *ScreenPointToRay* etwas näher vorstellen möchte.

*ScreenPointToRay* nimmt eine *Vector3*-Position auf dem Bildschirm und schickt von dort einen Strahl in die Spielewelt hinein. Häufig wird dieses Verfahren genutzt, um Gegenstände in einer Szene anzuklicken, auszuwählen o. Ä.

**Listing 6.2** Zerstöre das angeklickte Objekt

```
using UnityEngine;
using System.Collections;
public class ClickDestroyer : MonoBehaviour
{
    void Update()
    {
        if (Input.GetMouseButtonDown(0)) {
            Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
            RaycastHit hit;
            if (Physics.Raycast(ray,out hit)) {
                Destroy(hit.collider.gameObject);
            }
        }
    }
}
```

## ■ 6.4 Mehrere Kameras

Das Spannende ist, dass Sie nicht nur ein Kamera-Objekt in der Szene haben können, sondern beliebig viele. Daraus ergeben sich sehr spannende Möglichkeiten, wovon ich einige im Folgenden vorstellen möchte.

### 6.4.1 Kamerawechsel

Eine typische Art, mit mehreren Kameras zu arbeiten, ist das Wechseln zwischen den verschiedenen Kameraviews, wie z.B. zwischen einer First-Person-Kamera und Third-Person-Kamera bei einem Autorennspiel. Über ein Steuerungsskript wird jeweils ein *Camera-Component* aktiviert, während die anderen deaktiviert werden.

Das folgende Skript kann über das Array *cameras* beliebig viele *Camera*-Komponenten aufnehmen und schaltet diese über die Taste **C** der Reihe nach durch. Bei diesem Skript wird davon ausgegangen, dass jedes *Kamera-GameObject* eine eigene *AudioListener*-Komponente besitzt, weshalb diese parallel mitgeschaltet werden. Mehr zu diesem Thema erfahren Sie im Kapitel „Audio“.

**Listing 6.3** Kamerawechsel-Skript

```
using UnityEngine;
using System.Collections;
public class CamSwitcher : MonoBehaviour {

    public Camera[] cameras; //Kamera-Array
    private int currentIndex = 0; //Index der aktiven Kamera
    private int count = 0; //Anzahl der zugewiesenen Kameras

    void Start () {
        count = cameras.Length; // Zuweisen der Kameraanzahl
        Switch(0); //Als Erstes wird Kamera 0 eingeschaltet, die restlichen aus.
    }
    void Update () {
        //Wenn c gedrückt wird, wird die nächste Kamera eingeschaltet
        if (Input.GetKeyDown(KeyCode.C))
        {
            currentIndex++;
            //Wenn der Index zu gross ist, dann wieder auf 0 setzen
            if (currentIndex >= count)
                currentIndex = 0;
            Switch(currentIndex);
        }
    }
    void Switch(int activeIndex)
    {
        //Durchlaufen aller Kameras
        for(int i = 0; i < count; i++)
        {
            //Alle Kameras deaktivieren, die nicht den Index activeIndex haben
            if (i != activeIndex)
            {
```

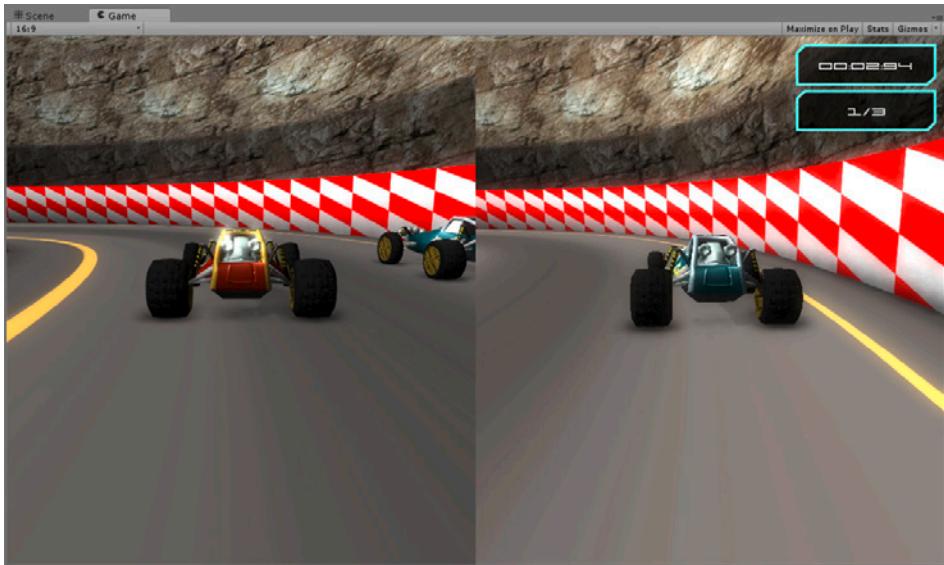
```

        //Kamera-Komponente deaktivieren
        cameras[i].enabled = false;
        //Auf das GameObject der Kamera zugreifen und dessen
        //AudioListener deaktivieren
        cameras[i].gameObject.GetComponent<AudioListener>().enabled = false;
    }
    else
    {
        //Die Kamera mit dem activeIndex aktivieren
        cameras[i].enabled = true;
        //Auf das GameObject der Kamera zugreifen und dessen
        //AudioListener aktivieren
        cameras[i].gameObject.GetComponent<AudioListener>().enabled = true;
    }
}
}

```

## 6.4.2 Split-Screen

Eine beliebte Technik bei PC- und Konsolen-Games, wo zwei Spieler gegeneinander spielen, ist die Split-Screen-Technik. Hierbei wird der Bildschirm in zwei Hälften aufgeteilt, wobei jede Seite die Sicht des einen Spielers auf das Game darstellt. Auf diese Weise können Sie auch mit nur einem Monitor z. B. Rennduelle oder Sport-Games wie Fußball realisieren.



**Bild 6.4** Split-Screen eines Rennspiels

Über die *Normalized View Port Rect*-Eigenschaft der *Camera*-Komponente können Sie kinderleicht solche Split-Screens darstellen. Um eine Bildschirmaufteilung wie in Bild 6.4 zu erhalten, platzieren Sie ein zweites Kamera-Objekt in Ihrer Szene und parametrisieren Sie die *Normalized View Port Rect*-Eigenschaften der beiden Kameras wie folgt:

#### ■ Kamera 1 (linkes Bild)

- **X 0.0** – Bild beginnt links
- **Y 0.0** – Bild beginnt unten
- **W 0.5** – Bild füllt die Hälfte der Breite aus
- **H 1.0** – Bild füllt die komplette Höhe aus

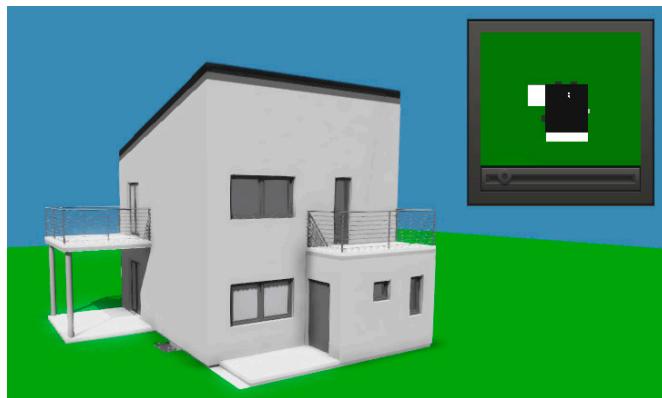
#### ■ Kamera 2 (rechtes Bild)

- **X 0.5** – Bild beginnt in der horizontalen Mitte
- **Y 0.0** – Bild beginnt unten
- **W 0.5** – Bild füllt die Hälfte der Breite aus
- **H 1.0** – Bild füllt die komplette Höhe aus

Damit die beiden Spieler bei solchen Games nun auch verschiedene Fahrzeuge bzw. Figuren im Spiel steuern können, müssen diese natürlich auch über unterschiedliche Eingaben angesteuert werden, was ich aber im Abschnitt „Steuerungen bei Mehrspieler-Games“ des Kapitels „Maus, Tastatur, Touch“ noch näher erläutern werde.

### 6.4.3 Einfache Minimap

Als Minimaps bezeichnet man kleine Übersichtskarten, die sich meistens in einer Ecke des Bildschirms befinden und das Spiel von oben darstellen. Meistens werden Sie diese in Rollenspielen oder auch in Rennspielen finden. Unity bietet hierfür gleich mehrere Ansätze, um solche Minimaps in einem Spiel zu integrieren.



**Bild 6.5**

Minimap in der Architekturvisualisierungssoftware PLANLIFE

Eine einfache Umsetzung, die auch mit der kostenlosen Unity-Version möglich ist, knüpft an die Vorgehensweise des Split-Screens an, verlangt aber zusätzlich noch die Berücksichtigung des *Depth*-Wertes der *Camera*-Komponente.

Der Parameter **Depth** legt die Rendering-Tiefe der Kamera fest. Umso negativer der Wert ist, desto früher wird das Bild gezeichnet. Bilder von Kameras mit positiveren Werten werden später gezeichnet und überschreiben die bisherigen Kamerabilder. Dies ist bei der Minimap sehr wichtig, da dessen Bild über das eigentliche Bild der „Main Camera“ gelegt wird.

Erzeugen Sie hierfür zunächst eine zweite Kamera und positionieren Sie diese über dem Spielgeschehen. Diese drehen Sie nun um  $90^\circ$  um die X-Achse, damit diese senkrecht auf das Spielgeschehen schaut. Ob Sie diese nun statisch über der gesamten Szene positionieren, diese direkt über dem Spieler platzieren und sie als Kind-Objekt zufügen oder die Kamera über ein zusätzliches Skript an ein zu beobachtendes Objekt koppeln, ist Ihnen überlassen. Wichtig sind jetzt aber vor allem die folgenden *Normalized View Port Rect*-Eigenschaften:

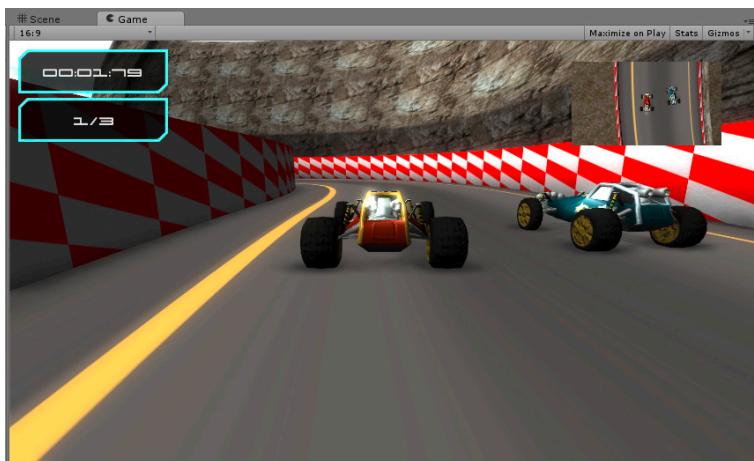
#### ▪ Spielkamera

- **X 0.0 -** Bild beginnt links
- **Y 0.0 -** Bild beginnt unten
- **W 1.0 -** Bild füllt die komplette Breite aus
- **H 1.0 -** Bild füllt die komplette Höhe aus
- **Depth -1.0 -** Bild wird als erstes gezeichnet.

#### ▪ Minimap Kamera

- **X 0.75 -** Beginnt z.B. ab  $\frac{3}{4}$  der Breite des Bildes
- **Y 0.75 -** Beginnt z.B. ab  $\frac{3}{4}$  der Höhe des Bildes
- **W 0.2 -** Kamerabildbreite beträgt  $1/5$  des Gesamtbildes
- **H 0.2 -** Kamerabildhöhe beträgt  $1/5$  des Gesamtbildes
- **Depth 0.0 -** Bild überzeichnet das Bild der normalen Spielkamera.

Mit den oben genannten Werten wird die Minimap oben rechts platziert, lässt aber zu den Seiten noch etwas Raum. Für gewöhnlich wird bei einer Minimap zudem noch der *Camera-Parameter Projection* auf *Orthographic* gestellt. Über den *Size*-Wert können Sie dann den dargestellten Ausschnitt des Spiels bestimmen. Das Bild 6.6 zeigt eine solche Minimap, dessen Kamera per Skript an den Spieler-Wagen gekoppelt wurde.



**Bild 6.6** Einfache Minimap

#### 6.4.4 Render Texture

Kamerabilder können nicht nur im *View Port* des Spiels, also auf dem Bildschirm dargestellt werden, sie können auch auf Texturen eines *Materials* projiziert werden. Hierfür benötigen Sie allerdings sogenannte *Render Textures*, die leider nur in Unity Pro zur Verfügung stehen.

Diese weisen Sie dann dem Kamera-Parameter *Target Texture* zu, dessen Kamerabild dann auf dieser angezeigt werden soll.

Mithilfe dieser speziellen Texturen können Sie beispielsweise Spiegelbilder auf einem Wandspiegel erzeugen, Monitore einer Überwachungskamera mit Leben füllen oder auch Reflexionen auf Wasseroberflächen realisieren.



Bild 6.7 Spiegelungen auf einer Wasseroberfläche

## ■ 6.5 Image Effects

Unity bietet die Möglichkeit, mit *Postprocessing*-Effekten das Gesamterscheinungsbild Ihres Spiels zu verändern und damit das Spiel grafisch noch interessanter zu gestalten. Die sogenannten „Image Effects“ sind dabei gewöhnliche Komponenten, die einfach den Kamera-Objekten zugefügt werden. Allerdings nutzen diese Effekte *Render Textures*, weshalb auch *Image Effects* nur in Unity Pro zur Verfügung stehen.

Ein *Image Effect* besteht vor allem aus einem Skript, in dem ein *Shader* eingebunden ist. Zur Laufzeit wird nun eine *Render Texture* erzeugt, die einem temporären Material zugewiesen wird. Der *Shader* wird schließlich ebenfalls dem Material zugewiesen, um dann den Effekt zu berechnen und das Zielbild zu erzeugen. Auf diese Weise können Sie Ihre Spielansicht

mit Effekten wie Bewegungsunschärfe, Fischaugen-Perspektiven und Ähnlichem ausstatten. Diese und viele weitere stellt Unity bereits in seinen *Standard Assets* „Image Effects“ der Pro-Version bereit.

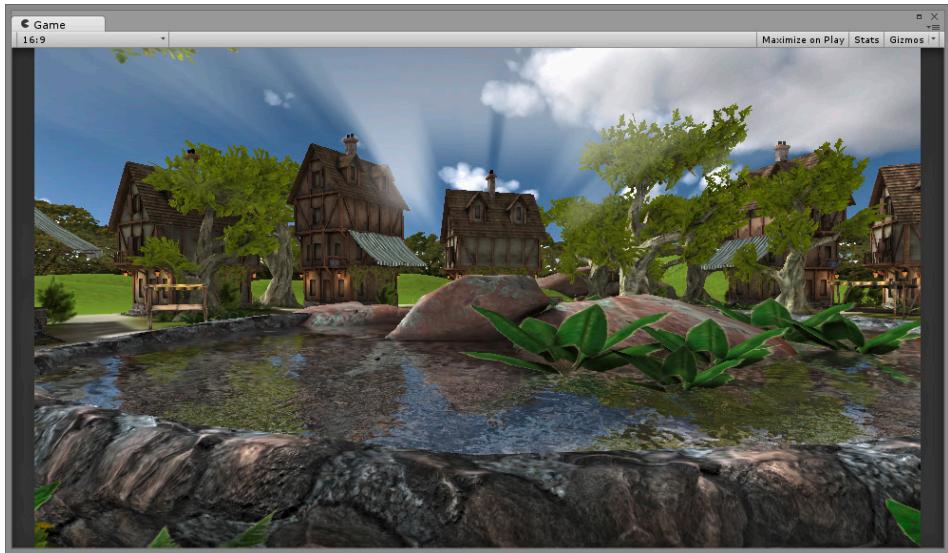


Bild 6.8 Image Effect „Sun Shafts“

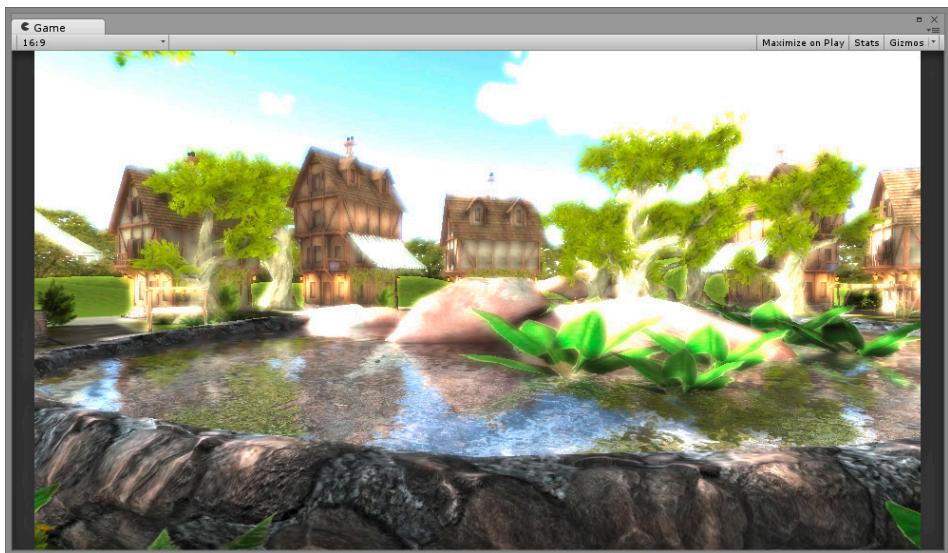
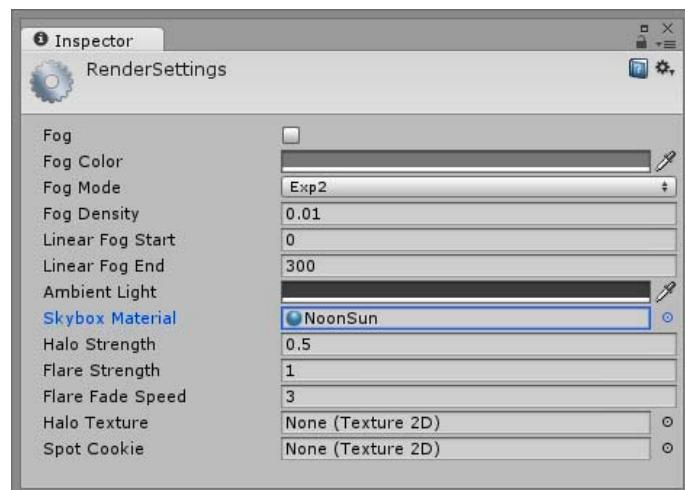


Bild 6.9 Image Effect „Glow“

## ■ 6.6 Skybox

Eine *Skybox* ist ein besonderes *Material*, das überall dort zu sehen ist, wo keine *GameObjects* vorhanden sind, also normalerweise am Himmel. Das Besondere dieses *Materials* ist der *Skybox-Shader*, der sechs unterschiedlichen Texturen nutzt. Alle Texturen des *Materials* bilden zusammen ein 360°-Panorama, das am Ende wie eine Art Hülle der Szene wirkt. Die Zuweisung der zu nutzenden *Skybox* machen Sie in den *Render Settings* (*Edit/Render Settings*), wo Sie diese dem Parameter *Skybox Material* zuweisen.



**Bild 6.10**

Render Settings mit dem Skybox-Material

Technisch wird dieser Effekt dadurch möglich, dass die *Skybox* vor allen anderen Objekten zuallererst gerendert wird (siehe *Camera*-Parameter *Clear Flags*). Alle anderen *GameObjects* werden nachträglich gezeichnet und überzeichnen die *Skybox*, sodass am Ende die *Skybox* eben nur noch dort zu sehen ist, wo kein *Mesh* ist.

Wenn Sie die *Skybox* zur Laufzeit eines Spiels ändern möchten, können Sie dies über die Klasse *RenderSettings* machen, die hierfür die statische Variable *skybox* bereitstellt.

### Listing 6.4 Skybox austauschen

```
public Material skyboxGreen;
void Awake () {
    RenderSettings.skybox = skyboxGreen;
}
```

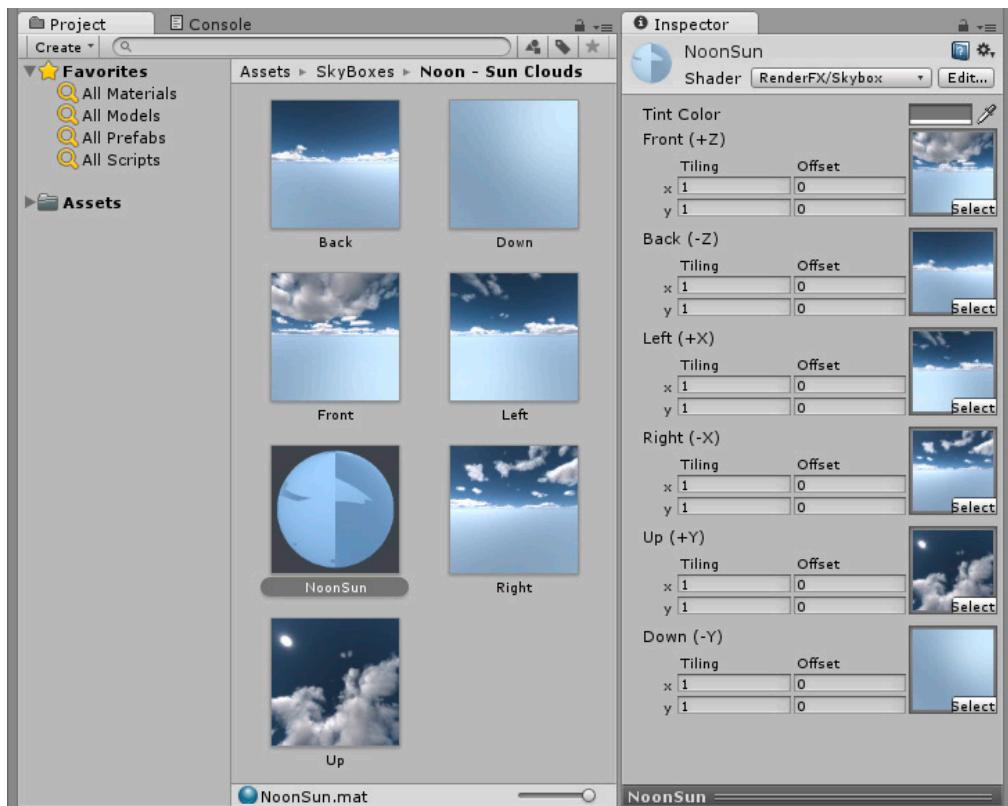
### 6.6.1 Skybox selber erstellen

Unity liefert in seinen *Standard Assets* „*Skyboxes*“ eine ganze Palette an fertigen *Skyboxes* mit, die für den Anfang reichen sollten. Wenn Ihnen diese nicht reichen, können Sie sich auch selber welche erstellen. Hierfür benötigen Sie sechs unterschiedliche Bilder, die zu-

sammen ein 360°-Panorama bilden. Dabei müssen diese natürlich nahtlos aneinander passen können, damit später keine grafischen Fehler am Horizont zu sehen sind.

1. Importieren Sie die sechs Bilder in Ihr Projekt.
2. Wählen Sie in den *Import Settings* der Bilder den *Wrap Mode Clamp*.
3. Erstellen Sie ein neues *Material*.
4. Weisen Sie dem Material den *Shader RenderFX/Skybox* zu.
5. Weisen Sie die sechs *Skybox*-Bilder dem *Material* zu.

Anschließend können Sie die neue *Skybox*, wie oben beschrieben, in den *Render Settings* dem Spiel zuweisen.



**Bild 6.11** Skybox-Material mit Bildern

# 7

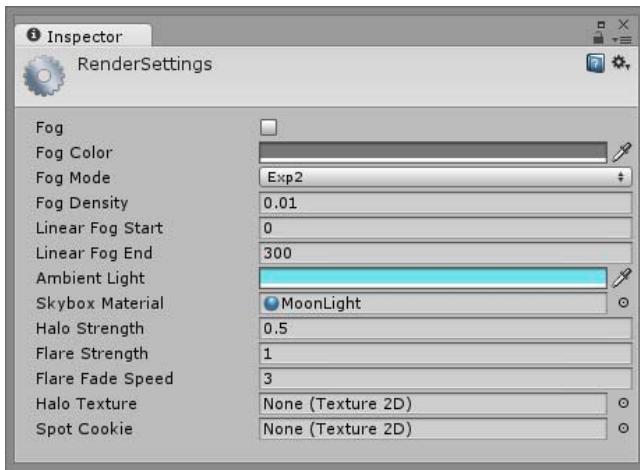
# Licht und Schatten

Damit eine Kamera die Textur eines *Mesh* und damit auch dessen Form darstellen kann, benötigt Ihre Szene zunächst einmal Licht. Licht hat wiederum einen enormen Einfluss auf die Darstellung der Textur, man denke nur an die Helligkeit, Position und Ausrichtung der Lichtquelle.

Unity bietet hierfür verschiedene Arten von sogenannten *Light*-Objekten an, die z.B. Echtzeitschatten erzeugen und mit anderen Effekten ausgestattet werden können. Neben dieser Echtzeitbeleuchtung unterstützt Unity auch *Lightmapping*, ein Verfahren, mit dem Sie Texturen generieren können, die beleuchtete Flächen vortäuschen. Da Lichtberechnungen sehr rechenintensiv sind, kann durch das Nutzen von *Lightmapping* die Performance erheblich gesteigert werden.

## ■ 7.1 Ambient Light

Unity besitzt eine globale Beleuchtung namens *Ambient Light*, die die komplette Szene mit einer Grundhelligkeit ausstattet. Diese finden Sie in den *Render Settings* über **Edit/Render Settings**. Möchten Sie ein Spiel entwickeln, das keine Grundbeleuchtung besitzen soll, weil es beispielsweise im Dunkeln spielt, können Sie das *Ambient Light* einfach auf Schwarz stellen.

**Bild 7.1**

Render Settings

## ■ 7.2 Lichtarten

Über das Menü **GameObject/Create Other** können Sie vier unterschiedliche Beleuchtungsobjekte Ihrer Szene hinzufügen: *Directional Light*, *Point Light*, *Spot Light* und das *Area Light*. Letzteres nimmt eine Sonderrolle ein, worauf ich noch eingehen werde. Alle Objekte besitzen eine *Light*-Komponente, die das Herzstück einer Lichtquelle darstellt. Die Komponente besitzt unterschiedliche Parameter, die je nach *Type* der *Light*-Komponente zur Verfügung stehen. Die folgenden Parameter stehen aber mit Ausnahme des *Area Lights* immer zur Auswahl:

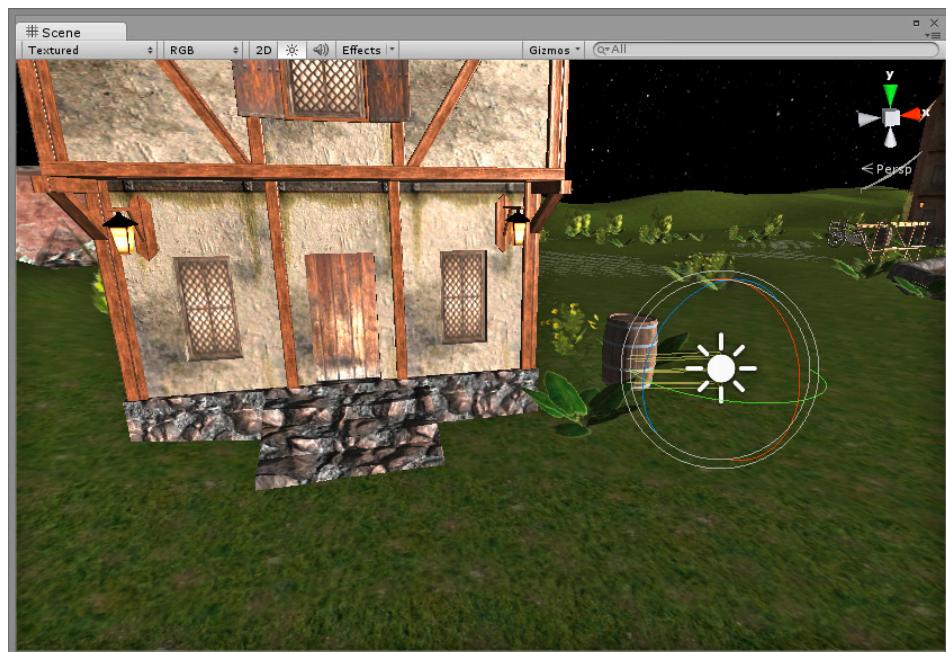
- **Type** bestimmt die Art des Lichtes und damit auch die zur Verfügung stehenden Parameter.
- **Range** setzt die Reichweite der Lichtquelle fest.
- **Intensity** definiert die Lichtstärke.
- **Color** legt die Lichtfarbe fest.
- **Cookie** ermöglicht, eine Lichtschablone vor eine Lichtquelle zu legen. Dies ist je nach *Type* entweder eine einzelne Schwarz-Weiß-Grafik oder eine *Cubemap*, bestehend aus sechs solcher Grafiken.
- **Shadow Type** legt die Art des Schattens fest. Allgemein stehen *No Shadows*, *Hard Shadows* und *Soft Shadows* zur Verfügung. Auf die Details werde ich gleich noch eingehen.
- **Draw Halo** bestimmt, ob ein *Light Halo* dargestellt werden soll.
- **Flare** legt ein *Flare*-Objekt zum Darstellen eines Lichtscheins für diese Lichtquelle fest (siehe „Flare“).
- **Render Mode** legt fest, ob dieses Licht beim *Forward Rendering* per Pixel oder per Vertex gerendert werden soll. *Auto*: Unity bestimmt, auf welche Art gerendert wird. *Important* bedeutet per Pixel, *Not Important* bedeutet per Vertex. Der Wert *Pixel Light Count* (zu fin-

den in den *Quality Settings*) bestimmt bei *Auto*, wie viele Lichtquellen insgesamt per Vertex gerendert werden können.

- **Culling Mask** definiert, welche Objekte eines *Layers* nicht von dieser Lichtquelle beeinflusst/beleuchtet werden.
- **Lightmapping** bestimmt das Verhalten beim Erstellen und Nutzen von *Lightmaps*. *RealtimeOnly* schließt dieses Licht beim Erstellungsprozess der *Lightmaps* (auch *baken* genannt) aus. *Auto* bindet dieses ein, schließt es in das *Rendering* ein, sobald der Spieler sich dem Licht nähert. *BakedOnly* bindet das Licht in das *Baken* ein, deaktiviert es aber während des normalen Spiels.

### 7.2.1 Directional Light

Ein *Directional Light* beleuchtet die komplette Szene aus einer Richtung. Die Position der Lichtquelle spielt dabei keine Rolle, nur die Rotation der Quelle ist hierbei wichtig. Ein *Directional Light* kann mit *Forward Rendering* (siehe Abschnitt „Rendering Paths“) sowohl in der Pro- wie auch in der Indie-Version Echtzeitschatten erzeugen. Allerdings unterstützt die Indie-Version nur den *Shadow Type Hard*, während die Pro-Version auch weiche Schatten, also mit weicheren Übergängen, unterstützt.



**Bild 7.2** Directional Light

## 7.2.2 Point Light

Das *Point Light* ist eine Lichtquelle, die in alle Richtungen gleichmäßig abstrahlt, vergleichbar mit einer Glühlampe an der Decke. Im Gegensatz zum *Directional Light* ist hier die Position, aber nicht die Rotation wichtig. Über den Parameter *Range* legen Sie die Reichweite der Lichtquelle fest. Echtzeitschatten von *Point Lights* werden nur in Kombination mit dem *Rendering Path Deferred Lighting* von Unity Pro unterstützt.

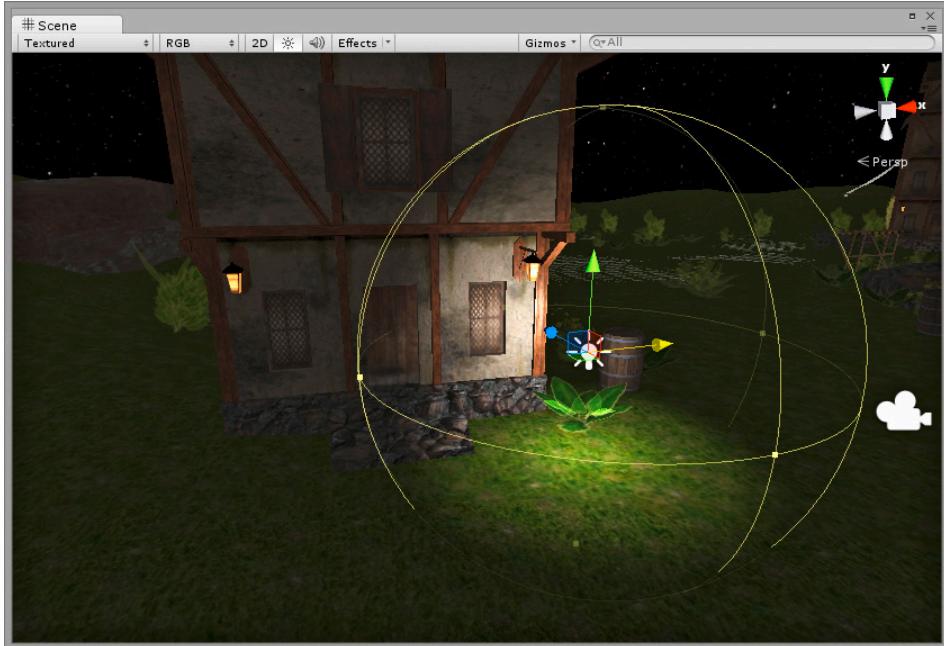


Bild 7.3 Point Light

## 7.2.3 Spot Light

Ein *Spot Light* scheint trichterförmig von der Lichtquelle in eine bestimmte Richtung, ähnlich wie eine Taschenlampe. Über den Parameter *Range* legen Sie fest, wie weit sie scheint. *Spot Angle* legt den äußeren Abstrahlwinkel fest. Echtzeitschatten von *Spot Lights* werden wie beim *Point Light* ebenfalls nur in Kombination mit dem *Rendering Path Deferred Lighting* von Unity Pro unterstützt.

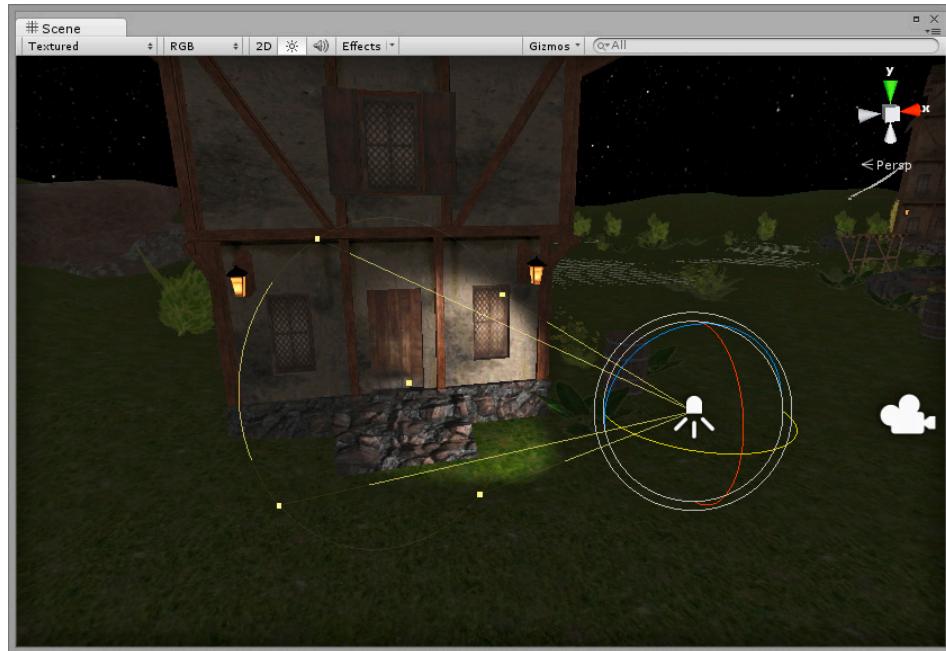


Bild 7.4 Spot Light

#### 7.2.4 Area Light

Ein *Area Light* nimmt eine Sonderstellung bei den *Light Types* ein, da es ausschließlich beim Erstellen von *Lightmaps* berücksichtigt wird, nicht aber während des normalen Spielverlaufs. Ein *Area Light* erscheint beim *Lightmapping* wie eine Rechteckfläche, die in alle Rich-

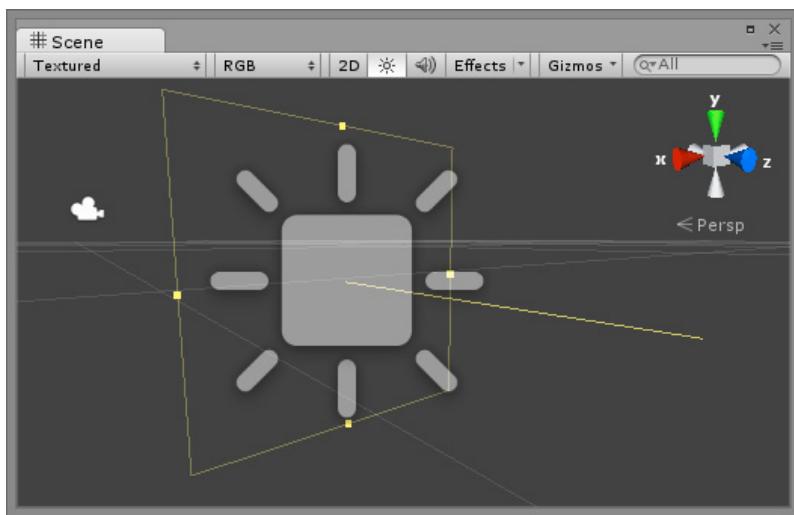


Bild 7.5 Area Light

tungen einer Seite scheint, vergleichbar mit dem Bildschirm eines Fernsehers. Die Richtung wird mit einer Linie dargestellt (siehe Bild 7.5). Die Fläche wird durch die Parameter *Width* und *Height* festgelegt. Und auch hier wird die Reichweite der Beleuchtung über die *Intensity* definiert.

## ■ 7.3 Schatten

Unity bietet zwei unterschiedliche Echtzeitschattenarten an: *Hard* und *Soft Shadows*. *Hard Shadows* sind eine performance-schonende Variante, *Soft Shadows* sind dafür detaillierter. Zudem können Sie einer Lichtquelle auch den Parameter *No Shadows* mitgeben. In dem Fall werfen angestrahlte Objekte überhaupt keine Schatten.



**Bild 7.6** Vergleich: Hard Shadows vs. Soft Shadows

Unity Pro unterstützt bei allen Lichtquellen beide Schattenarten, die kostenlose Version bietet nur beim *Directional Light* die Art *Hard Shadows* an. Neben der Schattenart bieten die Lichtquellen noch weitere Parameter an.

- **Strength** legt die Dunkelheit des Schattens fest.
- **Resolution** definiert die Qualität bzw. die Auflösung des Schattens. Standardmäßig wird hier für die Einstellung auf die *Quality Settings* (*Edit/Project Settings/Quality*) verwiesen. Sie kann aber auch überschrieben werden.
- **Bias** hat einen Einfluss auf den Abstand des Objektes zum Schatten. Beginnt der Schatten zu nah am Objekt, kann dies zu optischen Fehlern führen. Deshalb ist standardmäßig ein Wert von 0,05 vorgegeben.
- **Softness** ist ein Zusatzparameter für *Directional Lights*. Sie definiert, wie weich die Übergänge der Schattenkanten von *Soft Shadows* sein sollen.
- **Softness Fade** ist ebenfalls ein *Directional Lights*-Parameter. Dieser Wert definiert den Übergang des Schattens abhängig von der Entfernung der Kamera.

### 7.3.1 Einfluss des MeshRenderers auf Schatten

Über die *MeshRenderer*-Komponente eines jeden sichtbaren Objektes haben Sie die Möglichkeit zu steuern, ob ein Objekt überhaupt Schatten erzeugen soll oder nicht. Dies können Sie über die Eigenschaft **Cast Shadows** steuern. Genauso können Sie über die Eigenschaft **Receive Shadows** definieren, ob auf dem Objekt selber Schatten anderer Objekte dargestellt werden sollen.

Wenn Sie beispielsweise einem Spieler einen Unsichtbarkeitszauber zuführen, darf dieser selber keine Schatten werfen, aber auch keine Schatten darstellen, die andere Objekte auf ihn werfen. In diesem Fall wird der Schatten einfach zum nächsten Objekt „durchgeleitet“, sodass dort der Schatten dargestellt wird.

Das Bild 7.7 zeigt zwei Fässer. Während beim linken Fass beide Parameter aktiv sind und dieses einen Schatten wirft, wurden beim rechten Fass beide Parameter deaktiviert. Der Schatten des linken Objektes wird deshalb nicht auf dem rechten Fass dargestellt. Zudem wirft das rechte Fass auch selber keinen Schatten, sodass nur der Schattenwurf des linken gezeigt wird.

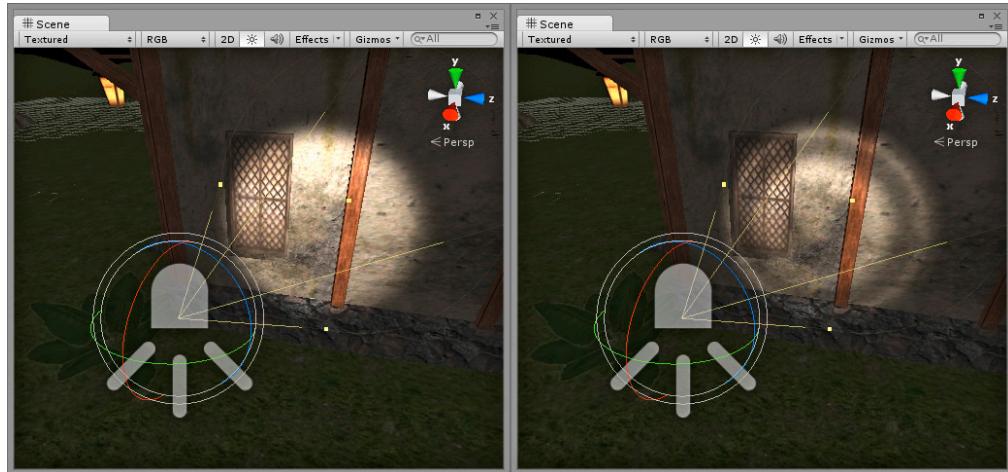


**Bild 7.7** Einfluss des „Cast Shadows“- und „Receive Shadows“-Parameter

## ■ 7.4 Light Cookies

Bei einem *Light Cookie* handelt es sich um eine Lichtschablone, die das abgegebene Licht in bestimmte Formen bringt. Wenn Sie ein Comic-Fan sind, dann kennen Sie sicher das Batman-Zeichen, dass die Polizei von Gotham-City an den Himmel wirft, um Batman zu rufen. Genau das ist ein *Light Cookie*, genauer gesagt ein *Spot Light* mit einem *Light Cookie*.

Das Bild 7.8 zeigt Ihnen einen ähnlichen Effekt. Dort wurde im rechten Motiv ein *Spot Light* mit einem *Light Cookie* versehen, das einen taschenlampenähnlichen Effekt erzeugt. Diesen wie auch weitere *Light Cookies* liefert Unity in seinen *Standard Assets „Light Cookies“* mit.



**Bild 7.8** Spot Light mit einem taschenlampenähnlichen Light Cookie

#### 7.4.1 Import Settings eines Light Cookies

Der Kern eines *Light Cookies* ist eine Schwarz-Weiß-Grafik, auch *Grayscale Texture* genannt. Schwarz bedeutet hierbei keine Lichtdurchlässigkeit, Weiß lässt das komplette Licht durch. Wichtig beim Import der Textur ist das Aktivieren von *Alpha from Grayscale*. Weiter spielt der *Wrap Mode* der Textur eine wichtige Rolle.

- Bei *Spot Lights* wird häufig der *Wrap Mode* auf *Clamp* gestellt, um auf diese Weise nur eine Abbildung des *Cookies* zu erhalten.
- Bei *Directional Lights* wird häufig der *Wrap Mode* auf *Repeat* gestellt, um so das *Cookie*-Motiv zu wiederholen. Ein *Directional Light* bietet hierfür noch den zusätzlichen Parameter *Cookie Size* an, der die Größe eines einzelnen Motives festlegt.

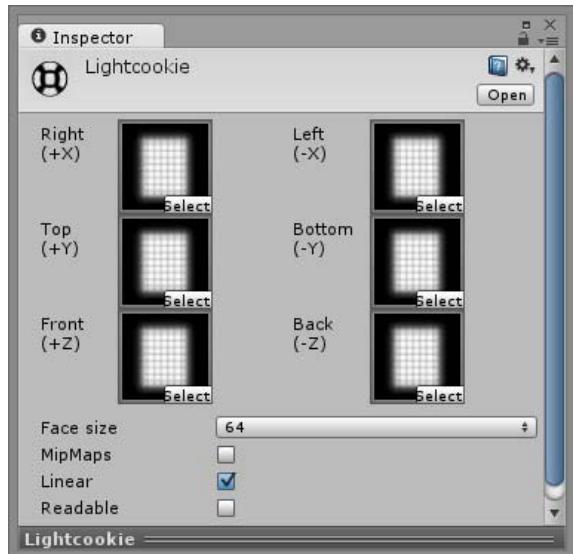
#### 7.4.2 Light Cookies und Point Lights

Da *Point Lights* in alle Richtungen strahlen, reicht es nicht aus, eine einfache Textur als *Cookie* zu nutzen. Hierfür werden *Cubemaps* genutzt – ein Würfel, bestehend aus sechs Texturen, in dessen Mitte sich die Lichtquelle befindet.

Eine *Cubemap* erzeugen Sie über **Asset/Create/Cubemap** oder über das Menü des Project Browsers.

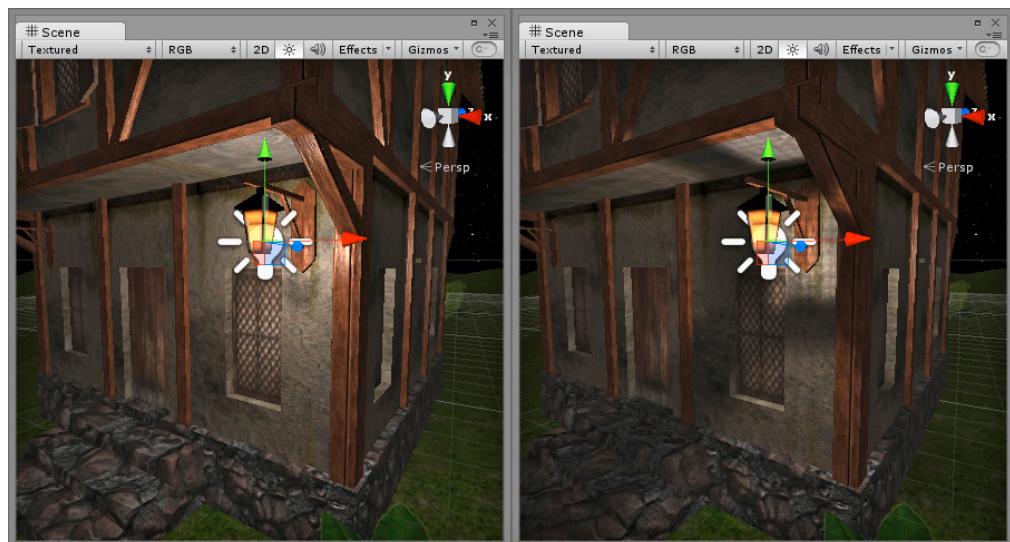
Den *Texture-Slots* werden nun je nach Effektwunsch verschiedene oder gleiche *Cookie*-Texturen zugewiesen. Anschließend wird die *Cubemap* dann dem *Cookie-Slot* der *Light*-Kompo-

nente vom *Point Light* zugewiesen, und schon haben wir auch dort den *Light Cookie*-Effekt, allerdings dieses Mal in alle Richtungen.

**Bild 7.9**

Beispiel eines Cubemaps Light Cookie

Das Bild 7.10 zeigt den Vergleich eines *Point Lights* ohne und mit einem *Light Cookie*.

**Bild 7.10** Einsatz eines Light Cookies bei einem Point Light

## ■ 7.5 Light Halos

Ein *Light Halo* ist ein Lichtschleier, der in der realen Welt durch Staubpartikel in der Luft entsteht. Da in Unity natürlich kein Staub vorhanden ist, wird dieser eben durch ein solches *Halo* simuliert. Über die *Light*-Komponenten-Eigenschaft *Draw Halo* können Sie einen Standard-*Halo* aktivieren, der sich an der Lichtstärke (*Intensity*), der Lichtfarbe (*Color*) sowie an der Reichweite (*Range*) der *Light*-Komponente orientiert.

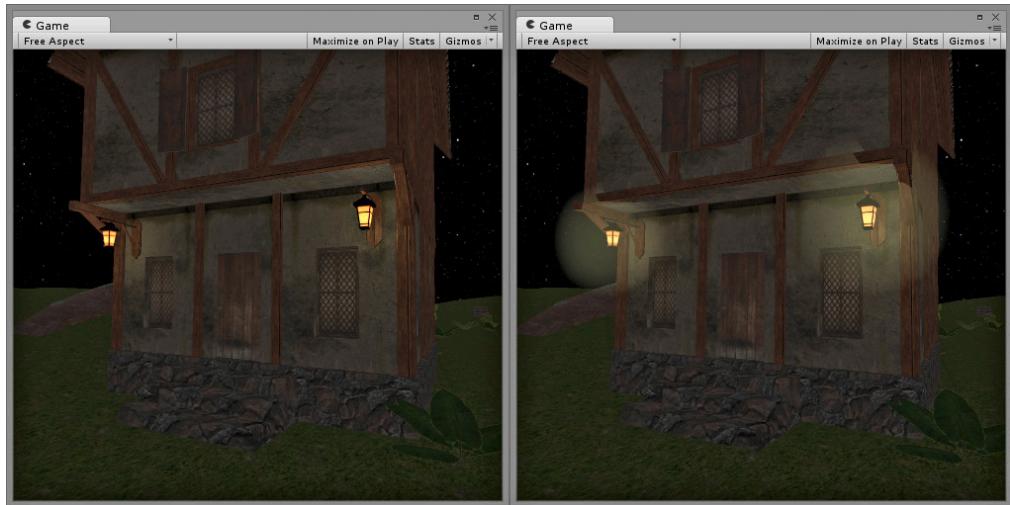


Bild 7.11 Light Halos

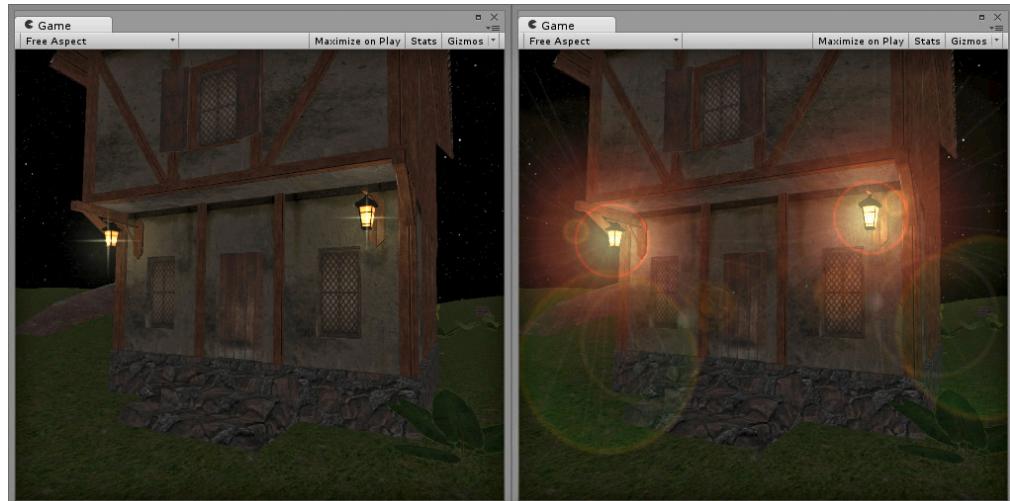
Das Bild 7.11 zeigt den Vergleich zweier *Point Lights* mit deaktiviertem und aktiviertem *Draw Halo*-Parameter.

### 7.5.1 Unabhängige Halos

Sie können neben den Standard-*Halos*, die Sie an den *Light*-Komponenten aktivieren können, auch unabhängige *Halos* erzeugen. Hierfür fügen Sie einem beliebigen *GameObject* über **Component/Effects/Halo** (oder über **Add Component** im *Inspector*) eine *Halo*-Komponente zu. Dies kann auch das gleiche Objekt sein, das bereits eine *Light*-Komponente besitzt. Der Unterschied ist nur, dass sich dieses *Halo* nicht an den Eigenschaften der *Light*-Komponente orientiert, sondern frei konfigurierbar ist.

## ■ 7.6 Lens Flares

*Lens Flares* simulieren Linsenreflexionen, also Effekte, die bei Gegenlicht in Kameralinsen entstehen. Das Bild 7.12 zeigt zwei unterschiedliche *Lens Flare*-Effekte, die Unity bereits in den *Standard Assets* „Light Flares“ bereitstellt. Das linke Motiv zeigt einen kleinen Effekt, der sich lediglich an der Lichtquelle selber zeigt, das rechte Teilbild zeigt einen sehr ausgeprägten Effekt, der auch verschobene Linseneffekte erzeugt.



**Bild 7.12** Lens Flares „Small Flare“ und „50 mm Zoom“

Zum Nutzen von *Lens Flares* sind mehrere Dinge wichtig. Zunächst benötigen Sie ein *Flare*-Objekt, das den eigentlichen Effekt und dessen Verhalten beschreibt (siehe *Standard Assets*). Da *Flare*-Objekte aber nicht alleine existieren können, müssen Sie diese nun einem *GameObject* zuweisen. Dies können Sie entweder über die *Flare*-Variable einer *Light*-Komponente machen oder aber über eine separate *Lens Flare*-Komponente, die Sie einem beliebigen *GameObject* zuweisen können. Diese finden Sie über **Component/Effects/Lens Flare**.

Als Letztes muss die Kamera noch eine *Flare Layer*-Komponente besitzen. Standardmäßig ist dies aber der Fall (siehe Kapitel „Kameras, die Augen des Spielers“).

### 7.6.1 Eigene Flares

Sie können natürlich nicht nur die mitgelieferten *Flares* nutzen, sondern auch eigene erzeugen. Dies machen Sie über das Menü **Assets/Create** bzw. über die rechte Maustaste im *Project Browser*.

Dem *Flare*-Asset können Sie einen *Texture-Atlas* zuweisen und anschließend das Verhalten an sich festlegen. Ein *Texture-Atlas* ist eine große Textur, die aus vielen kleinen Bildern besteht. Damit Unity weiß, an welcher Stelle sich ein Unterbild befindet, müssen diese *Flare*-

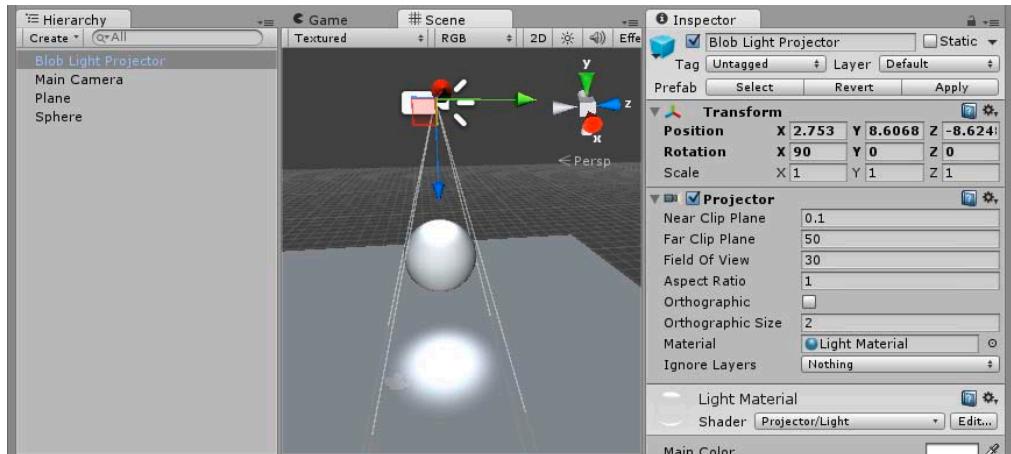
Texturen einen bestimmten Aufbau besitzen. Unity bietet hierfür sechs unterschiedliche Layouts an. Über die *Texture Layout*-Eigenschaft teilen Sie schließlich dem *Flare*-Objekt mit, welchen Aufbau Sie auf der Textur nutzen. Details erfahren Sie über den Hilfe-Button oben rechts im *Inspector* des *Flare*-Objekts.

## ■ 7.7 Projector

Eine weitere Möglichkeit, Licht und Schatten zumindest optisch darzustellen, sind sogenannte Projektoren bzw. *Projectors*. Wie der Name schon vermuten lässt, arbeitet dieser wie ein Beamer (oder ein Tageslichtprojektor oder DIA-Projektor), der ein Bild bzw. Material abstrahlt. Alle Objekte, die diesen Projektionskegel schneiden, werden dann mit diesem Material überlagert. Hierdurch können sehr interessante Effekte erzielt werden.

### 7.7.1 Standard Projectors

In den *Standard Assets* gibt es unter anderem einen *Blob Light Projector*, der eine weiße, kreisförmige Textur auf die angestrahlten Objekte projiziert. Dies wirkt wie ein Lichtkegel, nur dass es von der Berechnung her eben kein Licht, sondern nur eine halbtransparente Textur ist. Objekte können also auch keine Schatten werfen.



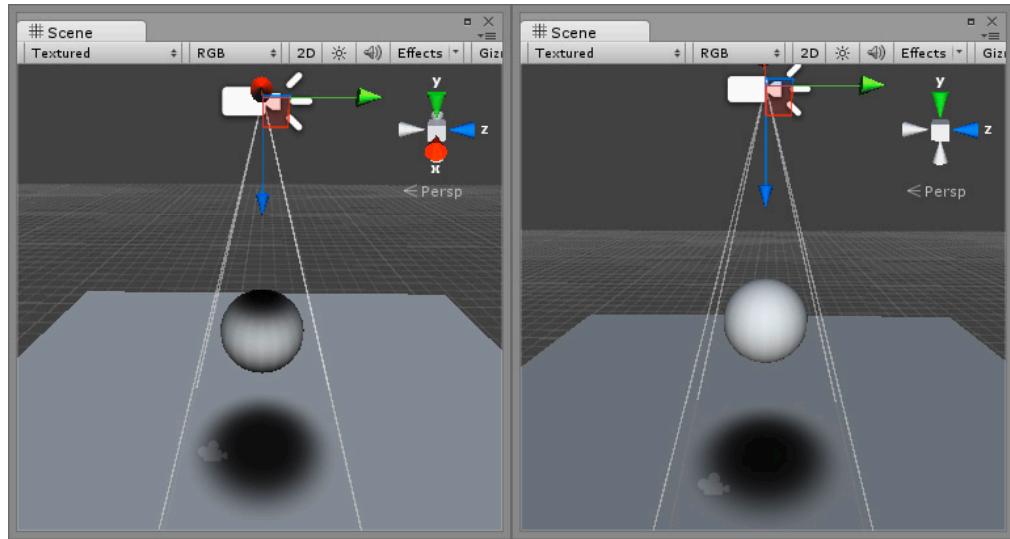
**Bild 7.13** Blob Light Projector

Als Gegenstück gibt es ebenso auch noch den *Blob Shadow Projector*. Dieser wirft keine helle Textur auf die Objekte, sondern eine schwarze. Platzieren Sie diesen *Projector* über einem Objekt und strahlen Sie auf diesen hinab, können Sie damit einen Schatten simulieren.

Hierfür müssen Sie dem schattenwerfenden Objekt einen Layer zuweisen, den Sie in der *Ignore Layers*-Eigenschaft des *Projectors* hinterlegen. Hierdurch wird die schwarze Textur

nicht mehr auf diesem Objekt, sehr wohl aber auf dem Untergrund angezeigt (siehe Bild 7.14). Am besten ist es natürlich, wenn Sie dem *Material*, das dem *Projector* zugewiesen wird, eine Textur zuweisen, die der Form des Objektes entspricht.

Damit der Schatten sich nun auch mit dem Objekt mitbewegt, empfiehlt es sich, den *Projector* als Kind-Objekt dem schattenwerfenden Objekt zuzuweisen – in Bild 7.14 wäre das die Kugel.



**Bild 7.14** Blob Shadow Projector

## ■ 7.8 Lightmapping

Licht- und Schattenberechnungen sind sehr rechenintensiv. Und umso detaillierter diese dargestellt werden sollen, desto mehr muss gerechnet werden. Um trotzdem in einem Spiel sehr detaillierte Schatten und Lichtszenarien zu erhalten, gibt es das sogenannte *Lightmapping*. Dieses Verfahren berechnet bereits zur Entwicklungszeit die Licht- und Schatteneffekte und erstellt Texturen, die dann über die Modelle gelegt werden. Danach können die Lichtquellen deaktiviert werden, und trotzdem wirken die Objekte, als würden diese angestrahlt werden.

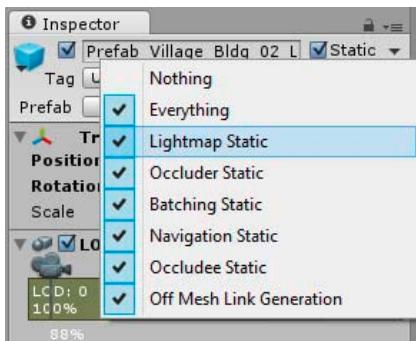
Damit dieses Verfahren funktioniert, müssen die 3D-Modelle das Überlagern einer weiteren *Lightmapping*-Textur unterstützen. Hierfür müssen Sie in den *Import Settings* der einzelnen Modelle die Eigenschaft *Generate Lightmap UVs* aktivieren.

Ein wichtiger Punkt beim *Lightmapping* ist der, dass nur Objekte berücksichtigt werden dürfen, die sich nicht bewegen. Der Grund hierfür ist ganz einfach: Stellen Sie sich vor, Sie berechnen den Schatten eines Autos und brennen dessen Schatten quasi in die Textur der Straße ein. Nun fährt das Auto weg und die Straßentextur mit dem Schatten bleibt an der

gleichen Stelle. Dies ist natürlich nicht gerade das, was man als realistisch bezeichnen würde. Deshalb berücksichtigt Unity beim *Lightmapping* nur Objekte, die statisch sind, also Objekte, die sich nicht bewegen.

Zum Markieren statischer Objekte besitzen alle *GameObjects* in einer Szene eine kleine Checkbox oben rechts im *Inspector* mit dem Namen *Static*. Setzen Sie diesen Haken bei allen Objekten, die sich nicht bewegen und beim *Lightmapping* berücksichtigt werden sollen.

Da sich mittlerweile mehrere Funktionen dieser *Static*-Eigenschaft bedienen, können Sie über den zusätzlichen Pfeil an der rechten Seite der *Static*-Checkbox ein weiteres Menü aufklappen. Hier können Sie definieren, für welche Funktionen diese *Static* Eigenschaft gilt. Achten Sie darauf, dass hier *Lightmap Static* aktiviert ist.



**Bild 7.15**

„Lightmap Static“-Option im Static-Menü

Jetzt müssen Sie nur noch im *Lightmapping* Fenster, das Sie über *Window/Lightmapping* erreichen, die gewünschten Einstellungen wählen (einige Features sind nur in Unity Pro verfügbar) und den Erstellungsprozess, auch *Bake* genannt, über den Button **Bake Scene** starten.

Daraufhin werden alle Lichter und deren Auswirkungen in der Szene berechnet, deren *Lightmapping*-Eigenschaft auf *Auto* oder *BakedOnly* gestellt sind, und in Texturen umgewandelt. Hierbei werden zudem auch *AreaLight*-Objekte wie auch Objekte mit *Emissive Materials* berücksichtigt.



### Emissive Materials

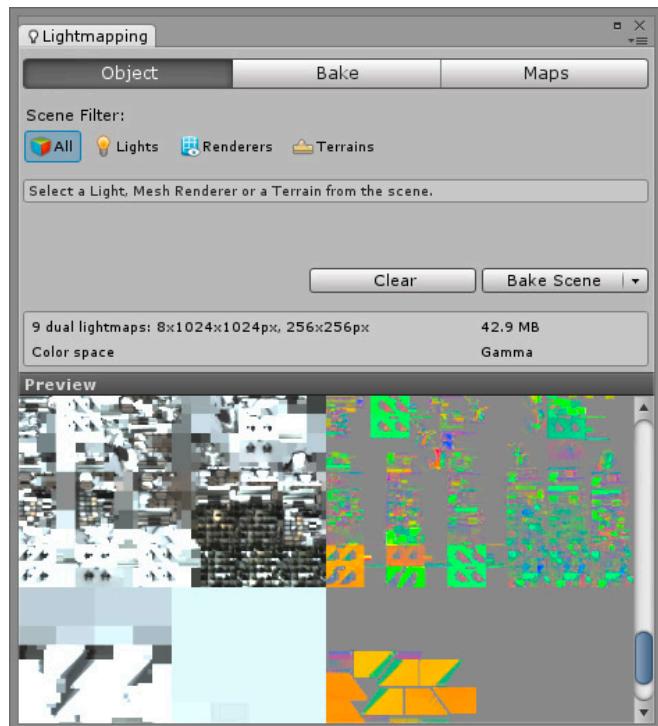
Als *Emissive Materials* werden Materialien bezeichnet, die einen *Shader* aus der „Self-Illuminated“-Familie nutzen. Das sind *Shader*, die Licht auf sich selber darstellen können, ohne dass eine Lichtquelle sie anstrahlt.

Für gewöhnlich werden diese *Shader* für Lichtquellen-Meshes wie Lampen, Laternen etc. genutzt. Denn wenn Sie z. B. eine Lampe im Raum platzieren, in dessen Zentrum eine Lichtquelle platziert wird, werden alle Objekte außer dem Lampenmodell angeleuchtet. Dies liegt daran, weil die sichtbaren Seiten des Lampen-Mesh von der Lichtquelle abgewandt sind und sich damit im Schatten der Lichtquelle befinden. Mit einem Self-Illuminated-*Shader* wird dieses Problem behoben.

Wenn Sie Lightmapping einsetzen, können Sie hier sogar auf die Lichtquelle mit dem zusätzlichen Light-Component verzichten. Die Lightmapping-Berechnung betrachtet Objekte mit diesen Shadern gleich selber als Lichtquellen und bindet sie in das Lightmapping mit ein.

Beachten Sie, dass *Emissive Materials* nur von Unity Pro unterstützt werden.

Da das Erstellen der *Lightmaps* je nach Einstellung und *Szenenaufbau* durchaus einige Zeit dauern kann, wird der Fortschritt des Vorgangs unten rechts als Balken angezeigt. Nach dem *Baken* werden die fertigen *Lightmaps* schließlich im *Preview*-Bereich des *Lightmapping*-Fensters angezeigt und in der Szene über die statischen Objekte gelegt.



**Bild 7.16**  
Lightmapping-Fenster  
mit Preview-Bereich

Sollten die erstellten Texturen noch nicht Ihren Ansprüchen genügen, können Sie die Texturen nach dem Baken auch in einem Grafikbearbeitungstool öffnen, das das OpenEXR-Format unterstützt, und diese dort weiter bearbeiten. Nach dem Speichern werden die Änderungen in Unity geladen und in der Szene dargestellt.



### Lightmapping-Beispiel

Auf der DVD finden Sie eine Video-Demonstration des *Lightmapping*-Verfahrens, in der die wichtigsten Parameter vorgestellt werden. Weiter werden auch *Emissive Materials* und das *Area Light* gezeigt.

## ■ 7.9 Rendering Paths

Um Licht und Schatten darzustellen, ist das Rendering ein wichtiger Punkt. Das Rendern berechnet hierbei das Bild, das am Ende auf dem Bildschirm des Spielers dargestellt wird.

Unity unterstützt hier gleich drei unterschiedliche *Rendering*-Techniken. Standardmäßig werden diese in den *Player Settings (Project Settings/Player)* hinterlegt. Das *Rendering*-Verfahren kann aber bei jeder Kamera noch einmal separat in der Eigenschaft *Rendering Path* eingestellt werden. Da hier aber per Default „Use Player Settings“ eingestellt ist, wird meistens die Einstellung aus den *Player Settings* genommen.

Ein Hauptunterschied dieser *Rendering*-Verfahren sind die eingesetzten Methoden zum Berechnen der Beleuchtung.

- Beim **Vertex Lighting** wird die Auswirkung einer Lichtquelle lediglich anhand der *Vertices* der beleuchteten 3D-Modelle bzw. der *Meshes* berechnet und auf die gesamten Flächen hochgerechnet.
- Beim **Pixel Lighting** wird die Beleuchtung für jeden einzelnen Pixel separat berechnet. Diese Vorgehensweise ist natürlich ressourcenintensiver, ermöglicht aber z.B. Normal-Mapping oder auch Echtzeitschatten. Allerdings sollten Sie beachten, dass einige ältere Grafikkarten *Pixel Lighting* nicht unterstützen.

### 7.9.1 Forward Rendering

*Forward Rendering* ist das Standard-*Rendering*-Verfahren in Unity. Da es unter anderem auch *Pixel Lighting Rendering* unterstützt, stellt dieses Verfahren auch *Shader*-Effekte wie *Normal-maps* sowie Echtzeitschatten dar, Letzteres allerdings nur von einem einzigen *Directional Light*, und zwar vom hellsten. Alle anderen Lichtquellen werden ohne Echtzeitschatten gerendert.

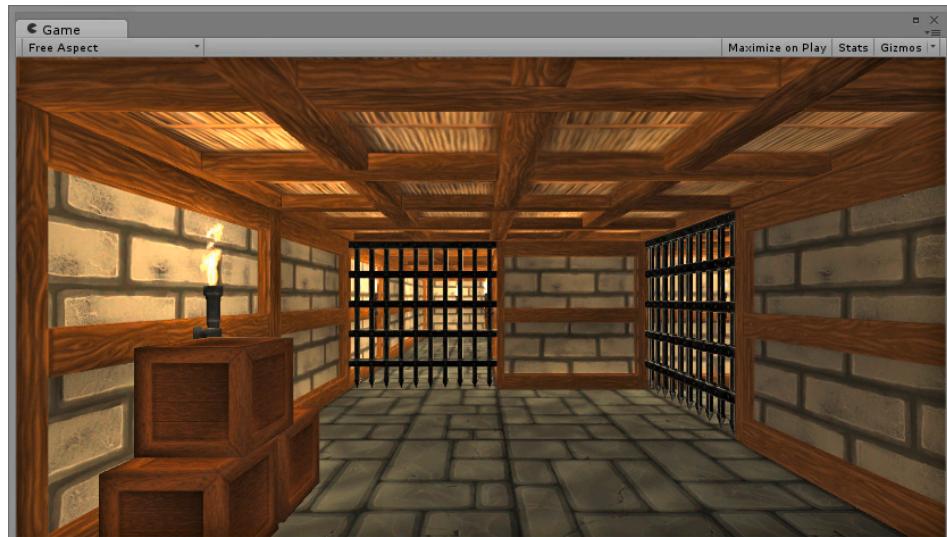


Bild 7.17 Mit Forward Rendering bearbeitete Szene

Per Default wählt Unity selbstständig die wichtigsten Lichtquellen aus und rendert diese im *Pixel-Lighting-Modus*. Wie viele Unity hierbei wählt, können Sie in den *Quality Settings* (**Edit/Project Settings/Quality**) über den Parameter *Pixel Light Count* festlegen. Die restlichen Beleuchtungen werden *Vertex-basiert* oder als *Spherical Harmonics Lights* berechnet. Letzteres ist ein Verfahren, das ebenfalls auf Basis von *Vertices* arbeitet und aufgrund von Näherungen sehr schnell berechnet werden kann.

### 7.9.2 Vertex Lit

*Vertex Lit* nutzt ausschließlich das *Vertex Lighting*. Es unterstützt keine Echtzeitschatten und auch keine *Shader*-basierten Effekte wie *Normalmaps*. Dafür ist es aber mit Abstand am performantesten und bietet die umfassendste Hardwareunterstützung.

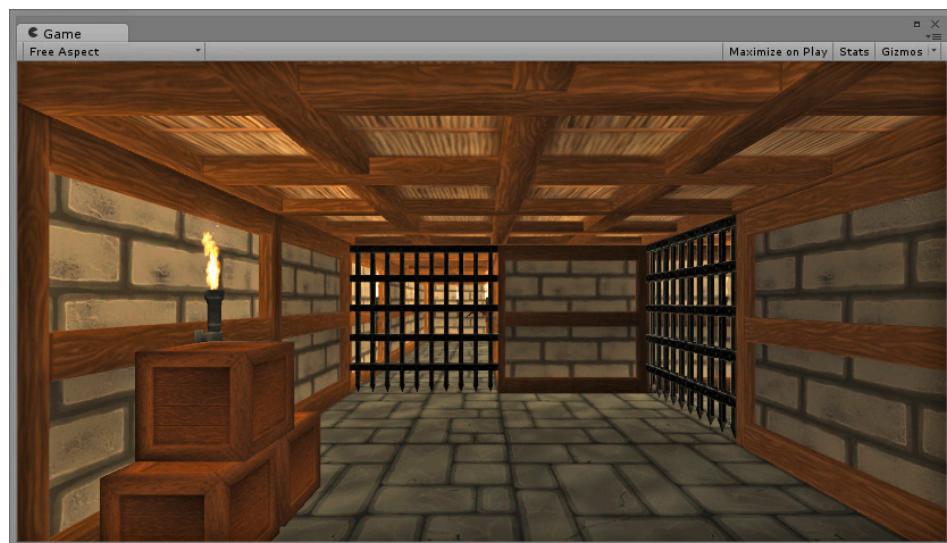


Bild 7.18 Gerenderte Szene mit Vertex Lit

### 7.9.3 Deferred Lighting

*Deferred Lighting* ist die anspruchsvollste Rendering-Art mit den detailreichsten Darstellungen von Schatten und Licht. Im Gegensatz zum *Forward Rendering* unterstützt es auch mehrere Lichtquellen mit Echtzeitschatten, was sich natürlich auch im Performance-Bedarf bemerkbar macht. Allerdings wird dieses Verfahren nur von Unity Pro unterstützt und damit nicht vom kostenlosen Unity Basic.

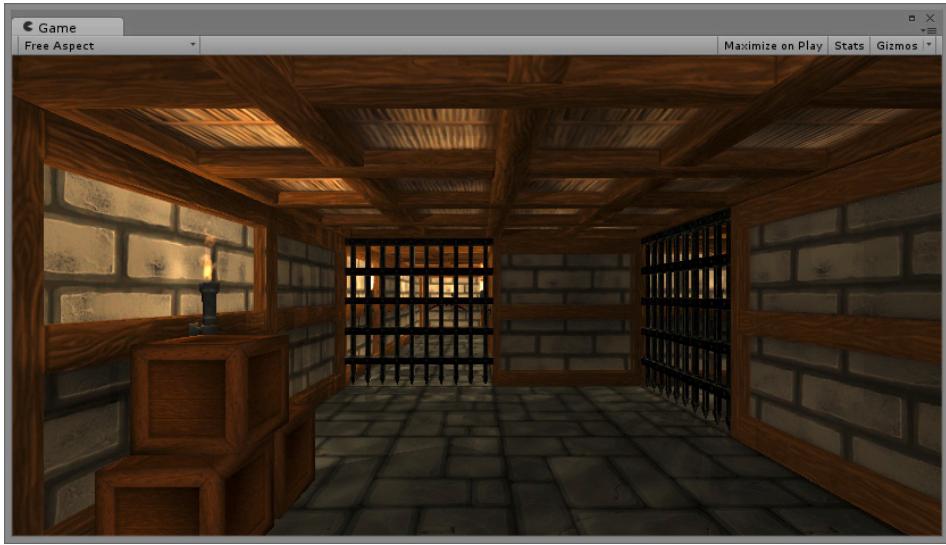


Bild 7.19 Gerenderte Szene mit Deferred Lighting

# 8

# Physik in Unity

Damit *GameObjects* miteinander interagieren können, benötigen diese physikalische Eigenschaften wie z. B. Massen, Kräfte und einiges mehr. Zum Simulieren dieser physikalischen Eigenschaften nutzt Unity die NVIDIA PhysX Physik-Engine.

Um diese auch nutzen zu können, liefert Unity nun eine ganze Reihe an Komponenten mit, die den *GameObjects* verschiedene physikalische Verhaltensweisen verleihen können. Beachten Sie, dass Unity die meisten der folgenden Komponenten immer in zwei Varianten anbietet: als Standardausführung und in einer 2D-Variante. Die normalen Komponenten erreichen Sie über **Component/Physics**, die 2D-Ausführungen erreichen Sie über **Component/Physics 2D**. Aus Vereinfachungsgründen stelle ich nur die Standardausführungen vor, die aber ebenfalls in 2D-Spielen eingesetzt werden können. Die 2D-Varianten sind lediglich für den zweidimensionalen Einsatz optimiert, weshalb auch die Parameter etwas limitierter sind.

## ■ 8.1 Physikberechnung

Die Berechnungen aller Physik-Eigenschaften finden regelmäßig in genau definierten Zeitabständen statt. Nachdem diese berechnet wurden, wird anschließend in allen Skripten, soweit vorhanden, die *FixedUpdate*-Methode aufgerufen. Das Intervall, in dem diese Kalkulationen stattfinden, wird über den *Fixed Timestep*-Parameter festgelegt und kann im *Time Manager* angepasst werden. Sie finden diesen im Hauptmenü **Edit/Project Settings/Time**. Umso kleiner *Fixed Timestep* ist, desto häufiger werden die Physik-Eigenschaften berechnet und aktualisiert. Diesen Wert können Sie im Übrigen auch im Programmcode über die Klasse *Time* und deren statischen Wert *fixedDeltaTime* abfragen.

**Listing 8.1** Ermittlung von *fixedDeltaTime*

```
using UnityEngine;
using System.Collections;
public class PrintFixedDeltaTime : MonoBehaviour {
    void Start ()
    {
```

```

        Debug.Log(Time.fixedDeltaTime);
    }
}

```

Gerade bei Spielen mit schnellen Objekten kann ein Senken von *Fixed Timestep* für ein besseres, realistischeres Verhalten sorgen. Ein Beispiel, wo dies sehr hilfreich ist, ist die Kollisionserkennung. Damit eine Kollision von der Physik-Engine erkannt werden kann, muss die Physikberechnung in dem Moment durchgeführt werden, in dem sich die Objekte auch tatsächlich berühren bzw. überschneiden. Bei schnellen Objekten führt dies dazu, dass die Berechnung recht häufig durchgeführt werden muss, da ansonsten die Kollision genau zwischen zwei Berechnungen stattfinden könnte und die Physik-Engine die Kollision einfach nicht mitbekommt. Ein Senken von *Fixed Timestep* hilft hierbei und senkt die Wahrscheinlichkeit, dass dies geschieht.

Auf der anderen Seite können Sie bei sehr langsamem Objekten wiederum das Intervall auch anheben, um so die Performance zu verbessern. Denn wenn die Physik seltener berechnet werden muss, sinkt natürlich auch die Prozessorbeanspruchung. Da das Ändern dieses Wertes aber Einfluss auf das gesamte Spiel hat, würde ich am Anfang auf solche Anpassungen verzichten und das Intervall auf dem Standardwert belassen.

## ■ 8.2 Rigidbodies

Die *Rigidbody*-Komponente ist der Kern der Unity-Physik. Jedes *GameObject*, das sich in einer Szene bewegt bzw. im Laufe des Spiels bewegt wird, muss ein *Rigidbody* besitzen.

Mithilfe eines *Rigidbody*s können Sie einem *GameObject* eine Kraft oder ein Drehmoment zufügen. Ein weiterer Punkt ist die Erdanziehungskraft. Wenn ein *GameObject* von einer virtuellen Gravitation angezogen werden soll, muss es ebenfalls ein *Rigidbody* besitzen. Außerdem baut die Kollisionserkennung in Unity auf *Rigidbodies* auf. Davon abgesehen sollten in Unity grundsätzlich alle örtlich veränderbaren Objekte *Rigidbodies* besitzen, dazu kommen wir aber noch später.

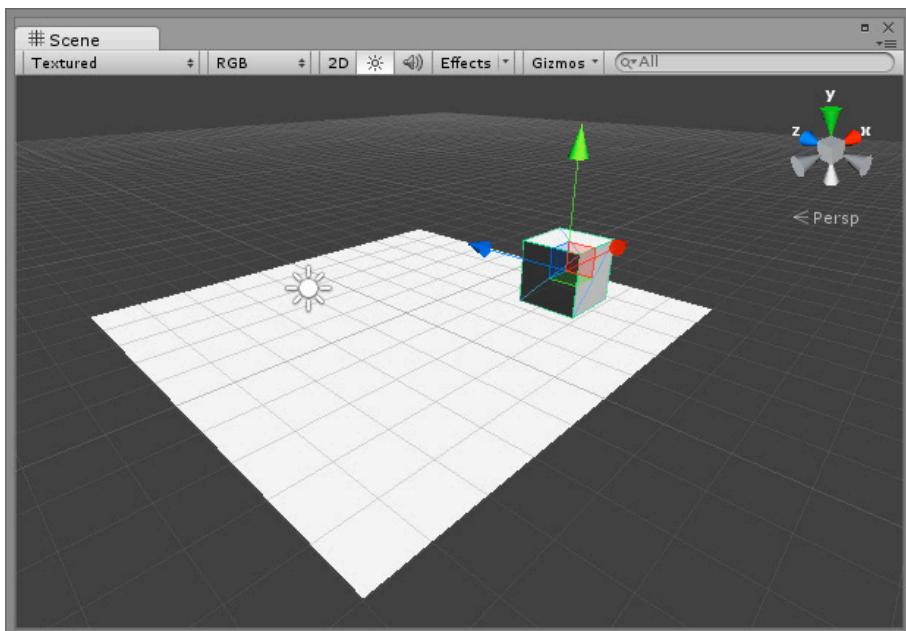
Im Folgenden möchte ich kurz die Parameter der *Rigidbody*-Komponente vorstellen:

- **Mass** legt die Masse des Objektes fest. Über ein Skript können Sie zudem das Zentrum der Masse, also den Schwerpunkt, definieren.
- **Drag** legt den Luftwiderstand fest, wenn das Objekt durch eine Kraftzuführung bewegt wird.
- **Angular Drag** legt den Luftwiderstand fest, wenn das Objekt durch eine Drehkraft zum Rotieren gebracht wird.
- **Use Gravity** aktiviert den Einfluss der Gravitation auf das Objekt.
- **Is Kinematic** unterbindet eine Beeinflussung durch die Physik-Engine, sodass das Objekt nur noch über die Eigenschaften und Methoden der *Transform*-Komponente bewegt werden kann. Trotzdem wird es in den Berechnungen der Physik-Engine berücksichtigt. Außerdem können Objekte mit diesem Parameter auch *OnTrigger*-Ereignisse auslösen.

- **Interpolate** ermöglicht einen weicheren Verlauf der Physik-Bewegungen. Da die Physikberechnungen in diskreten Schritten vorgenommen werden, das *Rendering* aber nicht, kann das zu Ruckeleffekten führen, was speziell bei Verfolger-Kameras auffällt, die dem Hauptcharakter folgen. Häufig wird dann für den Spielercharakter diese Eigenschaft aktiviert, für alle anderen Objekte mit *Rigidbodies* aber nicht.
- **Collision Detection** kann verhindern, dass schnelle Objekte durch andere *Collider* hindurchfliegen, ohne dabei erkannt zu werden. *Discrete* ist hierbei die Standardeinstellung. *Continuous* kann bei schnellen Objekten verwendet werden, die mit statischen *Collidern* kollidieren. Noch weiter geht *ContinuousDynamic*, das verwendet werden kann, wenn sich die anderen Objekte ebenfalls bewegen. In dem Fall müssen beide kollidierenden Objekte auf *ContinuousDynamic* oder *Continuous* eingestellt sein. Beachten Sie außerdem, dass die beiden letzteren Einstellungen nur von *Primitive Collidern* unterstützt werden.
- **Constraints** erlaubt Ihnen, Bewegungen und/oder Drehungen auf bestimmte Achsen einzuschränken. Dies macht z.B. bei Sidescrollern Sinn, wo ein Körper nur in die X und Y-Richtung bewegt werden darf, aber nicht in die Z-Richtung.

### 8.2.1 Rigidbodies kennenlernen

Um die Auswirkungen der verschiedenen *Rigidbody*-Parameter besser zu verstehen, eignet sich ein kleines Testszenario, das ich auch in den weiteren Abschnitten nutzen werde. Nutzen Sie hierfür das Übungsprojekt aus Kapitel 2 und ändern Sie die Position des Würfels auf die Koordinaten (0, 2, -4).



**Bild 8.1** Szenenaufbau zum Testen der Rigidbody-Parameter

Wenn Sie das Spiel nun starten, fällt der Cube zunächst auf die Plane. Stoppen Sie nun das Spiel und erhöhen Sie den *Drag*-Wert. Beim nächsten Start fällt der Cube durch den höheren Luftwiderstand langsamer. Auf diese Weise können Sie nun die ersten Tests machen und sich mit den unterschiedlichen Parametern vertraut machen. Übrigens, im Gegensatz zum *Drag*-Wert verändert der *Mass*-Wert nicht das Verhalten des Körpers beim Herunterfallen.

## 8.2.2 Masseschwerpunkt

Auch wenn die *Rigidbody*-Komponente im *Inspector* nicht das Definieren des Masseschwerpunktes zulässt, können Sie dies mithilfe eines kleinen Skriptes problemlos machen. Hierfür bietet die *Rigidbody*-Klasse die Eigenschaft *centerOfMass* an. Dieser weisen Sie eine Position relativ zum eigenen Nullpunkt zu. Die folgende Codezeile nutzt den Variablenzugriff, um auf die *Rigidbody*-Komponente des *GameObjects* zuzugreifen und dieser einen neuen Schwerpunkt zuzuweisen.

**Listing 8.2** Setzen des Masseschwerpunktes

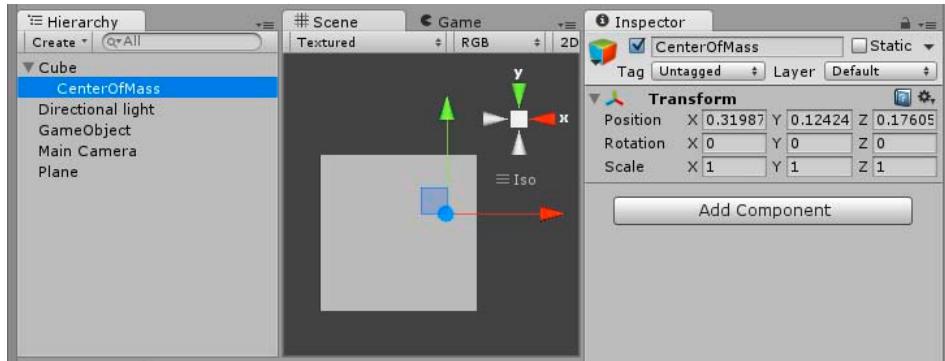
```
rigidbody.centerOfMass = new Vector3(0, -1, 0);
```

Eine gute Möglichkeit, diesen Punkt festzulegen, ist das Nutzen eines zusätzlichen *Empty Objects*. Je nach Unity-Version können Sie hierfür das eigentliche Hauptobjekt markieren (im Beispiel wäre das der Cube) und dann über **GameObject/Create Empty Child** diesem ein leeres *GameObject* zufügen oder aber über **GameObject/Create Empty** ein normales *GameObject* erzeugen, das Sie dem eigentlichen *GameObject* anschließend als zusätzliches Kind-Objekt zuweisen. Dieses nennen Sie dann beispielsweise „CenterOfMass“. Danach fügen Sie dem eigentlichen *GameObject* das folgende Skript zu und ziehen das neue „CenterOfMass“-*Empty Object* auf die *centerOfMass*-Variable.

**Listing 8.3** Masseschwerpunkt per Variable setzen

```
using UnityEngine;
using System.Collections;
public class SetCenterOfMass : MonoBehaviour {
    public Transform centerOfMass;
    void Start () {
        rigidbody.centerOfMass = centerOfMass.localPosition;
    }
}
```

Da in dem Skript die Position des *Empty Objects* in der *Start*-Methode der *centerOfMass*-Eigenschaft des *Rigidbody*s zugewiesen wird, bestimmt dieses, wo der Masseschwerpunkt liegen soll. Mithilfe des *Transform*-Handels des *Empty Objects* können Sie nun ganz einfach in der Szene bestimmen, wo dieser liegen soll.



**Bild 8.2** Masseschwerpunkt mit einem Kind-Objekt festlegen

### 8.2.3 Kräfte und Drehmomente zufügen

*GameObjects* mit einem *Rigidbody* werden typischerweise über das Zufügen von Kräften und Drehmomenten bewegt. Kräfte und Drehmomente können dabei sowohl per Skript zugefügt wie auch durch das Kollidieren mit anderen Objekten von diesen übertragen werden (man denke nur an Billard oder Kegeln).

Für das Zufügen per Skript bietet die *Rigidbody*-Klasse mehrere Methoden an, wovon ich einige kurz vorstellen möchte. Um dieses Vorgehen für den Entwickler etwas zu vereinfachen, bietet die *Component*-Klasse einen direkten Zugriff auf die *Rigidbody*-Komponente des eigenen *GameObjects* über die Variable *rigidbody*.

Da die Physikberechnungen immer in den von *Fixed Timestep* vorgegebenen Intervallen stattfinden, achten Sie darauf, dass Sie alle physikbezogenen Befehle immer in der *FixedUpdate*-Methode und nicht in *Update* oder *LateUpdate* vornehmen. Vor allem dann, wenn es sich um wiederholende Vorgänge handelt, ist dies wichtig. Würden Sie nämlich eine Kraft in *Update* zuweisen, die vielleicht unglücklicherweise zweimal in einem *FixedUpdate*-Intervall aufgerufen wird, wäre die Kraftzufuhr doppelt so hoch wie erwünscht.

#### 8.2.3.1 AddForce-Methode

Die *Rigidbody*-Klasse bietet gleich mehrere Funktionen, an einem Objekt bzw. einem *Rigidbody* eine Kraft zuzufügen. Die wohl am meist genutzte davon ist sicher die Methode *AddForce*. Mit dieser können Sie dem Objekt eine globale Kraft zufügen. Die Methode erwartet einen *Vector3*-Wert, der die globale Richtung und Intensität der zugefügten Kraft bestimmt.

##### **Listing 8.4** Zuführen einer nach oben gerichteten Kraft

```
void FixedUpdate() {
    rigidbody.AddForce(Vector3.up * 50);
}
```

Da die Methode eine Überladung besitzt, können Sie über einen zweiten Parameter den Typ der Kraft bestimmen. Hierfür stehen vier Typen zur Verfügung, die unterschiedliche Aus-

wirkungen haben. Wird wie im vorherigen Beispiel kein Typ gewählt, nimmt Unity per Default den Typ *Force*.

- **Force:** für eine kontinuierliche Kraft, die durch die Masse des *Rigidbody*s beeinflusst wird.
- **Acceleration:** für eine kontinuierliche Beschleunigung, die nicht durch die Masse beeinflusst wird.
- **Impulse:** für einen plötzlichen Kraftstoß, der durch die Masse beeinflusst wird.
- **VelocityChange:** für eine plötzliche Geschwindigkeitsänderung, die nicht durch die Masse beeinflusst wird.

Das folgende Skript fügt dem Objekt einen nach vorne gerichteten Impuls zu, sobald Sie dieses im Spiel anklicken. Dabei nutzt es die *OnMouseDown*-Methode, die aufgerufen wird, sobald der Nutzer den linken Mausknopf über ein GUI-Element oder einen *Collider* (s.u.) drückt.

**Listing 8.5** Per Mausklick ein Objekt anstoßen

```
using UnityEngine;
using System.Collections;
public class KickMe : MonoBehaviour {
    void OnMouseDown(){
        rigidbody.AddForce(Vector3.forward * 10,ForceMode.Impulse);
    }
}
```

Fügen Sie das obige Skript dem Cube der oben beschriebenen Testszene zu. Wenn Sie nun das Spiel starten und den Würfel mit der linken Maustaste anklicken, wird dieser weggestoßen. Da dem *Rigidbody* hier nur eine einmalige Kraft zugeführt wird, kann dies auch außerhalb der *FixedUpdate*-Methode gemacht werden.

Wie eingangs angedeutet, bietet die *Rigidbody*-Klasse noch weitere Methoden an, mit denen Sie Kräfte zuführen können. Hierzu gehören die Methoden *AddRelativeForce* (führt eine Kraft in eine lokale Richtung zu), *AddForceAtPosition* (wenn die Kraft an einem bestimmten Punkt dem Körper zugeführt werden soll) sowie *AddExplosionForce* (zum Simulieren einer Explosionskraft, die innerhalb eines Radius langsam abnimmt).

### 8.2.3.2 AddTorque-Methode

Um einem *Rigidbody* ein Drehmoment zuzufügen, stellt die *Rigidbody*-Klasse zwei Methoden bereit. Wie auch bei *AddForce* gibt es auch hier einmal die globale Variante *AddTorque* sowie *AddRelativeTorque*, die sich an der lokalen Ausrichtung des jeweiligen Objektes orientiert. *AddTorque* erwartet einen *Vector3*-Wert, der die Drehung anhand des Weltkoordinatensystems angibt und deren Stärke beschreibt. Das folgende Beispiel fügt dem *Rigidbody* in jedem Physik-Zyklus ein positives Drehmoment um die Y-Achse zu.

**Listing 8.6** Zuführen eines Drehmoments

```
void FixedUpdate(){
    rigidbody.AddTorque(Vector3.up * 50);
}
```



### Drehrichtung feststellen

Um herauszufinden, wie die Drehung sein wird, denken Sie an das linkshändische Koordinatensystem, das Unity nutzt. Machen Sie mit der linken Hand eine Faust und spreizen Sie den Daumen ab. Jetzt zeigen Sie mit dem Daumen in die Y-Richtung, also nach oben. Die Richtung der geschlossenen Finger zeigt die positive Drehrichtung dieser Achse.

Wie bei *AddForce* können Sie auch *AddTorque* einen weiteren *ForceMode*-Parameter übergeben, der die Art der Kraft festlegt. Die Auswahlmöglichkeit ist hierbei die gleiche wie bei *AddTorque*.

Das folgende Beispielskript fügt einem Körper bei einem Mausklick ein Drehmoment zu. Da wir dieses Mal sowohl die linke wie auch die rechte Maustaste nutzen wollen, setzen wir in diesem Beispiel die *OnMouseOver*-Methode ein. Diese wird ausgelöst, sobald Sie sich mit dem Mauszeiger über diesem Objekt befinden. Mithilfe der *GetMouseButtonDown*-Methode der Input-Klasse werten Sie nun aus, wann welche Maustaste heruntergedrückt wird. Bei der Übergabe von 0 wird die linke Maustaste ausgewertet, 1 wertet die rechte aus. Je nach Taste wird nun das Objekt links- oder rechtsherum gedreht.

#### **Listing 8.7** Objekte per Maustasten drehen

```
using UnityEngine;
using System.Collections;
public class RotateMe : MonoBehaviour {
    void OnMouseOver()
    {
        if (Input.GetMouseButtonDown(0))
            rigidbody.AddTorque(Vector3.up * 10,ForceMode.Impulse);
        if (Input.GetMouseButtonDown(1))
            rigidbody.AddTorque(-1 * Vector3.up * 10,ForceMode.Impulse);
    }
}
```

Fügen Sie das Beispielskript dem Cube der Testszene zu und entfernen Sie das *KickMe*-Skript aus dem *AddForce*-Beispiel. Wenn Sie nun noch *Use Gravity* vom *Rigidbody* des Cubes deaktivieren, können Sie jetzt komfortabel den Einfluss der *Angular Drag*-Eigenschaft testen.

#### **8.2.3.3 ConstantForce-Komponente**

Benötigen Sie keine großartige Logik, sondern möchten einem Objekt nur eine ständige Kraft und/oder ein Drehmoment zuführen, dann können Sie auch einfach dem *GameObject* eine *ConstantForce*-Komponente zufügen. Diese besitzt Parameter für *Force*, *Relative Force*, *Torque* und *Relative Torque*, die für eine permanente Kraft bzw. Drehkraft sorgen.

## ■ 8.3 Kollisionen

Ein weiterer wichtiger Bestandteil der Physik-Engine ist die Kollisionserkennung. Dabei ist diese nicht nur für das Erkennen von Zusammenstößen unterschiedlicher Objekte zuständig, sie spielt auch eine wichtige Rolle bei vielen anderen Spielmechanismen. So können diese beispielsweise Dialoge mit NPCs auslösen oder zum Rundenzählen eines Rennspiels eingesetzt werden.

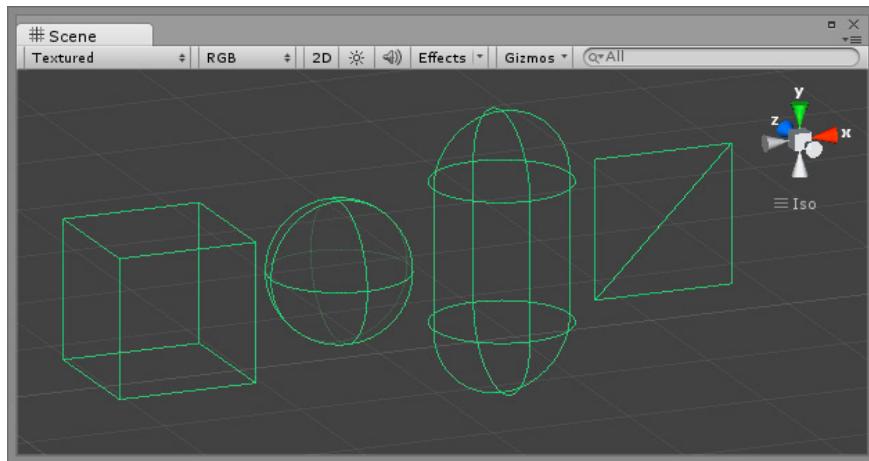
### 8.3.1 Collider

Die Hauptkomponente zum Registrieren von Kollisionen ist die *Collider*-Komponente. Diese erzeugt den eigentlichen *Collider*, der dann in der *Scene View* grün dargestellt wird.

*Collider*-Komponenten gibt es in vier unterschiedlichen Ausführungen. Zunächst gibt es die drei sogenannten *Primitive Collider*, die auch den zugehörigen Primitive-Objekten per Default zugefügt sind:

- *Box Collider*
- *Sphere Collider*
- *Capsule Collider*.

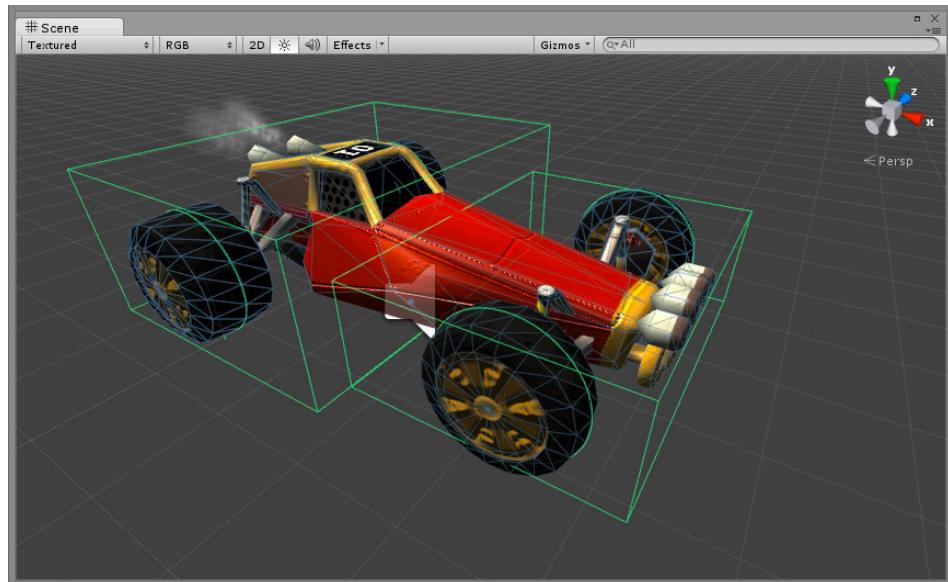
Komplettiert werden diese schließlich durch den *Mesh Collider*, der sich der Form des *Mesh* anpasst. Diesen finden Sie z. B. bei der Plane und dem Quad-Primitive.



**Bild 8.3** Die Collider-Typen Box, Sphere, Capsule und Mesh Collider

Zunächst einmal unterscheiden sich diese vier *Collider* natürlich in der Form. Die Unterschiede reichen aber noch weiter. Denn während *Box*, *Sphere* und *Capsule Collider* immer geschlossene dreidimensionale *Collider*-Körper darstellen, spielt der *Mesh Collider* eine Sonderrolle. Da dieser die Form des *Mesh* des jeweiligen *GameObjects* annimmt, muss er nicht zwingend geschlossen sein. Die Plane und das Quad sind hierfür gute Beispiele.

Beachten Sie, dass Sie jeden *Collider*, der per Default einem *GameObject* zugefügt wurde, natürlich auch entfernen und durch einen anderen ersetzen können. Denn *Collider* müssen nicht zwangsläufig die gleiche Größe und Fläche beschreiben, wie es das jeweilige *Mesh* des *GameObjects* macht. Im Gegenteil, häufig werden *Primitive Collider* genutzt, die das tatsächliche *Mesh* nur näherungsweise beschreiben, da diese nicht so performancehungrig wie *Mesh Collider* sind und noch einige weitere Vorteile gegenüber den *Mesh Collidern* besitzen.



**Bild 8.4** Performancefreundliche Collider-Anordnung bei einem Fahrzeug

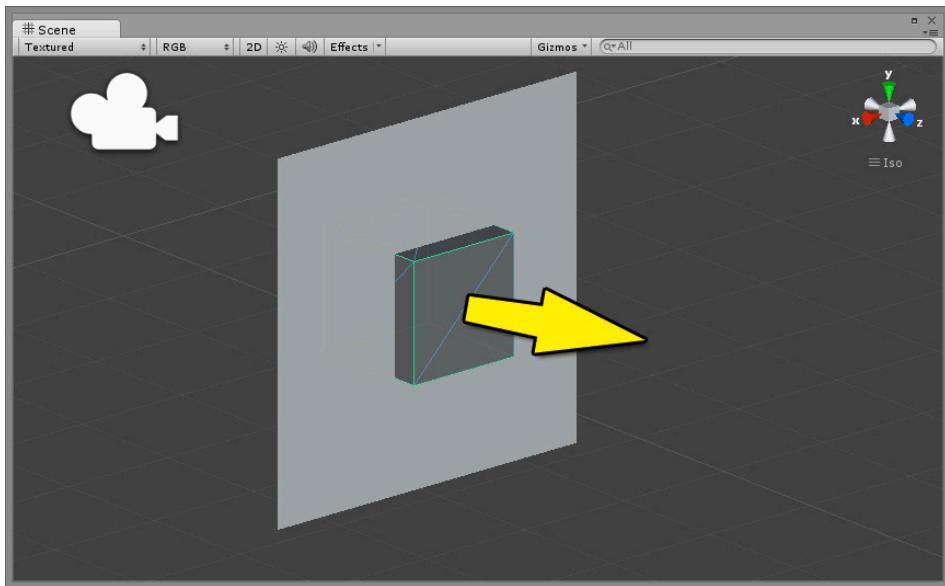
### 8.3.1.1 Mesh Collider

Da *Mesh Collider* sich der Form des jeweiligen *Mesh* anpassen, besitzen diese einige Parameter, die die geschlossenen *Primitive Collider* nicht besitzen, z.B. den Parameter *Convex*.

*Primitive Collider* erkennen Kollisionen aus allen Richtungen, selbst wenn sich das andere Objekt komplett in diesem befindet. *Mesh Collider* demgegenüber registrieren normalerweise nur Kollisionen, die von vorne kommen, also aus der Richtung, wohin die Normalen des *Mesh* zeigen. Kommt ein anderes Objekt aber von hinten, wird dieses erst mal ignoriert.

Damit *Mesh Collider* auch Kollisionen von hinten erkennen können, muss der Parameter **Convex** aktiviert werden. Dieser Parameter ermöglicht auch, dass ein *Mesh Collider* mit anderen *Mesh Collidern* kollidieren kann, was sonst ebenfalls nicht möglich ist. Bedenken Sie hierbei, dass das Aktivieren von *Convex* nur bei *Meshes* möglich ist, die weniger als 255 *Triangles* besitzen.

Schlussendlich gibt es beim *Mesh Collider* noch den Parameter **Smooth Sphere Collisions**, der den *Mesh Collider* bei den Übergängen zwischen den verschiedenen Teilloberflächen des *Mesh* etwas abrundet, um weichere Übergänge zu erzielen. Dies kann z.B. bei unebenen Untergründen mit *Mesh Collider* sinnvoll sein, auf denen andere Gegenstände entlangrollen sollen.

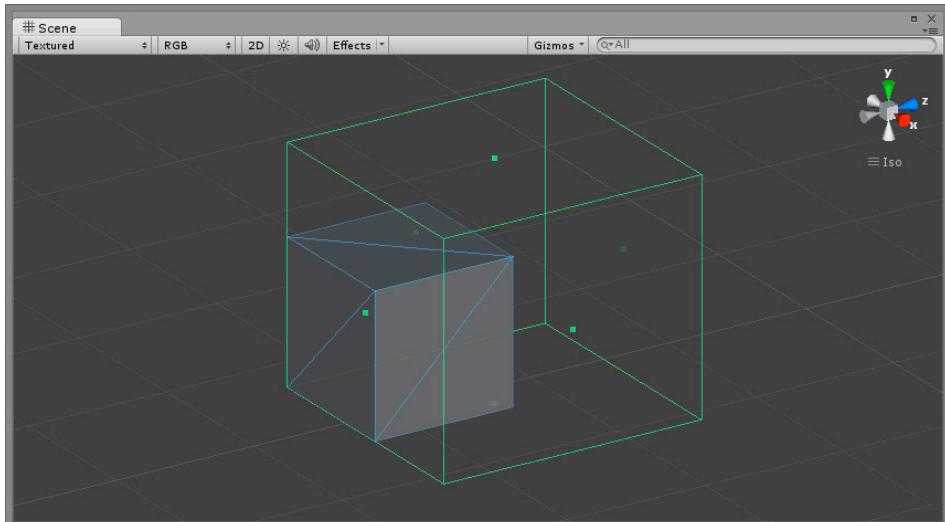


**Bild 8.5** Von hinten kommende Collider werden per Default von Mesh Collidern ignoriert.

### 8.3.1.2 Collider modifizieren

*Collider* orientieren sich in ihrer Skalierung und Anordnung immer relativ zu ihrem *Game-Object*. Davon abgesehen können alle *Primitive Collider* zusätzlich noch in ihrer Größe und der örtlichen Position angepasst werden.

Ab der Version 4.6 gibt es hierfür die Funktion **Edit Collider**, die im *Inspector* angeboten wird. In älteren Versionen erreichen Sie diese Funktion über das Drücken der **[Umsch]**-Taste

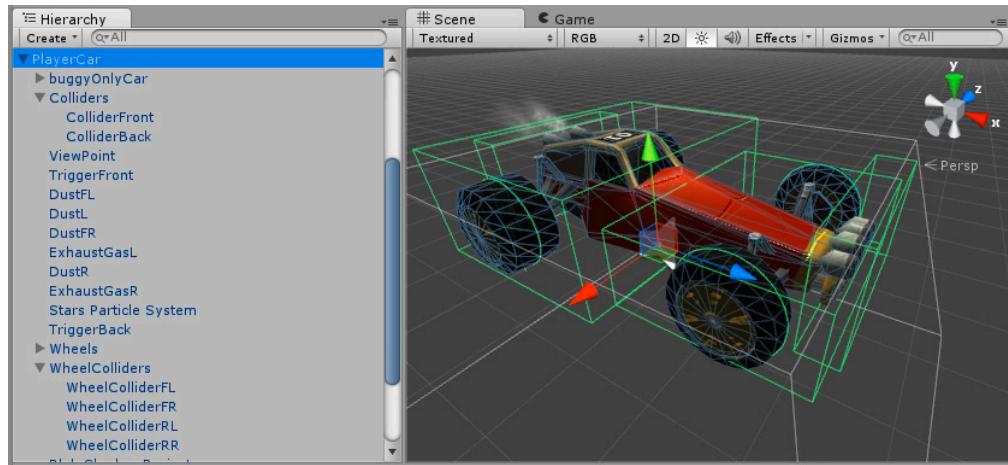


**Bild 8.6** Veränderter Collider mit Collider-Handle

(**[Shift]**). In der *Scene View* werden daraufhin grüne Handle am *Collider* angezeigt, über die Sie dann den *Collider* entsprechend anpassen können.

Zusätzlich können Sie die Position auch über die *Center*-Eigenschaft und die Größe über den *Size*-Parameter im *Inspector* verändern.

Nicht selten wird statt eines *Colliders* gleich ein ganzes *Primitive*-Objekt dem eigentlichen *GameObject* als zusätzliche Kind-Objekte hinzugefügt. Der *MeshRenderer* dieses *Primitives* wird dann deaktiviert oder ganz entfernt, sodass im Grunde nur noch der *Collider* übrig bleibt, der dann direkt über die *Transform*-Eigenschaften des Kind-Objektes skaliert und positioniert werden kann.



**Bild 8.7** Objekthierarchie mit zusätzlichen Collider-GameObjects

### 8.3.1.3 OnCollision-Methoden

Unity bietet zum Auswerten von Kollisionen mehrere Methoden an, die durch unterschiedliche Ereignisse ausgelöst werden.

Beachten Sie, dass die folgenden Methoden nur dann ausgeführt werden, wenn mindestens eines der beiden kollidierenden Objekte ein *Rigidbody* mit deaktiviertem *Is Kinematic*-Parameter besitzt.

- **OnCollisionEnter** wird in dem Moment ausgelöst, in dem zwei Objekte anfangen zu kollidieren, z.B. wenn zwei Objekte zusammenstoßen.
- **OnCollisionStay** wird regelmäßig ausgelöst, wenn die Kollision anhält, z.B. wenn ein Objekt auf einem anderen liegen bleibt.
- **OnCollisionExit** wird beim Beenden einer Kollision ausgelöst, wenn z.B. ein Objekt von einem anderen herunterfällt.

Alle drei Methoden können einen *Collision*-Parameter besitzen, der Informationen über die Kollision zwischen den beiden kollidierenden Objekten bereitstellt, z.B. den Kontaktpunkt oder die Aufprallgeschwindigkeit. Auch lässt sich hierüber auf das andere *GameObject* zugreifen bzw. dieses auswerten. Wenn Sie diesen Parameter nicht benötigen, können Sie auch auf die Angabe in der Methodendefinition verzichten.

**Listing 8.8 Vereinfachter Trampolin-Effekt**

```
using UnityEngine;
using System.Collections;
public class Trampoline : MonoBehaviour {
    void OnCollisionEnter(Collision collision)
    {
        collision.gameObject.rigidbody.AddForce(Vector3.up * 1000);
    }
}
```

Der vorherige Code fügt dem kollidierenden Objekt eine nach oben gerichtete Kraft zu. Wenn Sie das Skript der Plane der Testszene zufügen, wird der herunterfallende Würfel beim Aufprall auf die Fläche wieder nach oben katapultiert, ähnlich wie bei einem Trampolin.

### 8.3.2 Trigger

*Collider* können nicht nur als Schutzschicht gegen das Eindringen anderer Objekte genutzt werden. Sie können auch als Signalgeber dienen, wenn sich z. B. ein Objekt in einen Bereich hineinbewegt. Für diesen Zweck besitzen *Collider* einen Parameter namens **Is Trigger**, wodurch der *Collider* nun bei der normalen Kollisionserkennung von der Physik-Engine ignoriert wird und andere *Collider* in seinen eigenen *Collider*-Bereich hinein lässt. In diesem Fall wird auch von *Trigger-Collidern* oder nur von *Triggern* gesprochen, die zudem auch keine *Collider*-Ereignisse ausführen. Stattdessen lösen diese *Trigger*-Ereignisse aus, die wiederum eigene Methoden ausführen.

- **OnTriggerEnter** wird in dem Moment ausgelöst, in dem ein anderer *Collider* in den Trigger-Raum eindringt.
- **OnTriggerStay** wird regelmäßig ausgelöst, wenn ein fremder *Collider* in dem Trigger-Raum verbleibt.
- **OnTriggerExit** wird beim Verlassen eines *Colliders* aus dem Trigger-Raum ausgeführt.

Alle drei Methoden können einen *Collider*-Parameter besitzen, der Informationen über den fremden *Collider* wie auch dessen *GameObject* bereitstellt. Wenn Sie diesen Parameter nicht benötigen, müssen Sie diesen nicht zwingend angeben. Außerdem sollten Sie beachten, dass diese Methoden nur ausgeführt werden, wenn mindestens eines der beiden *Collider*-Objekte ein *Rigidbody* und natürlich einen aktiven *Is Trigger*-Parameter besitzt. Der *Is Kinematic*-Parameter spielt hierbei keine Rolle.



#### Kollisionserkennung mit Trigger-Collidern

Im Gegensatz zu den *OnCollision*-Methoden arbeiten die *OnTrigger*-Methoden unabhängig vom *Is Kinematic*-Parameter. Deshalb werden *Trigger-Collider* auch häufig bei beweglichen Objekten mit aktivierter *Is Kinematic*-Eigenschaft genutzt, um auch dort per Skript Kollisionen mit anderen Objekten, die gar kein *Rigidbody* oder ebenfalls einen aktivierten *Is Kinematic*-Parameter besitzen, auszuwerten.

**Listing 8.9** Einfacher Dialog eines NPC zum Spieler

```
using UnityEngine;
using System.Collections;
public class DeathArea : MonoBehaviour {
    public GUIText messageText;
    private string message = "Hallo Fremder! Willkommen in meinem Haus.";
    void OnTriggerEnter(Collider other) {
        messageText.text = message;
    }
    void OnTriggerExit(Collider other) {
        messageText.text = "";
    }
}
```

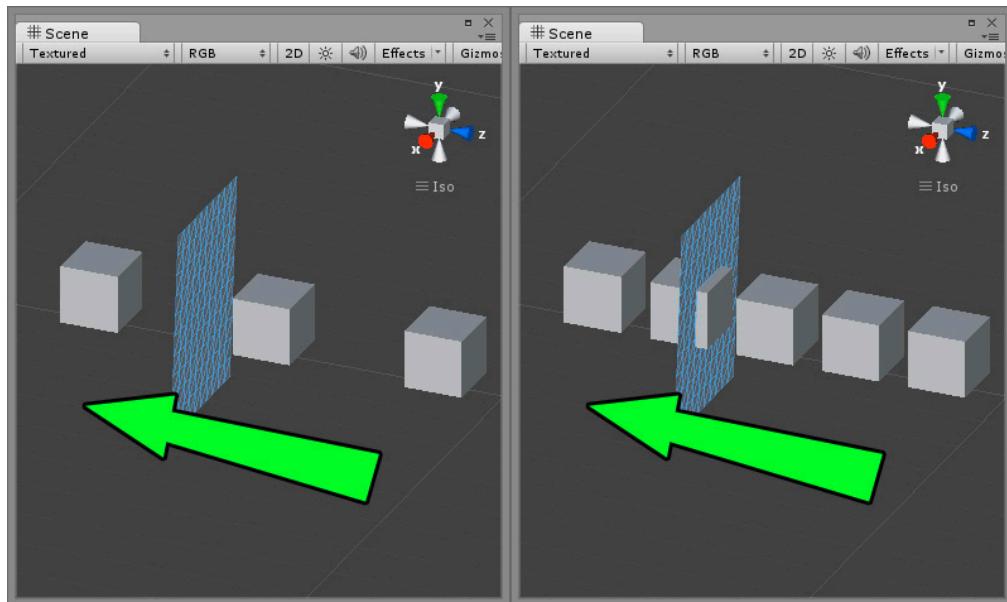
Der Beispielcode von Listing 8.9 könnte für NPCs eines Rollenspiels eingesetzt werden, die lediglich einige Worte zum Spieler sagen, wenn sich dieser nähert. Hierfür müssten Sie vor dem NPC-Modell einen *Trigger-Collider* mit diesem Skript positionieren. Nun brauchen Sie nur noch der Szene ein *GUIText*-Element (siehe Kapitel „GUI“) zufügen, das Sie der Variablen *messageText* zuweisen, und schon „spricht“ der NPC mit Ihnen, sobald Sie in seine Nähe bzw. in den *Trigger-Collider* hineintreten.

### 8.3.3 Static Collider

Objekte, die einen *Collider*, aber keinen *Rigidbody* besitzen, werden in Unity als *Static Collider* bezeichnet. Solche Objekte sollten nicht örtlich bewegt werden, weder per Code noch über Animationen. Um nämlich Kollisionen zu erkennen, merkt sich die Physik-Engine alle *Static Collider*-Positionen und beobachtet anschließend nur noch die *GameObjects*, die ein *Rigidbody* besitzen. Wenn nun doch ein *Static Collider* bewegt wird, berechnet Unity alle Positionen aller *Static Collider* neu, was natürlich Performance kostet. Deswegen sollten Sie jedem *GameObject*, das im Spiel bewegt werden könnte, ein *Rigidbody* geben. Wenn dieses ansonsten nicht auf die Physik reagieren soll, können Sie in dem Fall die *IsKinematik*-Eigenschaft aktivieren. In dem Fall können Sie das Objekt genauso behandeln, als wenn es kein *Rigidbody* besäße, nur dass es eben nicht so die Performance belastet.

### 8.3.4 Kollisionen mit schnellen Objekten

Kollisionserkennung bei schnellen Objekten stellen eine besondere Herausforderung dar. Denn die Physik-Engine überprüft Kollisionen nur in definierten Zeitabständen, die Sie über *Fixed Timestep* in **Edit/Project Settings/Time** festlegen. Ist diese zu hoch oder Ihre Objekte sind zu schnell, erkennt Unity die Kollision nicht, da diese genau zwischen zwei Intervallen stattfindet.



**Bild 8.8** Positionen eines fortbewegenden Würfels bei unterschiedlichen Fixed-Timestep-Intervallen

Um dies zu unterbinden, gibt es verschiedene Lösungsansätze, wovon ich einige vorstellen möchte:

- Nutzen Sie bei *Primitive Collidern* den *Collision Detection*-Parameter. Wenn das *GameObject* mit statischen *Collidern* kollidiert, kann die Einstellung *Continuous* die Kollisionserkennung bereits wesentlich verbessern. Sollte das Objekt mit beweglichen *GameObjects* kollidieren können, nutzen Sie *ContinuousDynamic*. Dabei müssen allerdings die anderen beweglichen Objekte ebenfalls einen der beiden Parameter-Einstellungen besitzen.
- Setzen Sie die *Fixed Timestep* herunter, um so das Zeitfenster klein zu halten, wo unentdeckte Kollisionen auftreten können.
- Vergrößern Sie die *Collider* der Objekte, damit die Kollisionszeit länger ist, in der sich die *Collider* der Objekte überschneiden. Beachten Sie hierbei, dass *Collider* durchaus größer sein können als das sichtbare *Mesh* des *GameObjects*. Allerdings kann dies besonders dann unrealistisch wirken, wenn sich die Objekte einmal nicht schnell bewegen und die Objekte bereits auf andere reagieren, obwohl sich diese noch weiter weg befinden.
- Benutzen Sie *Raycasts* (siehe „Raycasting“), um festzustellen, ob Sie sich durch ein Objekt hindurch bewegt haben. Hierfür müssen Sie ein Skript erstellen, das Sie dem bewegenden Objekt hinzufügen und das feststellt, welche Objekte sich vor diesem befinden. Im nächsten *Frame* vergleicht es dann, ob sich diese immer noch vor diesem oder bereits hinter diesem befinden. Hierfür gibt es bereits kleine kostenlose Skripte im Netz, die sich beispielsweise *DontGoThroughThings* nennen.

Keiner der Lösungsansätze ist ein Garant für eine fehlerfreie Kollisionserkennung bei jeder Geschwindigkeit. Deshalb werden diese nicht selten auch kombiniert. Hier gilt es am Ende zu testen, welche(s) Verfahren in der jeweiligen Situation am besten geeignet ist bzw. sind.

### 8.3.5 Terrain Collider

Eine weitere Sonderform von *Collidern* ist der *Terrain Collider*. Eigentlich ist es eine Sonderform des *Mesh Colliders*. Dieser ist speziell für *Terrains* konzipiert und bietet zusätzliche Features für die dort platzierten Bäume an. Mehr dazu erfahren Sie in dem Kapitel „Landschaften gestalten“.

### 8.3.6 Layer-basierende Kollisionserkennung

Während in der realen Welt jedes Objekt mit jedem anderen kollidieren kann, bietet Unity ein Feature namens *Layer-Based Collision Detection* an. Hierbei wird die *Layer*-Eigenschaft genutzt, die Sie oben rechts im *Inspector* eines jedes *GameObjects* finden.

Über die sogenannte *Layer Collision Matrix* in den *Physics Settings* (*Edit/Project Settings/Physics*) können Sie nun festlegen, welche Objekte eines *Layer*-Typs mit anderen Objekten kollidieren können. Standardmäßig sind hier immer alle Haken gesetzt, sodass alle Objekte zunächst einmal mit allen anderen kollidieren können. Wenn Sie einen neuen Layer definieren, werden bei diesem ebenfalls alle Haken gesetzt. Entfernen Sie nun einen Haken aus dieser Matrix, werden Kollisionen zwischen Objekten dieser Layer-Paare nicht mehr als solche erkannt.

		Default	TransparentFX	Ignore Raycast	Water	UI
		Default	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
		TransparentFX	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
		Ignore Raycast	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
		Water	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
		UI	<input checked="" type="checkbox"/>			

Bild 8.9

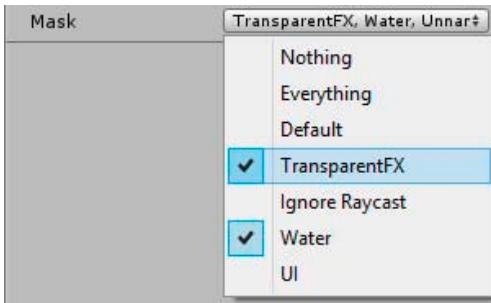
Layer Collision Matrix der Physics Settings

### 8.3.7 Mit Layer-Masken arbeiten

Neben den in Unity integrierten Features, die Layer berücksichtigen, können Sie auch selber *Layer* auswerten und entsprechend gezielt weitere Methoden aufrufen. Gerade bei *OnTrigger*- und *OnCollision*-Methoden macht dies häufig Sinn, da bei Kollisionen nicht selten abhängig von den kollidierenden Objekten unterschiedlich reagiert werden soll. So sollte z.B. ein Dialog eines Quest-Gebbers nur bei einem Spieler ausgelöst werden, nicht aber, wenn ein anderer NPC in den Trigger-Bereich hineintritt.

Über Variablen vom Typ *LayerMask* können Sie komfortabel eine Auswahl an *Layers* treffen, die dann in den jeweiligen Abfragen berücksichtigt werden können. Dabei werden normalerweise diese als Public-Variablen deklariert, da man diese im *Inspector* über ein

Popup-Menü einfach und komfortabel definieren kann. Ein großer Vorteil dieser *LayerMask*-Variablen ist zudem, dass die Auswahl immer den aktuellen *Layern* entspricht, die in diesem Projekt existieren. Erzeugen Sie z.B. über **Edit/Project Settings/Tags and Layers** einen neuen *Layer*, steht dieser auch gleich im Popup-Menü des *Inspectors* zur Verfügung.

**Bild 8.10**

Optionsauswahl einer Variablen vom Typ *LayerMask*

Allerdings ist das Auswerten dieser *LayerMask*-Variablen nicht ganz so einfach bzw. verständlich wie deren Definition, da diese als *Bit-Felder* abgelegt werden und dementsprechend anders ausgewertet werden müssen.

Ein *Bit* ist ja bekanntlich eine Informationseinheit, die entweder den Zustand 1 oder den Zustand 0 einnehmen kann (oder TRUE oder FALSE etc.). Ein **Bit-Feld** ist demnach eine Reihe von Einsen und Nullen, z.B. 0101.

In einer *LayerMask* wird nun jeder *Layer* des Projektes durch ein einzelnes Bit dargestellt, der angibt, ob dieser *Layer* in dieser Maske nun aktiviert oder deaktiviert wurde. Den ersten Layer finden Sie hierbei ganz rechts, den letzten *Layer* ganz links. Wenn Sie nun die vier *Layer* „Transparent FX“, „Ignore Raycast“, „Water“ und „Enemy“ haben, würde nun 0101 bedeuten, dass lediglich „Transparent FX“ und „Water“ aktiv sind.

Das folgende Beispiel wertet den *Layer* des kollidierenden *GameObjects* aus und gleicht diesen mit einer hinterlegten *LayerMask* namens *mask* ab. Wenn sich dieser in der *LayerMask* befindet, wird das *GameObject* zerstört.

#### **Listing 8.10** Abgleich eines Layers mit einer LayerMask

```
public LayerMask mask;
void OnTriggerEnter(Collider other) {
    if ((mask.value & 1 << other.gameObject.layer) == 1 << other.gameObject.layer){
        Destroy(gameObject);
    }
}
```

Der Befehl `1 << other.gameObject.layer` wandelt in dem Beispiel den *Layer* ebenfalls in ein *Bit-Feld* um. Dabei besitzt jeder *Layer* einen Ganzahl-Wert, der besagt, an welcher Stelle des Bit-Feldes dieser *Layer* definiert ist. Mit dem Befehl `<<` wird nun die Ziffer 1 um den Wert des *Layers* an die richtige Stelle nach links verschoben. Dieses Verfahren wird auch als „*Bitverschiebung*“ bezeichnet. Mehr zu diesem Thema erfahren Sie im Internet oder in C#-Büchern.

Diese beiden *Bit-Felder* werden nun mithilfe des einfachen **&**-Operators miteinander verglichen. Dabei werden diese bitweise UND-verknüpft. Das bedeutet, dass jede einzelne Stelle des *Bit-Feldes* verglichen wird, wodurch ein neues *Bit-Feld* als Ergebnis entsteht. Hierbei

ergibt sich immer nur dann eine 1, wenn *LayerMask* und *Layer* beide eine 1 haben. In allen anderen Fällen ist das Ergebnis 0.

**Listing 8.11** Bitweise UND-Verknüpfung

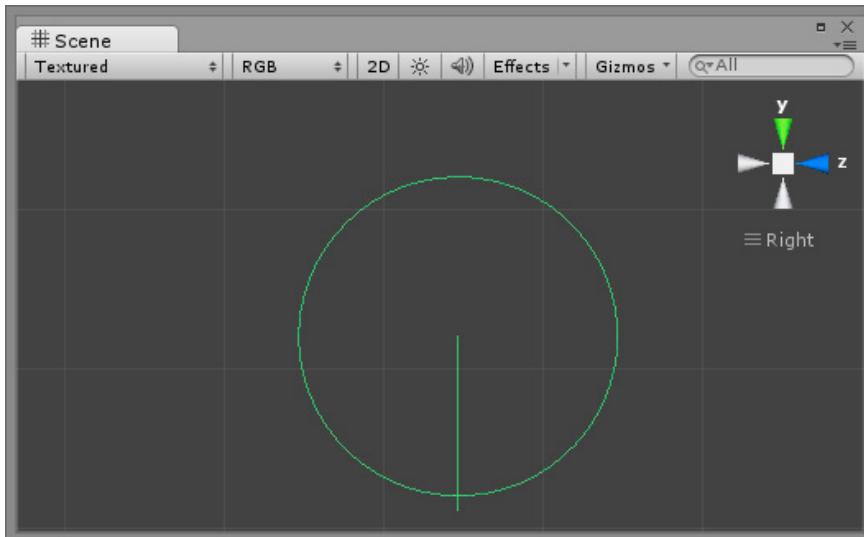
0101	(Layermask)
UND	0100 (Layer)
Ergebnis	0100

Wenn sich nun der gesuchte *Layer* in der *LayerMask* befindet, wird durch die beschriebene Logik das Ergebnis schließlich genauso aussehen wie die Bit-Folge des gesuchten *Layers*. Deshalb wird schließlich am Ende der if-Bedingung das Ergebnis dieses Vergleichs noch einmal mit dem eigentlichen *Layer* des kollidierenden Objektes verglichen. Und wenn das jetzt ein TRUE zurückgibt, dann befindet sich dieser *Layer* tatsächlich in dieser *LayerMask* und der Code kann ausgeführt werden (in dem Beispiel das Zerstören des *GameObjects*).

## ■ 8.4 Wheel Collider

Eine Sonderform von *Collidern* sind die sogenannten *Wheel Collider*, die für gewöhnlich zum Simulieren von Fahrzeugrädern genutzt werden. Sie simulieren aber nicht nur die Räder an sich, sondern auch das dazugehörige Fahrwerk inklusive Feder und Dämpfer, die natürlich ebenfalls sehr wichtig für das Verhalten eines Fahrzeugs sind.

Technisch gesehen funktioniert die Kollisionserkennung hier so, dass aus dem Zentrum des *Wheel Colliders* ein Abtaststrahl in die lokale negative Y-Richtung strahlt, dargestellt durch einen grünen Strich (siehe Bild 8.11). Die Länge ergibt sich dabei aus dem Durchmesser eines virtuellen Rades (grüner Kreis) zuzüglich der Federlänge.



**Bild 8.11** Wheel Collider

Beachten Sie bei der Darstellung des *Wheel Colliders*, dass der grüne Kreis nur die Eigenschaften des virtuellen Rades darstellt. Bei der Grundausrichtung des Reifen-*Mesh* und des *Wheel Colliders* ist dies sehr hilfreich. Zur Laufzeit wird für die Berechnung aber nur der Strahl genutzt.

*Wheel Collider* spielen in der Physikberechnung von Unity eine Sonderrolle, da sie ein spezielles Berechnungsmodell nutzen (siehe Abschnitt „Wheel Friction Curve“) und deshalb von den anderen Physikberechnungen getrennt durchgeführt werden. Dadurch unterstützen *Wheel Collider* aber auch nicht die sogenannten *Physic Materials*, die die Eigenschaften einer Oberfläche beschreiben. Stattdessen müssen unterschiedliche Oberflächen ebenfalls über die Eigenschaften der *Wheel Collider* abgebildet werden.

- **Mass** legt die Masse des Rads fest.
- **Radius** wird durch einen grünen Kreis dargestellt und beschreibt den Radius des *Wheel Colliders*.
- **Suspension Distance** ist die maximale Länge der Radaufhängung, dargestellt durch einen vertikalen Strich.
- **Center** legt den Ort des *Wheel Controllers* fest. Standardmäßig liegt dieser im Zentrum des *GameObjects*.
- **Suspension Spring** beschreibt das Federverhalten des Fahrwerks. *Spring* legt hierbei die Kraft der Feder fest (je höher, desto stärker federt es), *Damper* legt die Dämpferkraft fest (je höher, desto schwerfälliger wird gefedert) und *Target Position* legt einen prozentualen Anteil fest, der von der Federung mindestens ausgelenkt sein soll. Bei 0 wird der gesamte Federweg genutzt, bei 1 wäre die Federung komplett steif und das Rad würde sich immer im Abstand der *Suspension Distance* vom eigentlichen Zentrum befinden.
- **Forward Friction** legt die *Wheel Friction Curve*, also das Verhältnis von Kraftübertragung und Schlupf, in Fahrtrichtung fest. Diese Parameter sind für das Durchdrehen und Bremsen der Reifen wichtig.
- **Sideways Friction** legt die *Wheel Friction Curve*, also das Verhältnis von Kraftübertragung und Schlupf, zur Seite fest. Diese Werte sind für das seitliche Ausbrechen der Reifen bei Kurvenfahrten interessant.

#### 8.4.1 Wheel Friction Curve

Zum Beschreiben der Verhaltensweisen eines *Wheel Colliders* werden in Unity sogenannte *Wheel Friction Curves* eingesetzt. Sie beschreiben das Verhältnis von Kraftübertragung eines Reifens zum Schlupf. Der Schlupf beschreibt dabei, wie stark ein Reifen wegrutscht bzw. durchdreht.



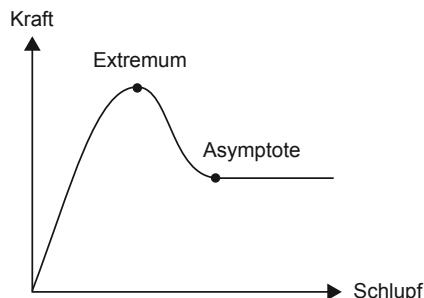
##### Reifen-Physik

Da Reifen nicht auf der Straße festgeklebt sind, wird die Kraftübertragung bei Autos über Reibung zwischen den Rädern und der Straße hergestellt.

In der Physik wird hierbei zwischen Haftreibung und Gleitreibung unterschieden.

Haftreibung sorgt zum Beispiel dafür, dass wir mit unseren Füßen auf dem

Boden stehen können. Gleitreibung findet wiederum beim Schlittschuhlaufen statt. Beim Autofahren ist dies etwas dynamischer. So kann es passieren, dass Sie bei einer normalen Kurvenfahrt keine Probleme haben und in der Spur bleiben. Fahren Sie diese aber schneller, kann es geschehen, dass Ihr Wagen auf einmal wegrutscht. Dieses Verhalten wird in Unity durch *Wheel Friction Curves* festgelegt. Sie beschreiben das Verhältnis der Kraftübertragung zum Durchdrehen bzw. Wegrutschen der Reifen, auch *Schlupf* genannt.



**Bild 8.12**  
Wheel Friction Curve

Jeder *Wheel Collider* besitzt zwei *Wheel Friction Curves*: *Forward Friction* und *Sideways Friction*, um das Verhalten in Fahrrichtung sowie zu den Seiten zu beschreiben. Dabei können folgende Parameter definiert werden.

- Der **Extremum-Punkt** der *Wheel Friction Curve* legt fest, wann das Verhalten eines Reifens instabil wird, sodass dieser z. B. durchdreht oder wegrutscht.
- Der **Asymptote-Punkt** wiederum beschreibt, wie viel Kraft beim kompletten Durchdrehen/Wegrutschen noch übertragen wird, um eine Restkontrolle über die Reifen zu beschreiben.
- Der **Stiffness Factor** ist ein Multiplikator für die *Extremum Value*- und *Asymptote Value*-Werte. Hierüber lassen sich schnell und einfach alle Werte in eine bestimmte Richtung schieben, um z. B. ein Rutschen während des Bremsens oder aufgrund einer vereisten Straße zu erreichen. Durch das Verändern dieses Wertes wird die gesamte Kurve aus Bild 8.12 nach oben gezogen oder nach unten gedrückt. Damit dies auch zur Laufzeit geht, können Sie über die Klasse *WheelFrictionCurve* auf diesen und auch die anderen Parameter per Code zugreifen.

Das folgende Skript demonstriert, wie Sie zur Laufzeit die Werte einer *Wheel Friction Curve* verändern können. Weisen Sie das folgende Beispielskript einfach einem *GameObject* zu, das eine *WheelCollider*-Komponente besitzt. Im *Inspector* können Sie dann zur Laufzeit über die Variable *stiffness* direkt auf den *Stiffness*-Wert der *Wheel Collider - Forward Friction* zugreifen.

```
using UnityEngine;
using System.Collections;
public class FrictionController : MonoBehaviour {
    public float stiffness = 1;
    WheelCollider wc;
    WheelFrictionCurve forwardFriction;
```

```
// Use this for initialization
void Awake () {
    wc = GetComponent<WheelCollider>();
    forwardFriction = wc.forwardFriction ;
}

void Update () {
    forwardFriction.stiffness = stiffness;
    wc.forwardFriction = forwardFriction;
}
}
```

## 8.4.2 Entwicklung einer Fahrzeugsteuerung

Autospiele sind im Gaming-Bereich sehr beliebt. Und da gerade die *Wheel Collider* doch einige Spezialitäten bieten, möchte ich nun mit Ihnen eine einfache Fahrzeugsteuerung entwickeln. Denn die Klasse *Wheel Collider* stellt alle Parameter zur Verfügung, die Sie für so etwas benötigen. Zu den elementarsten gehören sicher erst einmal die drei folgenden:

- **motorTorque** legt das Drehmoment für die Beschleunigung fest. Sie kann je nach Fahrt-richtung positiv und negativ sein.
- **brakeTorque** legt das Drehmoment für die Bremse fest. Es muss positiv sein.
- **steerAngle** legt den Lenkwinkel der lokalen Y-Achse in Grad fest.



### Beispiel auf der DVD

Auf der DVD finden Sie ein kleines Beispielprojekt mit dem Namen „CarExample“. In der Szene „Car“ finden Sie ein Automodell mit der Fahrzeugsteuerung, die wir im Folgenden entwickeln werden.

Ein ganz einfaches Controller-Skript mit den obigen Funktionalitäten könnte somit wie im folgenden *CarController*-Skript aussehen.

**Listing 8.12** Elementarer Aufbau einer Autosteuerung

```
using UnityEngine;
using System.Collections;

public class CarController : MonoBehaviour {
    public WheelCollider leftFrontWheel;
    public WheelCollider rightFrontWheel;
    public float maxTorque = 50.0F;
    public float maxBrakeTorque = 100.0F;
    public float steerAngle = 10;
    private float currentMotorTorque = 0;
    void FixedUpdate () {
        currentMotorTorque = maxTorque * Input.GetAxis("Vertical");
        leftFrontWheel.motorTorque = currentMotorTorque;
        rightFrontWheel.motorTorque = currentMotorTorque;
        if (Input.GetKey(KeyCode.Space))
```

```

{
    leftFrontWheel.brakeTorque = maxBrakeTorque;
    rightFrontWheel.brakeTorque = maxBrakeTorque;
}
else
{
    leftFrontWheel.brakeTorque = 0;
    rightFrontWheel.brakeTorque = 0;
}
leftFrontWheel.steerAngle = steerAngle * Input.GetAxis("Horizontal");
rightFrontWheel.steerAngle = steerAngle * Input.GetAxis("Horizontal");
}
}

```

Damit ist es natürlich noch lange nicht getan. Deswegen werden wir im Folgenden nun dieses Skript nach und nach um weitere Funktionalitäten erweitern. Am Ende dieses Abschnittes finden Sie schließlich das komplette Skript.

Als Erstes ergänzen wir nun das *CarController*-Skript um eine Maximalgeschwindigkeit. Denn **maxTorque** beschreibt nur die maximale Beschleunigung, nicht aber die maximale Geschwindigkeit. Zur Vereinfachung werde ich hierfür sowohl für vorwärts wie auch rückwärts die gleichen Grenzwerte nutzen. Zum Bestimmen der Geschwindigkeit nutze ich die **rpm**-Eigenschaft der *WheelCollider*-Klasse sowie den Radius des *Wheel Colliders*.

- **rpm** gibt die aktuelle Achsdrehzahl pro Minute zurück.
- **radius** ist der Radiuswert aus dem Inspector, den ich bereits oben erläutert habe.

Damit aber nicht für jeden Vergleich die aktuelle Geschwindigkeit berechnet werden muss, werde ich stattdessen die RPM-Werte vergleichen. Hierfür muss ich aber als Erstes den RPM-Wert bei Maximalgeschwindigkeit berechnen, den ich in der Private-Variablen **maxRpm** speichere. Für diese Berechnung brauchen Sie neben den obigen Werten noch die Kreisumfangsformel ( $2\pi r$ ) sowie das Beachten der Einheiten. Schließlich soll die Geschwindigkeit der Variablen **maxSpeed** in km/h angegeben werden können und nicht in Meter pro Minute.

#### **Listing 8.13** Berechnung der maximalen Drehzahl

```

private float maxRpm = 0;
void Awake (){
    maxRpm = maxSpeed * 1000 / (2* Mathf.PI * leftFrontWheel.radius * 60);
}

```

Für PI nutzen wir die Unity-Klasse **Mathf**, die viele mathematische Funktionen und Konstanten bereitstellt. Diese Klasse werden wir auch im nächsten Schritt nutzen.

Als Nächstes müssen wir nun noch vergleichen, ob unsere aktuelle Drehzahl bereits erreicht ist oder ob wir weiter beschleunigen dürfen. Die Schwierigkeit ist hierbei, dass wir sowohl das Vorwärtsfahren wie auch das Zurücksetzen berücksichtigen müssen. Hierfür nutzen wir die Methode **Abs**, die den Absolutwert bzw. Betrag einer Zahl zurückgibt (der Wert einer Zahl losgelöst vom Vorzeichen).

Wenn der Betrag erreicht ist, schauen wir als Nächstes, ob die beabsichtigte Beschleunigung, in die der Spieler möchte (**currentMotorTorque**), die gleiche ist, in die das Fahrzeug aktuell fährt (**rpm**). Sind diese nämlich gegensätzlich, kann der Spieler ja ruhig weiter in die entgegengesetzte Richtung beschleunigen.

Diesen Vergleich machen wir mit der Methode `Sign` der `Mathf`-Klasse, die lediglich das Vorzeichen einer Zahl auswertet und eine 1 bei einer positiven Zahl und eine -1 bei einer negativen Zahl zurückgibt. Wenn also eine positive 1 zurückgegeben wird, wird `currentMotorTorque` einfach auf 0 gesetzt, damit nicht weiter beschleunigt wird.

**Listing 8.14** Überprüfung der Drehzahl mit der maximalen Drehzahl

```
float rpm = leftFrontWheel.rpm;
if ((Mathf.Abs (rpm) >= maxRpm))
{
    if (Mathf.Sign (rpm * currentMotorTorque)==1)
        currentMotorTorque = 0;
}
```

Damit wir nicht ständig auf die Eigenschaften von `leftFrontWheel` zugreifen müssen, speichern wir dabei den `rpm`-Wert noch in einer Variablen namens `rpm` zwischen. Den obigen Block integrieren wir dann in die `FixedUpdate`-Methode vor der `motorTorque`-Zuweisung, damit auch der richtige `currentMotorTorque`-Wert den *Wheel Collider*n zugewiesen wird.

**Listing 8.15** `FixedUpdate`-Methode des Autoskriptes

```
void FixedUpdate () {
    currentMotorTorque = maxTorque * Input.GetAxis("Vertical");
    float rpm = leftFrontWheel.rpm;
    if ((Mathf.Abs (rpm) >= maxRpm))
    {
        if (Mathf.Sign (rpm * currentMotorTorque)==1)
            currentMotorTorque = 0;
    }
    leftFrontWheel.motorTorque = currentMotorTorque;
    rightFrontWheel.motorTorque = currentMotorTorque;
    if (Input.GetKey(KeyCode.Space))
    {
        leftFrontWheel.brakeTorque = maxBrakeTorque;
        rightFrontWheel.brakeTorque = maxBrakeTorque;
    }
    else {
        leftFrontWheel.brakeTorque = 0;
        rightFrontWheel.brakeTorque = 0;
    }
    leftFrontWheel.steerAngle = steerAngle * Input.GetAxis("Horizontal");
    rightFrontWheel.steerAngle = steerAngle * Input.GetAxis("Horizontal");
}
```

Als Nächstes steht nun das Animieren der Räder an, schließlich sollen sich unsere Reifen ja auch drehen. Hierfür ist es natürlich zwingend notwendig, dass diese aus einzelnen *Meshes* bestehen, die man separat auch drehen und verändern kann.

Zum Drehen der Räder programmieren wir eine kleine Methode, die die aktuelle Drehung des Transformers anhand der *Wheel Collider*-RPM-Eigenschaft berechnet und das *Transform* dann dementsprechend mit der `Rotate`-Methode dreht. Bei der Berechnung teilen wir den `rpm`-Wert des *Wheel Collider*s zunächst durch 60, um auf Sekunden zu kommen. Um jetzt die Gradzahl zu erhalten, multiplizieren wir das Ergebnis mit 360. Da wir diese Methode später in der `Update`-Methode aufrufen werden, multiplizieren wir das Ergebnis noch einmal mit `deltaTime`, um den *Frame*-Zustand zu erhalten.

**Listing 8.16** Methode zum Drehen der Rad-Mesh

```
void RotateVisuals(WheelCollider wc, Transform visualWheel)
{
    visualWheel.Rotate(wc.rpm / 60 * 360 * Time.deltaTime, 0, 0);
}
```

Damit wir alle Räder mit dieser Methode drehen können, müssen wir zuvor noch zwei weitere *public*-Variablen für die Hinterrad-*Wheel Collider* sowie vier für die *Transform*-Komponenten der Rad-*Meshes* anlegen.

Abschließend wollen wir noch dafür sorgen, dass die Vorderräder beim Lenken auch eingeschlagen werden. Hierfür werden wir einen kleinen Trick anwenden: Damit wir keine Konflikte mit den bereits oben getätigten Rotationen erhalten, werden wir zwei zusätzliche *GameObjects* dem Wagen zufügen, die als Container für die sich drehenden Rad-Objekte dienen. Diese werden als *Private*-Variablen deklariert und anschließend in der *Awake*-Methode wie folgt erstellt:

**Listing 8.17** Erstellen zusätzlicher GameObjects zum Lenken

```
//Neues GameObject erzeugen und den Namen "Left Steering" geben
leftSteering = new GameObject("Left Steering");
//Dem GameObject ein Eltern-Objekt zuweisen, und zwar das Hauptobjekt
leftSteering.transform.parent = transform;
//Position des Rad-Objektes zuweisen
leftSteering.transform.position = leftFrontVisuals.position;
//Drehung des Rad-Objektes zuweisen
leftSteering.transform.rotation = leftFrontVisuals.rotation;
//Dem Rad-Objekt das neue Objekt als Eltern-Objekt zuweisen
leftFrontVisuals.parent = leftSteering.transform;
```

Dieser Vorgang muss für das rechte Vorderrad natürlich noch wiederholt und ebenfalls in die *Awake*-Methode integriert werden. Damit sehen die vollständige Variablen-deklaration und die *Awake*-Methode wie folgt aus.

**Listing 8.18** Variablen und Awake-Methode des Autoskriptes

```
//Wheel Collider
public WheelCollider leftFrontWheel;
public WheelCollider rightFrontWheel;
public WheelCollider leftBackWheel;
public WheelCollider rightBackWheel;
//Mesh-Objekte
public Transform leftFrontVisuals;
public Transform rightFrontVisuals;
public Transform leftBackVisuals;
public Transform rightBackVisuals;
//Parameter
public float maxTorque = 50.0F;
public float maxBrakeTorque = 100.0F;
public float steerAngle = 10;
public float maxSpeed = 120;
private float maxRpm = 0;
private float currentMotorTorque = 0;
private GameObject leftSteering;
private GameObject rightSteering;
```

```

void Awake () {
    maxRpm = maxSpeed * 1000 / (2* Mathf.PI * leftFrontWheel.radius * 60);
    leftSteering = new GameObject("Left Steering");
    leftSteering.transform.parent = transform;
    leftSteering.transform.position = leftFrontVisuals.position;
    leftSteering.transform.rotation = leftFrontVisuals.rotation;
    leftFrontVisuals.parent = leftSteering.transform;

    rightSteering = new GameObject("Right Steering");
    rightSteering.transform.parent = transform;
    rightSteering.transform.position = rightFrontVisuals.position;
    rightSteering.transform.rotation = rightFrontVisuals.rotation;
    rightFrontVisuals.parent = rightSteering.transform;
}

```

Haben Sie das getan, können Sie diese Container dafür nutzen, die Räder nach links und rechts einschlagen zu lassen, ohne dass Konflikte mit den lokalen Rotationen der Rad-*Meshes* (in diesem Fall *leftFrontVisuals*) entstehen. Hierfür programmieren wir nun noch eine kleine Methode. Im Gegensatz zu dem ständigen Rotieren der Räder müssen wir hier aber bedenken, dass wir in diesem Fall das *Transform* lediglich um einen festen Winkel um die lokale Y-Achse drehen wollen, es aber nicht rotieren lassen möchten. Hier eignet sich die Methode *localEulerAngles*, mit der wir genau dies machen können.

#### **Listing 8.19** Methode zum Lenken der Vorderräder

```

void SteerVisuals(WheelCollider wc, Transform steeringObject)
{
    steeringObject.localEulerAngles = new Vector3(0,wc.steerAngle,0);
}

```

Diese Methode rufen wir dann mit den neuen *Steering-GameObjects* und den dazugehörigen *Wheel Collidern* ebenfalls in der Update-Methode auf, sodass diese dann wie folgt aussieht:

#### **Listing 8.20** Update-Methode der Autosteuerung

```

void Update(){
    RotateVisuals(leftFrontWheel,leftFrontVisuals);
    RotateVisuals(rightFrontWheel,rightFrontVisuals);
    RotateVisuals(leftBackWheel,leftBackVisuals);
    RotateVisuals(rightBackWheel,rightBackVisuals);
    SteerVisuals(leftFrontWheel,leftSteering.transform);
    SteerVisuals(rightFrontWheel,rightSteering.transform);
}

```

Obwohl dieses Skript schon recht umfangreich scheint, sind hier noch lange nicht alle Punkte berücksichtigt, die eine vernünftige Autosteuerung benötigt. So fehlt hier noch eine Vollbremsung mit blockierenden Rädern, Sound-Erzeugung, Bremslichter und einiges mehr.

Auch fehlt eine Positionierung der Rad-*Meshes*, wenn die Räder federn. Aktuell drehen sich diese lediglich und reagieren nicht auf die Federung der *Wheel Collider*. Ein Beispiel für eine solche Federung finden Sie auf der beiliegenden DVD im Projekt „CarExample“. Zur Demonstration der grundsätzlichen Funktionsweisen von *Wheel Collidern* sollte dies aber ausreichen.

### 8.4.2.1 CarController.cs

Das komplette Autosteuerungsskript sieht wie folgt aus:

**Listing 8.21** Fertiges CarController.cs-Skript

```
using UnityEngine;
using System.Collections;

public class CarController : MonoBehaviour {
    public WheelCollider leftFrontWheel;
    public WheelCollider rightFrontWheel;
    public WheelCollider leftBackWheel;
    public WheelCollider rightBackWheel;

    public Transform leftFrontVisuals;
    public Transform rightFrontVisuals;
    public Transform leftBackVisuals;
    public Transform rightBackVisuals;

    public float maxTorque = 50.0F;
    public float maxBrakeTorque = 100.0F;
    public float steerAngle = 10;
    public float maxSpeed = 120;
    private float maxRpm = 0;
    private float currentMotorTorque = 0;

    private GameObject leftSteering;
    private GameObject rightSteering;

    public float currentRPM = 0;
    void Awake ()
    {
        maxRpm = maxSpeed * 1000 / (2* Mathf.PI * leftFrontWheel.radius * 60);

        leftSteering = new GameObject("Left Steering");
        leftSteering.transform.parent = transform;
        leftSteering.transform.position = leftFrontVisuals.position;
        leftSteering.transform.rotation = leftFrontVisuals.rotation;
        leftFrontVisuals.parent = leftSteering.transform;

        rightSteering = new GameObject("Right Steering");
        rightSteering.transform.parent = transform;
        rightSteering.transform.position = rightFrontVisuals.position;
        rightSteering.transform.rotation = rightFrontVisuals.rotation;
        rightFrontVisuals.parent = rightSteering.transform;
    }

    void FixedUpdate ()
    {
        currentMotorTorque = maxTorque * Input.GetAxis("Vertical");
        float rpm = leftFrontWheel.rpm;

        if ((Mathf.Abs (rpm) >= maxRpm))
        {
            if (Mathf.Sign (rpm * currentMotorTorque)==1)
                currentMotorTorque = 0;
        }
    }
}
```

```

leftFrontWheel.motorTorque = currentMotorTorque;
rightFrontWheel.motorTorque = currentMotorTorque;

if (Input.GetKey(KeyCode.Space))
{
    leftFrontWheel.brakeTorque = maxBrakeTorque;
    rightFrontWheel.brakeTorque = maxBrakeTorque;
}
else
{
    leftFrontWheel.brakeTorque = 0;
    rightFrontWheel.brakeTorque = 0;
}

leftFrontWheel.steerAngle = steerAngle * Input.GetAxis("Horizontal");
rightFrontWheel.steerAngle = steerAngle * Input.GetAxis("Horizontal");
}

void Update()
{
    RotateVisuals(leftFrontWheel, leftFrontVisuals);
    RotateVisuals(rightFrontWheel, rightFrontVisuals);
    RotateVisuals(leftBackWheel, leftBackVisuals);
    RotateVisuals(rightBackWheel, rightBackVisuals);

    SteerVisuals(leftFrontWheel, leftSteering.transform);
    SteerVisuals(rightFrontWheel, rightSteering.transform);
}

void RotateVisuals(WheelCollider wc, Transform visualWheel)
{
    visualWheel.Rotate(wc.rpm / 60 * 360 * Time.deltaTime, 0, 0);
}

void SteerVisuals(WheelCollider wc, Transform steeringObject)
{
    steeringObject.localEulerAngles = new Vector3(0, wc.steerAngle, 0);
}
}

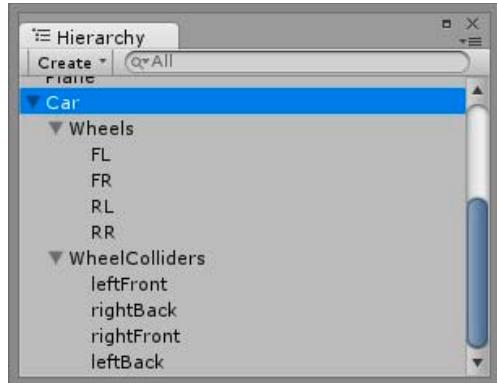
```

### 8.4.3 Autokonfiguration

Zum Verwenden des obigen Skriptes ist nun noch der Aufbau des Fahrzeugmodells wichtig. Wie schon eingangs erwähnt, ist es natürlich wichtig, dass alle Räder aus separaten *Meshes* bestehen. Weiter ist aber auch wichtig, dass die Ausrichtungen der Objekte korrekt sind. Zeigt z.B. die Autofront in die negative statt in die positive Z-Richtung, dann fährt der Wagen optisch auch in die falsche Richtung. Noch problematischer wird es, wenn die Rad-*Meshes* falsche Orientierungen besitzen. Dann kann es zu noch merkwürdigeren Effekten kommen.

Um solche Probleme von vornherein zu umgehen, können Sie hier einen kleinen Trick anwenden. Setzen Sie richtig ausgerichtete *Empty Objects* ein, die als Container für die eigentlichen *Mesh*-Objekte dienen. Und anstatt die *Mesh*-Objekte dann dem Skript zuzuweisen, weisen Sie einfach die *Empty Objects* zu, die dann die *Mesh*-Objekte einfach mitdrehen.

Die Struktur würde dann wie folgt aussehen: Ein richtig ausgerichteter Hauptcontainer (*Empty Object*) für das gesamte Auto-Objekt (im Bild 8.13 „Car“ genannt) sowie vier Kind-*Empty Objects* für die *Wheel Collider* und vier Kind-*Empty Objects* für die Reifen-*Mesh*-Objekte. Für eine saubere Übersicht wurden diese noch einmal zusätzlich in Container-Objekte gebündelt, die immer mit **Reset** des Kontextmenüs der Transform-Komponente im Zentrum des Hauptcontainers platziert werden.

**Bild 8.13**

Objektstruktur eines Automodells

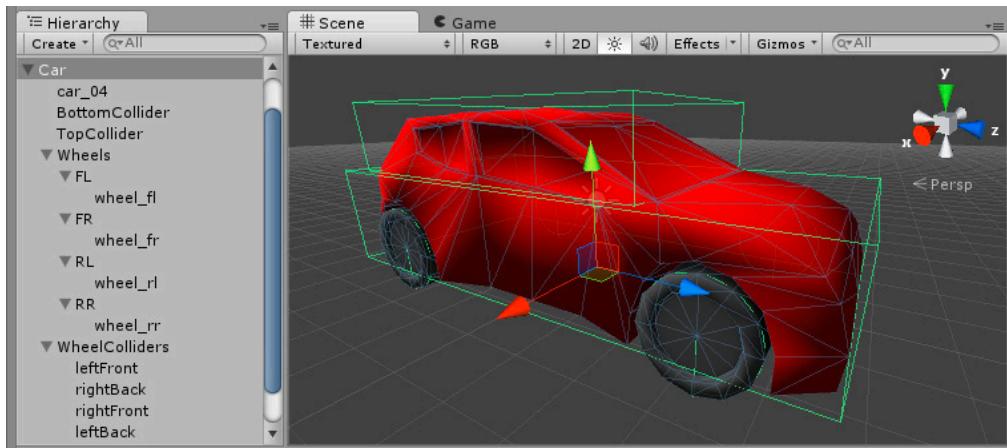
Das *GameObject* mit dem Haupt-*Mesh* (Karosserie) weisen Sie dann dem „Car“-Hauptcontainer als Kind-Objekt zu. „Car“ weisen Sie dann auch das entwickelte Autoskript *CarController* sowie ein *Rigidbody* zu. Zur Kollisionserkennung können Sie diesem dann auch noch einen oder mehrere Cubes als Kind-Objekte zufügen, die Sie dann in Form und Größe der Karosserie anpassen. Die *Renderer*-Komponenten dieser *Collider*-Objekte deaktivieren Sie natürlich.

Die separaten *Mesh*-Objekte der Reifen fügen Sie dann den jeweiligen Containern zu, und zwar so, dass die Mitte des Reifen-*Mesh* im Zentrum des jeweiligen *Empty Objects* liegt. Wenn beim Modelling des Reifens der Pivot-Punkt in das Zentrum gelegt wurde, können Sie das ganz einfach über die Funktion **Reset Position** des Kontextmenüs der *Transform*-Komponente machen, ansonsten müssen Sie dies manuell machen. Die richtige Positionierung der Räder an der Karosserie machen Sie dann anschließend über das Verschieben der Container *Empty Objects* „FL“, „FR“, „RL“ und „RR“.



### Reifen-*Mesh* und *Wheel Collider* positionieren

Um ein realistisches Verhalten der Reifen zu erzeugen, sollte der *Wheel Collider* die gleiche Position und auch den gleichen Radius besitzen wie das Reifen-*Mesh*. Durch die obige Vorgehensweise können Sie die Positionen der Container-Objekte der Reifen-*Meshe*s einfach auf die *Wheel Collider* übertragen. Anschließend brauchen Sie nur noch den Radius des *Wheel Colliders* dem Reifen-*Mesh* anzupassen.



**Bild 8.14** Konfiguriertes Auto-GameObject

#### 8.4.4 Fahrzeugstabilität

Ein weiterer wichtiger Bestandteil eines Fahrzeugs neben der eigentlichen Steuerung ist die Stabilität. Hier müssen Sie am Ende entscheiden, ab welcher Geschwindigkeit und Kurve ein Wagen sich überschlagen soll. Oder sollte er doch besser wegrutschen? Was so einfach klingt, bedeutet für Sie einiges an Testen und Abstimmen unterschiedlicher Parameter.

Zum einen können Sie hierfür Stabilisator-Skripte programmieren, die Ihr Fahrzeug vor allem in Kurvenfahrten stabilisieren (auch *Anti-Roll Bars* genannt). Aber auch die bereits erwähnten *Wheel Friction Curves* spielen hier eine große Rolle. Bevor der Wagen sich nämlich in einer Kurve überschlägt, weil dieser zu weit nach außen gedrückt wird, könnten die Reifen beispielsweise auch einfach wegrutschen.

Und auch das Verändern des Masseschwerpunkts kann für Stabilität sorgen. Ein Wagen mit einem tiefen Schwerpunkt überschlägt sich nicht so einfach wie ein Auto mit einem hochliegenden Schwerpunkt. Das Nutzen des bereits vorgestellten *SetCenterOfMass*-Skript kann hier behilflich sein. Außerdem können Sie durch das Heruntersetzen des *Fixed Timestep*-Intervalls für eine präzisere Physikberechnung sorgen, was ebenfalls stabilere Verhaltensweisen unterstützen kann.



#### Zusatzskripte zur Fahrzeugstabilität und Federung

In der Szene „Car-advanced“ des Unity-Projektes „CarExample“ finden Sie eine erweiterte Autosteuerung, die neben dem *SetCenterOfMass*-Skript auch „*Anti-Roll Bar*“-Skripte nutzt, um dem Fahrzeug mehr Stabilität zu verleihen. Außerdem wird ein Reifenfederung-Skript eingesetzt, das die Reifen-Meshes entsprechend der Federung eines Wheel Colliders automatisch neu positioniert.

## ■ 8.5 Physics Materials

Mit *Physics Materials* haben Sie die Möglichkeit, Oberflächeneigenschaften wie Reibung und Federung eines *Colliders* festzulegen. Dies machen Sie über die *Material*-Eigenschaft des *Colliders*. Ähnlich wie bei einer *Wheel Friction Curve* müssen Sie auch bei dieser Reibung zum einen die Haftreibung sowie die Gleitreibung berücksichtigen, die Sie in diesem Fall über separate Parameter festlegen. Vereinfacht gesagt ist die Gleitreibung der Wert, wie stark ein anderes Objekt auf diesem gleitet. Die Haftreibung ist wiederum der Anfangswiderstand, der zunächst überwunden werden muss, damit das andere Objekt überhaupt gleitet. *Physics Materials* können Sie wie alle anderen Assets auch über die rechte Maustaste im *Project Browser* anlegen oder über dessen **Create**-Menü. Sie haben folgende Eigenschaften (alle lassen sich in einem Bereich von 0 bis 1 einstellen):

- **Dynamic Friction** legt die Gleitreibung fest, also wenn das andere Objekt bereits auf diesem rutscht.
- **Static Friction** legt die Haftreibung fest, also wenn das andere Objekt auf diesem fest liegt.
- **Bounciness** legt das Federverhalten in einem Bereich von 0 bis 1 fest. 0 bedeutet kein Federn. 1 federt den Gegenstand mit der gleichen Energie wieder weg, wie es auftraf.
- **Friction Combine** legt fest, wie das Reibungsverhalten von unterschiedlichen Objekten kombiniert werden soll. Zur Verfügung stehen *Average* (Mittelwert), *Min* (der kleinere Wert), *Max* (der größere Wert) und *Multiply* (die Werte werden multipliziert).
- **Bounce Combine** legt fest, wie das Federverhalten von unterschiedlichen Objekten kombiniert werden soll. Die Auswahlmöglichkeiten sind wie bei *Friction Combine*.
- **Friction Direction 2** definiert eine bestimmte Richtung für eine zweite, von der obigen abweichende Reibung.
- **Dynamic Friction 2** legt die Gleitreibung für diese spezielle Richtung fest.
- **Static Friction 2** legt die Haftreibung für die spezielle Richtung fest.

Zum Testen der verschiedenen Parameter eignet sich das gleiche Testszenario, das Sie bereits im Unterkapitel *Rigidbody* genutzt haben. Setzen Sie die Parameter des *Rigidbodies* auf die Anfangswerte zurück, entfernen Sie alle Skripte und drehen Sie die Plane z.B. um -2 der Z-Achse, damit der herunterfallende Cube auch eine Möglichkeit zum Rutschen hat. Erstellen Sie nun ein neues *Physic Material*, das Sie dem Cube-*Collider* zuweisen. Zum Testen empfiehlt es sich, dem Parameter *Bounce Combine* den Wert *Max* zu geben und dem Parameter *Friction Combine* den Wert *Min* zuzuweisen. Nun können Sie die Werte des *Physic Materials* anpassen und nach dem Starten der Szene die Veränderungen überprüfen.

## ■ 8.6 Joints

*Joints* dienen dem Verbinden zweier *GameObjects* mit *Rigidbody*-Komponenten. Dabei stehen verschiedene Arten mit unterschiedlichen Eigenschaften zur Verfügung. Alle besitzen aber die Fähigkeit, bei Überlastung kaputtzugehen und dabei die Verbindung zu lösen. Hierfür besitzen alle *Joints* die folgenden Eigenschaften:

- **Connected Body** legt das andere *GameObject* fest, mit dem dieses Objekt über den *Joint* verbunden ist.
- **Break Force** legt die Kraft fest, bei der der *Joint* kaputtgeht bzw. sich löst. Bei der Einstellung *Infinity* kann sich die Verbindung gar nicht lösen.
- **Break Torque** legt das Drehmoment fest, bei dem der *Joint* kaputtgeht bzw. sich löst. Bei *Infinity* kann sich die Verbindung ebenfalls nicht lösen.



### Joint

Auf der DVD finden Sie ein Video, das die unterschiedlichen Joint-Varianten in der Praxis zeigt.

#### 8.6.1 Fixed Joint

*Fixed Joints* sind relativ starre Verbindungen, in etwa vergleichbar mit dem Eltern-Kind-Verhalten von *GameObjects*. Wenn kein *Rigidbody-GameObject* über die *Connected Body*-Eigenschaft angegeben wird, „hängt“ sich der *Joint* an seine aktuelle Position in der 3D-Welt.

#### 8.6.2 Spring Joint

Der *Spring Joint* agiert wie eine Feder, die zwei Objekte verbindet. Dabei versucht das bewegliche Objekt die Position zu erreichen, die es am Anfang der Szene bzw. in der *Scene View* hat.

- **Anchor** legt die Mitte des *Joints* fest.
- **Spring** legt die Federstärke fest.
- **Damp** legt die Dämpfung fest, also wie schwerfällig die Feder ist.
- **Min/Max Distance** definiert einen Entfernungsbereich, in dem die Feder keinen Einfluss hat.

### 8.6.3 Hinge Joint

Diese Verbindung verhält sich wie ein Scharnier bzw. eine Türangel. Dabei unterstützt sie aber nicht nur die Drehbewegung des Scharniers, sondern auch Spezialitäten wie das Zurückziehen einer Tür mithilfe einer Kraft.

- **Anchor** legt die Position des Scharniers bzw. der Drehachse fest, um die sich der Körper dreht.
- **Axis** legt die Ausrichtung der Drehachse fest.
- **Use Spring/Spring** ermöglicht einen Federeffekt, ähnlich wie die Feder an einer Ladentür.
- **Use Motor/Motor** ermöglicht das Hinzufügen einer zusätzlichen Motorkraft, die ein Verhalten wie z. B. von automatisch öffnenden Türen ermöglicht.
- **Use Limits/Limits** ermöglicht das Begrenzen der Drehung um das Scharnier.

## ■ 8.7 Raycasting

Eine weitere sehr nützliche Funktionalität in Unity ist das sogenannte *Raycasting*. Hierbei wird ein Strahl von einem bestimmten Punkt in eine vorgegebene Richtung gesendet. Dabei wird festgestellt, ob sich dort andere Objekte (mit *Collider*) befinden. Dies können Sie gut mit einem Laserpointer vergleichen, nur dass der Strahl beim *Raycasting* unsichtbar ist. Das Verfahren ist vielfältig einsetzbar und wird gerne beim Programmieren von künstlicher Intelligenz genutzt. Aber auch beim Schießen oder beim Selektieren eines Gegenstandes durch einen Mausklick wird dieses Verfahren sehr gerne gebraucht.

Da es so universell einsetzbar ist, unterstützen deshalb auch gleich mehrere Klassen das *Raycasting*. Im Kapitel „Kameras, die Augen des Spielers“ demonstriere ich zum Beispiel die Methode *ScreenPointToRay*, die die *Camera*-Klasse zur Verfügung stellt. Auch hier wird das *Raycasting* eingesetzt. Aber auch die *Physics*-Klasse unterstützt *Raycasting*. Die Methode *Raycast* kann hierbei vier Parameter aufnehmen:

- **origin** legt den Startpunkt des Strahls fest.
- **direction** bestimmt die Richtung des Strahls und wird als normalisierter *Vector3*-Wert angegeben.
- **hitInfo** ist ein Rückgabewert, der über den Parametermodifizierer *out* einen *RaycastHit*-Parameter zurückgibt. Dieser besitzt Informationen über den anderen *Collider* und den eigentlichen Trefferpunkt.
- **distance** legt die Länge des Strahls fest, also welche Reichweite dieser hat.
- **LayerMask** legt fest, welche *Layer* von dem Strahl erkannt und welche ignoriert werden sollen. Wenn Sie keine *LayerMask* angeben, werden alle *Layer* erkannt mit Ausnahme des *Layers IgnoreRaycast*. Wenn dieser *Layer* erkannt werden soll, müssen Sie eine *LayerMask* der Methode übergeben, die eben auch diesen *Layer* beinhaltet.

Zu diesen Parametern gibt es noch einen Rückgabewert, der sagt, ob ein Objekt überhaupt getroffen wurde oder eben nicht.

Die Methode existiert in mehreren überladenen Versionen, sodass Sie nicht unbedingt alle Parameter angeben müssen. Außerdem gibt es auch eine Variante mit einer Variablen vom Typ Ray. Diese stellt den Strahl dar und ersetzt *origin* und *direction*.

#### **Listing 8.22** Raycasting mit einer Ray-Variablen

```
using UnityEngine;
using System.Collections;
public class Destroyer : MonoBehaviour {

    void Update () {
        if (Input.GetButtonDown("Fire1")){
            Ray ray = new Ray();
            ray.origin = transform.position;
            ray.direction = transform.TransformDirection(Vector3.forward);

            RaycastHit hit;
            if (Physics.Raycast(ray, out hit,2))
                Destroy (hit.collider.gameObject);
        }
    }
}
```

Durch das Drücken der Fire1-Taste sendet das Skript einen Strahl 2 m nach vorne. Wenn sich dort ein Objekt befindet, wird dieses zerstört. Für das Senden des Strahls nimmt das Skript zunächst die eigene Position und die Ausrichtung und wandelt die Sichtrichtung in einen globalen Richtungsvektor. Beide werden einer Ray-Variablen zugewiesen. Wenn der *Raycast* nun ein Objekt trifft, dann gibt die *RaycastHit*-Variable Informationen über den getroffenen *Collider* zurück, um damit dann dessen *GameObject* zu zerstören.

## ■ 8.8 Character Controller

Auch wenn Spiele häufig realistisch wirken sollen, so agiert die Spielesteuerung meist doch eher unrealistisch und physikalisch inkorrekt. Dies ist aber nicht mangelnder Fähigkeiten der Entwickler geschuldet, sondern ganz allein dem Spielspaß, der ja auch nicht zu kurz kommen sollte. Die klassischen Shooter sind hierfür ein gutes Beispiel. Hier kann der Spieler aus dem Sprint heraus augenblicklich stehen bleiben, sich auf dem Punkt um 90 Grad drehen und auf eine Anhöhe springen. Für solche Verhaltensweisen stellt Unity die *Character Controller*-Komponente zur Verfügung. Diese fügt dem Objekt einen kapselförmigen *Collider* zu sowie Eigenschaften und Funktionen, um diesen zu bewegen und zu steuern.

*Character Controller* eignen sich sowohl für First Person Controller wie auch für Third Person Controller. In den *Standard Assets* finden Sie hierfür bereits zwei fertige *Prefabs*, die beide Einsatzmöglichkeiten demonstrieren. Aber auch für Spiele in der 2D- und 2,5D-Sicht, wie z.B. Sidescrolller, eignen sich diese hervorragend zur Spielesteuerung.

Der *Character Controller* stellt hierbei folgende Parameter im *Inspector* zur Verfügung:

- **Height** bestimmt die Höhe des *Colliders*.
- **Radius** legt den Radius des *Colliders* fest.
- **Slope Limit** legt die maximale Steigung fest, die der *Controller* hochgehen kann.
- **Step Offset** legt die Stufenhöhe fest, die maximal vom *Character Controller* überwunden werden kann.
- **Min Move Distance** legt einen Grenzwert für Bewegungen fest. Wenn sich der *Input* darunter befindet, wird der *Controller* nicht bewegt. In den meisten Fällen wird aber der Default-Wert 0 nicht verändert.
- **Skin Width** legt einen Toleranzbereich um den *Collider* herum fest, in dem sich der *Character Controller* mit anderen *Collidern* teilweise überlappen kann. Ein größerer Wert sorgt für weichere Übergänge, aber auch für ein nicht so gutes Erscheinungsbild. Ein kleinerer Wert sieht wiederum besser aus, kann aber dazu führen, dass der *Character Controller* bereits an kleinen Hindernissen hängen bleibt. Häufig empfiehlt sich ein Wert von 10 % des Radius.
- **Center** legt einen Offset für die *Collider*-Position fest.

Aber auch per Skript können Sie natürlich auf den Character Controller zugreifen. So bietet dieser gleich zwei unterschiedliche Methoden an, um diesen zu steuern. Je nach Komplexität der Steuerung können Sie zwischen **SimpleMove** und **Move** wählen.

### 8.8.1 SimpleMove

Mit dieser Methode können Sie auf einfache Weise eine Spielersteuerung entwickeln. Sie übergeben dieser lediglich einen *Vector3*-Wert, der die Richtung und die Geschwindigkeit in Metern beschreibt, sodass Sie sich keine Gedanken um *Frames* machen müssen. Auch eine Gravitationskraft brauchen Sie nicht hinzuzufügen, da sich die Methode selber darum kümmert. Dieser Automatismus hat aber auch einen Nachteil: Er ignoriert alle Y-Eingaben, sodass Sie mit dieser Methode keinen *Controller* erstellen können, der springen kann.

**Listing 8.23** Character Controller-Skript mit SimpleMove

```
using UnityEngine;
using System.Collections;
public class CCSimpleMove : MonoBehaviour {
    public float speed = 5.0F;
    private Vector3 moveDirection = Vector3.zero;
    private CharacterController controller;
    void Start() {
        controller = GetComponent<CharacterController>();
    }

    void Update() {
        moveDirection = new Vector3(Input.GetAxis("Horizontal"),
                                    0, Input.GetAxis("Vertical"));
        moveDirection = transform.TransformDirection(moveDirection);
    }
}
```

```

        moveDirection = moveDirection * speed;
        controller.SimpleMove(moveDirection);
    }
}

```

Das Skript nutzt unter anderem die `TransformDirection`-Methode der `Transform`-Klasse. Diese wandelt einen lokalen `Vector3`-Wert in einen globalen Wert um. In diesem Fall nimmt sie die Richtung, in die der Spieler schaut, und wandelt diese in einen globalen Richtungsvektor.

## 8.8.2 Move

Mit der `Move`-Methode sind komplexere Bewegungen möglich, die unter anderem auch das Springen unterstützen. Allerdings müssen Sie sich hier auch um die Gravitation kümmern und die *Frames* in Form von `deltaTime` berücksichtigen. Das folgende Skript realisiert die gleiche Steuerung wie das `SimpleMove`-Beispiel, nur mit dem Unterschied, dass dieses noch Springen unterstützt.

**Listing 8.24** Character Controller-Skript mit Move

```

using UnityEngine;
using System.Collections;
public class CCMove : MonoBehaviour {
    public float moveSpeed = 6.0F;
    public float jumpPower = 5.0F;
    public float gravity = 12F;
    private Vector3 moveDirection = Vector3.zero;
    private CharacterController controller;
    void Start() {
        controller = GetComponent<CharacterController>();
    }

    void Update() {
        if (controller.isGrounded)
        {
            moveDirection = new Vector3(Input.GetAxis("Horizontal"),
                                         -1, Input.GetAxis("Vertical"));
            moveDirection = transform.TransformDirection(moveDirection);
            moveDirection = moveDirection * moveSpeed;
            if (Input.GetButtonDown("Jump"))
                moveDirection.y = jumpPower;
        }
        else
        {
            moveDirection.y -= gravity * Time.deltaTime;
        }
        controller.Move(moveDirection * Time.deltaTime);
    }
}

```

Mithilfe der oben genutzten `isGrounded`-Eigenschaft des *Character Controllers* stellen Sie fest, ob der *Collider* vom *Controller* während der letzten Bewegung Berührungen mit dem Boden hatte.

### 8.8.3 Kräfte zufügen

Da ein *GameObject* mit einem *Character Controller* kein *Rigidbody* besitzt, reagiert das Objekt nicht auf andere Kräfte, kann aber auch gleichzeitig nicht ohne Weiteres anderen *Rigidbodies* Kräfte zuführen. Mit etwas C#-Code ist dies aber leicht möglich. Hierbei können Sie die *OnControllerColliderHit*-Methode nutzen, die ausgelöst wird, wenn ein *Character Controller* während einer Bewegung mit einem anderen *Collider* kollidiert.

**Listing 8.25** Mit einem Character Controller andere Objekte schieben

```
using UnityEngine;
using System.Collections;
public class PushObjects : MonoBehaviour {
    public float power = 20.0F;
    void OnControllerColliderHit(ControllerColliderHit hit) {
        Rigidbody body = hit.collider.attachedRigidbody;
        if (body == null || body.isKinematic)
            return;

        if (hit.moveDirection.y < -0.3F)
            return;

        Vector3 direction = new Vector3(hit.moveDirection.x, 0, hit.moveDirection.z);
        body.AddForce(direction * power);
    }
}
```

### 8.8.4 Einfacher First Person Controller

Ein typischer Einsatz eines *Character Controllers* ist die Spielersteuerung bei Spielen aus der Ich-Perspektive. Klassische First Person Games nutzen hierbei meistens die *WASD-Tastensteuerung* kombiniert mit den **[Q]**- und **[E]**-Tasten zum Rotieren. So eine Steuerung wollen wir im Folgenden nachbauen. Erstellen Sie hierfür ein neues Skript mit dem Namen „*PlayerController*“. Da häufig in diesen Spielen der Spieler nicht zu springen braucht, bietet sich hier die *SimpleMove*-Methode des *Character Controllers* an.

Als Grundlage werden wir den Code aus dem obigen *SimpleMove*-Beispiel nehmen und diesen noch um die Drehung ergänzen. In der *Update*-Methode werden wir deshalb den Tastendruck der Drehungstasten mithilfe der *GetButtonDown*-Methode auswerten. Wenn eine gedrückt wurde, werden wir eine Variable, die die Zielausrichtung speichert, um 90° in die jeweilige Richtung drehen. Diese Art des Drehens ist typisch für klassische Rollenspiele und FPS. Zum Speichern der Drehung nutzen wir hierbei eine Variable vom Typ *Quaternion* (*destRotation*). Da ein Drehen am einfachsten mit *Vector3*-Werten geht, nutzen wir für die Zuweisung die *eulerAngles*-Eigenschaft der *Quaternion*-Instanz, die die Drehwinkel im *Vector3*-Format zurückgibt.

**Listing 8.26** Tastenauswertung zum Drehen eines Controllers

```
if(Input.GetButtonDown("Left"))
    destRotation.eulerAngles = destRotation.eulerAngles - new Vector3(0, 90, 0);

if(Input.GetButtonDown("Right"))
    destRotation.eulerAngles = destRotation.eulerAngles + new Vector3(0, 90, 0);
```

Damit der Spieler bzw. die *Transform*-Komponente nicht schlagartig gedreht wird, nutzen wir hierfür die statische *RotateTowards*-Methode der *Quaternion*-Klasse. Mit dieser drehen wir in der Update-Funktion das Transform Schritt für Schritt zu der jeweiligen Zielausrichtung.

**Listing 8.27** Drehen eines Transform mit RotateTowards

```
float step = rotationSpeed * Time.deltaTime;
transform.rotation = Quaternion.RotateTowards(transform.rotation,
                                              destRotation, step);
```

Der erste Parameter der *RotateTowards*-Methode legt dabei die Ausgangsrotation fest, der zweite die Zielrotation und der dritte Parameter legt die Schrittweite der Drehung fest, wodurch die Geschwindigkeit der Drehung gesteuert wird. Um hier *frameunabhängig* zu sein, wird dieser Wert vorher noch einmal mit *Time.deltaTime* multipliziert.

**Listing 8.28** Fertiger First Person Controller

```
using UnityEngine;
using System.Collections;
public class PlayerController : MonoBehaviour {

    public float moveSpeed = 5.0F;
    public float rotationSpeed = 300.0F;
    private Vector3 moveDirection = Vector3.zero;
    private CharacterController controller;
    private Quaternion destRotation;

    void Start() {
        controller = GetComponent<CharacterController>();
        destRotation = transform.rotation;
    }

    void Update() {
        moveDirection = new Vector3(Input.GetAxis("Horizontal"),
                                   0, Input.GetAxis("Vertical"));
        moveDirection = transform.TransformDirection(moveDirection);
        moveDirection = moveDirection * moveSpeed;
        controller.SimpleMove(moveDirection);

        if(Input.GetButtonDown("Left")) {
            destRotation.eulerAngles = destRotation.eulerAngles - new Vector3(0, 90, 0);
        }

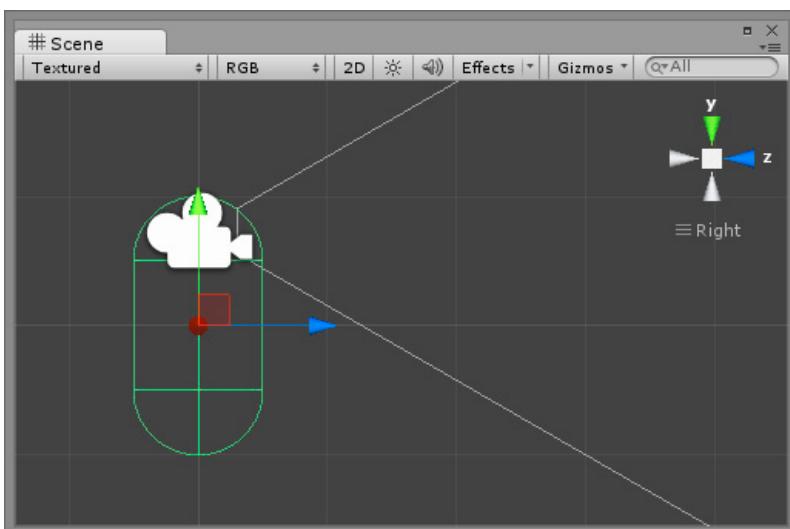
        if(Input.GetButtonDown("Right")) {
            destRotation.eulerAngles = destRotation.eulerAngles + new Vector3(0, 90, 0);
        }
    }
}
```

```

        float step = rotationSpeed * Time.deltaTime;
        transform.rotation = Quaternion.RotateTowards(transform.rotation,
                                                      destRotation, step);
    }
}

```

Um das obige Skript zu nutzen, fügen Sie ein *Empty Object* der Szene zu, das Sie „Player“ nennen. Fügen Sie „Player“ nun einen *Character Controller* und das obige Skript zu. Nun ziehen Sie jetzt die *Main Camera* auf „Player“ und zentrieren diese mithilfe der *Reset*-Funktion im Zahnrad-Menü. Jetzt sollte auch die Kamera so ausgerichtet sein, dass sie nach vorne zeigt. Danach können Sie die Kamera noch etwas in der Y-Richtung anheben, damit die Kamera nicht gerade auf Bauchhöhe, sondern eher auf Augenhöhe des *Character Controllers* liegt (siehe Bild 8.15).



**Bild 8.15** Character Controller mit Kamera

Damit die Steuerung auch tatsächlich funktioniert, müssen zuvor natürlich noch die neuen Inputs „Left“ und „Right“ in den *Input Settings* definiert werden. Ansonsten werden die Tasten **Q** und **E** nichts bewirken. Die *Input Settings* erreichen Sie über **Edit/Project Settings/Input**. Mehr dazu erfahren Sie im Kapitel „Maus, Tastatur, Touch“.

Lizenziert für gregwyss@hotmail.com.

© 2014 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

# 9

# Maus, Tastatur, Touch

Das Hauptmerkmal eines Computerspiels gegenüber anderen digitalen Medien ist die Interaktionsfähigkeit des Inhalts mit dem Spieler.

Unity stellt eine ganze Palette an Möglichkeiten bereit, damit Sie Aktionen des Nutzers auswerten und weiterverarbeiten können. Dies beginnt mit den normalen Eingaben über eine Tastatur und die Maus, über Joystick-Kommandos bis hin zu Touch-Eingaben und Sonderformen wie beispielsweise die Auswertung von Beschleunigungssensoren oder des Kompasses eines mobilen Endgerätes. In diesem Kapitel möchte ich Ihnen die gängigsten Funktionen vorstellen.

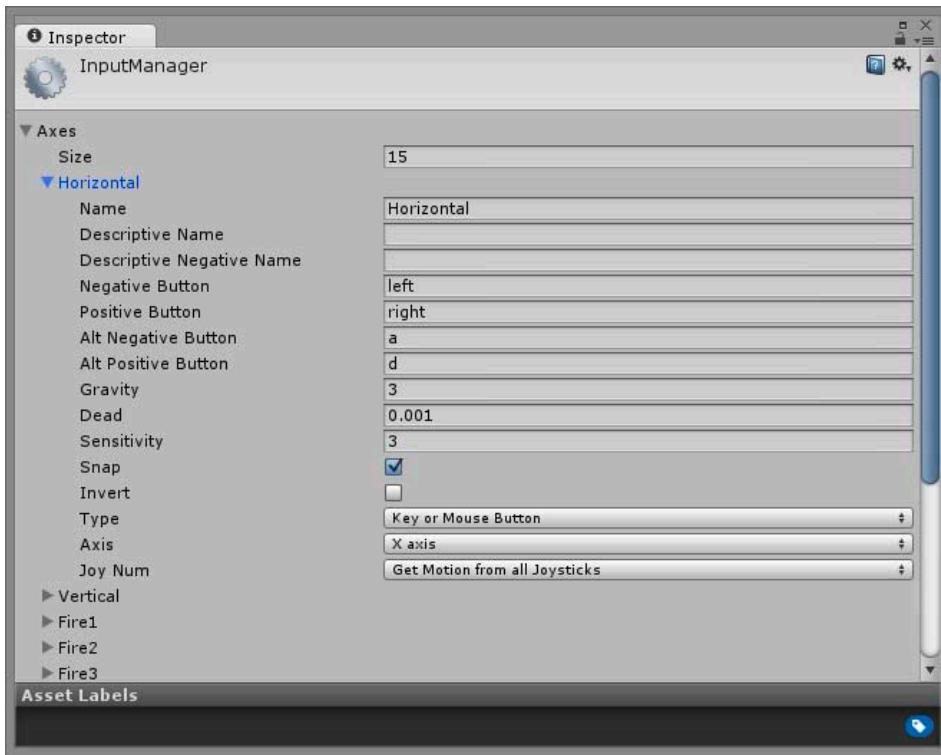
## ■ 9.1 Virtuelle Achsen und Tasten

Anstatt sich von vornherein auf Steuerungselemente und Tasten festzulegen, bietet Unity die Möglichkeit, die Spielesteuerung über virtuelle Achsen und Tasten abzubilden. Das bedeutet, dass Sie über ein Mapping in den Einstellungen eines sogenannte *Input-Managers* jederzeit die Tasten, ja sogar die kompletten Eingabegeräte (Joystick, Maus etc.) austauschen können und auf diese Weise recht flexibel sind. Sollte Ihr Spiel zudem über die Tastatur gesteuert werden, können Sie auch immer noch zusätzliche Alternativ-Tasten definieren, die die gleichen Funktionen übernehmen wie die eigentlichen Standardtasten.

### 9.1.1 Der Input-Manager

Über **Edit/Project Settings/Input** erreichen Sie den *Input-Manager*. Dieser dient dem Zweck, die virtuellen Steuerungsachsen eines Unity-Projektes zu definieren. Sie können alle Belegungen nach Belieben ändern, erweitern oder auch löschen.

Wenn Sie ein neues Unity-Projekt erstellen, erzeugt Unity hier bereits für die wichtigsten Steuerungsbefehle wie zur Navigation, zum Schießen oder auch zum Springen virtuelle Achsen und Tasten. Auch wenn diese für viele Spiele bereits ausreichen können, können Sie auch diese natürlich hier nach Belieben verändern.



**Bild 9.1** Input-Manager

Jede Achse besitzt mehrere Parameter, wobei diese nur nach Bedarf ausgefüllt werden müssen.

- **Name** legt die Bezeichnung fest, über die der Wert dieser Achse abgefragt wird. Auch wenn dies ein frei wählbarer Begriff ist, ist es für eine gute Lesbarkeit des Codes empfehlenswert, wenn dieser die Funktionalität widerspiegelt, die hierüber gesteuert werden soll.
- **Descriptive Name** ist besonders bei Stand-alone-Spielen wichtig. Hier kann der Spieler selbst die Tastenbelegungen wählen. Dieser Text wird als Beschreibungstext für die positive Achsenbezeichnung angezeigt.
- **Descriptive Negative Name** ist ebenfalls bei Stand-alone-Spielen wichtig. Dort wird dies als Beschreibungstext für die negative Achsenbezeichnung angezeigt.
- **Negative Button** bestimmt die Taste für den negativen Achsenwert.
- **Positive Button** bestimmt die Taste für den positiven Achsenwert.
- **Alt Negative Button** bestimmt eine alternative Taste für den negativen Achsenwert.
- **Alt Positive Button** bestimmt eine alternative Taste für den positiven Achsenwert.
- **Gravity** Geschwindigkeit in Einheiten pro Sekunde, in der der Wert auf 0 zurückfällt.
- **Dead** Alle Werte um diesen Toleranzwert herum werden auf 0 gesetzt.
- **Sensitivity** Geschwindigkeit in Einheiten pro Sekunde, in der der Wert auf den Zielwert ansteigt.

- **Snap** Wenn aktiv, dann wird der Achsenwert auf 0 zurückgesetzt, sobald die entgegengesetzte Richtung gedrückt wird.
- **Invert** Wenn aktiv, dann werden alle Werte invertiert.
- **Type** Eingabetyp (Taste, Maustaste, Mausbewegung oder Joystick-Achse)
- **Axis** Sollte ein externes Gerät (Joystick) benutzt werden, ist hier die Achse des verbundenen Steuergerätes auszuwählen.
- **Joy Num** Wenn mehrere Joysticks verbunden sind, ist hier der jeweilige Joystick auszuwählen.

### 9.1.2 Virtuelle Achsen

Achsen bzw. *Axes* zeichnen sich dadurch aus, dass sie sowohl positive Werte wie auch negative Werte annehmen können. Ein gutes Beispiel hierfür ist die horizontale Steuerung eines Spielers, die meistens sowohl nach links wie auch nach rechts verlaufen kann.

Achsen, die über Tastatureingaben gesteuert werden, werden über zwei Tasten gesteuert. Die eine Taste gibt einen Wertebereich von 0 bis -1 zurück, die andere gibt Werte von 0 bis +1 zurück. Unity weist den Parametern *Gravity* und *Sensitivity* hierbei standardmäßig etwas geringere Werte zu, um so einen gewissen Dämpfungseffekt zu erzielen. Dadurch wird bei einem Tastendruck nicht auf Anhieb der Zielwert erreicht, sondern etwas langsamer (ähnlich wie bei einem Joystick). Bei Steuerungen (egal ob Spielfiguren oder Fahrzeuge) ist dieses Verhalten meist sehr sinnvoll.

Unity definiert beim Erstellen eines neuen Projektes die folgenden zwei Achsen:

- **Horizontal** wird mit der Taste [links] (*Negative Button*) und der Taste [rechts] (*Positive Button*) sowie den alternativen Tasten [A] (*Alt Negative Button*) und [D] (*Alt Positive Button*) vorbelegt.
- **Vertical** wird mit der Taste [unten] (*Negative Button*) und der Taste [hoch] (*Positive Button*) sowie den alternativen Tasten [S] (*Alt Negative Button*) und [W] (*Alt Positive Button*) vorbelegt.

### 9.1.3 Virtuelle Tasten

Tasten sind im Grunde auch nur Achsen, die allerdings später über eine andere Methode ausgewertet werden (können). Außerdem wird diesen Achsen nur die positive Richtung einer Taste zugewiesen, sodass der Wert lediglich zwischen 0 und 1 rangieren kann. Zudem werden hier die Werte *Gravity* und *Sensitivity* sehr hoch gesetzt, um so auch vom Wert her einen sprunghaften Anstieg zu erzielen.

Unity definiert beim Erstellen eines neuen Projektes die folgenden Tasten:

- **Jump** wird mit der typischen -Taste (*Positive Button*) angesteuert.
- **Fire1** liegt auf der linken -Taste (*Positive Button*) sowie der linken Maustaste (*Alt Positive Button*).

- **Fire2** wird mit der linken **[Alt]**-Taste (*Positive Button*) und alternativ mit der rechten Maustaste (*Alt Positive Button*) ausgelöst.
- **Fire3** liegt auf der linken **[Umsch]**-Taste (*Positive Button*) und der mittleren Maustaste (*Alt Positive Button*).

## 9.1.4 Steuern mit Mauseingaben

Achsen, die Sie über Mausbewegungen steuern, nehmen eine Sonderrolle ein. Diese haben zwei Besonderheiten, die essenziell sind.

Im Gegensatz zu Tastatur- und Joystick-Eingaben können Rückgabewerte einer Mausbewegung einen höheren Wert als 1 und auch einen geringeren als -1 haben. Dies liegt an der Berechnung des Rückgabewertes der Mausbewegungen. Dieser wird nämlich aus den unterschiedlichen Mauspositionen zwischen dem aktuellen und dem vorherigen Frame berechnet. Wird die Maus also sehr schnell bewegt, können die Mauspositionen zwischen den einzelnen Frames so weit auseinander liegen, dass sich die Werte außerhalb des normalen Bereichs von -1 und 1 befinden.

Die zweite Besonderheit basiert auf dieser Vorgehensweise. Denn weil bei dieser Vorgehensweise die Framedauer schon berücksichtigt wird, brauchen Sie diese Werte nicht mit einem zusätzlichen `deltaTime`-Wert der `Time`-Klasse zu multiplizieren, wie es bei Tastatureingaben der Fall ist. Was das genau für die Praxis bedeutet, erfahren Sie im Abschnitt „Achsen- und Tasteneingaben auswerten“.

Unity definiert beim Erstellen eines neuen Projektes die folgenden drei Achsen bzw. Tasten, die ausschließlich durch die Maus gesteuert werden:

- **Mouse X** ergibt sich aus der Links- und Rechtsbewegung der Maus (*Type: Mouse Movement, Axis: X axis*).
- **Mouse Y** ergibt sich aus der Vorne- und Hintenbewegung der Maus (*Type: Mouse Movement, Axis: Y axis*).
- **Mouse ScrollWheel** wird vom Drehen des Mausrads gesteuert (*Type: Mouse Movement, Axis: 3rd axis bzw. Scrollwheel*).

## 9.1.5 Joystick-Inputs

Unity legt neben den Tastatureingaben auch Joystick-Kommandos fest, die ebenfalls die obigen Tasten und Achsen ansprechen können.

Da sich die Controller aber stark voneinander unterscheiden können (Achsen, Knöpfe, Joystick-Anzahl), werden Sie hier meistens controller-spezifisch noch nachjustieren müssen. Aus diesem Grund werden wir Joysticks hier auch nicht näher behandeln. Sollten Sie aber einen PC-tauglichen Joystick besitzen, können Sie gerne später die *Inputs* des Beispiel-Games auf Joystick umstellen und damit testen. Die wichtigen Parameter im *Input-Manager* sind hierfür:

- **Type** stellen Sie auf *Joystick Axis*.
- **Axis** legt eine Achse des Joysticks fest.
- **Joy Num** bestimmt den Joystick des Controllers.

### 9.1.6 Anlegen neuer Inputs

Im Folgenden werden wir zwei neue Tasten/Buttons anlegen, die wir auch im Kapitel „Physik in Unity“ für einen „Einfachen First Person Controller“ nutzen wollen. Um neue *Inputs* zu definieren, starten Sie zunächst über **Edit/Project Settings/Input** den *Input-Manager*.

1. Klappen Sie den *Axes*-Knoten auf und erhöhen Sie die Size um zwei (z. B. von 15 auf 17). Hinter der letzten *Axe*-Definition werden zwei weitere angelegt, wobei die Eigenschaften inklusive *Name* einfach vom vorherigen übernommen werden.
2. Klappen Sie den ersten Knoten auf und geben Sie diesem den neuen Namen „Left“.
3. Entfernen Sie alle anderen *Name*-Eigenschaften und Button-Zuweisungen und weisen Sie schließlich der Eigenschaft *Positive Button* ein „q“ zu.
4. Ansonsten sollten nur noch *Gravity* und *Sensitivity* mit 1000 ausgefüllt sein und *Dead* einen Wert von 0.001 besitzen. Außerdem sollte *Type* auf *Key or Mouse Button* stehen.
5. Beim zweiten Button weisen Sie den *Namen* „Right“ zu und wiederholen dort Punkt 3, nur dass Sie hier für *Positive Button* ein „e“ zuweisen. Auch hier sollte Punkt 4 zutreffen.

Um die spätere Steuerung des First Person Controllers noch etwas zackiger zu gestalten, können Sie zusätzlich bei den Achsen „Horizontal“ und „Vertical“ die Werte *Gravity* und *Sensitivity* auf 10 anheben.

## ■ 9.2 Achsen- und Tasteneingaben auswerten

Da virtuelle Achsen und Buttons sowohl Tastatureingaben, Joystick- oder auch Maussteuerungen sein können, stellt die Klasse *Input* besondere statische Methoden bereit, die für diese Abfragen genutzt werden können.

### 9.2.1 GetAxis

Die Methode *GetAxis* gibt den aktuellen Wert einer Achse zurück. Werte von Tastatureingaben, die in der *Update*-Methode ausgewertet werden, werden für gewöhnlich noch einmal mit dem Wert von *Time.deltaTime* multipliziert, um einen frame-unabhängigen Wert zu erhalten. Da der Wert von Mausbewegungen bereits das Wechseln der Frames berücksichtigt, ist dies dort nicht notwendig.

**Listing 9.1** Eingaben mit GetAxis auswerten

```
public float keySpeed = 5.0F;
public float mouseSpeed = 2.0F;
void Update() {
    float translation = Input.GetAxis("Vertical") * keySpeed * Time.deltaTime;
    transform.Translate(0, 0, translation);
    float h = mouseSpeed * Input.GetAxis("Mouse X");
    float v = mouseSpeed * Input.GetAxis("Mouse Y") * (-1);
    transform.Rotate(v, h, 0);
}
```

**9.2.2 GetButton**

Die Methode `GetButton` gibt einfach nur ein TRUE oder FALSE zurück, ob eine Achse aktuell gedrückt wird. Hierbei spielt der Achsenwert an sich keine Rolle.

Während `GetButton` die gesamte Zeit, in der der Button heruntergedrückt wird, ein TRUE zurückgibt, gibt die Methode `GetButtonDown` nur genau in dem Frame ein TRUE zurück, in dem der Button gerade heruntergedrückt wird. `GetButtonUp` ist das Gegenstück dazu und gibt nur in dem Frame ein TRUE zurück, in dem der Nutzer die Taste wieder hochnimmt.

**Listing 9.2** Eingaben mit GetButton auswerten

```
public bool autoFire = false;
void Update() {
    autoFire = false;
    if (Input.GetButtonDown("Fire1"))
        audio.Play();
    if (Input.GetButton("Fire1"))
        autoFire = true;
    if (Input.GetButtonUp("Fire1"))
        audio.Stop();
}
```

**■ 9.3 Tastatureingaben auswerten**

Neben virtuellen Achsen können Sie auch direkt Tastatureingaben abfragen und auswerten. Hierfür stellt die Klasse `Input` mehrere statische Funktionen und Variablen zur Verfügung. Die am meist verwendeten Funktionen möchte ich hier kurz vorstellen.

### 9.3.1 GetKey

Die GetKey-Methode arbeitet analog zu GetButton. Allerdings übergeben Sie hier nicht einen Achsenamen, sondern einen KeyCode oder den auszuwertenden Buchstaben als String. GetKey gibt dann wie GetButton ein TRUE zurück, solange die übergebene Taste gedrückt wird.

Die Methode GetKeyDown gibt nur in dem Frame ein TRUE zurück, in dem der Nutzer den Button herunterdrückt. GetKeyUp gibt schließlich nur in dem Frame ein TRUE zurück, in dem der Nutzer die Taste wieder hochnimmt.

Das folgende Beispiel zeigt die Übergabe eines Buchstabens als String wie auch als KeyCode. Bitte achten Sie bei der String-Übergabe darauf, dass Sie den Kleinbuchstaben nehmen.

**Listing 9.3** Eingaben mit GetKey auswerten

```
public bool autoFire = false;
void Update()
{
    autoFire = false;
    if (Input.GetKeyDown(KeyCode.A))
        audio.Play();
    if (Input.GetKey(KeyCode.A))
        autoFire = true;
    if (Input.GetKeyUp("a"))
        audio.Stop();
}
```

### 9.3.2 anyKey

Über die Eigenschaft anyKey können Sie auswerten, ob irgendeine Taste, egal ob Maus oder Tastatur, heruntergedrückt ist. Die Eigenschaft anyKeyDown arbeitet ähnlich, nur gibt diese Eigenschaft lediglich in dem Frame ein TRUE zurück, in dem die Taste gerade heruntergedrückt wird. Das erste Beispiel wertet anyKey aus und führt nur dann die Methode DoSomething aus, wenn keine Taste gedrückt wird.

**Listing 9.4** Eingabe aller Tasten mit anyKey überwachen

```
void Update() {
    if (!Input.anyKey)
        //tu was
}
```

Das zweite Beispiel lädt die Szene 1, sobald eine Taste gedrückt wird.

**Listing 9.5** Laden einer Szene bei Tastendruck

```
void Update() {
    if (Input.anyKeyDown)
        Application.LoadLevel (1);
}
```

## ■ 9.4 Mauseingaben auswerten

Mit Unity können Sie Mausbewegungen und andere Maus-Eigenschaften nicht nur über die virtuellen Achsen auswerten. Sie können dies auch direkt über die Input-Klasse machen.

### 9.4.1 GetMouseButton

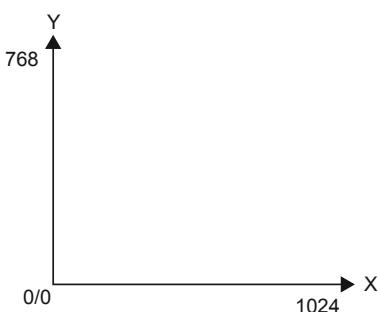
Um den Zustand einer Maustaste abzufragen, übergeben Sie der Methode `GetMouseButton` den Button-Wert, dessen Zustand Sie auswerten möchten. Der Wert `0` spricht die linke Maustaste an, `1` die rechte Maustaste und `2` meint die mittlere Maustaste. Die Methode `GetMouseButton` gibt dann ein `TRUE` zurück, wenn die übergebene Maustaste aktuell heruntergedrückt wird. Die Methode `GetMouseDown` gibt nur in dem Frame ein `TRUE` zurück, in dem die Maustaste gerade heruntergedrückt wird. `GetMouseButtonUp` gibt wiederum nur in dem Frame ein `TRUE` zurück, in dem der Nutzer die Maustaste wieder hochnimmt.

**Listing 9.6** Zustand der linken Maustaste abfragen

```
void Update() {
    if (Input.GetMouseButton(0))
        Debug.Log("Es wird die linke Taste gedrückt.");
}
```

### 9.4.2 mousePosition

Über die `Vector3`-Variable `mousePosition` erhalten Sie die aktuelle Position des Mauszeigers, angegeben in Pixelkoordinaten. Beachten Sie, dass sich hier die Koordinaten von unten links nach oben rechts ausbreiten. Unten links befindet sich also `(0, 0, 0)`, während oben rechts z.B. `(1024, 768, 0)` zu finden ist. Eine Tiefe besitzen die Pixel ebenfalls nicht, weshalb der Z-Wert immer `0` ist.



**Bild 9.2**  
Pixelkoordinatensystem

Sollte Ihr Spielfenster nicht maximiert sein, erhalten Sie über `mousePosition` auch dann Koordinaten, wenn Sie sich mit der Maus außerhalb des Spiels bewegen. Dadurch können sowohl extrem hohe Positionsangaben als auch negative Koordinaten entstehen.

**Listing 9.7** Ausgabe der aktuellen Mouse-Position im Inspector

```
using UnityEngine;
using System.Collections;
public class MousePos : MonoBehaviour {
    public Vector3 pos;
    void Update () {
        pos = Input.mousePosition;
    }
}
```

Mithilfe der Methode `ScreenPointToRay` der Klasse `Camera`, die Sie bereits im Kapitel „Kamera, die Augen des Spielers“ kennengelernt haben, können Sie nun aus dieser speziellen Pixelkoordinate ganz leicht einen Strahl erzeugen, der in die Spielwelt hineinragt, um z.B. Objekte zu selektieren oder andere Befehle auszuführen.

**Listing 9.8** Namen eines angeklickten Objektes in der Konsole ausgeben

```
using UnityEngine;
using System.Collections;
public class InfoClick : MonoBehaviour
{
    void Update()
    {
        if (Input.GetMouseButtonDown(0)) {
            Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
            RaycastHit hit;
            if (Physics.Raycast (ray,out hit)) {
                Debug.Log(hit.collider.gameObject.name);
            }
        }
    }
}
```

### 9.4.3 Mauszeiger ändern

Sollten Sie vorhaben, dem Mauszeiger im Spiel ein eigenes Design zu geben, ist dies sehr einfach möglich. Zum einen können Sie in den *Player Settings* (**Edit/Project Settings/Player**) einen Standard-Cursor hinterlegen.

- **Default Cursor** legt die Grafik des Cursors fest.
- **Cursor Hotspot** dient dem Positionieren dieser Grafik, abhängig von der tatsächlichen Mausposition. Achten Sie hierbei darauf, dass die Verschiebung nach links und nach oben vorgenommen wird. Haben Sie also beispielsweise eine Fadenkreuz-Textur in der Größe  $32 \times 32$ , müssen Sie diese um X 16 und Y 16 verschieben, damit die Texturmitte ebenfalls an der eigentlichen Mausspitze des Cursors erscheint.

Beachten Sie bei den *Import Settings* einer Cursor-Textur, dass der *Texture Type* hier auf *Cursor* steht.

Mithilfe der `SetCursor`-Methode der `Cursor`-Klasse können Sie die `Cursor`-Textur aber auch zur Laufzeit ändern. Übergeben Sie dieser einfach die Cursor-Textur, den bereits oben

beschriebenen Hotspot-Offset sowie die Angabe, ob der Cursor auf Plattform-Ebene oder Software-Ebene ersetzt werden soll.

Wählen Sie den *CursorMode Auto*, wird versucht, dass sich das Betriebssystem um das Darstellen des neuen Cursors kümmert. Das bedeutet eine frame-unabhängige Cursor-Darstellung, die wesentlich flüssiger wirkt als ein Software-Cursor. Da dieses Verfahren aber nur bei bestimmten Plattformen, z.B. den Desktops, unterstützt wird, fällt Unity bei anderen Systemen automatisch auf den Software-Cursor zurück, den Sie über den *CursorMode ForceSoftware* auch direkt wählen könnten.

Das folgende Beispiel wechselt durch das Drücken der virtuellen Taste „Fire2“ zwischen dem Standard-Cursor und einer Fadenkreuz-Textur.

#### **Listing 9.9** Fadenkreuz-Anzeige

```
using UnityEngine;
using System.Collections;
public class CrossHairs : MonoBehaviour {
    public Texture2D cursorTexture;
    Vector2 hotSpot = Vector2.zero;
    bool showCrossHairs = false;
    void Start()
    {
        hotSpot.x = cursorTexture.width / 2;
        hotSpot.y = cursorTexture.height / 2;
    }
    void Update()
    {
        if(Input.GetButtonDown("Fire2"))
        {
            showCrossHairs = !showCrossHairs;
            if(showCrossHairs)
                Cursor.SetCursor(cursorTexture,hotSpot,CursorMode.Auto);
            else
                Cursor.SetCursor(null,hotSpot,CursorMode.Auto);
        }
    }
}
```

## ■ 9.5 Touch-Eingaben auswerten

Da Sie mit Unity auch Spiele für Smartphones und Tablets entwickeln können, werden natürlich auch Touch-Eingaben unterstützt. Hierfür bietet die `Input`-Klasse eigene Methoden und Variablen an, um diese auswerten zu können.

Touch-Eingaben verhalten sich hierbei ähnlich wie ein Mausklick, nur dass Sie bei Touch natürlich bis zu zehn Eingaben gleichzeitig machen können. Deshalb gibt es hier auch einige Parameter mehr, die Sie auswerten können.

### 9.5.1 Der Touch-Typ

Eine Touch-Eingabe wird in Unity durch eine Variable vom Typ **Touch** dargestellt. Eine solche **Touch**-Variable stellt mehrere Parameter zur Verfügung, um Eingaben auch vernünftig auswerten zu können.

- **deltaPosition** gibt einen 2D-Vektor zurück, der beschreibt, welche Positionsänderung der Touch seit der letzten Änderung vollzogen hat (siehe `Touch.deltaTime`). In Kombination mit dem `phase`-Parameter können Sie mit dieser Eigenschaft zum Beispiel Wischrichtungen ermitteln.
- **deltaTime** gibt die Zeitdifferenz zurück, die seit der letzten Touch-Auswertung vergangen ist. Dieser Parameter, zur besseren Unterscheidung auch `Touch.deltaTime` genannt, ist ein anderer Wert als `Time.deltaTime`. Da sie zudem unabhängig von der Frameweitergabe ist, kann sie auch höher als diese liegen.
- **fingerId** ist ein eindeutiger Index der Touch-Eingabe, um die unterschiedlichen Touch-Eingaben zu identifizieren. Nachdem der Finger hochgenommen wird, wird der Wert bei der nächsten Touch-Eingabe wiederverwendet.
- **phase** beschreibt die aktuelle Phase des Touches. Es gibt die unterschiedlichen Phasen *Began*, *Canceled*, *Ended*, *Moved* und *Stationary*. Wenn Sie einen *Touch* mit einem Mausklick vergleichen, dann entspricht *Began* der Methode `GetMouseButton`.
- **position** gibt die aktuelle, absolute Position der Touch-Eingabe in Pixelkoordinaten zurück. Im Vergleich mit der Maus entspricht dieser Wert der Variablen `mousePosition`. Im Kapitel „Künstliche Intelligenz“ finden Sie hierzu eine *Touch*-Adaption des *ScreenPointToRay*-Beispiels aus dem Abschnitt „`mousePosition`“.
- **tapCount** gibt die Anzahl an, wie häufig mit diesem Touch auf den Bildschirm getippt wurde. Dieser Wert ist ein Absolutwert eines Touches. Erst wenn der Touch aufgelöst wird, also wenn der Finger hochgenommen und nicht mehr auf die gleiche Stelle gedrückt wird, wird auf 0 zurückgesetzt. Über diese Eigenschaft können Sie zum Beispiel einen „Doppelklick“ mitteln.

### 9.5.2 Input.touches

Über die statische Variable `touches` der Klasse `Input` erhalten Sie ein Array aller `Touch`-Instanzen des letzten Frames, die Sie direkt über die obigen Eigenschaften auswerten können. Über die Array-Standardeigenschaft `Length` erhalten Sie die Anzahl aller Touch-Eingaben.

**Listing 9.10** Touch-Phase abfragen

```
if (Input.touches.Length > 0)
{
    if (Input.touches[0].phase == TouchPhase.Began)
    {
        //tu was...
    }
}
```

In dem Kapitel „Künstliche Intelligenz“ werde ich dieses kleine Beispiel noch einmal aufgreifen und mit dieser eine touch-basierte Point & Click-Steuerung programmieren.

### 9.5.3 TouchCount

Anstatt die Menge der Touches, wie im obigen Beispiel, über die Length-Eigenschaft des *Touch*-Arrays zu ermitteln, können Sie diese auch direkt von *Input* erfahren. Hierfür steht die Eigenschaft *touchCount* zur Verfügung.

**Listing 9.11** Touch-Anzahl abfragen

```
if (Input.touchCount > 0)
{
    //...
}
```

### 9.5.4 GetTouch

Während Sie über *Input.touches* immer auf das ganze *Touch*-Array zugreifen, können Sie sich mit der Methode *GetTouch* der Klasse *Input* gezielt eine Touch-Eingabe holen und diese weiter verarbeiten.

**Listing 9.12** Alle Touch-Eingaben durchlaufen und die Position abfragen

```
void Update () {
    int count = Input.touchCount;
    for (int i = 0; i < count; i++)
    {
        Touch touch = Input.GetTouch(i);
        if (touch.phase == TouchPhase.Began)
        {
            //Code...
            if (touch.position.x > 100)
            {
                //Code...
            }
        }
    }
}
```

### 9.5.5 Touch-Controls

Nicht selten kommt es vor, dass man zum Steuern eines Games per Touch virtuelle Gamepads nutzen möchte. Um solche Touch-Controls zu erzeugen, benötigen Sie lediglich ein *UIElement* (siehe Kapitel „GUI“). Am besten ist natürlich ein *GUITexture*-Objekt, das einen Button oder Ähnliches darstellt. So ein *GUITexture*-Objekt können Sie ganz leicht über *GameObject/Create Other/GUI Texture* Ihrer Szene zufügen.

Jedes *GUIElement* besitzt eine Methode `HitTest`, die überprüft, ob eine übergebene Pixelkoordinate sich in dem Bereich des eigenen Elementes befindet. Hierüber lassen sich also sowohl eine Mausposition als auch die Position einer Touch-Eingabe überprüfen.

Das folgende Skript durchläuft alle Touch-Eingaben und überprüft mit `HitTest`, ob eine Position auf dem *GUIElement* gemacht wurde, dem dieses Skript angehängt wurde. Wenn nun dies zutrifft, wird die boolesche Variable `isPressed` auf TRUE gesetzt, ansonsten ist sie FALSE. Zum Nutzen dieses Skriptes müssen Sie also zunächst ein *GUIText* oder eine *GUITexture* Ihrer Szene zufügen und ihr dieses Skript anhängen.

#### **Listing 9.13** Touch-Button

```
using UnityEngine;
using System.Collections;

public class TouchControl : MonoBehaviour {
    public bool isPressed = false;
    void Update () {
        isPressed = false;
        int count = Input.touchCount;
        for (int i = 0; i < count; i++)
        {
            Touch touch = Input.GetTouch(i);
            if(guiTexture.HitTest(touch.position))
            {
                isPressed = true;
            }
        }
    }
}
```

## ■ 9.6 Beschleunigungssensor auswerten

Fast jedes Smartphone und Tablet besitzt heutzutage neben einer Touch-Steuerung auch einen Beschleunigungssensor, den Sie ebenfalls für die Steuerung Ihres Spieles nutzen können. Beschleunigungssensoren werden auch Accelerometer genannt und geben Informationen über die Lage des Gerätes zurück, also ob es gekippt oder gedreht gehalten wird.

Ein typisches Anwendungsbeispiel für einen Beschleunigungssensor ist das Anpassen des Displays, wenn das Gerät gedreht wird.



### Display-Orientierungen einschränken

Wenn Ihr Spiel kein automatisches Drehen, sondern nur eine besondere Ansicht (Orientierung) unterstützen soll, können Sie dies in den *Player Settings* (*Edit/Project Settings/Player*) unter *Default Orientation* genau festlegen. Da dies für jede Plattform separat eingestellt werden kann, finden Sie diese Einstellung in den plattformspezifischen Parametern.



**Bild 9.3** Portrait- und Landscape-Ansicht einer Website

### 9.6.1 Input.acceleration

Die Informationen des Beschleunigungssensors erhalten Sie über die statische *Vector3*-Variable *acceleration* der *Input*-Klasse. Dabei entsprechen die Achsen des Beschleunigungsmessers mit einer Ausnahme denen von Unity. So verläuft die X-Achse nach rechts und nach oben geht die Y-Achse. Lediglich die Z-Achse zeigt nicht in den Raum, sondern in die Richtung des Spielers. Dafür brauchen Sie sich aber keine Gedanken darüber zu machen, ob das Spiel nun in der Portrait- oder in der Landscape-Ansicht gespielt wird, darum wird sich bereits im Hintergrund gekümmert.

Jeder Achsenwert von *acceleration* gibt die G-Kraft zurück, die auf die jeweiligen Achsen einwirken. Dabei kann jeder Werte von -1 bis +1 reichen. Hierdurch können Sie mit diesen Parametern natürlich ähnlich verfahren wie mit virtuellen Achsen.

**Listing 9.14** Objekt mit einem Beschleunigungssensor steuern

```
public float speed = 10;
void Update () {
    transform.Translate(Input.acceleration.x * Time.deltaTime * speed, 0, 0);
}
```

Zum Testen des obigen Codes können Sie das Übungsprojekt aus dem Kapitel „Grundlagen“ nutzen. Entfernen Sie hierfür zunächst alle anderen Testskripte vom Cube, die Sie eventuell in vorherigen Beispielen dem Würfel zugefügt haben. Dann erzeugen Sie ein neues Skript, in das Sie den obigen Code einfügen, und hängen dieses dann anschließend dem Cube an. Da wir in diesem Fall zudem den Cube über die *Translate*-Methode bewegen wollen, müssen Sie zusätzlich noch die *Is Kinematic*-Eigenschaft des *Rigidbody*s aktivieren. Beachten Sie, dass Sie zum Testen natürlich ein mobiles Endgerät benötigen und dementsprechend das Projekt auf dieses bringen müssen. Mehr zu diesem Thema erfahren Sie im Kapitel „Spiele erstellen und publizieren“.

## 9.6.2 Tiefpass-Filter

Leider entsprechen die Ausgabewerte der Beschleunigungssensoren nicht immer genau dem, was man als Nutzer erwartet. So kommt es manchmal zu kurzen Fehlsignalen, die dementsprechend zu kurzen Schwankungen und Sprüngen führen, auch *Noise* genannt.

Umso größer nun ein anderer Parameter ist, mit dem Sie den Sensorwert multiplizieren, desto mehr macht sich dies natürlich bemerkbar. Damit dies aber nicht am Ende zu zitternden oder springenden Reaktionen im Spiel führt, werden normalerweise Verfahren eingesetzt, die die Werte bereinigen bzw. glätten.

Eine hier gern eingesetzte Methode ist das Nutzen eines Tiefpass-Filters. Dieser berechnet einen Zwischenwert aus dem aktuellen und dem letzten Wert, wodurch die hohen Ausschläge geglättet werden. Zwar wird dadurch auch gleichzeitig die gesamte Reaktion etwas gedämpft, was aber über Parameter gesteuert werden kann. Mit einem Tiefpass-Filter sieht dann der vorherige Code wie folgt aus.

**Listing 9.15** Beschleunigungssensorauswertung mit Tiefpass-Filter

```
public float speed = 10;
private float lowPassFactor = 0.7F;
private Vector3 accValue = Vector3.zero;
void Start () {
    accValue = Input.acceleration;
}
void Update() {
    CalcLowPassAccValue ();
    transform.Translate(accValue.x * Time.deltaTime * speed, 0, 0);
}
void CalcLowPassAccValue(){
    accValue = Vector3.Lerp(accValue, Input.acceleration, lowPassFactor);
}
```

In dem Beispiel sollte sich die Variable `lowPassFactor`, die die Schrittweite angibt, zwischen den Werten 0.01F und 0.99F bewegen. Umso kleiner dabei der Wert von `lowPassFactor` ist, desto gedämpfter ist die Reaktion auf eine Veränderung beim Beschleunigungssensor.

## ■ 9.7 Steuerungen bei Mehrspieler-Games

Erstellen Sie ein Mehrspieler-Game, stellt sich nun die Frage, wie die unterschiedlichen Spieler die jeweiligen Figuren steuern können. Hier müssen Sie unterscheiden zwischen einem Mehrspieler-Netzwerk-Game und einem Mehrspieler-Game, bei dem die Spieler an einem Rechner sitzen (z.B. Split-Screen-Games). Während bei einem Netzwerkspiel alle Spieler die gleichen Controller-Eingaben nutzen können, müssen sie bei einem Split-Screen-Game verteilt werden.

### 9.7.1 Split-Screen-Steuerung

Bei klassischen Split-Screen-Games (oder auch Kampfsportspielen) teilen sich zwei oder mehr Spieler den gleichen Computer und steuern hierbei unterschiedliche Figuren. Für solche Zwecke müssen natürlich die zu steuernden Figuren auf unterschiedliche Controller bzw. Eingaben reagieren.

Hierfür ist es vorteilhaft, wenn Sie im *InputManager* zusätzliche virtuelle Achsen bzw. Buttons anlegen, z.B. „Horizontal 2“ und „Vertical 2“, denen andere Tasten oder Joysticks etc. zugewiesen werden als „Horizontal“ und „Vertical“. In diesem Fall könnten Sie z.B. die *Alt-Button*-Belegungen den Standard-Inputs wegnehmen und den neuen „2er“-Inputs als Standardbelegungen zuweisen.

Später müssen Sie nur noch dafür sorgen, dass die beiden Figuren auch auf die unterschiedlichen *Inputs* reagieren, sprich der eine auf die Standard-Inputs und der andere auf die „2er“-Inputs. Dies können Sie z.B. dadurch erreichen, wenn Sie die Abfragen der *Input*-Achsen/Buttons mit *public*-Variablen flexibel gestalten und diese dann im *Inspector* entsprechend auswechseln (siehe Listing 9.16).

**Listing 9.16** Input-Abfrage mit auswechselbarer Achsenbezeichnung

```
public string horizontalInput = "Horizontal";

void Update () {
    if (Input.GetAxis(horizontalInput) != 0)
    {
        //Code...
    }
}
```

Ein Skript wie das aus Listing 9.16 müsste dann beiden Spielerfiguren zugefügt werden, während die Variable *horizontalInput* über den *Inspector* einmal den Namen „Horizontal“ und einmal den Namen des anderen Input-Parameters, z.B. „Horizontal 2“, zugewiesen bekäme.

### 9.7.2 Netzwerkspiele

Bei Netzwerk-Games sieht dies etwas anders aus. Hier können alle Spieler die gleichen Inputs nutzen, da an jedem PC immer nur ein einzelner Spieler sitzt. Denn vereinfacht gesagt sind Netzwerkspiele nichts anderes als ganz normale Games, bei denen lediglich einige NPCs nicht durch eine künstliche Intelligenz gesteuert werden, sondern durch eine Synchronisation mit einem anderen Rechner neu berechnet werden. Diese NPCs sind die Figuren der anderen Spieler, die wiederum auf ihrem Rechner diese Figuren steuern und deren Positionen und Handlungen bestimmen.

Am Ende werden meistens aber nicht nur die Figuren an sich synchronisiert, sondern auch die Teile bzw. Elemente einer Szene, die durch die Spieler verändert werden können. Schließlich soll auch das Level bei allen Spielern gleich aussehen.

Das Anspruchsvolle bei solchen Games ist deshalb auch nicht die Steuerung der unterschiedlichen Spieler, sondern die Synchronisation zwischen den teilnehmenden Spieler-PCs. Dies kann dann entweder von einem extra Server übernommen werden oder von einem PC der teilnehmenden Spieler, der dann zusätzlich die Aufgabe des Servers übernimmt. Für solche Netzwerksynchronisationen stellt Unity Ihnen ein eigenes Framework bereit, mit dem Sie dies umsetzen können.

Netzwerkspiele zu erstellen kann sehr aufwendig werden und verlangt einiges an Erfahrung, weshalb ich an dieser Stelle auf den „Network Reference Guide“ verweisen möchte, den Sie im Unity-Manual finden (erreichbar über die Hilfefunktionen von Unity). Dort werden Ihnen die Grundkonzepte sowie die unterschiedlichen Bausteine eines Netzwerk-Games erläutert.

Lizenziert für gregwyss@hotmail.com.

© 2014 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

# 10

## Audio

Ein wichtiger Aspekt eines jeden Computerspiels ist der Sound. Auch wenn dieser gar nicht mal so häufig als essenziell betrachtet wird, spielt er doch eine enorme Rolle beim Erzeugen einer bestimmten Atmosphäre. Stellen Sie sich doch nur mal ein Star Wars-Spiel mit elektronischer anstatt mit orchestraler Musik vor. Oder nehmen wir an, Sie möchten ein Outdoor-Spiel im Wald machen. Da macht es einen großen Unterschied, ob leichtes Blätterschaben im Hintergrund zu hören ist, Vogelgezwitscher, ein Orkan oder vielleicht überhaupt kein Geräusch.

Egal ob Sie nun Hintergrundmusik, Effektgeräusche oder auch GUI-Sounds wie das Klicken eines Buttons in Ihr Spiel integrieren möchten, Sie benötigen immer die folgenden drei Dinge:

- **AudioClip** ist die Audiodatei, die zu hören sein soll.
- **AudioSource** stellt das Wiedergabegerät (mit Lautsprecher) für den Sound dar.
- **AudioListener** ist der Empfänger, der den Sound empfängt und schließlich auch über die Computerboxen ausgibt.

Diese Basiskomponenten wie auch weitere Audio-Komponenten finden Sie im Hauptmenü über **Component/Audio/**.

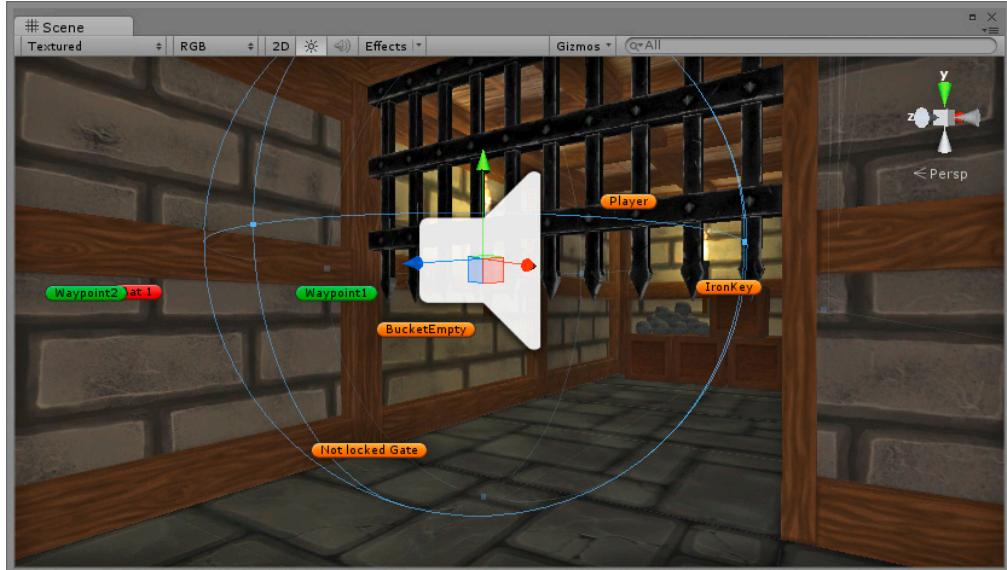
### ■ 10.1 AudioListener

Die *AudioListener*-Komponente stellt die Ohren des Spielers dar. Auch wenn Sie mehreren *GameObjects* diese Komponente zuweisen können, so kann es immer nur einen aktiven *AudioListener* in einer Szene geben. Deswegen achten Sie darauf, dass Sie nur eine aktive in der Szene haben.

Gibt es aber doch mehrere aktive *AudioListener* in einer Szene, so wählt Unity selbstständig einen aus. Da die *Main Camera* bereits einen *AudioListener* besitzt, brauchen Sie sich hierum nur zu kümmern, wenn Sie über das Menü **GameObject/Create Other** der Szene ein weiteres *Camera-GameObject* hinzufügen und dadurch zwei besitzen oder einem *GameObject* gezielt eine *AudioListener*-Komponente über das Menü **Component/Audio/AudioListener** anhängen.

## ■ 10.2 AudioSource

Eine *AudioSource* ist für die Wiedergabe einer Audiodatei zuständig. Am besten kann man sich dies wie einen Lautsprecher vorstellen, aus dem Musik oder ein Sound erklingt. Aus diesem Grund ist auch das Default-Gizmo, das eine *AudioSource* in der *Scene View* darstellt, ein Lautsprecher-Symbol.

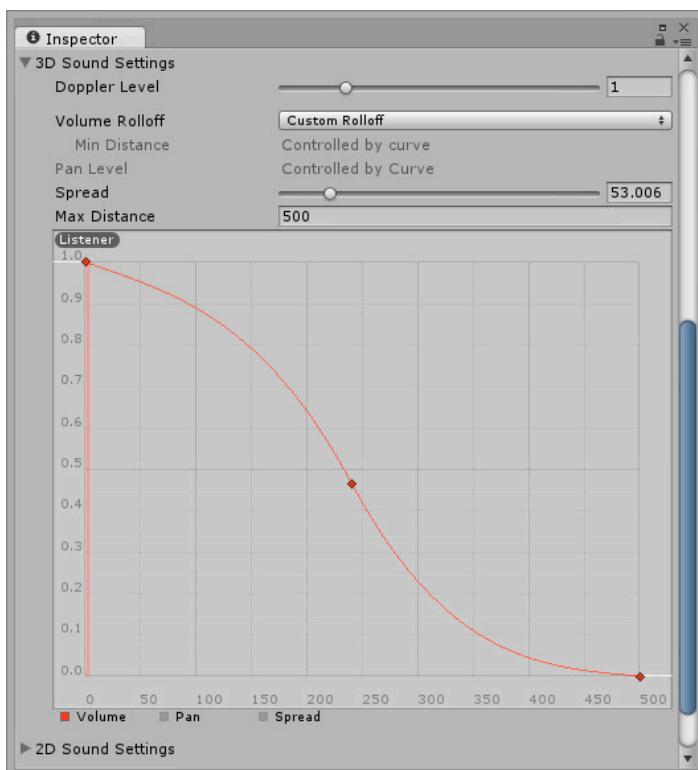


**Bild 10.1** AudioSource in der Scene View

Da *AudioSources* sowohl zum Abspielen von Musik als auch von Effekt-Sounds gedacht sind, müssen diese natürlich auch eine größere Palette von Parametern anbieten:

- **AudioClip** verweist auf die Audiodatei, die abgespielt werden soll.
- **Mute** schaltet den Ton aus. Die Audiodatei wird trotzdem im Hintergrund weiter abgespielt.
- **Bypass Effect/Listener Effects/Reverb Zones** lässt die  *AudioSource* bestimmte Effekte ignorieren.
- **Play On Awake** bedeutet, dass die Datei sofort abgespielt wird, wenn die Komponente aktiv wird.
- **Loop** wiederholt die Musikdatei, sobald das Ende erreicht ist.
- **Priority** legt eine Priorität von 0 (sehr wichtig) bis 256 (unwichtig) fest. Der Standardwert beträgt 128.
- **Volume** legt die Lautstärke der Wiedergabe fest.
- **Pitch** erhöht durch einen höheren Wert die Tonhöhe sowie die Abspielgeschwindigkeit.
- **Doppler Level** legt die Höhe des Dopplereffekts fest. 0 bedeutet kein Effekt.

- **Pan Level** definiert den Einfluss der 3D-Engine auf den Sound. Bei 0 bedeutet dies keinen Einfluss der 3D-Engine, sodass der Sound immer gleich laut ist, egal wo sich der *AudioListener* befindet.
- **Min/Max Distance** legt den Abstand fest, in dem der *AudioListener* diese Quelle hören kann.
- **Rolloff Mode** definiert, wie sich die Entfernung zwischen *AudioListener* und  *AudioSource* auf die Lautstärke auswirkt.
- **Spread** legt den Einfluss der räumlichen Position auf das Stereobild des Klages fest. Bei 0 wird der Klang komplett von der räumlichen Position der  *AudioSource* und des  *AudioListeners* bestimmt. Befindet sich die  *AudioSource* links vom  *AudioListener*, wird der Klang auch links zu hören sein. Bei 180 wird das Stereobild komplett von dem  *AudioClip* an sich bestimmt, unabhängig von den Positionen dieser Komponenten. Bei Hintergrundmusik würde man die einzelnen Instrumente im Stereobild so hören wie auch auf der Stereoanlage. Bei einem Wert von 360 wird schließlich das Stereobild komplett um 180° gedreht. In dem Fall würde der Sound rechts zu hören sein, obwohl sich die  *AudioSource* links vom  *AudioListener* befindet.
- **Pan 2D** stellt das dar, was man normalerweise als Stereoregler bezeichnen würde. Dieser Regler wirkt sich aber nur auf den nicht vom 3D-Raum beeinflussten Anteil eines Sounds aus, z. B. wenn Pan Level auf 0 gestellt oder wenn die Musikdatei kein 3D-Sound ist (siehe  *AudioClip*).



**Bild 10.2** Darstellung der Lautstärke abhängig zur Entfernung

Um das Verhalten entferungsabhängiger Parameter noch detaillierter beschreiben zu können, haben Sie im unteren Bereich zusätzlich noch die Möglichkeit, diese mithilfe von Kurven in einem Diagramm zu definieren.

### 10.2.1 Durch Mauern hören verhindern

Im Gegensatz zur realen Welt, gibt es in Unity keine Schalldämmung oder Ähnliches. So spielt es für den Klang, den der Zuhörer wahrnimmt, keine Rolle, ob sich zwischen der  *AudioSource*  und dem  *AudioListener*  eine Mauer befindet oder nicht.

Wenn aber eine Audioquelle in so einem Fall nicht zu hören sein soll, dann müssen Sie dies mit etwas Programmierung lösen. Eine mögliche Lösung wäre das Nutzen der  *mute* -Eigenschaft, die über ein Skript und einen  *Trigger-Collider*  sehr einfach geschaltet werden kann.

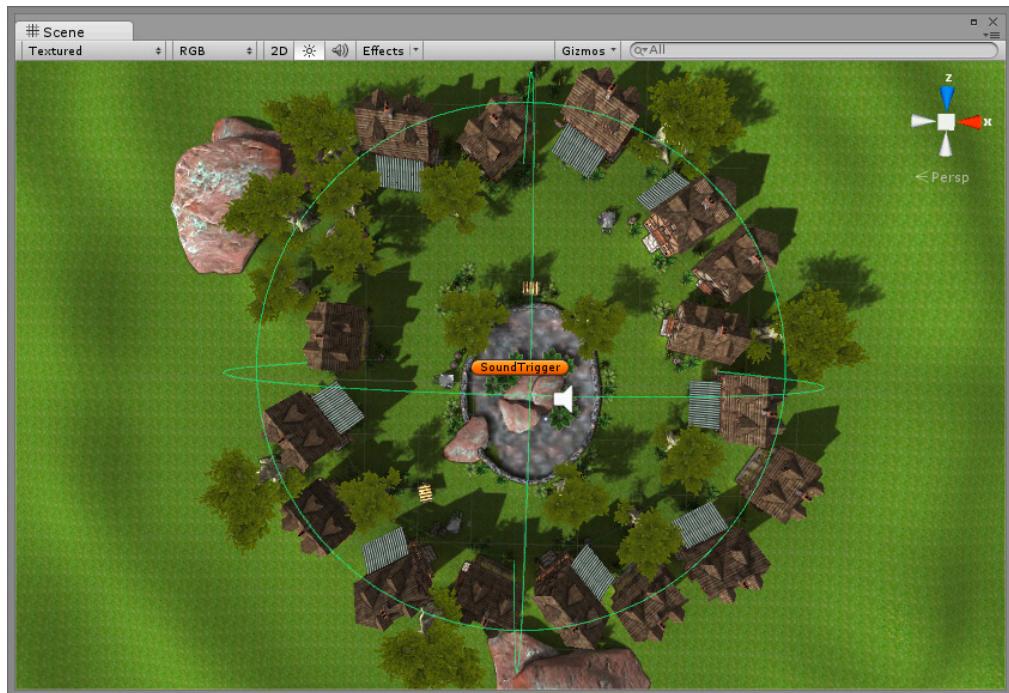
**Listing 10.1** Per Trigger-Collider die Mute-Eigenschaft schalten

```
using UnityEngine;
using System.Collections;
public class AudioArea : MonoBehaviour {
    public AudioSource source;
    public bool muteOnAwake = false;
    void Awake()
    {
        source.mute = muteOnAwake;
    }
    void OnTriggerEnter(Collider other) {
        if(other.CompareTag("Player"))
            source.mute = false;
    }
    void OnTriggerExit(Collider other) {
        if(other.CompareTag("Player"))
            source.mute = true;
    }
}
```

Das obige Skript  *AudioArea*  schaltet den  *Mute* -Parameter einer  *AudioSource* , je nachdem, ob ein anderes Objekt mit dem Tag „Player“ in einen  *Trigger-Collider*  hineintritt oder diesen verlässt, ein oder aus. Die  *AudioSource*  selber muss also für dieses Skript bereits gestartet sein, z.B. über den Parameter  *Play On Awake* .

Zum Nutzen dieses Skriptes fügen Sie Ihrer Szene ein neues  *Primitive-GameObject*  zu (z.B. einen Cube oder eine Sphere) und skalieren Sie dieses so, dass es den Raum beschreibt, in dem die  *AudioSource*  zu hören sein soll. Nun deaktivieren Sie den  *Renderer* , aktivieren den  *IsTrigger* -Parameter und fügen diesem das obige Skript zu. Auf den  *source* -Parameter ziehen Sie nun die  *AudioSource* , die Sie steuern möchten. Über den Parameter  *muteOnAwake*  stellen Sie ein, ob die  *AudioSource*  bereits beim Spielbeginn still geschaltet sein soll oder zu hören ist.

Da hier nur die  *Mute* -Eigenschaft geschaltet und nicht die  *Start* -Funktion genutzt wird, sollte bei diesem Mechanismus die  *Loop* -Eigenschaft der  *AudioSource*  aktiviert werden. Ansonsten könnte es sein, dass die  *AudioSource*  zwar aktiviert wird, aber die Sounddatei bereits zu Ende abgespielt wurde.



**Bild 10.3** Runder Trigger-Collider zum Schalten der AudioSource in der Mitte

In Bild 10.3 sehen Sie eine  *AudioSource*, die das Geplätscher eines kleinen Springbrunnens abspielt. Durch den runden „ *SoundTrigger*“ wird das Geräusch erst hörbar, sobald der Spieler in das Dorf hineintritt, außerhalb ist es nicht zu hören.

## 10.2.2 Sound starten und stoppen

Wie den  *mute*-Parameter können Sie natürlich auch alle anderen Eigenschaften per Code setzen. Zudem gibt es noch weitere Funktionen und Eigenschaften, die nicht im  *Inspector* zur Verfügung stehen, aber per Skript angesteuert werden können. Die wohl am meisten genutzten Methoden sind hierbei sicherlich  *Play* und  *Stop*, um das Abspielen des Sounds zu starten und zu unterbrechen.

Wie auch beim  *RigidBody* unterstützt die  *MonoBehaviour*-Klasse hier den Schnellzugriff über eine Variable, um auf die eigene  *AudioSource* zuzugreifen. Diese lautet aber nicht „ *audioSource*“, wie man denken könnte, sondern  *audio*. Das folgende Beispiel demonstriert sowohl diesen Schnellzugriff als auch die Methoden  *Play* und  *Stop*. Zusätzlich zeigt sie die *.isPlaying*-Eigenschaft, die überprüft, ob die Datei aktuell abgespielt wird. Wie beim obigen Mauern-Beispiel wird auch in diesem Fall wieder ein  *Trigger*-Objekt genutzt. Dieses Mal muss die  *AudioSource* aber dem  *Trigger* selbst zugefügt sein, da hier über die Variable  *audio* auf diese zugegriffen wird.

**Listing 10.2** Per Trigger-Collider eine AudioSource ein- und ausschalten

```
using UnityEngine;
using System.Collections;
public class SoundTrigger : MonoBehaviour {
    void OnTriggerEnter(Collider other) {
        if (!audio.isPlaying)
            audio.Play ();
    }

    void OnTriggerExit(Collider other) {
        audio.Stop();
    }
}
```

**Mehrere AudioSources pro GameObject**

Der Variablenzugriff auf eine *AudioSource* kann seine Tücken haben. *AudioSources* können einem *GameObject* mehrfach zugewiesen werden, was bei anderen Variablen wie *transform* oder *rigidbody* nicht der Fall ist. In diesem Fall haben Sie keinen Einfluss darüber, auf welche *AudioSource* Unity am Ende tatsächlich zugreift. Wenn Sie also einem *GameObject* mehrere *AudioSources* zuweisen, sollten Sie auf jeden Fall öffentliche Variablen nutzen, auf die Sie dann gezielt die verschiedenen *AudioSources* ziehen.

**10.2.3 Temporäre AudioSource**

Die  *AudioSource*-Klasse bietet eine statische Methode namens *PlayClipAtPoint* an. Mit dieser können Sie zur Laufzeit schnell und unkompliziert eine temporäre  *AudioSource* erstellen. Sie übergeben dieser einfach einen  *AudioClip*, eine Ortsangabe, wo die  *AudioSource* platziert werden soll, sowie optional die Lautstärke. Sobald Sie die Methode aufrufen, wird eine  *AudioSource* erstellt, die den Sound einmalig abspielt. Danach wird die  *AudioSource* gleich wieder zerstört. Diese Funktion eignet sich besonders für Sounds, die nur ab und zu erklingen und natürlich nicht geloopt werden, z.B. der Klick-Sound eines Buttons in der Game-GUI. Aber auch das Springen eines Spielers könnte durch einen auf diese Art abgespielten Sound unterstrichen werden:

**Listing 10.3** Jump-Sound per *PlayClipAtPoint* abspielen

```
using UnityEngine;
using System.Collections;
public class SoundExample : MonoBehaviour {
    public AudioClip clip;
    void Update()
    {
        if (Input.GetButtonDown("Jump"))
            AudioSource.PlayClipAtPoint(clip, transform.position,1);
    }
}
```

## ■ 10.3 AudioClip

*AudioClips* sind die eigentlichen Tondateien, die von einer  *AudioSource* abgespielt, vom  *AudioListener* empfangen und über die Computerboxen am Ende zu hören sind. Als  *AudioClip* wird dabei genauer gesagt der Asset-Typ bezeichnet, der die Audiodaten beinhaltet. Jede Audiodatei wird in Unity automatisch in einen  *AudioClip* umgewandelt. Dabei werden neben den typischen Audio-Formaten .wav, .mp3, .aif und .ogg auch sogenannte Tracker-module wie das .mod-Format unterstützt. Letzteres sind kleine Sounddateien ähnlich dem MIDI-Format. Allerdings besitzen diese nicht nur die Informationen der abzuspielenden Töne, sondern auch gleichzeitig noch die genutzte Soundbibliothek, die aus kurzen Samples besteht.

### 10.3.1 3D-Sound und Hintergrundmusik

Normalerweise werden alle Audiodateien als 3D-Sounds behandelt. Das bedeutet, dass sich der Ton später je nach Position des Hörers und der Abspielquelle leiser, lauter, weiter rechts oder weiter links im Stereobild befinden kann. Dies betrifft übrigens sowohl Stereo- als auch Mono-Sounds.

Dieses Verhalten kann häufig erwünscht sein, ab und zu aber nicht, wie zum Beispiel bei 2D-Spielen, Hintergrundmusik oder auch GUI-Sounds. Hier sollte sich der Klang nicht durch Bewegungen des Spielers bzw. der örtlichen Lage der  *AudioSource* verändern. In diesem Fall können Sie in den  *Import Settings* des  *AudioClips* die Eigenschaft **3D Sound** deaktivieren. Eine andere Möglichkeit ginge über den *Pan Level*-Parameter der  *AudioSource*, den Sie dann auf 0 stellen müssten.

### 10.3.2 Länge ermitteln

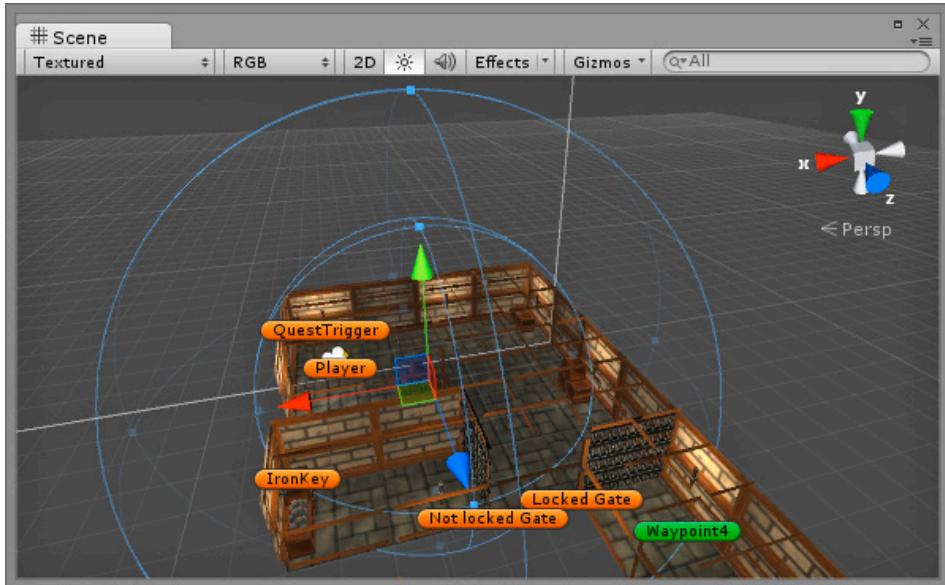
Die Klasse  *AudioClip* liefert noch einige zusätzliche Methoden und Variablen, die das Arbeiten mit Sounddateien erleichtern. Hierzu gehört unter anderem auch die Eigenschaften *length*, die die Länge der Sounddatei zurückgibt. Ein Zugriff auf diese Eigenschaft sieht wie folgt aus:

**Listing 10.4** Länge einer Audiodatei ermitteln

```
public AudioClip hitSound;  
private float length = 0;  
void Start () {  
    length = hitSound.length;  
}
```

## ■ 10.4 Reverb Zone

Eine *Reverb Zone* macht genau das, was der Name vermuten lässt: Sie beschreibt einen Raum und fügt den zu hörenden *AudioClips* einen Halleffekt zu. Eine *Reverb Zone*-Komponente können Sie einem beliebigen *GameObject* über das Menü **Component/Audio/Audio Reverb Zone** hinzufügen.



**Bild 10.4** Reverb Zone in einer Halle

Eine *Reverb Zone* besitzt mehrere Parameter, wovon zunächst drei am wichtigsten sind:

- **Min Distance** legt einen kugelförmigen Raum fest. Wenn sich der *AudioListener* in diesem Bereich befindet, wird der Halleffekt in voller Lautstärke wiedergegeben, vergleichbar mit dem Inneren einer Höhle.
- **Max Distance** ergänzt den Raum um eine weitere äußere Hülle. Zwischen dem inneren Raum und dem äußeren Rand fällt der Hallanteil langsam ab, sodass der Hall am äußeren Rand nur ganz marginal zu hören ist.
- **Reverb Preset** legt die Art des Halls anhand von vorkonfigurierten Halleffekten fest. Wenn Ihnen diese *Presets* nicht reichen, können Sie über die Einstellung *User* die tatsächlichen Effekt-Parameter freischalten, um so einen ganz individuellen Halleffekt kreieren zu können.

Es gibt einen großen Unterschied zwischen einer *Reverb Zone* und der Realität. Befindet sich ein Spieler in einer *Reverb Zone*, z.B. einer Höhle, dann werden alle Geräusche mit Hall versehen, unabhängig davon, wo sich die Kangerzeuger befinden. Es spielt also keine Rolle, ob sich die *AudioSources* ebenfalls in der Höhle befinden oder außerhalb liegen. Um dieses Problem zu lösen, können Sie z.B. beim Verlassen einer Höhle alle Sounds muten, die in der Höhle geschehen.

Weiter können Sie AudioSources, die nicht mit Hall ergänzt werden sollen, mit der eigenen Eigenschaft *Bypass Reverb Zones* davon ausklammern. Dies kann bei der Höhlen-Szene den Außenklang betreffen, der außerhalb der Höhle zu hören ist, aber auch die Hintergrundmusik, die vielleicht während der Höhlenerkundung zu hören sein soll.

## ■ 10.5 Filter

Unity Pro unterstützt noch zusätzliche *Audio-Filter*, mit denen Sie AudioSources separate Effekte zuweisen können. Unity bietet hier verschiedenste Effekte an, die von den typischen Hall- und Echoeffekten über Hoch- und Tiefpassfilter bis hin zu einem *Chorus* und einem *Distortion*-Effekt reichen. Diese erreichen Sie ebenfalls über das Menü **Component/Audio/**.

Lizenziert für gregwyss@hotmail.com.

© 2014 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

# 11

## Partikeleffekte mit Shuriken

Ein besonders leistungsfähiges Effekt-Werkzeug ist das in Unity integrierte *Shuriken Particle System*. Ein solches Partikelsystem erzeugt beliebig viele kleine Objekte, über die Sie (fast) völlige Kontrolle haben. Stellen Sie sich das wie eine Schneemaschine vor, bei der Sie bestimmten können, wie die Schneeflocken in Form, Farbe und Größe aussehen, in welche Richtung sie fliegen und wie sie sich während ihrer Existenz, bis sie sich schließlich selbst zerstören, verändern.

Auf diese Weise erzeugte Effekte werden auch Partikeleffekte genannt und können aus mehreren Partikelsystemen bestehen. Häufig werden sie genutzt, um zum Beispiel Feuer, Rauch, Regen oder auch Explosionen darzustellen.

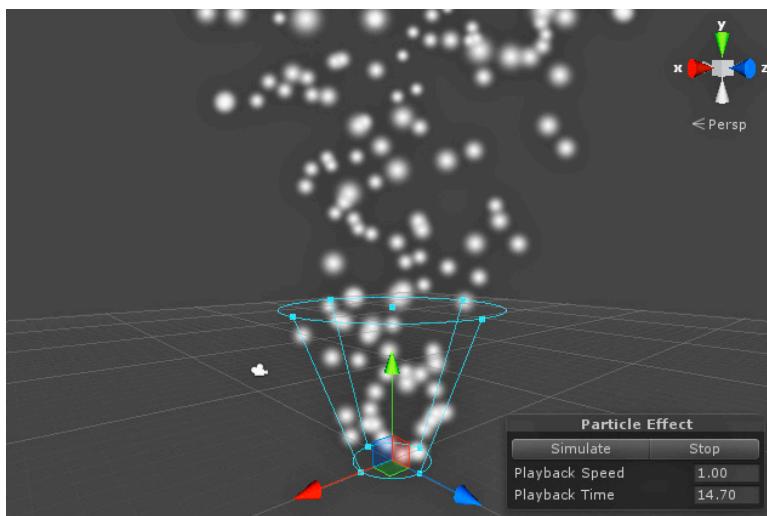


Bild 11.1 Shuriken-Partikelsystem

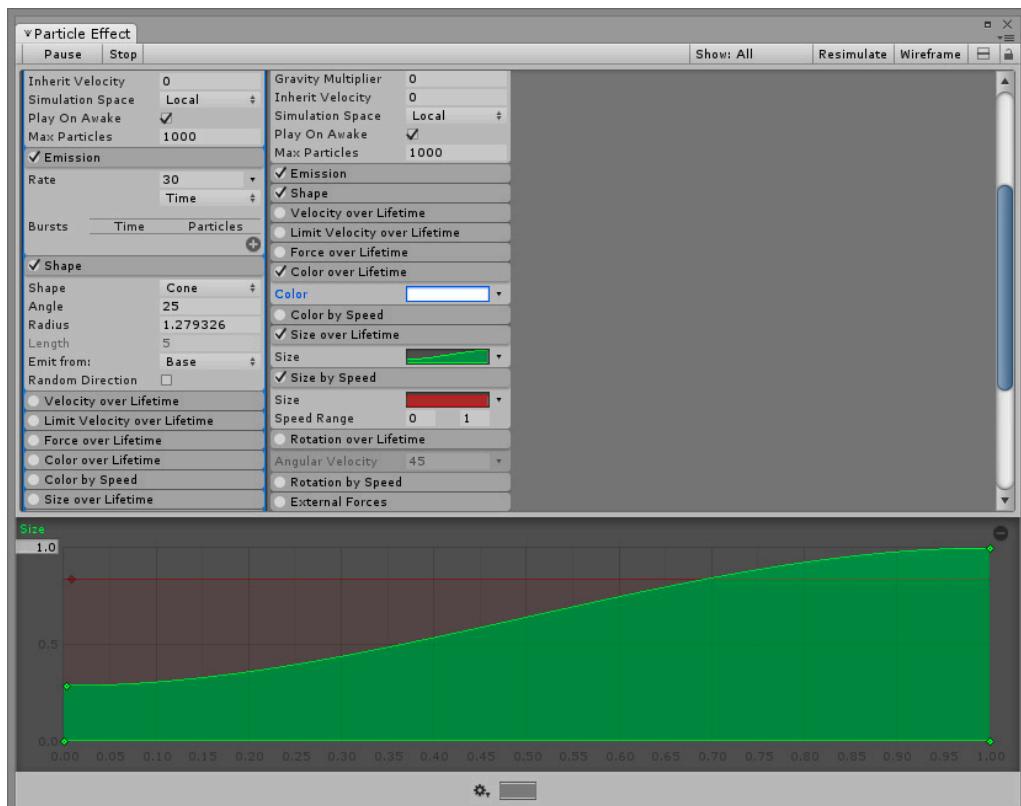
Um ein neues Partikelsystem zu erstellen, gibt es zwei Möglichkeiten:

- **ParticleSystem-GameObject** fügt einer Szene ein neues *GameObject* mit einer *ParticleSystem*-Komponente zu.
- **ParticleSystem-Komponente** fügt einem beliebigen *GameObject* eine *ParticleSystem*-Komponente zu.

Für gewöhnlich sind Partikel nur zweidimensionale Flächen, die stets so ausgerichtet sind, dass sie in Richtung der Kamera zeigen. Auch wenn dies zunächst einmal etwas merkwürdig klingt, so funktioniert es in der Praxis sehr gut. Denken Sie nur an eine Kerze. Wenn Sie diese drehen und nur auf die Flamme achten, werden Sie kaum einen Effekt des Drehens bemerken. Sollten Sie aber dennoch mal dreidimensionale Partikel benötigen, können Sie auch diese in *Shuriken* nutzen.

## ■ 11.1 Editor-Fenster

Ein Partikelsystem besitzt eine Fülle an Parametern. So können Sie einem Hauptsystem auch weitere Subsysteme hinzufügen, um auf diese Weise noch komplexere Partikeleffekte zu erzielen. Damit Sie hierbei nicht die Übersicht verlieren, bietet Unity ein Extra-Editor-Fenster für Partikelsysteme an, in dem Sie alle Parameter und Subsysteme eines Partikel-systems überblicken können. Dieses Fenster erreichen Sie über den Button **Open Editor** im *Inspector*.

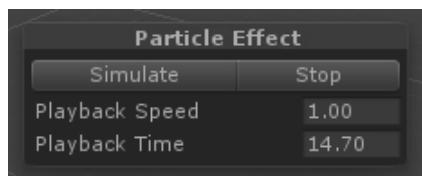


**Bild 11.2** Particle-System-Editor-Fenster

Im Kopfbereich dieses Extrafensters sehen Sie eine Toolbar mit Schnellzugriffen auf einige Parameter des Preview-Panels, das ich gleich noch vorstellen werde. Außerdem finden Sie dort die interessante Funktion *Show: All/Selected*, die bei komplexeren Systemen sehr hilfreich ist. Mit dieser haben Sie die Möglichkeit, nur die Partikeleffekte zu betrachten, die Sie gerade in der *Hierarchy* selektieren.

## ■ 11.2 Particle Effect Control

Eine weitere Besonderheit ist das *Particle Effect Control*, das in der *Scene View* erscheint, sobald Sie ein Partikelsystem in der Hierarchy selektieren.



**Bild 11.3**  
Particle Effect Control

Da der Effekt zu dem Zeitpunkt normalerweise in der *Scene View* als Vorschau angezeigt wird, haben Sie hier die Möglichkeit, diese Preview anzupassen oder ganz zu unterbrechen. Zur Auswahl stehen *Simulate*, *Pause* und *Stop*, die übrigens auch im obigen Editor-Fenster angezeigt werden.

Zusätzlich können Sie über *Playback Speed* die Wiedergabegeschwindigkeit des Effektes verändern und mit der *Playback Time* auf einen bestimmten Zustand des Effektes auf der Zeitachse springen. Letzteres hilft Ihnen z. B. dabei, der Frage nachzugehen, wie ein Effekt zu einem bestimmten Zeitpunkt aussieht, ohne auf diesen Zeitpunkt unter Umständen länger zu warten.

## ■ 11.3 Numerische Parametervarianten

Das *Shuriken Particle System* bietet vier Möglichkeiten, numerische Werte in den unterschiedlichen Moduleigenschaften zu definieren. Je nach Parameter können alle oder Teile davon zur Verfügung stehen.

- **Constant** definiert einen festen Wert, der die gesamte Zeit gilt. Dies kann ein einzelner Dezimalwert sein oder auch beispielsweise ein Vektor, der aus einem X-, Y- und Z-Wert besteht.
- **Curve** legt den Parameter anhand einer Kurvenform fest. Über die rechte Maustaste können Sie mit dem Button *Add Key* der Kurve weitere Wendepunkte hinzufügen, um diese so zu gestalten, wie Sie möchten. Mit dieser Parametervariante haben Sie die Möglichkeit, den eigentlichen Wert in Abhängigkeit einer anderen Größe zu beschreiben, z. B. zur

Lebenszeit des Partikels. Möchten Sie also einen Wert im Laufe der Zeit beispielsweise ansteigen oder abfallen lassen, dann ist diese Variante die richtige.

- **Random Between Two Constants** erlaubt Ihnen, ein Minimum und ein Maximum eines Parameters festzulegen, zwischen denen dann für jedes Partikel eins per Zufall ausgewählt und zugewiesen wird.
- **Random Between Two Curves** lässt Sie zwei Kurven definieren, die Minimum- und Maximum-Werte in Bezug zu einer weiteren Größe, wie der Zeit, beschreiben. Unity generiert für jedes Partikel eine neue Kurve, die sich zwischen diesen beiden bewegt. Auch hier können Sie über die rechte Maustaste und den Button *Add Key* den Kurven weitere Wendepunkte hinzufügen.

## ■ 11.4 Farbparameter-Varianten

Um Farben zu definieren, bietet Shuriken ebenfalls vier Möglichkeiten an, Farbwerte in den unterschiedlichen Moduleigenschaften zu definieren. Je nach Parameter können auch hier alle oder Teile davon zur Verfügung stehen.

- **Color** legt einen Farbwert fest.
- **Gradient** legt einen Farbverlauf fest, dessen Werte dann abhängig von einem zweiten Parameter, z.B. von der Zeit, dargestellt werden. Durch einen kurzen Klick mit der Maustaste fügen Sie dem Gradient-Editor einen neuen Farbpunkt zu. Durch ein längeres Drücken auf einen Farbpunkt selektieren Sie diesen, um ihn entweder auf der Achse zu verschieben oder dessen Wert über die unteren Steuerelemente anzupassen. Die oberen Einstellungen legen dabei die Farbdichte fest, die unteren die eigentliche Farbe.
- **Random Between Two Colors** erlaubt Ihnen, zwei Farben festzulegen. Für jedes Partikel wird dann ein Farbwert gewählt, der sich zwischen diesen beiden befindet.
- **Random Between Two Gradients** ermöglicht, zwei Farbverläufe zu definieren, die sich basierend auf einem weiteren Parameter, z.B. der Zeit, verändern. Unity definiert dann pro Partikel einen neuen Farbverlauf, dessen Farben sich an diesen beiden orientieren. Wenn Sie beispielsweise einen Farbverlauf von Rot nach Weiß und einen weiteren von Gelb nach Weiß definieren, wird sich die Startfarbe irgendwo zwischen Rot und Gelb befinden und sich im Laufe der Zeit zu Weiß verändern.

## ■ 11.5 Default-Modul

Das *Shuriken Particle System* ist ein funktional sehr mächtiges Werkzeug. Deshalb besitzt es auch eine große Anzahl an Parametern, die die Partikel auf unterschiedlichste Weise modifizieren können. Im oberen Teil des *Inspectors* finden Sie im *Default Modul* zunächst einmal die Grundeinstellungen des Partikelsystems:

- **Duration** ist die Dauer, in der das System Partikel erzeugt. Wenn *Looping* aktiviert ist, dann dauert ein Durchgang so lange wie dieser Wert.
- **Looping** legt fest, ob Partikel in einer Endlosschleife erzeugt werden sollen.
- **Prewarm** erweitert das *Looping* so, dass bereits am Anfang so viele Partikel existieren, als hätte das System bereits einen Durchlauf hinter sich. Eine Einsatzmöglichkeit wäre hier z. B. ein Regen-Effekt, der schon beim Szenenstart so existieren soll, als würde es schon die ganze Zeit regnen.
- **Start Delay** verzögert den Start der Partikelerzeugung. Dieser Parameter steht nicht im Zusammenspiel mit *Prewarm* zur Verfügung.
- **Start Lifetime** legt die Lebenszeit fest. Ein Partikel besitzt einen Counter, der heruntergezählt wird. Und wenn dieser Zähler 0 erreicht, stirbt dieses. Dieser Parameter legt den Anfangswert fest.
- **Start Speed** legt die Anfangsgeschwindigkeit fest.
- **Start Size** legt die Anfangsgröße fest.
- **Start Rotation** legt die Anfangsdrehung um den Mittelpunkt in Grad fest.
- **Start Color** legt die Startfarbe der Partikel fest.
- **Gravity Multiplier** legt fest, wie stark der Einfluss der Gravitation auf die Partikel ist.
- **Inherit Velocity** legt fest, wie viel Einfluss die Eigenbewegung des Systems auf die Partikel hat. Wenn sich das eigentliche Objekt z. B. vorwärts bewegt, dann fliegen auch die Partikel verstärkt in diese Richtung.
- **Simulation Space** legt fest, ob sich die Partikel relativ zum Partikelsystem bewegen sollen oder frei im Raum.
- **Play On Awake** legt fest, ob das System gleich zu Beginn an Partikel erzeugen soll.
- **Max Particles** legt die maximale Anzahl von gleichzeitig existierenden Partikeln fest. Wenn diese erreicht ist, erzeugt das System so lange keine weiteren, bis der Lebencounter eines Partikels 0 erreicht hat und sich selbst zerstört und damit die Grenze wieder unterschritten ist.

## ■ 11.6 Effekt-Module

Die eigentlichen Partikeleinstellungen und -animationen für Farben, Formen und Bewegungen werden unter dem *Default-Modul* in den einzelnen Effekt-Modulen vorgenommen. Jedes Modul ist hierbei für eine ganz bestimmte Eigenschaft und dessen Verhalten verantwortlich. Über Haken können Sie steuern, welche Module Sie nutzen möchten und welche nicht.

### 11.6.1 Emission

Dieses Modul beschreibt den eigentlichen Ausstoß der Partikel, also wann wie viele erzeugt werden.

- **Rate** legt fest, wie viele Partikel pro Sekunde (*Time*) oder je Welteinheit (*Distance*) erzeugt werden. Eine Welteinheit entspricht hierbei einem Meter.
- **Bursts** erzeugen plötzlich auftretende Partikelmengen. Über die Plus- und Minuszeichen können Sie zusätzliche Ausstöße hinzufügen bzw. entfernen. Über *Particles* legen Sie die Anzahl der zu erzeugenden Partikel fest und über *Time* steuern Sie den Zeitpunkt der Erzeugung innerhalb eines Loops (*Duration*).

### 11.6.2 Shape

Dieses Modul legt über den gleichnamigen Parameter einen dreidimensionalen Raum fest, in dem die Partikel erzeugt werden.

Über die weiteren formabhängigen Parameter können Sie weitere Einstellungen vornehmen, die ich im Folgenden vorstellen möchte.

#### 11.6.2.1 Sphere

Die Form *Sphere* beschreibt eine Kugel und stößt die erzeugten Partikel in jede Richtung aus.

- **Radius** legt den Radius der Kugel fest.
- **Emit from Shell** erzeugt Partikel nur am äußeren Rand der Kugel. Ansonsten werden sie zufällig irgendwo im Kugelinneren erzeugt.
- **Random Direction** lässt die erzeugten Partikel in eine zufällige Richtung fliegen. Normalerweise fliegen diese immer vom Mittelpunkt weg.

#### 11.6.2.2 HemiSphere

*HemiSphere* beschreibt eine Halbkugel und stößt die Partikel in alle Richtungen der einen Halbkugel.

- **Radius** legt den Radius der Halbkugel fest.
- **Emit from Shell** erzeugt Partikel nur am äußeren Rand der Halbkugel. Ansonsten werden sie zufällig irgendwo im Kugelinneren erzeugt.
- **Random Direction** lässt die erzeugten Partikel in eine zufällige Richtung fliegen. Normalerweise fliegen diese immer vom Mittelpunkt weg.

#### 11.6.2.3 Cone

Als *Cone* wird ein Kegelstumpf bezeichnet, der die Partikel wie ein Megafon den Schall aus der größeren Kreisfläche ausstößt. Dabei ist dieser in die positive lokale Z-Richtung gerichtet.

- **Angle** definiert den Winkel des Kegelstumpfes.
- **Radius** legt den Radius des kleinen Kreises fest.

- **Length** beschreibt die Länge des Kegelstumpfes.
- **Emit from** definiert, wo die Partikel im Kegelstumpf erzeugt werden und wo sie hinfliegen. Bei *Base* werden sie zufällige irgendwo auf der Fläche des kleinen Kreises erzeugt. Bei *Base Shell* entstehen sie am äußeren Rand des Kreises. Bei *Volume* werden sie zufällig im Inneren des Kegelstumpfes und bei *Volume Shell* irgendwo am Rand des Stumpfes erzeugt.

#### 11.6.2.4 Box

Diese Form legt eine Box fest, in deren Volumen die Partikel zufällig erzeugt werden. Normalerweise werden hier die Partikel aus der Fläche, die in der lokalen positiven Z-Richtung liegt, ausgestoßen.

- **Box X, Y, Z** legen die Maße der Box fest.
- **Random Direction** lässt die erzeugten Partikel in zufällige Flächenrichtungen der Box fliegen.

#### 11.6.2.5 Mesh

Mit dieser Einstellung können Sie ein beliebiges *Mesh* als Erzeugungskörper nutzen.

- **Vertex, Edge, Triangle** legt fest, auf welcher *Mesh*-Eigenschaft basierend die Partikel erzeugt werden. Diese Eigenschaft legt dabei nicht nur die zufälligen Positionsmöglichkeiten fest, sondern auch die Richtung, in die die Partikel dann standardmäßig fliegen sollen. Bei *Vertex* wird das Partikel in die Richtung der *Vertex*-Normalen fliegen. *Edge* wird durch zwei *Vertices* definiert, weshalb die Richtung des Partikels aus den beiden *Vertex*-Normalen berechnet wird. Bei *Triangle* wird die Ausrichtung der Flächennormalen genommen.
- **Mesh** dient dem Zuweisen des Erzeuger-*Meshes*. Hierbei wählen Sie ein *Mesh* aus Ihrem Projekt aus.
- **Random Direction** lässt die erzeugten Partikel in eine zufällige Richtung fliegen.

### 11.6.3 Velocity over Lifetime

Dieses Modul legt die Geschwindigkeit fest und kann diese auch über die *Lifetime* hinweg verändern.

- **X, Y, Z** legt die Richtung des *Velocity*-Vektors fest. Sie können zwischen den oben beschriebenen numerischen Parametervarianten wählen.
- **Space** legt fest, ob sich die Richtung an dem lokalen oder dem globalen Koordinatensystem orientieren soll.

### 11.6.4 Limit Velocity over Lifetime

Dieses Modul legt eine Dämpfung fest, die so lange wirkt, bis eine definierte Endgeschwindigkeit erreicht ist.

- **Speed** legt die Endgeschwindigkeit fest, auf die heruntergebremst werden soll.

- **Dampen** setzt den Dämpfungseinfluss zwischen 0 (keinen Einfluss) bis 1 (sofortiges Abdämpfen auf die Zielgeschwindigkeit) fest.
- **Separate Axis** gibt Ihnen die Möglichkeit, *Speed* für jede Achse einzeln einzustellen. Zudem können Sie festlegen, ob für die Achsenzuordnung das lokale oder globale Koordinatensystem genutzt werden soll.

### 11.6.5 Force over Lifetime

Dieses Modul fügt den Partikeln eine gerichtete Kraft zu, die deren Geschwindigkeit verändert. Die hierdurch erzeugte Geschwindigkeit kann durch andere Kräfte wie z. B. Wind noch zusätzlich beeinflusst werden.

- **X, Y, Z** legt den Kraftvektor fest. Sie können zwischen den oben beschriebenen numerischen Parametervarianten wählen.
- **Space** legt fest, ob sich die Richtung an dem lokalen oder dem globalen Koordinatensystem orientieren soll.
- **Randomize** verändert die zugefügte Kraft in jedem *Frame*. Dieser Parameter steht nur zur Verfügung, wenn oben bereits entweder *Random Between Two Constants* oder *Random Between Two Curves* ausgewählt wurde.

### 11.6.6 Color over Lifetime

Dieses Modul bestimmt die Farbe in Bezug zur Lebenszeit.

- **Color** legt die Farbe fest. Sie können zwischen *Gradient* und *Random Between Two Gradients* wählen.

### 11.6.7 Color by Speed

Dieses Modul bestimmt die Farbe in Relation zur Geschwindigkeit.

- **Color** legt die Farbe fest. Sie können zwischen *Gradient* und *Random Between Two Gradients* wählen.
- **Speed Range** legt die Minimum- und Maximum-Geschwindigkeit fest, in der die Farbe definiert werden kann.

### 11.6.8 Size over Lifetime

Dieses Modul ermöglicht Ihnen, die Partikelgröße anhand der Lebenszeit zu verändern.

- **Size** lässt einen veränderlichen Wert definieren, wobei die numerischen Parametervarianten *Curve*, *Random Between Two Constants* und *Random Between Two Curves* zur Auswahl stehen.

## 11.6.9 Size by Speed

Dieses Modul ermöglicht Ihnen, die Partikelgröße anhand der Geschwindigkeit zu verändern.

- **Size** lässt einen veränderlichen Wert definieren, wobei die numerischen Parametervarianten *Curve*, *Random Between Two Constants* und *Random Between Two Curves* zur Auswahl stehen.
- **Speed Range** legt die Minimum- und Maximum-Geschwindigkeit fest, in der die Größe definiert werden kann.

## 11.6.10 Rotation over Lifetime

Dieses Modul legt die Drehung eines Partikelobjektes in Bezug zur Lebenszeit fest.

- *Angular Velocity* legt die Rotationsgeschwindigkeit fest. Dabei werden alle numerischen Parametervarianten unterstützt.

## 11.6.11 Rotation by Speed

Dieses Modul legt die Drehung eines Partikelobjektes in Bezug zur Geschwindigkeit fest.

- **Angular Velocity** legt die Rotationsgeschwindigkeit fest. Dabei werden alle numerischen Parametervarianten unterstützt.
- **Speed Range** legt die Minimum- und Maximum-Geschwindigkeit fest, in der die Rotationsgeschwindigkeit definiert werden kann.

## 11.6.12 External Forces

Dieses Modul legt den Einfluss von *Wind Zones* auf die Partikel fest.

- **Multiplier** ist ein Faktor, mit dem externe Kräfte, wie die einer *Wind Zone*, multipliziert und den Partikeln zugefügt werden. Durch einen Wert kleiner 1 werden die Partikel träge und wirken so, als hätten diese eine Masse.

## 11.6.13 Collision

Dieses Modul legt fest, auf welche Weise die Partikel mit Objekten kollidieren können. Hierbei können Sie zunächst festlegen, ob Kollisionen nur mit einzelnen Ebenen oder mit echten Objekten stattfinden sollen.

- **Planes/World** legt fest, ob die Partikel lediglich mit virtuellen Ebenen kollidieren können (*Plane*) oder aber mit allen *Collidern* einer Szene (*World*). Beide Varianten bieten weitere, zusätzliche Parameter an, auf die wir im Folgenden noch zu sprechen kommen.

- **Dampen** dämpft die Geschwindigkeit eines Partikels nach dem Aufprall gleichmäßig in alle Richtungen. Der Wert kann zwischen 0 bis 1 liegen, wobei 0 keine Dämpfung bedeutet. Soll ein Partikel also nach dem Aufprall einfach liegen bleiben, vergleichbar mit einer Schneeflocke, sollten Sie diesen Wert auf 1 setzen.
- **Bounce** bestimmt, wie stark das Partikel von der Oberfläche abfedert. Der Wert kann zwischen 0 und 1 liegen. Im Gegensatz zu **Dampen** wird hier nur die Federrichtung beeinflusst. Setzen Sie hier den Wert auf 0, was keine Federwirkung bedeutet, würde der Partikel zwar nicht wieder abgestoßen werden, trotzdem könnte sich das Partikel nach der Kollision noch an der Kollisionsfläche entlangbewegen. Vergleichen Sie dies mit einer Stahlkugel, die auf den Boden fällt. Auch diese würde nicht vom Boden abfedern. Trotzdem könnte sie nach dem Aufprall noch etwas zur Seite rollen.
- **Lifetime Loss** bestimmt, um wie viel sich die Lebenszeit durch eine Kollision verkürzt. Der Wert kann zwischen 0 bis 1 liegen. Wenn Sie das Partikel gleich beim ersten Aufprall zerstören möchten, dann setzen Sie diesen Wert auf 1.
- **Min Kill Speed** legt eine Geschwindigkeit fest, bei deren Unterschreitung das Partikel zerstört wird. Im Gegensatz zum obigen Parameter kann hier ein Partikel öfter kollidieren und langsam heruntergebremst werden, bis es schließlich diesen Wert unterschreitet und zerstört wird.
- **Send Collision Messages** legt fest, ob bei einer Partikelkollision die `OnParticleCollision`-Methode ausgelöst werden soll (siehe Abschnitt „`OnParticleCollision`“). Da hierfür die Kollision mit einem *Collider* Voraussetzung ist, funktioniert dies aber nur bei *World*. Kollisionen mit *Planes* rufen aktuell nicht `OnParticleCollision` auf.

### 11.6.13.1 Planes

*Planes* ermöglicht Ihnen, bis zu sechs virtuelle Ebenen zu definieren, mit denen Ihre Partikel kollidieren können. Diese Variante ist eine sehr performante Möglichkeit, Partikelkollisionen zu realisieren.

- **Planes** nimmt bis zu sechs Transforms auf, die die Positionen der virtuellen Ebenen (*Planes*) festlegen. Über das untere Plus- und Minuszeichen können Sie diese Liste erweitern. Sie können hier beliebige Objekte Ihrer Szene hinzufügen. Sie können aber auch über das rechte Pluszeichen neue virtuelle *Plane*-Objekte erzeugen. Diese sind dann unsichtbar und dienen nur der Definition, wo eine Kollision stattfinden soll. Aber egal ob Sie eine virtuelle Ebene oder ein normales *GameObject* aus Ihrer Szene nutzen, für die Kollisionserkennung ist lediglich die Position des *Transform-Nodes* sowie die Drehung des Transforms relevant. Die tatsächliche Größe des *GameObjects* spielt hierbei keine Rolle.
- **Visualization** bestimmt die Darstellung der virtuellen *Planes*, wenn Sie das *Particle System* in der *Hierarchy* selektieren. Sie können zwischen Solid, also einer normalen *Plane*-Darstellung, oder einer Drahtgitterdarstellung (*Grid*) wählen. Der Vorteil der *Grid*-Darstellung liegt darin, dass hier das Gitter von allen Seiten gesehen wird, während die *Plane* nur von der *Normalen*-Ausrichtung sichtbar ist.
- **Scale Plane** skaliert die Darstellung der virtuellen Platten. Dies hat aber keinen Einfluss auf die tatsächliche Kollision.
- **Particle Radius** legt den Radius eines kreisförmigen Partikel-*Colliders* fest.

### 11.6.13.2 World

Bei dieser Kollisionsvariante können Partikel mit allen *Collidern* Ihrer Szene kollidieren. Allerdings ist dieses Verfahren sehr rechenintensiv.

- **Collides With** legt die *Layer* fest, deren Objekte mit diesen Partikeln kollidieren können.
- **Collision Quality** legt die Qualität der Kollisionserkennung fest. *High* ist sehr präzise, aber auch recht rechenintensiv, da jeder Partikel in jedem *Frame* ein *Raycast* erstellt. *Medium* betrachtet hierbei in jedem *Frame* nur eine Untermenge aller Partikeln und kann deshalb auch einige Kollisionen übersehen. *Low* nutzt die gleiche Technik, allerdings nur jeden vierten *Frame*. Ein wichtiger Nachteil bei *Medium* und *Low* ist allerdings der, dass nur statische Objekte bzw. nur *GameObjects* ohne ein *Rigidbody* berücksichtigt werden.
- **Voxel Size** legt die Größe und damit die Dichte der *Voxel* fest, die bei den Qualitätsstufen *Medium* und *Low* zum Zwischenspeichern der Partikelkollisionen dienen. Häufig eignen sich Werte zwischen 0,5 und 1,0.

### 11.6.14 Sub Emitter

Dieses Modul ermöglicht es, Sub-Partikelsysteme zu erstellen. Diese werden zusätzlich in der *Hierarchy* als Kind-Objekte des Hauptpartikelsystems dargestellt. Die Subsysteme erzeugen Partikel basierend auf den Positionen des Hauptpartikelsystems. Das bedeutet:

- **Birth**-Subsysteme erzeugen Partikel auf Basis der Partikelpositionen des Hauptsystems.
- **Death**-Subsysteme erzeugen Partikel, wenn Partikel des Hauptsystems das Ende der Lifetime erreicht haben.
- **Collision**-Subsysteme erzeugen Partikel, wenn Partikel des Hauptsystems mit Objekten kollidieren (siehe *Collision-Modul*).

### 11.6.15 Texture-Sheet-Animation

Dieses Modul erlaubt Ihnen, für die Partikeldarstellung anstelle eines starren Bildes eine *Sprite-Animation* zu nutzen. Unter einer *Sprite-Animation* versteht man das schnelle Abspielen verschiedener Bilder, die auf diese Weise ein bewegtes Motiv darstellen, vergleichbar mit einem Daumenkino (siehe Kapitel 17.2.3.2).

Hierfür müssen Sie zunächst im *Renderer-Modul* ein **Material** zuweisen, das einen *Texture-Atlas* als Textur besitzt. Ein *Texture-Atlas* ist wiederum ein einzelnes Bild, auf dem sich alle Einzelbilder einer *Sprite-Animation* horizontal und vertikal angeordnet befinden.

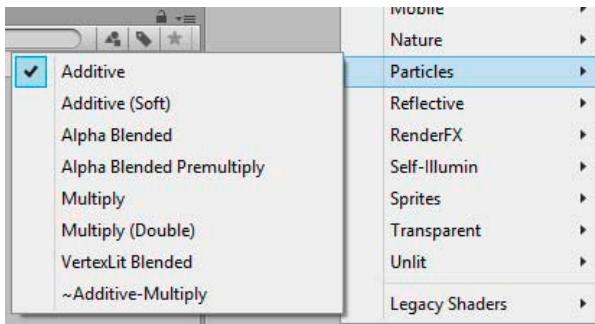
- **Tiles** definiert die Anzahl der Einzelbilder auf der Textur in X- und Y-Richtung.
- **Animation** legt fest, was animiert werden soll. Die Standardeinstellung ist hierbei *Whole Sheet*. Hier ergeben alle Bilder des *Texture-Atlas* eine einzige Animation, die auch so abgespielt wird. Alternativ können Sie aber auch die Einstellung *Single Row* nutzen. Hierbei wird lediglich eine Zeile als komplette Animation betrachtet und auch nur diese abgespielt.
- **Row** definiert die Zeile des *Texture-Atlas*, die bei *Single Row* abgespielt werden soll.

- **Random Row** wählt bei *Single Row* für jedes Partikel eine zufällige Zeile aus, die für die *Animation* genutzt wird.
- **Frame over Time** legt die Abspielgeschwindigkeit über die gesamte Lebenszeit hinweg fest. So können Sie z. B. die Geschwindigkeit im Laufe der Zeit erhöhen oder auch abschwächen lassen. Ihnen stehen alle bereits vorgestellten numerischen Parametervarianten zur Verfügung. Wenn Sie eine konstante Geschwindigkeit wünschen, nutzen Sie eine ansteigende Gerade (ist standardmäßig bereits eingestellt).
- **Cycles** legt die allgemeine Geschwindigkeit fest.

### 11.6.16 Renderer

Dieses Modul legt fest, wie die Partikel auf dem Bildschirm dargestellt werden sollen.

- **Render Mode** legt die Art des *Renderns* fest. Dies hat großen Einfluss darauf, wie die Partikel verformt werden. Diese verschiedenen Modi werden im Folgenden noch einmal etwas näher erläutert.
- **Normal Direction** legt fest, wie die *Normalen* der Partikel ausgerichtet sind. Bei 1 ist die *Normale* in Richtung der Kamera ausgerichtet, wodurch ein Partikel je nach *Material* bzw. Textur recht flach wirkt. Senken Sie diesen Wert, so drehen Sie die *Normale* in Richtung einer Ecke. Hierdurch entsteht ein Helligkeitsverlauf, wodurch der Partikel etwas plastischer wirken kann.
- **Material** stellt das Partikelmaterial.
- **Sort Mode** legt fest, in welcher Reihenfolge die Partikel gezeichnet werden. Es stehen *None*, *By Distance*, *Youngest First* und *Oldest First* zur Auswahl. Eine Reihenfolge sollte nur dann gesetzt werden, wenn bei der Darstellung Artefakte (Darstellungsfehler) auftreten.
- **Sorting Fudge** legt fest, wann grundsätzlich Partikel dieses Partikelsystems gezeichnet werden. Umso höher dieser Wert liegt, desto später werden diese gezeichnet, was dazu führt, dass diese Partikel vor anderen Objekten erscheinen.
- **Cast Shadows** legt fest, ob Partikel Schatten werfen sollen.
- **Receive Shadows** legt fest, ob Partikel Schatten erhalten können, also ob andere Schatten Einfluss auf deren Helligkeit haben.
- **Max Particle Size** bestimmt, wie viel Platz des Bildschirms ein Partikel maximal einnehmen darf. 0,5 bedeutet die Hälfte, 1 bedeutet den gesamten Bildschirm.



**Bild 11.4**

Shader-Typen für Partikelmaterial

Die charakteristischste Eigenschaft dieses Moduls ist ganz sicher das *Material*. Es entscheidet am Ende, ob es sich bei dem Effekt um Regen, Nebel, Schnee oder doch Feuerfunken handeln soll. Beim Erstellen des Partikelmaterials sollten Sie beachten, dass Unity hierfür spezielle *Shader*-Typen zur Verfügung stellt. Diese finden Sie im Zweig *Particles*.

#### 11.6.16.1 Billboard

In diesem Modus werden Partikel als Quadrate dargestellt, die immer zur Kamera ausgerichtet sind.

#### 11.6.16.2 Stretched Billboard

In diesem Modus werden Partikel anhand von verschiedener Parameter gestreckt, aber nicht immer direkt zur Kamera ausgerichtet:

- **Camera Scale** bestimmt den Einfluss der Kamerageschwindigkeit auf die Länge.
- **Speed Scale** bestimmt den Einfluss der Eigengeschwindigkeit der Partikel auf deren Länge.
- **Length Scale** bestimmt die Länge im Verhältnis zur Breite des Partikels.

#### 11.6.16.3 Horizontal Billboard

Ähnlich wie im Billboard-Modus werden auch hier die Partikel als Quadrate dargestellt. Allerdings werden diese immer so ausgerichtet, dass diese im Welt-Koordinatensystem mit ihrer Fläche nach oben zeigen. Während eine typische Top-down-Kamera diese also perfekt sehen kann, sind die Partikel von der Seite nur schwer oder gar nicht zu sehen.

#### 11.6.16.4 Vertical Billboard

Ähnlich wie im vorherigen Modus werden auch hier die Partikel als Quadrate dargestellt. Allerdings werden diese immer so ausgerichtet, dass sie im Welt-Koordinatensystem mit ihrer Fläche zur Seite zeigen und am besten gesehen werden können, wenn die Kamera wie bei einem Sidescroller in die positive Y-Richtung gerichtet ist.

#### 11.6.16.5 Mesh

In diesem Modus können Sie dem Partikelsystem ein *Mesh* zuweisen, das anstelle einer zweidimensionalen Fläche als Partikel genutzt wird. Allerdings darf dieses *Mesh* nicht beliebig komplex sein, sondern nur ein einziges Sub-*Mesh* besitzen.

- **Mesh** legt das darzustellende *Mesh* fest.

## ■ 11.7 Partikelemission starten, stoppen und unterbrechen

Zum Steuern eines Partikelsystems können Sie verschiedene Methoden und Eigenschaften nutzen, die die `ParticleSystem`-Klasse zur Verfügung stellt. Zum Kontrollieren, wann Partikel erzeugt werden und wann nicht, stellt die Klasse gleich mehrere Möglichkeiten zur Verfügung, die ich im Folgenden kurz vorstellen möchte. Unity bietet übrigens auch hier einen Schnellzugriff über die Variable `particleSystem` an.

**Listing 11.1** Partikelsystem per Tastatur steuern

```
using UnityEngine;
using System.Collections;
public class ParticleController : MonoBehaviour {
    ParticleSystem ps;
    void Start () {
        ps = particleSystem;
        ps.Stop ();
    }
    void Update () {
        if(Input.GetKeyDown(KeyCode.P))
            ps.Play();
        if(Input.GetKeyDown(KeyCode.S))
            ps.Stop ();
        if(Input.GetKeyDown(KeyCode.B))
            ps.Pause();
        if(Input.GetKeyDown(KeyCode.E))
            ps.enableEmission = !ps.enableEmission;
    }
}
```

### 11.7.1 Play

Über die Methode `Play` starten Sie die Emission. Dabei setzt das System den Zeitzähler auf den Startwert 0. Wenn Sie also einen Burst beim Zeitpunkt 0.00 definiert haben, dann wird dieser nach dem Aufruf von `Play` sofort ausgeführt. Die einzige Ausnahme ist, wenn sich das System zuvor im *Pause*-Modus befand (siehe unten). In dem Fall wird der Zähler nicht auf 0 zurückgesetzt.

### 11.7.2 Stop

Mit der Methode `Stop` halten Sie die Emission weiterer Partikel an. Die bereits emittierten Partikel führen ihre Bewegungen aber noch zu Ende aus.

### 11.7.3 Pause

Über die Methode `Pause` halten Sie die Emission neuer Partikel sowie alle bereits erzeugten Partikel an. Das System und dessen Partikel werden quasi eingefroren. Über die Methode `Play` heben Sie diesen Zustand auf und das Partikelsystem wird dort weitergeführt, wo es angehalten wurde.

### 11.7.4 enableEmission

Mithilfe der Eigenschaft `enableEmission` steuern Sie das *Emission-Modul* des Partikelsystems. Normalerweise werden Sie die Eigenschaft auf `TRUE` belassen. Setzen Sie den Wert während des Spiels nun auf `FALSE`, so wird die Emission neuer Partikel ausgesetzt. Allerdings wird der Zeitzähler nicht angehalten oder zurückgesetzt. Wenn Sie den Wert wieder auf `TRUE` setzen, kann es daher sein, dass das System bei einem nicht loopenden System keine Partikel mehr emittiert, weil die Duration bereits abgelaufen ist.

## ■ 11.8 OnParticleCollision

Wenn Sie das *Collision-Modul* aktivieren und den Parameter `Send Collision Messages` aktivieren, wird bei Kollisionen eines Partikels mit einem *Collider* (siehe hierzu die Einstellungen des *Collision-Moduls*) die Methode `OnParticleCollision` ausgeführt. Auf diese Weise können Sie zum Beispiel feststellen, ob Ihr Spieler von einem Feuerfunken getroffen wurde.

#### **Listing 11.2** OnParticleCollision auswerten

```
void OnParticleCollision(GameObject other) {
    Debug.Log("Autsch!");
}
```

Wie bereits im *Collision-Modul* erwähnt, ist aktuell neben *Send Collision Messages* auch der Kollisionstyp *World* Voraussetzung für das Auslösen von `OnParticleCollision`. Das Skript mit dieser Methode kann entweder dem Partikelsystem selbst angehängt werden oder dem Objekt, das mit den Partikeln kollidiert, z.B. dem Spieler. Wenn Sie das Skript dem Spieler anhängen, dann enthält der `GameObject`-Parameter das Partikelsystem-Objekt. Fügen Sie das Skript dem Partikelsystem zu, so enthält der Parameter Informationen über das `GameObject`, mit dessen *Collider* ein Partikel gerade kollidiert ist.

### 11.8.1 GetCollisionEvents

Um Informationen über die Partikel zu erhalten, die `OnParticleCollision` ausgelöst haben, benötigen Sie die zusätzliche Methode `GetCollisionEvents` der `ParticleSystem`-Klasse. Dieser übergeben Sie zum Ersten das `GameObject`, das mit den Partikeln kollidiert, sowie

zum Zweiten ein *CollisionEvent*-Array, über das Sie die Informationen zurück erhalten. Als Rückgabeparameter ist ein Integer vorgesehen, der die Anzahl der Kollisionsergebnisse festlegt, die über das Array zurückgegeben werden.

In dem folgenden Beispiel wird ein *CollisionEvent*-Array in der Größe von 8 erzeugt und der Methode *GetCollisionEvents* übergeben. So können bis zu acht Kollisionen zurückgegeben werden, die in dem Moment mit dem *GameObject* auftreten. Mithilfe des numerischen Rückgabewertes wird danach das Array durchgelaufen und überall dort, wo eine Kollision stattfindet, für eine halbe Sekunde ein *Prefab* erzeugt. Hierfür wird die *Intersection*-Eigenschaft des *CollisionEvents* genutzt.

**Listing 11.3** Erzeugen neuer Objekte nach einer Partikelkollision

```
public GameObject go;
void OnParticleCollision(GameObject other) {
    ParticleSystem particleSystem;
    particleSystem = other.GetComponent<ParticleSystem>();
    ParticleSystem.CollisionEvent[] collisionEvents =
        new ParticleSystem.CollisionEvent[8];
    int quantityCollisionEvents =
        particleSystem.GetCollisionEvents(gameObject, collisionEvents);
    int i = 0;
    while (i < quantityCollisionEvents) {
        Vector3 pos = collisionEvents[i].intersection;
        Destroy(Instantiate (go, pos, Quaternion.identity), 0.5F);
        i++;
    }
}
```

## ■ 11.9 Feuer erstellen

Mithilfe dieser vielen verschiedenen Module können Sie nun sehr interessante Effekte erzeugen. Als Demonstration möchte ich an dieser Stelle zeigen, wie Sie mit *Shuriken* das Feuer einer Fackel erstellen, das übrigens auch in dem Beispiel-Game eingesetzt wird.



### Nachbilden natürlicher Effekte

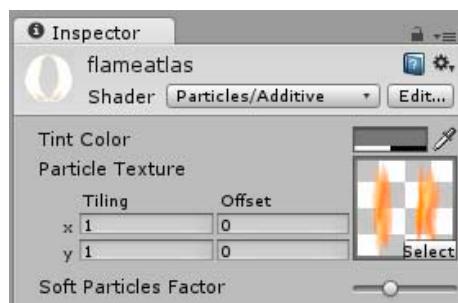
Wenn Sie Effekte aus der Natur nachbilden wollen, z. B. Regen, Feuer oder Rauch, hilft ein genaues Beobachten der Natur. Am besten nutzen Sie eine Videoaufnahme, die Sie z. B. auch in Slow Motion abspielen können, um das Verhalten des nachzubildenden Effektes detailliert zu analysieren.

Das Besondere an diesem Partikelleffekt ist der, dass wir hier zwei Partikelsysteme einsetzen werden, die gleichzeitig Ihre Effekte erzeugen. Dabei werden wir mit dem einen die Flamme des Feuers darstellen und das andere für den Rauch nutzen.

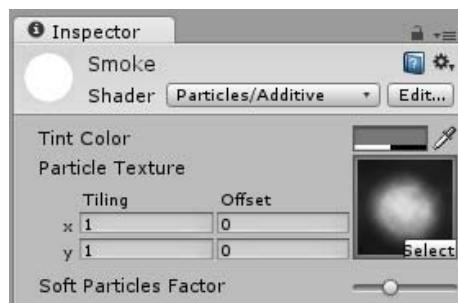
### 11.9.1 Materials erstellen

Zunächst importieren Sie die Texturen und legen die *Import Settings* fest. Für dieses Beispiel können Sie die Texturen „smoke“ und „flame\_atlas“ aus dem Beispiel-Game nutzen, die sich dort im „Textures“-Ordner befinden. Hierbei ist vor allem die *Max Size* wichtig, die bei der „smoke“-Textur auf 256 und bei der größeren „flame\_atlas“-Textur auf 512 stehen sollte. Aber auch den *Wrap Mode* sollten Sie noch einmal kontrollieren und auf *Clamp* stellen.

Als Nächstes erstellen Sie nun in Ihrem *Project Browser* über das Kontextmenü der rechten Maustaste mit **Create/Material** zwei *Materials* und weisen beiden den *Shader* „Additive“ aus dem „Particle“-Bereich (*Particles/Additive*) zu (siehe Bild 11.5 und 11.6). Ansonsten verbleiben die restlichen Parameter der beiden *Materials* auf ihren Default-Einstellungen. Das Flammen-Material nennen Sie nun „Flameatlas“ und das andere nennen Sie „Smoke“. Die restlichen Parameter verbleiben auf ihren Default-Einstellungen.



**Bild 11.5**  
Feuer-Material



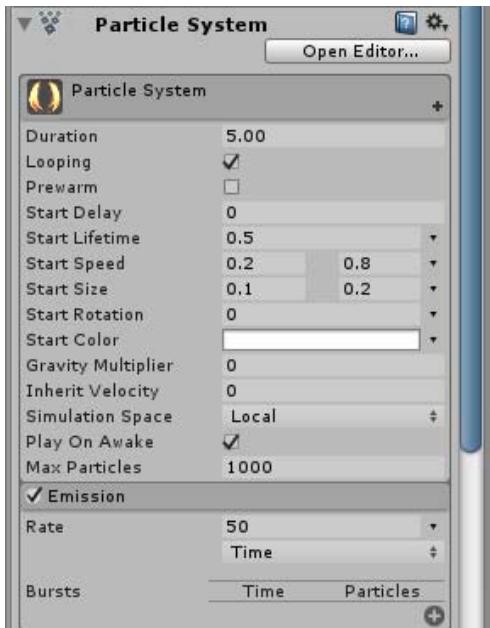
**Bild 11.6**  
Rauch-Material

### 11.9.2 Feuer-Partikelsystem

Erstellen Sie zunächst über das Hauptmenü *GameObject/Create Other/Particle System* ein neues Partikelsystem und nennen Sie es beispielsweise „Torchflames“.

Als Erstes stellen Sie nun die Partikelsystem-Parameter des *Default-Moduls* entsprechend Bild 11.7 ein. Bei *Start Speed* und *Start Size* nutzen Sie hierbei *Random Between Two Constants*. Sie könnten noch zusätzlich *Prewarm* aktivieren, da aber die Flammen sehr schnell da sind, ist dies nicht wirklich entscheidend.

Nach diesem Modul folgen die Einstellungen im *Emission-Modul*, das Sie ebenfalls in der Abbildung sehen. Über die Höhe des *Rate*-Wertes steuern Sie, wie viel Feuer am Ende von der Fackel aufsteigt.



**Bild 11.7**

Default-Modul und Emission-Modul  
des Feuers

Anschließend definieren Sie mit dem *Shape-Modul* die Form und Größe des Flammenausstoßes. In Bild 11.8 sehen Sie zudem auch die Einstellungen des *Force over Lifetime-Moduls* und von *Size over Lifetime*, wozu auch die untere Kurve gehört.

*Force over Lifetime* dient in diesem Fall dazu, die Flamme hin und her tanzen zu lassen und kleine Flämmchen mal leicht zu den Seiten fliegen zu lassen. Nutzen Sie hierfür wieder die Einstellung *Random Between Two Constants* und zusätzlich den Parameter *Randomize*. Diese Einstellungen dienen dem zufälligen Hin- und Herflackern des Feuers.

*Size over Lifetime* sorgt wiederum dafür, dass die Flammen nach oben kleiner werden und nicht die ganze Zeit die gleiche Größe behalten. Wählen Sie hierfür *Random Between Two Curves* und selektieren Sie die Kurve. Im unteren Kurvenfenster wird nun diese in vergrößerter Form angezeigt (siehe Bild 11.8). Nun ziehen Sie den rechten Punkt der oberen Gerade im Kurvenfenster etwas nach unten. Sobald Sie den Punkt selektieren, erscheint ein zusätzlicher Punkt, mit dem Sie anschließend den Kurvenverlauf genauer bestimmen können.

Als Nächstes aktivieren Sie das *Sub Emitters-Modul* und fügen *Birth* einen neuen *Sub Emitter* hinzu. Dieses dient später dem Erzeugen des Rauchs (siehe Bild 11.9). Da diese zusätzlichen Partikel aber die Sicht verhindern, empfehle ich Ihnen, erst einmal das Modul wieder zu deaktivieren.

Danach folgt nun das *Texture Sheet Animation Modul*, das Sie ebenfalls aktivieren. Dieses nutzen wir, da unsere Flammentextur zwei nebeneinander angeordnete Flammen-Abbildungen beinhaltet. Diese sollen nun von jedem einzelnen Partikel abwechselnd dargestellt

werden, weshalb die Parameter entsprechend Bild 11.9 definiert werden. Durch das Wechseln der beiden Flammendarstellungen entsteht bei jedem einzelnen Partikel ein eigener Flackereffekt, wodurch diese noch einmal lebendiger wirken.

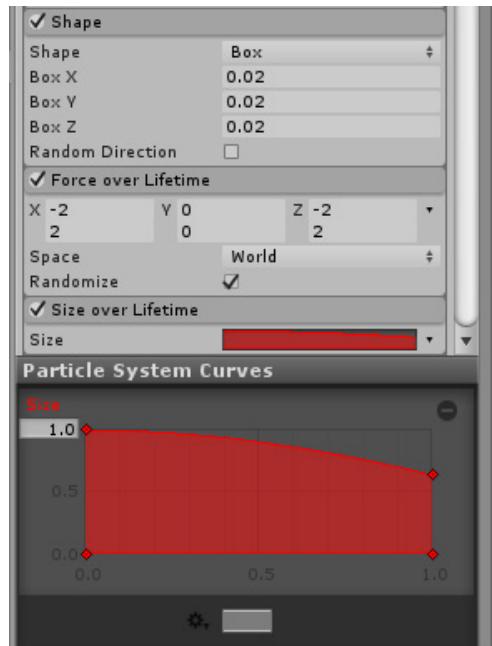


Bild 11.8

Shape, Force over Lifetime, Size over Lifetime des Feuers

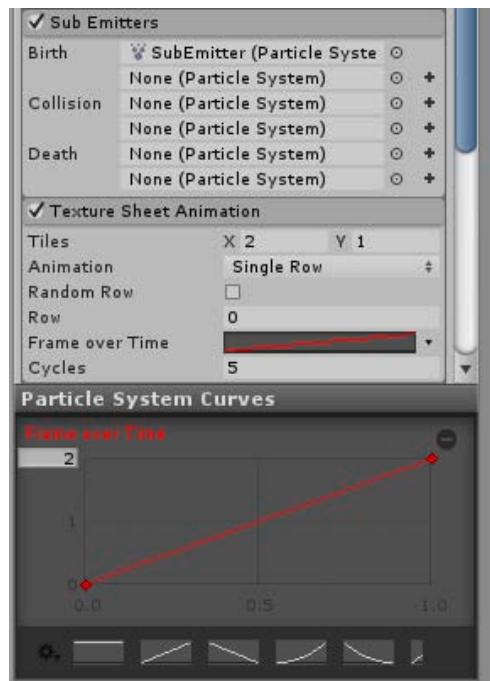
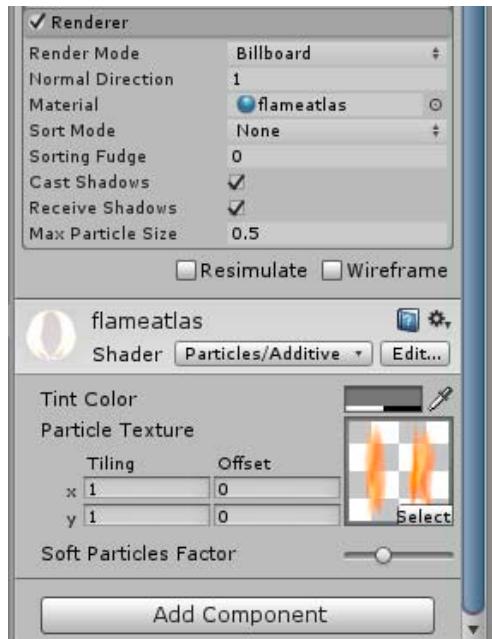


Bild 11.9

Sub Emitters und Texture-Sheet-Animation des Feuers

Zum Schluss des Hauptpartikelsystems kommt nun noch das *Renderer-Modul* (siehe Bild 11.10), dem Sie jetzt nur noch das „Flameatlas“-Material zuweisen müssen. Und schon haben Sie eine flackernde Flamme mit dem *Shuriken*-Partikelsystem erzeugt.

Dies soll aber noch nicht alles sein. Mithilfe des *Sub Emitters* wollen wir nun noch etwas Rauch erzeugen, der noch zusätzlich von dieser Flamme aufsteigt.



**Bild 11.10**

Renderer-Modul des Feuers

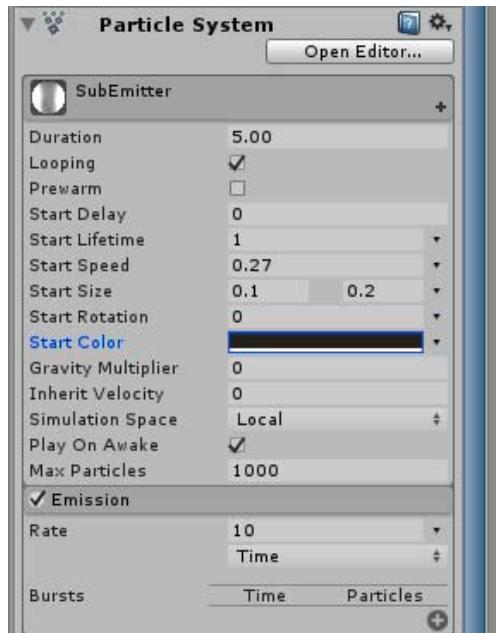
### 11.9.3 Rauch-Partikelsystem

Im zweiten Schritt definieren Sie nun die Parameter des *Sub Emitters*. Aktivieren Sie hierfür wieder das *Sub Emitters-Modul* im Hauptpartikelsystem „Torchflames“.

In der *Hierarchy* sollten Sie nun unter dem Hauptpartikelsystem ein Kind-Objekt namens „*SubEmitter*“ finden. Selektieren Sie dieses und stellen Sie gemäß Bild 11.11 die Parameter des *Default-Moduls* und des *Emission-Moduls* ein. *Start Size* wird auch hier wieder auf *Random Between Two Constants* gestellt.

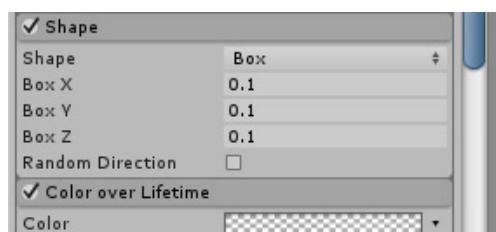
Beachten Sie zudem in Bild 11.11 den Parameter *Start Color*. Durch einen Klick auf den Farbbalken erscheint ein dazugehöriges Farbauswahlfenster. Weisen Sie dort den einzelnen Parametern folgende Werte zu:

- R 36
- G 31
- B 23
- A 255

**Bild 11.11**

Default-Modul und Emission-Modul des Rauchs

Im nächsten Schritt definieren Sie wieder die Form des Emitters über das Modul *Shape* (Bild 11.12).

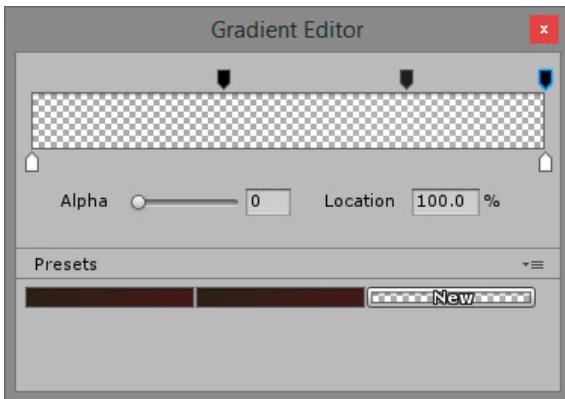
**Bild 11.12**

Shape-Modul und Color over Lifetime des Rauchs

Danach folgt dieses Mal das *Color over Lifetime-Modul*, das bei den Flammen noch nicht zum Einsatz kam. Dieses nutzen wir dafür, dass der Rauch erst später und nicht schon ganz unten am Emitter zu sehen ist. Über das kleine Drop-down-Menü rechts wählen Sie *Gradient* aus und definieren mit einem Klick auf das Farbfeld den genauen Farbverlauf (siehe Bild 11.13).

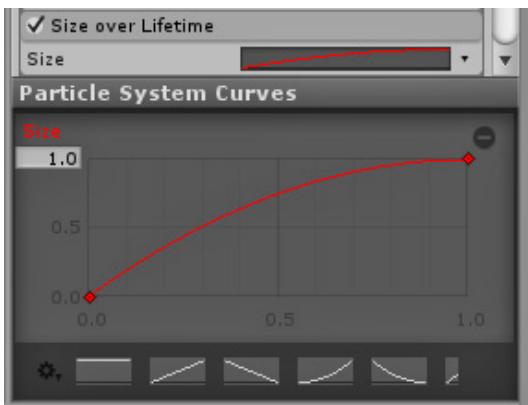
In dem *Gradient Editor* von *Color over Lifetime* definieren Sie per Klick oben insgesamt drei Farbpunkte (siehe Bild 11.13), denen Sie die folgenden Werte von links nach rechts zuweisen:

- Alpha 0, Location 37,4
- Alpha 30, Location 72,9
- Alpha 0, Location 100

**Bild 11.13**

Gradient Editor von Color over Lifetime des Rauchs

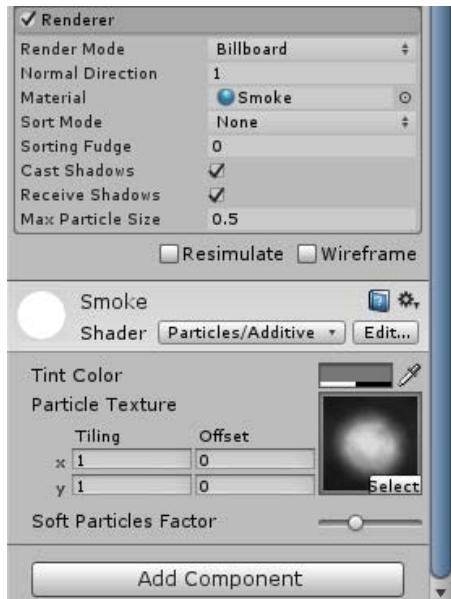
Als nächstes Modul nehmen wir wieder das *Size over Lifetime-Modul*. Hier nutzen wir dieses Mal eine der Standardkurven, die Ihnen bereits am unteren Rand des Kurvenfensters angeboten werden (siehe Bild 11.14). Klicken Sie hierfür auf das Zahnrad, das Ihnen dort links angezeigt wird, und es werden Ihnen weitere Kurven zur Auswahl angeboten. Wählen Sie hier die ansteigende Kurve, die am Ende abflacht und sich schließlich dem Grenzwert 1 nähert.

**Bild 11.14**

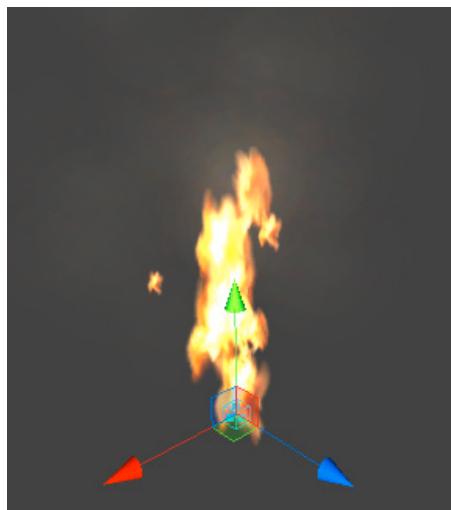
Size over Lifetime des Rauchs

Zuletzt bestimmen Sie wieder durch die Zuweisung des *Materials* im *Renderer-Modul* das eigentliche Aussehen der Partikel, in diesem Fall also des Rauchs. Während Sie für die Flammen das „Flameatlas“-Material zugewiesen hatten, weisen Sie dieses Mal das „Smoke“-Material (siehe Bild 11.15) mit der „smoke“-Textur zu. Und schon haben Sie auch den Raucheffekt Ihres Fackelfeuers erstellt.

Als Ergebnis erhalten Sie nun einen Flammeneffekt, dessen Feuer nicht nur leicht flackert, sondern auch aufsteigenden Rauch abgibt (Bild 11.16). Damit Sie diesen nun beliebig oft wiederverwenden können, ziehen Sie nun das komplette Objekt in den *Project Browser*, um aus dem Effekt ein *Prefab* zu erstellen. Hier empfiehlt es sich, einen Ordner „Prefabs“ anzulegen, in dem alle *Prefabs* des Projektes abgelegt werden.



**Bild 11.15**  
Renderer-Modul des Rauchs



**Bild 11.16**  
Feuer-Partikeleffekt mit Rauch

In dem Beispiel-Game werden wir diesen Partikeleffekt oberhalb einer Fackel positionieren und damit dem 3D-Modell etwas Leben einhauchen. Genutzt wird dieses dann als Wandfackel, das das Dungeon ausleuchten soll.

Da so ein Partikeleffekt natürlich selber nicht leuchtet bzw. Licht abgibt, müssen hier natürlich noch Light-Komponenten eingesetzt werden. In diesem Fall werden wir der Fackel zu diesem Partikeleffekt noch ein weiteres *PointLight*-Objekt hinzufügen, das dann, farblich auf die warme Lichtfarbe des Feuers abgestimmt, den Raum beleuchtet. Wie dies dann am Ende tatsächlich wirkt, sehen Sie im Kapitel „Beispiel-Game“ sowie auf der DVD, wo das komplette Game inklusive des Partikeleffekts enthalten ist.

## ■ 11.10 Wassertropfen erstellen

Neben dem Darstellen von Feuer werden Partikeleffekte auch gerne zum Darstellen von Schnee, Hagel oder auch Regen genutzt.

Da sich diese Effekte doch recht ähneln, möchte ich Ihnen dies anhand von Wassertropfen demonstrieren. Diese sollen aber nicht nur einfach herunterfallen, sondern auch zerplatzen, sobald diese auf dem Boden auftreffen. Wie auch das Feuer, werden wir auch diesen Effekt in dem Beispiel-Game nutzen.

### 11.10.1 Tropfen-Material erstellen

Zuerst erstellen Sie wieder ein *Material-Asset*. Wie beim Feuer machen Sie dies über das Kontextmenü der rechten Maustaste im *Project Browser* mit **Create/Material**. Geben Sie dem *Material* den Namen „Drop“ und weisen Sie wieder den *Shader* „Additive“ aus dem „Particle“-Bereich (**Particles/Additive**) zu (siehe Bild 11.7).

Als Textur können Sie die Textur „waterdrop“ aus dem Beispiel-Game nutzen, die Sie ebenfalls im „Textures“-Ordner finden sollten. Die *Max Size* sollte hierbei auf 64 stehen und der *Wrap Mode* ebenso auf *Clamp* eingestellt sein.

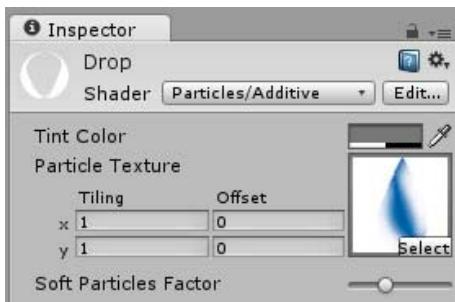


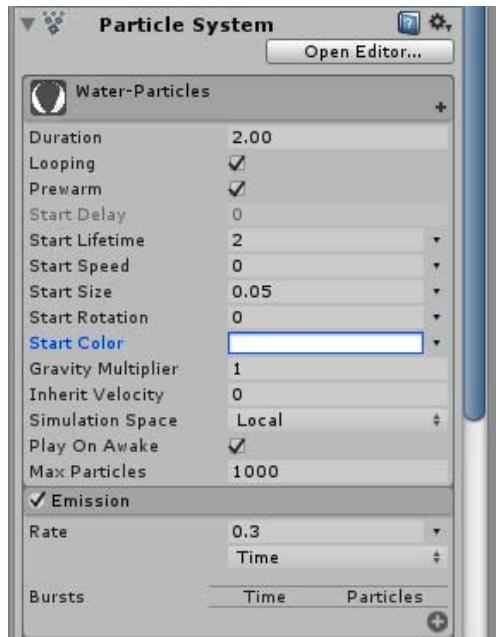
Bild 11.17  
Wassertropfen-Material

### 11.10.2 Wassertropfen-Partikelsystem

Erstellen Sie ein neues Partikelsystem und nennen Sie dieses „Waterdrops“. Zuerst kommt wieder das *Default-Modul* an die Reihe (siehe Bild 11.18). Beachten Sie hier besonders den Parameter *Gravity Multiplier*. Dieser sorgt dafür, dass der Tropfen auch tatsächlich hier herunterfällt.

Beim folgenden *Emission-Modul* legen Sie die Menge der Wassertropfen fest, die erzeugt werden sollen. Hier können Sie gerne selber herumexperimentieren, welche Menge Ihnen persönlich am besten gefällt. Wollen Sie einen leichten Nieselregen erzeugen, sollte dieser Wert natürlich höher liegen als bei einem tropfenden Wasserhahn.

Genauso können Sie auch den Raum, in dem die Wassertropfen erzeugt werden, ganz einfach über die Parameter des *Shape-Moduls* ändern (siehe Bild 11.19). In dem Beispiel-Game wol-

**Bild 11.18**

Default-Modul und Emission-Modul vom Wassertropfen

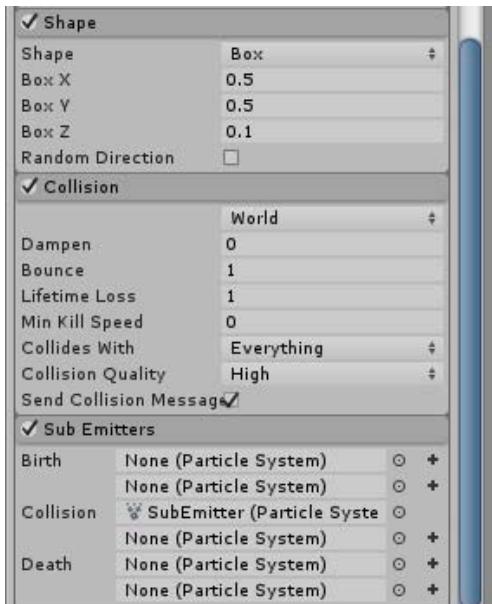
len wir diesen Partikeleffekt dafür nutzen, dass an einer Stelle der Decke Wasser heruntertropft. Soll der Raum, in dem die Tropfen erzeugt werden, größer sein, können Sie dies über diese Parameter sehr einfach steuern.

Da die Wassertropfen zerplatzen sollen, sobald diese auf den Boden (oder einem anderen Objekt) aufkommen, benötigen Sie dieses Mal auch das *Collision Modul*. Dieses soll dafür sorgen, dass einerseits das herunterfallende Partikel bei einer Kollision sofort zerstört und dass die Methode *OnParticleCollision* ausgelöst wird. Diese benötigen wir später, um noch zusätzlich einen *AudioClip* abzuspielen, sobald der Tropfen auf dem Boden aufkommt. Konfigurieren Sie hierfür das Modul wie in Bild 11.19 dargestellt.

Damit aber nun auch tatsächlich irgend etwas zerplatzt, soll in dem Moment, wo die Kollision stattfindet, auch ein neues Partikelsystem erzeugt werden. Dieses stellt schließlich das Zerplatzen dar. Für diesen Zweck deklarieren Sie im folgenden *Sub Emitters-Modul* wieder einen neuen *Sub Emitter*, der dieses Mal aber bei einer Kollision erzeugt werden soll.

Am Ende fügen Sie dem *Renderer Modul* noch das anfangs erzeugte „Drop“-Material zu (Bild 11.20), und fertig ist der Wassertropfen.

Jetzt brauchen Sie nur noch das Zerplatzen des Tropfens zu erstellen, was wir aber auch gleich im nächsten Abschnitt machen werden. Hierzu gehört aber nicht nur der Partikelleffekt, der das Zerplatzen optisch darstellt. In diesem soll auch, wie bereits oben erwähnt, ein Tropfgeräusch zu hören sein. Hierfür werden wir am Ende noch ein kleines Skript namens *WaterdropSound* erstellen, das wir ebenfalls noch diesem Partikelsystem anhängen werden (siehe Bild 11.20).

**Bild 11.19**

Shape-, Collision- und Sub Emitters-Modul vom Wassertropfen

**Bild 11.20**

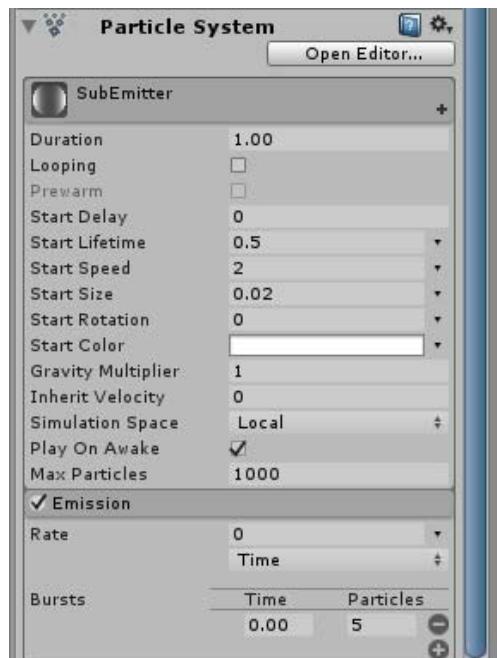
Renderer-Modul und Waterdrop-Skript

### 11.10.3 Kollisionspartikelsystem

Unter dem Hauptpartikelsystem sollte sich wieder ein Kind-Objekt mit Namen „SubEmitter“ befinden. Dieses erzeugt ein Partikelsystem, sobald ein Partikel des Hauptsystems mit etwas anderem kollidiert. Mit diesem System wollen wir nun die Wasserspritzer erstellen, die durch das Zerplatzen des Wassertropfens entstehen.

Zum Testen empfiehlt es sich hier, unter dem Partikelsystem eine Plane oder einen Cube zu platzieren, damit es auch tatsächlich zu einer Kollision kommen kann.

Das *Default-Modul* und das *Emission-Modul* parametrisieren Sie entsprechend dem Bild 11.21. Für die Bewegung der Wasserspritzer, die durch den Aufprall entstehen sollen, sind hierbei vor allem die Anfangsgeschwindigkeit (*Start Speed*) sowie der *Gravity Multiplier* verantwortlich.

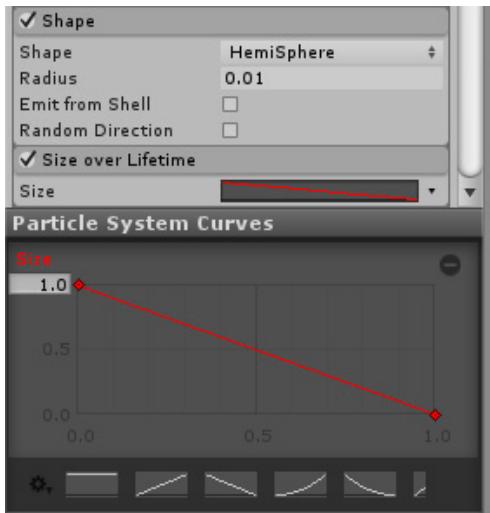


**Bild 11.21**  
Sub Emitter für Wasserspritzer

Aber auch die Form des Emitters, die Sie über das *Shape-Modul* definieren, ist für das Erzeugen der Wasserspritzer wichtig. Beachten Sie hierbei die Form *HemiSphere* und auch den eingestellten Radius, der in diesem Fall punktgenau auf dem Kollisionspunkt liegen sollte und deshalb sehr klein ist (siehe Bild 11.20).

Nach diesem Modul folgt dann das *Size over Lifetime-Modul*, das Sie ebenfalls auf dem Bild 11.20 sehen. Dieses dafür sorgt, dass die Partikel nach dem Erscheinen linear kleiner werden und schließlich verschwinden.

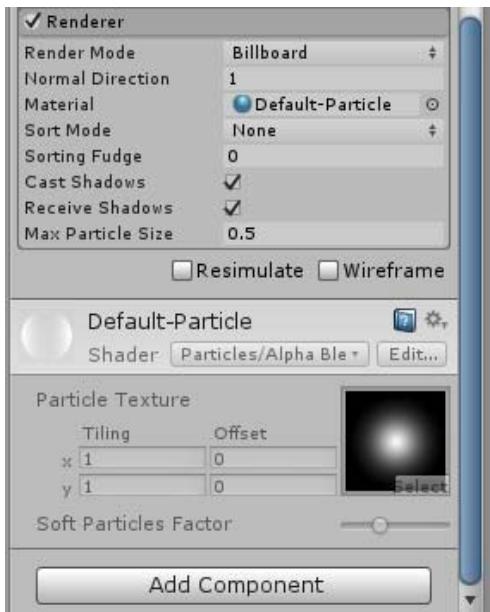
Damit haben Sie es auch schon im Grunde geschafft. Jetzt kommt nur noch das *Renderer-Modul*, das noch gebraucht wird. Da die erzeugten Partikel sehr klein sind, können Sie in diesem Fall einfach das mitgelieferte Default-Material „Default-Particle“ von Unity nutzen

**Bild 11.22**

Shape- und Size over Lifetime-Modul der Wasserspritzer

und brauchen hier nichts mehr zu ändern. Dementsprechend sieht das *Renderer-Modul* des zweiten Partikelsystems für den Kollisionseffekt wie in Bild 11.23 aus.

Die Wasserspritzer werden nun direkt nach der Erzeugung durch eine Kollision kurz nach oben in die verschiedensten Richtungen geschleudert und fallen gleich wieder durch die Gravitation angezogen (*Gravity Multiplier*) herunter und verschwinden schließlich durch die Parametrisierung im *Size over Lifetime-Modul*.

**Bild 11.23**

Renderer-Modul der Wasserspritzer

## 11.10.4 Kollisionssound

Die Kollision eines Wassertropfens soll nicht nur durch das obige Partikelsystem dargestellt, sondern auch noch durch Sound untermauert werden. Hierfür nutzen wir die *OnParticleCollision*-Methode und spielen überall dort, wo eine Kollision stattfindet, mithilfe der statischen Methode *PlayClipAtPoint* der  *AudioSource*-Klasse einen Sound ab.

Erzeugen Sie hierfür ein neues Skript mit dem Namen „WaterdropSound“ und hängen Sie das ausprogrammierte Skript (Listing 11.4) dem Hauptpartikelsystem an.

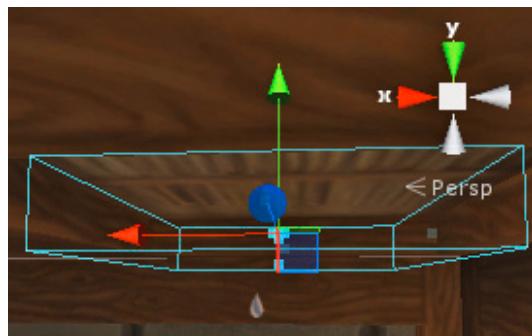
**Listing 11.4** Sounderzeugung durch eine Partikelkollision

```
using UnityEngine;
using System.Collections;

public class WaterdropSound : MonoBehaviour {
    public AudioClip clip;

    void OnParticleCollision(GameObject other) {
        ParticleSystem.CollisionEvent[] collisionEvents =
            new ParticleSystem.CollisionEvent[5];
        int quantityCollisionEvents =
            particleSystem.GetCollisionEvents(other, collisionEvents);
        int i = 0;
        while (i < quantityCollisionEvents) {
            Vector3 pos = collisionEvents[i].intersection;
            AudioSource.PlayClipAtPoint(clip, pos, 0.3F);
            i++;
        }
    }
}
```

Das Bild 11.24 zeigt das fertige Partikelsystem, wie es gerade einen Tropfen erzeugt. Dieser wird schließlich nach unten fallen und bei einer Kollision mit einem anderen *Collider*-Objekt zerplatzen. Hierbei wird dann mithilfe des *WaterdropSound*-Skriptes ein *AudioClip* abgespielt, dessen Originaldatei Sie vorher der *clip*-Variablen zugewiesen haben (z. B. „waterdrop.wav“, das Sie im Ordner „Audio“ des Beispiel-Games finden).



**Bild 11.24**  
Fertiges Wassertropfen-Partikelsystem

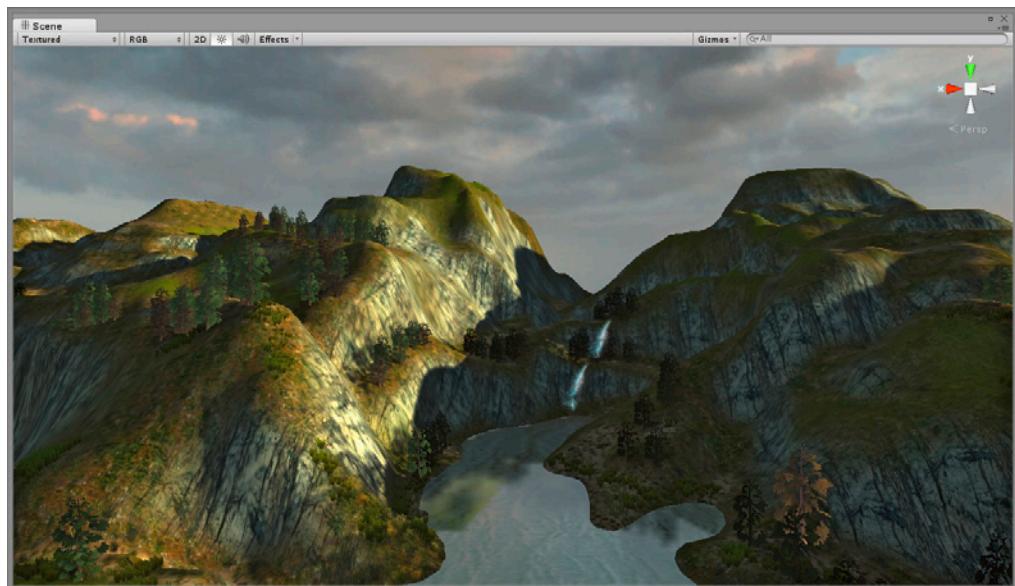
Lizenziert für gregwyss@hotmail.com.

© 2014 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

# 12

## Landschaften gestalten

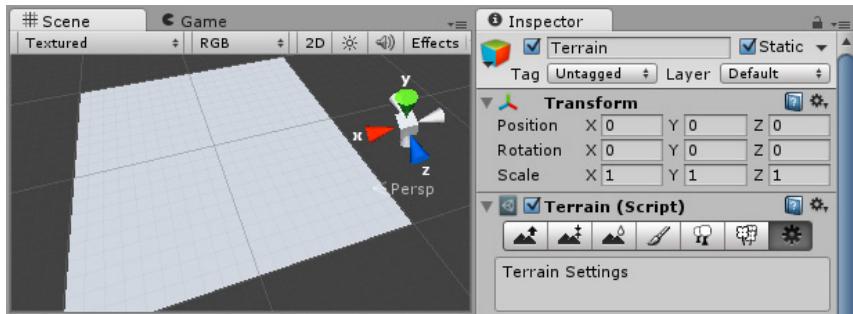
Um Landschaften von Outdoor-Spielen zu gestalten, bietet Unity den Objekttyp *Terrain* an, der nicht nur eine Landschaft an sich darstellen kann, sondern auch die dementsprechenden Tools mitbringt, um diese zu gestalten. Mit *Terrains* können Sie Levels gestalten, die flach, hügelig oder gar gebirgig sind. Sie können diese aufwendig texturieren, diese mit Geröll, Felsen, Gräsern oder auch Bäumen bedecken. *Terrains* finden Sie im Menü **GameObject/Create Other/Terrain** bzw. ab Version 4.6 **GameObject/Create General/Terrain**.



**Bild 12.1** Eine auf Terrains basierende Landschaft

## ■ 12.1 Was Terrains können und wo die Grenzen liegen

Zunächst einmal sind *Terrains* rechteckige Flächen, die Sie aber mithilfe von verschiedenen Werkzeugen verformen, mit Texturen bemalen und nicht zuletzt auch mit Bäumen und Gräsern bepflanzen können. Ein spezieller Terrain-Collider passt sich dabei den Verformungen des *Terrains* an, um so der Oberfläche zu entsprechen.



**Bild 12.2** Ein Terrain mit seinen Terrain-Tools

Allerdings haben *Terrains* auch Grenzen. So können diese nur in die Y-Richtung verformt werden, nicht aber in die X- oder Z-Richtung. Das bedeutet, dass Sie zwar Hügel, Gebirge und Ähnliches erstellen können, Sie können aber keine Höhlen oder gar Tunnel designen. Möchten Sie so etwas in Ihre Spielwelt integrieren, werden Sie nicht um Extratools herumkommen, mit denen Sie dann solche Objekte designen. Das können z.B. herkömmliche 3D-Modelling-Programme wie Blender sein, aber auch spezielle *Editor Extensions* aus dem *Asset Store* könnten so etwas leisten.

*Terrains* haben aber noch weitere Besonderheiten, die sie von anderen *GameObjects* in Unity gravierend unterscheiden. Ein wichtiges Merkmal ist hier die *Transform*-Komponente. So können Sie ein Terrain weder über *Rotation* drehen, noch mit *Scale* skalieren. Zum Skalieren finden Sie hier noch passende Parameter in den *Terrain Settings*, ein Rotieren ist aber auch dort nicht möglich.

## ■ 12.2 Terrainhöhe verändern

Ein Terrain bietet Ihnen gleich mehrere Möglichkeiten an, die Oberfläche zu verändern. So dienen die ersten drei Werkzeuge der Terrain-Tools, die Sie in Bild 12.1 sehen können, dem Verformen der Oberfläche mithilfe von Pinselwerkzeugen.

Es gibt aber auch die Möglichkeit, die Höhenverhältnisse mit einer *Heightmap* zu bestimmen. Hierbei handelt es sich um eine Textur, die mit unterschiedlichen Farben die Höhenunterschiede beschreibt. Diese Textur wird dann auf das Terrain angewendet, wodurch die Oberfläche dann entsprechend verformt wird.

## 12.2.1 Pinsel

Im oberen Bereich der Tools wird Ihnen eine Palette mit Pinselformen (*Brushes*) angeboten. Je nach Form des gewählten Pinsels wird schließlich die Höhe des Terrains entsprechend angehoben.

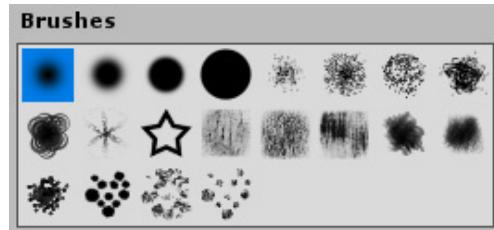


Bild 12.3  
Pinsel-Auswahl

Je stärker die Farbe des Pinsels dabei ist, desto stärker wird das Terrain an der Stelle angehoben. Jeder *Brush* hat dabei folgende Parameter:

- **Brush Size** legt die Größe des Pinsels fest und damit die Fläche, die angehoben wird.
- **Opacity** legt die Intensität des Pinsels fest und damit, wie stark die Fläche angehoben wird.

## 12.2.2 Oberflächen anheben und senken

Mit dem ersten Werkzeug, dem *Raise/Lower-Tool*, können Sie die Oberfläche, wo sich Ihr Mauszeiger befindet, über Ihre linke Maustaste anheben und absenken.

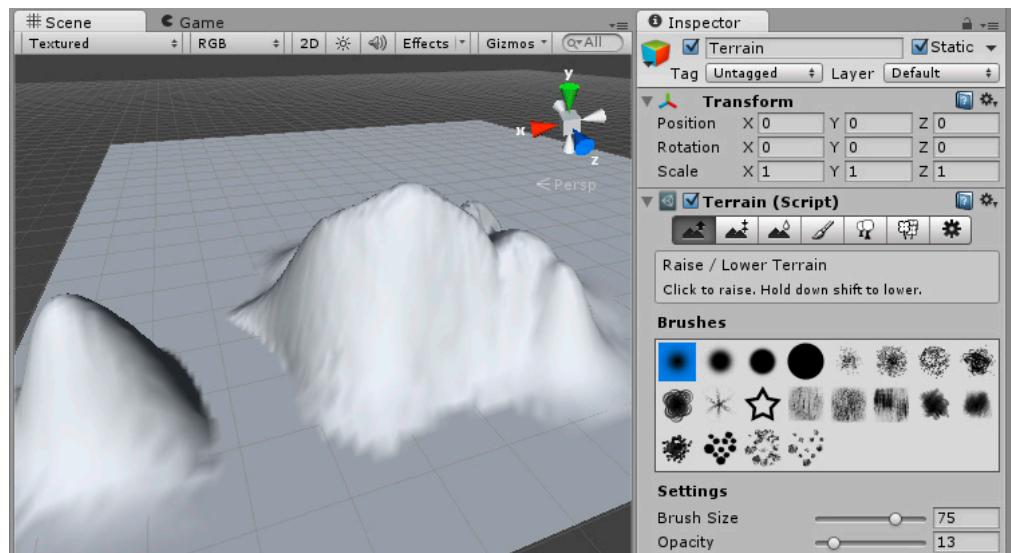
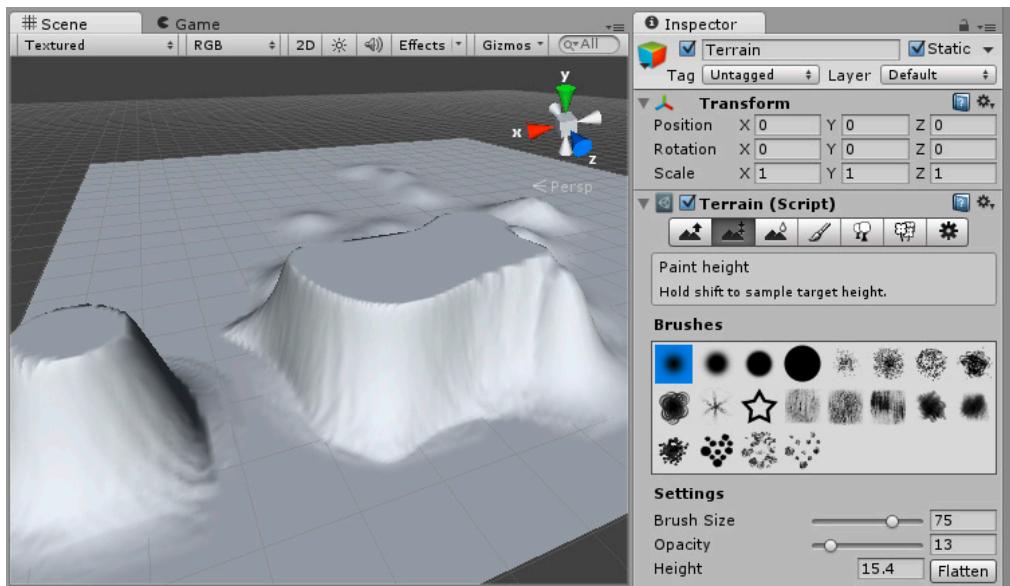


Bild 12.4 Einsatz des Raise/Lower-Tools

Sie können diesen Effekt aber auch umdrehen. Halten Sie nämlich zu der linken Maustaste auch die **Umsch**-Taste (**Shift**) gedrückt, senken Sie das *Terrain* stattdessen ab. Allerdings kann die Fläche nicht tiefer als der Nullpunkt des *Terrains* liegen. Sollten Sie also ein Tal designen wollen, müssen Sie die gesamte Oberfläche zunächst einmal anheben und den Teil des Tals nachträglich absenken oder gleich vom Anheben aussparen. Für ein solches Vorgehen eignet sich die nächste Funktion aber etwas besser.

### 12.2.3 Plateaus und Schluchten erstellen

Mit der zweiten Funktion, dem *Paint Height-Tool*, heben Sie die Flächen zunächst einmal genauso an wie mit dem *Raise/Lower-Tool*, aber mit dem Unterschied, dass die Oberfläche nicht höher angehoben werden kann, als es der im zusätzlichen *Height*-Parameter hinterlegte Wert festsetzt. Alle Flächen, die diesen Wert erreichen, werden auf dieser Höhe geebnet.

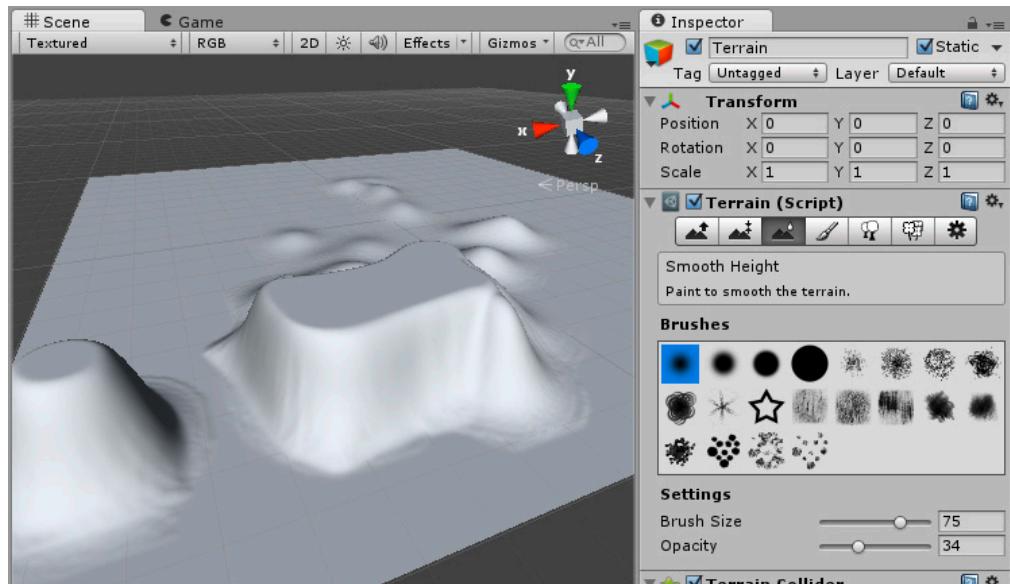


**Bild 12.5** Einsatz der Paint Height-Tools

Über den Button *Flatten* können Sie jetzt die gesamte Fläche des *Terrains* auf die angegebene Höhe mit einem einzigen Klick anheben. Für Landschaften mit Tälern und/oder Schluchten ist dies sehr nützlich, da Sie auf diese Art zunächst das gesamte *Terrain* anheben können und anschließend die Täler und Schluchten wieder etwas absenken. Hierfür senken Sie einfach den *Height*-Parameter und führen das Tool mit dem gewünschten Pinsel aus. Alternativ können Sie natürlich auch das *Raise/Lower-Tool* nutzen und mit der **Umsch**-Taste das Tal herausarbeiten.

## 12.2.4 Oberflächen weicher machen

Das *Smooth Height-Tool* ist ein typisches Werkzeug für den Feinschliff. Es kann scharfe Übergänge abrunden oder auch ganze Hügel abflachen. Im zweiten Fall wirkt es dann ähnlich wie ein Radiergummi, der langsam die Farbe wegrubbelt, nur dass er anstelle von Farbe eben den ganzen Hügel entfernt. Sie können aber auch den Übergang eines steilen Hangs zum Boden abrunden und diesen weicher gestalten. In diesem Fall zieht das Tool den scharfen Knick etwas hervor, damit die Ecke zu einer weichen Kurve wird.



**Bild 12.6** Einsatz des Smooth Height-Tools

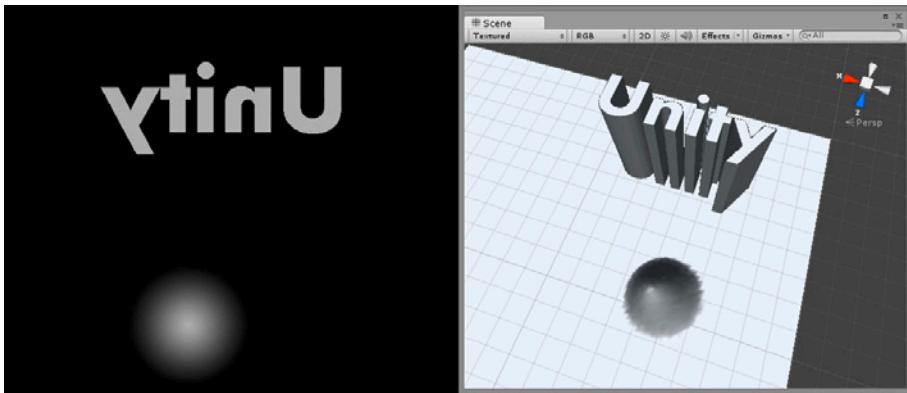
## 12.2.5 Heightmaps

Im letzten Menübereich *Terrain Settings* finden Sie unter anderem den Bereich *Heightmap*. Eine *Heightmap* ist zunächst einmal nichts anderes als eine Schwarz-Weiß-Textur mit Graustufen (Grayscale genannt). Heightmaps können auch bei Shadern eingesetzt werden, anhand derer dann Höhenunterschiede optisch vorgetäuscht werden. Bei Terrains werden diese aber dafür genutzt, um tatsächlich ein Terrain entsprechend dieser Textur zu verformen.

Unity unterstützt aktuell nur *Grayscale*-Grafiken im RAW-Format, weshalb der Import-Button auch **Import Raw** und die Export-Funktion **Export Raw** lautet. Die Engine legt diese Textur dann über das *Terrain* und hebt dann dieses je nach Graustufe mehr oder weniger an. Schwarz bedeutet hierbei das niedrigste Niveau, also 0. Weiß bedeutet wiederum das höchste Niveau. Dies legen Sie mit dem Parameter *Terrain Height* im Bereich *Resolution* fest.

Wenn Sie nun einen Verlauf von Schwarz nach Weiß auf der *Heightmap* darstellen, wird eine Steigung auf dem *Terrain* erzeugt. Stellt die Textur einen weißen Kreis auf einer schwarzen Fläche dar, so wird das *Terrain* so verformt, als würde dort eine Säule stehen.

Ein wichtiger Punkt ist hierbei unbedingt zu beachten: Die *Heightmap* verhält sich so, als wenn diese von unten auf das *Terrain* projiziert wird, wodurch die Anordnung einmal gedreht wird. Wenn Sie diese Drehung nicht möchten, müssen Sie vor dem Import die Grafik noch einmal horizontal spiegeln.



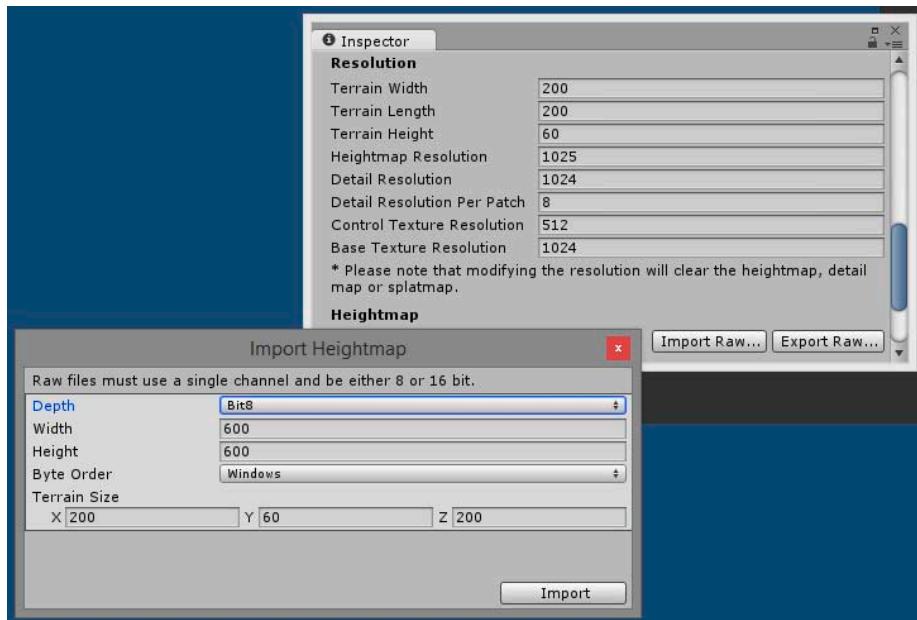
**Bild 12.7** Anwendung einer horizontal gespiegelten Heightmap

### 12.2.5.1 Reale Landschaften als Vorlagen nutzen

Eine *Heightmap* kann natürlich nicht nur einfache Elemente darstellen, sondern beliebig komplex sein. Auf diese Weise können Sie zum Beispiel reale Landschaften in Ihrem Spiel abbilden. Sie brauchen nur einen Screenshot einer Landkarte zu nehmen. Diese färben Sie dann je nach Höhenlage in verschiedenen Grautönen ein (Flüsse könnten zum Beispiel schwarz sein, die normale Höhenlage hat ein Dunkelgrau, kleine Erhebungen zeigen ein etwas helleres Grau usw.), und am Ende erstellt Unity daraus ein Terrain, das ähnlich wie die Karte aussieht. Für den Feinschliff können Sie anschließend wieder auf die obigen Werkzeuge zurückgreifen und beispielsweise die Höhenunterschiede mit dem *Smooth Height-Tool* optimieren.

Um ein gutes Ergebnis zu erzielen, sollte die *Heightmap* die gleichen Maße besitzen wie das *Terrain* selbst. Die Maße finden Sie ebenfalls im Bereich *Resolution* der *Terrain Settings*, wo Sie natürlich diese auch ändern können. Wenn Sie eine Textur nun ausgewählt haben, müssen Sie noch darauf achten, dass die korrekte *Byte Order* und auch die Bit-Tiefe korrekt eingestellt sind. Hierbei werden Sie wohl meistens 8-Bit-Texturen importieren. Bei der *Byte Order* kommt es auf Ihr Betriebssystem und den *Texture-Export* der Grafikbearbeitungssoftware an.

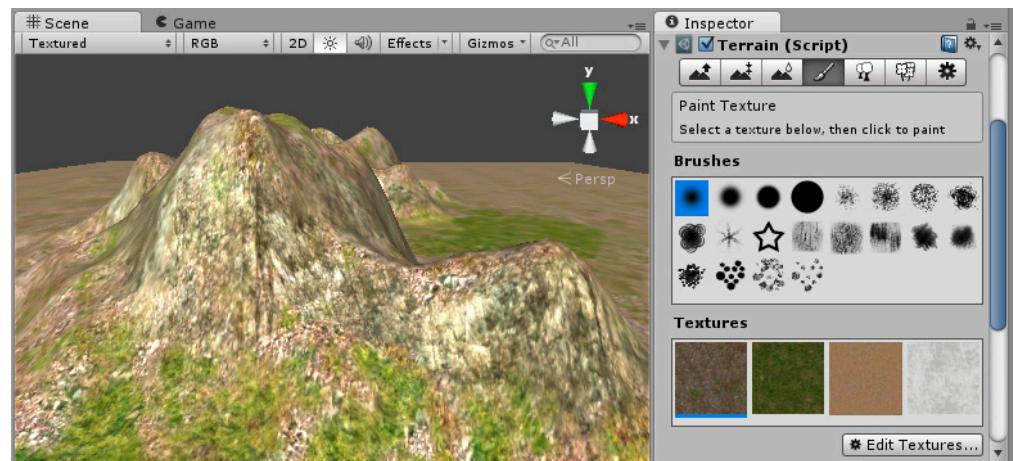
Sie können aber nicht nur eine *Heightmap* auf das *Terrain* anwenden, Sie können auch anhand des *Terrains* mit Unity eine neue *Heightmap* erstellen. In den *Terrain Settings* finden Sie deshalb auch eine Export-Funktion (**Export Raw ...**), wo Sie ebenfalls die gewünschte *Byte Order* und Bit-Tiefe angeben müssen. Auf diese Weise können Sie auf einfache Weise ein *Terrain* sichern oder auch in ein anderes Projekt übernehmen.



**Bild 12.8** Einstellungen beim Heightmap-Import

## ■ 12.3 Terrain texturieren

Mit dem *Paint Texture-Tool* können Sie das *Terrain* mit verschiedenen Texturen bemalen. Sie können hierbei mehrere Texturen hinterlegen, die sich später dann auch überlagern können. Auf diese Weise können Sie sehr komplexe grafische Landschaften erschaffen.



**Bild 12.9** Texturiertes Terrain mit dem Paint Texture-Tool

Achten Sie beim Texturieren darauf, dass Sie nahtlose Texturen (auch *Tiling Textures* genannt) nutzen, also Texturen, die Sie nebeneinander platzieren können, ohne dass ein Übergang zu erkennen ist.

Über den *Asset Import (Assets/Import Package)* finden Sie das *Terrain Assets Package*, das einige solcher nahtlosen Grundtexturen mitbringt. In Bild 12.9 sehen Sie einige in der Praxis. Aber auch im *Asset Store (Window/Asset Store)* finden Sie Unmengen von kostenlosen und kostenpflichtigen Texturen. Aber auch Bäume und anderes Zubehör für Ihre Landschaften können Sie dort finden.

### 12.3.1 Textur-Pinsel

Um die Texturen aufzutragen, werden auch hier Pinsel eingesetzt. Ähnlich wie beim Ändern der Höhe, können Sie auch hier aus der gleichen Pinselpalette einen geeigneten wählen. Allerdings unterscheiden sich die Auswirkungen der Parameter etwas:

- **Brush Size** legt die Größe des Pinsels fest und damit die Fläche, die texturiert wird.
- **Opacity** legt die Intensität des Pinsels fest und damit, wie intensiv ein Pinselklick deckt. Wird doppelt geklickt, verdoppelt sich auch die Deckkraft.
- **Target Strength** legt die maximale Deckkraft fest. Auch wenn Sie mehrfach klicken, kann die Deckkraft nicht höher als dieser Wert sein.

### 12.3.2 Texturen verwalten

Über den Button **Edit Textures** öffnen Sie ein kleines Button-Menü, in dem Sie über **Add Texture** Ihrer Texturen-Palette neue Texturen zufügen können.

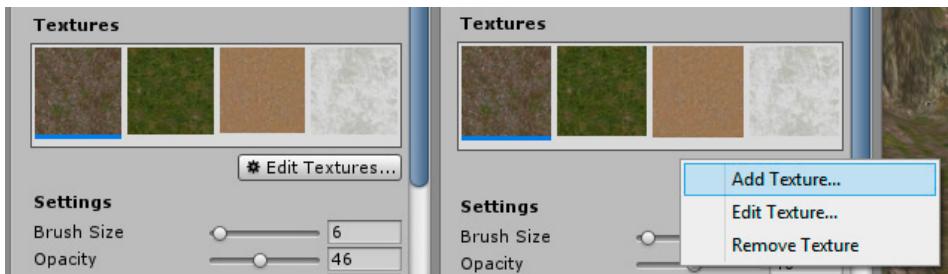
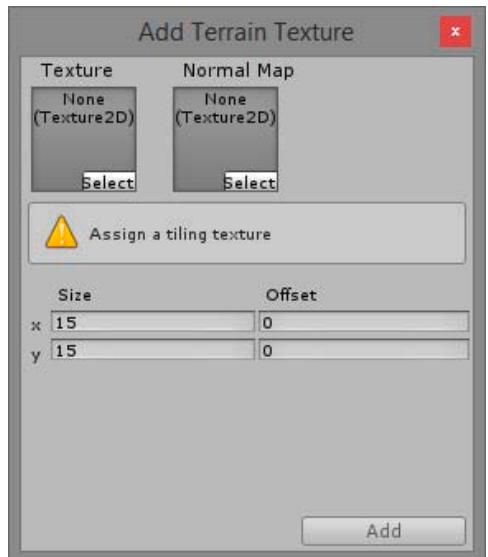


Bild 12.10 Texturen-Verwaltung

Mit **Edit Texture** können Sie diese später dann editieren und auch wieder entfernen (**Remove Texture**). Wenn Sie nun auf **Add Texture** klicken, öffnet sich ein kleines Fenster.

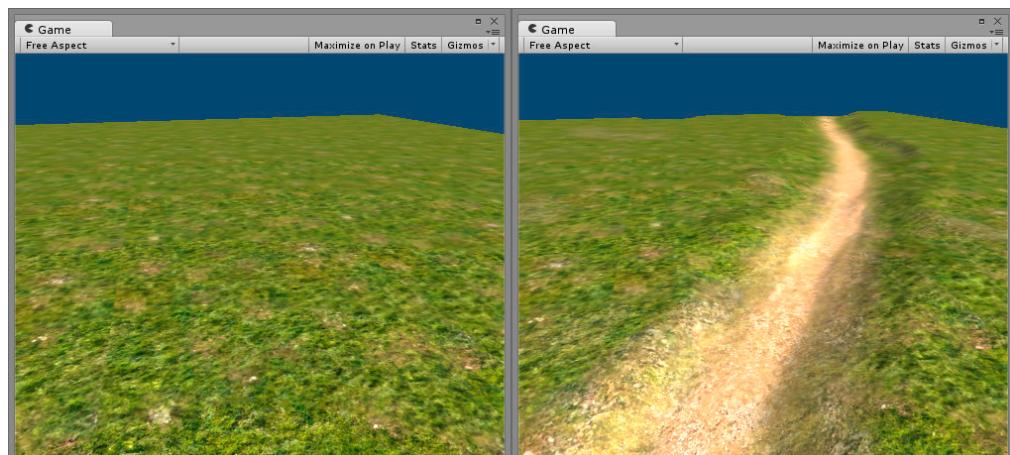
Über die *Select-Buttons* können Sie nun Ihre Projekttexturen durchsuchen und eine geeignete auswählen. Falls erwünscht, können Sie auch eine dazugehörige *Normalmap* auswählen, was aber natürlich nicht zwingend notwendig ist.

**Bild 12.11**

„Add Terrain Texture“-Fenster

- **Size** bestimmt, wie häufig eine Textur wiederholt wird. Ein Verändern dieses Parameters wirkt sich übrigens auch unmittelbar auf die bereits bemalten Stellen des Terrains aus. Wenn Sie also über die Funktion **Edit Texture** des **Edit Textures**-Button diesen Wert nachträglich ändern, können Sie direkt auf dem Terrain verfolgen, welchen Einfluss dieser Parameter hat.
- **Offset** versetzt den Startpunkt der Textur. Da die Textur später wiederholt wird, können Sie so bestimmen, mit welcher Stelle der Textur an den Seiten begonnen wird.

Beim Zusammenstellen der Texturen sollten Sie darauf achten, dass die allererste Textur wie eine Art Grundierung sofort dem gesamten *Terrain* zugewiesen wird. Möchten Sie eine andere nehmen, dann löschen Sie die ausgewählte einfach mit **Delete Texture** aus der *Texture*-Palette und fügen eine neue hinzu. Natürlich können Sie die zuerst ausgewählte

**Bild 12.12** Einfach texturiertes Terrain vs. Terrain mit Akzenten

auch mit anderen Texturen wieder übermalen. Aber es ist vorteilhaft, wenn Sie dies von vornherein beachten.

Die nachfolgenden Texturen werden dann über diese erste Textur gemalt und setzen damit dann Akzente, z.B. um einen Sandweg anzudeuten oder mit einer Steintextur einen Berghang darzustellen. Der Schlüssel für ein natürliches Aussehen ist hier die Abwechslung. Nichts wirkt künstlicher als eine flache grüne Wiese mit einer einzigen Textur (siehe Bild 12.12).

## ■ 12.4 Bäume und Sträucher

Zu einer natürlich wirkenden Landschaft gehört nicht nur eine gute Bodenstruktur und -textur, es gehören auch Bäume, Sträucher und andere platzierte Objekte dazu. Hierfür gibt es das *Place Tree-Tool*.

Mit diesem können Sie Ihrem *Terrain* eben solche Modelle zufügen. Ein besonders interessantes Feature ist hierbei, dass Sie die Bäume später auf Wunsch auch noch mit *Wind Zones* (siehe Kapitel „Wind Zones“) zusätzlich animieren können. Für diese Bäume, Sträucher etc. gibt es in den *Standard Assets* wie auch im *Asset Store* bereits jede Menge kostenlose und auch kostenpflichtige *Asset Packages*, die Sie nutzen können.



Bild 12.13 Place Tree-Tool mit platzierten Bäumen

### 12.4.1 Bedienung des Place Tree-Tools

Über die Funktion **Edit Trees** stellen Sie eine Auswahl an Bäumen (Tree-Objekte) zusammen, die Sie nutzen wollen. Eine Besonderheit hierbei ist der **Bend Factor**, den Sie definieren können, sobald Sie ein Objekt ausgewählt haben. Dieser legt fest, wie stark eine *Wind Zone* das Objekt später biegt. Bei leichten Sträuchern sollten Sie diesen Wert also etwas

höher legen als bei einer starren Eiche. Wenn Sie aber keine *Wind Zone* nutzen, können Sie den Wert auch ignorieren.

Mit gedrückter linker Maustaste erzeugen Sie Objekte des aktuell ausgewählten Objekttyps auf dem Terrain. Wie viele Sie erstellen, hängt von den gewählten Pinseleigenschaften (siehe unten) ab. Wie auch beim *Raise/Lower-Tool* können Sie hier durch zusätzliches Halten der **[Umsch]**-Taste den Effekt umdrehen. In diesem Fall bedeutet es das Entfernen der bereits platzierten Bäume.

Im Falle des *Place Tree-Tools* bietet der Pinsel einige Zusatzeigenschaften an.

- **Brush Size** legt den Radius des Pinsels fest, mit dem Sie die Bäume platzieren.
- **Tree Density** legt fest, wie nah die Bäume aneinander stehen.
- **Color Variation** sorgt dafür, dass sich die platzierten Tree-Instanzen farblich unterscheiden.
- **Tree Height** legt die Höhe der Bäume fest.
- **Height Variation** sorgt dafür, dass die Bäume sich in der Höhe unterscheiden dürfen.
- **Tree Width** legt die Höhe fest.
- **Width Variation** sorgt dafür, dass die Durchmesser der Bäume sich unterscheiden dürfen.

## 12.4.2 Wälder erstellen

Mit der Funktion **Mass Place Trees** können Sie eine beliebige Anzahl an *Tree*-Objekten auf dem gesamten *Terrain* per Zufall platzieren lassen. Die Anzahl geben Sie hierbei nach dem Klick auf den Funktionsknopf an.

## 12.4.3 Mit Bäumen kollidieren

*Tree*-Objekte haben häufig keinen *Collider*. Nichtsdestotrotz spricht nichts dagegen, dass Sie das *Prefab* des Baums mit einem *Collider* ausstatten. Bitte beachten Sie hierbei, dass nur *Capsule-Collider* unterstützt werden. Damit diese nun auch wirklich funktionieren, müssen Sie zusätzlich noch in der *Terrain Collider*-Komponente die Eigenschaft *Create Tree Collider* aktivieren.

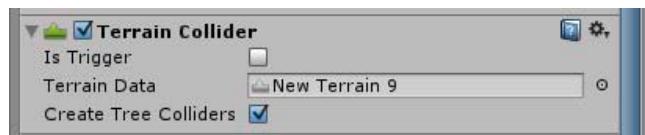


Bild 12.14

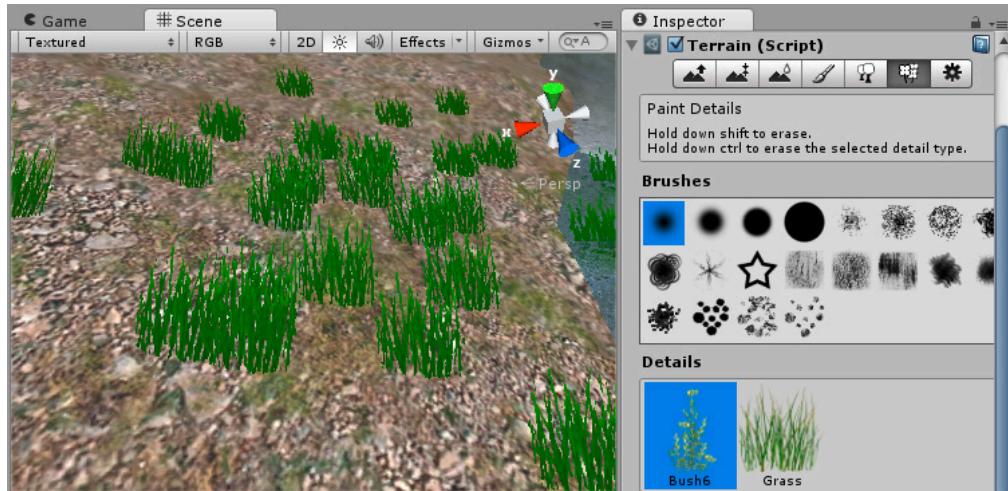
Terrain Collider-Komponente

Erst jetzt werden die *Collider* auch von der Physik-Engine in ihren Berechnungen berücksichtigt. Sollten Sie allerdings hierbei eine Fehlermeldung erhalten, könnte es daran liegen, dass Sie zu viele Bäume in Ihrer Szene haben. Denn Unity begrenzt hier aktuell die Zahl der *Collider* auf 65536. Und würde diese Grenze durch die Bäume überschritten werden, wird

ein Fehler beim Aktivieren ausgelöst. Allerdings sollten Sie auch schon aus Performance-Gründen vermeiden, zu viele Bäume auf diese Weise in das Spiel einzubauen; denn am Ende bremst es doch ziemlich die Performance.

## ■ 12.5 Gräser und Details hinzufügen

Im Gegensatz zu größeren Objekten wie Bäume oder Sträucher, die Sie mit dem *Place Tree-Tool* zufügen können, bietet das *Terrain* als weiteres Werkzeug das *Paint Details-Tool* an, welches durch ein Blumen-Symbol gekennzeichnet wird.



**Bild 12.15** Paint Details-Tool mit platziertem Grass

Mit diesem können Sie statische *Meshes* wie Steine oder auch Gräser, die im Endeffekt sogar nur aus einer Textur bestehen, dem Terrain zufügen.

Über *Edit Details* fügen Sie der *Details-Palette* Objekte hinzu, die Sie dann dem *Terrain* zufügen können. Dabei gibt es zwei unterschiedliche Detailtypen:

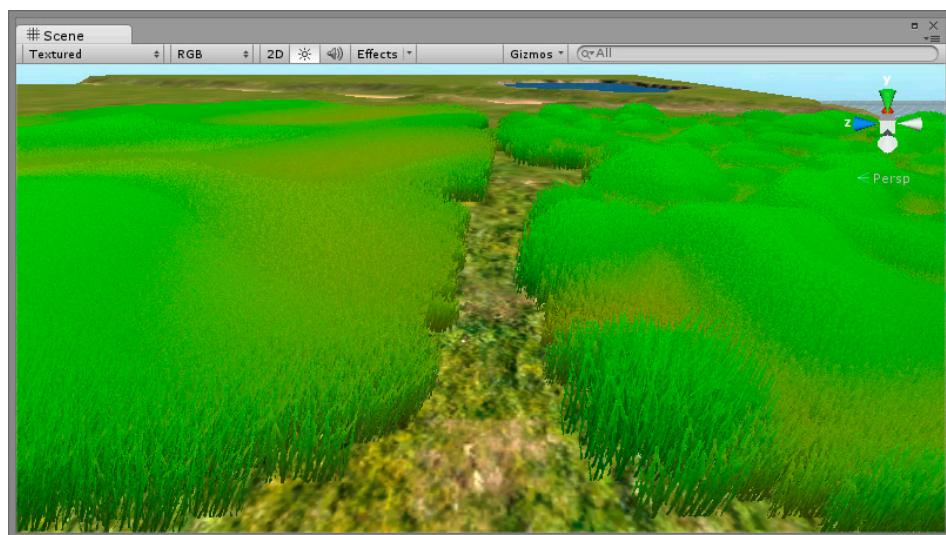
- **Add Grass Texture** wird genutzt, um Texturen für Gräser der Palette zuzufügen.
- **Add Detail Mesh** wird genutzt, um einfache *Mesh*-Objekte der Palette zuzufügen.

### 12.5.1 Detail-Meshs

*GameObjects*, die Sie über **Add Detail Mesh** der Palette zufügen, haben einen rein dekorativen Zweck, z.B. Felsbrocken, kleinere Büsche, und sollten nicht zu aufwendig gestaltet sein. Im Gegensatz zum *Place Tree-Tool* werden diese Objekte nicht von *Wind Zones* beeinflusst.

Wenn Sie hierüber ein *Mesh* auswählen, werden Ihnen dabei folgende Parameter angeboten:

- **Detail** legt das *Mesh* fest.
- **Noise Spread** legt die Menge von Objektansammlungen fest. Die Detailobjekte werden als Anhäufungen (*Cluster*) von Objekten dargestellt, wo die Objekte im Mittelpunkt am größten sind und nach außen immer kleiner werden. Je höher dieser Wert ist, desto mehr gibt es von solchen *Clustern* mit eigenen Mittelpunkten.
- **Random Width** legt fest, wie stark die Breite variieren darf. Gleichzeitig ist das Maximum der Wert, den die Objekte im Zentrum eines *Clusters* haben.
- **Random Height** legt fest, wie stark die Höhe variieren darf. Gleichzeitig ist das Maximum der Wert, den die Objekte im Zentrum eines *Clusters* haben.
- **Healthy Color** legt die Farbe fest, die die Objekte im Zentrum eines *Clusters* besitzen.
- **Dry Color** legt die Farbe fest, die die Objekte am äußeren Rand eines *Clusters* besitzen.
- **Render Mode** legt den *Render Mode* für diese Objekte fest. Es stehen *Vertex Lighting* und *Grass Lighting* zur Verfügung. Bei Detailobjekten wie Steinen sollten Sie *Vertex Lighting* nutzen, da hier auch die Form der Objekte dargestellt werden soll. Beim *Grass Lighting* werden dagegen lediglich die Texturen ohne Berücksichtigung der dreidimensionalen *Mesh*-Form dargestellt.



**Bild 12.16** Gräser mit Noise Spread-Werten 0.1 (links) und 0.5 (rechts)

### 12.5.2 Gräser

Möchten Sie Gräser oder andere reine Texturdetails der Szene zufügen, wählen Sie **Add Grass Texture**. In diesem Fall stellt Unity die Texturen auf kleinen 2D Planes dar, ähnlich der Partikel eines Partikelsystems. Beim Auswählen einer Textur stehen Ihnen folgende Parameter zur Auswahl, um die Textur abwechslungsreicher zu gestalten:

- **Detail Texture** legt die Textur fest.
- **Min Width** legt die minimale Breite fest.
- **Max Width** legt die maximale Breite fest.
- **Min Height** legt die minimale Höhe fest.
- **Max Height** legt die maximale Höhe fest.
- **Noise Spread** legt die Menge von Objektansammlungen fest. Die Grastexturen werden in sogenannten *Clustern* dargestellt. Das bedeutet, dass Unity nicht nur einzelne Grasbüschel irgendwo platziert, sondern immer mehrere gemeinsam erzeugt. In diesen Objektansammlungen bzw. *Clustern* besitzen die Gras-Objekte, die sich im Mittelpunkt befinden, die hinterlegten Max-Werte (also *Max Width* und *Max Height*). Nach außen hin nehmen die Objekte dann immer kleinere Werte an. Die äußersten besitzen schließlich die hinterlegten Min-Werte (*Min Width* und *Min Height*). Je höher der Wert *Noise Spread* nun ist, desto mehr gibt es von solchen *Clustern* mit eigenen Mittelpunkten.
- **Healthy Color** legt die Farbe fest, die die Grastexturen im Zentrum eines *Clusters* besitzen.
- **Dry Color** legt die Farbe fest, die die Grastexturen am äußeren Rand eines *Clusters* besitzen.
- **Billboard** legt fest, ob die Frontseite der Grastexturen immer zur Kamera ausgerichtet ist oder ob sich diese nicht an der Kamera ausrichten soll.

### 12.5.3 Quelldaten nachladen

Sowohl beim *Place Tree-Tool* wie auch beim *Paint Details-Werkzeug* gibt es einen **Refresh**-Button. Dieser dient dem Nachladen der Quelldateien der aktuell genutzten Assets.

Wenn Sie zum Beispiel während der Arbeit in Unity merken, dass die Grastextur nicht ganz passt, dann können Sie die Originaldatei parallel in einer Grafiksoftware wie *GIMP* oder *Photoshop* laden und dort verändern. Wenn Sie diese nun dort speichern, können Sie die veränderte Datei mit *Refresh* nachladen und so das gesamte Projekt aktualisieren. Sie müssen also nicht alle zugefügten Grastexturen vom *Terrain* erst löschen und dann wieder neu hinzufügen, was ein enormer Vorteil ist.

## ■ 12.6 Terrain-Einstellungen

Als letzter Punkt im *Terrain*-Menü finden Sie die *Terrain Settings*, worüber Sie etliche Einstellungsmöglichkeiten für das *Terrain* und die dort platzierten Objekte finden. Für die ersten Tests werden Sie hier nicht unbedingt Eingriffe vornehmen müssen. Wenn Sie aber später Optimierungen vornehmen, werden Sie hier viele Möglichkeiten finden, um das *Terrain* entweder performanter oder stattdessen ansehnlicher zu gestalten.

## 12.6.1 Base Terrain

Der Bereich *Base Terrain* fasst allgemeine Eigenschaften des Terrains zusammen.

- **Pixel Error** steuert die Menge der zulässigen Fehler bei der *Terrain*-Darstellung. Dies ist vor allem ein Parameter, der sich mit der Darstellung von weit weg liegenden Objekten beschäftigt. Je höher dieser Wert ist, desto geringer wird die Darstellungsdichte bei weiter weg liegenden Objekten, vergleichbar mit dem *Detail Density*-Parameter.
- **Base Map Dist.** definiert die Grenze, bis wo die normale *Splat Map* genutzt wird, um Texturen darzustellen, und ab wann die *Base Texture* genommen werden soll.
- **Cast Shadows** legt fest, ob das *Terrain* Schatten darstellen soll.
- **Material** gibt Ihnen die Möglichkeit, ein Grundmaterial zuzuweisen, das alle anderen Texturen noch einmal überlagert.
- **Physics Material** legt das physikalische Verhalten des *Terrains* fest.

## 12.6.2 Resolution

Im Bereich *Resolution* finden Sie Grundeinstellungen für das *Terrain*, die sich auf die Auflösungen und Maße der verschiedenen Elemente des *Terrains* beziehen. Während die ersten eher Grundeinstellungen sind, finden Sie am Ende eher welche, die sich an erfahrenere Entwickler richten, um die Performance noch etwas zu verbessern.

- **Terrain Width** legt die Breite auf der X-Achse fest.
- **Terrain Length** legt die Länge auf der Z-Achse fest.
- **Terrain Height** legt die maximale Höhe des Terrains auf der Z-Achse fest.
- **Heightmap Resolution** legt die Auflösung der *Terrain-Heightmap* fest. Die *Heightmap* ist eine quadratische Textur mit einer Breite und Höhe dieses Wertes. Dabei spielt es keine Rolle, welche Maße das Terrain selber hat.
- **Detail Resolution** legt die Auflösung der *Detail-Map* fest, die alle Detail-Objekte steuert. Je höher der Wert, desto detaillierter lassen sich Objekte platzieren, was allerdings auch mehr Performance kostet.
- **Control Texture Resolution** legt die Auflösung der Karte fest, auf der die Texturen gezeichnet werden (*Splat Map* genannt).
- **Base Texture Resolution** legt die Auflösung einer sogenannten *Base Texture* fest. Diese wird von Unity anstelle der oben beschriebenen *Splat Map* genutzt, wenn sich der Betrachter in einer weiten Entfernung zum *Terrain* befindet. Diese kann im Bereich *Base Terrain* definiert werden.

### 12.6.3 Tree & Details Objects

Hier finden Sie Einstellungsmöglichkeiten, die sich auf die Objekte beziehen, die Sie mithilfe der *Terrain-Tools* auf dem *Terrain* platziert haben.

- **Draw** legt fest, ob die Objekte dargestellt werden sollen.
- **Detail Distance** bestimmt, ab wann *Detail*-Objekte gerendert werden.
- **Detail Density** bestimmt, wie viele der *Detail*-Objekte tatsächlich dargestellt werden sollen. Die Angabe 1 bedeutet 100 % und 0 bedeutet 0 % aller Objekte werden gerendert.
- **Tree Distance** legt fest, ab welcher Entfernung Bäume dargestellt werden.
- **Billboard Start** bestimmt, ab welcher Entfernung Bäume und *Detail*-Objekte nicht mehr als *Mesh*, sondern in einer performancesparenden Flächendarstellung (*Billboard*-Darstellung) zu sehen sind.
- **Fade Length** legt ein zusätzliches Entfernungsoffset fest, in dem von der *Billboard*-Darstellung in die *Mesh*-Darstellung übergegangen wird. Bei 0 wird beim Erreichen der *Billboard Start*-Grenze direkt in die *Mesh*-Darstellung gewechselt, was häufig etwas ruckelig wirkt. Deshalb können Sie hier z.B. 5 angeben, sodass bereits fünf Einheiten (Meter) vorher angefangen wird, in die *Mesh*-Darstellung überzugehen.
- **Max Mesh Trees** definiert, wie viele Baum-Objekte als *Mesh* maximal dargestellt werden.

### 12.6.4 Wind Settings

Jedes *Terrain* liefert ein integriertes Tool mit, das Wind-Effekte erzeugen kann. Im Gegensatz zu den *Wind Zones* (siehe Kapitel „Wind Zones“), die die *Tree*-Objekte animieren, hat diese Simulation lediglich auf Graselemente Einfluss, sprich *Detail*-Objekte, die über den Button **Add Grass Texture** zugefügt wurden.

- **Speed** legt die Geschwindigkeit des Windes fest und damit auch, wie schnell das Gras hin und her schwingt und die Farbe sich ändert.
- **Bending** bestimmt, wie stark die Grastextur durch den Wind gedehnt wird.
- **Grass Tint** legt eine Farbe fest, mit der die *Grass*-Texturen in bestimmten Abständen zusätzlich eingefärbt werden. Im Zusammenspiel mit dem folgenden Parameter *Size* können so interessante Effekte erzielt werden. Zum Beispiel kann bei größeren Grasflächen auf diese Weise der Eindruck erweckt werden, dass die Gräser nicht nur hin und her schwingen, sondern sich wie Wellen bewegen.
- **Size** legt fest, wie groß der Anteil der Grasfläche ist, der durch den Wind beeinflusst wird. Im Zusammenspiel mit *Grass Tint* bedeutet dies, dass bei 0 alle Graselemente gleichzeitig eingefärbt werden und auch gleichzeitig dessen Einfärbung wieder abklingt. Für das obige Beispiel bedeutet dies keinen Welleneffekt. Der Wert 1 bedeutet wiederum einen besonders starken „Welleneindruck“.

## 12.6.5 Zur Laufzeit Terrain-Eigenschaften verändern

Wie auch in allen anderen Bereichen haben Sie natürlich auch hier wieder die Möglichkeit, auf die bereits vorgestellten Parameter per Code zuzugreifen und diese zur Laufzeit zu verändern. Normalerweise sollte dies nicht alltäglich sein. Möchten Sie aber Ihrem Spieler die Möglichkeit anbieten, die Qualität des Spiels über ein Menü zu regeln, könnten Sie hier verschiedene Parameter des *Terrains* steuern. Gerade *Terrains* können durch das Ändern bestimmter Einstellungen viel Einfluss auf die Performance des Spiels nehmen, was ja der wichtigste Effekt einer Qualitätseinstellung sein sollte.



### Quality Settings

In Unity können über *Edit/Project Settings/Quality* verschiedene Qualitäts-einstellungen eingestellt werden, die dann bestimmte Einstellungen beim *Renderen* und der Schattendarstellung vornehmen. Diese können Sie zum einen natürlich selber anpassen und ändern. Zum anderen können Sie aber auch während des Spiels zwischen diesen Einstellungen wechseln und so die Spiel-qualität verändern. Hierfür stellt Unity die Klasse *QualitySettings* bereit, die z. B. die Methoden *SetQualityLevel* und *GetQualityLevel* besitzt.

Das folgende Beispiel nutzt die statische Variable *activeTerrain* der *Terrain*-Klasse, mit der Sie direkt auf das Hauptterrain der Szene zugreifen. In der Methode *SetQuality* wird dann zunächst die Qualitätsstufe der Klasse *QualitySettings* und danach werden einige Parameter des Hauptterrains gesetzt. Beachten Sie, dass dies nur ein Beispiel ist und die Parameter nur bei zwei übergebenen Levels gesetzt werden.

**Listing 12.1** Qualität des Terrains und in den *QualitySettings* setzen

```
using UnityEngine;
using System.Collections;

public class QualityController : MonoBehaviour {

    // Use this for initialization
    void Start () {
        SetQuality(5);
    }

    public void SetQuality(int level)
    {
        QualitySettings.SetQualityLevel(level);

        switch(level)
        {
            case 0: //andere Einstellungen...
            case 1: //andere Einstellungen...
            case 2:
                Terrain.activeTerrain.detailObjectDensity = 0.8F;
                Terrain.activeTerrain.detailObjectDistance = 80;
                Terrain.activeTerrain.treeBillboardDistance = 30;
                break;
        }
    }
}
```

```

        case 3: //andere Einstellungen...
        case 4: //andere Einstellungen...
        case 5: //andere Einstellungen...
            Terrain.activeTerrain.detailObjectDensity = 1;
            Terrain.activeTerrain.detailObjectDistance = 80;
            Terrain.activeTerrain.treeBillboardDistance = 50;
            break;
        }
    }
}

```

## ■ 12.7 Der Weg zum perfekten Terrain

Auch wenn es jetzt kein Patentrezept gibt, an dem man sich entlanghangelt und durch das man dann ein perfektes Terrain erhält, so gibt es doch einige grundlegenden Dinge, die man bedenken sollte.

- Beginnen Sie immer zuerst mit den groben Strukturen und verfeinern Sie Ihre Landschaft dann langsam.
- Texturierung und Oberflächenbeschaffenheit (Anhöhen, Täler, Klippen etc.) gehen meistens Hand in Hand miteinander. An steilen Abhängen sind z.B. meist Fels- oder Steintexturen zu finden. Und auch Trampelpfade zeichnen sich nicht nur durch besondere Texturen aus (z.B. Sand), sondern auch durch flache Furchen im Boden.
- Nehmen Sie sich die Natur als Vorbild. Dort sehen Sie am besten, wie Dinge zusammenhängen.
- Sorgen Sie für Abwechslung. So können auch Felsen stellenweise mit Gras bewachsen sein und auch eine Wüste besteht nicht nur überall aus dem gleichen Sand. Nutzen Sie zusätzliche Texturen, die Sie z.B. teils transparent und mit speziellen Pinseln an verschiedenen Stellen hier und da auf die eigentliche Haupttextur legen.
- Setzen Sie Bäume und die Detailobjekte (Gräser) mit Bedacht ein und nur dort, wo sie einen spielerischen Mehrwert bieten. Ansonsten kosten diese nur unnötig Performance.
- Um große Waldstücke im Hintergrund darzustellen, können Sie diese z.B. auch grafisch in die *Skybox* integrieren. Sie können aber auch zusammenhängende Baumgruppen in einer 3D-Modelling-Software designen und diese dann in Ihrer Szene platzieren.

Am Ende gilt aber beim Erstellen von Terrains das Gleiche wie in eigentlich allen Bereichen: Übung macht den Meister. Um interessante und auch nicht allzu leistungshungrige Terrains zu gestalten, braucht es doch einiges an Erfahrung.



Auf der DVD finden Sie ein Video, das Ihnen das Nutzen der Terrain-Tools in der Praxis zeigt. Hierbei wird auch die *Wind Zone*-Komponente (siehe Kapitel 13) und der *Asset Store* mit seinen vielen kostenlosen Terrain-Assets vorgestellt.

## ■ 12.8 Gewässer

Wenn Sie Gewässer Ihren Landschaften hinzufügen möchten, bieten sich die *Water-Prefabs* aus den *Standard Assets* von Unity an (*Assets/Import Package/*). Dort finden Sie sowohl ein Package für Unity Basic und eines für Unity Pro. Beide Pakete kommen mit zwei fertigen *Prefabs*, jeweils eines für die Tagesansicht und eines für die Darstellung bei Nacht. Der Hauptunterschied dieser *Packages* liegt dabei vor allem in der Fähigkeit, Spiegelungen auf der Wasseroberfläche zu generieren. Dies unterstützt nur die Pro-Fassung, da hierfür *RenderTextures* eingesetzt werden, die nur von der Pro-Version unterstützt werden.

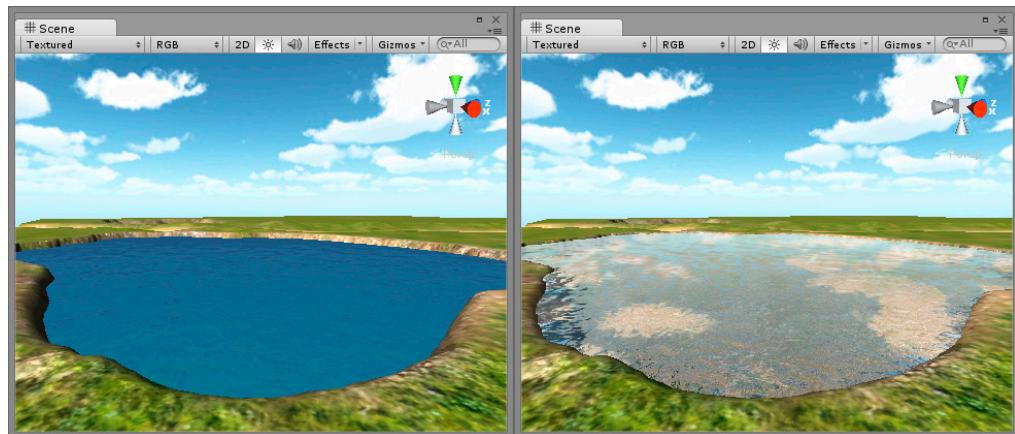


Bild 12.17 Prefabs „Daylight Simple Water“ und „Daylight Water“ (Pro)

Das Platzieren dieser *Prefabs* geht wie immer einfach über Drag & Drop, die Sie dann anschließend auf die benötigten Größen skalieren können. Dabei wird normalerweise zuvor das gesamte *Terrain* angehoben und dort, wo das Wasser sein soll, werden Vertiefungen gemacht. Auf diese Weise können Sie die eigentlich runden *Prefabs* so platzieren, dass sie an den Seiten in den ansteigenden *Terrain*-Oberflächen verschwinden und die eigentlichen Kanten des Wasser-*Prefabs* nicht zu sehen sind (siehe Bild 12.17).

Lizenziert für gregwyss@hotmail.com.

© 2014 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

# 13

## Wind Zones

Möchten Sie atmosphärische Outdoor-Szenerien erstellen oder industrielle Maschinen wie Turbinen und Hubschrauber in Ihrer Spielwelt etwas realistischer wirken lassen, dann brauchen Sie eine gute Möglichkeit, Wind zu simulieren.



**Bild 13.1** Windeinfluss auf Bäume eines Terrains

Hierfür gibt es in Unity die sogenannte *Wind Zone*-Komponente, die ähnlich wie ein Ventilator funktioniert. Auch wenn diese *Wind Zones* nicht auf alle *GameObjects* Einfluss nehmen können, so beeinflussen sie doch wichtige windsensible Elemente wie Bäume bzw. die Tree-Objekte eines *Terrains* und Partikel des *Shuriken Particle Systems* (siehe *External Forces*).

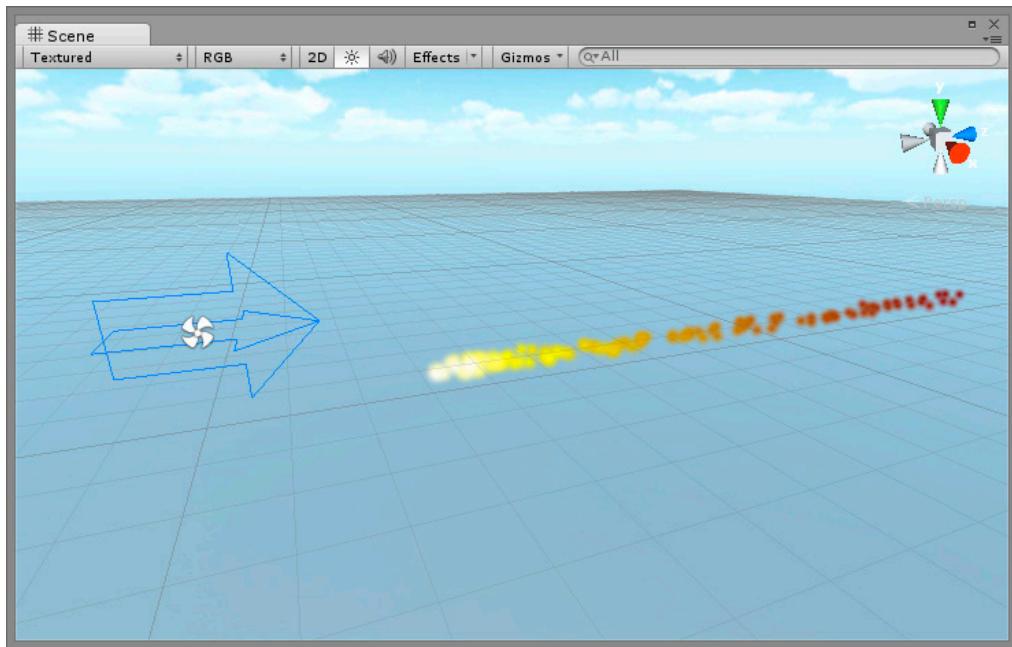
*Wind Zones* können Sie zum einen als fertiges *Wind Zone*-Objekt über das *GameObject*-Menü hinzufügen (*GameObject/Create General/Wind Zone*), zum anderen haben Sie aber natürlich auch die Möglichkeit, die einzelne *Wind Zone*-Komponente einem *GameObject* anzuhängen. Richtig gut wirkt eine *Wind Zone* vor allem in Kombination mit einer  *AudioSource*, die auch den passenden Klang dazu ausgibt.

## ■ 13.1 Spherical vs. Directional

Ein *Wind Zone*-Objekt bzw. die Komponente bietet Ihnen zwei unterschiedliche Modi an: *Spherical* und *Directional*. Diese haben folgende Auswirkungen:

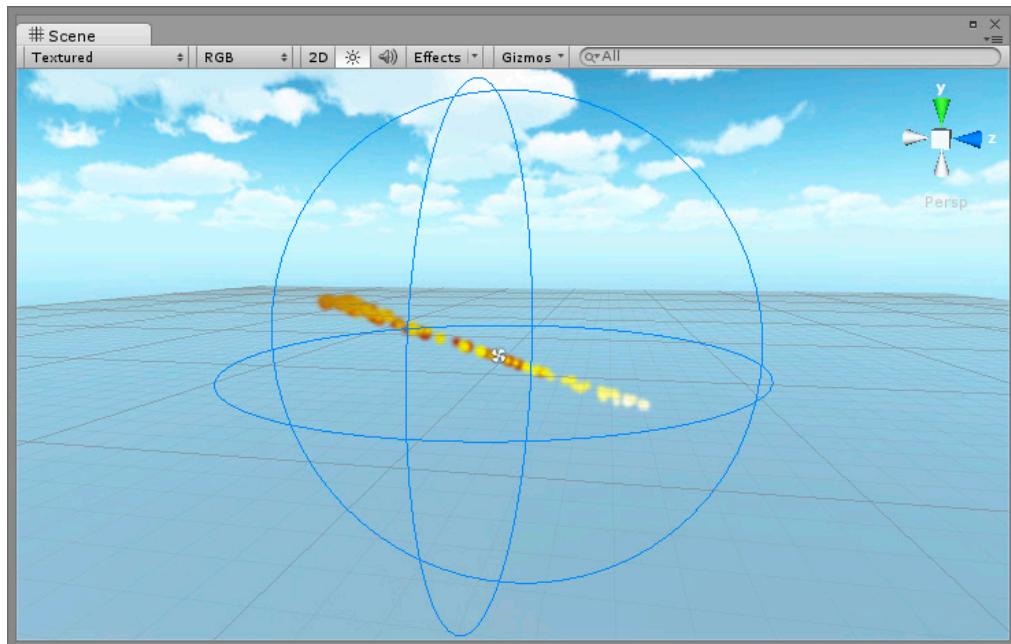
- **Directional** fügt der gesamten Szene eine gerichtete Windkraft zu. Die Kraft ist hierbei überall gleich stark und zeigt auch immer in die gleiche Richtung.
- **Spherical** beeinflusst nur einen festgelegten Bereich. Hierbei dreht sich die Kraft ähnlich wie in einem Strudel. Die Kraft nimmt hierbei mit Abstand zum Zentrum ab.

Das Bild 13.2 zeigt den Einfluss eines gerichteten Windes („Directional“) auf ein Partikelsystem, dessen Partikel sich zur besseren Veranschaulichung von Weiß über Gelb zu Dunkelrot verändern. Das Partikelsystem an sich fügt den Partikeln keine Kraft zu, sodass diese nur durch die *Wind Zone* bewegt werden.



**Bild 13.2** Directional-Beispiel

Das Bild 13.3 zeigt das gleiche Partikelsystem, das sich dieses Mal aber im Einflussbereich einer *Wind Zone* vom Typ „Spherical“ befindet. Die Partikel werden dabei zum Zentrum hin gezogen. Dabei fliegen diese etwas zu weit und werden dann durch den Windstrudel wieder zurückgezogen.



**Bild 13.3** Spherical-Beispiel

## ■ 13.2 Wind Zone – Eigenschaften

Eine Wind Zone besitzt verschiedene Parameter, um das Verhalten der Bäume und Partikel sehr fein zu definieren. Im Folgenden möchte ich die Eigenschaften kurz vorstellen. Auch wenn sich diese gegenseitig beeinflussen, will ich versuchen, die Parameter anhand des Verhaltens der *Tree*-Objekte etwas zu verdeutlichen:

- **Mode** legt fest, ob *Spherical* oder *Directional* genutzt wird.
- **Radius** legt den Radius fest, wenn der *Spherical*-Modus gewählt wurde.
- **Wind Main** legt die Hauptkraft vom Wind fest. Bei Bäumen beeinflusst dies, wie stark sich diese im Gesamten in die Windrichtung biegen.
- **Wind Turbulence** legt zusätzliche Turbulenzen fest, die für das eigentliche Schwingen der *Tree*-Objekte verantwortlich sind. Der Wert wirkt sich hauptsächlich auf die Äste in Auf- und Abbewegungen aus.
- **Wind Pulse Magnitude** bestimmt, wie stark der Wind wechselt. Dies wirkt sich auf das gesamte *Tree*-Objekt aus, wie der Stamm sich biegt und wie stark dessen Äste hin und her schwingen.
- **Wind Pulse Frequency** beschreibt die Schnelligkeit der Windwechsel. Der Wert beschreibt, wie schnell sich die *Tree*-Objekte im Wind hin und her wiegen.

## ■ 13.3 Frische Brise

Es gibt viele Möglichkeiten, eine *Wind Zone*-Komponente zu konfigurieren. Um zum Beispiel für eine Outdoor-Szene den Effekt einer frischen Brise zu erzeugen, die Äste und Bäume bewegen lässt, können Sie beispielweise die folgenden Einstellungen nutzen:

- *Mode: Directional*
- *Wind Main: 0,5*
- *Wind Turbulence: 0,1*
- *Wind Pulse Magnitude: 0,5*
- *Wind Pulse Frequency: 0,25*

Die Platzierung dieses *Wind Zone*-Objektes spielt hierbei keine Rolle. Lediglich die Rotation ist wichtig, was vergleichbar mit dem Verhalten eines *Directional Light* ist.

## ■ 13.4 Turbine

Bei einer Turbine kommt im Gegensatz zur Brise der *Spherical*-Modus zum Einsatz. Turbinen finden Sie oft in Industriehallen, wo meist auch Partikelsysteme für Dämpfe etc. eingesetzt werden. Um diese stärker zu beeinflussen, können Sie hier mit einem höheren *Wind Main*-Wert arbeiten. Allerdings sollten Sie vermeiden, in diesem Bereich *Tree*-Objekte zu platzieren, da diese durch solch hohen Werte doch etwas unrealistisch verformt werden.

- *Mode: Spherical*
- *Radius: 5*
- *Wind Main: 5*
- *Wind Turbulence: 0*
- *Wind Pulse Magnitude: 2*
- *Wind Pulse Frequency: 0*

Auch dies ist natürlich nur ein Beispiel, dessen Werte Sie Ihren Wünschen und der jeweiligen Szene entsprechend anpassen sollten. Der Radius sollte dabei dem der Turbine entsprechen bzw. etwas größer sein. Das *Wind Zone*-Objekt sollte dann im Zentrum der Turbine platziert werden, um so einen passenden Effekt zu erzielen.

# 14

## GUI

Der Aufbau der grafischen Benutzeroberfläche (*Graphical User Interface*, kurz *GUI*) eines Spiels ist ein nicht zu unterschätzendes Thema bei der Spieleentwicklung. Ist diese unübersichtlich oder nicht intuitiv gestaltet, kann das den Spielfluss und damit den Spielspaß schon negativ beeinflussen, bevor das eigentliche Spiel überhaupt gestartet wurde. Im Normalfall beginnt eigentlich jedes Spiel mit einem Startmenü, über das man Einstellungen vornehmen kann oder Informationen zum Spiel erfährt. Erst danach startet das Spiel, wo natürlich ebenfalls eine *GUI* vorhanden ist und gestaltet werden will.

Im Gegensatz zur normalen 3D-Welt spielt die *GUI* hier eine Sonderrolle. Diese stellt zweidimensionale Objekte dar, die über das normale Kamerabild nochmals hinüber gelegt werden. Historisch bedingt bietet Unity Ihnen hierfür unterschiedliche Möglichkeiten an, *GUI*-Objekte in Ihr Spiel zu integrieren. So gibt es die sogenannten *UIElements*, mit denen Sie lediglich Text und Grafiken anzeigen können, ein *GUI*-Scripting-System namens *UnityGUI*, mit dem Sie per Code die verschiedenen Controls definieren, sowie das neue *uGUI*-System, das per Drag & Drop *GUI*-Objekte positionieren lässt und mit der Version 4.6 erscheint.

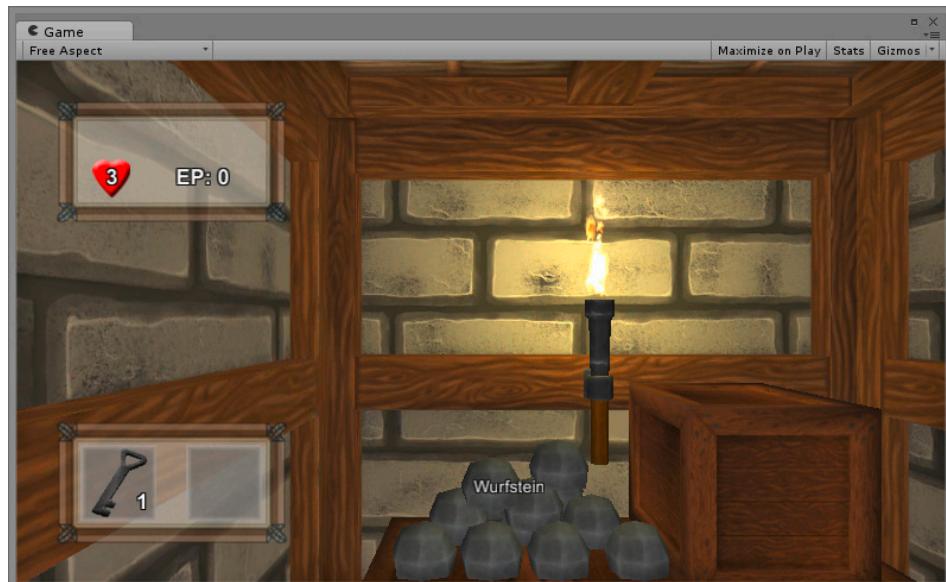


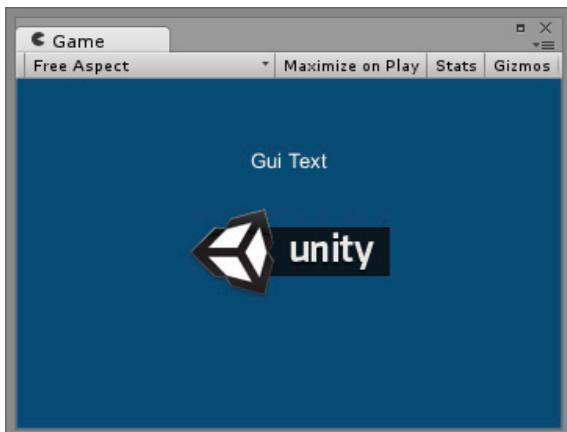
Bild 14.1 GUI-Beispiel

## ■ 14.1 GUIElements

Zum Anzeigen einfacher Bild- und Textinformationen können Sie die sogenannten *GUIElements* nutzen, die über das *GameObject*-Menü zur Verfügung gestellt werden. Wenn Sie nun *GUIElements* der Szene zufügen, werden diese zwar in der *Game View* angezeigt, sie werden aber nicht in der *Scene View* dargestellt. Dementsprechend können sie dort auch nicht bearbeitet werden, sodass sie nur über den *Inspector* konfiguriert werden können.

Für die Darstellung dieser *GUIElements* benötigt die Kamera eine *GUILayout*-Komponente, die aber schon per Default jedem *Camera-GameObject* zugefügt wird und auch die „Main Camera“ von Anfang an besitzt.

Zum Darstellen von Textinformationen nutzen Sie hier das *GUI Text-GameObject*, das Sie über **GameObject/Create Other/** bzw. **GameObject/Create General/** (ab Version 4.6) dem Spiel hinzufügen. Der Name röhrt von der Komponente *GUIText*, die den Kern dieses Objektes darstellt. Analog hierzu gibt es auch das *GUI Texture-GameObject* mit der *GUITexture*-Komponente. Mit diesem können Sie Bilder in der GUI darstellen.



**Bild 14.2**  
GUIText und GUITexture

### 14.1.1 GUIElements ausrichten

Wenn Sie die *GUIElements* der Szene zufügen, wird der Inhalt zunächst zentral in der Mitte des Bildschirms angezeigt. Diese generelle Ausrichtung steuern Sie über das Transform des *GameObjects*, das die Ausrichtung in einem prozentualen Wert von 0 bis 1 des Bildschirms angibt, dem sogenannten *Viewport Space*.

- X steuert die horizontale Ausrichtung des Inhalts. 0 bedeutet linksbündig, 1 bedeutet rechtsbündig. Mit 0.5 richten Sie den Inhalt mittig aus.
- Y steuert die vertikale Ausrichtung des Inhalts. 0 bedeutet unten, 1 bedeutet oben ausgerichtet. Mit 0.5 richten Sie den Inhalt mittig aus.
- Z steuert die Tiefenebene des Inhalts. Je niedriger der Wert ist, desto weiter ist der Inhalt vom Betrachter entfernt. Möchten Sie beispielsweise ein Hintergrundbild nutzen und da-

über einen Text legen, muss das *GUIText*-Objekt einen höheren Z-Wert besitzen als das *GUITexture*-Objekt mit dem Hintergrundbild.

Die genaue Positionierung der Inhalte nehmen Sie anschließend über die Komponenten-Eigenschaften *Pixel Offset* und *Pixel Inset* der *GUIElemente* vor, mit denen Sie pixelgenau nachjustieren können.

### 14.1.2 GUIText positionieren

Während Sie die Ausrichtung über das *Transform* vorgeben, legen Sie die genaue Position über die Eigenschaften der *GUIText*-Komponente fest.

- **Anchor** legt den Punkt fest, an dem sich der Text am *Transform* orientieren soll. Meistens macht es Sinn, die *Transform*-Ausrichtung auf diesen Parameter zu übertragen. Das heißt, wenn Sie z. B. das *Transform* auf (0, 0, 0) gesetzt haben, sollten Sie auch den *Anchor*-Wert auf *lower left* legen. Bei (1, 0, 0) legen Sie den Wert auf *lower right* usw.
- **Alignment** ist bei mehrzeiligen Texten wichtig und legt die Ausrichtung der Zeilen untereinander fest.
- **Pixel Offset** legt einen Abstand zu der eigentlichen Ausrichtung fest. Möchten Sie beispielsweise einen Text unten links mit einem kleinen Freiraum definieren, kann dieser z. B. (20,20) betragen. Wenn Sie den Text rechts oben positionieren, muss der Abstand negativ definiert werden, also (-20, -20).



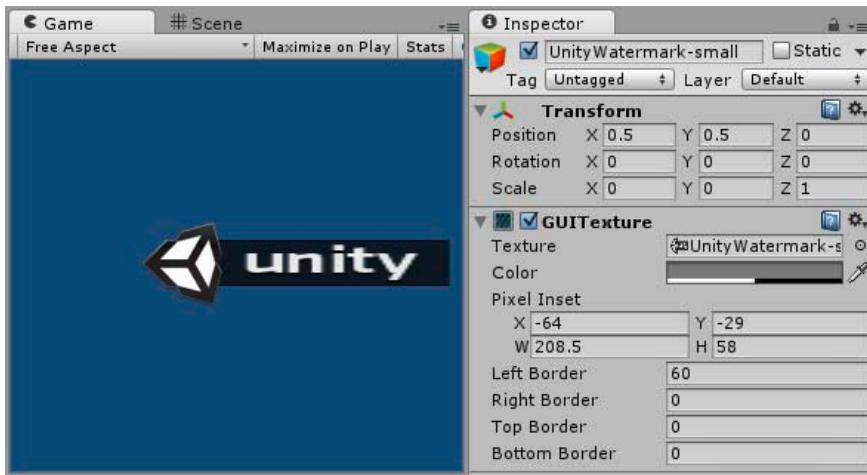
#### GUIText verschwunden?

Sollte der Text einmal nicht zu sehen sein, liegt es häufig an der falschen Einstellung des *Anchor*-Wertes und/oder dem zugewiesenen *Pixel Offset*.

### 14.1.3 GUITexture skalieren und positionieren

Während Sie die Grundausrichtung über das *Transform* bereits vorgenommen haben, können Sie über die *Component*-Eigenschaften von *GUITexture* die Feineinstellungen vornehmen.

- **Pixel Inset** definiert sowohl die Position wie auch die Größe der Grafik. X und Y definieren dabei die Verschiebung der Textur von der vorgegebenen *Transform*-Position von unten links. W und H legen die Größe der Grafik fest. Möchten Sie z. B. ein Bild mit der Größe 128 × 64 unten links positionieren, legen Sie das *Transform* auf (0,0,0), X und Y auf 0, W auf 128 und H auf 64.
- **Border** legt einen Rand fest, der durch das Skalieren der Grafik über W und H von *Pixel Inset* keine Auswirkung hat. Weisen Sie W und H andere Werte zu, als das Originalbild eigentlich hat, wird logischerweise das Bild gestreckt bzw. gestaucht. Bei einem Bild mit abgerundeten Ecken können Sie auf diese Weise dafür sorgen, dass nur ein bestimmter Bereich in der Mitte von diesem Strecken beeinflusst wird und der äußere Rand mit den Abrundungen eben nicht.



**Bild 14.3** Gestreckte Textur mit einem nicht streckbaren Bereich (Left Border)



#### Texture Type beachten:

Bei Texturen, die Sie in der GUI anzeigen möchten, sollten Sie darauf achten, dass Sie in den *Import Settings* der Textur den *Texture Type* „GUI“ wählen. Vor allem dann, wenn Sie die Textur kleiner darstellen möchten, als sie eigentlich ist, macht sich diese Einstellung bemerkbar.

### 14.1.4 Interaktivität

Mithilfe der *UIElements* können Sie nicht nur Informationen anzeigen, Sie können auch dank verschiedener Event-Methoden, die  anbietet, Interaktionen ausführen. Diese werden bei Objekten mit *Collidern* wie auch bei *UIElements* ausgeführt. Hier eine kleine Auswahl solcher Methoden:

- **OnMouseDown** wird ausgelöst, wenn sich der Mauszeiger auf/über dem Objekt befindet und ein Mausbutton heruntergedrückt wird.
- **OnMouseUp** wird ausgelöst, wenn sich der Mauszeiger auf/über einem Objekt befindet und ein Mausbutton hochgenommen wird. Dies wird auch dann ausgelöst, wenn der Mausbutton woanders heruntergedrückt, dann auf dieses Objekt bewegt und dann hochgenommen wird.
- **OnMouseUpAsButton** wird ausgelöst, wenn sich der Mauszeiger auf/über demselben Objekt befindet, wo der Button auch *OnMouseDown* ausgelöst hat und nun der Mausbutton hochgenommen wird.
- **OnMouseEnter** wird ausgelöst, wenn der Mauszeiger auf ein Objekt bewegt wird.
- **OnMouseOver** wird ausgelöst, solange sich der Mauszeiger über dem Objekt befindet.
- **OnMouseExit** wird ausgelöst, wenn der Mauszeiger vom Objekt weggenommen wird.

Mithilfe dieser Funktionen können Sie ein kleines Skript programmieren, das Sie einem *UIElement* zufügen, und dadurch ein Button-Verhalten erzeugen. Das folgende Beispiel-skript ist für ein *GUITexture-GameObject* gedacht und erzeugt einen Hover-Effekt, der die Textur wechselt, sobald Sie sich mit dem Mauszeiger über diesem Objekt befinden. Wenn Sie dann noch einen Klick auf dieses machen, wird in die Konsole „Click“ geschrieben. Letzteres können Sie natürlich durch ein beliebiges Verhalten, wie z.B. das Starten einer neuen Szene, ersetzen.

**Listing 14.1** Einfacher Button

```
using UnityEngine;
using System.Collections;
public class SimpleButton : MonoBehaviour {
    public Texture2D normal;
    public Texture2D hover;
    void Start () {
        guiTexture.texture = normal;
    }

    void OnMouseDown () {
        Debug.Log("Click");
    }

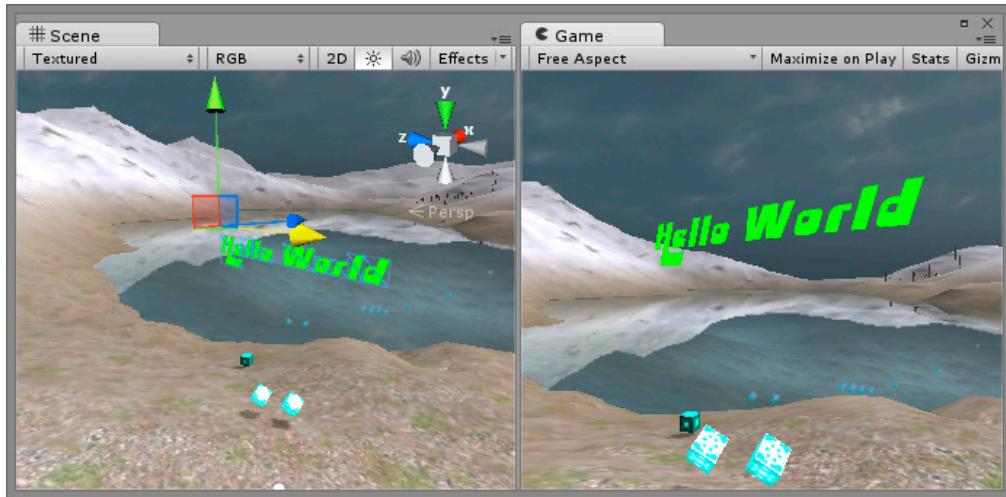
    void OnMouseEnter() {
        guiTexture.texture = hover;
    }

    void OnMouseExit() {
        guiTexture.texture = normal;
    }
}
```

Im Kapitel „Maus, Tastatur, Touch“ finden Sie im Abschnitt „Touch-Controls“ ein weiteres Beispiel, wie Sie Touch-Eingaben auf einem *GUITexture*-Objekt auswerten können, um dieses als virtuellen Controller zu nutzen.

## ■ 14.2 3DText

Möchten Sie Texte direkt in der Spielwelt darstellen, z.B. über einem *GameObject*, bietet sich das *3DText-GameObject* an. Dieses stellt den Text mithilfe der *Text Mesh*-Komponente in einem beliebigen Schriftfont in der Szene dar. Beachten Sie hierbei, dass der Text an sich aber noch zweidimensional, also ohne Tiefe ist. Auch wenn sich der Text also in der dritten Dimension befindet und von der Seite betrachtet oder sogar umwandert werden kann, besitzt dieser selbst keine Tiefeninformationen.



**Bild 14.4** Ein 3DText-Objekt in einer Szene



#### Verschwommene 3DText-Darstellungen beheben

3DText-Elemente können, je nach Schriftgröße, unsauber aussehen. Dies können Sie beheben, indem Sie die Skalierungen (Scale) heruntersetzen, z. B. auf 0.1, und im Gegenzug die *Font Size* anheben.

## ■ 14.3 OnGUI-Programmierung

Unity besitzt ein recht umfangreiches *GUI Scripting System* namens *UnityGUI*, mit dem Sie per Code die GUI-Objekte definieren und abfragen können. Sie können Subfenster erstellen, scrollbare Bereiche definieren und Controls wie Slider oder Toggles nutzen.

Die Vorgehensweise hierbei ist wie folgt: Die Klassen *GUI* und *GUILayout* besitzen für jedes Control eine eigene statische Methode, mit der Sie dieses in der *OnGUI*-Methode definieren. Die Methode *OnGUI* wird dabei ereignisgesteuert aufgerufen, sodass die Controls z. B. bei jedem Mausklick oder Tastendruck neu gezeichnet werden.

```
void OnGUI() {
    GUILayout.Label("GUILayout positioniert die Objekte automatisch.");
    GUI.Label(new Rect(100,50,400,30), "Die GUI-Klasse überlässt " +
        "dem Programmierer das Positionieren.");
}
```

### 14.3.1 GUI

Mithilfe der GUI-Klasse beschreiben Sie, welche Labels, Buttons oder auch Subfenster an welcher Stelle auf dem Bildschirm erscheinen sollen. Dabei geben Sie jedem Control nicht nur den Inhalt mit, sondern mit einer *Rect*-Variablen auch immer deren Größe und Position.

Damit nun nicht in jedem *OnGUI*-Aufruf immer wieder die *Rect*-Variable aufs Neue erstellt werden muss, empfiehlt es sich meistens, diese einmal zu definieren und dann immer wieder zu verwenden.

**Listing 14.2** Erstellen eines Buttons mit der Klasse GUI

```
private Rect rect;
void Start() {
    rect = new Rect(50, 50, 200, 30);
}
void OnGUI() {
    if (GUI.Button(rect, "Start"))
        Application.LoadLevel(1);
}
```

Im Folgenden möchte ich Ihnen einige Beispiele der zur Verfügung stehenden Controls zeigen. Für einen umfassenden Überblick empfehle ich Ihnen die Scripting Reference sowie das Online-Manual, das Unity mitliefert. Dort finden Sie einen ausführlichen „GUI Scripting Guide“, in dem alle Controls detailliert vorgestellt werden.

#### 14.3.1.1 Label

Labels der GUI-Klasse dienen der Anzeige von nicht interaktiven Inhalten. Sie können ihr sowohl einen Text oder eine Grafik übergeben, der/die dann an der angebenden Position gezeichnet wird.

**Listing 14.3** Erstellen eines Labels mit der GUI-Klasse

```
void OnGUI()
    GUI.Label(new Rect(100,50,400,30), "Dies ist ein Label.");
}
```

#### 14.3.1.2 Button und RepeatButton

Wie Sie schon dem Listing 14.2 entnehmen konnten, werden Buttons normalerweise so definiert, dass sie als Bedingung innerhalb einer if-Anweisung stehen. In dem Moment, wo dieser dann heruntergedrückt wird, wird auch der Code ausgeführt.

**Listing 14.4** GUI-Button mit Grafik

```
public Texture2D pic;
private Rect rect;
void Start() {
    rect = new Rect(50, 50, 200, 30);
}
void OnGUI() {
```

```

    if (GUI.Button(rect, pic))
        Application.LoadLevel(1);
}

```

Dabei können Sie einem Button sowohl Textinhalte als auch eine `Texture2D`-Variable übergeben, die dann auf dem Button dargestellt wird (wie in Listing 14.4 zu sehen).

Möchten Sie allerdings den Code nicht nur einmal, sondern stetig ausführen, eben so lange wie der Button gedrückt wird, müssen Sie stattdessen den `RepeatButton` nutzen. Dieser lässt sich genauso konfigurieren wie der herkömmliche Button, nur dass dieser den Code in jedem Aufruf von `OnGUI` ausführt.

#### **Listing 14.5** Beispiel für einen RepeatButton

```

public int mySize = 0;
private Rect rect;
void Start()
{
    rect = new Rect(50, 50, 200, 30);
}

void OnGUI()
{
    if (GUI.RepeatButton(rect, "Size +"))
        mySize++;
}

```

##### **14.3.1.3 TextField und TextArea**

Möchten Sie dem Nutzer Eingabemöglichkeiten bieten, wo er zum Beispiel seinen Namen oder Ähnliches eingeben kann, können Sie die beiden Methoden `TextField` und `TextArea` nutzen. Während `TextField` nur einen einzeiligen Text zulässt, unterstützt `TextArea` auch mehrere Zeilen untereinander.

Beide Methoden erwarten einen Rückgabeparameter, der den neuen Textinhalt zurückgibt. Zudem benötigt die Methode eine Textvariable als Übergabeparameter mit dem Textinhalt, der angezeigt werden soll. Im Normalfall wird beiden Parametern dieselbe Variable zugewiesen, die dann entweder `public` oder `private` sein muss, um den Inhalt bis zum nächsten `OnGUI`-Aufruf nicht wieder zu vergessen. Es könnten aber auch theoretisch unterschiedliche Variablen zugewiesen werden.

Sie können zudem beiden Methoden einen zusätzlichen Parameter mitgeben, der die maximale Anzahl an Zeichen festlegt, die eingegeben werden können. Dieser Parameter ist aber nicht zwingend erforderlich und kann auch weggelassen werden.

#### **Listing 14.6** Beispiel der Methoden `TextField` und `TextArea`

```

public string singleLine = "";
public string multiLines = "";
void OnGUI()
{
    singleLine = GUI.TextField (new Rect(100, 50, 250, 30), singleLine,30);
    multiLines = GUI.TextArea (new Rect(100, 100, 250, 100), multiLines,150);
}

```

#### 14.3.1.4 Subfenster

Sie können mithilfe der Methode `Window` der GUI-Klasse auch Subfenster erstellen, die nicht nur Controls aufnehmen kann, sondern auch verschiebbar (engl. „draggable“) sind.

Die Methodik ähnelt dabei dem vorherigen `TextArea`-Control. Auch hier benötigen Sie eine *public*- oder *private*-Variable, die einmal die Position und Maße des Fensters per Parameter zuweist und zum anderen die neuen Werte über einen Rückgabeparameter zurückhält. Nur dieses Mal sind diese Parameter vom Typ `Rect`.

Das eigentliche Subfenster wird dann durch eine gesonderte Methode dargestellt, die das Fenster beschreibt und ebenfalls von der Methode `Window` aufgerufen wird.

Insgesamt benötigt die Methode vier Übergabeparameter:

- eine Fenster-ID vom Typ `int`
- die Fensterposition und -größe vom Typ `Rect`
- den Namen der Methode, die das Fenster und dessen Verhalten beschreibt
- ein Text oder eine Grafik, die im Kopf des Fensters angezeigt wird

Als Rückgabeparameter gibt die Methode dann ebenfalls einen `Rect`-Wert zurück.

**Listing 14.7** Darstellung eines Subfensters

```
public Rect windowRect = new Rect(50, 50, 220, 150);
void OnGUI()
{
    windowRect = GUI.Window(0, windowRect, DrawMyWindow,"Titel");
}

void DrawMyWindow(int windowID) {
    GUI.Label(new Rect(10, 20, 100, 20), "Hallo!");
}
```

Möchten Sie nun das Fenster verschiebbar machen, brauchen Sie innerhalb der Fenster-Methode (in Listing 14.7 wäre dies `DrawMyWindow`) lediglich noch zusätzlich die Methode `DragWindow` der GUI-Klassen aufzurufen.

**Listing 14.8** Verschiebbares Subfenster

```
void DrawMyWindow(int windowID) {
    GUI.Label(new Rect(10, 20, 100, 20), "Hallo!");
    GUI.DragWindow();
}
```

#### 14.3.2 GUILayout

Da Sie beim Platzieren der GUI-Controls mit der GUI-Klasse jedes Control separat mit einem `Rect`-Parameter platzieren müssen, kann dies bei vielen Controls schon mal etwas unübersichtlich werden. Hierfür bietet Unity mit der Klasse `GUILayout` eine Alternative an.

`GUILayout` stellt größtenteils die gleichen Methoden zur Verfügung wie die Klasse `GUI`, nur mit dem Unterschied, dass Sie hier keine `Rect`-Variable übergeben müssen. Die Größen und

Positionen der `GUILayout`-Controls werden direkt von Unity festgelegt und auch zugewiesen. Gerade beim Prototyping hat dies einen großen Vorteil, da Sie so sehr schnell Eingabe- oder Ausgabe-Controls erstellen können, ohne großartig die Formatierung beachten zu müssen.

#### **Listing 14.9** Darstellung eines Buttons mit `GUILayout`

```
void OnGUI()
{
    if (GUILayout.Button("Start"))
        Application.LoadLevel (1);
}
```

#### **14.3.2.1 Ausrichtung festlegen**

Unity ordnet die `GUILayout`-Controls per Default immer untereinander an. Sie können aber auch selber festlegen, wenn Steuerelemente nebeneinander oder untereinander platziert werden sollen. Hierfür müssen Sie Bereiche definieren, in denen die Controls entsprechend angeordnet werden sollen. Die Methoden dafür lauten:

- `BeginHorizontal`
- `EndHorizontal`
- `BeginVertical`
- `EndVertical`

Ähnlich wie bei HTML-Tags müssen Sie auch hier immer daran denken, dass jeder Bereich, der mit einer *Begin*-Methode startet, auch wieder mit einer *End*-Methode beendet werden muss. Dabei können diese Bereiche natürlich auch ineinander verschachtelt werden.

#### **Listing 14.10** Verschachtelte Anordnungen von `GUILayout`-Labels

```
GUILayout.BeginHorizontal();

GUILayout.BeginVertical();
GUILayout.Label("Spalte 1, Zeile 1");
GUILayout.Label("Spalte 1, Zeile 2");
GUILayout.EndVertical();

GUILayout.BeginVertical();
GUILayout.Label("Spalte 2, Zeile 1");
GUILayout.Label("Spalte 2, Zeile 2");
GUILayout.EndVertical();

GUILayout.EndHorizontal();
```

#### **14.3.3 GUIStyle und UISkin**

Das Aussehen dieser *GUI*- und `GUILayout`-Controls definieren Sie über *GUIStyles*. Jeder Control-Typ hat dabei einen eigenen Style, der u.a. festlegt, wie die Außenmaße dieser Controls sind, welche Fonts und Texturen diese nutzen oder auch welche Hover-Effekte sie besitzen.

Sie können auch ein *GUILayout*-Objekt direkt einem speziellen GUI-Control zuweisen, sodass diese Definition nur für dieses eine gilt. Hierfür bieten alle Control-Methoden extra Überlagerungen an, die das zusätzliche Mitgeben eines *GUILayout*-Parameters unterstützen.

#### **Listing 14.11** Zuweisung eines GUIStyles

```
GUILayout style = new GUIStyle();
style.fontSize = 18;
GUI.Label(rect, "Exrabutton", style);
```

Neben dem Typ *GUILayout* gibt es nun auch den Typ *GUISkin*. In einem *GUISkin* sind alle Control-Arten definiert, die von den GUI- und GUILayout-Klassen unterstützt werden. Eine Variable von diesem Typ ist also nichts anderes als ein Array bzw. eine Sammlung von *GUIStyles* für alle unterstützten Controls.

Über **Assets/Create/GUI Skin** können Sie sich ein *GUISkin* als Asset im *Project Browser* erstellen. Diesen können Sie dann im *Inspector* definieren und dann über eine *public*-Variable der statischen Variablen *skin* der GUI-Klasse zuweisen. Auf diese Weise können Sie auch mehrere *GUISkins* definieren, die dann vom Spieler ausgewählt werden könnten.

#### **Listing 14.12** Zuweisung eines GUISkin

```
public GUISkin skin;
void OnGUI() {
    GUI.skin = skin;
    GUI.Label(rect, "Willkommen im Dungeon des Grauens!");
}
```

Sie können die *skin*-Variable nicht nur zum Zuweisen, sondern auch zum Abfragen eines Styles nutzen. In dem Listing 14.11 weisen wir z. B. dem Label eine andere Schriftgröße zu. Allerdings besitzt dieser neue *GUILayout* wirklich nur diese *fontSize*-Definition, sonst nichts. Um nun den aktuellen Style zu übernehmen und nur die Größe zu verändern, können wir nun über die *GetStyle*-Methode der *skin*-Variablen den Style des jeweiligen Control-Typs übernehmen und modifiziert wieder zuweisen.

#### **Listing 14.13** Kopieren und Ändern eines GUIStyles

```
GUILayout gs = new GUILayout(GUI.skin.GetStyle("Button"));
gs.fontSize = 18;
if (GUILayout.Button(rect, "Beenden", gs))
{
    Application.Quit();
}
```

## ■ 14.4 uGUI

Aber der Version 4.6 bietet Unity ein komplett neues GUI-System namens *uGUI* an. Dieses nutzt eigene 2D-Objekte und ermöglicht Ihnen, GUI-Steuerelemente direkt in der *Scene View* wie andere *GameObjects* auch per Drag & Drop anzurichten und zu verändern. Die dazugehörigen *uGUI*-Controls erreichen Sie in Unity 4.6 über **GameObject/Create UI/**. Nutzen Sie zum Arbeiten mit diesem System die 2D-Ansicht (erreicht über den Button **2D** in der Toolbar der *Scene View*).

Zum Darstellen dieser *uGUI*-Controls benötigt jedes Control eine Komponente namens *UIRenderer*, die den *GameObjects* aber automatisch zugefügt wird. Außerdem nutzen diese Controls keine herkömmlichen Transform-Komponenten, sondern *RectTransforms*. Diese besitzen einige besondere Eigenschaften, die speziell bei der Gestaltung grafischer Nutzerschnittstellen wichtig sind. Mehr zu diesem Thema erfahren Sie im Abschnitt „*RectTransform*“.

Beachten Sie, wenn Sie *uGUI*-Komponenten in Ihre Skripte einbinden, dass Sie den Namespace `UnityEngine.UI` ebenfalls in diese einbinden (mehr zu *Namespaces* erfahren Sie im Kapitel „C# und Unity“).



### Beta-Version

Während ich dieses Buch geschrieben habe, befand sich *uGUI* noch in der Entwicklungsphase, weshalb sich meine Erläuterungen auf die Beta-Versionen dieses GUI-Systems beziehen. Es kann also sein, dass bis zur finalen Version noch das eine oder andere geändert bzw. ergänzt wurde.

### 14.4.1 Canvas

Das Hauptobjekt von *uGUI* stellt das sogenannte *Canvas*-Objekt dar. Jedes GUI-Control, das von diesem System dargestellt werden soll, muss ein Kind-Objekt eines *Canvas* sein. Ein *Canvas* wird durch ein Rechteck dargestellt, das sich, je nach *Render Mode*, dem Bildschirm selbstständig anpassen kann.

Auch wenn ein einzelnes *Canvas*-Objekt vollkommen ausreicht, um eine GUI zu gestalten, können Sie auch mehrere in einer Szene einsetzen. Sie können diese sogar als Kind-Objekte eines anderen *Canvas*-Objektes zufügen, wodurch Sie *uGUI*-Controls gruppieren können, die dann über die Eigenschaften der jeweiligen *Canvas* steuerbar sind.

- **Alpha** legt die Transparenz aller GUI-Controls, die diesem *Canvas* als Kind-Objekte zugeordnet wurden, fest. Hierdurch können Sie das Ein- und Ausblenden einer GUI bzw. eines GUI-Bereiches steuern.
- **Render Mode** legt fest, auf welche Weise dieses *Canvas* mit seinen *uGUI*-Controls in das Spiel eingebunden und dargestellt wird. Mehr dazu erfahren Sie im folgenden Abschnitt „*Render Modes*“.

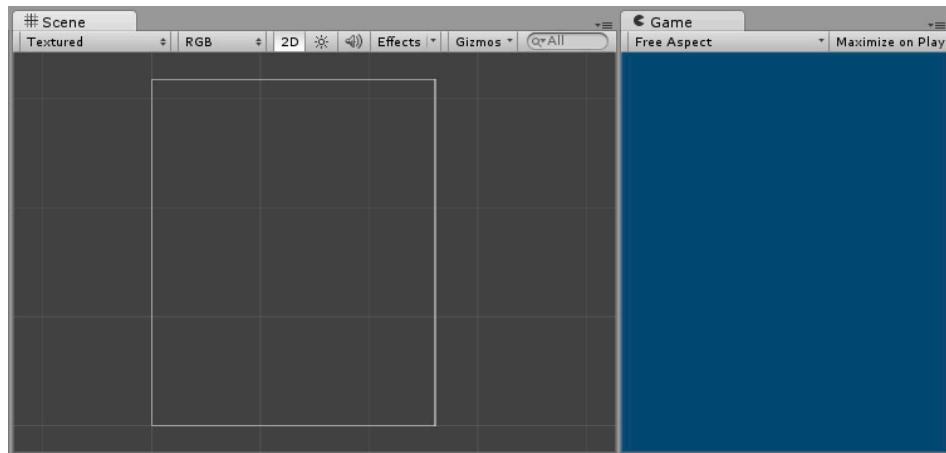


Bild 14.5 Ein angepasstes Canvas auf die Maße der Game View

Damit die Controls standardmäßig immer einem *Canvas* zugeordnet werden, unterstützt Sie Unity an dieser Stelle. So fügt die Entwicklungsumgebung jedes *uGUI*-Control automatisch erst einmal einem *Canvas*-Objekt zu, wenn Sie eines erzeugen. Existiert noch kein *Canvas*, so wird dieses automatisch angelegt.



#### Darstellung von übereinandergelegten Objekten

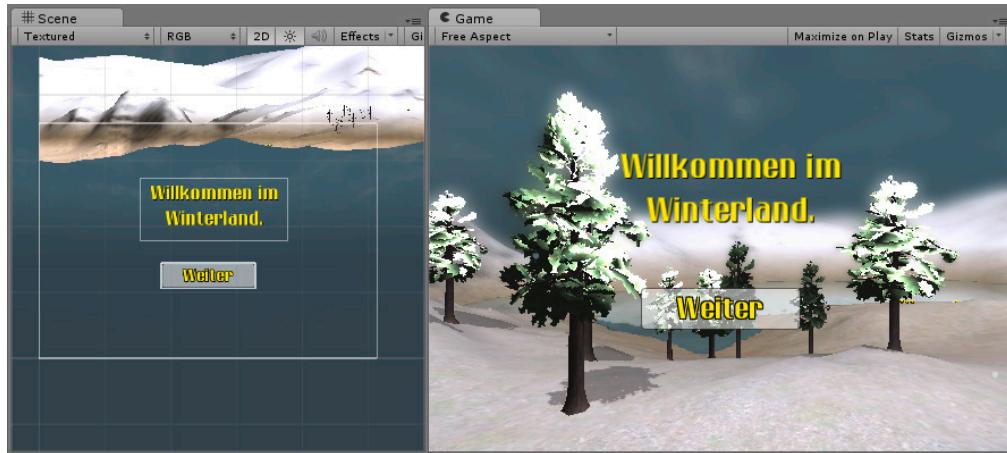
Wollen Sie später mehrere Objekte übereinanderlegen, z. B. eine Hintergrundgrafik und darüber einen Text, dann wird durch die Reihenfolge in der Hierarchy die Zeichnungsreihenfolge bestimmt. Die untersten Elemente werden dabei zuletzt gezeichnet. Per Drag & Drop können Sie dieses natürlich schnell entsprechend anpassen. Alternativ können Sie auch den Text dem Hintergrund als Kind-Objekt unterordnen.

##### 14.4.1.1 Render Modes

Die wichtigste Eigenschaft des *Canvas* ist der *Render Mode*. Dieser legt fest, auf welche Art bzw. von wem der *Canvas* mit all seinen Objekten gerendert werden soll.

##### Screen Space – Overlay

Der *Render Mode* „Screen Space – Overlay“ ist die Standardeinstellung. Der *Canvas* passt sich der Größe des aktuellen Bildschirms an und wird mit seinen Objekten über das normale Kamerabild gelegt.



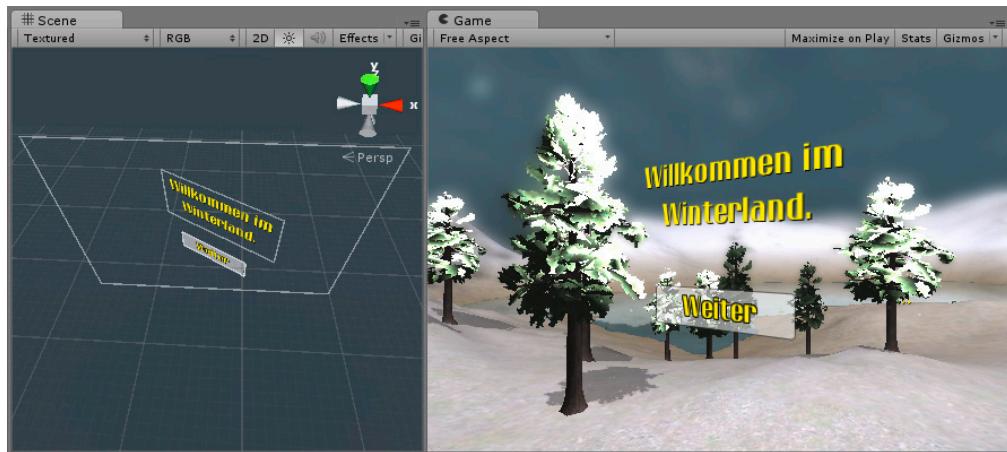
**Bild 14.6** GUI mit dem Render Mode Screen Space – Overlay

### Screen Space – Camera

In diesem Modus passen sich die Maße des *Canvas* ebenfalls dem Bildschirm an. Dieses Mal wird aber dem *Canvas* eine Kamera zugewiesen, über die der Inhalt dargestellt wird. Deshalb können Sie auch die Parameter der Kamera für die Darstellung der GUI nutzen. Wenn der *Projection*-Modus der Kamera z. B. auf „Perspective“ steht, können Sie nun auch die *uGUI*-Controls mit räumlicher Tiefe darstellen, was im vorherigen Modus nicht geht.

Fügen Sie für diesen Modus am besten der Szene ein zusätzliches *Camera-GameObject* mit folgenden Einstellungen zu:

- *Culling Mask*, dort nur den Layer „UI“
- *Depth*, darin einen höheren Wert als „Main Camera“
- *Clear Flags*, hier nehmen Sie „Don't Clear“.



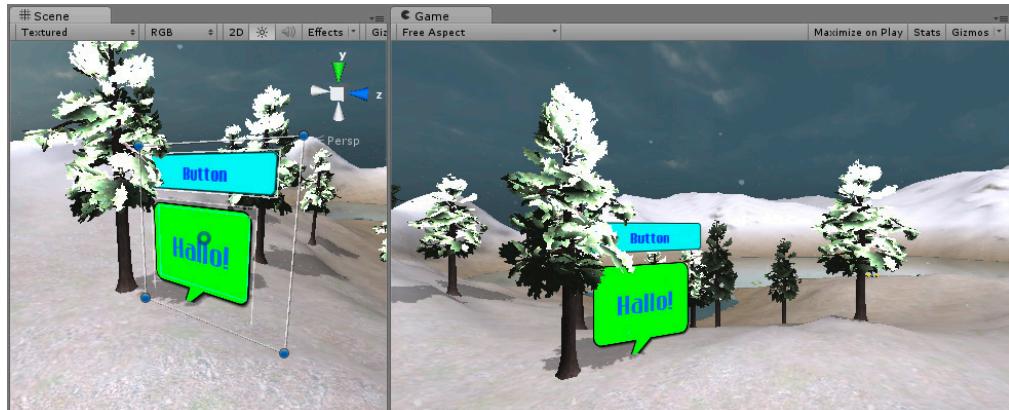
**Bild 14.7** Gedrehte Controls im Render Mode „Space Screen – Camera“

Beachten Sie zudem den zusätzlichen Parameter *Plane Distance*, der vom *Canvas* bei dieser Einstellung angeboten wird. Dieser definiert die Entfernung der Kamera zum *Canvas*.

## World Space

Der Modus „World Space“ lässt den *Canvas* als normales *GameObject* innerhalb einer Szene nutzen. Sie können es also skalieren, drehen und in der Szene platzieren.

Wollen Sie Buttons oder andere interaktive Controls auf diese Weise in der Szene platzieren, beachten Sie hierbei, dass Sie dem neuen *Canvas*-Parameter *Event Camera* die Kamera zuweisen, die das Bild rendert (meistens wird das die „Main Camera“ sein).



**Bild 14.8** In ein Spiel integrierter Canvas mit verschiedenen Controls

### 14.4.2 RectTransform

Eine Besonderheit aller *uGUI-GameObjects* ist, dass diese keine herkömmliche *Transform*-Komponente besitzen, sondern ein sogenanntes *RectTransform* nutzen. Dieses besitzt zu den bekannten *Transform*-Eigenschaften *Position*, *Rotation* und *Scale* noch weitere Eigenschaften, die für die GUI-Gestaltung und das Verhalten dieser Objekte wichtig sind.

Hierbei gehören auch Parameter, die die Außenmaße eines Controls bestimmen. Normalerweise werden die Maße eines *GameObjects* ja durch das *Mesh* bestimmt, das einem *GameObject* über den *MeshRenderer* zugewiesen wird. Da wir aber einem *uGUI-GameObject* keine *Meshe*s zuweisen, werden die Außenmaße stattdessen nun über das *RectTransform* und die folgenden Parameter bestimmt:

- **Width** bestimmt die Länge des rechteckigen GUI-Elementes.
- **Height** bestimmt die Höhe des rechteckigen GUI-Elementes.

Zum Bearbeiten der Eigenschaften einer *RectTransform*-Komponente empfiehlt es sich, das *Rect-Tool* zu nutzen, das Sie in den *Transform-Tools* der *Toolbar* ganz rechts finden. Wenn Sie dieses aktivieren, erscheinen an den Ecken des *RectTransforms* blaue Punkte (*RectHandles* auch genannt), über die Sie das Objekt verändern können. Wenn Sie hierbei die **[Umsch]**-Taste gedrückt halten, verändern Sie zudem die Größen in alle Richtungen gleichzeitig.



**Bild 14.9** Blaue Handles beim Nutzen des Rect-Tools



### Welche Parameter streckt das Rect-Tool?

Wenn das Objekt, das Sie mit dem Rect-Tool strecken bzw. stauchen, eine *RectTransform*-Komponente besitzt, verändern Sie dadurch immer die *Height*- und *Width*-Parameter. Besitzt das Objekt lediglich eine herkömmliche Transform-Komponente, werden stattdessen die *Scale*-Parameter verändert.

Zusätzlich besitzt jedes *RectTransform* einen eigenen *Pivot*-Punkt, den Sie manuell verschieben können. Wenn Sie das Objekt rotieren oder skalieren, wird diese Position als Ausgangspunkt genommen. Beachten Sie hierbei, dass Sie hierfür den *Pivot/Center Toggle* in der *Toolbar* auf *Pivot* stellen. Nun können Sie in der *Scene View* den *Pivot* manuell verschieben. Dieser erscheint als blauer Kreis, sobald Sie das *Rect-Tool* aktiviert haben.

Rotieren Sie danach das Objekt, wird es um den neu platzierten *Pivot-Punkt* gedreht. Skalieren Sie das Objekt, werden die vom *Pivot-Punkt* weiter entfernten Seiten stärker skaliert als die zum *Pivot-Punkt* näher liegenden.

Nun können Sie das Control über den **Blueprint**-Button einrasten lassen, womit Sie das rotierte Objekt skalieren und verschieben, als wäre dies nicht gedreht. Zudem aktiviert diese Funktion einen Snapping-Modus, der Ihnen beim Positionieren des Controls hilft.



### Mit dem Rect-Tool arbeiten

*uGUI*-Controls lassen sich am besten in der 2D-Ansicht mit dem Rect-Tool bearbeiten. Sollten Sie nach dem Wechseln in die 2D-Ansicht die Controls nicht sofort sehen, fokussieren Sie Ihre Sicht mithilfe der Taste **F** auf einen *Canvas* oder das zu bearbeitende *uGUI*-Control. Jetzt können Sie mit dem *Rect-Tool* sowohl die Größe verändern, das Objekt drehen als auch das GUI-Element verschieben.

Für das Rotieren bewegen Sie Ihren Mauszeiger (ohne gedrückte Maustaste) bei einem *Rect-Handle* außerhalb des Rechtecks, bis ein Rotations-Symbol

erscheint. Drücken Sie nun die linke Maustaste und Sie können das Objekt um den *Pivot-Punkt* drehen. Diese Funktionalität steht nur zur Verfügung, wenn der „Blueprint“-Button nicht aktiviert ist.



Bild 14.10 Rotiertes Control mit aktivierter „Blueprint“-Button

#### 14.4.2.1 Anchors

Ein weiteres Feature, das die *RectTransform*-Komponente bietet, sind *Anchors*. Hierbei handelt es sich um ein Layout-Konzept, das festlegt, wie sich ein Control-Objekt relativ zum Eltern-Objekt verhalten soll. Wird zum Beispiel das Anwendungsfenster des Spiels verkleinert, kann damit festgelegt werden, ob sich das Control an der linken oder der rechten Seite orientieren soll. Gleicher gilt natürlich auch für oben und unten.

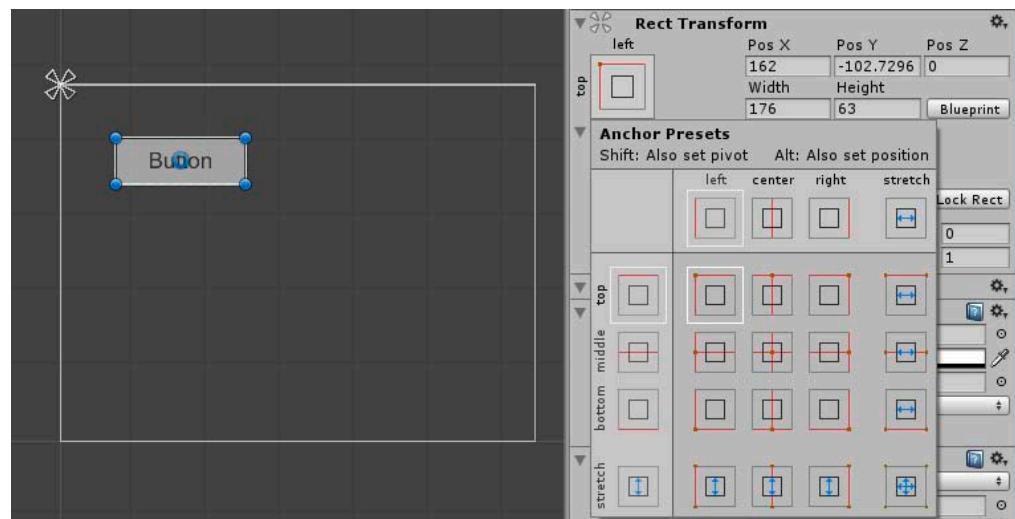
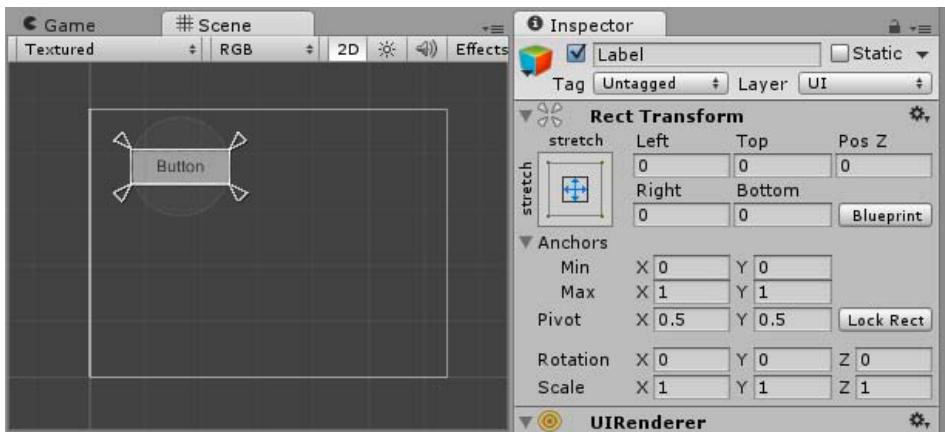


Bild 14.11 Anchor Presets

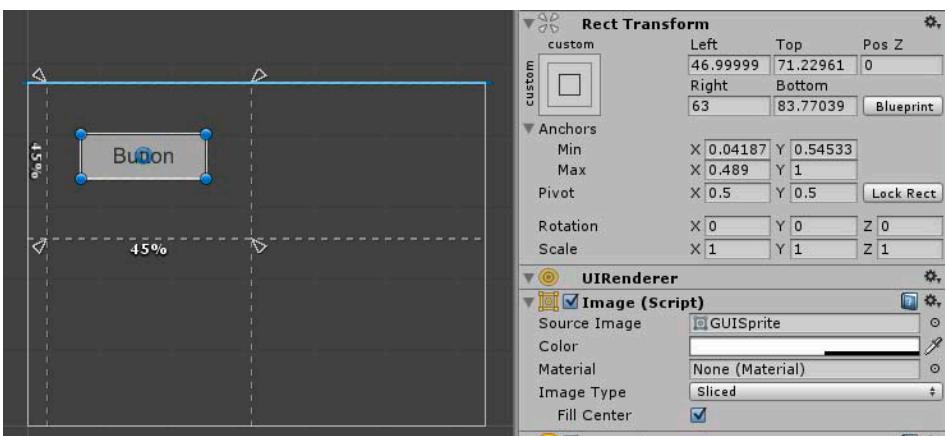
Unity bietet Ihnen bereits 16 vordefinierte Presets an, die für viele Anwendungsfälle bereits reichen sollten. Zum Einblenden dieser Presets klicken Sie hierfür einmal auf das Anchor-Symbol in der *RectTransform*-Komponente (dargestellt durch ein Quadrat oben links im Inspector). Ihnen wird nun eine Matrix angezeigt, in der Sie je nach horizontalem und vertikalem Verhalten ein Preset auswählen können.

Möchten Sie z. B. einen Button oben links platzieren, sollten Sie für diesen „top-left“ wählen, damit dieser in der linken oberen Ecke verbleibt, wenn sich die Fenstermaße ändern (wie im Bild 14.11 zu sehen). Soll sich ein Objekt aber nicht an einer, sondern an beiden orientieren, sollten Sie für die jeweilige Achse „stretch“ wählen. Bei einem Label, das sich immer im Zentrum des Buttons befinden soll, sieht das dann wie in Bild 14.12 aus.



**Bild 14.12** Zentrierung eines Labels in der Mitte eines Buttons

Diese Presets dienen aber nicht nur dem Setzen des *Anchors* an sich. Sie können durch das zusätzliche Drücken der **[Alt]**-Taste beim Auswählen eines Presets auch gleichzeitig eine Position dem aktuellen *uGUI*-Objekt zuweisen. Genauso können Sie mit **[Umsch]** (**[Shift]**) auch die Pivot-Position des Objekts bestimmen. Sowohl die Objektposition als auch die Pivot-Position können Sie natürlich danach manuell wieder ändern.



**Bild 14.13** Selbst definierte Anchors

Wenn Ihnen diese Presets nicht reichen, können Sie mithilfe der *Anchor-Handles* das Verhalten auch selber definieren und dieses noch differenzierter gestalten. Hierbei können Ihnen die numerischen *Anchors*-Werte helfen, die Ihnen zusätzlich im *Inspector* angezeigt werden.

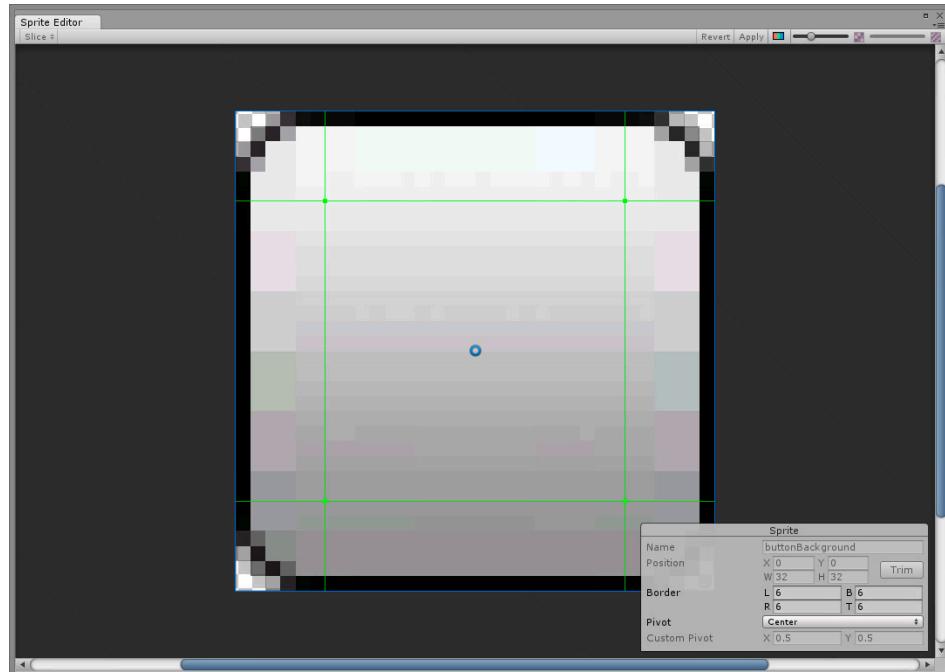
### 14.4.3 uGUI-Sprite Import

Unity nutzt zur grafischen Darstellung seiner Controls sogenannte *Sprites*. Dies ist ein spezieller *Texture Type*, den Sie in den *Import Settings* einer Textur wählen können. Für *uGUI* gibt es hier extra Import-Eigenschaften, die vor allem für Hintergrundgrafiken interessante Funktionalitäten bieten. Mit diesen können Sie dafür sorgen, dass auch Grafiken, die gestreckt, gestaucht oder verzerrt werden, in den Ecken im Originalzustand bleiben. Speziell bei Hintergrundgrafiken ist dies sehr interessant, da dort die Ecken meist abgerundet oder verschönert sind und deshalb am besten nicht verzerrt werden sollten.

Hierzu wählen Sie bei solch einer Hintergrundgrafik folgende Einstellungen in den *Import Settings*:

- **Texture Type** „Sprite“
- **Sprite Mode** „Single“

Danach öffnen Sie den *Sprite Editor* über den gleichnamigen Button. In dem folgenden Fenster können Sie nun sowohl den Pivot-Punkt als auch mithilfe der *Border*-Parameter (oder wahlweise per Drag & Drop der Hilfslinien in dem Vorschaubild) das Bild in neun Rechtecke aufteilen.



**Bild 14.14** Sprite Editor eines uGUI Sprites

Die hierbei entstehenden Rechtecke definieren dann das Verhalten der Grafik, wenn diese später gestreckt wird. Die Texturteile, die sich in den Ecken befinden, verbleiben dann im Originalzustand. Lediglich der Mittelteil wird komplett in alle Richtungen gestreckt.

Haben Sie Ihre Einstellungen vorgenommen, bestätigen Sie diese mit dem **Apply**-Button des *Sprite Editors*.

#### 14.4.4 Grafische Controls

Das *uGUI*-System bietet Ihnen zum Gestalten von GUIs verschiedene Steuerelemente an, die Sie beliebig kombinieren können. Das Besondere hierbei ist, dass die Controls selber modular aufgebaut sind, sodass bestimmte Basis-Controls bzw. deren Komponenten auch bei komplexeren Controls eingesetzt werden.

Grafische Controls, die keine direkten Aktionen des Benutzers erwarten, sind hiervon weniger betroffen als die interaktiven Controls, auf die wir gleich noch zu sprechen kommen. Die hier vorgestellten passiven Controls stellen daher auch eher die Basis-Controls dar, auf die die Interaktiven wieder aufbauen.

Zu jedem Control gibt es zum einen die nach ihm benannte Komponente, die Sie über **Component/UI/** finden, sowie ein fertig konfiguriertes *GameObject*, das Sie über **GameObject/Create UI/** erreichen können. Da die hier vorgestellten passiven Controls sehr einfach aufgebaute Steuerelemente sind, bestehen diese Control-*GameObjects* auch immer nur aus der Control-Komponente an sich und einem *RectTransform*.

##### 14.4.4.1 Text

Die *Text*-Komponente ist diejenige, die für das Darstellen von Textinformationen in *uGUI* verantwortlich ist. Sie erzeugt eine Art Textbox, in der der Text auch mehrzeilig angezeigt werden kann. Die Größe dieser Box wird durch das *RectTransform* bestimmt und kann natürlich auch über das *Rect-Tool* verändert werden. Da Unity auch Unicode unterstützt, können Sie hier natürlich auch das „ß“, deutsche Umlaute usw. anzeigen lassen.

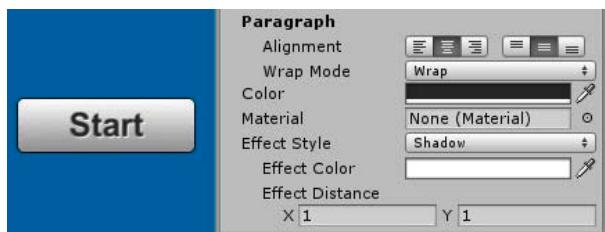
Sie können die einzelne *Text*-Komponente über das Menü **Component/UI** einem *GameObject* zufügen oder gleich ein komplettes *Text-GameObject* über **GameObject/Create UI** der Szene zufügen. Die meisten *uGUI*-Controls besitzen solche als Kind-Objekte zum Darstellen der Beschriftungen. Dabei werden diese meistens als „Label“ bezeichnet.

Mithilfe der verschiedenen Eigenschaften der *Text*-Komponente können Sie dann diese je nach Einsatzzweck passend konfigurieren.

- **Font** bestimmt den einzusetzenden Schriftfont. Unity unterstützt *True Type Fonts* (.ttf und .dfont) und *Open Type Fonts* (.otf). Um einen eigenen Schriftfont nutzen zu können, müssen Sie diesen in den *Project Browser* ziehen.
- **Font Style** legt fest, ob die Schrift normal, fett, kursiv oder beides dargestellt werden soll.
- **Font Size** definiert die Schriftgröße.
- **Line Spacing** definiert den Abstand bei mehrzeiligen Texten zwischen zwei Zeilen.
- **Rich Text** legt fest, ob Spezialeffekte wie HTML-ähnliche Tags unterstützt werden. So können Sie durch eine Aktivierung dieses Parameters bei einem Text wie "Hallo <color=#00ffff>Heil</color>!" die Farbe eines einzelnen Wortes verändern.

- **Alignment** legt fest, woran sich der Inhalt der Textbox horizontal wie auch vertikal orientieren soll.
- **Wrap Mode** bestimmt das Verhalten des Textes, wenn dieser zu lang ist. Bei der Option *Wrap* wird versucht, Zeilenumbrüche zu erzeugen. Reicht der Platz dann immer noch nicht, wird der Rest ausgeblendet. *ResizeBounds* passt die Textbox dem Text an. Die Option *ShrinkText* verkleinert die Textgröße, damit der Inhalt komplett in der bestehenden Textbox angezeigt werden kann.
- **Color** legt die Farbe des Textes fest.
- **Effect Style** ermöglicht zusätzliche Effekte wie das Zufügen eines Schattens (Shadow) oder eines Rahmens um den Text (Outline). Zusätzlich können Sie über *Effect Color* die Farbe des Schattens bzw. des Rahmens definieren und über *Effect Distance* den Abstand zur Originalschrift festlegen.

Auch wenn die Auswahl an *Effect Styles* aktuell noch nicht sehr umfangreich ist, können Sie mit diesen aber mehr als nur einfache Schatten und Einrahmungen erstellen. So können Sie mit der Einstellung *Shadow*, einer dunklen Schriftfarbe (*Color*) und einer hellen Farbe für *Effect Color*, wie z.B. Weiß, den Text wie eine Inschrift erscheinen lassen. In dem Bild 14.15 wurde dies gemacht, um der Beschriftung eines Buttons etwas mehr Tiefe zu verleihen.



**Bild 14.15**

Parameter und Auswirkung eines hellen Schattens bei Text

Sollten Sie bei niedrigen *Font Size*-Werten Probleme mit der Schärfe der Buchstaben bekommen, können Schatten und Umrandungen helfen, diese schärfen wirken zu lassen. Zudem können Sie noch den *Font Style* auf „Bold“ setzen, was z.B. in Bild 14.11 ebenfalls getan wurde.

**! Zeilenumbrüche**

Wenn Sie den Textinhalt per Code und nicht im Inspector definieren, können Sie durch die Angabe von "\n" einen Zeilenumbruch innerhalb eines Strings erzwingen. Daher könnte ein Begrüßungstext so aussehen: "Hallo, mein Held!\nSchön, dass Du da bist.".

#### 14.4.4.2 Image

Die *Image*-Komponente ist die Standardkomponente von *uGUI*, um Grafiken darzustellen. Dabei müssen die Grafiken als Sprite vorliegen, also in den *Import Settings* mit dem *Texture Type* „Sprite“ definiert sein. Mehr dazu im Abschnitt „uGUI-Sprite-Import“.

Jedes sichtbare Control (abgesehen von dem *Text-GameObject*) besitzt mindestens eine solche Komponente, um seine Hintergrundgrafik darstellen zu können. Einige, wie z.B. das *Toggle*-Control, besitzen sogar gleich mehrere: eines für den Hintergrund und ein weiteres zur Darstellung des ein- und ausschaltbaren Control-Elementes (meistens dargestellt durch einen Haken, Punkt oder ein Kreuz).

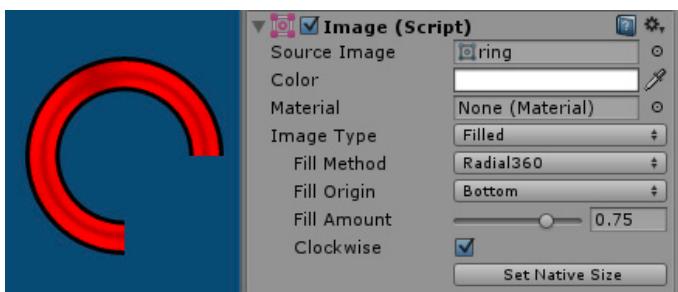
Auch hier können Sie entweder die *Image*-Komponente einem *GameObject* hinzufügen oder ein fertig konfiguriertes *Image-GameObject* der Szene hinzufügen.

Das *Image-Component* besitzt dabei folgende Parameter:

- **Source Image** bestimmt die dazustellende *Sprite*-Grafik. Weisen Sie dieser Eigenschaft ein einzelnes *Sprite* zu (nicht das gesamte *Sprite-Asset*). Klappen Sie hierfür das gewünschte *Sprite-Asset* auf und ziehen das einzelne *Sprite* auf die Eigenschaft.
- **Color** legt eine Farbe fest, mit der die Grafik noch einmal zusätzlich eingefärbt wird.
- **Image Type** definiert die Darstellungsart des Sprites innerhalb des *RectTransforms*, die je nach Einsatzzweck der Grafik eingestellt werden sollte.

Jeder *Image Type* hat eine besondere Funktionalität, die ich im Folgenden vorstellen möchte.

- **Simple** skaliert die Grafik gleichmäßig auf die Größe, die durch das *RectTransform* vorgegeben wird. Mit dem Button **Set Native Size** wird diese Größe auf die der Originalgrafik zurückgesetzt.
- **Slices** nutzt die 3×3-Aufteilung des Sprite-Imports (siehe „uGUI-Sprite-Import“). Bei dieser werden die Ecken in der Originalauflösung belassen und nur der Rest wird in entsprechende Richtungen skaliert. Dieser Modus eignet sich vor allem für Control-Hintergrundgrafiken.
- **Tiled** wiederholt die Grafik in der Originalgröße über das gesamte *RectTransform*.
- **Filled** stellt zunächst die Grafik wie Simple dar. Zusätzlich bietet es aber die Möglichkeit, mithilfe des *Fill Amount*-Parameters nur Anteile des Bildes darzustellen. Über *Fill Method* und *Fill Origin* können Sie zudem definieren, auf welche Art der Bildanteil gefüllt bzw. ausgeblendet wird. Mit dieser Funktionalität können Sie z.B. sehr einfach balken- oder kreisförmige Lebensanzeigen erstellen.



**Bild 14.16**

Parameter und Auswirkung des *Image Types* „Filled“ am Beispiel eines Rings

#### 14.4.4.3 Raw Image

Sollte es vorkommen, dass Sie kein *Sprite*, sondern eine herkömmliche Grafik mit dem *Texture Type* „Texture“ oder auch „GUI (Legacy)Editor“ darstellen wollen, können Sie die Komponente *Raw Image* nutzen. Auch diese Komponente können Sie wieder über das Menü

**Component/UI** einem *GameObject* zufügen oder als komplettes *Raw Image-GameObject* über *GameObject/Create UI* der Szene zufügen.

Beachten Sie an dieser Stelle, dass Unity Technologies in der Dokumentation extra darauf hinweist, immer die *Image*-Komponente der *Raw Image*-Variante zu bevorzugen und diese nur dann zu nutzen, wenn es wirklich notwendig ist.

#### 14.4.5 Interaktive Controls

Die interaktiven Controls bestehen eigentlich immer aus einem Haupt-*GameObject* mit dem Control-Skript, in dem die eigentliche Logik des Controls implementiert ist, und einigen Kind-Objekten. Diese Unterobjekte dienen dann dem Anzeigen des Controls, z.B. der Beschriftung oder eines Grafiksymbols. Je komplexer dabei ein Steuerelement ist, desto mehr Unterobjekte besitzt dieses. Vor allem die beiden Komponenten *Text* und *Image*, die Sie bereits im Abschnitt „Grafische Controls“ kennengelernt haben, finden sich deshalb eigentlich bei jedem interaktiven Control wieder.

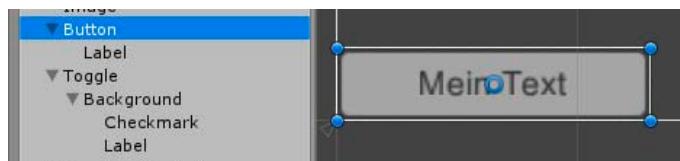


Bild 14.17  
Button-Struktur

Ein Button besteht zum Beispiel aus dem „Button“-Hauptobjekt und einem „Label“-*GameObject*, das den Text des Buttons darstellt (siehe Bild 14.17). Während das Hauptobjekt die Logik des Buttons in Form der *Button*-Komponente und die Hintergrundgrafik über eine *Image*-Komponente bestimmt, ist das Kind-Objekt „Label“ lediglich ein *Text-GameObject*, das die Beschriftung anzeigt.

Das eigentliche Interaktionsverhalten wird in den control-spezifischen Komponenten beschrieben, die durch die strikte Trennung zur visuellen Darstellung aber recht übersichtlich ausfallen. Diese bestehen eigentlich immer aus vier Bereichen, wobei die ersten beiden immer gleich sind.

#### Transition

Der Bereich **Transition** definiert das optische Verhalten des jeweiligen Controls bei Interaktion mit dem Benutzer. Hierbei bestimmen Sie, wie das Control in den Zuständen Normal, Highlighted, Pressed und Disabled aussehen soll. Dabei können Sie die Art der Veränderung bestimmen und dann natürlich auch die Zustände an sich.

- *None* deaktiviert alle Zustandseffekte.
- *ColorTint* färbt die Texturen bei jedem Zustand mit einer anderen Farbe ein. Über den Parameter *Fade Duration* bestimmen Sie zusätzlich, wie lange das Wechseln zwischen den Farben dauern soll.
- *SpriteSwap* zeigt je nach Zustand eine andere Grafik im *Sprite*-Format an. Weisen Sie hierfür allen *Sprite*-Slots ein *Sprite* zu. Sie können natürlich auch für mehrere Zustände die gleiche nutzen.

- *Animation* animiert das Control je nach Zustand. Über den Button **Auto Generate Animation** können Sie Standardanimationen erstellen, die Sie sofort nutzen, später aber auch nach Belieben verändern können. Mehr zu diesem Bereich erfahren Sie in Abschnitt 14.4.7.

## Navigation

Mit der **Navigation**-Eigenschaft legen Sie fest, zu welchen interaktiven Controls der Benutzer von diesem aus über die Pfeiltasten wechseln kann.

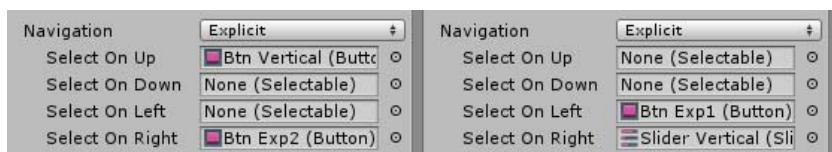
- *None* deaktiviert das Wechseln mit den Pfeiltasten zu einem anderen Control.
- *Horizontal* erlaubt das Wechseln zu Controls, die seitlich von diesem angeordnet sind.
- *Vertical* lässt den Benutzer den Fokus auf vertikal positionierte Controls wechseln.
- *Automatic* lässt Unity bestimmen, was für Controls über welche Tasten erreichbar sind.
- *Explicit* erlaubt Ihnen genau festzulegen, welche Controls über welche Richtungstasten den Fokus erhalten. Dabei können Sie jeder Pfeiltaste ein Control zuweisen, das über diese angesteuert werden soll.



**Bild 14.18**

Control-Anordnung mit zugewiesenen Navigation-Parametern

In Bild 14.18 wurde den Controls die Navigation-Option zugewiesen, wie sie beschriftet wurden, wobei der rechte Slider die Option „Horizontal“ erhalten hat. Die Zuweisungen der beiden Explicit-Buttons können Sie dem Bild 14.19 entnehmen.



**Bild 14.19** Explizite Zuweisungen der ansteuerbaren Controls

Nach den *Navigation*-Einstellungen folgen die control-bezogenen Parameter. Hier nehmen Sie spezielle Einstellungen vor, die sich lediglich auf den jeweiligen Control-Typ beziehen. Auf diese kommen wir gleich noch zu sprechen.

## Event-Delegates

Die **Event-Delegates** bilden den Schluss jeder interaktiven Control-Komponente. Alle *uGUI*-Controls unterstützen dabei sogenannte *Events*, die bei unterschiedlichen Ereignissen beliebig viele Methoden aufrufen können. Das Aufrufen dieser Methoden wird dabei über Verweise (sogenannte Delegates) erzeugt, die Sie über eine Liste beliebig oft anlegen können.



**Bild 14.20** Aufruf einer Methode über einen Event-Delegate

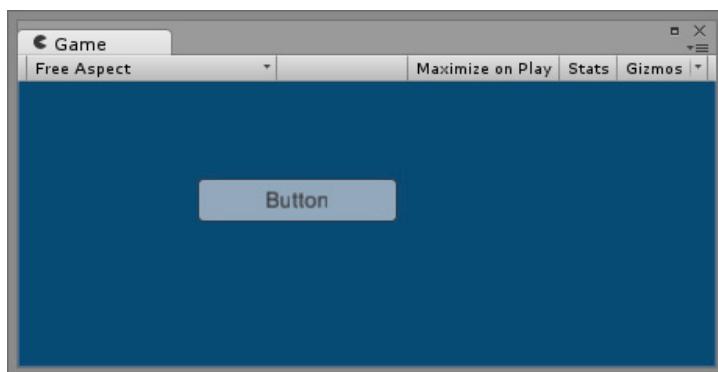
Jedes interaktive Control besitzt nun ein *Event* direkt in der Komponente und kann darüber Methoden anderer Skripte und auch anderer *GameObjects* ausführen. Dabei gehen Sie wie folgt vor:

1. Fügen Sie der *Delegate*-Liste des *Events* über das Pluszeichen eine neue Zeile zu.
2. Weisen Sie dem ersten Feld ein *GameObject* zu, von dem Sie etwas ausführen möchten.
3. Im folgenden Drop-down-Menü bestimmen Sie dann eine Methode einer Komponente, die ausgeführt werden soll. Beachten Sie hierbei, dass Ihnen nur *public*-Methoden angeboten werden.
4. Als Letztes legen Sie den Übergabewert der ausgewählten Methode fest. Dies kann z.B. eine *Transform*-Komponente sein, wenn Sie z.B. „UnityEngine.Transform/LookAt (Transform)“ ausgewählt haben (diese Methode dreht das Objekt in die Richtung eines anderen Objektes bzw. dessen Transforms). Sollte die Methode keinen Parameter verlangen, ist dieses letzte Feld deaktiviert.

Beachten Sie, dass die aufrufbaren Methoden auch einen Übergabeparameter besitzen dürfen. So können z.B. Werte vom Typ *string* oder auch *int* vom Control übergeben werden. Als besonderes Feature können Sie dem Übergabeparameter den Typ des aufrufenden Controls geben. In diesem Fall besitzt der Parameter dann eine Referenz auf das Control, dessen Event die Methode aufgerufen hat, womit Sie dann Informationen über dessen Zustand etc. abfragen können.

#### 14.4.5.1 Button

Der Button ist das wohl am meist eingesetzte interaktive Control einer jeden GUI. Wie alle interaktiven Control-Komponenten besitzt auch die Button-Komponente natürlich die *Transition*-Einstellungen und die *Navigation*-Eigenschaft. Spezielle Parameter besitzt sie aber nicht, sodass Sie nur noch das „On Click“-Event konfigurieren müssen, das auslöst, sobald auf den Button gedrückt wird.



**Bild 14.21** Default-Design eines Buttons

Wollen Sie z.B. eine Methode eines eigenen Skriptes aufrufen, weisen Sie zunächst das *GameObject* dem ersten Slot zu, dem Sie das Skript zugewiesen haben. Danach wählen Sie im Drop-down-Menü das Skript und anschließend den Namen des Skriptes. Achten Sie darauf, dass die Methode als *public* deklariert werden muss. Das Bild 14.22 zeigt eine solche Zuweisung, indem die Methode *LoadLevel* des Skriptes *MainMenu* aufgerufen wird, das wiederum der „Main Camera“ zugewiesen wurde. Im Abschnitt „Toggle“ finden Sie ein Beispiel, wie dieses Skript aussehen könnte.



**Bild 14.22** Aufruf einer Skript-Methode, die der Kamera angehängt wurde

#### 14.4.5.2 InputField

Möchten Sie vom Spieler Daten wie Namen, E-Mail-Adressen oder Ähnliches erfassen, eignet sich das *InputField*-Control. Dieses nutzt eine *Text*-Komponente, um den eingegebenen Text anzuzeigen, sowie eine *Image*-Komponente, um die Hintergrundgrafik des *InputField*-Controls grafisch darzustellen.

Ein besonderes Feature dieses Controls ist die Eigenschaft *Character Limit*, mit der Sie die Möglichkeit haben, die Zeichenlänge des Eingabestrings zu begrenzen. Der Wert 0 hebt dabei die Begrenzung auf und lässt unbegrenzt viele Zeichen zu. Gerade wenn Sie den Text später in eine Datenbank schreiben wollen, ist es wichtig, dass Sie die Zeichenlänge auf die Größe des Datenbank-Feldes begrenzen. Ansonsten tritt später beim Zurückschreiben ein Fehler auf, was natürlich verhindert werden sollte.

#### 14.4.5.3 Toggle

Das *Toggle*-Control dient dem Anzeigen und Steuern eines binären Wertes, der über den Parameter *Active* dargestellt wird.

Das Control kann dabei auf zwei unterschiedliche Weisen genutzt werden: Zum einen können Sie es als Checkbox einsetzen, dessen Wert der Benutzer ein- und ausschalten kann. Zum anderen können Sie das Control auch als Radio-Button nutzen. Hierbei verbinden Sie mehrere *Toggles* über eine *Toggle Group*, bei der immer nur ein *Toggle* zurzeit aktiv sein kann und entsprechend die anderen ausschaltet. Da die mir vorliegenden Beta-Versionen noch keine *Toggle Groups* unterstützen, werde ich mich auf die Einzelnutzung eines *Toggles* beschränken.

Jede *Toggle*-Komponente besitzt ein „On Value Changed“-Event, das auslöst, sobald sich dessen Inhalt ändert. Dieses Event bietet die gleichen Methodenaufrufe und Funktionalitäten wie das „On Click“-Event des Buttons, sodass sie natürlich auch Skript-Methoden aufrufen können, sobald sich der Inhalt ändert.

Wenn Sie nun in der aufgerufenen Methoden noch den Zustand des Controls haben möchten, um z.B. abhängig vom Zustand des Toggles etwas auszuführen, fügen Sie der Methode einfach einen Übergabeparameter vom Typ *Toggle* zu. Über diesen erhalten Sie dann eine Referenz auf das aufrufende Control, über die Sie dann alle Informationen erhalten sollten,

die Sie benötigen. Das Listing 14.14 zeigt Ihnen eine solche Methode. Über die Eigenschaft `isOn` des Übergabeparameters können Sie dann den Zustand des *Toggles* ermitteln.

#### **Listing 14.14** Toggle-Zustand ermitteln

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;
public class MainMenu : MonoBehaviour {

    public void ToggleValueChange(Toggle tgl)
    {
        if (tgl.isOn)
            Debug.Log("Active");
        else
            Debug.Log("Deactive");
    }

    public void LoadLevel()
    {
        Application.LoadLevel(1);
    }
}
```

Achten Sie bei Listing 14.14 auf den *Namespace* `UnityEngine.UI`, der zwingend eingebunden werden muss. Ansonsten können Sie den *Toggle*-Typ nicht im Skript nutzen.

#### **14.4.5.4 Slider**

Mit *Slidern* können Sie *float*-Werte zwischen 0 und 1 über die Länge eines Darstellungsobjektes darstellen und natürlich auch verändern. *Slider* können entweder horizontal oder vertikal ausgerichtet sein. Um den Wert in einem Skript abzufragen, stellt der *Slider*-Typ die Variable `value` bereit.

Damit man mit *Slidern* unterschiedlichste Anwendungsszenarien realisieren kann, bieten diese verschiedene Parameter an.

- **Direction** bestimmt die Richtung des Werteverlaufs vom *Slider*. Hier stehen „*LeftToRight*“, „*RightToLeft*“, „*BottomToTop*“ und „*TopToBottom*“ zur Verfügung.
- **Value** gibt Ihnen die Möglichkeit, zur Entwicklungszeit hier den Startwert zu hinterlegen.
- **Number Of Steps** gibt die Menge der Stufen an, die der *Slider* einnehmen kann. Der Wert 0 bedeutet hier ein stufenloses Verschieben des Handles.

#### **14.4.5.5 Scrollbar**

Mit einer *Scrollbar* erstellen Sie einen Regler, mit dem Sie wie beim *Slider* einen *float*-Wert zwischen 0 und 1 steuern können. Über die *Scrollbar*-Variable `value` können Sie den aktuellen Wert per Skript abfragen. Wie der *Slider* kann auch eine *Scrollbar* horizontal oder vertikal ausgerichtet werden.

Die *Scrollbar* stellt Ihnen mehrere Parameter zur Verfügung, um diese den Anforderungen entsprechend anzupassen.

- **Direction** bestimmt die Richtung des Werteverlaufs der *Scrollbar*. Dabei stehen „LeftToRight“, „RightToLeft“, „BottomToTop“ und „TopToBottom“ zur Verfügung.
- **Value** gibt Ihnen die Möglichkeit, zur Entwicklungszeit hier den Startwert zu hinterlegen.
- **Size** legt die Größe des Reglers (*GameObject* „Handle“) prozentual von der Gesamtlänge der *Scrollbar* fest.
- **Number Of Steps** gibt die Menge der Stufen an, die die *Scrollbar* einnehmen kann. Der Wert 0 bedeutet dabei ein stufenloses Verschieben des Handles.

#### 14.4.5.6 Selection List

Ein *Selection List*-Control können Sie zum Erstellen von Auswahlmenüs nutzen. Als Hauptevent besitzt die Komponente das „On Selection Changed“-Event, das wieder beliebig viele Skript-Methoden und andere Funktionalitäten aufrufen kann.

Über eine *Options*-Liste können Sie der Control-Komponente beliebig viele Items zufügen, die dann in der Auswahl angezeigt werden. Sie können aber auch zur Laufzeit dem Control weitere Items per Code zufügen. Auf die Item-Liste können Sie über die Objekt-Variable *items* zugreifen, die wiederum ein *List*-Objekt vom Typ *SelectionList.ItemData* ist. Hierüber haben Sie alle Möglichkeiten, die Sie auch bei anderen List-Objekten haben.

Im folgenden Listing fügen wir der Liste ein neues Item hinzu und ermitteln hierbei die aktuelle Menge, um diese mit in den Namen/Text einzubinden. Schließlich fragen wir den Text des aktuell ausgewählten Items in einer gesonderten Methode ab.

**Listing 14.15** Items einer Selection List zufügen und auswerten

```
public SelectionList selectionList;
void Start ()
{
    SelectionList.ItemData item = new SelectionList.ItemData();
    item.text = "Item No. " + (selectionList.items.Count + 1).ToString ();
    selectionList.items.Add (item);
}
void ListSelectionChange() {
    Debug.Log(selectionList.selection.text);
}
```

Die Methode *ListSelectionChange* des Listings 14.15 könnte z.B. über das „On Selection Changed“-Event der *Selection List* aufgerufen werden, sodass dann über *selectionList.selection* direkt auf das neu ausgewählte Item zugegriffen werden kann.

#### 14.4.6 Controls designen

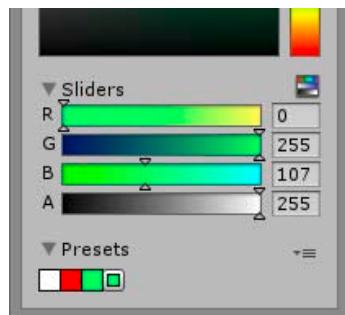
Das Anpassen des Designs eines Steuerelementes ist im Grunde sehr einfach, da das Aussehen aller *uGUI*-Controls nur durch *Image*- und *Text*-Komponenten bestimmt wird. Sie müssen nur beachten, dass komplexere Controls, wie der *Toggle* oder die *Selection List*, auch Kind-Objekte besitzen, die teilweise ebenfalls diese Komponenten besitzen und für die Darstellung des Controls zuständig sind.

Dabei folgen die Strukturen der Controls natürlich auch Regeln, die ich im Folgenden anhand der Controls der Beta-Versionen etwas erläutern möchte.

- Die *Image*-Komponente für die Hintergrundgrafik befindet sich bei einfachen Controls direkt auf dem Hauptobjekt. Bei komplexeren befindet sich diese auf einem Kind-Objekt mit dem Namen „Background“.
- Die *Text*-Komponenten für die Beschriftungen befinden sich auf Kind-Objekten, die den Namen „Label“ tragen.
- Je nach Komplexität des Control-Typs gibt es noch Zusatzobjekte, die dann ihrer Aufgabe entsprechend benannt werden, z.B. „Handle“ für den Schieberegler einer *Scrollbar* oder „Checkmark“ für den Marker (Kreuz, Haken etc.) eines *Toggles*. Bei aufklappbaren Controls wie der *Selection List* wird der aufgeklappte Zustand mithilfe eines Template-*GameObjects* definiert, das aber zur Laufzeit deaktiviert ist.

Wenn Sie später aufwendigere GUIs mit mehreren Controls gestalten wollen, empfiehlt es sich, für jedes Control zunächst ein *Prefab* zu erstellen und dieses dann zu nutzen. Auf diese Weise brauchen Sie nur einmal jedes Control zu designen und können das Aussehen auf die anderen Instanzen einfach übertragen.

Sollten Sie mit Farben arbeiten, sei es bei den *Transition*-Effekten oder den Color-Eigenschaften der *Image*-Komponenten, ist es hilfreich, mit *Presets* zu arbeiten. Hierbei können Sie im Color-Dialog Farben einer *Presets*-Library zufügen, die Ihnen dann jedes Mal beim Öffnen des Color-Dialogs zur Auswahl stehen. Zudem können Sie solche Preset-Libraries auch abspeichern. Im Drop-down-Menü, das Sie unten rechts im Color-Dialog finden, gibt es den Menüpunkt **Create New Library**, worüber Sie ein *Asset* erzeugen können, das diese Farbinformationen speichert und dann auch in anderen Projekten genutzt werden kann.



**Bild 14.23**

Presets-Library und Drop-down-Menü des Color-Dialogs

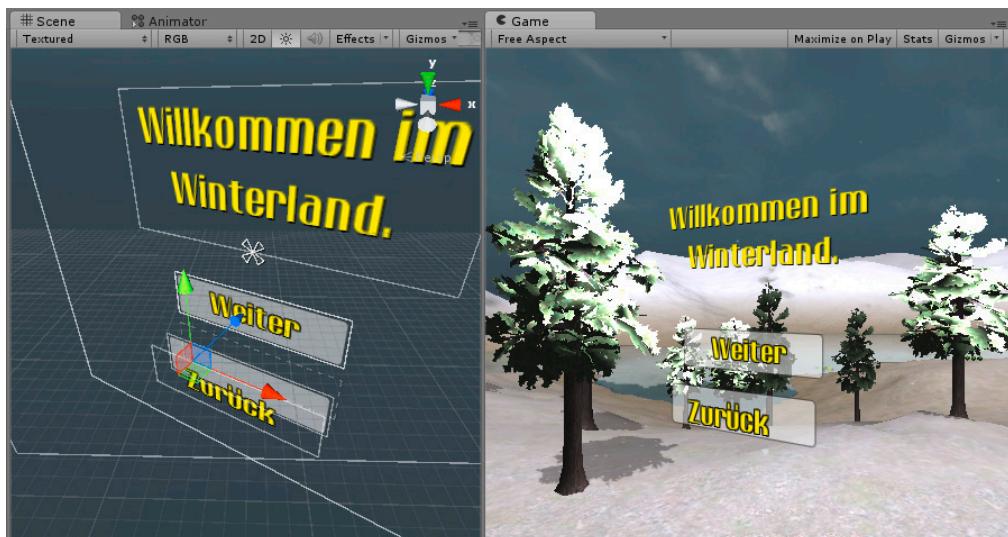
#### 14.4.7 Animationen in uGUI

Ein enormes Potenzial dieses GUI-Systems ist die Animationsfähigkeit der *uGUI*-Objekte. Da aber bei den effektvollsten Animationen meistens auch die räumliche Tiefe zum Tragen kommt (Objektdrehungen, Rein-/Rausfliegen von Controls usw.), sollten Sie bei solchen animierten GUIs gleich den *Render Mode* auf „Screen Space – Camera“ einstellen, da der Default-Modus dieses nicht unterstützt.

Alle interaktiven Controls unterstützen bereits standardmäßig für die *Transitions* eigene Animationen. Das bedeutet, dass die Controls auf Interaktionen des Benutzers mit Anima-

tionen reagieren können. Über den Button **Auto Generate Animation** können Sie hierbei Standardanimationen für jeden Zustand automatisch anlegen lassen, die Sie dann später über das Animationsfenster verändern können. Dabei werden auch alle notwendigen Komponenten erzeugt und dem Control zugewiesen. Alternativ können Sie auch komplett eigene Animationen erstellen. Mehr zum Thema Animationen erfahren Sie im gleichnamigen Kapitel 17.

Neben diesen Standardanimationen können Sie aber auch jedes Control zusätzlich noch mit Animationen ausstatten. Wollen Sie komplette Menüs animieren, empfiehlt es sich, zunächst alle Controls in einem gemeinsamen Objekt, z. B. auf einem *Panel*- bzw. *Window*-Objekt, zu gruppieren. Auf diese Weise brauchen Sie nur noch ein Objekt zu animieren und nicht mehr mehrere. Beachten Sie, dass *Canvas*-Objekte im „Screen Space – Camera“ – Render Mode nicht animiert werden können, da sich diese immer an der Kamera bzw. am Bildschirm orientieren.



**Bild 14.24** Eigene Highlighted Animation lässt Button nach vorne kommen.

#### 14.4.8 Event Trigger

Alle *uGUI*-Controls unterstützen ein von Unity entwickeltes Event-System, das Ihnen hilft, auf Nutzeraktivitäten bzw. -eingaben reagieren zu können. Sie können jedem Control, auch den grafischen, eine *Event Trigger*-Komponente zufügen, über die Sie dann die auszuwertenden *Events* auswählen und Delegates erzeugen, um entsprechende Methoden auszuführen.

Die interaktiven Controls des *uGUI*-Systems besitzen bereits per Default ein *Event*, mit dem Sie Vorgänge anstoßen können. Hierfür müssen Sie dem *Event* nur ein *GameObject* zuweisen, von dem Sie dann die Funktionalitäten der angehängten Komponenten ausführen können. Trotzdem können Sie diesen Controls auch noch zusätzlich einen *Event Trigger* zufügen, um weitere Events abzufangen.

Haben Sie die Komponente einem Control zugefügt, fügen Sie zunächst über den Button „Add New“ ein neues *Event* dem *Event Trigger* zu. Dabei definieren Sie, welche Art von Ereignissen überwacht bzw. ausgewertet werden soll, hier können Sie z.B. „PointerEnter“, „PointerExit“, „Drag“, „Drop“, „Scroll“ und einiges mehr auswählen. Anschließend können Sie über die *Delegates*-Liste des *Events* verschiedene *GameObjects* dem *Event* zuweisen, dessen Methoden ausgelöst werden sollen, sobald das *Event* ausgelöst wird.

Das Vorgehen ist hierbei so, dass Sie im ersten Slot das *GameObject* zuweisen. Im folgenden Drop-down-Menü wählen Sie dann eine Komponente aus sowie eine Methode, die ausgeführt werden soll. Als Letztes wählen Sie dann, wenn erforderlich, einen Parameter der vorher ausgewählten Methode. Dies kann der Name einer Methode sein, die per *SendMessage* gestartet werden soll, ein *Transform*, das als Ziel für ein *LookTo*-Befehl dienen soll, usw.

Auf diese Weise können Sie auch die Transition-Effekte der interaktiven Controls noch interessanter machen. Während man normalerweise entweder einen Farbwechsel, eine Animation oder einen Sprite-Wechsel machen kann, können Sie z.B. mithilfe eines zusätzlichen *PointerEnter-Events* beliebige Reaktionen ausführen, z.B. einen Farbwechsel der Schrift erzwingen, Animationen anderer Objekte anstoßen oder auch andere *uGUI*-Controls ein- bzw. ausblenden. Die Möglichkeiten werden dadurch riesig.

Aber auch verschiebbare Controls können mit *Event Triggern* leicht umgesetzt werden. Eine einfache Variante zeigt das Listing 14.16. Fügen Sie dieses einem beliebigen *uGUI*-Control zu. Danach fügen Sie diesem noch eine *Event Trigger*-Komponente zu und legen ein *Drag-Event*. Dieses ruft dann wiederum die Methode *Dragging* auf. Das war es schon. Danach können Sie das Spiel starten und per Maus das Control verschieben.

#### **Listing 14.16** Verschiebbares uGUI-Control

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;
public class DragMe : MonoBehaviour {

    public void Dragging()
    {
        gameObject.transform.position = Input.mousePosition;
    }
}
```

## ■ 14.5 Screen-Klasse

Elemente einer GUI werden normalerweise immer abhängig von der aktuellen Bildschirmgröße positioniert und auch skaliert. Während bei den alten Techniken hierfür noch viel Handarbeit nötig ist, wird beim *uGUI*-System dank der Anchors-Technologie bereits vieles automatisch im Hintergrund gemacht. Trotzdem ist es auch dort manchmal sehr sinnvoll, auf die aktuellen Daten des Bildschirms zuzugreifen.

Hierfür bietet Unity die Klasse `Screen` an. Über deren statische Methoden haben Sie Zugriff auf die Bildschirmhöhe (`height`), die Breite (`width`), die Auflösung (`currentResolution`) und noch einige weitere Informationen.

### 14.5.1 Schriftgröße dem Bildschirm anpassen

Mithilfe der `Screen`-Klassen können Sie z.B. die Schriftgröße von GUI-Objekten anpassen. Das folgende Beispiel zeigt, wie anhand der Bildschirmbreite die Schriftgröße einer *uGUI-Text*-Komponente angepasst wird. Weisen Sie hierfür das Skript einfach einem beliebigen *Text*-GameObject zu, z.B. dem „Label“-Objekt eines Buttons.

**Listing 14.17** Schriftgröße der Bildschirmbreite anpassen

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

public class SetFontSize : MonoBehaviour {
    private Text text;
    void Start () {
        text = GetComponent<Text>();
    }

    void Update () {
        if (Screen.width < 300)
            text.fontSize = 16;
        else
            text.fontSize = 30;
    }
}
```

Das obige Beispiel ist natürlich nur dann sinnvoll, wenn die Bildschirmbreite während des Spiels verändert werden kann, z.B. in einem Browser.

Bei einer starren Auflösung wäre es dagegen besser, die `fontSize`-Zuweisung gleich in der `Start`-Methode durchzuführen. Ein einmaliges Abfragen der Breite und Zuweisen der Größe würden dann reichen und es entstünden nicht unnötig viele Zugriffe in der `Update`-Methode.

# 15

## Prefabs

Mithilfe von *Prefabs* können Sie Vorlagen bzw. Prototypen von *GameObjects* definieren, um von diesen während der Entwicklung und zur Laufzeit beliebig viele Kopien (Instanzen) zu erzeugen. Ein typisches Beispiel hierfür ist das Erstellen von Schussprojektilen oder Gegenständen, die in Szenen platziert werden, z.B. Fackeln, Stühle und Tische.

Ein *Prefab* speichert hierbei alle wichtigen Informationen, die zu diesem *GameObject* gehören, wie Materialien, Texturen, *Collider*, zugehörige Audiodateien, Animationen usw.

### ■ 15.1 Prefabs erstellen und nutzen

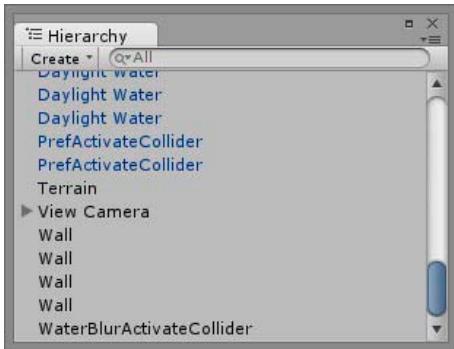
Da *Prefabs* ein fundamentaler Bestandteil von Unity sind, ist auch das Erstellen eines neuen *Prefabs* sehr einfach. Sie können einfach ein beliebiges *GameObject* aus der *Hierarchy* einer Szene per Drag & Drop in den *Project Browser* ziehen und dort fallen lassen. Unity erstellt nun automatisch aus diesem *GameObject* ein neues *Prefab*, das zunächst auch den Namen des *GameObjects* besitzt. Wenn Sie möchten, können Sie aber natürlich auch das *Prefab* nach Belieben umbenennen.

### ■ 15.2 Prefab-Instanzen erzeugen

Möchten Sie nun eine Kopie bzw. eine Instanz eines *Prefabs* erstellen, brauchen Sie einfach nur dieses aus dem *Project Browser* in die Szene hineinziehen. Hierbei wird automatisch eine Instanz des *Prefabs* erstellt, die dann der Szene zugefügt wird. Hierdurch können Sie diesen Vorgang beliebig oft wiederholen und unzählige Instanzen eines *Prefabs* in der Szene platzieren. Das Erstellen von solchen Kopien können Sie natürlich auch über Programmcode realisieren, was im Folgenden noch erklärt wird.

Haben Sie nun (manuell oder via Code) eine Instanz erstellt, können Sie diese Prefab-Instanz in der Szene wieder beliebig verändern. Auch wenn jede Instanz im Hintergrund noch

mit dem Original-*Prefab* verbunden ist, so haben Modifikationen der einzelnen Kopien erst einmal keinen Einfluss auf die eigentliche *Prefab*-Vorlage oder auf die anderen Instanzen des *Prefabs*. Aber zur Unterscheidung einer *Prefab*-Instanz von einem gewöhnlichen *Game-Object* werden diese in der *Hierarchy* trotzdem noch farblich unterschiedlich dargestellt.

**Bild 15.1**

Blaue Darstellung von Prefabs in der Hierarchy

### 15.2.1 Instanzen per Code erstellen

Möchten Sie nun nur Laufzeit eines Spiels Instanzen eines Prefabs erzeugen, müssen Sie hierfür die statische Methode `Instantiate` der `Object`-Klasse nutzen. Da aber jedes `MonoBehaviour`-Skript sowieso am Ende von der Klasse `Object` erbt (siehe Kapitel „Skript-Programmierung“), brauchen Sie einfach nur `Instantiate` aufzurufen. Dieser übergeben Sie nun das eigentlich *Prefab* sowie eine Positionsangabe und eine Drehung, die im speziellen *Quaternion*-Format (siehe Kapitel „Objekte in der dritten Dimension“) angegeben werden muss. Für die Zuweisung des *Prefabs* nutzen wir im folgenden Beispiel den Datentyp `GameObject`.

#### **Listing 15.1** Erzeugen einer Prefab-Instanz

```
using UnityEngine;
using System.Collections;
public class Prefab : MonoBehaviour {
    public GameObject myPrefab;
    void Start () {
        Instantiate(myPrefab,new Vector3(10,0,0),Quaternion.identity);
    }
}
```

### 15.2.2 Instanzen weiter bearbeiten

Häufig müssen die erstellten *Prefab*-Instanzen noch weiter bearbeitet werden. So wird zum Beispiel einem Projektil meistens noch eine Kraft zugefügt, damit es in die gewünschte Richtung fliegt, oder einem Gegner wird sein aktuelles Ziel zugewiesen. Für diese Zwecke besitzt `Instantiate` einen Rückgabewert vom Typ `Object`, das die erzeugte Instanz beinhaltet. Dieses können Sie einer Variablen zuweisen, die Sie dann weiter bearbeiten können.

Im folgenden Beispiel wollen wir der erstellten Instanz einen *Tag* zuweisen, um dieses später hierüber besser identifizieren zu können. Da der Typ *Object* die Variable *tag* allerdings nicht kennt, müssen wir für diesen Zweck den Rückgabewert zunächst in eine *GameObject*-Variable umwandeln. Dies funktioniert, weil das erzeugte *Prefab* nicht nur ein *Object*, sondern auch ein *GameObject* ist.

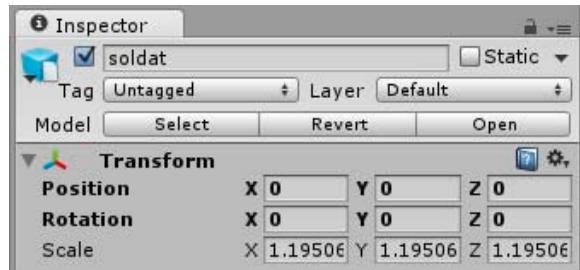
Bitte beachten Sie bei solchen Zuweisungen, dass Sie per Code nur *Tags* zuweisen, die auch unter **Edit/Project Settings/Tags and Layers** definiert wurden. Unity überprüft hier nicht im Vorwege Ihren Code. Und sollten Sie doch am Ende einen *Tag* zuweisen wollen, der nicht existiert, führt dies dann im Spiel zu einem Fehler.

#### **Listing 15.2** Einer Prefab-Instanz einen neuen Tag zuweisen

```
using UnityEngine;
using System.Collections;
public class Prefab : MonoBehaviour {
    public GameObject myPrefab;
    void Start () {
        GameObject go = (GameObject) Instantiate(myPrefab,
            new Vector3(0,0,0),Quaternion.identity);
        go.tag = "Respawn";
    }
}
```

## ■ 15.3 Prefabs ersetzen und zurücksetzen

Sollten Sie eine Instanz eines *Prefabs* erstellt haben, werden Ihnen im *Inspector* dieser Instanz drei zusätzliche Knöpfe angezeigt: *Select*, *Revert* und *Apply*.



**Bild 15.2**  
Inspector-Buttons eines Prefabs

Diese Buttons geben Ihnen folgende Möglichkeiten:

- **Select** springt im *Project Browser* zum Original-*Prefab* dieser Instanz.
- **Revert** verwirft alle Anpassungen an dieser Instanz und stellt den Originalzustand des *Prefabs* wieder her.
- **Apply** überträgt alle Änderungen dieser Instanz auf das Original-*Prefab*. Wollen Sie also, dass der aktuelle Zustand dieser Instanz auf das eigentliche *Prefab* übertragen wird, müssen Sie hierfür diesen Knopf drücken. Beachten Sie hierbei, dass die Änderungen nicht

nur auf das *Prefab* übertragen werden, sondern auch auf alle anderen bereits erstellten Instanzen dieses *Prefabs*.

Sie können auch manuell ein *Prefab* ersetzen. Ziehen Sie ein beliebiges *GameObject* auf ein *Prefab* im *Project Browser*, wird das *Prefab* ebenfalls ersetzt. Auch in diesem Fall werden alle Instanzen in der *Scene View* durch dieses ersetzt.

# 16

# Internet und Datenbanken

Dass Games auf Webinhalte zugreifen, gehört heute zur Normalität. Aber nicht nur Multiplayer-Games, sondern schon die kleinsten Apps legen heutzutage Spielerprofile im Web ab, speichern Spielstände in Online-Highscores oder bieten Community-Funktionalitäten an. Für solche Online-Kommunikationen bietet Unity die Klasse `WWW` an. Sollten Sie bereits ein paar Grundkenntnisse in der Webentwicklung, speziell in PHP oder einer anderen serverseitigen Skriptsprache, besitzen, dann ist dies sicher nicht von Nachteil.

## ■ 16.1 Die WWW-Klasse

Die Kernklasse, um mit anderen Websites zu kommunizieren, ist in Unity die Klasse `WWW`. Ein Aufruf einer Seite ist hierbei kinderleicht. Sie müssen lediglich eine Instanz dieser Klasse erstellen und dieser eine URL mitgeben. Als Rückgabeparameter erhalten Sie dann den Inhalt dieser URL. Da das Übermitteln des Inhalts etwas dauern kann, können Sie mithilfe der `isDone`-Eigenschaft überprüfen, wann der Vorgang abgeschlossen ist. Eine weitere Möglichkeit ist der Aufruf in einer *Coroutine*, die mit `yield` so lange wartet.

**Listing 16.1** Daten einer URL abfragen

```
IEnumerator WebRequest() {
    WWW www = new WWW(url);
    yield return www;
}
```

Wenn Sie eine Mobile-App oder ein Desktop-Game entwickeln, können Sie mit `WWW`-Objekten auf jede beliebige Website zugreifen. Sollten Sie allerdings ein Webplayer-Game erstellen, verhindert eine Sicherheits-Sandbox das Zugreifen auf Seiten, die sich nicht auf dem gleichen Server befinden, wo das Game gehostet wird.

### 16.1.1 Rückgabewert-Formate

Auch wenn Sie auf diese Weise einfach Seiten aufrufen können, um dort Vorgänge anzustoßen, werden Sie doch meistens eher auch Inhalte bzw. Ergebnisse der URL zurückerhalten wollen.

Damit Sie aber nicht jeden Webinhalt erst einmal umständlich in ein bestimmtes Datenformat umwandeln müssen, bietet die `WWW`-Klasse fünf Eigenschaften an, über die Sie die Rückgabewerte in verschiedene Datentypen umwandeln können. Dies sind:

- `audioClip`
- `byte`
- `movie`
- `text`
- `texture`

Da Sie im Normalfall wissen werden, ob Sie nun z.B. einen `String` oder einen `AudioClip` zurück erwarten, brauchen Sie für gewöhnlich einfach nur die dementsprechende Eigenschaft abzufragen, und schon haben Sie den Webinhalt in der passenden Variablen. Mehr zum Auswerten von Rückgabewerten erfahren Sie im Abschnitt „Rückgabewerte parsen“.

Ein Abfragen eines Textes bzw. eines Strings kann damit wie folgt aussehen:

**Listing 16.2** Abfragen eines Textinhaltes aus dem Web

```
private string webString = "";
public IEnumerator LoadData() {
    string loadUrl = "http://www.IhreDomain.de/php/load.php";
    WWW www = new WWW (loadUrl);
    yield return www;
    webString = www.text;
}
```

Ein weiteres Beispiel demonstriert auf einfache Weise, wie Sie die Textur eines `GameObjects` von Ihrem Server aus steuern könnten. Das Skript lädt hierfür eine Textur aus dem Web herunter und weist sie der `mainTexture` des eigenen `Materials` zu. Dieses erhalten Sie über den `Renderer` des `GameObjects`. Wird diese Textur nun auf dem Server ausgewechselt, wird beim nächsten Szenenstart auch das Objekt anders aussehen.

**Listing 16.3** Einfacher Textur-Download

```
using UnityEngine;
using System.Collections;
public class TextureDownload : MonoBehaviour {
    public string url = "http://www.hummelwalker.de/hanser/test.png";
    void Start() {
        StartCoroutine ("DownloadFile");
    }

    IEnumerator DownloadFile() {
        WWW www = new WWW(url);
        yield return www;
        renderer.material.mainTexture = www.texture;
    }
}
```

## 16.1.2 Parameter übergeben

Meistens werden Sie die WWW-Klasse nicht dafür nutzen, um statische HTML-Seiten oder andere festgelegte Inhalte auf-/abzurufen. Sie werden vielmehr auf dynamische Websites oder gar Datenbanken zugreifen wollen. Um aber zum Beispiel mit einer Datenbank zu kommunizieren, werden Sie in den allermeisten Fällen auf Parameter zurückgreifen, um die benötigten Informationen zu spezifizieren.

Zur Übermittlung von Parametern an eine Website gibt es in der Webentwicklung zwei typische Übertragungsmethoden, die von allen wichtigen Skriptsprachen unterstützt werden: *GET* und *POST*. In einer PHP-Datei können Sie zum Beispiel die übermittelten Parameterwerte über die gleichnamigen Arrays, sogenannte *Superglobals*, abfragen:

**Listing 16.4** GET- und POST-Superglobals in PHP

```
$variable1 = $_POST["parametername"];
$variable2 = $_GET["parametername"];
```

Bei *GET* werden die Parameter und deren Werte innerhalb der URL übergeben. Sicher haben Sie schon einmal so unheimlich lange URLs gesehen, wo Fragezeichen und andere Sonderzeichen vorkommen. Das sind die *GET*-Parameter und die dazugehörigen Werte.

Im Gegensatz zu *GET* stehen bei *POST* die Parameter im HTTP-Header und sind damit nicht sichtbar für die Nutzer. Da in Unity die aufgerufenen Websites und deren URLs aber sowieso nicht zu sehen sind, ist dies in unserem Fall erst einmal kein entscheidender Unterschied. Viel wichtiger ist die praktische Nutzung. Während Sie bei der *GET*-Methode eine URL manuell zusammensetzen müssen, ist das Handling mit *POST* dank einer Helferklasse namens *WWWForm* recht komfortabel. Außerdem können Sie via *POST* viel größere Datens Mengen übertragen als mit *GET*, was ebenfalls spannend sein kann, wenn Sie zum Beispiel Bilder, größere Texte oder Ähnliches an den Server senden oder von ihm abfragen möchten.

### 16.1.2.1 WWWForm

*WWWForm* ist die Helferklasse, die für die Übertragung der *POST*-Parameter wichtig ist. Sie speichert die Parameter und deren Werte und wird dann dem *WWW*-Objekt als zusätzlicher Parameter mit übergeben. Die wichtigste Methode einer *WWWForm*-Instanz ist die *AddField*-Methode. Sie erwartet als Erstes den Namen des Parameters und als Zweites den Wert, der als *String* oder *Integer* übergeben werden kann.

**Listing 16.5** Verwenden von Parametern zum Versenden von Daten

```
public IEnumerator SaveData(string name , int points) {
    string saveUrl = "http://www.IhreDomain.de/php/save.php";
    WWWForm form = new WWWForm();
    form.AddField("name",name);
    form.AddField("points",points);
    WWW www = new WWW (saveUrl, form );
    yield return www;
}
```

Wie Sie dem obigen Beispiel schon entnehmen können, steckt in dem C#-Code keinerlei Logik. Was am Ende mit den übertragenen Parametern tatsächlich gemacht wird, entscheidet ganz alleine die PHP-Datei, die das *WWW*-Objekt aufruft.

## ■ 16.2 Datenbank-Kommunikation

Heutzutage nutzen sehr viele Spiele (sowohl aus dem Mobile-, Web- und auch Desktop-Bereich) Online-Datenbanken, wo sie zum Beispiel Spielstände, Highscores oder Spielerdaten abspeichern. Da Sie aber in Unity nicht direkt auf Datenbanken im Web zugreifen können, zumindest nicht mit der *WWW*-Klasse, müssen Sie hier den Umweg über Webseiten machen, die die eigentliche Kommunikation mit der Datenbank vornehmen. Da dies normalerweise mit serverseitigen Skriptsprachen wie PHP umgesetzt wird (ergänzt mit SQL), sind hier zusätzlich Kenntnisse aus diesen Bereichen notwendig. In den nächsten Abschnitten möchte ich Ihnen deshalb erklären, wie Sie eine einfache Datenbankkommunikation zwischen Unity und einer im Internet zugänglichen Datenbank umsetzen können.

Da es verschiedene Möglichkeiten gibt, wie man im Detail so eine Kommunikation aufbauen kann, möchte ich Ihnen im Folgenden eine einfache Variante zeigen, die sich für mich aber als praktisch und vor allem stabil erwiesen hat. Wollen Sie die folgenden Beispiele selber umsetzen, benötigen Sie hierfür dementsprechend Webspace mit PHP sowie eine mySQL-Datenbank.



### SQL

SQL ist eine Datenbanksprache, die zum Erstellen von Datenstrukturen in relationalen Datenbanken sowie zum Bearbeiten und Abfragen deren Daten dient. Je nach Datenbankhersteller unterscheiden sich die Befehle (meist nur) in kleineren Details voneinander.

Da dies weder ein mySQL- noch ein PHP-Buch ist, werde ich mich in den nächsten Abschnitten auf das Wesentliche beschränken und Sicherheitsabfragen, Verschlüsselungen etc. weglassen. Auch werde ich mich nicht bis ins letzte Detail mit den verschiedenen Befehlen beschäftigen, sondern diese nur anhand der Skriptbeispiele erläutern. Für weitergehende Informationen empfehle ich, einen Blick ins Internet zu werfen, wo Sie unzählige Bücher und Einführungs-Tutorials zum Thema PHP und mySQL finden sollten.

### 16.2.1 Daten in einer Datenbank speichern

Als Erstes wollen wir einen Ergebnis-Datensatz in eine Highscore-Tabelle hineinschreiben. Dieser bestehe am einfachsten nur aus einem Namen und der Punktanzahl. Wie so etwas in Unity aussehen könnte, können Sie der Methode *SaveData* aus dem vorherigen Beispiel entnehmen. Dort wird eine Datei namens *save.php* aufgerufen und dieser werden die

zwei Variablen *name* und *points* übergeben. Die folgende PHP-Datei zeigt, wie diese Datei *save.php* aussehen könnte.

**Listing 16.6** PHP-Datei zum Speichern eines Highscores in einer Datenbank

```
<?php
    require_once 'db.inc.php';
    $name = $_POST['name'];
    $points = $_POST['points'];
    $query = "INSERT INTO highscore VALUES (NULL, '$name', $points)";
    $result = mysql_query($query);
?>
```

In der ersten Zeile nach dem PHP-Tag `<?php` wird zunächst einmal eine Verbindung zum Datenbankserver aufgebaut und die Datenbank gewählt. In diesem Fall wurde dies in eine externe Datei namens *db.inc.php* ausgegliedert (siehe Abschnitt „Datenbankverbindung in PHP“).

In den beiden nächsten Zeilen werden die zwei übergebenen Parameter *name* und *points* (die im *WWWForm*-Objekt definiert und übergeben wurden) ausgelesen und eigenen Variablen zugewiesen. Anschließend wird ein *INSERT INTO*-SQL erstellt, der mit den Parameterwerten in der Tabelle *highscore* einen neuen Datensatz anlegen soll. Der Befehl wird dann in der folgenden Zeile mithilfe von *mysql\_query* an die MySQL-Datenbank geschickt. Als Letztes wird schließlich der PHP-Bereich mit dem Tag `?>` wieder geschlossen.

Beachten Sie, dass in der Tabelle *highscore* natürlich nur dann neue Datensätze angelegt werden können, wenn sie vorher auch in der Datenbank erstellt wurde. Eine solche Tabelle kann mit einem SQL-Befehl angelegt werden, der z. B. direkt über das SQL-Eingabefenster der Datenbank-Administrationskonsole ausgeführt wird (z. B. phpMyAdmin). Der SQL-Befehl könnte dann so aussehen:

**Listing 16.7** SQL zum Erzeugen einer Highscore-Tabelle

```
CREATE TABLE highscore (
    id MEDIUMINT NOT NULL AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    points INT NOT NULL,
    PRIMARY KEY (id)
);
```

Der *CREATE TABLE*-SQL erstellt eine Tabelle *highscore*, die aus drei Spalten besteht. Die erste Spalte wird aufgrund von *AUTO\_INCREMENT* automatisch befüllt, sobald ein neuer Datensatz erzeugt wird. Das ist auch der Grund, weshalb beim vorherigen *INSERT INTO*-SQL der erste Parameter *NULL* ist.

## 16.2.2 Daten von einer Datenbank abfragen

Als Nächstes wollen wir nun die besten fünf Ergebnisse aus der Highscore-Tabelle abfragen, z. B. um diese in der GUI des Spiels anzuzeigen. Da wir aber bei einer Abfrage auch nur einen Rückgabewert zurück erhalten können (in diesem Fall einen Text), müssen wir nun die fünf Ergebnis-Datensätze in diesem einen Text verpacken.

Hier gibt es natürlich viele Möglichkeiten, wie man dabei vorgehen kann. So können Sie die Daten z. B. in einem XML-Format verpacken und dieses XML an Unity übertragen. Oder Sie verpacken die Daten in einem CSV-Format und übertragen das. Egal welche Variante Sie am Ende wählen, wichtig ist nur, dass Sie das Format in Unity wieder entschlüsseln können, um aus diesem den einzelnen Daten wieder zu erhalten.

In diesem Beispiel wollen wir einfach alle Werte der Datensätze hintereinander auflisten, die lediglich von einem Sonderzeichen (in diesem Fall ist es ein Pipe, also ein vertikaler Strich) getrennt werden. Hierbei ist es natürlich extrem wichtig, dass dieses Trennzeichen nicht in den Werten selber vorkommt. In diesem Fall betrifft dies nur den Namen, da in der Punktzahl logischerweise kein Pipe vorkommen kann. In der Praxis empfiehlt es sich deshalb, bereits bei der Namenswahl des Nutzers darauf zu achten, dass dieser das Trennzeichen nicht wählen kann (z. B. durch Zeichenersetzung mit der Replace-Methode, die jede String-Variable besitzt).

Um Daten nun von einer SQL-Datenbank abzufragen, benötigen Sie ein *SELECT*-Statement. Wie dies für einen Highscore, gepaart mit der Formatierung des Rückgabewertes, aussehen kann, zeigt der folgende PHP-Code.

#### **Listing 16.8** PHP-Datei zum Abfragen eines Highscores

```
<?php
require_once 'db.inc.php';
$query = "SELECT name, points FROM highscore ORDER BY points DESC LIMIT 0, 5";
$result = mysql_query($query) or die('Query failed: ' . mysql_error());

$num_results = mysql_num_rows($result);
$return = "";
for($i = 0; $i < $num_results; $i++)
{
    $row = mysql_fetch_array($result);
    $return = $return . $row['name'] . "|" . $row['points'] . "|";
}
echo $return;
?>
```

Auch bei diesem PHP-Skript wird zunächst die Datenverbindung mithilfe der Datei *db.inc.php* aufgebaut. Danach wird ein *SELECT*-SQL zusammengesetzt, der dann der Variablen *\$query* zugewiesen wird. Dieses SQL-Statement sortiert nach Punkten absteigend die Datensätze und holt von diesen dann die Werte der Spalten *name* und *points* der ersten fünf.

Im zweiten Schritt wird der SQL-Befehl mit *mysql\_query* ausgeführt und das Ergebnis der Variablen *\$result* zugewiesen. Danach wird mit *mysql\_num\_rows* festgestellt, wie viele Datensätze tatsächlich zurückgegeben wurden.

In einer For-Schleife werden nun alle Datensätze durchlaufen, um die einzelnen Werte zu einem gemeinsamen String zusammenzusetzen. Dabei werden von jedem Satz die Inhalte der Spalten *name* und *points* genommen und mit einem Pipe-Zeichen getrennt an den String angehängt. Am Ende könnte ein String wie der folgende herauskommen.

#### **Listing 16.9** Beispiel eines formatierten Rückgabe-Strings

```
$return = "Conny|122|John|102|Steve|89|Kim|78|Michael|59|"
```

Um nun die Daten des Highscores in ein Unity-Projekt zu holen, müsste diese PHP-Datei nur noch über einen *WWW*-Aufruf, wie in der *LoadData*-Methode vom Anfang dieses Kapitels, aufgerufen werden. In der String-Variablen *webstring* stünde dann der Inhalt des zusammengesetzten Rückgabe-Strings (siehe Listing 16.9).

Jetzt haben Sie zwar den aktuellen Highscore in Unity, nur können Sie mit diesem aktuell nicht viel anfangen. Für eine vernünftige Weiterverarbeitung müssen diese Daten jetzt erst einmal in ein nutzbares Format umgewandelt werden (*Parsen* genannt).

### 16.2.3 Rückgabewerte parsen

Wie im vorherigen Abschnitt werden Sie meistens als Rückgabewert einen Text erhalten. Auch wenn im einfachsten Fall dieser Text nur aus einer einzigen Information, zum Beispiel dem Spielernamen, besteht, wird dies meistens eher selten der Fall sein. Meist werden Sie, wie im vorherigen Beispiel, mit Texten zu tun haben, in denen mehrere Informationen enthalten sind. Um nun aus diesem Text die einzelnen Informationen wieder zu erhalten, müssen Sie diesen Rückgabe-String parsen, das heißt ihn zerlegen und in ein anderes, nutzbares Format umwandeln.

Im obigen Beispiel hatten wir zum Trennen dieser Informationen ein Pipe-Zeichen genutzt. Um nun die einzelnen Daten wieder zu erhalten, müssen Sie diesen Rückgabewert überall dort auseinandernehmen, wo ein Pipe vorkommt, und diese Teilstrings am besten auf unterschiedliche Variablen verteilen. Für dieses Vorgehen eignet sich die *Split*-Methode, die jede *String*-Variable besitzt. Diese erwartet ein oder mehrere *Character*, also einzelne Zeichen, die den Trenner darstellen. Anhand dessen zerteilt dann die Methode den String und gibt schließlich ein *String*-Array mit diesen Teil-Strings zurück.

In dem Beispiel von oben müssen Sie allerdings bedenken, dass auch am Ende des Übergabe-Strings ein Pipe-Zeichen steht. Da dieses Zeichen aber kein Trenner zu einer weiteren Information darstellt, müssen Sie dieses Zeichen zunächst einmal löschen. Hierfür eignet sich die Methode *Remove*. Damit sieht das komplette Parsen wie folgt aus:

**Listing 16.10** Parsen eines Rückgabe-Strings

```
webString = webString.Remove(webString.Length-1);
char c = '|';
string[] highscoreArray;
highscoreArray = webString.Split(c);
```

Beachten Sie bei der Zuweisung der Char-Variablen, dass die Angabe des Pipe-Zeichens in einfachen Hochkommas vorgenommen wird, nicht in doppelten.

Theoretisch könnten Sie jetzt den Highscore anzeigen. Dabei müssten Sie dieses Array durchlaufen und abwechselnd die Elemente als Name und als Points-Wert in der GUI anzeigen. Da dieses Vorgehen aber nicht unbedingt komfortabel ist, werden für solche zusammengehörigen Werte häufig eigene Datentypen erstellt, die das Arbeiten mit den Informationen wesentlich vereinfachen.

#### 16.2.4 Datenhaltung in eigenen Datentypen

Wenn Sie aus einer Datenbank zusammengehörige Daten erhalten, macht es meistens Sinn, diese in gemeinsamen Objekten abzulegen. Da es aber wohl eher selten passende Objekte bereits hierfür gibt, sollten Sie hierfür eigene Datentypen erstellen. So könnten Sie eine Klasse erstellen, die für jeden Wert eines Datensatzes eine Eigenschaft (oder eine Public-Variable) besitzt. Haben Sie mehrere Datensätze, erzeugen Sie eben mehrere Objekte dieser Klasse. Diese können Sie anschließend wieder in einem Array, z.B. in einem List-Objekt (siehe Kapitel „C# und Unity“), wieder zusammenfassen.

Für das Highscore-Beispiel würde es bedeuten, dass Sie zunächst eine Klasse namens Score programmieren, die nichts anderes macht, als für jeden Datensatz den Name- und den Points-Wert zu speichern. Im folgenden Beispiel werden wir sogenannte „automatisch implementierte Eigenschaften“ nutzen. Das ist nichts anderes als eine vereinfachte Schreibform der normalen Eigenschaften aus dem Kapitel „C# und Unity“. Diese Form können Sie immer dann nutzen, wenn im Get- und Set-Accessor kein Extra-Code benötigt wird.

**Listing 16.11** Speicherklasse eines Highscore-Datensatzes

```
public class Score{
    public string PlayerName {get;set;}
    public string Points {get;set;}
}
```

Beachten Sie, dass die Klasse Score kein *MonoBehaviour*-Skript ist. Es erbt also nicht von *MonoBehaviour* und könnte dementsprechend auch nicht einem *GameObject* als Komponente zugewiesen werden.

Da wir nun für jeden Highscore-Datensatz eine eigene Score-Instanz benötigen, speichern wir nun anschließend alle Instanzen in einem gemeinsamen Array-Objekt. C# bietet hierfür den generischen Datentyp *List* an, den Sie bereits im Kapitel „C# und Unity“ kennengelernt haben. Um ein List-Objekt erzeugen zu können, müssen Sie zunächst den Namespace mit `using System.Collections.Generic` am Anfang des Klassenskriptes einbinden, in dem Sie ein *List*-Objekt erstellen wollen. Danach können Sie die List-Variable anlegen. In dem Beispiel nutzen wir eine Public-Variable (eine Private reicht aber je nach Weiterverwendung auch), sodass die Deklaration wie folgt aussieht:

**Listing 16.12** List-Objekt für Highscore-Datensätze

```
public List<Score> highscore = new List<Score>();
```

Jetzt können Sie das String-Array auf verschiedene Score-Objekte verteilen, die Sie dann schließlich dem List-Objekt zufügen. Hierfür erzeugen wir eine neue Methode namens *FillHighscore*, der Sie dann das String-Array übergeben. *FillHighscore* durchläuft nun das Array in einem Zweier-Schritt und erzeugt jedes Mal ein neues Score-Objekt. Diesem wird das aktuelle Array-Item der Eigenschaft *PlayerName* und das nächste Item der *Points*-Eigenschaft zugewiesen. Anschließend wird das Score-Objekt über die Methode *Add* dem List-Objekt *highscore* zugefügt.

**Listing 16.13** FillHighscore-Methode

```
void FillHighscore(string[] input) {
    highscore.Clear();
    int length = input.Length;

    for (int i=0; i < length; i+=2)
    {
        Score score = new Score();
        score.PlayerName = input[i];
        score.Points = input[i+1];
        highscore.Add(score);
    }
}
```

Beachten Sie, dass in der Methode vor dem Durchlaufen des *String*-Arrays erst einmal das List-Objekt mit der Methode *Clear* geleert wird. Dies dient dazu, dass bei mehrmaligem Aufrufen der *FillHighscore* Methode keine Datensätze doppelt vorhanden sind.

Nehmen Sie nun als Grundlage für das Holen der Highscore-Daten die Methode *LoadData* vom Anfang dieses Kapitels, könnte *LoadData* dann wie folgt aussehen (die PHP-Datei aus Listing 16.8 müsste dann natürlich *load.php* heißen):

**Listing 16.14** Methode zum Holen eines Highscores

```
public IEnumerator LoadData() {
    string loadUrl = "http://www.IhreDomain.de/php/load.php";
    WWW www = new WWW(loadUrl);
    yield return www;
    webString = www.text;
    webString = webString.Remove(webString.Length-1);
    char c = '|';
    string[] highscoreArray;
    highscoreArray = webString.Split(c);
    FillHighscore (highscoreArray);
}
```

Jetzt muss die Methode *LoadData* nur noch mit *StartCoroutine* aufgerufen werden und dann können Sie mit diesen Daten über die Variable *highscore* nach Belieben weiterarbeiten.

Um zu demonstrieren, wie diese Datentypen das Weiterverarbeiten vereinfachen, wollen wir nun noch den C#-Code um ein einfaches Darstellen der Highscore-Daten in der *OnGUI*-Methode erweitern. Mithilfe der *List*- und der *Score*-Klasse ist das nun kinderleicht.

**Listing 16.15** Darstellung des Highscores in OnGUI

```
void OnGUI()
{
    foreach(Score s in highscore)
    {
        GUILayout.Label(s.PlayerName + " " + s.Points);
    }
}
```

### 16.2.5 HighscoreCommunication.cs

Das Skript, das die gesamte Datenbankkommunikation für den Highscore übernimmt, sieht schließlich dann wie folgt aus:

**Listing 16.16** Fertiges Skript HighscoreCommunication.cs

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

public class HighscoreCommunication : MonoBehaviour {
    public List<Score> highscore = new List<Score>();
    private string webString = "";

    void Start()
    {
        //StartCoroutine(SaveData ("Test",23)); //Beispieldaten schreiben
        StartCoroutine(LoadData()); //Beispielaufruf des Highscores
    }
    //Daten laden
    public IEnumerator LoadData() {
        string loadUrl = "http://www.IhreDomain.de/php/load.php";
        WWW www = new WWW (loadUrl);
        yield return www;
        webString = www.text;
        webString = webString.Remove(webString.Length-1);
        char c = '|';
        string[] highscoreArray;
        highscoreArray = webString.Split(c);
        FillHighscore (highscoreArray);
    }
    //Daten speichern
    public IEnumerator SaveData(string name , int points) {
        string saveUrl = "http://www.IhreDomain.de/php/save.php";
        WWWForm form = new WWWForm();
        form.AddField("name",name);
        form.AddField("points",points);
        WWW www = new WWW (saveUrl, form );
        yield return www;
    }
    //Daten-Objekte fuellen
    void FillHighscore(string[] input) {
        highscore.Clear();
        int length = input.Length;
        for (int i=0; i < length; i+=2)
        {
            Score score = new Score();
            score.PlayerName = input[i];
            score.Points =input[i+1];
            highscore.Add(score);
        }
    }
    //Daten beispielhaft darstellen
    void OnGUI()
    {
        foreach(Score s in highscore)
```

```
        {
            GUILayout.Label(s.PlayerName + " " + s.Points);
        }
    }
}
```

Um einen Datensatz nun in die Datenbank zu schreiben, muss `SaveData` aufgerufen werden, dargestellt durch den auskommentierten Bereich in der `Start`-Methode.

## 16.2.6 Datenbankverbindung in PHP

Um das Beispiel abzuschließen, möchte ich Ihnen noch die Datei `db.inc.php` vorstellen, die in den PHP-Beispielen für das Aufbauen der Datenverbindung zuständig ist. Sie baut zunächst eine Verbindung mit dem Server auf und legt danach die Datenbank fest, mit der kommuniziert werden soll.

### **Listing 16.17** Datenbankverbindung aufbauen

```
<?php
    require_once 'config.inc.php';
    // Oeffne Verbindung zum Datenbankserver
    mysql_connect( DB_HOST, DB_USER, DB_PASS )
        or die( 'Konnte keine Verbindung herstellen : ' . mysql_error() );
    // waehle Datenbank
    mysql_select_db( DB_NAME )
        or die( 'Kann DB "' . DB_NAME . '" nicht auswählen: ' . mysql_error() );
?>
```

Auch diese Datei nutzt wieder eine zusätzliche Datei. In der Datei `config.inc.php` werden die vier Konstanten `DB_HOST`, `DB_USER`, `DB_PASS` und `DB_NAME` definiert, die schließlich in dieser Datei für den Verbindungsaufbau notwendig sind.

Zur Vollstndigkeit sehen Sie hier nun noch die Datei *config.inc.php*, die die Variablen mit Werten fllt. Beachten Sie hier, dass die Angaben 'Benutzer', 'Passwort' und 'MeineDB' natrlich nur als Platzhalter fr die eigentlichen Daten Ihres Datenbankservers dienen.

**Listing 16.18** Verbindungsdaten statischen Variablen zuweisen

```
<?php
    // Verbindungsdaten für die Datenbank
    /** Hostname des Datenbankservers */
    define( 'DB_HOST', 'localhost' );
    /** Benutzername, mit dem auf die Datenbank zugegriffen wird */
    define( 'DB_USER', 'Benutzer' );
    /** Passwort des verwendeten Benutzers */
    define( 'DB_PASS', 'Passwort' );
    /** Name der zu verwendenden Datenbank */
    define( 'DB_NAME', 'MeineDB' );
?>
```

Durch das Auslagern dieser Verbindungsdaten sind Sie flexibler. Ein Ändern dieser Daten ist viel einfacher, als wenn Sie diese in jeder Datei ändern müssten. Ein Umzug des Projektes von einem Testsystem auf ein Produktivsystem ist beispielsweise dadurch sehr einfach.

Lizenziert für gregwyss@hotmail.com.

© 2014 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

# 17

## Animationen

Die Veredelung eines Spiels sind gut animierte 3D-Modelle. Dies kann von einfachen Animationen wie dem Drehen von Windmühlenflügeln bis hin zu komplexen Bewegungsabläufen eines Humanoiden reichen, die mit Umgebungsobjekten wie Wänden und Untergründen interagieren.



**Bild 17.1** Animierte 3D-Modell eines Soldaten

Da dies ein recht großes Feld ist, gibt es in der Spieleindustrie extra Animationsspezialisten, die sich ausschließlich mit diesem Themengebiet beschäftigen. Dementsprechend besitzt auch Unity hier eine umfangreiche Palette an Möglichkeiten, Animationen zu implementieren und zu steuern. Da ein Beleuchten aller Features und Möglichkeiten hier den Rahmen sprengen würde, werde ich mich im Folgenden auf Grundlagen und Kernfunktionen beschränken.

## ■ 17.1 Allgemeiner Animation-Workflow

Um in Unity Animationen nutzen zu können, sind mehrere Schritte erforderlich. Diese können in die folgenden Schritte unterteilt werden:

- 1. Erstellen der Animation:** Zunächst müssen Sie die eigentliche Animation erstellen. Dies kann in Unity geschehen, in Modelling-Tools wie Blender, Maja etc. oder mit *Motion Capturing*-Verfahren (kurz MoCap), die unter anderem beim Film verwendet werden. Innerhalb von Unity wird jede Animation als *Animation-Clip* dargestellt.
- 2. Charakter Import Settings:** Beim Import des zu animierenden 3D-Modells müssen Sie die richtigen *Rig-* und *Animations*-Einstellungen wählen. Bei der Wahl *Humanoid* definieren Sie über den Button *Configure* den Avatar, der für humanoide Modelle das Skelett und die Muskelpartien des Modells beschreibt und testet.
- 3. Animation-Clip-Einstellungen:** Je nach Art der Erstellung und Verwendung der Animation muss die Animation bzw. der *Animation-Clip* im *Inspector* parametrisiert werden. Bei *Animation-Clips*, die sich in einem FBX-Container befinden, wird dies über den *Animations*-Reiter der *Import Settings* gemacht.
- 4. Animator Controller erstellen:** Als Nächstes benötigen Sie einen *Animator Controller*. Dieser definiert verschiedene *Animation States*, also Zustände, die das animierte Objekt annehmen kann, und legt fest, welche *Animation-Clips* in diesen abgespielt werden. Außerdem werden dort Parameter definiert, die bestimmen, wann ein Zustand in einen anderen wechselt.
- 5. Controller-Skripte programmieren:** Als letzter Schritt werden die eigentlichen Skripte programmiert, die die Parameter des *Animator Controllers* setzen, die wiederum dafür sorgen, dass der *Animator Controller* in den richtigen Zustand wechselt.

## ■ 17.2 Animationen erstellen

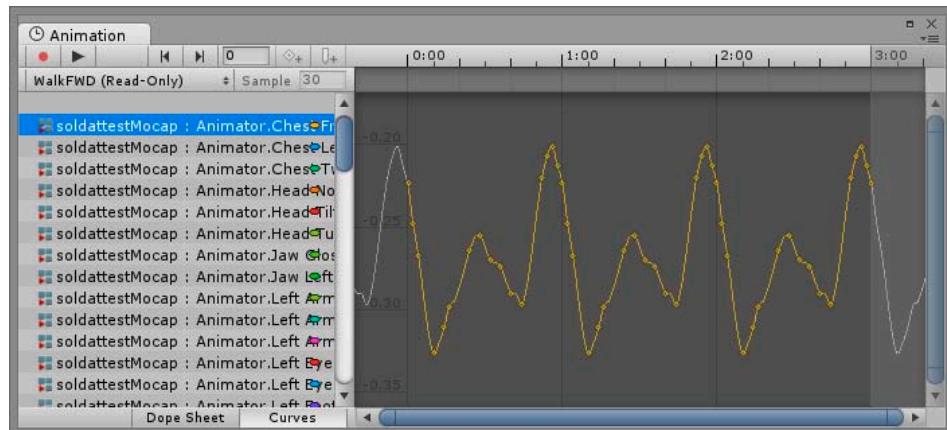
Das Erstellen von Animationen kann je nach Zielobjekt ein sehr komplexes Thema sein. Bei großen Produktionen kommen hierfür manchmal auch *Motion Capturing*-Verfahren und Schauspieler zum Einsatz. Wollen Sie aber nur einfache Vorgänge animieren, wie z.B. das Hochfahren eines Gatters, dann können Sie dies auch direkt in Unity machen.

Hierbei können Sie aber nicht nur Positionen und Formen eines Objektes ändern, Sie können fast alle möglichen Komponenten und deren Eigenschaften animieren. Selbst *Collider*-Eigenschaften wie *Is Trigger* oder auch die Materialfarbe des *Mesh Renderers* sind hierüber leicht steuerbar. Beachten Sie dabei aber, dass Objekte mit *Collidern*, deren *Transform*-Eigenschaften Sie animieren, ein *Rigidbody* besitzen sollten (siehe Kapitel „Physik in Unity“).

## 17.2.1 Animation View

Über das Menü **Window/Animation** öffnen Sie den *Animation-Editor*, mit dem Sie Animationen aufzeichnen und verändern können.

Eine Animation wird in Unity durch *Keyframes* (Schlüsselpositionen) dargestellt. Ein *Keyframe* besteht aus einem Zeitpunkt auf der Zeitachse und einem Wert der zu animierenden Eigenschaft, z.B. der Position oder der Farbe. Bei herkömmlichen *Keyframe*-Animationen würden nun alle Werte zwischen den definierten Zeitpunkten interpoliert, also rechnerisch ermittelt werden. Unity bietet hier aber noch die Möglichkeit, diese Übergänge mithilfe von Kurven zu beschreiben. Den Editor können Sie sowohl für 3D- als auch für 2D-Animationen nutzen, wo dann auch noch sogenannte *Sprite-Animationen* (siehe Abschnitt „Sprite-Animationen“) einsetzbar sind.



**Bild 17.2** Animationskurve eines animierten Parameters im Animation-Editor

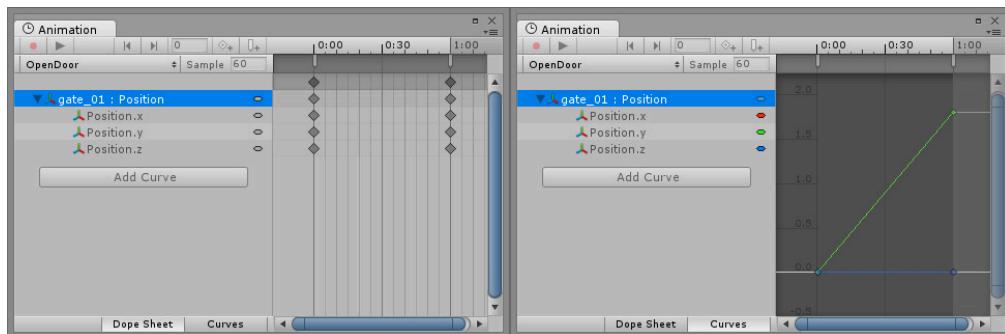
Um Animationen eines Objektes aufzunehmen oder zu bearbeiten, müssen Sie zunächst das jeweilige Objekt in der *Scene View* bzw. in der *Hierarchy* markieren. Dann können Sie folgende Funktionen im *Animation View* dementsprechend nutzen:

- **Aufnahme-Knopf** startet und beendet eine Animationsaufnahme.
- **Start-Knopf** startet das Abspielen einer Animation.
- **Vorheriger Keyframe** springt zurück zur vorherigen Schlüsselposition.
- **Nächster Keyframe** springt vor zur nächsten Schlüsselposition.
- **Add Keyframe** erzeugt eine neue Schlüsselposition.
- **Add Event** erzeugt an der Position der roten Keyframe-Linie ein Ereignis, das eine beliebige Methode aus einem Skript mit einem Parameter aufruft. Beachten Sie, dass nur Methoden aufgerufen werden können, die sich in Skripten befinden, die dem animierten *GameObject* angehängt wurden. Mehr zu diesem Thema erfahren Sie im Kapitel „Animation Events“.
- **Animation-Auswahlbox** wählt die zu bearbeitende Animation aus und erzeugt über *Create New Clip* einen neuen *Animation-Clip*.

- **Sample** definiert die aufzuzeichnenden *Frames pro Sekunde* der Animation. Durch ein nachträgliches Verändern dieses Wertes verändert sich die Geschwindigkeit der aufgezeichneten Animation.

### 17.2.2 Curves vs. Dope Sheet

Über zwei Knöpfe am unteren Rand der *Animation View* können Sie in der Darstellung zwischen *Dope Sheet* auf *Curves* wechseln. In beiden Ansichten sehen Sie links die zu animierenden Eigenschaften sowie, im aufgeklappten Zustand, die aktuellen Animationswerte. Der Unterschied ist auf der Zeitachse zu sehen. Während *Curves* die klassische Darstellung der Animationsverläufe inklusive der Kurvendarstellung anzeigt, ist die *Dope Sheet*-Ansicht eine vereinfachte Darstellung der Animation, die lediglich die *Keys* auf der Zeitachse anzeigen.



**Bild 17.3** Animationsdarstellungen als Dope Sheet und Curves

Gerade bei *Sprite-Animationen*, wo es normalerweise gar keine Zwischenwerte/-zustände zwischen den Einzelbildern gibt, macht diese Darstellung besonders viel Sinn (siehe Abschnitt „Sprite-Animationen“). Sie kann aber auch bei 3D-Animationen genommen werden, um z. B. bei komplexeren Animationen eine übersichtlichere Ansicht zu erhalten.

### 17.2.3 Animationsaufnahme

Beim Erstellen einer Animation gehen Sie wie folgt vor:

1. Selektieren Sie das zu animierende Objekt.
2. Wählen Sie in der Auswahlbox **Create New Clip** und legen Sie einen Dateipfad sowie einen Animationsnamen fest.
3. Drücken Sie den Button **Add Curve** und wählen Sie anschließend die Komponente sowie die zu animierende Eigenschaft aus.
4. Falls noch nicht die Aufnahmefunktion automatisch aktiviert wurde, drücken Sie nun auf den Aufnahme-Knopf. Ist diese bereits aktiv, wird der Knopf rot dargestellt.
5. Positionieren Sie die rote *Keyframe*-Linie auf der Zeitachse an den richtigen Punkt, wo Sie einen neuen *Keyframe* erzeugen möchten, und drücken Sie **Add Keyframe**.

6. Legen Sie den Animationswert fest. Dies können Sie über die Zahlenwerte im *Animation-Fenster* oder manuell über die Kurvenansicht (*Curves*, siehe „Curves vs. Dope Sheet“) machen. Sie können aber auch direkt die Eigenschaft in der *Scene View* oder im *Inspector* verändern. Beim Animieren der Position können Sie z.B. das Objekt an die Stelle hinziehen, wo es hingeschoben werden soll. Ein Sonderfall sind die bereits angesprochenen *Sprite-Animationen*. Ziehen Sie hier einfach ein Einzelbild des zu animierenden *Sprite-Assets* auf den *Keyframe* (Details hierzu siehe Abschnitt „Sprite-Animationen“).
7. Bei mehreren *Keyframes* können Sie nun noch über die *Curve*-Ansicht den Kurvenverlauf anpassen. Hierfür drücken Sie mit der rechten Maustaste auf die *Keyframes* der einzelnen Kurven, um die Steigungen dieser zu verändern. Außerdem können Sie über die Funktion **Add Key** im Kontextmenü der rechten Maustaste noch zusätzliche Zwischenpunkte zufügen.
8. Um die Animationsaufnahme abzuschließen, drücken Sie noch einmal auf den Aufnahme-Knopf. Ist sie beendet, wird der Knopf wieder normal dargestellt.
9. Um die Animation zu testen, können Sie diese nun über den Start-Knopf starten.

Beim Animieren von *Transform*-Eigenschaften, wie der Position, sollten Sie beachten, dass die Veränderungen (z.B. Bewegungen) immer relativ zum Elternobjekt aufgezeichnet werden. Objekte der obersten Ebene werden deshalb immer relativ zum Weltsystem animiert. Für eine Positionsanimation bedeutet dies, dass das *GameObject* beim Animationsstart immer auf die Originalposition zurückgesetzt wird, an der es ursprünglich animiert wurde.

Um dieses Problem zu beheben, können Sie einfach ein *EmptyObject* erzeugen, das als Container-Objekt für das zu animierende *GameObject* dient (siehe Abschnitt 17.2.4, „Beispiel Fallgatter-Animation“). In diesem Fall wird dann das *GameObject* relativ zum *EmptyObject* animiert, sodass das gesamte Container-Konstrukt nun problemlos verschoben werden kann.

### 17.2.3.1 Ansicht im Animation-Fenster anpassen

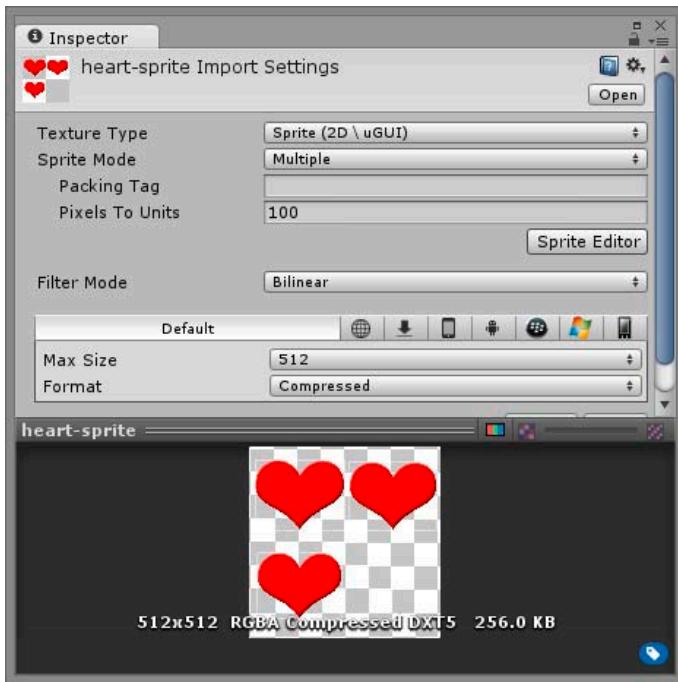
Durch Drücken der **[Strg]**-Taste und das Drehen am Mausrad können Sie die Ansicht auf der X-Achse (Zeitachse) skalieren. Wenn Sie die **[Umsch]**-Taste drücken und das Mausrad drehen, skalieren Sie die Y-Achse.

### 17.2.3.2 Sprite-Animationen

Ein Sonderfall im Bereich der Animationen sind *Sprite-Animationen*. Auch wenn bei 2D-Spielen die Objekte ebenfalls rotiert und verschoben werden können, werden die eigentlichen Bewegungsabläufe, wie z.B. das Schwingen einer Waffe oder die Laufanimation, normalerweise mit sogenannten *Sprite-Animationen* dargestellt. Stellen Sie sich dies wie ein Daumenkino vor, wo die Zustände einer Bewegung als einzelne Bilder gezeichnet und diese dann schnell nacheinander abgespielt werden.

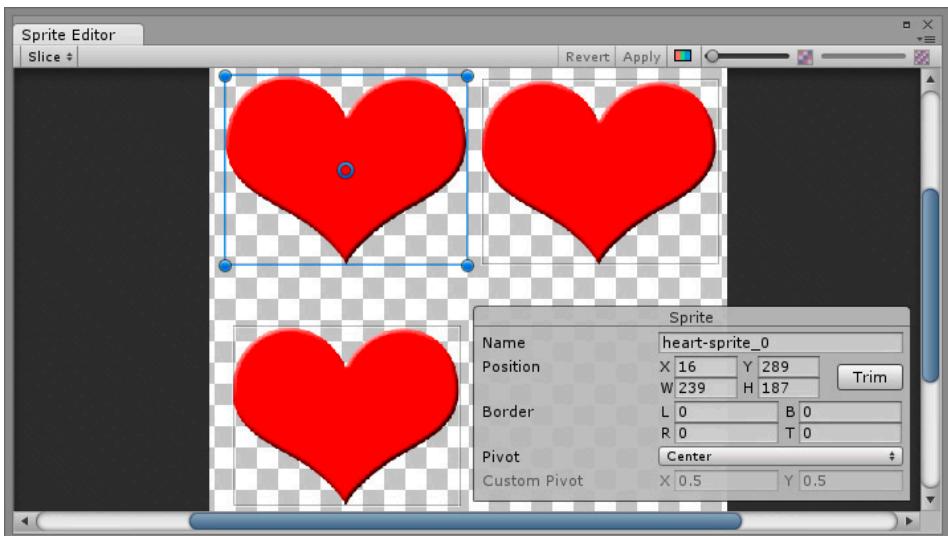
Für diesen Animationstyp gibt es das *Texture Type Sprite*, das Sie in den *Import Settings* einer Textur zuweisen können. Achten Sie außerdem auf eine korrekte Einstellung in *Max Size*, da ansonsten die Bilder unscharf aussehen könnten.

Haben Sie nun innerhalb einer Grafik mehrere Teilbilder (wie z.B. in Bild 17.4), dann können Sie diese mithilfe des *Sprite Modes Multiple* aufteilen. Hierfür nutzen Sie den *Sprite Editor* (ebenfalls in den *Import Settings* zu finden), mit dem Sie eingrenzen können, welche

**Bild 17.4**

Import Settings einer Sprite-Grafik

Bereiche dieser Grafik jeweils ein Einzelbild darstellen. Dies können Sie über den **Slice**-Button machen, der verschiedene Einstellmöglichkeiten erlaubt (siehe Bild 17.5).

**Bild 17.5** Einzelbilder mit dem Sprite Editor definieren

Wählen Sie dort den **Type Automatic** und starten Sie den Vorgang über den Button **Slice**. Es werden nun die Einzelgrafiken dieser Bilddatei automatisch aufgeteilt und mithilfe von Ras-

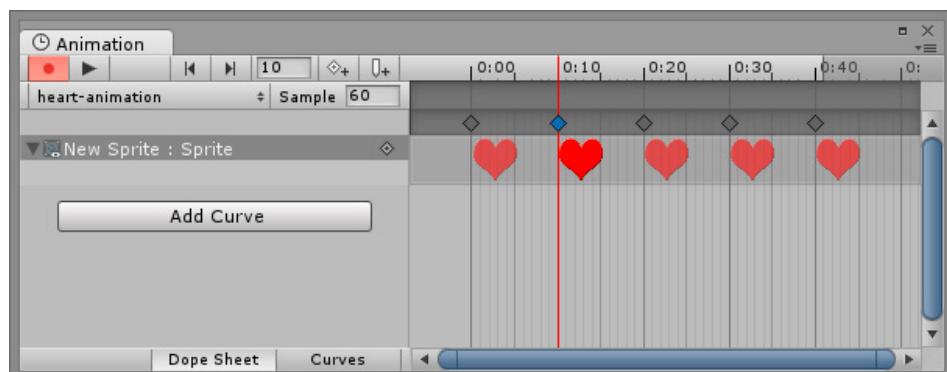
tern dargestellt. Mit einem Klick auf eine solche Rasterlinie können Sie diese natürlich auch noch nachjustieren und die Eingrenzungen ändern.

Eine weitere Möglichkeit neben *Automatic* ist **Grid**. Hierbei wird die Bilddatei in ein Gitternetz aufgeteilt. Dies ist besonders dann sinnvoll, wenn Sie die Teilbilder bereits beim Erstellen nach solch einem Gitter angeordnet haben. Die Gittergröße bestimmen Sie über die *Pixel Size*-Werte.

Nach dem Schließen wird Ihre Bilddatei als neues *Sprite-Asset* im *Project Browser* dargestellt. Ein Klick auf dieses Asset zeigt Ihnen alle Einzelbilder, die Sie mit dem *Sprite Editor* vorher gekennzeichnet haben.

Um jetzt daraus eine *Sprite-Animation* zu erstellen, erzeugen Sie als Nächstes über **GameObject/Create Other/Sprite** ein *Sprite-GameObject*. Dieses besitzt einen *Sprite Renderer*, dessen *Sprite*-Eigenschaft Sie Ihrem neuen *Sprite-Asset* zuweisen.

Danach erstellen Sie im *Animation-Fenster* eine neue Animation und wählen Sie über **Add Curve** (siehe Animationsaufnahme) die Eigenschaft *Sprite* des *Sprite Renderers* aus. Als Vorschlag werden zwei *Keyframes* automatisch angelegt, denen beide das erste Teilbild des Sprites zugewiesen wurde. Für unser Beispiel mit den drei Herzen verschieben Sie den rechten *Keyframe* auf die Position 0:40 und fügen per Drag & Drop das zweite Teilbild auf die Position 0:10 der Timeline hinzu. Durch das Fallenlassen der Grafik wird automatisch ein neuer *Keyframe* angelegt. Dies wiederholen Sie mit der gleichen Grafik auf der Position 0:30 sowie mit der dritten Herzgrafik, also der kleinsten, auf der Position 0:20 (siehe Bild 17.6).



**Bild 17.6** Sprite-Animation eines pochenden Herzens

Starten Sie nun die Animation, werden die Teilbilder der Reihe nach abgespielt. Unser Beispiel würde dann wie ein pochendes Herz wirken. Für eine bessere Darstellung können Sie jetzt in den 2D-Modus wechseln, den Sie über den Button **2D** in der *Scene View*-Toolbar aktivieren können.



#### Beispiel auf der DVD

Das komplette Sprite-Animationsbeispiel finden Sie auf der beiliegenden DVD unter dem Namen „SpriteAnimationExample“.

## 17.2.4 Beispiel Fallgatter-Animation

Als Anschauungsobjekt einer Animation möchte ich ein Fallgatter nehmen, das wir auch später in dem Kapitel „Beispiel-Game“ einsetzen werden. Dieses Objekt wollen wir nun animieren. Wenn Sie sich das Projekt von der DVD auf Ihre Festplatte kopiert haben, können Sie dieses starten. Dort finden Sie ein Fallgatter mit dem Namen „gate\_01“. Für dieses wollen wir eine Animation erzeugen, die das Tor anhebt und später über den *Animator Controller* auch wieder absenkt.

1. Legen Sie im *Project Browser* einen Ordner „Animations“ an.
2. Ziehen Sie das Objekt „gate\_01“ in Ihre Szene.
3. Erzeugen Sie über **GameObject/Create Empty** ein neues *EmptyObject* und nennen Sie dieses in „Gate“ um. Ziehen Sie nun „gate\_01“ auf dieses, sodass „Gate“ dessen Elternobjekt ist.
4. Positionieren Sie „gate\_01“ im Zentrum von „Gate“. Selektieren Sie hierfür „gate\_01“, gehen auf das Zahnrad im *Inspector* der *Transform*-Komponente und führen die Funktion **Reset** aus. Sollte „gate\_01“ nun auf der Seite liegen, müssen Sie noch einmal die X-Achse von „gate\_01“ um -90 drehen.
5. Markieren Sie „Gate“ in der *Hierarchy* und drücken Sie die **F**-Taste, um den Fokus der *Scene View* auf das Fallgatter zu legen.
6. Falls nicht mehr selektiert, markieren Sie das Elternobjekt „Gate“ und öffnen über **Window/Animation** das *Animation-Fenster*.
7. Erzeugen Sie über **Create New Clip** im Drop-down-Menü (bzw. Auswahlbox) des Fensters einen neuen *Animation-Clip* und speichern diesen im „Animations“-Ordner unter dem Namen „OpenDoor“.
8. Wählen Sie über **Add Curve** „gate\_01/Transform/Position“ die *Position*-Eigenschaft aus und bestätigen die Auswahl mit dessen Pluszeichen.
9. Wechseln Sie nun die *Curves*-Ansicht.
10. Klappen Sie die *Position*-Eigenschaft auf und markieren Sie die „Position.y“.
11. Der erste *Keyframe* kann auf 0 bleiben, den zweiten *Keyframe* der (vermutlich) grünen Kurve schieben Sie auf der Zeitachse bei 1:00 hoch auf den Wert 1.8. Über das Mausrad und die **[Umsch]**-Taste (**Shift**) können Sie nun rein- und rausscrollen, was das Hochschieben des Punktes vereinfacht. Alternativ können Sie den Wert auch direkt in der Parameterliste eintragen.
12. Testen Sie die Animation, indem Sie auf die Play-Taste drücken. Das „Gate“ sollte jetzt in einem Wiederholungsmodus immer wieder hochfahren.
13. Beenden Sie die Aufnahme durch Drücken auf das rot leuchtende Aufnahme-Symbol.
14. Schließen Sie das *Animation-Fenster* (Klick auf das **X** oben rechts).
15. Öffnen Sie im *Project Browser* den Ordner „Animations“ und selektieren Sie den neuen *Animation-Clip* „OpenDoor“.
16. Sollte im *Inspector* der Parameter *Loop Time* nun aktiviert sein, deaktivieren Sie diese Eigenschaft.

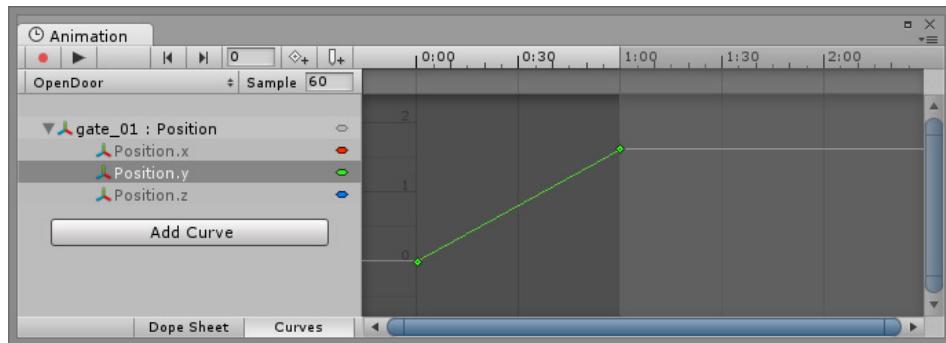


Bild 17.7 Animation eines sich öffnenden Fallgatters

## ■ 17.3 Animationen importieren

Für gewöhnlich werden Animationen über eine FBX-Datei in Unity hineingeladen. Dies kann ein einzelnes FBX-File sein, das lediglich die Animation selbst beinhaltet, oder aber ein 3D-Modell, das zusätzlich die Animationen innehat. Für diesen Zweck besitzen die *Import Settings* der FBX-Dateien neben dem Reiter *Model* (siehe Kapitel „Objekte in der dritten Dimension“) auch noch die Reiter *Rig* und *Animations*.

- **Model** legt die Importeinstellungen für das 3D-Modell fest, die u. a. Einfluss auf das *Mesh* und die Materialien haben.
- **Rig** definiert die Schnittstelle zwischen dem Modell und möglicher Animationen.
- **Animations** legt die Importeinstellungen für die in der FBX-Datei enthaltenen Animation(en) fest.

### 17.3.1 Rig

Der große Unterschied zwischen dem Animieren in Unity und dem in Modelling-Tools ist der, dass Sie in Modelling-Tools ein Skelett für Ihr Modell definieren können (auch *Rig* genannt). Über das sogenannte *Skinning* verbinden Sie dann die Modelloberfläche, also das *Mesh*, mit dem Skelett, sodass Sie durch das Animieren des Skeletts das *Mesh* verformen können. Mit dieser Vorgehensweise können Sie sehr spannende Animationen, wie z. B. Laufbewegungen oder auch Sprechanimationen, erstellen.

Damit dies auch in Unity funktioniert, nutzt Unity eine Schnittstelle, die über den Reiter *Rig* definiert wird. Ein wichtiger Bestandteil davon ist der sogenannte *Avatar*, der u. a. die Skelettdaten des Modells speichert. Unity unterscheidet hierbei zwischen zwei generellen Arten, den sogenannten *Animation Types*:

- **Generic** dient dem Generieren einer Animationsschnittstelle, die universell einsetzbar ist. Hierbei wird automatisch ein *Avatar* erstellt.

- **Humanoid** ist speziell für humanoide Wesen geeignet und ermöglicht später spezielle Animationsfunktionen wie zum Beispiel das leichte Wiederverwenden eines *Avatars* und die dazu passenden Animationen. Auch hier wird automatisch ein *Avatar* erzeugt, der aber manuell noch nachjustiert werden kann.

Weiter gibt es noch die *Animation Type*-Einstellung *None* für Modelle, die nicht animiert werden sollen. Zuletzt gibt es noch die Einstellung *Legacy*. Diese dient als Schnittstelle für Animationen aus bestehenden Projekten, die noch ein altes Animationssystem von Unity nutzen. Da dieses System aber nicht mehr weiterentwickelt wird, möchte ich darauf auch nicht weiter eingehen.

Wenn Sie schließlich mit Ihren Einstellungen fertig sind, bestätigen Sie diese mit **Apply** in den *Import Settings*.

### 17.3.1.1 Generic

Wählen Sie den *Animation Type Generic*, brauchen Sie eigentlich kaum noch etwas zu machen. Unity erstellt automatisch einen passenden *Avatar*, und das war es schon. Haben Sie nun für dieses Modell eigene Animationen, brauchen Sie diese nur noch zu importieren und abzuspielen (darauf kommen wir noch zu sprechen).

Der einzige Parameter, der bei diesem *Animation Type* manuell eingestellt werden muss, ist die Zuweisung des *Root Nodes*. Dieser ist unter anderem notwendig, um Fortbewegungen zu berechnen, die durch die Animationen selbst ausgelöst werden.

### 17.3.1.2 Humanoid

Sollten Sie den *Animation Type Humanoid* auswählen, erscheint ein zusätzlicher Knopf mit der Beschriftung **Configure**. Über diesen können Sie das *Bone-Mapping* und die *Muscle-Definitionen* festlegen und testen. Gelingt die automatische Konfiguration von Unity, erscheint vor dem Knopf ein Haken, ansonsten wird ein Kreuz eingeblendet. Unabhängig davon sollten Sie aber auf jeden Fall diese Konfiguration einmal aufrufen und testen.



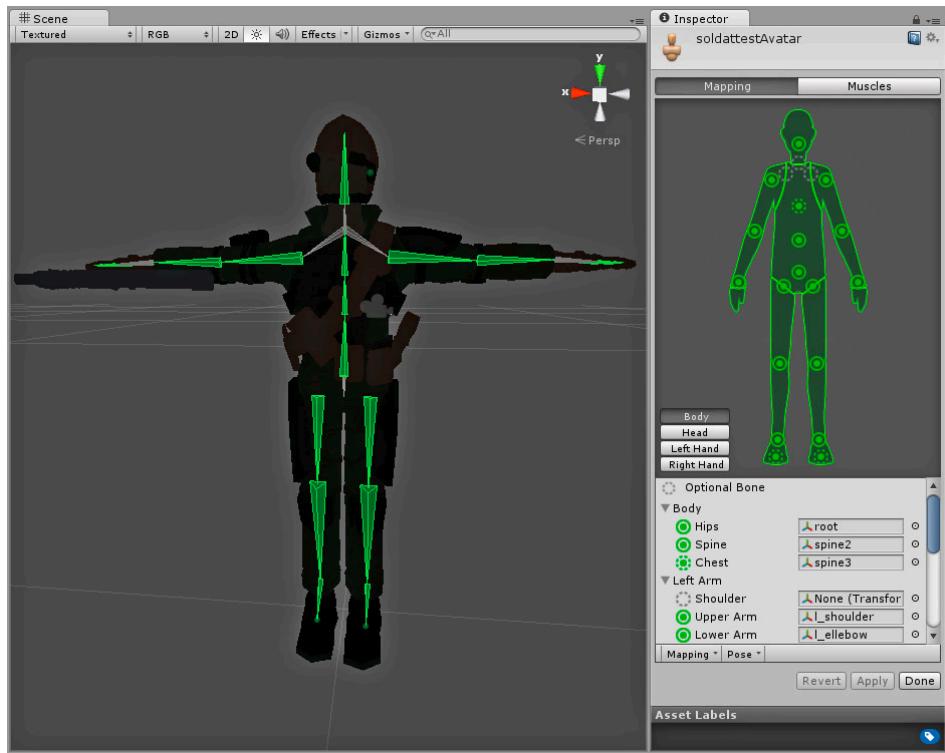
#### Bones-Mindestanzahl beachten

Ein Humanoid-Avatar benötigt mindestens 15 Bones. Wenn Ihr Modell nicht so viele enthält, lässt Unity das Mapping des Humanoid-Avatars nicht zu.

Wenn Sie *Humanoid* gewählt haben und auf den Button **Configure** klicken, wird eine neue Testszene gestartet, in die das Modell hineingeladen wird. Das Besondere hierbei ist, dass nun auch das Skelett des Modells angezeigt wird (siehe Bild 17.8), was normalerweise in Unity nicht der Fall ist.

Gemeinsam mit der Übersichtsgrafik des *Avatars* und den *Mapping-Slots* darunter können Sie fehlerhafte Zuweisungen korrigieren. Zusätzlich können Sie auf dem *Muscle*-Reiter mithilfe von Schiebereglern die *Bones* verdrehen und strecken. Auf dieses Weise fallen falsche Zuweisungen schnell auf.

Um einen funktionierenden *Humanoid-Avatar* zu erhalten, müssen Sie alle *Bones* richtig zuweisen, die in der Übersichtsgrafik mit einem geschlossenen Kreis dargestellt werden. Alle gestrichelten Kreise sind optionale *Bones*, die das Gesamte ergänzen können, aber



**Bild 17.8** Skelettdarstellung und Avatar-Mapping

nicht zwingend erforderlich sind. Über das Button-Menü neben der Grafik können Sie auch weitere optionale *Bones* im Gesicht und in den Händen definieren. Sind Sie fertig, bestätigen Sie Ihre Anpassungen mit **Apply** und schließen die Konfiguration danach mit **Done**.

Dieses gesamte Prozedere brauchen Sie glücklicherweise jetzt nicht für jeden Charakter zu machen. Wenn Sie mehrere humanoide Charaktere in Ihrer Modelling-Software erstellen und allen die gleichen Skelettbezeichnungen geben, können Sie diese Zuweisungen einfach übernehmen.

Hierfür gibt es auf dem Reiter *Rig* der *Import Settings* den Parameter *Avatar Definition*, wo Sie *Copy From Other Avatar* wählen können. Unity blendet einen weiteren Parameter *Source* ein, auf den Sie den bereits konfigurierten *Avatar* ziehen können.



#### Konfiguration eines Humanoid-Avatar

Ein Video auf der DVD zeigt Ihnen das Konfigurieren eines Humanoid-Avatars.

## 17.3.2 Animationen

Die eigentlichen Einstellungen für den Import der Animationen finden Sie auf dem Reiter **Animations**. Damit die in der FBX-Datei enthaltenen Animationen aber auch importiert werden, müssen Sie zunächst auch *Import Animation* aktivieren.

Haben Sie dies gemacht, können Sie über *Anim. Compression* noch die Animationskompression festlegen, was Sie auch immer tun sollten. Hierdurch wird die Datengröße der importierten Animationen reduziert und gleichzeitig auch der zur Laufzeit notwendige Speicherbedarf heruntergesetzt. Drei weitere Parameter geben Ihnen die Möglichkeit, die Genauigkeiten der Kompressionen für die einzelnen animierten Parameter zu bestimmen:

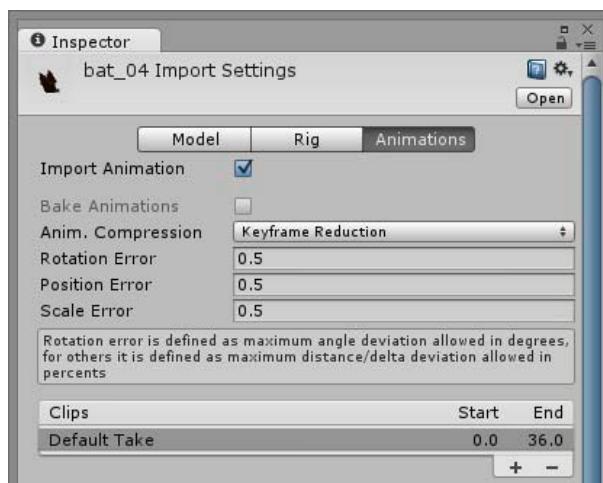
- **Rotation Error**
- **Positionen Error**
- **Scale Error**

Je kleiner hier der zugewiesene Wert ist, desto höher ist die Genauigkeit der komprimierten Animation gegenüber der Originalanimation. Demgegenüber steigt dann aber auch natürlich wieder der Speicherbedarf. Zu Anfang sollten Sie diese Werte erst einmal so belassen. Später finden Sie hier aber möglicherweise noch etwas Potenzial, wo Sie Ihre Projekte optimieren können. Als Nächstes müssen Sie nun noch die Animationen ggf. aufteilen und weiter einrichten.

### 17.3.2.1 Animationen aufteilen

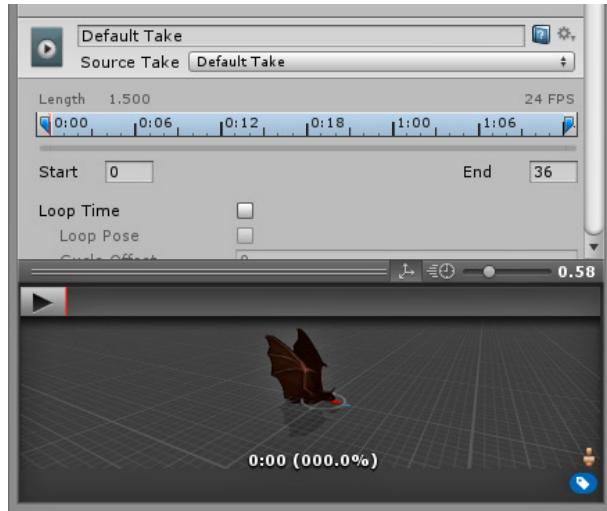
Ein FBX-File kann beliebig viele Animationen beinhalten. Je nach Art des Exports aus dem eigentlichen Animationsprogramm werden diese Animationen dann entweder als separate *Animation-Clips* importiert oder als ein gemeinsamer Clip, der dann noch aufgeteilt werden muss. Der bzw. die Clips erscheinen dann in der Liste *Clips*.

Möchten Sie den Clip aufteilen, müssen Sie einfach auf das Pluszeichen in der *Clips*-Liste drücken, und schon erhalten Sie einen neuen *Animation-Clip*. Über die Zeitpunkte *Start* und *End* legen Sie den Ausschnitt fest, der vom Gesamtclip für diesen *Animation-Clip* genutzt



**Bild 17.9**

Importierte Animation



**Bild 17.10**  
Preview-Bereich des Animationsimports

werden soll. In dem unteren *Preview*-Fenster können Sie über das *Play*-Symbol den aktuellen Ausschnitt betrachten.

Als Nächstes können Sie nun Einstellungen an den einzelnen *Animation-Clips* vornehmen, um noch Nachjustierungen vorzunehmen. Am Ende müssen Sie die Änderungen schließlich mit **Apply** bestätigen.

### 17.3.2.2 Importierte Animation-Clips bearbeiten

Im Normalfall stellt ein *Animation-Clip* genau eine Handlung dar. Dies kann z.B. eine Wurfanimation sein, ein Sprung oder auch Vorgänge, die in einer Endlosschleife abgespielt werden können wie Lauf- oder Idle-Animationen, also Animationen, die im Ruhezustand eines Charakters abgespielt werden. Wenn Ihre Animation mehrere hintereinander abspielt, sollten Sie diese zunächst einmal auf mehrere *Animation-Clips* verteilen (siehe oben).

Nach dem Aufteilen können Sie nun jeden Clip nachbearbeiten, indem Sie diesen zunächst in der *Clips*-Liste selektieren. Neben dem *Animation-Clip*-Symbol (ein Play-Button-Zeichen) finden Sie zunächst den Namen, den Sie hier verändern können. Darunter sehen Sie den *Start*- und den *End*-Punkt der Animation. Diese können Sie hier sowohl über die Timeline wie auch über die Parameter verändern. Speziell für das Konfigurieren von sich wiederholenden Animationen eignet sich hier die Timeline. Verschieben Sie hier den *Start*- und *End*-Reiter in der Timeline, überprüft Unity automatisch, ob *Start*- und *End*-Punkt für einen Loop passen (dargestellt über die *loop match*-Ampeln auf der rechten Seite). Dabei muss es aber nicht zwangsläufig so sein, dass für einen guten Loop auch alle Ampeln auf Grün stehen. Manchmal ist dies auch gar nicht möglich, wie im Falle einer Laufanimation, die mit dem MoCap-Verfahren aufgenommen wurde. Mit dem Loopen beschäftigt sich dann auch gleich der erste Parameterblock der *Animation-Clip*-Einstellungen:

- **Loop Time** aktiviert das automatische Wiederholen der Animation.
- **Loop Pose** passt die Animation automatisch an, damit Anfang und Ende nahtlos hintereinander passen.

- **Offset** fügt dem eigentlichen Animations-Start einen Offset zu. Haben Sie beispielsweise eine drei Sekunden lange Animation, so können Sie diese erst ab Sekunde 2 starten lassen. Trotzdem wird nach dem ersten Durchlauf auch der Teil der Animation abgespielt, der sich vor dem Offset befindet.

Nach diesen Einstellungen kommen nun die *Root Transform*-Parameter, die sich auf Drehungen und Positionsänderungen beziehen, die durch die Animation auf das *GameObject* und dessen *Mesh* übertragen werden (*Root Motion* genannt). Diese teilen sich in drei Parameterblöcke auf, die sich zum einen auf die Rotationen, zum zweiten auf die vertikalen Bewegungen und zum dritten auf die Bewegungen in die X- und Z-Richtungen beziehen. Jeder Block besitzt drei Parameter, die ich hier zusammenfassend erläutern möchte:

- **Bake Into Pose** ignoriert die Rotation bzw. die Bewegungsrichtung, die durch die Animation dem Modell zugefügt wird. Anstelle dessen wird ein konstanter Wert genommen, der anhand des *Avatars* berechnet wurde (*Humanoid Animation Type*) bzw. durch das als *Root Node* zugewiesene *Transform (Generic Animation Type)* vorgegeben wird. Gerade wenn Sie mit *Motion Capture*-Animationen arbeiten, kann es gut sein, dass diese Animationen nicht hundertprozentig die gewünschte Ausrichtung haben. So kann z.B. eine Laufanimation auch mal nicht genau geradeaus verlaufen, sondern einen leichten Bogen machen. Im Bereich *Root Transform Rotation* kann das über diese Eigenschaft unterbunden werden.
- **Based Upon** legt fest, worauf sich der Parameter, also die Rotation bzw. die Bewegungsrichtung, beziehen soll. Hier stellt Unity je nach Typ (Rotation, Y-Bewegung, X-/Z-Bewegung) passende Auswahlmöglichkeiten zur Verfügung. Bei „Original“ bezieht sich die Animation auf die im Clip hinterlegte Ausrichtung. Bei Parametern wie „Body Orientation“ nimmt sie die Ausrichtung des *GameObjects* bzw. des *Avatars*.
- **Offset** ermöglicht Ihnen, den eingestellten Wert nachzuregeln. Ist beispielsweise *Bake Into Pose* aktiv, kann an dieser Stelle der automatisch berechnete Wert nachgeregelt werden. Für eine Laufanimation (siehe oben) kann das notwendig sein, wenn der berechnete Wert einen leichten Drall zur Seite aufweist.

Der letzte Parameter **Mirror** spiegelt schließlich links und rechts der Animation. Würde beispielsweise durch die Animation der linke Arm eines humanoiden Modells gehoben werden, wird bei aktivem *Mirror*-Parameter nun der rechte Arm gehoben.

### 17.3.2.3 Mask

Über diesen Reiter definieren Sie, welche Teile der Animation tatsächlich Anwendung finden. Die deaktivierten Bereiche werden dabei von der Animation ignoriert. Sie können die Wirkungsweise ganz einfach testen, indem Sie je nach *Animation Type* über die Humanoid-Grafik oder über den *Transform*-Baum einige Bereiche deaktivieren und die Auswirkung im *Preview*-Fenster verfolgen.

### 17.3.2.4 Events

Mit dem Button **Add Event** erzeugen Sie an der aktuellen Position der Animation (gekennzeichnet durch den roten Strich) ein Ereignis, das eine Methode aus einem Skript aufruft. Es kann ein *Float*-, *Int*- oder ein *String*-Wert übergeben werden. Wählen Sie hierfür zunächst über den Parameter *Object* das Skript aus, in dem die Methode enthalten ist. Dann tragen

Sie in *Function* den Namen der Methode ein. Als Letztes tragen Sie noch je nach Parametertyp den zu übergebenden Wert ein. Beachten Sie, dass später jedes *GameObject*, das diesen Animation-Clip nutzt, auch das hier verlinkte Skript besitzt.

## ■ 17.4 Animationen einbinden

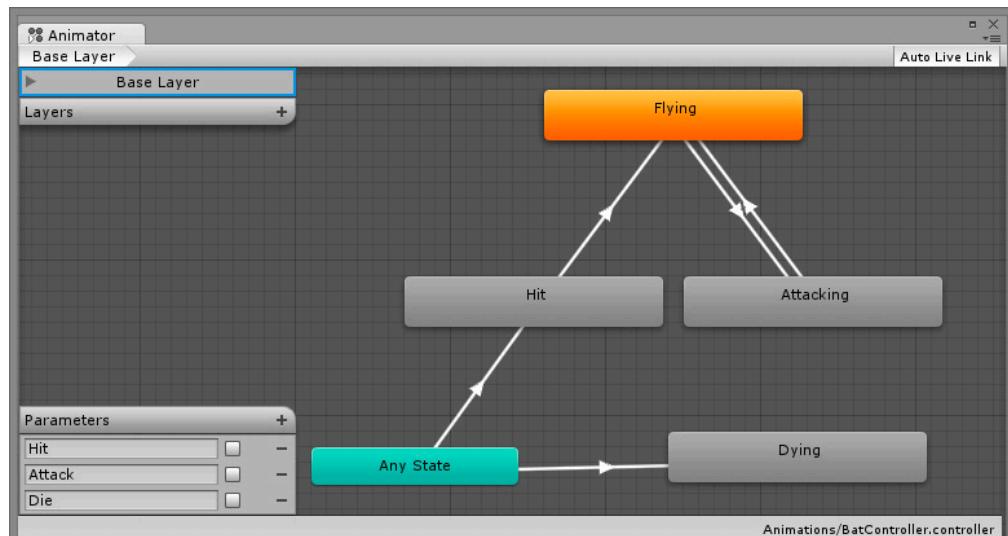
Möchten Sie in Ihrem Spiel ein *GameObject* animieren, reicht es nicht aus, nur einen *Animation-Clip* zu erstellen, Sie müssen diese auch auf das jeweilige *GameObject* anwenden. Hierfür benötigen Sie zwei Dinge: eine *Animator*-Komponente und einen *Animator Controller*.

Der *Animator Controller* definiert bei mehreren Animationen, wann welche Animation abgespielt werden soll und wie zwischen diesen gewechselt wird. Die *Animator*-Komponente wiederum wird dem *GameObject* zugefügt und stellt hauptsächlich die Verbindung zwischen dem *GameObject* und dem *Animator Controller* her.

Wenn Sie innerhalb von Unity eine Animation aufnehmen, werden beide Elemente automatisch angelegt und dem animierten Objekt zugewiesen.

### 17.4.1 Animator Controller

Ein *Animator Controller* arbeitet nach dem Prinzip einer *State Machine* (Zustandsautomat). Jede Animation stellt hierbei einen Zustand, einen *State*, dar, zwischen denen, abhängig von Bedingungen, gewechselt wird.



**Bild 17.11** States eines Animator Controllers

Am einfachsten erzeugen Sie einen *Animator Controller* über Ihre rechte Maustaste im *Project Browser*, genauer gesagt über **Create/Animator Controller**. Aber auch über das Hauptmenü **Assets/Create/Animator Controller** sowie über das Drop-down-Menü vom *Project Browser* können Sie diesen erstellen.

Haben Sie nun einen erstellt, öffnet sich über einen Doppelklick auf den *Animator Controller* ein neues Fenster mit der Überschrift „Animator“.

Zu Anfang besitzt der Animator Controller lediglich den grünen State „Any State“, den Sie auch in Bild 17.11 sehen. Dieser symbolisiert, wie der Name schon sagt, jeden Zustand der *State Machine*. Dieser State dient dem Wechseln in Zuständen, die unabhängig davon sind, in welchem man sich aktuell befindet.

Sollte irgendwann das Fenster zu klein werden, so können Sie mit gedrückter **Alt**-Taste und linker Maustaste die Ansicht verschieben.

#### 17.4.1.1 Animation States erstellen

Um einen neuen *Animation State* im *Animator Controller* anzulegen, ziehen Sie einfach via Drag & Drop einen *Animation-Clip* aus dem *Project Browser* in den *Controller*. Sobald Sie diese fallen lassen, wird ein neuer *Animation State* erzeugt. Alternativ können Sie auch über **Create State/Empty** ein leeres *Animation State* erzeugen, das zunächst keinen *Animation-Clip* abspielt. Über den Parameter *Motion* im *Inspector* können Sie aber Ihrem *State* eine Animation zuweisen. Über den Parameter *Speed* können Sie bestimmen, wie schnell der *Animation-Clip* abgespielt wird. Mit *Mirror* können Sie schließlich links und rechts der Animation spiegeln.



#### Animationen rückwärts abspielen

Sie können jeden *Animation-Clip* auch rückwärts abspielen. Hierfür müssen Sie einfach dem *Speed*-Parameter einen negativen Wert zuweisen, z. B. -1.

Der erste *Animation State*, den Sie in einem *Animator Controller* erzeugen, wird immer orange dargestellt. Dieser *State* ist der Default-Zustand, der auch zu Beginn gleich gestartet wird. Wenn Ihr *Controller* mehrere *Animation States* besitzt, können Sie über die Funktion *Set As Default* im Kontextmenü Ihrer Maus diesen auch nachträglich ändern.

#### 17.4.1.2 Animation State wechseln

Über sogenannte *Transitions* können Sie Bedingungen festlegen, wann welcher *Animation State* aktiv ist. Selektieren Sie hierfür den Ausgangs-*State*, z.B. den Default-Zustand, und führen im Kontextmenü der Maus die Funktion *Make Transition* aus. Nun führen Sie die Maus zu dem Ziel-*State* und klicken Sie auf diesen. Es wird ein Pfeil gezeichnet, der diese *Transition* darstellt.

Wenn Sie eine Transition, also den Pfeil, selektieren, wird im *Inspector* die Eigenschaft angezeigt. Der Bereich *Conditions* legt hierbei die Bedingungen fest, wann vom Ausgangs-*State* zum Ziel-*State* gewechselt wird. Standardmäßig wird mit dem Parameter *Exit Time* ein Wechsel nach Ablauf der Animation durchgeführt. Die Angabe 0.90 besagt hierbei, dass der Wechsel startet, wenn die Animation zu 90% abgelaufen ist. Hier können Sie nun jeden *Parameter* aus der *Parameter*-Liste auswählen und für die Wechselbedingung nutzen.

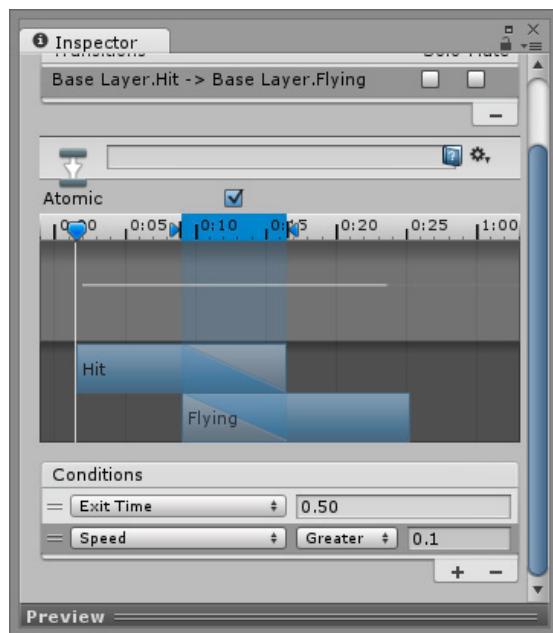


### Bedingungen verketten

Sie können beliebig viele Bedingungen verketten, sodass ein Wechsel nur durchgeführt wird, wenn mehrere Bedingungen erfüllt sind. Hierfür brauchen Sie nur auf das Pluszeichen in den *Conditions* zu drücken und weitere Bedingungen einzutragen.

Im *Animator Controller* können Sie in der *Parameter-Liste* beliebig viele neue Variablen definieren, die für das Steuern dieser Wechsel zur Verfügung stehen. Bei einem Klick auf das Pluszeichen in der *Parameter-Liste* müssen Sie zunächst den Typ des Parameters auswählen. Danach wird ein neuer Parameter in der Liste angelegt, den Sie dann den Verwendungszwecken entsprechend umbenennen sollten (z.B. Speed, Direction, Jump).

Um einen Parameter zu löschen, brauchen Sie einfach nur auf das daneben stehende Minuszeichen zu drücken. Haben Sie hier nun beispielsweise eine *Float*-Variable definiert, können Sie diese in der *Transition* nutzen. Dabei wird in diesem Fall zusätzlich eine Auswahlbox mit *Less* und *Greater* angeboten, mit der Sie nun beschreiben können, was für einen *Float*-Wert die Variable besitzen muss, damit der Wechsel stattfindet. Auf das Setzen dieser Variablen komme ich gleich noch zu sprechen.



**Bild 17.12**  
Wechselbedingungen eines States

#### 17.4.1.3 Any State nutzen

Wie bereits erwähnt, besitzt jeder *Animator Controller* den Eintrag „Any State“, der alle Status symbolisiert. Sollten Sie *Animation States* haben, die unabhängig von anderen *States* zu jeder Zeit eintreten können, dann kann das mit einer Bedingung von „Any State“ aus realisiert werden.

Ein typisches Beispiel wäre der Tod eines Spielers. Denn ob ein Spieler tödlich verletzt wird, ist meistens unabhängig davon, ob er gerade steht, springt, läuft oder sich in einem anderen *Animation State* befindet. In diesem Fall könnte es einen *Bool*-Parameter namens „Dead“ geben, der gleichzeitig auch die Bedingung der *Transition* ist, die von „Any State“ zum *Animation State* „Dying“ geht, der wiederum die Sterben-Animation des Spielers abspielt.

#### 17.4.1.4 Animationsübergänge bearbeiten

Wenn Sie eine *Transition* im *Animator Controller* markieren, können Sie nicht nur die Bedingungen festlegen, wann ein Wechsel stattfindet, Sie können auch die Art und Weise des Wechsels bestimmen. Dies können Sie in einer Timeline-Ansicht einstellen, wo beide Animationen untereinander angezeigt werden (siehe Bild 17.12).

Der blaue überlappende Bereich, in dem auch die Animationen abfallend und aufsteigend dargestellt werden, beschreibt den Wechsel zwischen den Animationen. Stellen Sie sich vor, Sie haben eine *Transition* zwischen einer Idle- und einer Laufanimation. Ist nun die Wechselbedingung erfüllt, kann der Übergang so gestaltet werden, dass der Charakter sofort losläuft oder aber erst langsam beschleunigt, bis er schließlich die Endgeschwindigkeit erreicht. Die Dauer dieses Überganges definieren Sie über die Länge der Überlappung.

Mit dem Kenner *Atomic* legen Sie zudem fest, ob die *Transition* von einer anderen unterbrochen werden kann oder ob diese, einmal gestartet, auch zu Ende läuft.

#### 17.4.1.5 Blend Trees

*Blend Trees* stellen zunächst einmal ganz normale *States* dar, die wie andere *Animation States* eingebunden und gestartet werden. Im Gegensatz zu diesen besteht ein *Blend Tree* aber aus mehreren ähnlichen *Animation-Clips*, die anhand eines *Float*-Parameters weich ineinander geblendet werden. Dabei können auch Zwischenstufen dargestellt werden, wo Unity mehrere Animationen abhängig vom Parameterwert mixt und daraus neue Bewegungsabläufe berechnet. Ein typisches Beispiel ist die Vorwärtsbewegung eines Spielers, der sowohl eine Geh- wie auch eine Laufanimation besitzt. Damit der Wechsel zwischen diesen beiden Animationen nicht zu ruckartig verläuft, kann ein *Blend Tree* genutzt werden.

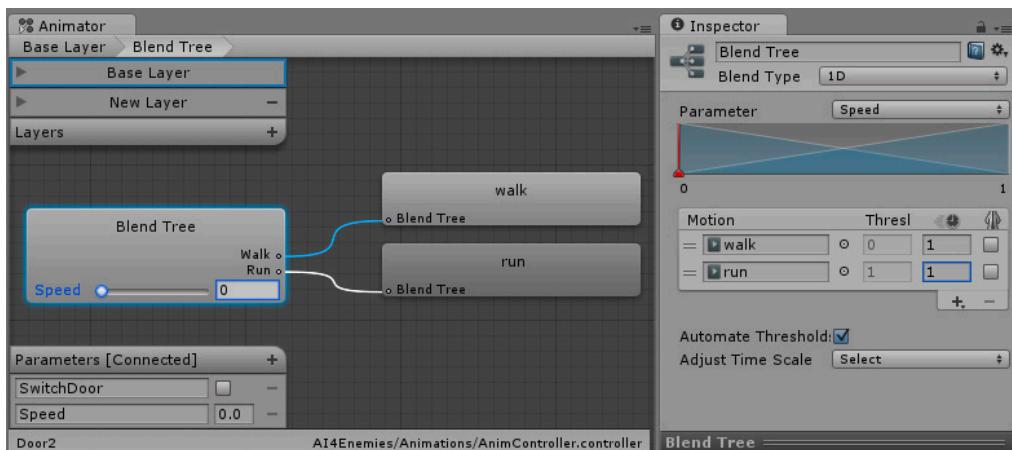


Bild 17.13 Blend Tree zum Überschwenken zwischen zwei Animationen

Einen *Blend Tree* erzeugen Sie über die rechte Maustaste, und zwar über **Create State/From New Blend Tree**. Mit einem Doppelklick auf diesen gelangen Sie in den *Blend Tree*, wo Sie die eigentlichen Animationen definieren, die gemixt werden sollen. Dies machen Sie über das Pluszeichen in der *Motion-Liste*. Über die Kreise fügen Sie dem *Slot* dann den eigentlichen *Animation-Clip* zu. Eine weitere wichtige Eigenschaft ist der Parameter, der diese Animationsweiche steuert. Über der Grafik finden Sie hierfür den Parameter. Weiter können Sie den Wertebereich festlegen. Dies machen Sie direkt in der Grafik, deren Zahlen editierbar sind. Um zur ursprünglichen Ansicht des *Animator Controllers* zu gelangen, drücken Sie in der obigen Navigationsleiste des *Controllers* auf „Base Layer“.



### Mit Blend Trees Animationsgeschwindigkeiten steuern

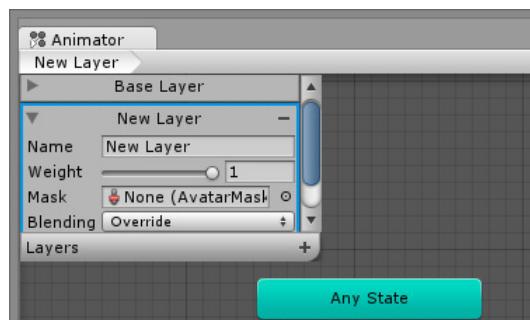
Sie können auch *Blend Trees* mit einer einzigen Animation nutzen. Hierfür erzeugen Sie z. B. zwei *Motion-Slots* und fügen beiden den gleichen *Animation-Clip* zu. In der rechten Spalte können Sie nun noch die Abspieldgeschwindigkeit der Animation einstellen. Hier ändern Sie einen der beiden. Auf diese Weise können Sie ganz einfach mit der Variablen, die Sie als Parameter hinterlegen, die Geschwindigkeit der Animation steuern.

Bedenken Sie beim Definieren der *Transition* zu einem *Blend Tree*, dass diese nicht zwingend den *Float*-Parameter besitzen muss, der für die Weichenstellung im *Blend Tree* verantwortlich ist. Da der *Blend Tree*-Parameter einen Wertebereich abdeckt, z. B. von -1 bis 1, macht dies häufig auch keinen Sinn. Deshalb dienen meistens andere Parameter als Auslöser für diesen *State* bzw. *Blend Tree* (z. B. ein *Boolean* oder ein *Trigger*), die Auswertung des *Float*-Parameters findet dann im *Blend Tree* selber statt.

#### 17.4.1.6 Layer und Avatar Body Masks

Nicht selten gibt es Animationen, die parallel zu anderen abgespielt werden sollen. Nehmen wir z. B. eine Wurfanimation. Der Spieler soll nicht nur werfen können, wenn er steht. Er soll auch werfen können, wenn er vorwärts geht, läuft oder sich rückwärts bewegt.

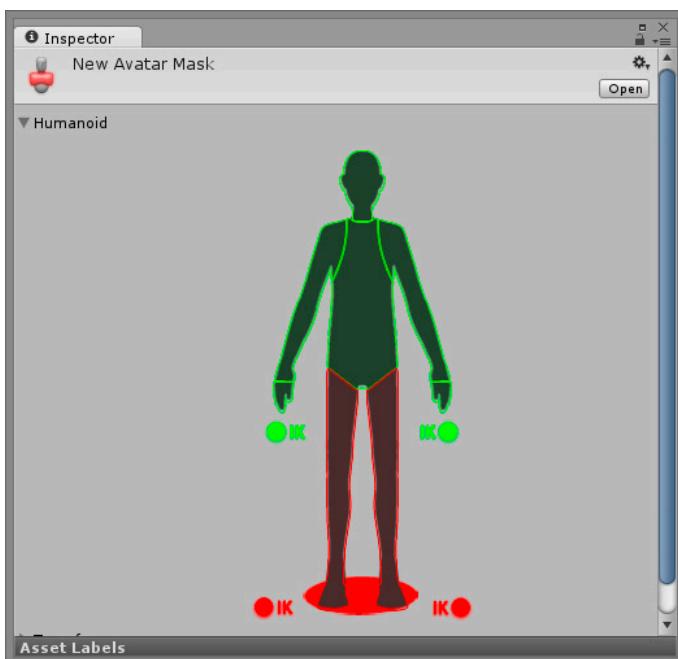
Damit Sie nicht für jede mögliche Variante eine neue Animation erstellen müssen, können Sie Animationen übereinanderlegen. Hierfür bietet Unity im *Animator Controller* die Möglichkeit, Animationsschichten, also *Layer* zu definieren. Der erste *Layer* heißt standardmäßig immer „Base Layer“, kann bei Bedarf aber auch umbenannt werden. Um einen weiteren *Layer* zu erstellen, drücken Sie auf das Pluszeichen in der *Layers*-Liste.



**Bild 17.14**  
Layer-Eigenschaften

Über die *Blending*-Eigenschaft legen Sie fest, wie die Animation dieses *Layers* mit der des „Base Layers“ (und denen der anderen *Layer*) kombiniert werden soll. Bei *Override* werden die anderen Animationen einfach ignoriert und nur diese wird dargestellt. Bei *Additive* werden die Animationen gemischt und es werden Zwischenwerte für die animierten Werte ermittelt. *Weight* beschreibt hierbei, wie stark der Einfluss dieses *Layers* ist. Wenn dieser *Layer* alles überlagern soll, dann setzen Sie diesen Wert auf 1 und *Blending* auf *Override*.

Damit im Falle einer Wurfanimation aber nur die Arme über die anderen Animationen gelegt werden und nicht auch noch die Beine etc., gibt es eine *Avatar Body Mask*, die Sie einem *Layer* hinterlegen können. In dieser können Sie bestimmen, welche Teile animiert werden sollen. Eine *Avatar Body Mask* erzeugen Sie direkt im *Project Browser* über das Kontextmenü Ihrer Maus oder über dessen Fenster-Menü.



**Bild 17.15**  
Avatar Body Mask

In den zusätzlichen *Layer* definieren Sie lediglich die *Animation States*, die für diesen *Layer* und dessen Animationen notwendig sind. Die *Transitions* werden parallel zum „Base Layer“ überprüft, um die *Animation States* dann ggf. auszuführen. Meist ist der „Base Layer“ recht komplex, während die weiteren, wenn diese überhaupt vorhanden sind, eher einfach aufgebaut sind.

## 17.4.2 Animator-Komponente

Haben Sie einen *Animator Controller* erstellt, müssen Sie nun noch dafür sorgen, dass dieser auch das richtige *GameObject* steuert. Hierfür gibt es die *Animator*-Komponente, die Unity meistens schon den importierten Objekten automatisch zufügt. Sollte das zu animierende

*GameObject* noch keine besitzen, können Sie diesem eine über das Menü **Component/Miscellaneous/Animator** zufügen. Ein *Animator* hat folgende Eigenschaften:

- **Controller** legt den zu nutzenden *Animator Controller* fest. Ziehen Sie hierauf Ihren *Controller*.
- **Avatar** legt den *Avatar* fest, der für dieses *GameObject* genutzt werden soll. Normalerweise sollte hier schon der richtige hinterlegt sein.
- **Apply Root Motion** bestimmt, ob das *GameObject* über die Bewegungen der Animationen verschoben (aktiv) oder ob das Neupositionieren über ein gesondertes Skript erledigt werden soll (deaktivierter Parameter). Wenn die aufgezeichnete Bewegung nicht dem entspricht, was man als Entwickler haben möchte, können Sie diese über ein Skript steuern (z. B. über *Transform-* oder *Rigidbody*-Eigenschaften).
- **Animation Physics** legt fest, ob die Animation in *FixedUpdate* (aktiv) oder in der *Update*-Methode aktualisiert werden soll (deaktivierte Eigenschaft). Sollte das Modell mit einem *Rigidbody* ausgestattet werden, um mit der Physik zu interagieren, sollten Sie es also aktivieren.
- **Culling Mode** legt fest, wann die Animationen abgespielt werden sollen. Bei *Always animate* werden die Animationen immer abgespielt. Bei *Based on Renderers* wird das *GameObject* nur animiert, wenn es sich auch im Kamerabild befindet und gerendert wird. Bewegungen, die über *Root Motion* verursacht werden, werden dem *GameObject* aber auch im verdeckten Zustand zugeführt.

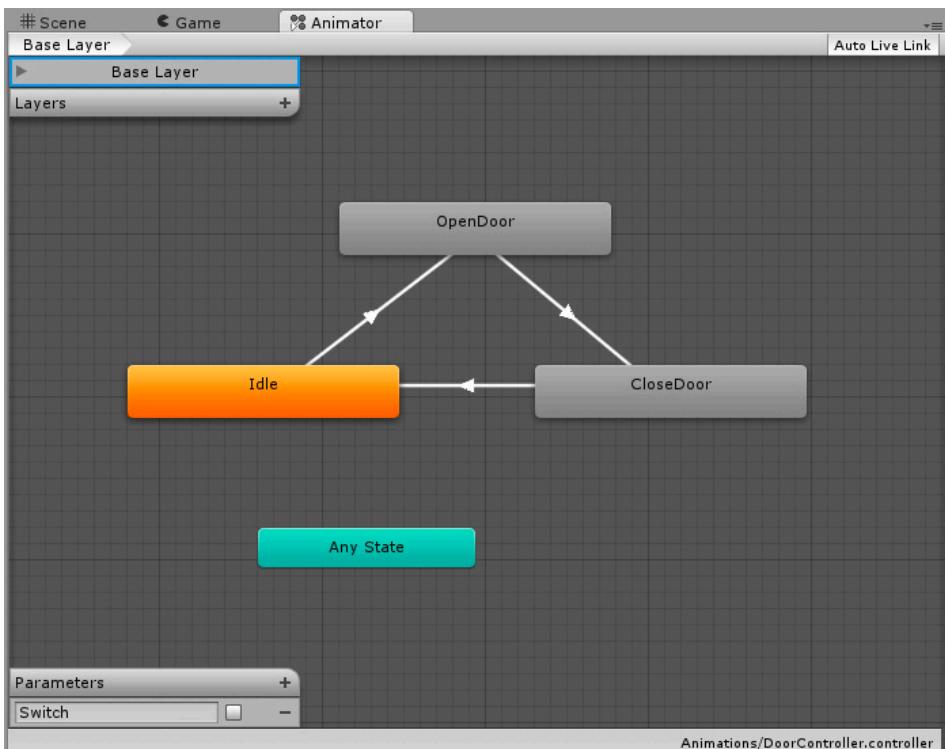
### 17.4.3 Beispiel Fallgatter Animator Controller

Für das bereits animierte Fallgatter „Gate“ wollen wir nun einen *Animator Controller* erstellen. Eine *Animator*-Komponente sollte das Eltern-Objekt bereits automatisch erhalten haben. Sollte dies nicht der Fall sein, fügen Sie diesem eine über **Component/Miscellaneous/Animator** zu. Als Nächstes kommt jetzt der *Animator Controller* dran.

1. Erzeugen Sie einen neuen *Animator Controller* im Animations-Ordner. Klicken Sie hierfür im *Project Browser* in den „Animations“-Ordner mit der rechten, wählen **Create/Animator Controller** und geben dem neuen *Animator Controller* den Namen „DoorController“. Alternativ können Sie den *Animator Controller* auch über das Drop-down-Menü vom *Project Browser* oder über das **Asset**-Menü im Hauptmenü erstellen.
2. Öffnen Sie den *Animator Controller* mit einem Doppelklick.
3. Erzeugen Sie im *Animator Controller* mit der rechten Maustaste über **Create State/Empty** einen neuen *Animation State*. Dieser sollte nun orange eingefärbt als Default-State angezeigt werden.
4. Selektieren Sie den *State* und benennen Sie diesen im *Inspector* in „Idle“ um.
5. Fügen Sie dem *Animator Controller* per Drag & Drop den „OpenDoor“-*Animation-Clip* zu.
6. Erzeugen Sie eine *Transition* von „Idle“ nach „OpenDoor“, indem Sie als Erstes auf den Idle-State mit der rechten Maustaste klicken und **Make Transition** auswählen. Dann klicken Sie mit der linken Maustaste auf „OpenDoor“.
7. Legen Sie im *Animator Controller* einen neuen Parameter „Switch“ vom Typ *Trigger* an.

8. Markieren Sie die erstellte *Transition* und legen Sie bei *Conditions* „Switch“ als einzige Bedingung fest.
9. Ziehen Sie ein zweites Mal den „OpenDoor“-*Animation-Clip* in den *Animator Controller* und benennen Sie den neuen *Animator State* im *Inspector* in „CloseDoor“ um.
10. Setzen Sie *Speed* von „CloseDoor“ auf -1. Hierdurch wird die Animation rückwärts abgespielt.
11. Legen Sie eine *Transition* von „OpenDoor“ zu „CloseDoor“.
12. Legen Sie auch hier unter *Conditions* „Switch“ als einzige Bedingung fest.
13. Erzeugen Sie noch eine letzte *Transition*, die von „CloseDoor“ zu „Idle“ geht.
14. Legen Sie dort unter *Conditions* die Bedingung *Exit Time* auf 1.00. Hierdurch wird automatisch in „Idle“ gewechselt, sobald der *Animation-Clip* von „CloseDoor“ einmal komplett durchlaufen wurde.
15. Ziehen Sie den *Animator Controller* „DoorController“ auf den *Controller-Slot* vom „Gate“-*Animator*. Sollte von Unity bereits automatisch ein *Animator Controller* erstellt worden sein, der auch der *Animator-Eigenschaft* zugewiesen wurde, können Sie diesen unbenutzten *Animator Controller* löschen und durch „DoorController“ ersetzen.

Am Ende sollten Sie einen *Animator Controller* erhalten, der drei *Animation States* enthält und nur einen einzigen Parameter benötigt. „Any State“ wird in diesem Fall nicht genutzt.



**Bild 17.16** Animator Controller für ein Fallgatter

## ■ 17.5 Controller-Skripte

Zum Ausführen der unterschiedlichen *Animation States* werden Sie in Ihrem *Animator Controller* einen oder mehrere Parameter definiert haben. Im letzten Schritt geht es nun darum, diese Parameter mit Werten zu füllen. Hierfür stellt die *Animator*-Komponente, die das zu animierende *GameObject* besitzt, verschiedene Methoden zur Verfügung. Zunächst braucht das *GameObject* aber ein neues Skript, das auf diesen *Animator* zugreift. Für gewöhnlich wird die Referenz auf diesen in einer *private*- oder *public*-Variablen gespeichert, um nicht unnötig viele Zugriffe zu erzeugen.

**Listing 17.1** Zugriff auf eine Animator-Komponente

```
private Animator anim;
void Start () {
    anim = GetComponent<Animator>();
}
```

### 17.5.1 Parameter des Animator Controllers setzen

Wenn Sie die Verbindung zum *Animator* hergestellt haben, können Sie die Parameter Ihres *Animator Controllers* setzen. Hierfür gibt es für jeden Parametertyp eine eigene Methode. Das folgende Beispiel setzt einen *Float*-Parameter namens „Speed“, der im *Animator Controller* zuvor definiert wurde. Beachten Sie hierbei die Groß- und Kleinschreibung des Variablenamens, die im *Animator Controller* und in diesem Aufruf identisch sein müssen.

**Listing 17.2** Parameter eines Animator Controllers setzen

```
void FixedUpdate () {
    float v = Input.GetAxis ("Vertical");
    anim.SetFloat ("Speed",v);
}
```

#### 17.5.1.1 Mit Hash-Werten arbeiten

Alternativ zum sprechenden Namen können Sie auch den *Hash*-Wert des Parameters übergeben. Der *Hash*-Wert ist im Grunde nichts anderes als eine numerische Darstellung des Namens, den Unity intern nutzt. Und weil Unity diese Umwandlung sowieso macht, erleichtert es den internen Rechenaufwand, wenn dieser nur einmal am Anfang gemacht werden muss und nicht bei jeder Parameterzuweisung. Für diese manuelle *Hash*-Umwandlung stellt die *Animator*-Klasse die statische Methode *StringToHash* zur Verfügung.

**Listing 17.3** Mit Parameter-Hash-Werten arbeiten

```
int speedFloat = 0;
void Start () {
    speedFloat = Animator.StringToHash("Speed");
}
void FixedUpdate () {
```

```

float v = Input.GetAxis ("Vertical");
anim.SetFloat (speedFloat,v);
}

```

In dem obigen Beispiel habe ich den *Hash*-Wert in der *Integer*-Variablen `speedFloat` gespeichert. Dieser Name ist mit Absicht gewählt, da ich auf diese Weise darstelle, dass dies der *Hash*-Wert des *Float*-Parameters „Speed“ ist. Bei einem *Boolean*-Wert könnte die *Integer*-Variable beispielsweise `jumpBool` lauten.

### 17.5.2 Animation States abfragen

Mit `GetCurrentAnimatorStateInfo` stellt Ihnen die *Animator*-Klasse eine Methode zur Verfügung, mit der Sie den aktuellen *Animation State* ermitteln können. Allerdings enthält das Rückgabergebnis nur eine Möglichkeit, den *Hash*-Wert zu ermitteln, nicht den Namen. Deshalb müssen Sie auch hier den zu suchenden *Animation State* erst einmal in seinen *Hash*-Wert umwandeln, was ähnlich wie bei den Parametern funktioniert. Sie müssen lediglich vor dem *State*-Namen auch noch den *Layer* angeben.

**Listing 17.4** Hash-Wert eines Animation States ermitteln

```

int runState;
void Start () {
    runState = Animator.StringToHash("Base Layer.Run");
}

```

Nun übergeben Sie noch den *Layer*-Index (der erste Layer „Base Layer“ hat den Index 0), von dem Sie den aktuellen *Animation State* wissen möchten, und schon können Sie über die Methode `nameHash` den *Hash*-Wert von diesem erhalten. In dem folgenden Beispiel wird überprüft, ob der Charakter aktuell läuft. Ist das der Fall, so wird ein Laufgeräusch von der  *abgespielt.*

**Listing 17.5** Animation State überprüfen

```

void FixedUpdate()
{
    if(anim.GetCurrentAnimatorStateInfo(0).nameHash == runState)
    {
        if(!audio.isPlaying)
            audio.Play ();
    }
    else
    {
        audio.Stop ();
    }
}

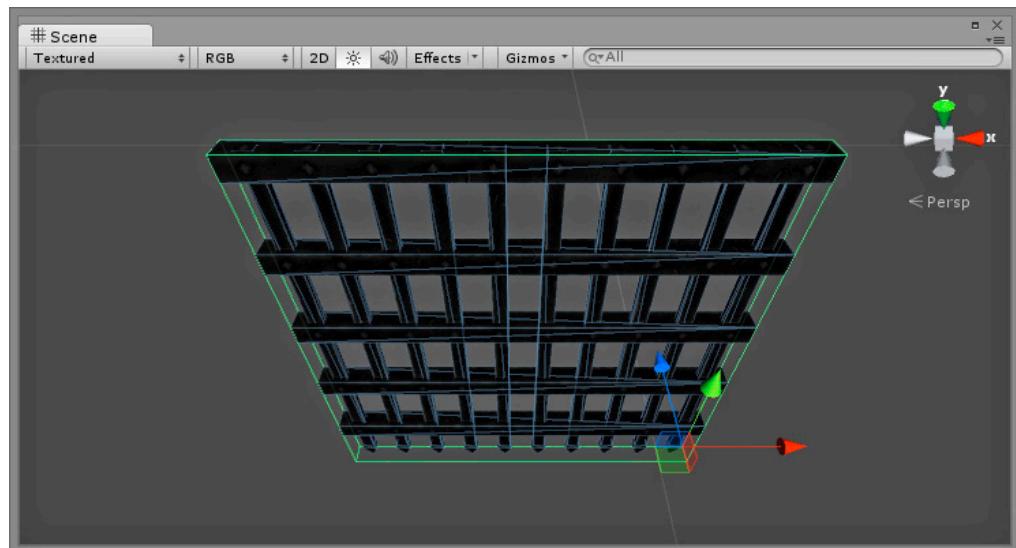
```

### 17.5.3 Beispiel Fallgatter Controller-Skript

Das Fallgatter unseres Beispiel-Games soll nun mit einem Controller-Skript ausgestattet werden. Das Rauf- und Runterfahren soll dabei durch einen Mausklick des Spielers auf das Gatter ausgelöst werden. Hierfür wollen wir die Methode `OnMouseDown` nutzen, die ausgelöst wird, sobald auf den *Collider* des *GameObjects* geklickt wird. Deshalb müssen wir zunächst dafür sorgen, dass das *GameObject*, in unserem Fall ist es das „gate\_01“, auch tatsächlich einen *Collider* besitzt.

Die einfachste Möglichkeit ist das Aktivieren von *Generate Colliders* in den *Import Settings* der FBX-Datei von „gate\_01“. Hierdurch wird dem *GameObject* automatisch ein *MeshCollider* hinzugefügt. *MeshCollider* haben allerdings einige Nachteile. Neben den im Kapitel „Physik in Unity“ bereits angesprochenen, gibt es hier noch einen weiteren. Durch die Gitterform des *Mesh* kann es vorkommen, dass der Spieler nicht direkt auf eine Strieme des Gatters klickt, sondern in ein Loch des Gatters. In dem Fall würde die Methode `OnMouseDown` nicht ausgelöst werden. Deshalb empfehle ich, statt der oberen Variante manuell einen *BoxCollider* hinzuzufügen.

Markieren Sie hierfür in der Szene das Kind-Objekt „gate\_01“ und führen **Component/Physics/Box Collider** aus. Der *BoxCollider* sollte sich automatisch der Größe des *Mesh* anpassen. Sollte das nicht der Fall sein, müssten Sie über die *Collider*-Eigenschaften *Center* und *Size* im *Inspector* noch etwas nachhelfen.



**Bild 17.17** Fallgatter mit einem BoxCollider

Wenn unser animiertes Objekt nun einen *Collider* besitzt, sollten Sie jetzt noch sicherstellen, dass das *GameObject* auch ein *Rigidbody* hat. Wie ich bereits im Kapitel „Physik in Unity“ erläutert hatte, sollten alle *GameObjects* mit *Collidern*, die sich im Spielverlauf bewegen bzw. bewegen könnten, ein *Rigidbody* besitzen. Fügen Sie deshalb dem „gate\_01“ einen hinzu, deaktivieren Sie *Use Gravity* und aktivieren Sie *Is Kinematic*.

Für das eigentliche *Controller*-Skript erzeugen Sie jetzt ein neues Skript mit dem Namen „AnimateDoor“, indem Sie den einzigen *Trigger*-Parameter „Switch“ aufrufen.

#### **Listing 17.6** Einfaches Controller-Skript eines Fallgatters

```
using UnityEngine;
using System.Collections;
public class AnimateDoor : MonoBehaviour {
    Animator anim;
    int switchTrigger;
    void Start () {
        //Holt den Hash-Wert des Trigger-Parameters Switch
        switchTrigger = Animator.StringToHash ("Switch");
        //Holt den Animator vom Elternobjekt
        anim = transform.parent.GetComponent<Animator>();
    }
    void OnMouseDown() {
        //Loest den Trigger Switch aus.
        anim.SetTrigger (switchTrigger);
    }
}
```

Eine Besonderheit des obigen Skriptes ist die Zuweisung des Animators. Da der *Collider* dem Kind-Objekt „gate\_01“ gehört, muss nämlich auch das Skript diesem zugewiesen werden. Der steuernde *Animator* gehört aber dem Eltern-Objekt „Gate“. Deshalb muss der Variablen *anim* der *Animator* des Eltern-Objekts zugewiesen werden. Über die *parent*-Eigenschaft der *Transform*-Komponente ist dies möglich.

Nachdem Sie das Skript verfasst haben, fügen Sie dem Kind-Objekt „gate\_01“ dieses Skript zu. Wenn Sie nun die „Main Camera“ vor dem Fallgatter positionieren, können Sie die Szene starten und die Funktionen testen.

##### **17.5.3.1 Controller-Skript erweitern**

Das obige Skript erfüllt funktional alles, was die Animation benötigt, um gestartet zu werden. Allerdings möchten wir in diesem Fall das Rauf- und Runterfahren des Fallgatters mit Sound untermalen. Hierfür nutzen wir die statische *PlayClipAtPoint*-Methode der *AudioSource*-Klasse, die automatisch eine temporäre  *AudioSource* erstellt. Diese wird dann zusätzlich in der *OnMouseDown*-Methode erzeugt. Außerdem benötigen wir hierfür noch eine *public*-Variable, der wir den  *AudioClip* zuweisen. Zur einfacheren Lesart nennen wir diese einfach mal *doorClip*.

#### **Listing 17.7** Fallgatter mit Sound-Untermalung

```
using UnityEngine;
using System.Collections;
public class AnimateDoor : MonoBehaviour {
    public AudioClip doorClip;
    Animator anim;
    int switchTrigger;
    void Start () {
        switchTrigger = Animator.StringToHash ("Switch");
        anim = transform.parent.GetComponent<Animator>();
    }
}
```

```

void OnMouseDown() {
    anim.SetTrigger (switchTrigger);
    AudioSource.PlayClipAtPoint(doorClip,transform.position);
}
}

```

Jetzt brauchen Sie der Variablen nur noch einen *AudioClip* zuzuweisen, und schon haben wir Sound. Nutzen Sie hierfür den beiliegenden Clip „chains“. Allerdings haben wir jetzt noch das Problem, dass Sie auf das Tor klicken und dadurch bereits die nächste Animation starten können, bevor das Fallgatter an seiner Endposition angekommen ist. Dies soll noch unterbunden werden. Im nachfolgenden Abschnitt „Animation Events“ werden wir dieses Skript deshalb noch etwas erweitern.

## ■ 17.6 Animation Events

Nicht selten gibt es die Notwendigkeit, einen *Animation-Clip* mit anderen Skript-Methoden zu synchronisieren. Dabei ist es vor allem dann herausfordernd, wenn ein bestimmter Moment der Animation mit anderen Vorgängen synchronisiert werden soll, z. B. das Instanziieren eines Feuerballs, wenn der Arm des Zauberers ganz ausgestreckt ist. Für diesen Zweck gibt es in Unity *Animation Events*, die sowohl in den *Import Settings* des *Animations-Reiters* wie auch beim Erstellen des *Animation-Clips* direkt im *Animation View* erzeugt werden können. *Animation Events* können Methoden aus beliebigen Skripten aufrufen, wobei diese genau einen Übergabeparameter besitzen müssen. Dies kann ein *String*, *Int*- oder ein *Float*-Parameter sein. Wenn Sie das *Event* im *Animation View* definieren, können Sie zusätzlich auch noch den Typ *Object* nutzen, was ein universeller Parametertyp ist.

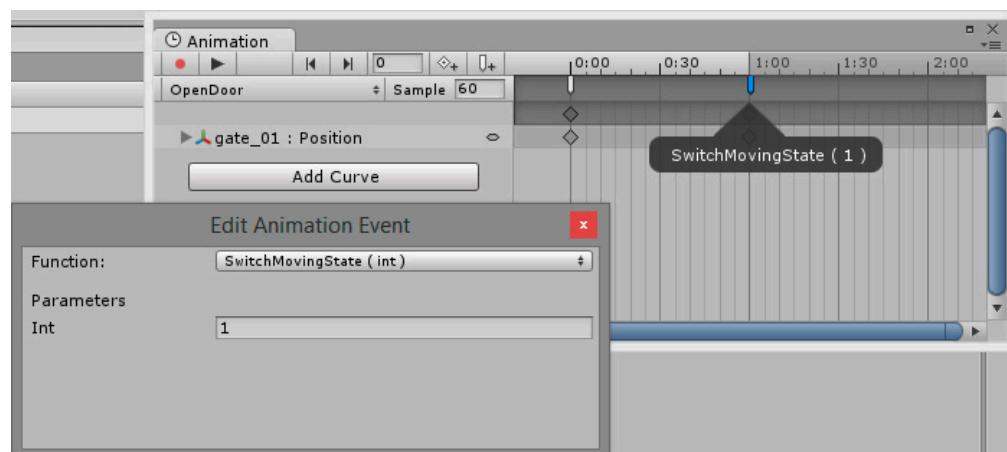


Bild 17.18 Animation Event

Bitte beachten Sie, dass auf diese Weise eingebundene Skripte auch später dem animierten *GameObject* zugefügt werden. Ansonsten versucht das *Animation Event* auf ein Skript zuzugreifen, das nicht existiert.

### 17.6.1 Beispiel Fallgatter-Bewegung

In dem Fallgatter-Beispiel soll abschließend noch verhindert werden, dass auf dieses mehrfach geklickt werden kann und damit schon während des Hochfahrens das folgende Herunterfahren ausgelöst wird. Deshalb muss festgestellt werden, wann sich das Gatter gerade auf- oder abbewegt.

Das Schließen ist hierbei recht einfach zu ermitteln. Denn wie im Kapitel „Animation States abfragen“ beschrieben, kann man prüfen, ob der *State* „CloseDoor“ aktuell ausgeführt wird. Und sobald das Tor geschlossen ist, wechselt der *State* in „Idle“. Allerdings wird am Ende des Öffnens nicht der *State* geändert. Damit das Tor nicht gleich wieder schließt, bleibt der *Animation Controller* im *State* „OpenDoor“ und hält das animierte *Transform* auf der Endposition der Animation.

Dies führt nun zu dem Dilemma, dass wir nicht feststellen können, wann das Fallgatter beim Hochfahren tatsächlich an seiner Endposition angekommen ist. Um dieses Problem zu lösen, werden wir hierfür nun *Animation Events* nutzen. Diese sollen so eingesetzt werden, dass in einem Skript eine Variable den Bewegungszustand des Fallgatters darstellt.

#### 17.6.1.1 Angesteuertes Skript

Für die obige Aufgabe benötigen wir lediglich ein kleines Skript namens *DoorMovingState*, das die kleine Methode *SwitchMovingState* besitzt. Wird die Methode aufgerufen, wechselt einfach nur der Zustand der *isMoving*-Variablen. Fügen Sie dem Eltern-Objekt „Gate“ dieses Skript zu.

**Listing 17.8** Skript zum Anzeigen des Bewegungsstatus

```
using UnityEngine;
using System.Collections;

public class DoorMovingState : MonoBehaviour {
    public bool isMoving=false;

    void SwitchMovingState(int id)
    {
        isMoving = !isMoving;
        //Debug.Log (id);
    }
}
```

Aufgrund der Anforderung, dass Methoden, die von *Animation Events* aufgerufen werden sollen, einen Parameter besitzen müssen, besitzt die Methode *SwitchMovingState* einen *int*-Parameter *id*, den man später ggf. noch auswerten kann.

### 17.6.1.2 Animation Events hinzufügen

Zum Steuern der Variablen benötigt die Originalanimation zwei Animation Events, die wir nun im Nachhinein dem *Animation-Clip* noch zufügen wollen.

1. Selektieren Sie das Objekt „Gate“.
2. Öffnen Sie den *Animation View (Window/Animation)*.
3. Wählen Sie die Animation „OpenDoor“ aus.
4. Klicken Sie in der *Dope Sheet*-Ansicht auf einen der linken *Keyframes*, sodass die *rote Keyframe-Linie* ebenfalls dort erscheint.
5. Erzeugen Sie über **Add Event** ein neues *Animation Event*. Unterhalb der Zeitachse erscheint nun ein neues Event-Symbol, das dieses darstellt (siehe Bild 17.19).
6. Wählen Sie in dem neu erscheinenden Fenster über *Function* die Methode *Switch MovingState* aus, weisen Sie *id* den Wert „0“ zu und schließen Sie das Fenster.
7. Selektieren Sie nun einen der rechten *Keyframes*, damit die *rote Keyframe-Linie* dort erscheint.
8. Wiederholen Sie die Punkte 5 und 6, aber legen Sie den Parameter *id* auf 1.

Für die Implementierung der *Animation Events* war es das schon. Jetzt muss nur noch zum Lösen der eigentlichen Problemstellung die gesteuerte Variable in das eigentliche *Controller*-Skript integriert werden.

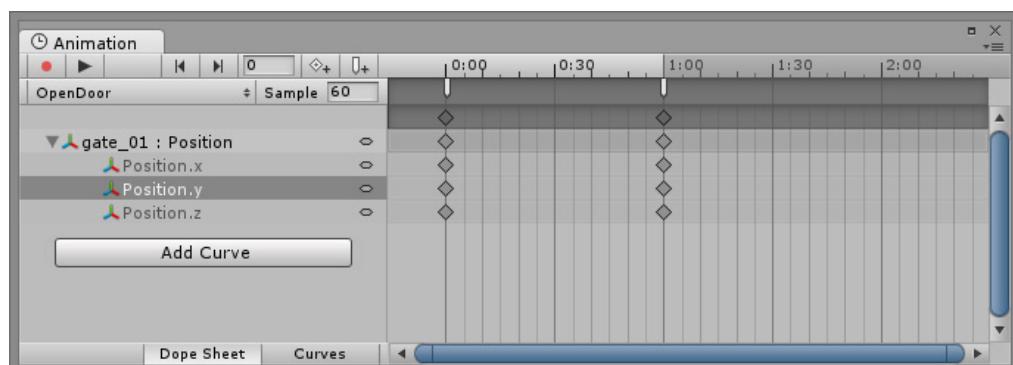


Bild 17.19 Animation-Clip mit zwei Animation Events

### 17.6.1.3 Parameter in Controller-Skript implementieren

Damit wir den Parameter in dem *Controller*-Skript auch nutzen können, müssen wir das Skript zunächst in unser Skript einer Variablen zuweisen. Da *DoorMovingState* aber dem Parent-Objekt zugefügt wurde, müssen wir uns dieses vom Parent des *Transforms* holen. Der eigentliche Einsatz findet dann in *OnMouseDown* statt, wo wir den ursprünglichen Code nur dann ausführen, wenn der Wert von *isMoving* FALSE ist, also das Tor nicht in Bewegung ist.

**Listing 17.9** Fallgatter-Skript mit Bewegungsüberprüfung

```
using UnityEngine;
using System.Collections;

public class AnimateDoor : MonoBehaviour {
    public AudioClip doorClip;

    private Animator anim;
    private int switchTrigger;
    private DoorMovingState doorMovingState;
    // Use this for initialization
    void Start () {
        switchTrigger = Animator.StringToHash ("Switch");
        anim = transform.parent.GetComponent<Animator>();
        doorMovingState = transform.parent.GetComponent<DoorMovingState>();
    }

    void OnMouseDown() {
        if (!doorMovingState.isMoving)
        {
            anim.SetTrigger (switchTrigger);
            AudioSource.PlayClipAtPoint(doorClip,transform.position);
        }
    }
}
```

# 18

## Künstliche Intelligenz

Ein besonders anspruchsvoller Aspekt eines Spiels bzw. der Spieleprogrammierung ist die Entwicklung von Nichtspielercharakteren (NSCs bzw. engl. NPCs) und deren künstlicher Intelligenz. Leider gibt es nicht die eine *Künstliche Intelligenz* (kurz KI), die man immer wieder nutzen kann. Je nach Art des Spiels und Aufgabe des NPC muss die KI andere Dinge können. Die KI eines Schachspiels ist natürlich eine vollkommen andere als die eines Vogelschwarm in einem Adventure-Game. Aber auch die Fähigkeiten eines Nahkämpfers in einem Beat 'em Up unterscheidet sich von der Nahkämpfer-KI eines Rollen- oder Strategiespiels. Vieles muss deshalb individuell für das jeweilige Spiel entwickelt werden.

Allerdings gibt es auch Aufgaben, die sich ähneln. Hierzu gehört das sogenannte *Pathfinding*, die Wegfindung zwischen zwei Punkten. Wollen Sie zum Beispiel einen NPC von einem Raum in einen anderen gehen lassen, dann muss die *Künstliche Intelligenz* natürlich feststellen, wie dieser dort hinkommt, ohne andere Gegenstände umzurennen und am besten auch ohne großartige Umwege zu machen.

Für solche Problemstellungen gibt es natürlich bereits schon etliche Algorithmen, die von findigen Leuten entwickelt wurden, z. B. der A\*-Algorithmus. Da solche *Pathfinding*-Verfahren doch einiges an Programmierarbeit bedeuten, bietet Unity hierfür fertige Funktionalitäten an, die Ihnen beim Finden des optimalen Weges helfen. *Pathfinding* wird aber nicht nur bei NPCs eingesetzt. Es wird genauso für die Spielersteuerung in Point & Click-Adventures genutzt oder zum Setzen von Armeen in Strategiespielen. Am Ende dieses Kapitels werde ich hierauf noch einmal etwas genauer eingehen.

Ein ausführliches Anwendungsbeispiel für die folgenden Funktionalitäten finden Sie im Kapitel „Beispiel-Game“. Dort werden wir im Abschnitt „Gegner erstellen“ eine KI für einen NPC entwickeln, die die folgenden Pathfinding-Funktionen nutzt und diese auch mit dem Animationssystem kombiniert.

## ■ 18.1 NavMeshAgent

Die eigentliche Intelligenz der Wegberechnung befindet sich in der *NavMeshAgent*-Komponente. Diese müssen Sie dem wegsuchenden Objekt, z.B. einem NPC, zufügen. Sie finden den *NavMeshAgent* im Menü über **Component/Navigation/Nav Mesh Agent**.

Die Komponente berechnet den optimalen Weg von der eigenen Position zu einem vorgegebenen Punkt und greift hierfür auf die Informationen des *NavigationMesh* zu, das die Informationen über die Umgebung speichert. Der *NavMeshAgent* übernimmt aber nicht nur die Berechnung des Weges, sondern auch gleichzeitig die Steuerung des Objektes, um dieses zum gewünschten Zielpunkt zu bringen. Beachten Sie dabei, dass jedes Objekt mit einem *NavMeshAgent* auch ein *Rigidbody* mit aktivem *Is Kinematic*-Parameter besitzt sollte (siehe Kapitel „Physik in Unity“).

### 18.1.1 Eigenschaften der Navigationskomponente

Da der *NavMeshAgent* sowohl für Wegberechnung wie auch für die Steuerung zuständig ist, benötigt dieser auch einige Parameter:

- **Radius** legt den Radius des Objektes für die Wegberechnung fest. Da anhand dieses Wertes der einzuhaltende Abstand zu anderen Objekten ermittelt wird, kann er gerne etwas größer als das eigentliche Objekt sein.
- **Speed** definiert die maximale Bewegungsgeschwindigkeit des Objektes.
- **Acceleration** legt die maximale Beschleunigung fest. Gleichzeitig bestimmt dieser Wert auch die negative Beschleunigung, also die Verzögerung.
- **Angular Speed** definiert die maximale Drehgeschwindigkeit in Grad/Sekunde. Beachten Sie hierbei, dass dies nicht die tatsächliche Drehgeschwindigkeit des *Agents* ist. Diese hängt noch zusätzlich von den Parametern *Speed* und *Acceleration* ab und lässt sich leider nicht direkt über diesen Parameter bestimmen.
- **Stopping Distance** stellt einen Abstand ein, ab dem der Agent anfängt zu bremsen (siehe *Auto Braking*).
- **Auto Traverese Off Mesh Link** automatisiert die Bewegung auf und ab von einem *Off MeshLinks*.
- **Auto Braking** bestimmt, ob der *Agent* genau bei der Entfernung von **Stopping Distance** stehen bleibt (aktiviert) oder ob er bei dieser Distanz anfängt zu bremsen (deaktiviert). Der Vorteil eines deaktivierten *Auto Braking* ist ein realistisch wirkendes Verhalten, gerade wenn das Objekt recht schnell ist. Allerdings muss dann mit den anderen Parametern wie *Acceleration* und *Stopping Distance* getestet werden, wie hoch diese Werte sein müssen, damit das Objekt auch am Ende genau dort steht, wo es hin soll.
- **Auto Repath** berechnet automatisch einen neuen Pfad, wenn sich der aktuelle als fehlerhaft erweist.
- **Height** legt die Höhe des *Agents* fest.

- **Base Offset** legt ein vertikales Offset fest, um das eigentliche Objekt mit dem *MeshRenderer* besser zu platzieren.
- **Obstacle Avoidance Type** legt die Qualität beim Ausweichen gegenüber anderen Gegenständen fest. Wie immer gilt auch hier: Je genauer/besser, desto performancehungriger.
- **Avoidance Priority** definiert eine Priorität beim Objektausweichen. *Agents* mit niedriger Priorität werden von diesem Agent ignoriert.
- **NavMesh Walkable** definiert die *Navigation Layer* (siehe unten), die von diesem *Agent* bei der Wegberechnung berücksichtigt werden. Lediglich „Not Walkable“ wird immer ignoriert (siehe *Layers*).

### 18.1.2 Zielpunkt zuweisen

Die *NavMeshAgent*-Komponente besitzt etliche Methoden und Eigenschaften, mit denen Sie den *Agent* kontrollieren können. Die aber wohl wichtigste Methode heißt `SetDestination`. Sie weist der *NavMeshAgent*-Komponente das Ziel zu, das ihr sagt, wohin es zu gehen hat. Nachdem Sie eine Position im *Vector3*-Format übergeben haben, berechnet der *Agent* den Weg zu diesem Punkt und beginnt, dorthin zu „laufen“.

**Listing 18.1** Einem *NavMeshAgent* ein Ziel zuweisen

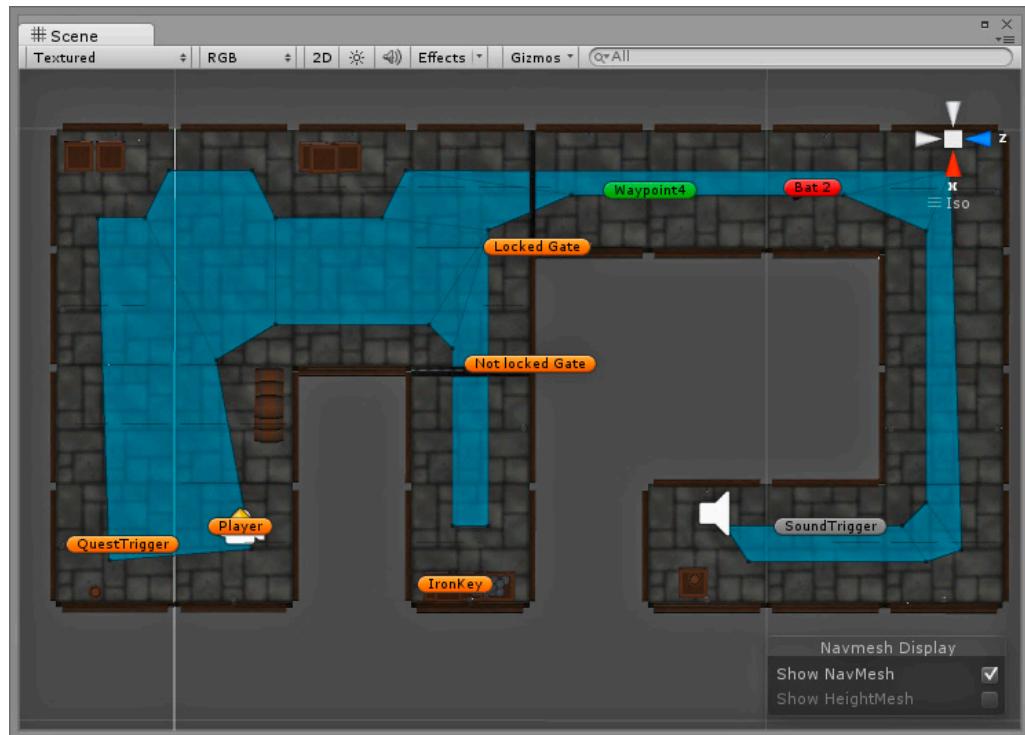
```
public Vector3 dest;
private NavMeshAgent agent;
void Start() {
    agent = GetComponent<NavMeshAgent>();
    agent.SetDestination(dest);
}
```

Über die *Agent*-Variable `destination` können Sie jederzeit den aktuellen Zielpunkt erfahren. Da diese Variable nicht nur Lese-, sondern auch Schreibzugriffe erlaubt, können Sie auch über diese alternativ das Ziel zuweisen.

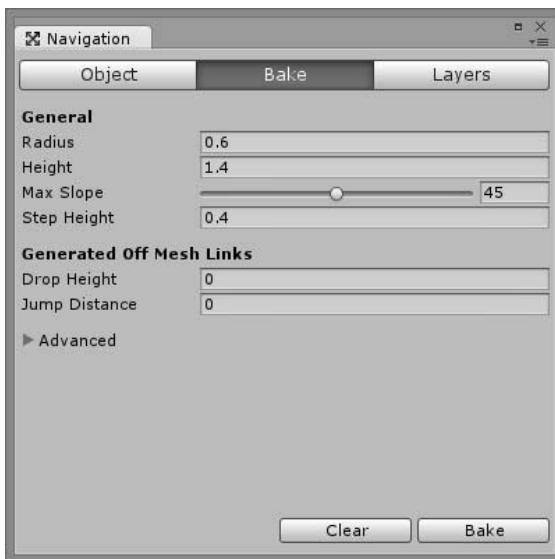
## ■ 18.2 NavigationMesh

Neben einem *GameObject* mit einem *NavMeshAgent* benötigen Sie als Zweites noch ein *NavigationMesh*. Das *NavigationMesh*, kurz *NavMesh*, beinhaltet alle Informationen über die Begehbarkeit der tatsächlichen *Meshes* einer Szene. Es beschreibt, welche Flächen begehbar sind, wo sich Hindernisse befinden und welche Wege zu bevorzugen sind. Anhand dieser Informationen wird dann vom *NavMeshAgent* der optimale Weg zum Zielpunkt, der sich natürlich auf diesem *NavMesh* befinden muss, berechnet.

Um ein *NavMesh* zu erzeugen, müssen Sie zunächst das *Navigation*-Fenster öffnen, das Sie über **Window/Navigation** erreichen. Das Fenster beinhaltet drei Tabs, die für die Erstellung des *NavMesh* zuständig sein sollen: *Object*, *Bake* und *Layers*.



**Bild 18.1** Berechnetes NavigationMesh



**Bild 18.2**  
Navigation-Fenster

Haben Sie die notwendigen Einstellungen getätigt, können Sie unten in dem Fenster über den Knopf **Bake** das *NavigationMesh* jederzeit erstellen und über **Clear** wieder löschen.

## 18.2.1 Object Tab

Die Wegberechnung in Unity hat einige Grenzen. So kann Unity nur den optimalen Weg anhand von statischen Objekten berechnen. Es ist also aktuell nicht möglich, einer KI das Nutzen von beweglichen Plattformen oder Ähnliches zu ermöglichen, jedenfalls nicht mit Hilfe der integrierten *Pathfinding*-Funktionalität. Deshalb muss vor dem Erstellen des *NavMesh* zunächst einmal gesagt werden, welche Objekte in der Szene statisch sind und bei der Berechnung berücksichtigt werden dürfen. Genau dies können Sie in dem Tab *Object* machen.

- **Scene Filter** filtert die in der *Hierarchy* angezeigten Objekte vor.
- **Navigation Static** definiert ein Objekt als statisch. Selektieren Sie einfach ein Objekt in der *Hierarchy* und aktivieren Sie diese Eigenschaft. Damit wird das jeweilige Objekt für das *Pathfinding* als statisch betrachtet und fließt mit in die Wegberechnung ein. Natürlich können Sie auch mehrere oder auch gleich alle Objekte auf einmal selektieren und den Parameter für alle gleichzeitig setzen. Alternativ können Sie auch direkt über das *Static-Menü* im *Inspector* des jeweiligen Objektes den Parameter *Navigation Static* aktivieren.
- **Off-Mesh Link Generation** verbindet zwei begehbar Flächen bzw. *NavMeshes*. Ein *NavMesh* ist immer eine durchgehende Fläche. Wenn Sie zwei Flächen besitzen, die z.B. durch einen Höhenunterschied oder einen Fluss getrennt sind, dann sind dies zwei verschiedene *NavMeshes*. Damit aber auch hier ein Weg von der einen zur anderen Fläche führen kann, gibt es *Off-Mesh Links*. Sie definieren die Übergänge zwischen diesen. Aktuell ist dies ein Unity Pro-Feature.
- **Navigation Layer** legt den jeweiligen *Navigation Layer* für das aktuelle Objekt fest. Diese speziellen *Layer* werden im Bereich *Layers* definiert und beschreiben zusätzliche Eigenschaften für die Wegberechnung. „Not Walkable“ definiert hierbei, dass dieser Bereich nicht begehbar ist und somit bei der Wegberechnung ignoriert wird. Aktuell ist dies ebenfalls ein Unity Pro-Feature.

## 18.2.2 Bake Tab

In diesem Bereich definieren Sie, welche Eigenschaften ein Bereich besitzen muss, um vom *NavMeshAgent* begehbar zu sein. Die Voreinstellungen sind bereits gut gewählt, können aber natürlich beliebig geändert werden.

- **Radius** legt fest, wie nah das *NavMesh* an eine Wand maximal gehen darf. Legen Sie diesen Wert minimal auf den Radius des *NavMeshAgents* fest, da ansonsten der *Agent* mit den Wänden kollidiert. Deshalb ist ein höherer Wert auch nicht kritisch, kleiner sollte er aber nicht sein.
- **Height** definiert, wie viel Freiraum nach oben gegeben sein muss. Dieser Wert sollte mindestens der Höhe des *NavMeshAgents* entsprechen.
- **Max Slope** legt die maximale Steigung fest, die der *Agent* hochgehen kann/darf.
- **Step Height** definiert die maximale Stufenhöhe, die der *Agent* überwinden kann.
- **Drop Height** legt eine maximale Fallhöhe fest, die er herunterfallen bzw. herunterspringen kann.

- **Jump Distance** definiert die maximale Sprungweite des *Agents*.
- **Min Region Area** bestimmt die minimale Größe einer Fläche, die als *NavMesh* definiert werden kann.
- **Width Inaccuracy** legt die Genauigkeit der Berücksichtigung des Radiuswertes in der *NavMesh*-Berechnung fest. Je kleiner dieser Wert ist, desto genauer ist, aber auch länger dauert das *Baken*.
- **Height Inaccuracy** regelt die Genauigkeit der Berücksichtigung des *Height*-Wertes in der *NavMesh*-Berechnung.

### 18.2.3 Layers Tab

In diesem Bereich definieren Sie die *Navigation Layer*, die Sie im Bereich **Object** zuweisen können. Jeder *Navigation Layer* hat sogenannte Kosten, englisch *Cost*, die bei der Berechnung des optimalen Pfades herangezogen werden.

Ein gutes Anschauungsbeispiel für die Nutzung von diesen *Layern* ist die Entscheidung, ob es besser ist, den kurzen Weg durch das Wasser oder den über die weiter entfernte Brücke zu nehmen. Hierfür weisen Sie dem Objekt des Flusses einen *Layer* mit höheren *Costs* und der Brücke einen *Layer* mit geringeren *Costs* zu. Hierdurch kann nun der *Agent* den Weg über die Brücke wählen, obwohl er theoretisch auch durch das Wasser gehen könnte. Lediglich beim Layer „Not Walkable“ werden die Kosten bei der Berechnung komplett ignoriert, da Objekte mit diesem *Layer* bei der Wegberechnung gar nicht beachtet werden.

## ■ 18.3 NavMeshObstacle

Wie Sie vielleicht bereits bemerkt haben, hat das *Pathfinding* einen gravierenden Nachteil: Das Verfahren berücksichtigt nur statische Objekte. Das bedeutet aber nicht nur, dass hier ausschließlich statische Wege berücksichtigt werden, es werden auch statische Hindernisse beachtet.

Schon eine Tür stellt ein großes Problem dar. Entweder ist sie ein Hindernis oder sie ist immer durchschreitbar. Werden Türen normalerweise immer im geschlossenen Zustand modelliert, kann diese beim *Baken* entweder *static* oder nicht *static* sein.

Um dieses Problem nun zu lösen, gibt es die *NavMeshObstacle*-Komponente, die aber leider aktuell nur in der Unity Pro-Version zur Verfügung steht. Zum Nutzen dieser Komponente definieren Sie das bewegliche Objekt, z. B. die Tür, als *Navigation Static* und fügen diesem dann eine *NavMeshObstacle*-Komponente zu (**Component/Navigation/Nav Mesh Obstacle**). Und schon wird das Objekt nach dem *Baken* wie gewünscht als bewegliches Objekt berücksichtigt.

Wenn Sie nicht die Pro-Version besitzen, können Sie sich häufig auch mit kleinen Skripten behelfen, wie Sie in dem Beispiel-Game dieses Buches noch sehen werden. Aber generell ist das Nutzen so einer *NavMeshObstacle*-Komponente natürlich wesentlich einfacher und auch effektiver, da es logischerweise auf das integrierte *Pathfinding*-System abgestimmt ist.

## ■ 18.4 Point & Click-Steuerung für Maus und Touch

Mithilfe der ScreenPointToRay-Methode der Camera-Klasse und der SetDestination-Methode vom *NavMeshAgent* können Sie sehr einfach eine Spielersteuerung eines Point & Click-Spiels entwickeln. Das folgende Beispiel könnte ein Teil so eines Spieler-Controllers sein. In diesem überprüft das Skript in der Update-Methode, ob der Spieler mit seiner Maus in das Spiel klickt, und steuert dann das *GameObject* des *NavMeshAgents* zu dem jeweiligen Punkt.

**Listing 18.2** Einfache Point & Click-Maussteuerung

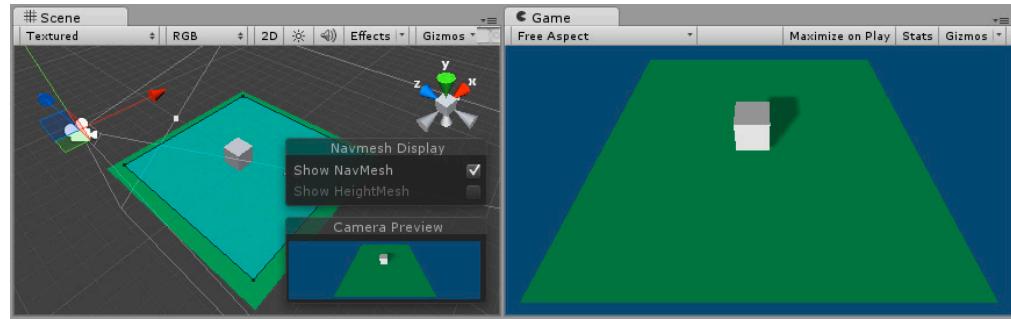
```
using UnityEngine;
using System.Collections;
public class Pathfinding : MonoBehaviour {
    private NavMeshAgent agent;
    void Awake() {
        //Die Komponente wird einer Variablen zugewiesen
        agent = GetComponent<NavMeshAgent>();
    }
    void Update() {
        RaycastHit hit;
        //Wurde die linke Maustaste gedrückt?
        if (Input.GetMouseButtonDown(0)) {
            //Erzeuge einen Strahl,
            //der von der aktuellen Mausposition in die Szene hineinragt
            Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
            // Taste mit dem Strahl ab, ob dieser ein Objekt berührt
            if (Physics.Raycast(ray, out hit))
                agent.SetDestination(hit.point); //Neuen Zielpunkt setzen
        }
    }
}
```

Zum Testen des obigen Skriptes brauchen Sie lediglich eine Szene mit einer Plane und einem Cube, den Sie auf dieser platzieren. Fügen Sie dem Cube einen *NavMeshAgent* und ein *Rigidbody* zu. Aktivieren Sie beim *Rigidbody Is Kinematic* und deaktivieren Sie *Use Gravity*. Außerdem fügen Sie dem Cube das obige Skript zu.

Zum Schluss müssen Sie jetzt noch das *NavMesh* erstellen. Deklarieren Sie zunächst alle *Static*-Objekte, was in diesem Fall nur die Plane ist, und starten Sie über den **Bake**-Button im *Navigation*-Fenster das Erstellen des *NavMesh*.

Nach dem Erstellen können Sie auf eine beliebige Stelle der Plane klicken, zu der dann der Cube automatisch hinläuft. Hierfür sollte die Kamera am besten von oben oder schräg von oben auf die Plane schauen. Ansonsten kann der Spieler nicht oder zumindest schlecht auf die Plane klicken, um einen neuen Zielpunkt zu setzen. Das Bild 18.3 zeigt diese Szene, wobei der Plane zur besseren Darstellung noch ein grünes Material zugewiesen wurde.

Dieses Beispiel eignet sich auch hervorragend für Spiele mit einer Touch-Steuerung. Dabei ist eine reine Top-down-Kamerasicht besonders vorteilhaft, da der Spieler auf diese Weise eine größere Fläche hat, um den jeweiligen Zielpunkt auf dem Untergrund zu berühren.



**Bild 18.3** Testszene

Positionieren Sie hierfür die Kamera genau über der Spielfläche und drehen Sie diese 90 Grad um die X-Achse.

Der folgende Code unterscheidet sich von dem vorherigen Skript dementsprechend nur in der Auswertung der Eingabe. Denn statt der Maus werden hier die Touch-Eingaben, genauer gesagt die erste Touch-Eingabe, ausgewertet. Mehr zur Touch-Auswertung erfahren Sie im Kapitel „Maus, Tastatur, Touch“.

**Listing 18.3** Einfache Point & Click-Touch-Steuerung

```
using UnityEngine;
using System.Collections;
public class TouchPathfinding : MonoBehaviour {
    private NavMeshAgent agent;
    void Awake() {
        agent = GetComponent<NavMeshAgent>();
    }

    void Update () {
        RaycastHit hit;
        if (Input.touchCount > 0)
        {
            if(Input.touches[0].phase == TouchPhase.Began)
            {
                Ray ray = Camera.main.ScreenPointToRay(Input.touches[0].position);
                if (Physics.Raycast(ray, out hit))
                    agent.SetDestination(hit.point); //Neuer Zielpunkt
            }
        }
    }
}
```

# 19

# Fehlersuche und Performance

Unity gibt Ihnen verschiedene Werkzeuge an die Hand, um Ihr Spiel genauer zu analysieren, Fehler zu finden und die Qualität des Spiels zu verbessern. Im Folgenden möchte ich Ihnen zeigen, wie Sie im Programmcode nach Fehler suchen und die Performance analysieren können.

## ■ 19.1 Fehlersuche

Kein Spiel ist ohne Programmierfehler, jedenfalls nicht von Anfang an. Und deshalb gibt es für die Fehlersuche im Programmcode, das sogenannte Debugging, in MonoDevelop einen hilfreichen Debugger. Mit diesem können Sie, während Sie Ihr Game testen, das gesamte Spiel anhalten und Schritt für Schritt Ihre Skripte durchlaufen, um zu beobachten, wie sich Ihr Code und die Variablen im Spielverlauf verhalten.

Da dies aber Unity ausbremst, müssen Sie vor solch einem Test zuerst MonoDevelop an den Unity-Prozess hängen. Dies machen Sie im MonoDevelop-Menü über **Run/Attach To Process**.

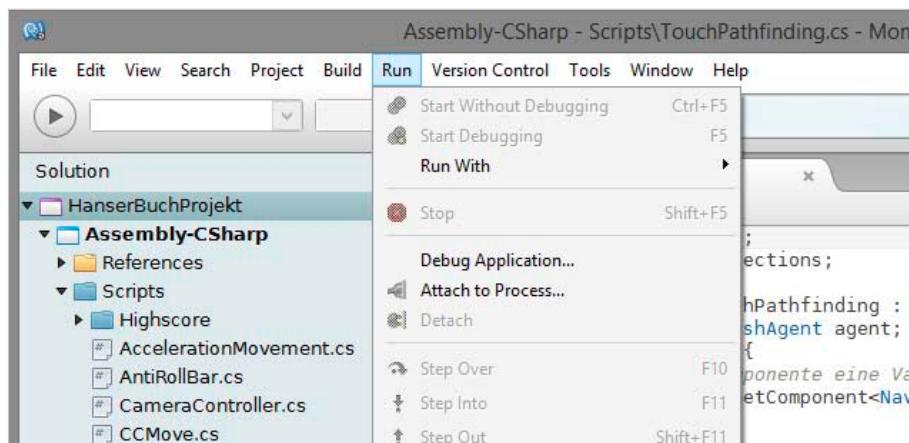


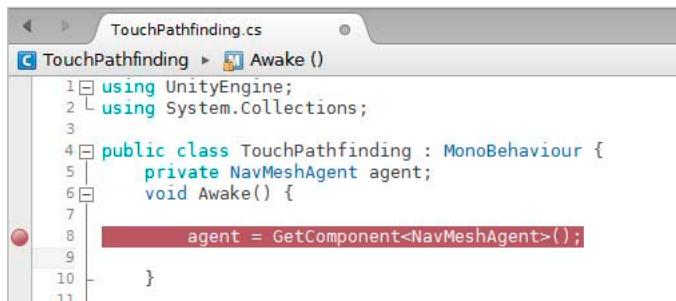
Bild 19.1 Run-Menü

Für gewöhnlich wird hier ein Prozess mit dem Namen „Unity Editor (Unity)“ angezeigt, den Sie selektieren und mit **Attach** bestätigen müssen.

Starten Sie jetzt in Unity Ihr Game, werden Sie bereits jetzt merken, dass dieser Vorgang nun länger dauert. Da dies bei kleineren Tests nur unnötig viel Zeit dauert, ist es ratsam, das Anhängen von MonoDevelop an Unity nur dann zu machen, wenn Sie auch wirklich testen möchten. Das Beenden des Debuggings machen Sie dann über **Run/Detach**. Sie können das übrigens jederzeit machen, also sowohl im Spielverlauf als auch im gestoppten Zustand.

### 19.1.1 Breakpoints

Mit einem *Breakpoint*, zu Deutsch Haltepunkt, können Sie im Programmcode einen Kennen setzen, der das Spiel anhält, bevor diese Zeile ausgeführt wird. *Breakpoints* werden mit **[F9]** erstellt und links neben der Programmzeile braunlich dargestellt. Um einen *Breakpoint* wieder wegzunehmen, drücken Sie ebenfalls **[F9]**.



The screenshot shows the MonoDevelop IDE with the file "TouchPathfinding.cs" open. The code is as follows:

```

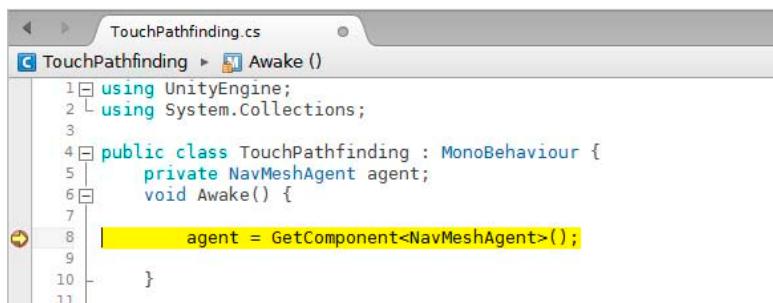
1 using UnityEngine;
2 using System.Collections;
3
4 public class TouchPathfinding : MonoBehaviour {
5     private NavMeshAgent agent;
6     void Awake() {
7
8         agent = GetComponent<NavMeshAgent>();
9
10    }
11

```

A red circular icon with a white dot, representing a breakpoint, is positioned to the left of the line number 8. The line containing the breakpoint is highlighted in red.

**Bild 19.2**  
Breakpoint im Code

Wenn Sie nun MonoDevelop an Unity hängen und das Spiel starten, hält das Spiel sofort an, sobald der Code bei diesem Breakpoint ausgeführt werden soll. Dabei wird die Zeile, die als Nächstes ausgeführt werden soll, immer gelb markiert. Am Anfang ist es also die Zeile, wo sich der *Breakpoint* befindet. Mit der Taste **[F11]** können Sie nun den Code dieses Skriptes Zeile für Zeile ausführen und verfolgen, was passiert und wo der Code z.B. bei einer if-Verzweigung entlangwandert. **[F11]** springt hierbei auch in Methoden rein, die vom Code aufgerufen werden. Soll dies nicht getan werden, können Sie auch statt **[F11]** die Taste **[F10]** nutzen. Hier bleibt der Debugger in der aktuellen Ebene, ohne den Code der Methode anzuzeigen.



The screenshot shows the MonoDevelop IDE with the file "TouchPathfinding.cs" open. The code is identical to the previous screenshot. The line containing the breakpoint (line 8) is now highlighted in yellow, indicating it is the next line to be executed. A small yellow circle with a red border is placed to the left of the line number 8, indicating the current execution point.

**Bild 19.3** Angehaltener Code

Wenn Sie nun fertig sind, können Sie mit **[F5]** den Code weiter ausführen lassen, bis er zum nächsten *Breakpoint* gelangt. Alle Funktionen der F-Tasten können natürlich auch über das Menü ausgeführt werden. Sie finden diese ebenfalls im Bereich **Run** des Hauptmenüs.

### 19.1.2 Variablen beobachten

Wenn das Spiel nun durch einen *Breakpoint* angehalten wurde und Sie sich im Code befinden, sollte der *View*-Modus von MonoDevelop in *Debug* gesprungen sein. Sollte dies nicht der Fall sei, können Sie diesen auch manuell über das Menü *View/Debug* wählen.

Ist dieser Modus aktiv, werden Ihnen verschiedene Zusatzfenster angezeigt, wozu auch das *Locals*-Fenster gehört. In diesem werden alle zurzeit gültigen Variablen des Skriptes mit deren Werten angezeigt. Hier können Sie verfolgen, wie sich die Werte der Variablen ändern. Zusätzlich können Sie auch den Mauszeiger einfach über die Variablen halten, sodass der jeweilige Wert in einem *Tooltip* angezeigt wird.

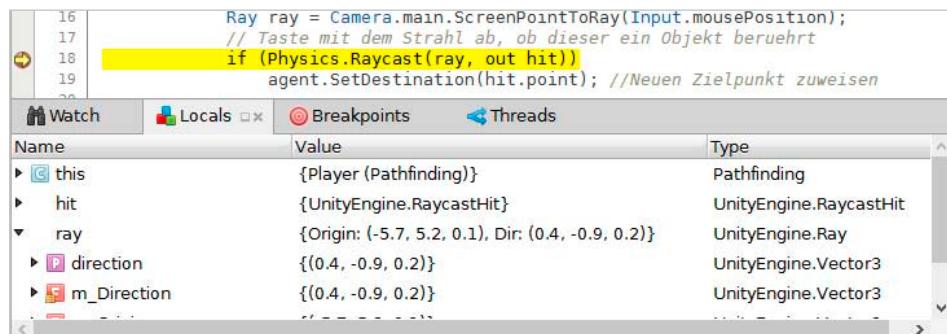


Bild 19.4 Locals-Fenster

Möchten Sie mehrere Variablen beobachten, können Sie auch das Fenster *Watch* nutzen, das ebenfalls während des *Debug*-Modus eingeblendet sein sollte. Diesem können Sie beliebige Variablen aus diesem Skript zufügen, deren Werte hier dargestellt werden. Um eine Variable hier einzufügen, müssen Sie entweder den Namen manuell eintragen oder per *Copy & Paste* einfügen.

### 19.1.3 Console Tab nutzen

Neben den Möglichkeiten innerhalb von MonoDevelop ist es häufig auch sehr sinnvoll, das *Console*-Tab (Konsole) in Unity zu nutzen. Nicht nur, dass Ihnen dort hilfreiche Hinweismeldungen bis zu Fehlermeldungen angezeigt werden (mit einem Klick auf diese gelangen Sie zum Verursacher der Meldung), dank der *Debug*-Klasse können Sie dort auch selber Informationen ausgeben.

Auf diese Weise können Sie Werte von Variablen während des Spiels ausgeben, ohne MonoDevelop im *Debug*-Modus zu betreiben oder gar das Spiel mit einem *Breakpoint* anzuhalten.

## ■ 19.2 Performance

Ein wichtiger Faktor eines fertigen Spiels ist die Performance, schließlich möchte niemand ein Game spielen, das ständig ruckelt und stockt. Auch wenn man der Meinung sein könnte, dass Prozessoren und Grafikkarten heutzutage schnell genug sind und diese Punkte deshalb vernachlässigt werden können, ist das nur die halbe Wahrheit. Erstens werden Spiele immer anspruchsvoller und dadurch ressourcenhungrier und zweitens werden viele Games nicht mehr auf PCs, sondern auf Smartphones und Tablets gespielt. Und diese sind wieder nicht so leistungsstark wie die heimischen PCs.

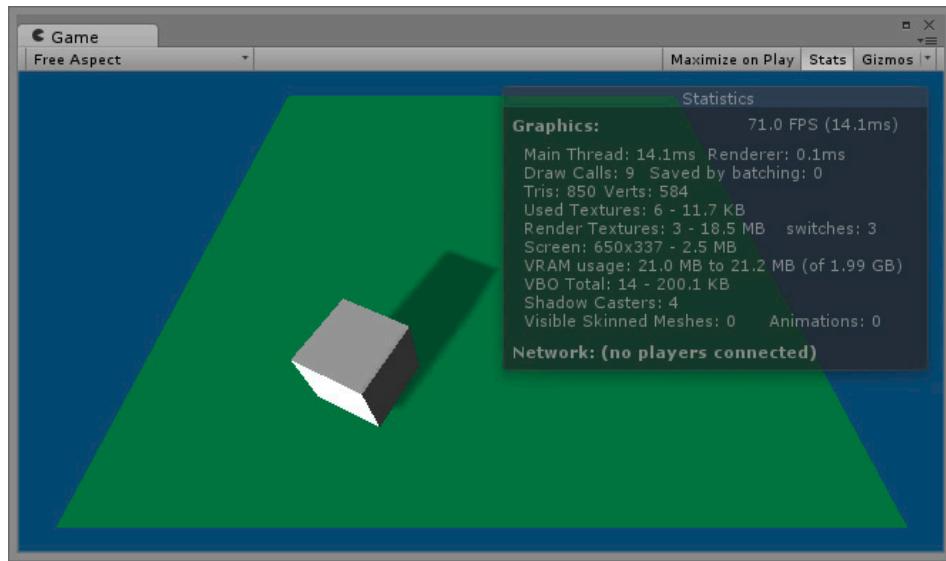
Umso größer und umfangreicher nun ein Spiel wird, desto wichtiger ist ein Beachten der verschiedenen Ressourcenfresser und der Optimierungsmöglichkeiten. Einige Punkte, die auf Performance stärkeren Einfluss nehmen, hatten wir bereits in den vorherigen Kapiteln dieses Buches angesprochen. In diesem soll es nun um die Tools gehen, die Unity Ihnen bietet, das Spiel zu analysieren und Optimierungspotenziale aufzudecken. Dabei muss allerdings auch gesagt werden, dass Spielanalysen und darauf aufbauende Optimierungen nicht unbedingt etwas für den blutigen Anfänger sind. Nichtsdestotrotz ist es aus meiner Sicht nie zu früh, sich mit dem Beachten grundsätzlicher Performancegesichtspunkte zu beschäftigen.

### 19.2.1 Rendering-Statistik

Direkt in der Game View finden Sie bereits eine Kurzanalyse Ihres Spieles. Das Fenster besitzt im oberen Bereich ein schmales Menüband, wo sich unter anderem der Button **Stats** befindet. Ein Klick darauf und es wird Ihnen oben rechts in der *Game View* die *Rendering-Statistik* eingeblendet.

In dieser finden Sie eine Zusammenfassung der wichtigsten Kennzahlen bezüglich des *Renderings* Ihres Spiels. Da das Rendern einen erheblichen Einfluss auf die Performance eines Spiels hat, können Sie bereits anhand dieser Werte Game-Optimierungen ableiten.

- **FPS** gibt die Frames pro Sekunde an. Dahinter befindet sich in Klammern der Kehrwert, der die Dauer für das Rendern eines einzelnen Frames darstellt.
- **Draw Calls** gibt die Anzahl der benötigten Zeichnungsvorgänge für die Darstellung eines Frames an.
- **Saved by batching** gibt die eingesparten Zeichnungsvorgänge durch *Batching* an (siehe Abschnitt „Draw Calls“).
- **Tris** gibt die Anzahl der dargestellten *Triangles* im Bild an (siehe Kapitel „Objekte in der dritten Dimension“).
- **Verts** gibt die Anzahl der dargestellten *Vertices* im Bild an.
- **Used Textures** gibt die Anzahl der verschiedenen Texturen an, die in dem aktuellen Bild genutzt werden.
- **Render Textures** gibt die Anzahl der Render Textures an.
- **Screen** gibt die Maße des Bildschirms an.
- **VRAM usage** gibt den aktuell gebundenen Video-RAM (VRAM) an, der für die Darstellung der Bildinformationen benutzt wird.



**Bild 19.5** Eingeblendete Rendering-Statistik

- **VBO total** gibt die Anzahl der verschiedenen *Meshes* an, die in die Grafikkarte geladen werden.
- **Visible Skinned Meshes** gibt die Anzahl der gerenderten *Skinned Meshes* an.
- **Animations** gibt die Anzahl der abgespielten Animationen an.

### 19.2.1.1 Draw Calls

Die Kennzahl *Draw Calls* gibt an, wie viele Zeichnungsdurchgänge benötigt werden, um das aktuelle Bild, also alle Objekte, die sich im Sichtfeld der Kamera befinden, darzustellen.

Dabei sind zwei Punkte wichtig: die *Meshes*, die die Formen der Objekte beschreiben, und die *Materials*, die sagen, wie die Oberflächen dieser Objekte aussehen sollen. Pro *Draw Call* wird nun genau ein Material eines *Mesh* gezeichnet. Haben Sie nun zwei *Meshes*, benötigen Sie auch zwei *Draw Calls*.

Damit dies aber nun nicht ins Unendliche wächst, hilft die Engine mit sogenannten *Batching*-Verfahren, diese *Draw Calls* zu reduzieren und damit die Performance zu steigern. Unity unterscheidet hier zwischen *Dynamic Batching* und *Static Batching*, die Sie in den *Player Settings* (*Edit/Project Settings/Player*) für jede Plattform separat einstellen können.

- **Dynamic Batching** optimiert die Zeichnungsvorgänge von Objekten. Dabei bündelt Unity verschiedene Objekte, die die gleichen *Materials* nutzen, die gleiche Skalierung haben und noch einige weitere Kriterien erfüllen, um diese dann gemeinsam in einem *Draw Call* zu zeichnen. Auch die Lichtquellen, die die Objekte beleuchten, spielen hierbei eine Rolle. Wenn Sie jetzt dieses Verfahren in den *Player Settings* aktiviert haben, wird das *Dynamic Batching* automatisch ohne weiteres Zutun ausgeführt.
- **Static Batching** dient dem Optimieren von statischen Objekten. Dieses Verfahren kann auch Objekte zusammenführen, die z.B. nicht die gleiche Skalierung besitzen. Zum Nutzen dieses Verfahrens müssen Sie natürlich dem System sagen, welche Objekte statisch

sind. Nutzen Sie hierfür den Parameter *Batching Static* des **Static**-Menüs im *Inspector*. Sie können natürlich auch das Objekt komplett als *Static* definieren, *Batching Static* würde aber eben auch reichen. Alternativ können Sie auch alle Objekte, die zusammengefasst werden sollen (und können), in ein Container-Objekt packen, dem Sie dann der Methode *StaticBatchingUtility.Combine* übergeben. Beachten Sie, dass *Static Batching* ein Unity Pro-Feature ist.

**Listing 19.1** GameObject mit Unterobjekten zum Static Batching zufügen

```
UnityEngine;
using System.Collections;
public class CombineForStaticBatching : MonoBehaviour {
    void Awake () {
        StaticBatchingUtility.Combine(gameObject);
    }
}
```

Egal welches *Batching*-Verfahren Sie nun nehmen, bei beiden ist die Voraussetzung, dass nur Objekte zu einem *Draw Call* zusammengeführt werden können, die auch das gleiche *Material* nutzen. Umso weniger *Materials* Sie also in einem Spiel einsetzen, desto mehr *Draw Calls* können Sie potenziell durch die *Batching*-Verfahren einsparen. *UV-Mapping* ist hierbei ein gutes Mittel, die Texturen und damit auch die *Materials* zu reduzieren. Mehr zu diesem erfahren Sie im Kapitel „Objekte in der dritten Dimension“.



### Während des Imports skalieren

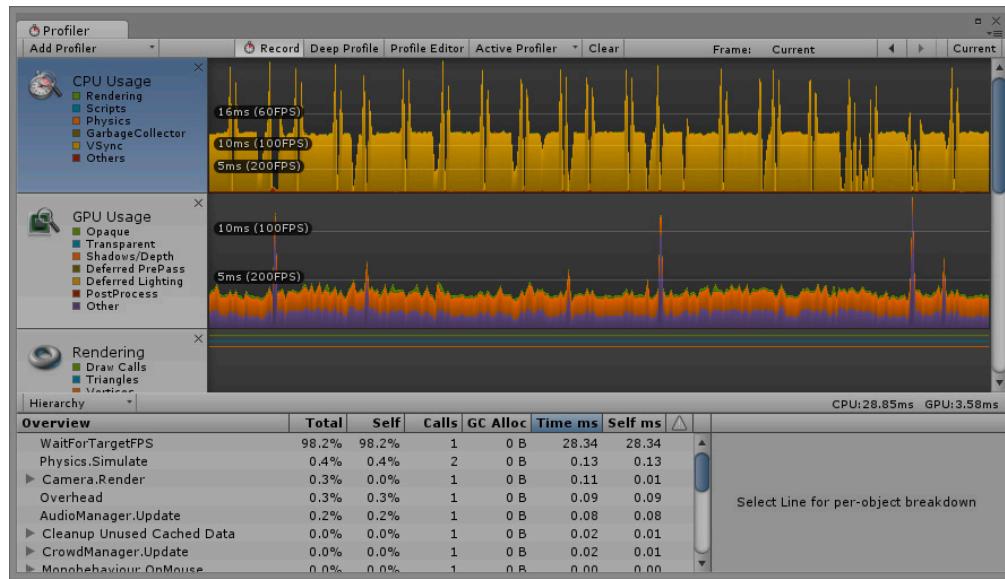
Beim *Dynamics Batching* ist es unter anderem wichtig, dass die Objekte neben dem gleichen *Material* auch die gleiche Skalierung besitzen, also z. B. die *Scale*-Werte (1,1,1) besitzen. Muss doch mal ein Objekt skaliert werden, weil das Modell ursprünglich zu klein modelliert wurde, kann das Objekt auch stattdessen in den *Import Settings* über den *Scale Factor* auf die passende Größe gebracht werden. In diesem Fall muss das Objekt nicht mehr in der Szene extra skaliert werden, sodass *Static Batching* trotzdem funktioniert.

## 19.2.2 Analyse mit dem Profiler

Bei kleineren Spielen reicht häufig schon ein einfaches Beachten der Performance-Tipps und ein Blick in die *Rendering-Statistik* aus, um eine vernünftige Performance zu erhalten. Entwickeln Sie aber ein größeres, umfangreicheres Spiel, können häufig nur tiefergehende Analysen helfen, den Performancefressern auf die Schliche zu kommen.

Für solche Analysen bietet Unity Pro ein Tool namens *Profiler* an, das Ihnen viele detaillierte Analysemöglichkeiten bietet. Das *Profiler*-Fenster finden Sie im Hauptmenü über **Window/Profiler**.

Um dieses Tool zu nutzen, starten Sie einfach Ihr Spiel und öffnen den *Profiler*. Dort werden nun nach Kategorien aufgeteilt verschiedene Performance-Kennzahlen grafisch dargestellt. Ob dies nun beispielsweise die CPU- oder GPU-Benutzung betrifft, die Rendering-, Audio-



**Bild 19.6** Profiler

oder die Physik-Berechnungen angeht, hier erhalten Sie passende Kennwerte mit den dazugehörigen Grafen.

Möchten Sie weiter in die Tiefe einsteigen, so können Sie auf einen Grafen einer Kategorie klicken. Das Spiel geht in den Pausen-Modus und Ihnen werden unten in der Overview-Liste die verschiedenen Elemente dieser Kategorie aufgeschlüsselt dargestellt. Möchten Sie beispielsweise Ihre Skripte analysieren, so klicken Sie einfach in den *CPU-Usage*-Grafen. In der *Overview*-Liste werden nun die unterschiedlichen Methoden der Skripte angezeigt und verschiedene Kennzahlen wie benötigte Rechenzeit, Aufrufe oder auch prozentualer Anteil der notwendigen Zeit. Und so besitzt jede Kategorie ihre eigenen Kennzahlen. Mit einem Klick auf die Kopfbezeichnungen dieser Werte können Sie nach diesen die Sortierung ändern.

Gerade bei der Skriptanalyse können Sie noch zusätzlich die Funktion *Deep Profile* nutzen, um zusätzliche Infos zu den Funktionsaufrufen in den Skripten zu erhalten. *Deep Profile* finden Sie in der schmalen Menüleiste oberhalb des *Profilers*. Dort finden Sie auch das Drop-down-Menü *Active Profiler*, das Sie im nächsten Abschnitt benötigen.

### 19.2.3 Echtzeit-Analyse auf Endgeräten

Häufig treten Performance-Probleme nicht beim Testen in Unity auf, sondern erst in der Endanwender-Umgebung. Gerade bei Smartphones und Tablets ist die zur Verfügung stehende Leistung doch eine gravierend andere als die vom Computer, auf der das Spiel entwickelt wird. Aber auch bei Browser-Games macht es doch einen gewaltigen Unterschied, ob Sie Dateien lokal oder aus dem Web laden.

Deshalb bietet Unity die Möglichkeit an, die Spiele auch in der späteren Laufzeitumgebung zu testen. So können Sie z.B. das ausgeführte Spiel auf einem Smartphone analysieren

oder im Browser als Webplayer-Game. Wichtig hierfür ist das Aktivieren des sogenannten *Developer Build* und die besondere *Autoconnected Profiler*-Eigenschaft beim Erstellen des Spiels (siehe „Spiele erstellen und publizieren“). Zusätzlich können Sie auch den Debugger in die Echtzeit-Analyse einbinden, wofür Sie den zusätzlichen Parameter *Script Debugging* aktivieren müssen. Hierfür müssen Sie natürlich einen *Breakpoint* setzen, damit der Code auch irgendwann anhält. Außerdem müssen Sie nach dem Starten des Spiels auf dem Endgerät MonoDevelop via *Attach To Process* an das Spiel hängen (**Run/Attach To Process**). In diesem Fall sollte neben dem Unity-Editor eben auch das zu debuggende Spiel in der Auswahl angezeigt werden.

Da bei der Konfiguration dieser Echtzeit-Analyse sowohl das Betriebssystem der Entwicklungsumgebung als auch die Zielplattform berücksichtigt werden müssen, variieren hier die Vorgehensweisen schon stark. Zwei Varianten möchte ich Ihnen im Folgenden zeigen. Alle weiteren Anleitungen finden Sie hierzu im Online-Manual von Unity, das Sie über das Hauptmenü **Help/Unity Manual** erreichen. Dort können Sie dann im Suchfenster nach „*Profiler*“ suchen.

### 19.2.3.1 Webplayer-Game remote analysieren

Um ein Webplayer-Game im eigenen Browser zu testen, aktivieren Sie vor dem Build-Prozess die vorher genannten Parameter. Dann erstellen Sie das Webplayer-Game mit **Run and Build**. Wird jetzt das Game in Ihrem Browser ausgeführt, aktivieren Sie den *Release Channel* „Development“ vom Webplayer. Halten Sie hierfür die **Alt**-Taste gedrückt und rufen Sie über die rechte Maustaste das Kontextmenü des Web-Players auf. Hier gehen Sie auf **Release Channel/Development**.

Danach starten Sie in Unity den *Profiler* und wählen dort noch im *Profiler* über das Drop-down-Menü *Active Profiler* Ihre Geräte-IP aus. Diese erfahren Sie bei Windows über die *Eingabeaufforderung*. Sie finden Sie dort über die Suche nach dem Anwendungsnamen „cmd“. Haben Sie diese gestartet, geben Sie nun den Befehl „ipconfig“ ein. Hinter dem Eintrag „IPv4-Adresse“ finden Sie Ihre IP, die Sie nun im *Active Profiler*-Menü hinterlegen müssen und anschließend auch auswählen sollten. Wenn Sie nun wieder in den Browser wechseln und das Spiel spielen, protokolliert der *Profiler* die Anwendung.

### 19.2.3.2 Android-Game remote analysieren

Bei Android-Geräten können Sie gleich über zwei Wege testen, per WiFi und ADB (Android Debug Bridge). Um per WiFi zu testen, müssen Sie hierfür zunächst einmal das mobile Internet im Smartphone deaktivieren und das Gerät per WLAN mit dem eigenen Netzwerk verbinden, mit dem auch der Entwickler-PC verbunden ist. Achten Sie hierbei darauf, dass beide die gleiche Subnetzmaske nutzen.

Jetzt verbinden Sie das Gerät per USB-Kabel mit dem PC, um das Spiel wie beim Webplayer-Game zu erstellen und dieses auf das Gerät zu übertragen. Sobald das Spiel dort läuft, können Sie in Unity den *Profiler* starten und das Spiel dort analysieren. Eventuell müssen Sie auch hier noch im *Profiler* über das Drop-down-Menü *Active Profiler* das Gerät auswählen.

# 20

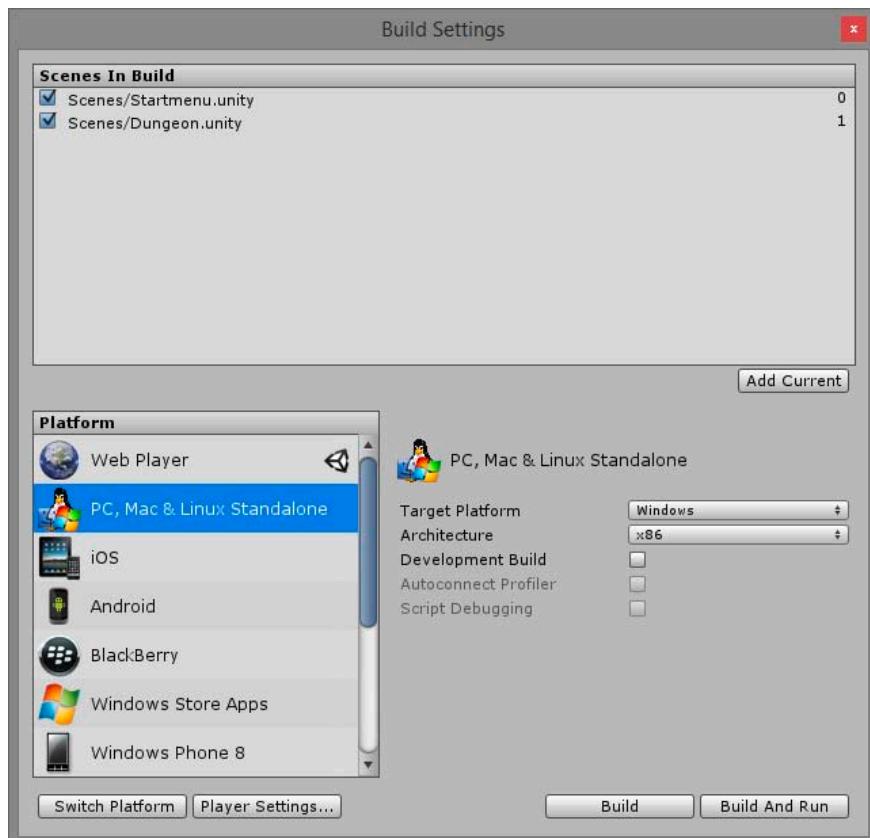
# Spiele erstellen und publizieren

Bisher haben Sie Ihr Spiel nur innerhalb von Unity ausgeführt. Möchten Sie das Spiel aber nun außerhalb spielen, vielleicht sogar der Öffentlichkeit zugänglich machen, müssen Sie das Spiel erst einmal erstellen. In Unity nennt sich dieser Vorgang *Build*, also „Bauen“. Damit ist aber noch nicht das Ende der Spieelerstellung erreicht. Nun muss das Spiel auch noch passend bereitgestellt werden, was, je nachdem, ob Sie das Game als App, Browser-Game oder Desktop-Anwendung erstellen, sehr unterschiedlich sein kann.

## ■ 20.1 Der Build-Prozess

Alle Funktionen, die zum Erstellen eines Spiels gehören, finden Sie unter dem Menüpunkt **File**. In den *Build Settings*, die Sie über **File/Build Settings** erreichen, können Sie alle Einstellungen für den *Build*-Vorgang vornehmen und diesen schließlich auch starten. Dabei können Sie das Spiel entweder über den Button **Build** nur erstellen oder auch über **Build and Run** direkt im Anschluss gleich starten lassen.

Haben Sie dort die Einstellungen einmal vorgenommen, können Sie das „Erstellen und Starten“ auch direkt über **File/Build and Run** anstoßen. Beachten Sie hierbei, dass Sie natürlich bei Mobile-Games das jeweilige Testgerät per USB-Kabel an Ihren Entwickler-Computer angeschlossen haben müssen.



**Bild 20.1** Build Settings

### 20.1.1 Szenen des Spiels

Während der Spieleentwicklung ist es häufig praktisch, kleine Testszenen zu erstellen, in denen Sie bestimmte Funktionalitäten oder Effekte erst einmal testen können. Damit diese aber nicht am Ende auch in das Spiel gelangen und die Spieldatei unnötig aufzublähen, können Sie in den *Build Settings* definieren, welche Szenen in das fertige Spiel übernommen und welche ausgelassen werden sollen. Hierfür gibt es im oberen Teil der *Build Settings* die Liste *Scenes in Build*. Über den Button **Add Current** können Sie die aktuelle Szene zu dieser Liste zufügen. Entfernen Sie den Haken vor der Szene, so wird diese nicht mehr in das Spiel eingebunden.

Außerdem sehen Sie auf der rechten Seite eine laufende Nummer. Dies ist der *Level Index*, über den Sie die Szene direkt ansprechen können. Im Kapitel „Skript-Programmierung“ haben Sie bereits die Methode `LoadLevel` der `Application`-Klasse kennengelernt. Diese startet eine neue Szene und erwartet hierfür entweder den Namen der zu startenden Szene oder dessen *Level Index*.

**Listing 20.1** Zwei Möglichkeiten, eine Szene zu starten

```
//Variante 1: Start per Name  
Application.LoadLevel ("StartMenu");  
//Variante 2: Start per Level Index  
Application.LoadLevel (0);
```

Per Drag & Drop können Sie die Szenen nach Belieben sortieren. Die Indices werden dabei automatisch angepasst. Wichtig ist hierbei, dass beim fertigen Spiel immer die Szene als Erstes gestartet wird, die den Index 0 besitzt. Alle anderen Szenen starten Sie dann selber über Befehle wie die `LoadLevel`-Methode.

## 20.1.2 Plattformen

In den *Build Settings* wird neben den einzubindenden Szenen auch definiert, für welche Plattform das Spiel erstellt werden soll. Dabei werden zwar alle Plattformen angezeigt, aber nur bei denjenigen, für die Sie auch eine Lizenz besitzen, werden weitere Optionen angeboten. Nur dort können Sie auch tatsächlich **Build** ausführen.

Wie Sie sehen, hat die Entwicklung des Spiels erst einmal nichts mit der jeweiligen Zielplattform zu tun, für die Sie das Spiel „builden“ wollen. Sie können also ein Unity-Projekt zunächst für eine Plattform „bauen“ und danach das Spiel für eine andere Umgebung erstellen. Zumindest ist das die Theorie. In der Praxis ist es natürlich etwas anders. So sollte klar sein, dass ein normaler PC keinen Beschleunigungssensor besitzt und ein Smartphone keine Maus (jedenfalls nicht normalerweise). Solche plattformspezifischen Dinge müssen dann natürlich doch wieder angepasst werden. Aber insgesamt funktioniert das schon wirklich gut.

Allerdings muss man hier den eigentlichen Build-Prozess noch etwas differenzierter betrachten. Gerade wenn Sie für Mobile-Plattformen Spiele „builden“ wollen, benötigen Sie noch einige weitere Komponenten neben Unity selbst.

## 20.1.3 Notwendige SDKs

Je nach Zielplattform kann es sein, dass Sie für den *Build* Ihrer Unity-Anwendung zusätzliche Programme bzw. SDKs (Software Development Kits) installieren müssen. Eigentlich baut Unity nur bei den *Stand-alone-Builds* und beim Web-Player das Spiel komplett alleine. Für alle anderen Plattformen nutzt Unity zusätzlich noch die plattformeigenen Entwicklerwerkzeuge.

Das bedeutet, dass Unity in diesen Fällen sein eigenes Projekt lediglich in das für die jeweilige Zielplattform typische Entwicklerformat übersetzt. Das tatsächliche Erstellen der App überlässt Unity dann den jeweiligen plattformzugehörigen Entwicklerwerkzeugen. Im Falle von Android heißt das, dass Sie für das Erstellen einer APK (das Datenformat, in dem sich Android-Apps befinden) neben Unity auch das Android SDK mit den Plattform-Tools und den USB-Treibern installieren müssen. Für Windows Store Apps und Windows Phone müssen Sie wiederum Visual Studio installieren und für iOS benötigen Sie XCode.

Was im Detail benötigt wird, hängt von der jeweiligen Zielplattform und auch der eigenen Entwicklerumgebung ab. Unity stellt hierfür zu jeder Zielplattform ein eigenes „Getting started“ bereit, das Sie im Unity-eigenen Manual finden. Um diese Hilfe-Dokumente aufzurufen, starten Sie über **Help/Unity Manual** das Manual und geben dann in der Suchmaske „Getting started“ ein.

### 20.1.4 Plattformspezifische Optionen

Jede Plattform besitzt individuelle *Build*-Optionen, über die noch einmal Spezialeinstellungen vorgenommen werden können.

Beim *Stand-alone-Build* können Sie beispielsweise das Ziel-Betriebssystem wählen, was aktuell Windows, Mac OS X oder Linux sein kann. Bei Android sind spezielle Texturkompressionen möglich und bei Windows Store Apps können Sie definieren, in welcher Zielsprache das gebaute Visual Studio-Projekt erstellt werden soll.

### 20.1.5 Developer Builds

Bei jeder Plattform gibt es neben den obigen *Build*-Optionen auch den sogenannten *Developer Builds*-Parameter. An diesem sind je nach Zielplattform auch noch weitere Funktionalitäten gekoppelt. Der Parameter *Developer Builds* alleine setzt zunächst einmal nur die Eigenschaft `isDebugBuild` der `Debug`-Klasse auf TRUE. Hierüber können Sie im Programmcode steuern, wenn Code beispielsweise nur während der Entwicklung ausgeführt werden soll. `isDebugBuild` gibt hierbei nur dann ein TRUE zurück, wenn der obige Parameter aktiviert wurde oder aber das Spiel direkt in Unity ausgeführt wird.

**Listing 20.2** Nutzen der `isDebugBuild`-Variablen

```
void Start() {
    if (Debug.isDebugBuild) {
        //...
    }
}
```

#### 20.1.5.1 Autoconnect Profiler

Bei vielen Plattformen wird beim Aktivieren von *Developer Builds* der Parameter *Autoconnect Profiler* verfügbar.

Haben Sie diesen Parameter während des *Build*-Prozesses aktiviert, können Sie später das ausgeführte Spiel über den in Unity integrierten *Profiler* analysieren. Mehr zu diesem Thema finden Sie im Kapitel „Fehlersuche und Performance“.

#### 20.1.5.2 Script Debugging

Je nach Plattform kann beim Aktivieren von *Developer Builds* auch der Parameter *Script Debugging* verfügbar werden.

Durch Aktivierung dieses Parameters können Sie über MonoDevelop den ausgeführten Code im fertigen Programm debuggen, so als würde das Programm in Unity ausgeführt werden. Hierfür müssen Sie zunächst das Spiel auf dem Endgerät starten und MonoDevelop via *Attach To Process* an das Spiel hängen. Beachten Sie hierbei, dass in diesem Fall das zu debuggende Spiel sowie der Unity-Editor in der Auswahl angezeigt werden. Mehr zu diesem Thema finden Sie im Kapitel „Fehlersuche und Performance“.

## ■ 20.2 Publizieren

In Zeiten von Apps und Online-Stores ist das Veröffentlichen von Spielen rein technisch gesehen einfacher als je zuvor. Ich spreche deshalb von „technisch“, weil natürlich das gesamte Marketing, das noch dazukommt, nicht unbedingt dadurch vereinfacht wird, wenn das eigene Produkt mit Tausenden anderer Apps in dem gleichen Store um die Gunst des Käufers buhlt.

Im Folgenden möchte ich mich aber weniger um das Marketing kümmern, sondern mehr auf die technischen Anforderungen eingehen und beleuchten, wo die Besonderheiten und Unterschiede zwischen einer App-Veröffentlichung, einem Web-Game und einem herkömmlichen Desktop-Spiel liegen, wenn Sie sich selber um die Veröffentlichung kümmern wollen.



### Ein paar kleine Marketing-Tipps

Auch wenn es kein Spieleentwickler gerne hört, aber egal wie gut oder schlecht Ihr Spiel am Ende ist, wenn keiner von diesem weiß, wird es auch niemand kaufen bzw. herunterladen.

Marketing muss aber nicht immer teuer sein. In Zeiten von Online-Stores und Suchmaschinenoptimierung (SEO) kann schon bereits die Wahl des richtigen Namens einen enormen Einfluss auf den Erfolg des Spiels nehmen. Genauso sollten Sie bei Apps immer darauf achten, dass Sie aussagekräftige Beschreibungstexte mit den richtigen Schlagwörtern nutzen und diese mit hochwertigen Bildern und optimaler Weise auch mit Video-Trailern ergänzen.

Weiter sollten Sie auf jeden Fall Ihre sozialen Kanäle nutzen, um Ihre Spiele bekannter zu machen – natürlich nur dann, wenn es thematisch passt. Nur achten Sie darauf, dass Sie nicht anfangen, Ihre Bekannten und Freunde „zuzuspammeln“.

### 20.2.1 App

Die Veröffentlichung einer App gehört zu den unkompliziertesten Arten eines Spiel-Release. Ob Android, iOS oder Windows Phone – jede App-Technologie hat mindestens einen Store, in dem Sie Ihre Apps recht unkompliziert anbieten können.

Für diese Stores benötigen Sie normalerweise lediglich einen Account und eine Entwicklerlizenz, die einmalig oder regelmäßig (meist jährlich) bezahlt werden muss. Sie brauchen dort nur die App-Datei und einige Infomaterialien wie Bilder und Beschreibungstexte im Store hochzuladen, den Preis festzulegen, und schon können Ihre Spiele dort angeboten werden.

Vor der eigentlichen Veröffentlichung durchlaufen diese noch vorher interne Kontrollen, wo überprüft wird, ob die App den Qualitätsanforderungen und Regularien des Online-Geschäfts entsprechen. Dies kann je nach Store von wenigen Minuten bis zu mehreren Tagen dauern. Wenn nun alles okay ist, wird Ihre App dort angeboten und sie kann heruntergeladen werden.

Android spielt hierbei eine Sonderrolle. Nicht nur, dass es mehrere Stores für Android-Apps gibt, Sie können Ihre Apps auch selber anbieten. So können Sie diese z.B. auf Ihrer Website zum Download bereitstellen oder sie per E-Mail verschicken. Der Kunde muss lediglich in seinem Android-Betriebssystem das Installieren aus Nicht-Store-Quellen zulassen.

## 20.2.2 Browser-Game

Eine ebenfalls sehr einfache Art, das Game zu veröffentlichen, kann das Browser-Game sein. Auch hier können Sie das Game selber auf Ihrem Webspace hosten und der Öffentlichkeit zugänglich machen.

Wenn Sie ein Webplayer-Game *builden*, erstellt Unity eine unity3d-Datei sowie eine HTML-Datei, in der das Game eingebettet ist. Möchten Sie nun Ihr Game selber hosten, kopieren Sie einfach beide Dateien in ein gemeinsames Web-Verzeichnis auf Ihren Webspace, und schon kann Ihr Spiel weltweit gespielt werden.

Es gibt aber auch Portale, die sich auf Unity-Games spezialisiert oder zumindest einen eigenen Bereich dafür eingerichtet haben. Als Beispiele kann man hier die Portale Kongreta, Shockwave oder Musegames nennen. Aber auch Facebook bietet Unity-Entwicklern die Möglichkeit, Games zu hosten.

Wie bei den Stores benötigen Sie auch bei diesen Portalen natürlich wieder Accounts. Inwiefern hier allerdings Gebühren fällig sind, ist abhängig von den Portalen. Genauso variieren hier die Genehmigungen der einzelnen Games. Nicht selten wird aber gefordert, dass in das Spiel eigene Portal-Plug-ins eingebunden werden, um zum Beispiel eigene Funktionen zu integrieren oder ein besseres Tracking zu ermöglichen.

## 20.2.3 Desktop-Anwendung

Auch Desktop-Anwendungen können Sie wie Apps über Portale anbieten. Der wohl bekannteste Anbieter ist hier Steam. Es gibt aber auch andere Anbieter wie Gamersgate, GOG, BigFish, Greenman Gaming oder auch Desura. Und auch Amazon mischt in diesem Bereich mit seinem „Indie Games Store“ mit, auch wenn dieser aktuell nur in den USA verfügbar ist.

Bei den großen App-Stores ist das Anbieten eigener Desktop-Spiele im Gegensatz zu den großen Anbietern wie Steam etwas schwieriger. So gibt es bei Steam extra ein Verfahren,

bei dem die Steam-Community ein Spiel bewertet. Von dieser Bewertung hängt dann schließlich die Veröffentlichung ab.

Eine Veröffentlichung in Eigenregie ist ebenfalls etwas schwieriger und im Gegensatz zu den anderen Game-Typen mit mehr Aufwand verbunden. Wenn Unity bei *Stand-alone* ein Desktop-Game erstellt, wird beim *Build*-Vorgang von Unity lediglich eine ausführbare Programmdatei, bei Windows ist das die altbekannte Executable (.exe), sowie ein Dateiordner mit zugehörigen Daten erzeugt. Auch wenn Sie diese Datei nun starten und spielen können, um die eigentliche Installationsfähigkeit dieses Spiels müssen Sie sich jetzt noch selber kümmern. Dies wird bei den Portalen durch deren Technik, wie den Steam-Client, übernommen.

Natürlich können Sie zum Testen auch einfach die Programmdatei und den Ordner auf einen anderen Rechner kopieren, um sie dort zu testen. Auch reicht dies möglicherweise für das kostenlose Bereitstellen des Games im Web als Zip-Datei. Für ein professionelles, verkaufsfähiges Spiel reicht dies aber nicht aus.

Bei professionellen Spielen übernimmt eigentlich immer ein Installationsprogramm die Installation, das sowohl das Programm wie auch die zugehörigen Dateien zielgerichtet in ein Installationsverzeichnis kopiert. Meistens gehören hierzu auch Hilfedokumente, die dem Spiel beiliegen. Auch werden von so einem Installer normalerweise noch Links auf dem Desktop und dem Startmenü gesetzt, Icons in den Shortcuts hinterlegt und Registry-Einträge vorgenommen, um den aktuellen Patch-Stand, den Installationspfad o. Ä. abzulegen. Außerdem werden gewöhnlich noch die Systemanforderungen abgeglichen und die AGBs eingebunden, die beim Installieren bestätigt werden müssen.

Solch einen ausgefeilten Installer mit einer Software zu konfigurieren und zu testen, kann schon mal mehrere Stunden dauern. Gerade das Testen sollte auch auf verschiedenen Systemen/Geräten durchgeführt werden, da es kaum was Unangenehmeres gibt, als wenn bereits bei der Installation des Spiels schon Fehler auftreten oder dies sogar komplett nicht geht.

Einige Beispiele für Softwareprodukte, mit denen Sie solche Installationsprogramme erstellen können, sind InstallBuilder, InstallShield oder NSIS.

Haben Sie nun einen Installer für Ihr Spiel erstellt, können Sie das Spiel auch selber auf der eigenen Website und/oder über die herkömmlichen Verkaufsplattformen anbieten. Allerdings sollten Sie auch hier immer die jeweiligen AGBs der Plattformen hinsichtlich dem Verkauf eigener Produkte, speziell von Software, beachten.

Lizenziert für gregwyss@hotmail.com.

© 2014 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

# 21

## Beispiel-Game

Nachdem Sie nun die Grundlagen von Unity kennengelernt haben, wollen wir dieses Wissen im Rahmen eines Spiels in der Praxis anwenden. Hierfür finden Sie auf der beiliegenden DVD ein Beispiel-Game, das in drei verschiedenen Fassungen vorliegen sollte:

- **Empty BeispielGame:** Dieses Projekt beinhaltet lediglich die verschiedenen Assets wie die 3D-Modelle, Texturen, Soundfiles usw.
- **Template BeispielGame:** Dieses besitzt neben den Assets auch eine Szene mit einem vordefinierten Level-Design, auf das die späteren Beschreibungen in diesem Kapitel zugeschnitten sind.
- **Beispiel-Game:** Dieses dient als Anschauungsobjekt und zeigt das Beispiel-Game, wie es am Ende aussehen soll.

Im Rahmen dieses Beispiel-Games wollen wir ein kleines, einem Rollenspiel ähnliches 3D-Game aus dem *Dungeon Crawler*-Genre entwickeln. Bei diesem Genre geht es meist darum, in einem Gemäuer (Dungeon) Rätsel zu lösen und feindlich gesonnene NPCs zu bekämpfen. Da hier sehr viele unterschiedliche Techniken eingesetzt werden können, eignet sich dies sehr gut als Anschauungsbeispiel.

Das Entwickeln eines kompletten *Dungeon Crawlers* würde natürlich etwas den Rahmen sprengen, weshalb wir uns hier nur mit einer einzelnen Quest beschäftigen werden, also mit einer Aufgabe, die der Spieler lösen muss. In diesem Fall soll es darum gehen, einen Auffangbehälter zu finden, um heruntertropfendes Trinkwasser aufzufangen.

Bei der Suche nach diesem Behälter stößt der Spieler sowohl auf feindliche Fledermäuse, die er bekämpfen, sowie auf verschlossene Tore, die er öffnen muss.

Beachten Sie hierbei, dass wir die GUI zunächst mit *GUIElements* (*GUITexture*, *GUIText*) und der *OnGUI*-Programmierung umsetzen werden. Am Ende folgt ein Zusatzabschnitt, in dem ich noch einmal darauf eingehende, was geändert werden muss, um die neuen *uGUI*-Controls zu nutzen, die ab der Version 4.6 zur Verfügung stehen.

Da mir während des Schreibens des Buches nur Beta-Fassungen der neuen *uGUI*-Objekte vorlagen und ich mit Ihnen hier ein funktionierendes Beispiel entwickeln möchte, wird das Beispiel zunächst auf den herkömmlichen GUI-Techniken aufsetzen.

Sobald das neue *uGUI*-System offiziell erhältlich ist, werde ich Ihnen auf meinem Blog eine zweite Variante zum Herunterladen anbieten, die dann die neuen *uGUI*-Controls nutzt. Mehr Infos erfahren Sie hierzu in der „Einleitung“ dieses Buches.

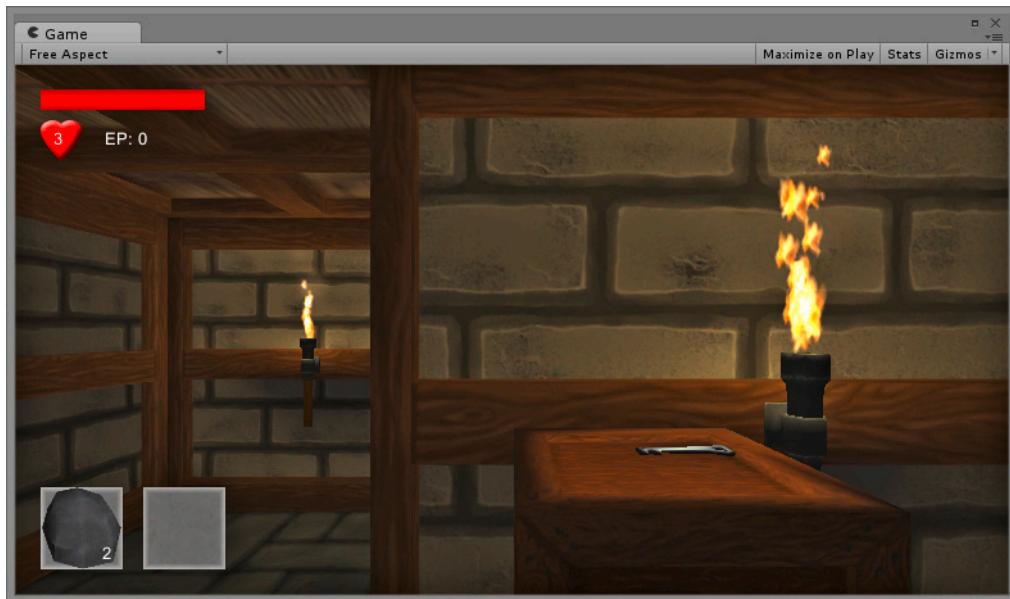


Bild 21.1 Szene aus dem fertigen Beispiel-Game



### Sicherheitskopien und Versionsverwaltung

Wenn Sie ein Spiel entwickeln, sollten Sie natürlich in regelmäßigen Abständen Ihr Projekt speichern. Zudem ist es empfehlenswert, Zwischenstände Ihres Projektes noch einmal zusätzlich zu kopieren und als Backup in einem anderen Verzeichnis (oder sogar extern) abzulegen. Es muss ja nicht immer gleich zum Plattencrash kommen, es reicht ja schon, dass z. B. eine Änderung nicht das bewirkt, was Sie sich vorgestellt haben. Dann ist es vorteilhaft, wenn Sie auf einfache Weise zu einem vorherigen Stand zurückwechseln können. Für diesen Zweck gibt es auch extra Software, sogenannte Versionsverwaltungssoftware (Version Control System), die diese Aufgabe übernimmt. Durch das Kopieren Ihres Projektes können Sie das aber auch auf einfache Weise manuell vornehmen.

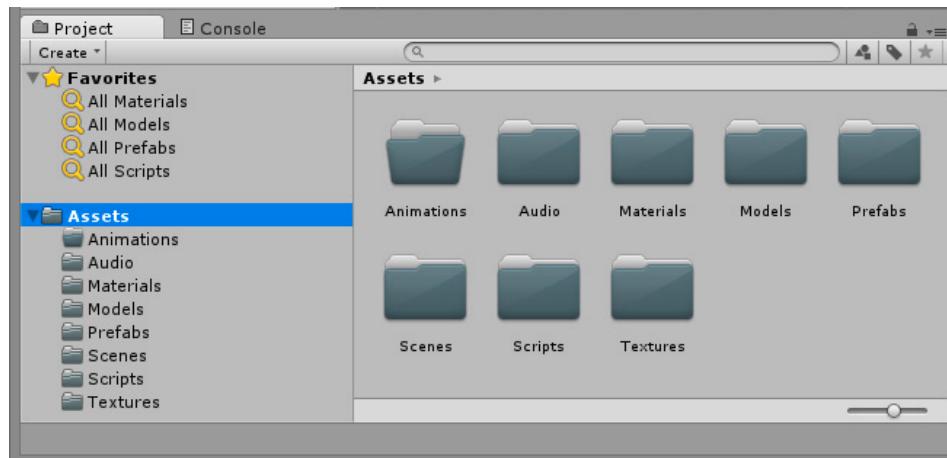
## ■ 21.1 Level-Design

Auch wenn sich die anderen Abschnitte auf das Vorlagenprojekt beziehen, möchte ich Ihnen im Folgenden kurz erläutern, wie Sie mithilfe der verschiedenen 3D-Modelle auch selber ein Level gestalten können.

## 21.1.1 Modellimport

Sollten Sie selbst designete 3D-Modelle nutzen wollen, müssen Sie natürlich zunächst die FBX-Dateien, Texturen usw. in Ihr Projekt importieren. Dies geht ganz einfach per Drag & Drop in den *Project Browser* des Projektes.

Am besten nutzen Sie hierfür gleich eigene Ordner, um die verschiedenen Asset-Typen im Project Browser zu sortieren und den Überblick zu behalten, z. B. „Materials“, „Models“, „Scenes“ usw. In dem Beispiel-Game werden Sie dies ebenfalls vorfinden.



**Bild 21.2** Ordnerstruktur für unterschiedliche Asset-Typen

### 21.1.1.1 Allgemeine Import Settings

Bei den mitgelieferten Modellen und Texturen sollten Sie ganz grundsätzlich erst einmal überall folgende *Import Settings* nutzen:

- **Model/Scale Factor:** 1
- **Model/Generate Colliders:** deaktiviert
- **Animations/Import Animation:** deaktiviert

Da einige Modelle aber spezielle Einstellungen benötigen, werden wir diese jetzt noch nachträglich einstellen.

### 21.1.1.2 bat\_03 Import Settings

Das Fledermaus-Modell „bat\_03“ besitzt eigene Animationen. Deshalb muss in den *Import Settings* dieses Modells zusätzlich folgende Einstellung vorgenommen werden:

- **Animations/Import Animations:** aktiv

### 21.1.1.3 crate\_01 Import Settings

Das Modell der Holzkiste „crate\_01“ ist vom Maßstab größer. Damit wir bei diesem Modell auch vom *Dynamic Batching* (siehe Kapitel „Fehlersuche und Performance“) profitieren kön-

nen, dürfen wir das Modell nicht nachträglich in der Szene skalieren. Deshalb ändern wir bereits die Skalierung beim Import. Ansonsten bleiben die restlichen Einstellungen wie bei den „Allgemeinen Import Settings“.

- **Model/Scale Factor:** 0.25

#### 21.1.1.4 stone\_01 Import Settings

Auch das 3D-Modell des Steins wollen wir bereits beim Import der Größe anpassen. Zudem wollen wir die Kanten des Modells etwas weicher gestalten, weshalb wir die Normalen innerhalb von Unity nachkalkulieren und über *Smoothing Angle* nachsteuern. Der Rest wird so wie im Abschnitt „Allgemeine Import Settings“ erläutert eingestellt.

- **Model/Scale Factor:** 0.1
- **Model/Normals Calculate:** aktiv
- **Model/Smoothing Angle:** 100

#### 21.1.1.5 floor\_01 Import Settings

Damit unser Spieler auf dem Boden gehen kann, wollen wir dem Bodenmodell „floor\_01“ automatisch einen *Collider* zufügen. Natürlich gibt es auch andere Möglichkeiten, aber in diesem Fall wollen wir mit den automatisch erzeugten *Collidern* arbeiten.

- **ModelGenerate Colliders:** aktiv

### 21.1.2 Materials zuweisen

Beim Import der FBX-Dateien werden gewöhnlich die dazugehörigen *Materials* angelegt. Importieren Sie zusätzlich noch Texturen und weisen Sie diesen *Materials* zu. Außerdem müssen Sie noch die *Shader* auswählen.

- *Materials*, die neben den Texturen auch noch *Normalmaps* besitzen, können den *Shader* „Bumped Diffuse“ nutzen, dem Sie dann dementsprechend beide Grafiken zuweisen.
- Bei *Materials*, wo Sie nur die eigentliche Textur haben, wählen Sie dann den *Shader* „Diffuse“.

### 21.1.3 Prefabs erstellen

Anstatt nun einfach die Modelle in die Szene zu ziehen und das Dungeon zu kreieren, wollen wir vorher aus den Modellen eigene *Prefabs* erstellen, die noch zusätzliche Komponenten besitzen. Erst aus diesen *Prefabs* werden wir dann das Dungeon erstellen.

Würden wir nicht so vorgehen, müssten wir jede Modellinstanz, die wir in die Szene hineinziehen, separat mit diesen Komponenten ausstatten, was natürlich einen immensen Mehraufwand bedeuten würde.

### 21.1.3.1 Wall Prefab

Für unsere Wände erstellen wir ein *Prefab* namens „Wall“:

1. Ziehen Sie das Modell „wall\_01“ in die Szene.
2. Fügen Sie dem Modell einen Box Collider zu.
3. Ziehen Sie das Modell in den „Prefabs“-Ordner und benennen Sie es in „Wall“ um.

### 21.1.3.2 Torch Prefab

Als Nächstes wollen wir ein *Prefab* für die Wandfackeln erstellen.

4. Ziehen Sie die Objekte „torch\_01“ und „torch\_holder\_01“ in die Szene
5. Weisen Sie die Fackel „torch\_01“ dem Fackelhalter „torch\_holder\_01“ als Kind-Objekt zu und positionieren Sie die Fackel auf (0,0,0,13).
6. Ziehen Sie nun das „torch\_holder\_01“-Objekt in den „Prefabs“-Ordner und nennen Sie es um in „Torch“.



**Bild 21.3**  
Fackel-Prefab

### 21.1.3.3 Wall\_Torch Prefab

Jetzt wollen wir ein zweites Wand-*Prefab* erzeugen, das allerdings noch zusätzlich die Wandfackel besitzt.

1. Kopieren Sie das *Prefab* „Wall“ und nennen Sie es in „Wall\_Torch“ um.
2. Ziehen Sie nun „Wall\_Torch“ und „Torch“ in die Szene und fügen Sie „Torch“ als Kind-Objekt von „Wall\_Torch“ zu.
3. Rotieren Sie Torch auf (-90,90,0).

Als Nächstes wollen wir nun mithilfe des Vertex-Snappings den Fackelhalter auf der Holzoberfläche positionieren:

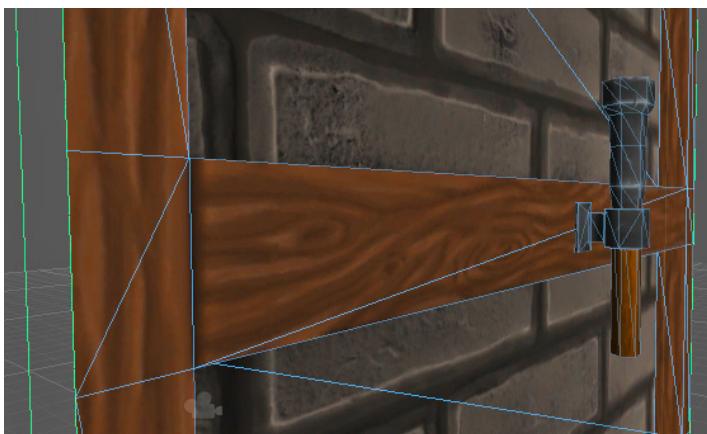
4. Markieren Sie hierfür „Wall\_Torch“ und drücken Sie die Taste **V**. Sie können durch Bewegen der Maus einen Vertex des Objektes auswählen.

5. Haben Sie nun einen Vertex am äußeren Rand des Halters, drücken Sie die linke Maustaste und schieben Sie die Fackel zur Wand. Die Fackel wird zur Wand springen und auf einem Vertex von „Wall“ platziert. Vermutlich wird dies irgendwo am Rand der Wand sein. Wichtig ist hier nun, dass dies ein Vertex am äußeren Rand ist, sodass sich die Fackel nicht in der Wand, sondern außerhalb befindet. Mehr wollen wir hiermit nicht erreichen und Sie können die Maustaste und  wieder loslassen.

Wenn Sie nun den Fackelhalter an der äußeren Fläche positioniert haben, brauchen Sie nur noch manuell „Torch“ auf dem mittleren Holzbalken zu positionieren.

Verändern Sie deshalb von „Torch“ manuell die *X-Position* auf „1“ und die *Y-Position* auf „-1“.

6. Anschließend bestätigen Sie die Änderungen mit **Apply**, damit das „Wall\_Torch“-*Prefab* alle Änderungen übernimmt.



**Bild 21.4**  
Fackel auf dem Mittelholz positioniert

#### 21.1.3.4 Floor Prefab

Für den Bodenbelag wollen wir ebenfalls ein *Prefab* erstellen. Auch wenn sich dieses im ersten Schritt nicht von dem normalen *Model-Prefab* unterscheidet, machen wir dieses, um zukünftig flexibler zu sein. Sollten Sie später das zugefügte Modell noch ergänzen wollen, können Sie diese Anpassungen nur an die anderen Instanzen übertragen, wenn Sie ein *Prefab* nutzen, nicht aber, wenn Sie das *Model-Prefab* an sich in die Szene ziehen.

7. Ziehen Sie „floor\_01“ in die Szene.
8. Ziehen Sie dieses ohne Anpassungen direkt in den „Prefabs“-Ordner und benennen dieses in „Floor“ um.

#### 21.1.3.5 Ceiling Prefab

Als Letztes erstellen wir noch ein *Prefab* für das Deckenmodell. Dies machen wir aus dem gleichen Flexibilitätsgrund wie beim Bodenbelag.

9. Ziehen Sie „ceiling\_01“ in die Szene.
10. Ziehen Sie dieses ebenfalls ohne Anpassungen direkt in den „Prefabs“-Ordner und benennen dieses in „Ceiling“ um.

## 21.1.4 Dungeon erstellen

Mithilfe der verschiedenen *Prefabs* können wir nun das Dungeon entwickeln, in dem das Spiel stattfinden soll. Erstellen Sie hierfür zunächst eine neue Szene im „Scenes“-Ordner, z. B. mit dem Namen „Dungeon“.

Damit wir später bei den vielen *GameObjects* in der Hierarchy nicht die Übersicht verlieren, erzeugen wir in der Szene als Erstes drei leere *GameObjects* über **GameObject/Create Empty**, die Sie in „Floors“, „Walls“ und „Ceilings“ umbenennen. Diese dienen später als Container für die jeweiligen *Prefab*-Instanzen, die Sie nach Wunsch einfach ein- und ausklappen können.

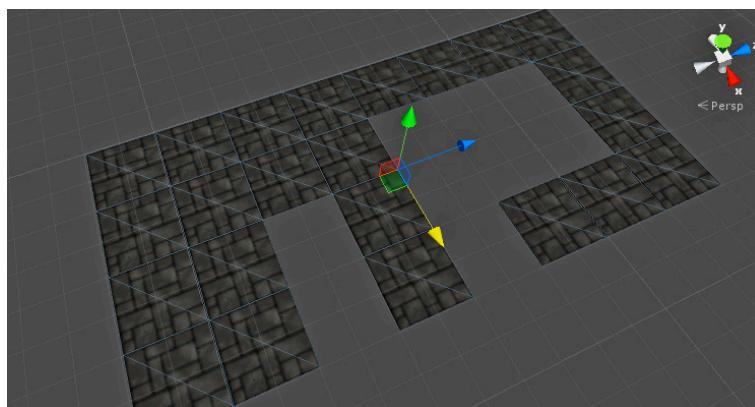
Alle drei Container-Objekte setzen Sie mithilfe der **Reset**-Funktion im *Inspector* auf die Default-Werte zurück.

### 21.1.4.1 Boden erstellen

Zum Erstellen der Bodenfläche ziehen Sie als Erstes das *Floor-Prefab* in den „Floors“-Container und setzen Sie dieses ebenfalls mit **Reset** auf die Standardwerte zurück. Rotieren Sie nun die Floor-Instanz auf (-90,0,0), damit dieser wie eine Bodenplatte positioniert ist. Und schon haben wir die erste Bodenplatte fertig.

Kopieren Sie nun diese Platte beliebig oft und platzieren Sie diese nebeneinander. Dabei ist es am einfachsten, immer eine einzelne Platte zu kopieren, z. B. mit **Strg**+**D**, und diese gleich auf die neue Position, z. B. (0,0,2), zu verschieben, bevor Sie sich der nächsten Bodenplatte annehmen. Das Neupositionieren würde ich dabei immer über die *Position*-Parameter im *Inspector* machen, da das am schnellsten und genauesten ist.

Auf diese Weise legen Sie nun nach und nach fest, wo später die Gänge und Hallen Ihres Dungeons sein sollen.



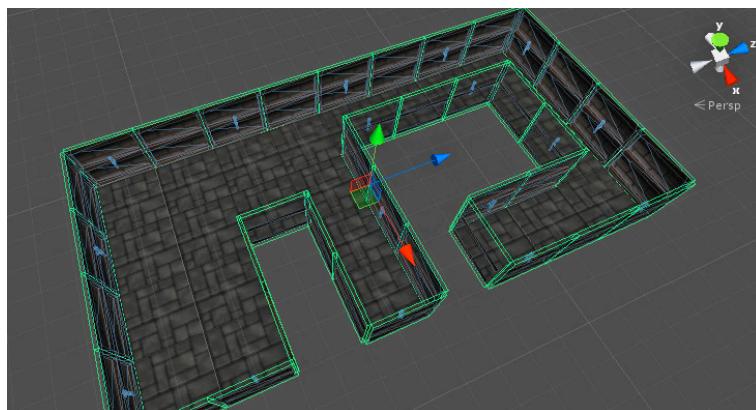
**Bild 21.5** Dungeon-Boden

### 21.1.4.2 Wände erstellen

Zum Erstellen der Wände fügen Sie nun sowohl eine „Wall“-Instanz wie auch eine „Wall\_Torch“-Instanz dem „Walls“-Container zu und setzen beide wieder mit **Reset** auf die Standardwerte zurück.

Positionieren Sie diese nun aufrecht an den äußeren Bodenplatten, sodass die sichtbaren Flächen nach innen zu den Bodenflächen zeigen, z.B. auf den Positionen (0,2,-2) und (2,2,-2). Diese Instanzen kopieren Sie nun wieder wie bei den „Floor“-Instanzen und ordnen diese nach Belieben an den äußeren Rändern der Bodenplatten, sodass am Ende die gesamte Bodenfläche einmal eingerahmt ist.

Da die Fackeln der „Wall\_Torch“-Instanzen für die Beleuchtung später zuständig sind, sollten Sie darauf achten, dass ausreichend viele Instanzen im Level verteilt sind, damit auch alle Ecken ausgeleuchtet sind.



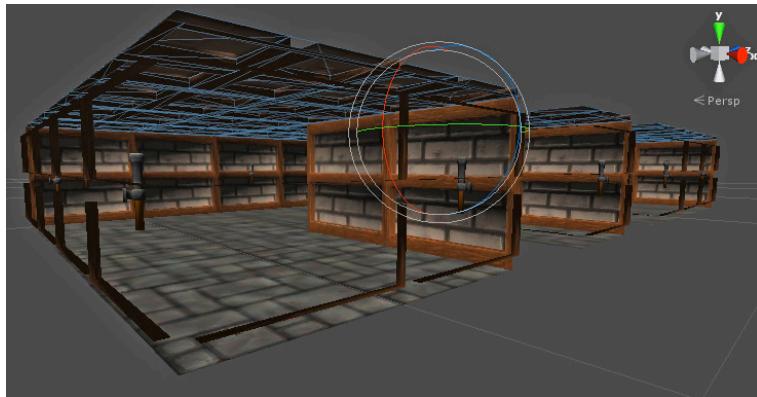
**Bild 21.6** Dungeon mit Boden und Wänden

### 21.1.4.3 Decke erstellen

Zum Schluss muss noch die Decke des Dungeon erstellt werden. Hierbei ist das Vorgehen fast identisch wie bei den Bodenflächen, nur dass hier die sichtbare Seite natürlich nach unten zeigt.

Fügen Sie hierfür eine „Ceiling“-Instanz dem „Ceilings“-Container zu und setzen Sie dieses mit **Reset** zurück. Verschieben Sie dieses nun auf die Position (0,2,0) und rotieren Sie dieses mit dem Rotation-Parameter auf (90,0,0), sodass diese genau über einer „Floor“-Instanz im Abstand der Höhe einer Wall-Instanz liegt.

Nun kopieren Sie diese Instanz und verschieben Sie diese so häufig, bis schließlich das gesamte Dungeon einmal komplett von den Boden-, Wand- und den Decken-Elementen eingeschlossen ist.



**Bild 21.7** Dungeon mit Decke

#### 21.1.4.4 Feuer und Licht zufügen

Erzeugen Sie nun wie im Kapitel „Partikeleffekte mit Shuriken“ beschrieben den Partikel-effekt eines Fackelfeuers und erzeugen Sie daraus ein *Prefab* namens „TorchFlames“. Hierbei ist es vorteilhaft, wenn Sie hierfür eine neue Testszene erstellen, in der Sie diesen Partikeleffekt entwickeln.

Haben Sie nun den Partikeleffekt erstellt, ziehen Sie noch ein „Wall\_Torch“-*Prefab* in die Szene. Jetzt ziehen Sie das „TorchFlames“-Objekt auf das „Torch“-Objekt von „Wall\_Torch“, damit die Flamme ein Kind-Objekt der Fackel wird. Positionieren Sie die Flamme direkt über der Fackel (-0,1,0,0,35). Markieren Sie nun „Wall\_Torch“ in der Szene und drücken Sie in dessen *Inspector* auf **Apply**, damit die Anpassungen auf das *Prefab* übertragen werden.

Nun können Sie die Testszene speichern und Ihre „Dungeon“-Szene öffnen. Wie Sie sehen werden, haben auch alle „Wall\_Torch“-Instanzen Ihrer Dungeons-Szene die Flammen erhalten.

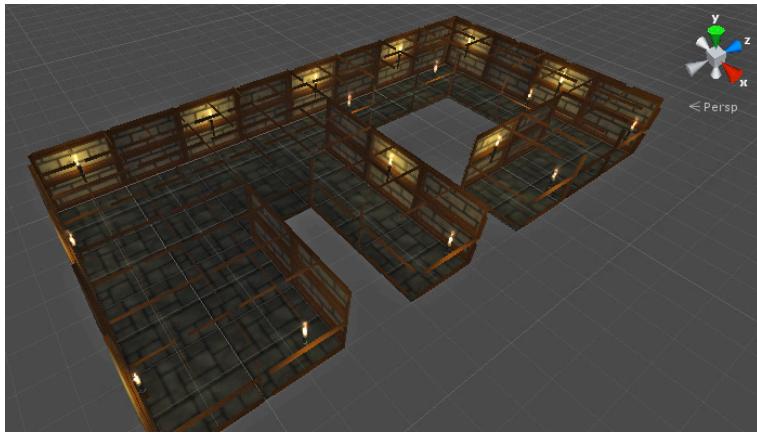
Wechseln Sie wieder in die vorherige Testszene, in der Sie die Partikeleffekte erstellt haben, und vervollständigen die Fackel. Fügen Sie hierzu der Szene über **GameObject/Create Other/Point Light** ein neues *PointLight* hinzu. Dieses ziehen Sie ebenfalls auf „Torch“ von „Wall\_Torch“ und positionieren es dort auf (0,0,0,4), sodass sich das *PointLight* direkt in der Flamme befindet.

Nun legen Sie noch folgende Parameterwerte der *Light*-Komponente fest:

- **Color:** R 254, G 254, B 139, A 255
- **Range:** 3
- **Intensity:** 1

Abschließend selektieren Sie wieder „Wall\_Torch“ und drücken **Apply** im *Inspector* der *Prefab*-Instanz. Jetzt können Sie wieder in Ihr „Dungeon“ wechseln, wo das Gemäuer durch die *PointLights* erhellt wird.

Im Grunde sind wir nun fertig. Allerdings möchte ich an dieser Stelle noch das *Ambient Light* dieser Szene anheben, damit wir im gesamten Dungeon eine gewisse Grundhelligkeit bekommen. Hierfür gehen Sie im Hauptmenü auf **Edit/Render Settings** und klicken dort auf den Farbbalken des *Ambient Light*-Parameters. Dort setzen Sie die R-, G- und B-Werte auf jeweils 100. Den Alphakanal belassen Sie auf 255.



**Bild 21.8** Dungeon mit Fackeln, Licht und angehobenem Ambient Light

## 21.1.5 Dekoration erstellen

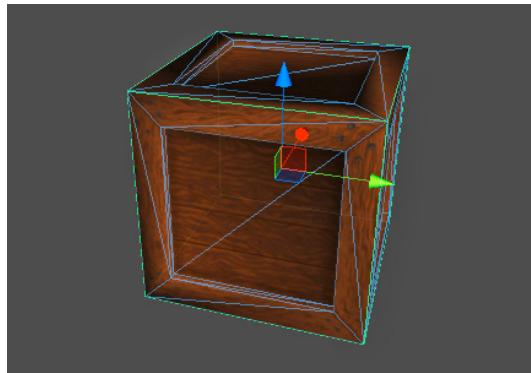
Zu einer Szene gehören nicht unbedingt nur spielrelevante Gegenstände. Normalerweise wird dort auch jede Menge Objekte aus rein dekorativen Zwecken platziert. In unserem Game haben wir hierfür ebenfalls einige Objekte, die wir in dem Dungeon platzieren können.

Um auch hier stets die Übersicht zu behalten, wollen wir alle Deko-Gegenstände ebenfalls in einem gemeinsamen Container-Objekt gruppieren. Erstellen Sie wieder ein leeres GameObject und nennen es dieses Mal „Deco“. Setzen Sie auch diese Container-Werte anschließend mit **Reset** zurück.

### 21.1.5.1 Crate Prefab

Als Erstes wollen wir für die bereits importierte Holzkiste ein *Prefab* erstellen.

1. Ziehen Sie das Modell „crate\_01“ in die Szene und benennen Sie es dort in „Crate“ um.
2. Überprüfen Sie die Skalierung, dass dieses (1,1,1) beträgt.
3. Kontrollieren Sie das zugewiesene Material. Dies sollte „placeables\_01\_diff“ sein, das die gleichnamige Textur nutzt und den *Shader* „Diffuse“ eingestellt hat.
4. Fügen Sie dem *GameObject* einen *Box Collider* zu.
5. Erzeugen Sie nun daraus ein *Prefab*, indem Sie das *GameObject* in den „Prefabs“-Ordner ziehen.

**Bild 21.9**

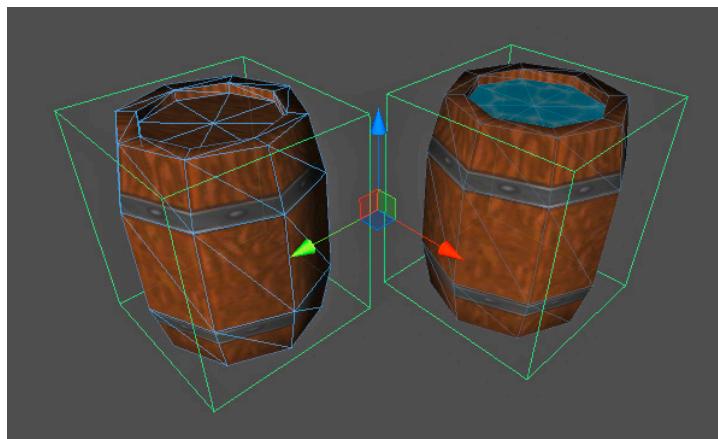
Prefab einer Holzkiste

### 21.1.5.2 Barrel Prefabs

Nun wollen wir noch aus den importierten Fässern eigene *Prefabs* erstellen. Hier haben wir nämlich zwei unterschiedliche: ein verschlossenes Fass und eines, das mit Wasser gefüllt ist.

1. Ziehen Sie das Modell „barrel\_01“ in die Szene und benennen Sie es dort in „Barrel“ um.
2. Kontrollieren Sie auch hier das zugewiesene Material. Dies sollte ebenfalls „placeables\_01\_diff“ sein, das die gleichnamige Textur nutzt und den *Shader* „Diffuse“ eingestellt hat.
3. Aus Vereinfachungsgründen fügen Sie dem *GameObject* ebenfalls einen *Box Collider* zu. Um es genauer zu machen, können Sie aber auch stattdessen in den *Import Settings* dieses Modells *Generate Colliders* aktivieren.
4. Erzeugen Sie nun daraus ein *Prefab*, indem Sie das *GameObject* in den „Prefabs“-Ordner ziehen.

Das gleiche Vorgehen machen Sie nun auch mit dem Modell „barrel\_01\_water“ und nennen es einfach „Barrel\_Water“.

**Bild 21.10** Prefabs zweier Fässer

## ■ 21.2 Inventarsystem erstellen

Ein Kernelement der allermeisten Rollenspiele ist ein Inventarsystem, das dazu dient, Gegenstände (Items) zu verwalten, die der Held im Laufe des Spiels findet. Meistens spielt es auch eine wichtige Rolle beim Lösen von Quests (Aufgaben), wo es häufig darum geht, bestimmte Items zu finden und diese zu einer Person oder zu einem bestimmten Ort zu bringen. Auch in unserem Beispiel-Game wird der Spieler verschiedene Gegenstände finden müssen, die schlussendlich mit diesem Inventarsystem verwaltet werden.

### 21.2.1 Verwaltungslogik

Als Kernelement unseres Inventarsystems soll ein Objekt der generischen Klasse `Dictionary<TKey, TValue>` dienen, die Sie bereits im Kapitel „C# und Unity“ kennengelernt haben.

Ein *Dictionary* verwaltet sogenannte Schlüssel-Wert-Paare. Jeder Datensatz hat hierbei also einen eindeutigen Identifizierer, dessen Wert aber beliebig verändert werden kann. Das bedeutet in unserem Fall, dass der Name des Gegenstandes als Schlüssel genommen werden könnte und als Wert die Anzahl der einzelnen Gegenstände genommen wird. Wird nun ein neues Inventar-Item dem *Dictionary* zugefügt, brauchen Sie einfach nur den Wert des jeweiligen Schlüssels um eins hochzuzählen. Im Kapitel „C# und Unity“ hatte ich bereits ein solches Beispiel vorgestellt.

#### **ItemProperties**

In unserem Fall wollen wir aber noch ein bisschen weitergehen. Denn wir wollen nicht nur die Gegenstände zählen, wir wollen sie auch in der GUI unseres Spiels anzeigen. Hierfür brauchen wir aber noch einen weiteren Wert, der im *Dictionary* gespeichert werden muss, nämlich die zum Gegenstand zugehörige Grafik. Aus diesem Grund können wir nicht einfach die Anzahl als Wert nehmen, sondern wir müssen einen eigenen Typ erstellen, der sowohl die Anzahl wie auch die Grafik des Items beinhaltet.

Hierfür erstellen wir eine kleine Klasse, die wir `ItemProperties` nennen. Sie besitzt lediglich eine `Texture2D`-Variable und eine `int`-Variable, die die beiden genannten Parameter für ein Inventar-Item speichert. Da wir die Klasse aber nicht als Komponente nutzen werden, erbt sie auch nicht von . Den Inhalt dieser Klasse können Sie dem Listing 21.1 entnehmen, das komplette Skript finden Sie im Abschnitt „`ItemProperties.cs`“.

**Listing 21.1** Inhalt von `ItemProperties.cs`

```
public class ItemProperties {
    public Texture2D texture;
    public int quantity;
}
```

## Inventory

Mithilfe der Klasse `ItemProperties` können wir nun für unsere Inventarverwaltung das `Dictionary`-Objekt erstellen, das wir in unserem Fall `items` nennen werden.

Zusätzlich brauchen wir noch zwei Array-Objekte, mit denen wir die Inhalte des Inventars in der GUI anzeigen. Das erste Array dient dem Anzeigen der Grafiken, das zweite zeigt die Mengen an. Dank der Arrays sind Sie später flexibler, wie groß Ihr Inventar tatsächlich sein soll bzw. wie viel die GUI darstellen kann.

### **Listing 21.2** Variablendeclaration von Inventory.cs

```
public GUITexture[] guiItemTextures;
public GUIText[]guiItemQuantities;
private Dictionary<string,ItemProperties> items =
    new Dictionary<string, ItemProperties>();
```

Beachten Sie hierbei, dass Sie für das Nutzen der `Dictionary`-Klasse vorher mit `using System.Collections.Generic` den dazugehörigen Namespace einbinden müssen.

Um die Inhalte nun anzuzeigen, entwickeln wir eine kleine Methode, die später jedes Mal aufgerufen werden soll, wenn sich im Inventar etwas verändert hat und dies in der GUI aktualisiert werden soll. Diese durchläuft alle Items der Arrays und füllt sie mit den jeweiligen Inhalten, nachdem alle zuerst mit Leerwerten initialisiert wurden.

### **Listing 21.3** UpdateView-Methode von Inventory.cs

```
void UpdateView() {
    int index = 0;
    int guiCount = guiItemTextures.Length;

    for(int i = 0; i< guiCount; i++)
    {
        guiItemTextures[i].texture = null;
        guiItemQuantities[i].text = "";
    }

    foreach(KeyValuePair<string,ItemProperties> current in items)
    {
        guiItemTextures[index].texture = current.Value.texture;
        guiItemQuantities[index].text = current.Value.quantity.ToString();
        index++;
    }
}
```

Dies ist auch das Erste, was ausgeführt wird, sobald das Skript gestartet wird.

### **Listing 21.4** Start-Methode von Inventory.cs

```
void Start()
{
    UpdateView();
}
```

Nun kommen die eigentlichen Methoden, mit denen Sie das Inventar füllen und entfernen. Hierfür wollen wir zwei Methoden erstellen. Die Methode `AddItem` dient dem Zufügen eines neuen Items. Ihr werden der Name, also der Schlüssel des Datensatzes, sowie die Textur übergeben, durch die das Item in der GUI dargestellt werden soll.

Die Methode kontrolliert dann, ob das Item bereits im *Dictionary* enthalten ist, und zählt dann die Menge hoch. Damit wir nicht mehr Items in unser Inventar aufnehmen, als wir anzeigen können, wird noch vor dem Erzeugen eines neuen *Dictionary*-Datensatzes überprüft, ob bereits alle GUI-Objekte mit Item-Grafiken belegt sind. Mehr zu diesem Punkt erfahren Sie noch im Abschnitt „Oberfläche des Inventarsystems“.

**Listing 21.5** AddItem von Inventory.cs

```
public bool AddItem(string itemName, Texture2D texture)
{
    if(!items.ContainsKey(itemName))
    {
        if (items.Count < guiItemTextures.Length)
        {
            ItemProperties ip = new ItemProperties();
            ip.texture = texture;
            ip.quantity = 1;
            items.Add(itemName, ip);
            UpdateView();
            return true;
        }
        else
        {
            return false;
        }
    }
    else
    {
        items[itemName].quantity += 1;
        UpdateView();
        return true;
    }
}
```

Zuletzt kommt noch die RemoveItem-Methode, die dafür sorgt, dass ein Item aus dem Inventar entfernt wird. Über einen booleschen Rückgabewert signalisieren wir, ob dies erfolgreich bzw. ob das abgefragte Item überhaupt noch im Inventar enthalten war.

**Listing 21.6** RemoveItem von Inventory.cs

```
public bool RemoveItem(string itemName)
{
    if (items.ContainsKey(itemName))
    {
        if(items[itemName].quantity == 1)
            items.Remove(itemName);
        else
            items[itemName].quantity -= 1;
        UpdateView();
        return true;
    }
    else
        return false;
}
```

Das komplette Skript finden Sie im Abschnitt „Inventory.cs“.

## InventoryItem

Zuletzt wollen wir noch ein Skript entwickeln, das jedem Item angehängt wird, das dem Inventar zugefügt werden können soll. Hier definieren wir drei *public*-Variablen zum Festlegen des Item-Namens, der Textur, die in der GUI angezeigt werden soll, und einem *AudioClip*, der beim Aufpicken abgespielt werden soll. Weiter benötigen wir noch zwei *private*-Variablen, die das Inventar-Skript und den Spielercharakter (bzw. dessen *Transform*) zwischenspeichern.

**Listing 21.7** Variablen Deklaration im Kopf von InventoryItem.cs

```
public string itemName = "";
public Texture2D texture;
public AudioClip picSound;
private Inventory inventory;
private Transform player;
```

Die beiden *private*-Variablen werden dann in der Start-Methode zugewiesen

**Listing 21.8** Start-Methode von InventoryItem.cs

```
void Start () {
    player = GameObject.FindGameObjectWithTag("Player").transform;
    inventory = GameObject.FindGameObjectWithTag("Inventory").
        GetComponent<Inventory>();
}
```

Beachten Sie hierbei, dass wir dem *GameObject* mit dem Inventar-Skript später den Tag „Inventory“ und dem Spieler den Tag „Player“ zuweisen müssen.

Im nächsten Schritt wollen wir nun, dass das *GameObject*, dem dieses Skript angehängt wird, auch dem Inventar zugefügt wird. Dies wollen wir per Klick machen. Sobald also jemand dieses Item mit der Maus anklickt, soll der Gegenstand aus der Spielwelt entfernt werden und in das Inventar wandern. Um den Vorgang zu unterstreichen, wird hierbei der *picSound* abgespielt.

**Listing 21.9** OnMouseDown-Methode von InventoryItem.cs

```
void OnMouseDown()
{
    if (inventory.AddItem(itemName,texture))
    {
        if (picSound != null)
            AudioSource.PlayClipAtPoint(picSound,player.position);
        Destroy(gameObject);
    }
}
```

### 21.2.1.1 ItemProperties.cs

Das im vorherigen Abschnitt entwickelte ItemProperties-Skript sieht dann wie im folgenden Listing aus.

**Listing 21.10 ItemProperties.cs**

```
using UnityEngine;
using System.Collections;

public class ItemProperties {
    public Texture2D texture;
    public int quantity;
}
```

**21.2.1.2 Inventory.cs**

Das Skript *Inventory* ist das Herzstück dieses Inventarsystems und sieht wie in dem folgenden Listing 21.11 aus.

**Listing 21.11 Inventory.cs**

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
public class Inventory : MonoBehaviour {

    public GUITexture[] guiItemTextures;
    public GUIText[] guiItemQuantities;
    private Dictionary<string,ItemProperties> items =
        new Dictionary<string, ItemProperties>();

    void Start()
    {
        UpdateView();
    }

    public bool AddItem(string itemName, Texture2D texture)
    {
        if(!items.ContainsKey(itemName))
        {
            if (items.Count < guiItemTextures.Length)
            {
                ItemProperties ip = new ItemProperties();
                ip.texture = texture;
                ip.quantity = 1;
                items.Add(itemName, ip);
                UpdateView();
                return true;
            }
        }
        else
        {
            return false;
        }
    }
    else
    {
        items[itemName].quantity += 1;
        UpdateView();
        return true;
    }
}
```

```

public bool RemoveItem(string itemName)
{
    if (items.ContainsKey(itemName))
    {
        if(items[itemName].quantity == 1)
            items.Remove(itemName);
        else
            items[itemName].quantity -= 1;
        UpdateView();
        return true;
    }
    else
        return false;
}

void UpdateView()
{
    int index = 0;
    int guiCount = guiItemTextures.Length;

    for(int i = 0; i < guiCount; i++)
    {
        guiItemTextures[i].texture = null;
        guiItemQuantities[i].text = "";
    }

    foreach(KeyValuePair<string, ItemProperties> current in items)
    {
        guiItemTextures[index].texture = current.Value.texture;
        guiItemQuantities[index].text = current.Value.quantity.ToString();
        index++;
    }
}
}

```

### 21.2.1.3 InventoryItem.cs

Im nächsten Listing sehen Sie nun das zuletzt entwickelte Skript zum Zufügen von Gegenständen in das Game.

**Listing 21.12** InventoryItem.cs

```

using UnityEngine;
using System.Collections;

public class InventoryItem : MonoBehaviour {

    public string itemName = "";
    public Texture2D texture;
    public AudioClip picSound;
    private Inventory inventory;
    private Transform player;
    void Start () {
        player = GameObject.FindGameObjectWithTag("Player").transform;
        inventory = GameObject.FindGameObjectWithTag("Inventory").
            GetComponent<Inventory>();
    }
}

```

```

    }

    void OnMouseDown()
    {
        if (inventory.AddItem(itemName, texture))
        {
            if (picSound != null)
                AudioSource.PlayClipAtPoint(picSound, player.position);
            Destroy(gameObject);
        }
    }
}

```

## 21.2.2 Oberfläche des Inventarsystems

Zum Darstellen der Inhalte dieses Inventarsystems benötigen Sie pro Item zwei GUI-Objekte: ein grafisches GUI-Objekt zum Darstellen der Item-Grafik und ein textorientiertes GUI-Objekt zum Darstellen der Menge.

In unserem Fall wollen wir es einfach halten und das Inventarsystem unten links im Spiel einblenden. Das Einzige, was zu den obigen Objekten noch hinzukommt, ist eine Hintergrundgrafik pro Item, die dauerhaft eingeblendet bleibt, um so den Platz des Inventarsystems darzustellen.

Insgesamt werden wir in dem Spiel mit drei unterschiedlichen Item-Typen zu tun haben (siehe Abschnitt „Inventar-Items“), wobei durch den Spielverlauf höchstens zwei Typen gleichzeitig in dem Inventar sein können. Da wir deshalb auch nur zwei Item-Plätze benötigen, brauchen wir dementsprechend auch nur

- zwei Texturelemente für die Hintergründe
- zwei Texturelemente für die Item-Grafiken
- zwei Textelemente zum Darstellen der Mengen

Sollten Sie später das Spiel erweitern wollen, werden Sie sicher mehr Plätze brauchen. Da wir aber in dem *Inventory*-Skript mit Arrays arbeiten, steht dem Erweitern des Inventar-GUIs nichts im Wege.

In den *Import Settings* dieser Item-Grafiken („key“, „stone“ und „bucket“) sollten Sie nun noch den *Texture Type* auf *GUI* und die *Max Size* auf 64 stellen.

Bevor wir aber mit den eigentlichen GUI-Objekten beginnen, erstellen Sie zuvor ein leeres *GameObject*, das Sie mit **Reset** auf die Default-Werte zurücksetzen und in „Inventory“ umbenennen. Fügen Sie diesem nun das gleichnamige Skript *Inventory* zu, das wir weiter oben erstellt haben. Nun erstellen Sie in der Tag-Liste einen neuen Tag „Inventory“ und weisen diesen dem *GameObject* „Inventory“ zu.

Als Erstes erzeugen Sie zwei *GUIText-Objects* und vier *GUITexture-Objects* und fügen diese dem *GameObject* „Inventory“ als Kind-Objekte zu.

Benennen Sie die vier *GUITexture-Objects* in „Tile1-Bg“, „Tile2-Bg“, „Tile1-Value“ und „Tile2-Value“ um. Nennen Sie die beiden *GUIText-Objects* um in „Tile1-Quantity“ und „Tile2-Quantity“. Achten Sie darauf, dass die *GUITexture*-Objekte die Skalierung (0,0,1) besitzen.

Weisen Sie den *Texture*-Eigenschaften von „Tile1-Bg“ und „Tile2-Bg“ die Textur „inventory-tile“ zu. Stellen Sie bei dieser den *Texture Type* ebenfalls auf *GUI* und die *Max Size* auf 64. Danach stellen Sie bei beiden Objekten die *Position* auf (0,0,-1), damit diese weiter hinten angeordnet werden. Setzen Sie bei „Tile1-Value“ und „Tile2-Value“ die *Position* auf (0,0,0), damit diese vor den beiden Hintergrundtexturen erscheinen.

Nun setzen Sie die *Pixel Inset*-Werte dieser vier Objekte. Bei „Tile1-Bg“ und „Tile1-Value“ setzen Sie *X* 20, *Y* 20, *W* 64, *H* 64. Bei „Tile2-Bg“ und „Tile2-Value“ setzen Sie *X* 100, *Y* 20, *W* 64, *H* 64.

Als Letztes müssen die beiden Text-Objekte „Tile1-Quantity“ und „Tile2-Quantity“ mit den richtigen Parameter bestückt werden. Die Mengen der jeweiligen Items sollen dabei immer unten rechts bei der jeweiligen Grafik angezeigt werden. Setzen Sie deshalb bei beiden Objekten die folgenden Werte:

- *Position*: (0,0,1).
- *Anchor*: *upper right*
- *Alignment*: *right*

Abschließend setzen Sie noch die *Pixel Offset*-Werte, die die Positionen der Texte bestimmen. Bei „Tile1-Quantity“ setzen Sie *X* 75, *Y* 40 und bei „Tile2-Quantity“ *X* 155, *Y* 40.

Jetzt legen Sie die *Size* von *guiItemTextures* und *guiItemQuantities* vom *Inventory*-Skript auf 2. Danach fügen Sie „Tile1-Value“ und „Tile2-Value“ der Reihenfolge entsprechend dem *guiItemTextures*-Array zu und „Tile1-Quantity“ und „Tile2-Quantity“ dem *guiItemQuantities*-Array zu.

Zum Testen können Sie nun die *Text*-Eigenschaft der beiden „Quantity“-Objekte mit unterschiedlichen Zahlenwerten belegen und den *Texture*-Eigenschaften der „Value“-Objekte Item-Grafiken (z.B. die „Key“-Textur) zuweisen. Wenn Sie ohne das Spiel zu starten die *Game View* betrachten, sollte nun das Inventar unten links zu sehen sein.



**Bild 21.11** Inventar-GUI mit GUIElements

### 21.2.3 Inventar-Items

In unserem Spiel haben wir es mit drei unterschiedlichen Item-Typen zu tun. Diese wollen wir in diesem Abschnitt erstellen.

#### HoverEffects

Zuvor wollen wir aber noch ein kleines Skript programmieren, das einen Informationstext einblenden soll, sobald der Spieler mit der Maus auf ein Objekt zeigt. Außerdem soll dabei dem Spieler mitgeteilt werden, ob er dieses Objekt aufnehmen und seinem Inventar zufügen kann. Dies wollen wir durch das Ändern des Cursors erzielen.

Da der Informationstext immer direkt bei dem jeweiligen Item erscheinen soll, gibt es nun zwei unterschiedliche Varianten der Umsetzung. Entweder wir platzieren das Text-Objekt direkt in der dreidimensionalen Welt bei dem jeweiligen Objekt (z.B. mit einer 3DText-Komponente), oder wir nutzen ein normales GUI-Objekt, das dann in dem zweidimensionalen Pixelkoordinatensystem der GUI so positioniert wird, dass es vor dem jeweiligen Objekt erscheint.

Der Vorteil des letzteren Verfahrens besteht darin, dass die Schrift immer gleich groß wäre, unabhängig davon, wie weit ein Objekt entfernt ist. Und deshalb wollen wir auch dieses Verfahren nehmen. Wir werden also als Erstes ein allgemeines *GUIText*-Objekt erstellen, das wir später immer an der jeweiligen Position platzieren wollen, wo ein Informationstext angezeigt werden soll.

1. Erzeugen Sie hierfür zunächst ein *GUIText*-Objekt über **GameObject/Create Other/GUI Text**, das Sie „TooltipText“ nennen.
2. Positionieren Sie das *GameObject* auf (0,0,0).
3. Nun legen Sie einen neuen Tag „Tooltip“ an und weisen diesen dem *GameObject* „TooltipText“ zu.
4. Stellen Sie *Anchor* auf „middle center“ und *Alignment* auf „center“.
5. Entfernen Sie den zugewiesenen Default-Text.
6. Wenn Sie möchten, können Sie auch einen anderen TrueType Font bzw. OpenType Font dem Parameter *Font* zuweisen und über die Eigenschaften *Font Size*, *Font Style* und *Color* diesen beliebig anpassen.

Nun erzeugen wir ein neues Skript namens „HoverEffects“. Zunächst brauchen wir eine Referenz auf das *GUIText*-Objekt bzw. auf die Komponente sowie eine Variable für den Informationstext. Außerdem wollen wir den Cursor verändern. Da wir bei unserem Spiel sowieso einen anderen Cursor als den Standard-Betriebssystemzeiger nutzen wollen, verwenden wir hier nun zwei Variablen: *standardCursor* und *clickableCursor*. Mit einer zusätzlichen Variablen *hotSpot* geben wir noch die Position des Cursors an, die aber einfach auf *zero* verbleibt.

**Listing 21.13** Variablen und Start-Methode von HoverEffects

```
public Texture2D standardCursor;
public Texture2D clickableCursor;
public string tooltipText;
private Vector2 hotSpot = Vector2.zero;
```

```

private GUIText msgBox;

void Start()
{
    msgBox = GameObject.FindGameObjectWithTag("Tooltip").GetComponent<GUIText>();
    msgBox.transform.position = Vector3.zero;
}

```

Die MouseEnter-Methode, die ausgelöst wird, sobald sich die Maus über diesem Objekt befindet, sieht damit wie folgt aus.

#### **Listing 21.14** OnMouseEnter-Methode von HoverEffects

```

void OnMouseEnter()
{
    if (clickableCursor != null)
        Cursor.SetCursor(clickableCursor, hotSpot, CursorMode.Auto);
    msgBox.text = tooltipText;
    Vector2 pos;
    pos.x = Input.mousePosition.x ;
    pos.y = Input.mousePosition.y + 20; //wir legen den Text etwas über den Zeiger.
    msgBox.pixelOffset = pos;
}

```

Den Exit-Effekt gliedern wir in eine extra Methode aus, da wir den Code zum Zurücksetzen des Cursors sowohl in OnMouseExit wie auch in OnDestroy aufrufen wollen.

#### **Listing 21.15** Zurücksetzen der Hover-Effekte in HoverEffects

```

void OnMouseExit()
{
    ExitEffect();
}

void OnDestroy()
{
    ExitEffect();
}

void ExitEffect()
{
    if (standardCursor != null)
        Cursor.SetCursor(standardCursor, hotSpot, CursorMode.Auto);
    if (msgBox != null)
        msgBox.text = "";
}

```

### **Default Cursor**

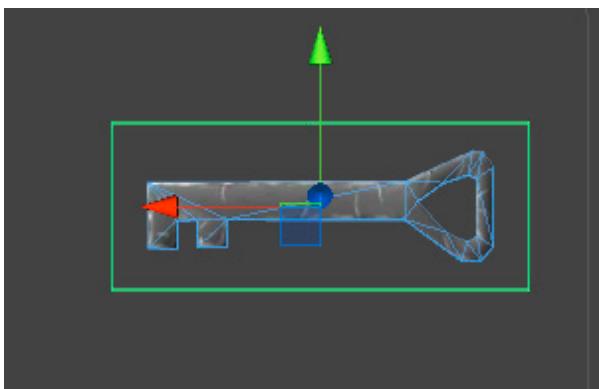
Und wo wir gerade beim Cursor sind, ändern wir auch gleich noch den allgemeinen *Default Cursor* des Spiels. Statt des normalen Cursors wollen wir einen eigenen Zeiger in Form eines Schwertes im Spiel anzeigen. Diese Bild weisen wir dann auch später der Variablen *standardCursor* des HoverEffects-Skriptes zu.

Gehen Sie zum Zuweisen des *Default-Cursors* auf **Edit/Project Settings/Player** und weisen Sie dem Parameter „Default Cursor“ die Grafik „icon\_01\_32x32“ zu.

## Schlüssel

Nun können wir uns um unsere Items kümmern. Als Erstes wollen wir einen Schlüssel erstellen, den wir später zum Öffnen eines verschlossenen Tores benötigen.

1. Ziehen Sie das Modell „iron\_key\_01“ in Ihre Szene und nennen Sie das *GameObject* in „IronKey“ um.
2. Weisen Sie diesem das Material „placeables\_01\_diff“ zu, das die gleichnamige Textur mit dem *Shader* „Diffuse“ nutzt.
3. Fügen Sie ihm nun einen *Box Collider* zu und vergrößern Sie diesen *Collider* (nicht das *GameObject*) über dessen *Size* leicht in alle Richtungen, z.B. auf (0,2,0,08,0,02). Da der Schlüssel recht klein ist, vergrößern wir hierdurch die Kontaktfläche des Schlüssels, damit der Spieler ihn später leichter selektieren kann.
4. Fügen Sie dem Schlüssel nun das *HoverEffects*-Skript zu. Weisen Sie *standardCursor*, „icon\_01\_32x32“, *clickableCursor* „icon\_02\_32x32“ und *tooltipText* „Eiserner Schlüssel“ zu. Kontrollieren Sie bei beiden Texturen die Import Settings. Der *Texture Type* sollte auf „Cursor“ stehen, der *Wrap Mode* auf „Clamp“ und die *Max Size* auf 32.
5. Fügen Sie abschließend dem Schlüssel noch das Skript *InventoryItem* zu. *itemName* lautet „Key“, *texture* weisen Sie die Textur „key“ zu und als *picSound* nehmen Sie „picUp“. Auch bei der Textur „key“ sollten Sie noch einmal die *Import Settings* überprüfen. Hier sollte der *Texture Type* auf „GUI“ stehen und die *Max Size* auf 64.
6. Als Letztes ziehen Sie das fertige *GameObject* in den „Prefab“-Ordner, um ein *Prefab* zu erstellen.



**Bild 21.12** Schlüssel

## Wasserbehälter

Das Gleiche machen Sie nun mit einigen Änderungen auch mit dem Modell „bucket\_01\_empty“. Diese Änderungen will ich kurz erläutern:

- Der Name des *GameObjects* wird auf „BucketEmpty“ geändert.
- Bei *HoverEffects* weisen Sie als *toolTipText* „Wasserbehälter“ zu.
- Bei *InventoryItem* weisen Sie als *itemName* „Bucket“ und als *texture* „bucket“ zu.

- Da wir diesen Behälter noch für andere Zwecke benötigen, fügen wir dem *GameObject* noch ein *Rigidbody* zu.
- Vergrößern Sie bei diesem Objekt die *ColliderSize* nicht!

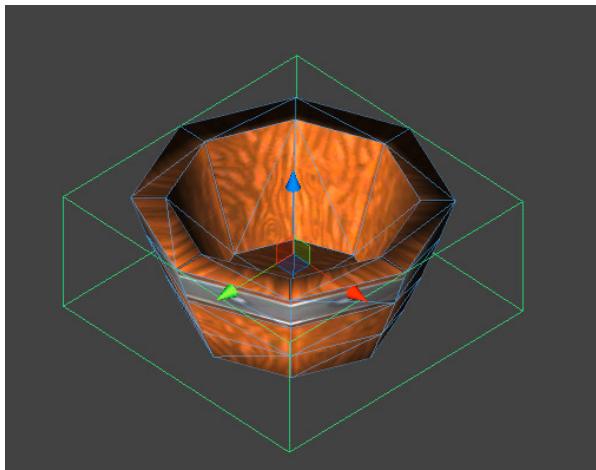


Bild 21.13 Wasserbehälter

## Stein

Und zuletzt machen Sie das auch mit dem Modell „stone\_01“. Allerdings gibt es auch einige Unterschiede, die ich kurz erläutern möchte:

- Der Name wird auf „Stone“ geändert.
- Fügen Sie statt des *Box Colliders* dieses Mal einen *Sphere Collider* zu und reduzieren dessen *Radius* auf 0.08.
- Bei *HoverEffects* weisen Sie *toolTipText* den Text „Wurfstein“ zu.
- Bei *InventoryItem* weisen Sie als *itemName* „Stone“ und als *texture* „stone“.
- Auch hier brauchen Sie nicht die *ColliderSize* zu ändern.

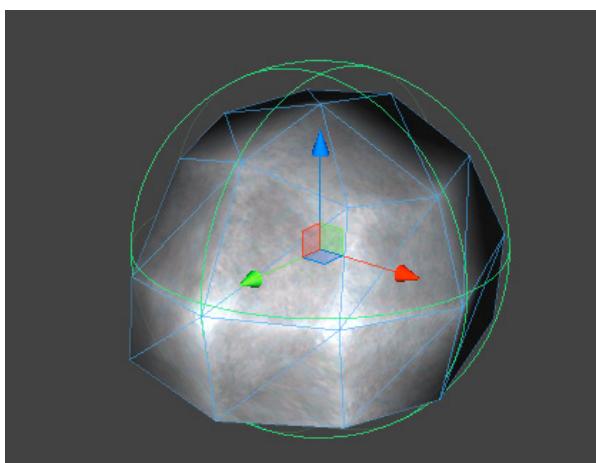


Bild 21.14 Stein

Denken Sie daran, dass Sie aus jedem der drei Inventar-Items auch immer ein *Prefab* erstellen.

### 21.2.3.1 HoverEffects.cs

Das *HoverEffects*-Skript können Sie jedem beliebigen Objekt zuweisen, zu dem in der GUI Informationen angezeigt werden sollen, also auch *GameObjects*, die nicht dem Inventar zugefügt werden können. Weisen Sie diese Textinformation der Variablen *tooltipText* zu. Über die beiden *Texture2D*-Variablen können Sie zudem signalisieren, dass ein Objekt anklickbar und z. B. dem Inventar zugefügt werden kann. Hierfür sollten Sie *clickableCursor* eine cursor-taugliche Textur mit einer Hand oder einem ähnlichen Symbol zuweisen.

**Listing 21.16** HoverEffects.cs

```
using UnityEngine;
using System.Collections;

public class HoverEffects : MonoBehaviour {
    public Texture2D standardCursor;
    public Texture2D clickableCursor;
    public string tooltipText;
    private Vector2 hotSpot = Vector2.zero;
    private GUIText msgBox;

    void Start()
    {
        msgBox = GameObject.FindGameObjectWithTag("Tooltip").GetComponent<GUIText>();
        msgBox.transform.position = Vector3.zero;
    }

    void OnMouseEnter()
    {
        if (clickableCursor != null)
            Cursor.SetCursor(clickableCursor,hotSpot,CursorMode.Auto);

        msgBox.text = tooltipText;

        Vector2 pos;
        pos.x = Input.mousePosition.x ;
        pos.y = Input.mousePosition.y + 20; // um Text etwas höher anzuziegen.
        msgBox.pixelOffset = pos;
    }

    void OnMouseExit()
    {
        ExitEffect();
    }

    void OnDestroy()
    {
        ExitEffect();
    }

    void ExitEffect()
    {
        if (standardCursor != null)
```

```
        Cursor.SetCursor(standardCursor, hotSpot, CursorMode.Auto);  
  
        if (msgBox != null)  
            msgBox.text = "";  
    }  
}
```

## ■ 21.3 Game Controller

In einem Game gibt es immer wieder Funktionalitäten, die sich nicht speziell einem *GameObject* zuweisen lassen, sondern sich auf das komplette Game oder zumindest auf die aktuelle Szene beziehen. Ein gutes Beispiel hierfür ist ein Pausenmenü oder die Hintergrundmusik.

Solche Komponenten werden für gewöhnlich einem *GameObject* namens „Game Controller“ zugewiesen. „Game Controller“ sind typische Bestandteile eines Spiels, weshalb Unity schon per Default einen Tag „GameController“ zur Verfügung stellt, um diese schneller identifizieren zu können.

Da auch wir in unserem Spiel einen „Game Controller“ benötigen, erstellen Sie nun ein neues *GameObject* mit dem Namen „Game Controller“ und weisen diesem den Tag „GameController“ zu.

Als Erstes übergeben Sie nun diesem eine  *AudioSource*, bei der Sie sowohl *Play On Awake* als auch *Loop* aktivieren. Weisen Sie nun die Datei „backgroundMusic“ der  *AudioClip*-Eigenschaft zu und stellen Sie die Lautstärke (*Volume*) zunächst einmal auf 0.3, was Sie aber später gerne Ihrem Geschmack entsprechend anpassen können.

Damit die Hintergrundmusik nun auch überall gleich laut ist und sich auch immer gleichmäßig auf beide Lautsprecher verteilt, deaktivieren wir in den  *Import Settings* von „backgroundMusic“ den Parameter *3D Sound*.

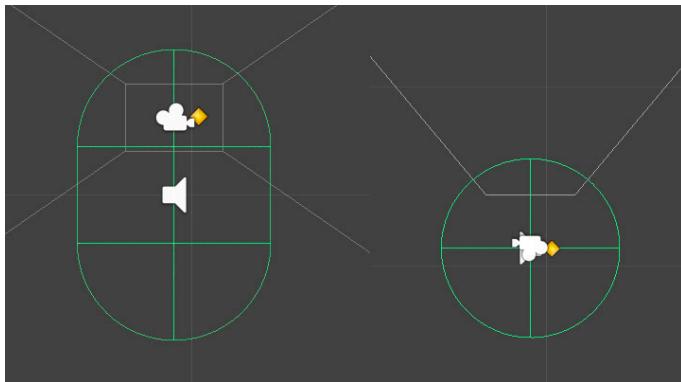
## ■ 21.4 Spieler erstellen

Als Nächstes wollen wir uns nun um den Spieler kümmern. Aufgrund des Umfangs möchte ich dies in drei Teile aufgliedern.

Zunächst werden wir den Spieler mit seinen Hauptkomponenten ausstatten. Danach werden wir die Lebensverwaltung entwickeln, die auch für unseren späteren Gegner wichtig sein wird. Erst danach werden wir uns um die eigentliche Spielersteuerung und das Angreifen kümmern.

1. Zuallererst fügen wir unserer Szene ein neues *Empty GameObject* zu und nennen dieses „Player“. Damit wir diesen später per Code einfacher finden können, geben wir diesem auch gleich noch den Tag „Player“. Unity stellt diesen Tag schon per Default bereit, deshalb brauchen wir uns nicht weiter um das Anlegen dieses Tags zu kümmern.

2. Nun fügen wir dem „Player“ über **Component/Physics/Character Controller** eine *Character Controller*-Komponente zu, die wir später über ein gesondertes Skript steuern werden. Den *Radius* setzen Sie auf 0.5, *Height* setzen Sie auf 1.5.
3. Danach fügen Sie dem *GameObject* eine  *AudioSource* zu, über die wir später alle Sprachsamples des Spielers abspielen wollen. Die Lautstärke können Sie dann Ihren Wünschen entsprechend anpassen. Achten Sie darauf, dass *Play On Awake* und *Loop* deaktiviert sind.
4. Jetzt fügen Sie dem „Player“ die „Main Camera“ als Kind-Objekt zu und setzen diese mit **Reset** auf die Standardwerte zurück. Markieren Sie hierfür „Main Camera“ und drücken Sie **Reset** im *Transform*-Kontextmenü. Anschließend justieren Sie die Position auf die *Position* (0,0,4,0).
5. Nun erzeugen Sie noch ein *Empty GameObject* und nennen dieses „SpawnPoint“. Dieses Objekt dient später als Position für das Werfen der Waffen. Zur besseren Darstellung können Sie dem „SpawnPoint“ im *Inspector* des *GameObjects* ein eigenes Icon geben, z.B. eine gelbe Raute.
6. Fügen Sie das Objekt „SpawnPoint“ dem „Player“ als Kind-Objekt zu. Setzen Sie den „SpawnPoint“ nun ebenfalls mit **Reset** zurück und verschieben ihn anschließend auf die *Position* (0.125,0.4,0), sodass sich dieser direkt neben der Kamera befindet.



**Bild 21.15**  
Player-GameObject  
von hinten und von oben

7. Jetzt drehen Sie den „SpawnPoint“ noch ganz leicht mit einer *Rotation* von (0,357.5,0) nach links. Da die Wurfobjekte sich später an der Ausrichtung dieses Objektes orientieren werden, ermöglichen wir dem Spieler damit ein einfacheres, gezielteres Werfen.
8. Abschließend fügen Sie dem „SpawnPoint“ ebenfalls eine  *AudioSource* zu. Diese wird für das Erzeugen des Wurfgeräusches verantwortlich sein. Deaktivieren Sie wieder *Play On Awake* und *Loop* und fügen der  *AudioSource* den  *AudioClip* „throwing“ zu.
9. Damit der Spieler auch in dunkleren Ecken des Dungeons noch etwas erkennen kann, fügen wir dem „Player“ nun noch eine eigene Lichtquelle zu. Erzeugen Sie hierfür ein  *Point Light-Object* und fügen Sie dieses dem „Player“ als Kind-Objekt zu.
10. Zentrieren Sie das  *Point Light* mit der **Reset**-Funktion und reduzieren Sie die *Intensity* auf 0.5.

Das war es auch mit der Konfiguration des „Player“-*GameObjects*. Platzieren Sie nun Ihren Helden im Dungeon und starten Sie einmal das Spiel, um die Funktionen zu testen.

Wenn alles funktioniert, können Sie Ihren Player-Prototypen in den „Prefabs“-Ordner ziehen, um daraus ein *Prefab* zu erstellen. Falls Sie später das Game um weitere Levels/Szenen erweitern wollen, können Sie so Ihren Helden leichter wiederverwenden.

Als Nächstes wollen wir uns um das Programmieren der Lebensverwaltung und der Spielersteuerung kümmern. Haben Sie diese entwickelt, sollen Sie diese natürlich ebenfalls dem „Player“-*Prefab* zufügen.

### 21.4.1 Lebensverwaltung

In diesem Spiel möchte ich mit Ihnen eine Lebensverwaltung erstellen, die aber nicht nur den einfachen Gesundheitszustand des Spielers verwaltet. Der Spieler soll auch mehrere Lebenspunkte besitzen, sodass das Spiel erst dann zu Ende ist, wenn auch alle Leben verbraucht sind.

#### GUI-Elemente

Zunächst wollen wir aber die *GUIElements* für diese Lebensverwaltung erstellen.

1. Als Erstes erstellen wir ein neues *Empty GameObject*, das wir einfach „GUI“ nennen. Dieses soll als Container-Objekt der GUI-Objekte dienen.
2. Setzen Sie das *GameObject* „GUI“ mit der Funktion **Reset** auf die Default-Werte zurück.
3. Erzeugen Sie gleich noch ein *GUIText*-Objekt, das Sie „LpText“ nennen. Dieses soll dem Anzeigen der Lebenspunkte dienen. Fügen Sie „LpText“ als Kind-Objekt dem Container-Objekt „GUI“ zu. Positionieren Sie nun „LpText“ auf (0,1,0) und einen *Pixel Offset* (33,-49).
4. Um den Wert in der GUI etwas ansprechender zu gestalten, wollen wir den Wert noch mit einer ansprechenderen Grafik unterlegen. Erzeugen Sie hierfür ein *GUITexture*-Objekt und nennen dieses „LpBg“. Setzen Sie *Position* auf (0,1,-1) und *Scale* auf (0,0,1). Legen Sie *Pixel Offset* auf (20,-75,32,32), wobei die letzten Werte für W und H stehen. Weisen Sie *Texture* die Grafik „lifepointHeart“ zu, deren *Texture Type* „GUI“ und *Max Size* 32 sein sollte.
5. Neben den Lebenspunkten wollen wir auch den aktuellen Gesundheitszustand anzeigen. Erstellen Sie hierfür ein *GUITexture*-Objekt, das Sie „HealthBar“ nennen, setzen Sie *Position* auf (0,1,0) und *Scale* auf (0,0,1). Da auch dieses zu der GUI gehört, fügen Sie dieses *GameObject* ebenfalls dem Container „GUI“ zu. Legen Sie *Pixel Offset* von „HealthBar“ auf (20,-35,128,16) und weisen Sie *Texture* die Grafik „healthbar“ zu. Legen Sie den *Texture Type* von „healthbar“ wieder auf „GUI“ und die *Max Size* auf 128.

#### MessageText

Um später Textausgaben wie „Game Over“ oder Ähnliches realisieren zu können, benötigen wir noch ein zusätzliches *GUIText*-Objekt. Zur besseren Identifikation wollen wir dieses zudem noch mit dem Tag „Message“ ausstatten.

1. Erzeugen Sie ein *GUIText-GameObject* über **GameObject/Create Other/GUI Text** (bzw. **GameObject/Create General/GUI Text**) und nennen Sie es „MessageText“.
2. Ziehen Sie es auf das Container-Objekt „GUI“, um aus „MessageText“ ein Kind-Objekt von „GUI“ zu machen.
3. Zentrieren Sie nun das „MessageText“ mithilfe der *Position* (0.5,0.5,0).
4. Jetzt zentrieren Sie auch noch den Text selber innerhalb des *GameObjects* mit *Anchor* „middle center“ und *Alignment* „center“.
5. Entfernen Sie den Default-Text.
6. Legen Sie abschließend einen neuen *Tag* „Message“ an, den Sie dem *GameObject* „MessageText“ zuweisen.

### LifePointController

Kommen wir aber wieder zurück zur eigentlichen Lebensverwaltung. Für diese benötigen wir zunächst ein neues Skript mit dem Namen **LifePointController**, das die Lebenspunkte verwalten soll. Da diese Lebensverwaltung aber eine übergeordnete Ebene darstellt, soll dieses Skript nicht direkt dem Spieler zugefügt werden, sondern dem **Game Controller**.

Haben Sie das Skript dem „Game Controller“ zugefügt, kommen wir nun zur Programmierung des **LifePointController**-Skriptes. Als Variablen benötigen wir lediglich eine *private*-Variable zum Verwalten der Lebenspunkte sowie eine *public*-Variable für den Zugriff auf ein *GUIText*-Objekt, der wir dann das Objekt „LpText“ zuweisen.

#### **Listing 21.17** Variablen des LifePointControllers

```
public GUIText lpText;
private int lifePoints = 0;
```

Da wir den Zugriff auf die *lifePoints*-Variable über Eigenschaftsmethoden realisieren werden, wird die *int*-Variable als *private* deklariert. Dies machen wir aus dem Grund, weil wir bei der Zuweisung des Wertes dafür sorgen wollen, dass *lifePoints* nie einen kleineren Wert als 0 haben kann. Außerdem wollen wir dafür sorgen, dass nach jeder Zuweisung eines neuen Wertes die GUI aktualisiert wird. Hierfür werden wir wieder eine Methode *UpdateView* implementieren, die das übernimmt.

#### **Listing 21.18** LifePoints-Eigenschaft vom LifePointController

```
public int LifePoints
{
    get
    {
        return lifePoints;
    }
    set
    {
        lifePoints = value;
        if(lifePoints < 0)
            lifePoints = 0;
        UpdateView();
    }
}
```

In der `UpdateView`-Methode wird dann lediglich der neue `lifePoints`-Wert dem GUI-Objekt zugewiesen.

**Listing 21.19** `UpdateView`-Methode des `LifePointController`s

```
void UpdateView()
{
    lpText.text = lifePoints.ToString();
}
```

Als Letztes müssen wir uns jetzt noch darum kümmern, dass wir am Anfang einer Szene die aktuelle Anzahl der Lebenspunkte dem Controller zuweisen. Dies werden wir mit der `PlayerPrefs`-Klasse und dem Parameter „LPs“ umsetzen. Am Anfang des Spiels werden wir in einer Eröffnungsszene den Parameter „LPs“ mit einem Startwert abspeichern. Wird nun die „Dungeon“-Szene mit dem `LifePointController` gestartet, lädt sich dieser den Wert in die `lifePoints`-Variable.

**Listing 21.20** `Awake`-Methode vom `LifePointController`

```
void Awake()
{
    LifePoints = PlayerPrefs.GetInt("LPs");
}
```

Am Ende der Szene schreibt das Skript dann seinen aktuellen Wert wieder weg. Hierfür wird die `OnDestroy`-Methode genutzt, die ausgeführt wird, wenn das Skript zerstört wird (was beim Beenden einer Szene der Fall ist).

**Listing 21.21** `OnDestroy`-Methode vom `LifePointController`

```
void OnDestroy()
{
    PlayerPrefs.SetInt("LPs", lifePoints);
}
```

Sollte die Szene nun deshalb beendet werden, weil der Spieler gestorben ist, wird hierbei ein niedrigerer Wert in „LPs“ geschrieben, als er am Anfang hatte. Sobald der Spieler stirbt, wird im `PlayerHealth`-Skript, das wir gleich noch programmieren werden, ein Lebenspunkt von `lifePoints` abgezogen. Und so wird beim neuen Start der Szene ein niedriger Wert geladen. Das komplette Skript `LifePointController` finden Sie im Anschluss dieses Abschnitts.

## HealthController

Kommen wir nun zur eigentlichen Verwaltung des Gesundheitszustandes des Spielers. Da aber auch unsere Gegner ebenfalls einen Gesundheitszustand besitzen, wollen wir nun mit Vererbung arbeiten. Wir werden eine Basisklasse `HealthController` erstellen, auf die dann die Skripte des Spielers wie auch der Gegner aufbauen werden.

Die Klasse `HealthController` besteht nur aus der *public*-Variablen `health`, die den Gesundheitszustand speichert, und der *private*-Variablen `isDead`.

**Listing 21.22** Variablen des HealthControllers

```
public float health = 3;
private bool isDead = false;
```

Weiter besitzt die Klasse die Methode `ApplyDamage`, über die dem Besitzer Schaden zugefügt wird. Dabei wird `health` um den Schadenswert reduziert und es wird eine Methode `Damaging` aufgerufen. Wenn der Schaden allerdings `health` auf 0 oder sogar ins Negative bringt, wird stattdessen eine Methode namens `Dying` aufgerufen und es wird die Variable `isDead` auf TRUE gesetzt.

**Listing 21.23** `ApplyDamage` vom HealthController

```
void ApplyDamage(float damage)
{
    health -= damage;

    if(health <= 0 && !isDead)
    {
        isDead = true;
        Dying();
    }
    else
    {
        Damaging();
    }
}
```

Da sich die beiden Methoden `Damaging` und `Dying` aber zwischen dem Spieler und dem Gegner stark unterscheiden, werden diese nur als leere überschreibbare Methoden deklariert.

**Listing 21.24** Überschreibbare `Damaging`- und `Dying`-Methoden

```
public virtual void Damaging()
{
}

public virtual void Dying ()
{
}
```

## PlayerHealth

Nun können wir auf der Klasse `HealthController` aufbauend das `PlayerHealth`-Skript erstellen. Auch dieses Skript wird wieder eine `UpdateView`-Methode besitzen, in der dieses Mal der `health`-Wert des Spielers grafisch dargestellt werden soll. Allerdings wollen wir in diesem Fall nicht einfach eine Zahl anzeigen, sondern diesen Wert in Form eines Balkens darstellen.

Für diesen Balken nutzen wir das `GUITexture`-Objekt „`HealthBar`“, das wir bereits erstellt haben. Das Objekt weisen wir schließlich der `public`-Variablen `healthGui` zu (siehe Listing 21.26). In der `UpdateView`-Methode wird die Länge dieses Grafik-Objektes schließlich abhängig vom `health`-Wert und einer vorgegebenen Maximallänge berechnet.

**Listing 21.25** UpdateView-Methode von PlayerHealth

```
void UpdateView()
{
    if(healthGui != null) {
        guiRect.width = guiMaxWidth * health/maxHealth;
        healthGui.pixelInset = guiRect;
    }
}
```

Beachten Sie in Listing 21.25 die verschiedenen Variablen, die dort nicht deklariert wurden. Dies sind *public*- und *private*-Variablen, die im Klassenkopf deklariert wurden und in der Start-Methode ihre Werte zugewiesen bekommen. *guiRect* erhält hier die Originalwerte, die dann auch gleich an *guiMaxWidth* weitergegeben werden. Und auch *maxHealth* erhält seinen Wert vom Startwert von *health*.

Zu diesen kommen noch zwei weitere Variablen, die in Start zugewiesen werden. Der Variablen *lifePointController* wird die jeweilige Instanz des *GameControllers* zugewiesen und der Variablen *messageText* wird eine *GUIText*-Komponente zugewiesen, die einem *GameObject* mit dem Tag „Message“ gehört, das wir bereits oben erstellt haben. Abschließend wird in der Start-Methode wieder *UpdateView* aufgerufen, damit in der GUI die richtige Balkenlänge dargestellt wird.

**Listing 21.26** Start-Methode und Variablen von PlayerHealth

```
public AudioClip hurtClip;
public AudioClip deathClip;
public GUITexture healthGui;
private LifePointController lifePointController;
private float maxHealth;
private GUIText messageText;
private Rect guiRect;
private float guiMaxWidth;
void Start () {
    messageText = GameObject.FindGameObjectWithTag ("Message").
        GetComponent<GUIText>();
    lifePointController = GameObject.FindGameObjectWithTag("GameController").
        GetComponent<LifePointController>();
    guiRect = new Rect (healthGui.pixelInset);
    guiMaxWidth = guiRect.width;
    maxHealth = health;
    UpdateView();
}
```

Jetzt brauchen wir nur noch die beiden überschreibbaren Methoden *Damaging* und *Dying* zu programmieren, die in der Basisklasse zwar aufgerufen werden, aber leer sind.

Im Fall einer Verletzung soll ein *AudioClip* abgespielt werden, dessen *public*-Variable *hurtClip* bereits in Listing 21.26 definiert wurde. Hierbei wird der Clip der  *AudioSource*-Komponente des Spielers zugewiesen, über die alle Sprachsamples des Spielers abgespielt werden sollen. Dies muss zwar nicht unbedingt gemacht werden, wollen wir aber für ein einheitliches Hörerlebnis in diesem Fall machen.

Außerdem muss in der überschreibenden Methode *UpdateView* aufgerufen werden, damit die GUI aktualisiert wird.

**Listing 21.27** Damaging-Methode von PlayerHealth

```
public override void Damaging ()
{
    audio.clip = hurtClip;
    audio.Play ();
    UpdateView();
}
```

Und auch in der Dying-Methode wird der dazugehörige *AudioClip* (*deathClip*) der eigenen  *AudioSource* zugewiesen und abgespielt und die GUI aktualisiert.

Hinzu kommt jetzt aber noch das Aktualisieren der Lebenspunkte, die um 1 reduziert werden müssen. Sind nun noch Lebenspunkte vorhanden, wird das Level bzw. die Szene nach einer kurzen Wartepause neu gestartet. Sind alle Lebenspunkte aufgebraucht, soll „Game Over“ eingeblendet werden.

Für das Einblenden von „Game Over“ nutzen wir hier wieder die *GUIText*-Komponente, deren Objekt und Komponente wir bereits in der Start-Methode mithilfe von *FindGameObjectWithTag* der Variablen *messageText* zugewiesen haben.

**Listing 21.28** Dying-Methode von PlayerHealth

```
public override void Dying ()
{
    audio.clip = deathClip;
    audio.Play ();
    UpdateView();

    lifePointController.LifePoints -= 1;

    if(lifePointController.LifePoints > 0)
        Invoke("Restart",1);
    else
        messageText.text = "Game Over";
}
```

Wie Sie dem Listing 21.28 entnehmen können, wird für das Neustarten des Spiels eine Methode *Restart* mit einer Sekunde Verzögerung aufgerufen. Diese letzte Methode des Skriptes macht nichts anderes, als mit *LoadLevel* die aktuelle Szene neu zu starten.

**Listing 21.29** Restart-Methode von PlayerHealth

```
void Restart()
{
    Application.LoadLevel(Application.loadedLevel);
```

Haben Sie das *PlayerHealth*-Skript nun fertig, können Sie dieses ebenfalls Ihrem „Player“ anhängen. Der Variablen *hurtClip* weisen Sie den *AudioClip* „hit“ zu, als „Death Clip“ nehmen Sie die Datei „death“. *healthGui* weisen wir das bereits angelegte *GUITexture*-Objekt „HealthBar“ zu.

### 21.4.1.1 LifePointController.cs

Das Skript `LifePointController`, das Sie dem *GameObject* „Game Controller“ zufügen, sieht zusammengefasst wie im nächsten Listing aus. Das Skript fügen Sie dem „Game Controller“ zu.

**Listing 21.30** LifePointController.cs

```
using UnityEngine;
using System.Collections;

public class LifePointController : MonoBehaviour {
    public GUIText lpText;
    private int lifePoints = 0;

    public int LifePoints
    {
        get
        {
            return lifePoints;
        }
        set
        {
            lifePoints = value;
            if(lifePoints < 0)
                lifePoints = 0;
            UpdateView();
        }
    }

    void Awake()
    {
        LifePoints = PlayerPrefs.GetInt("LPs");
    }

    void UpdateView()
    {
        lpText.text = lifePoints.ToString();
    }

    void OnDestroy()
    {
        PlayerPrefs.SetInt("LPs", lifePoints);
    }
}
```

### 21.4.1.2 HealthController.cs

Die Basisklasse `HealthController`, von der auch die Klasse `PlayerHealth` erbt, sieht wie im folgenden Listing aus.

**Listing 21.31** HealthController.cs

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

```

public class HealthController : MonoBehaviour {
    public float health = 3;
    private bool isDead = false;

    void ApplyDamage(float damage)
    {
        health -= damage;

        if(health <= 0 && !isDead)
        {
            isDead = true;
            Dying();
        }
        else
        {
            Damaging();
        }
    }

    public virtual void Damaging()
    {
    }

    public virtual void Dying ()
    {
    }
}

```

#### 21.4.1.3 PlayerHealth.cs

Die eigentliche Verwaltung des Gesundheitszustandes vom Spieler wird vom PlayerHealth-Skript übernommen, das von der Klasse HealthController erbt. Fügen Sie dieses dem Player direkt zu. Den beiden AudioClip-Variablen hurtClip und deathClip weisen Sie dann die *AudioClips „hit“ und „death“* zu.

##### **Listing 21.32** PlayerHealth.cs

```

using UnityEngine;
using System.Collections;

public class PlayerHealth : HealthController {

    public AudioClip hurtClip;
    public AudioClip deathClip;
    public GUITexture healthGui;

    private LifePointController lifePointController;
    private float maxHealth;
    private GUIText messageText;
    private Rect guiRect;
    private float guiMaxWidth;
    void Start () {
        messageText = GameObject.FindGameObjectWithTag ("Message").
            GetComponent<GUIText> ();
        lifePointController = GameObject.
            FindGameObjectWithTag ("GameController").

```

```
GetComponent<LifePointController>();  
guiRect = new Rect (healthGui.pixelInset);  
guiMaxWidth = guiRect.width;  
maxHealth = health;  
UpdateView();  
}  
  
public override void Damaging ()  
{  
    audio.clip = hurtClip;  
    audio.Play ();  
    UpdateView();  
}  
  
public override void Dying ()  
{  
    audio.clip = deathClip;  
    audio.Play ();  
    UpdateView();  
  
    lifePointController.LifePoints -= 1;  
  
    if(lifePointController.LifePoints > 0)  
        Invoke("Restart",1);  
    else  
        messageText.text = "Game Over";  
}  
  
void Restart()  
{  
    Application.LoadLevel(Application.loadedLevel);  
}  
  
void UpdateView()  
{  
    if(healthGui != null) {  
        guiRect.width = guiMaxWidth * health/maxHealth;  
        healthGui.pixelInset = guiRect;  
    }  
}
```

## 21.4.2 Spielersteuerung

Es gibt unzählige Möglichkeiten, den Helden eines Spiels zu steuern. Um es dem Spieler aber einfach zu machen, ist es meistens sinnvoll, bekannte Mechanismen, die bereits in ähnlichen Spielen eingesetzt werden, zu nutzen. Aus diesem Grund möchte ich für unser Beispiel-Game die traditionelle *WASD-Tastensteuerung* einsetzen.

Da unsere Spielersteuerung keinen physikalischen Gesetzmäßigkeiten folgen muss, können wir die Steuerung mit einem *Character Controller* umsetzen. Die passende Komponente hatten wir ja bereits dem „Player“ zugefügt.

## PlayerController

Im Kapitel „Physik in Unity“ hatte ich Ihnen im Abschnitt „Einfacher First Person Controller“ demonstriert, wie eine Steuerung mit einem *Character Controller* aussehen könnte. Dort finden Sie auch eine ausführliche Beschreibung zur Entwicklung dieses Skriptes. Programmieren Sie dieses der Beschreibung entsprechend nach und fügen Sie das *PlayerController*-Skript dem „Player“ zu. Beachten Sie hierbei auch das Anlegen der zwei zusätzlichen virtuellen Achsen „Left“ und „Right“ (Taste **Q** für „Left“ und **E** für „Right“). Am Ende des Abschnitts „Einfacher First Person Controller“ finden Sie schließlich auch das komplette Skript.

Im Folgenden wollen wir dieses Skript namens „PlayerController“ noch um spielspezifische Funktionalitäten erweitern. Hierzu gehört auch das Unterbinden der Steuerung, sobald der Spieler das Game gewonnen hat oder dieser tot ist.

Als Erstes wollen wir nun zwei Variablen erstellen: die boolesche Variable `gameEnded` und die Variable `playerHealth` vom Typ `PlayerHealth`. Da `gameEnded` später von einem anderen Skript gesetzt werden soll, das die Erfüllung der Quest feststellt, muss diese Variable mit `public` deklariert werden. Da wir aber die Variable nicht im *Inspector* setzen wollen, können wir sie mit einem sogenannten *Attribut* versehen, das dafür sorgt, dass die Variable trotz der `public`-Deklaration nicht im *Inspector* zu sehen ist. Dieses *Attribut* lautet `HideInInspector` und wird in eckigen Klammern über der Variablen angegeben.

**Listing 21.33** Alle Variablen von PlayerController

```
public float moveSpeed = 5.0F;
public float rotationSpeed = 300.0F;
[HideInInspector]
public bool gameEnded = false;
private Vector3 moveDirection = Vector3.zero;
private CharacterController controller;
private PlayerHealth playerHealth;
private Quaternion destRotation;
```

In der Start-Methode wird nun neben den bereits erstellten Zuweisungen auch noch das `PlayerHealth`-Skript der dazugehörigen Variablen zugewiesen.

**Listing 21.34** Start-Methode von PlayerController

```
void Start() {
    playerHealth = GetComponent<PlayerHealth>();
    controller = GetComponent<CharacterController>();
    destRotation = transform.rotation;
}
```

Die einzige wirkliche Änderung zum Originalskript aus dem Kapitel „Physik in Unity“ ist nun das Abfragen der booleschen Variablen `gameEnded` und des aktuellen Gesundheitszustands, bevor die Steuerung ausgeführt wird. Ist aber die Variable `health` von `playerHealth` „0“ oder `gameEnded` TRUE, wird die Steuerung nicht ausgeführt und der `SimpleMove`-Methode wird der Wert `Vector3.zero` übergeben, was nichts anderes bedeutet, als dass der *Character Controller* stehen bleiben soll. Wie dies im Code aussieht, können Sie im kompletten Skript sehen, das Sie im folgenden Abschnitt „PlayerController.cs“ finden.

## Footsteps

Mit der bloßen Steuerung ist es aber meistens nicht getan, auch nicht in unserem Fall. Da unser Held auch Schrittgeräusche erzeugen soll, sobald sich dieser bewegt, erzeugen wir nun ein neues Skript mit dem Namen *Footsteps*. Dieses benötigt fünf Variablen: eine für den *AudioClip*, der wir den Klang eines einzelnen Schrittes zuweisen. Dann brauchen wir eine *float*-Variable, mit der wir das Intervall definieren, in dem die Schritte zu hören sind, sowie eine *float*-Variable für die Lautstärke der Schrittgeräusche. Außerdem brauchen wir noch eine *private*-Variable, in der wir die Referenz zum *Character Controller* in der *Start*-Methode speichern, sowie eine letzte *float*-Variable, die die vergangene Zeit seit dem letzten Schrittgeräusch speichert.

### Listing 21.35 Variablen von Footsteps

```
public AudioClip audioClip;
public float stepLength = 0.4F;
public float volume = 0.7F;
private CharacterController controller;
private float delay = 0;
```

Die Schrittgeräusche sollen nun immer dann abgespielt werden, wenn sich der Spieler auf dem Boden bewegt. Hierfür prüfen wir nun zuerst über *sqrMagnitude* mit einer gewissen Toleranz, ob sich der Spieler bewegt, und über *isGrounded* testen wir, ob dieser auch den Boden berührt.

### Listing 21.36 if-Bedingung zur Bewegungskontrolle des Spielers

```
if (controller.velocity.sqrMagnitude > 0.2F && controller.isGrounded)
{
    //...
}
```

Wenn dies der Fall ist, spielen wir nun im zeitlichen Abstand von *stepLength* den *AudioClip* mit *PlayClipAtPoint* ab. Die vergangene Zeit stellen wir hierbei mit der *float*-Variablen *delay* fest, deren Wert wir in jedem Frame um *deltaTime* erhöhen. Hat *delay* nun *stepLength* erreicht, wird der *AudioClip* abgespielt und *delay* wird wieder auf 0 gesetzt. Hierfür muss der Code natürlich in jedem Frame aufgerufen werden, weshalb wir das Vorgehen passenderweise in die *Update*-Methode legen.

### Listing 21.37 Schrittgeräusche in Update abspielen

```
void Update()
{
    if (controller.velocity.sqrMagnitude > 0.2F && controller.isGrounded)
    {
        if (delay >= stepLength)
        {
            AudioSource.PlayClipAtPoint(audioClip, transform.position, volume);
            delay = 0;
        }
        delay += Time.deltaTime;
    }
}
```

Das fertige *Footstep*-Skript fügen Sie nun Ihrem „Player“ zu und weisen der Variablen *audioClip* die Audiodatei „footstep“ zu. Beachten Sie hierbei, dass Sie bei dieser Datei in den *Import Settings* die Eigenschaft *3D Sound* deaktivieren. Dies ist wichtig, weil sich die durch *PlayClipAtPoint* temporär erzeugten *AudioSources* nicht mit dem Spieler mitbewegen, sondern an der Stelle bleiben, wo sie erzeugt wurden.

## Shooting

Zum Schluss wollen wir noch dafür sorgen, dass unser Held auch kämpfen kann. Als Waffe soll er Steine nutzen können, die er während des Spiels einsammeln und anschließend werfen kann.

Für diesen Mechanismus ist natürlich ein Zusammenspiel mit dem Inventarsystem wichtig, weshalb wir natürlich zunächst in unseren Variablendeclarationen eine Variable vom Typ *Inventory* erstellen. Weiter benötigen wir eine Referenz auf das *PlayerHealth*-Skript unseres Spielers, damit wir feststellen können, ob wir überhaupt in der Lage sind zu werfen, dann brauchen wir noch eine *GameObject*-Variable für unser *Prefab*.

Wenn Ihnen das Werfen von Steinen zu langweilig ist, können Sie dies auch durch andere 3D-Modelle wie z.B. durch Messer oder Pfeile eines Bogens oder einer Armbrust ersetzen. Da das Prinzip gleich ist, spielt das dann keine wirkliche Rolle mehr. Deshalb werde ich das Skript aber auch ganz allgemein „Shooting“ nennen und die *Prefab*-Variable *projectile* nennen.

**Listing 21.38** Variablendeclaration des Shooting-Skriptes

```
public GameObject projectile;
private PlayerHealth playerHealth;
private Inventory inventory;
private GUIText messageText;
private string info = "Zum Angreifen brauchst Du etwas zum Werfen";
private bool messageShown = false;
```

Wie Sie dem Listing 21.38 entnehmen können, benötigen wir zu den bereits genannten Variablen noch drei weitere, auf die ich aber noch zu sprechen komme. Wichtig ist hierbei zunächst einmal, dass die Variable *messageText* in der *Start*-Methode auf die *GUIText*-Komponente des *GameObjects* mit dem Tag „Message“ verweist, das wir bereits im *PlayerHealth*-Skript genutzt haben.

Wenn unser Held nun einen Stein wirft, stellt sich jetzt die Frage, wo denn nun das *Prefab* eigentlich erzeugt werden soll. Zum Definieren solcher Punkte gibt es sogenannte *Spawn-Points*. Ein *Spawn-Point* ist meist nur ein *Empty Object*, das mithilfe seiner *Transform*-Komponente die Position bestimmt. In unserem Fall wird der *Spawn-Point* ein Kind-Objekt unseres Players sein, da sich dieser logischerweise immer mit dem Spieler mitbewegt (siehe Einleitung dieses Abschnitts).

Das Besondere dieses Skriptes ist nun, dass es nicht dem „Player“ zugefügt, sondern direkt dem *GameObject* „*SpawnPoint*“ angehängt wird. Das hat den Charme, dass wir dem Skript nicht extra sagen müssen, wo denn die *Prefabs* erzeugt werden sollen. Außerdem nutzen wir gleich die Drehung dieses *GameObjects*, um die Richtung des Wurfes vom *Spawn-Point* zu übernehmen.

Allerdings müssen wir dies auch beim Zugriff auf andere Komponenten bedenken. Da wir in der Start-Methode auf das *Inventory* und *PlayerHealth* verweisen, müssen wir deshalb nun beim Letzteren auf das Eltern-Objekt zugreifen. Dies machen wir mit `transform.parent`.

#### **Listing 21.39** Start-Methode vom Shooting-Skript

```
void Start () {
    playerHealth = transform.parent.GetComponent<PlayerHealth>();
    inventory = GameObject.FindGameObjectWithTag("Inventory").
        GetComponent<Inventory>();
    messageText = GameObject.FindGameObjectWithTag ("Message").
        GetComponent<GUIText>();
}
```

Das Werfen bzw. Schießen wollen wir nun mit einer Methode `Shoot` auslösen. Diese instantiiert einfach nur eine *Prefab*-Instanz von `projectile` an der eigenen Position mit der gleichen Drehung wie der „*SpawnPoint*“. Außerdem soll ein Wurfgeräusch zu hören sein, das wir in dem gleichen Moment ebenfalls abspielen wollen. Hierfür haben wir zu Beginn dem „*SpawnPoint*“ bereits eine eigene  *AudioSource* zugefügt, dem auch ein passender Sound zugewiesen wurde.

#### **Listing 21.40** Shoot-Methode vom Shooting-Skript

```
void Shoot()
{
    Instantiate(projectile,transform.position,transform.rotation);
    audio.Play ();
}
```

Als Letztes müssen wir nun noch `Shoot` auslösen. Dies machen wir in der `Update`-Methode, wo wir den virtuellen Button „*Fire2*“ nutzen werden. Hierfür würde ich in den *Input Settings* die Taste von *Positive Button* vielleicht von „left alt“ auf „space“ ändern. Aber das können Sie je nach Vorliebe selber bestimmen, welche Taste Sie da zum Werfen nutzen wollen.

Für jeden Wurf wird schließlich aus dem Inventar ein Item mit dem Tag „*Stone*“ entfernt. Wenn keines mehr vorhanden ist, kann auch nicht geworfen werden.

Ein wichtiger Punkt, der spätestens dann auffallen wird, wenn der erste Tester das Game spielen wird, ist ein Hinweis darauf, dass der Spieler erst einmal die Gegenstände einsammeln muss, die er werfen möchte. Ansonsten drückt dieser die Angriffstaste und wundert sich, dass nichts passiert.

Hierfür wollen wir einen kleinen Infotext einblenden, der erscheinen soll, wenn der Spieler das erste Mal versucht, ohne einen im Inventar vorhandenen Stein zu attackieren. Diese wollen wir in einer Coroutine zeigen, die den Text für zwei Sekunden anzeigt. Dabei merken wir uns in der Variablen `messageShown`, wenn dieser Hinweise gezeigt wurde. In der `Update`-Methode überprüfen wir, sobald kein Stein im Inventar vorhanden ist, wie der Status dieser Variablen ist, und starten dann dementsprechend die Coroutine.

**Listing 21.41** Update-Methode vom Shooting-Skript

```
void Update ()
{
    if(playerHealth.health > 0)
    {
        if (Input.GetButtonDown ("Fire2"))
        {
            if (inventory.RemoveItem("Stone"))
                Shoot();
            else if (!messageShown)
                StartCoroutine>ShowInfoText();
        }
    }
}
IEnumerator ShowInfoText()
{
    messageText.text = info;
    messageShown = true;
    yield return new WaitForSeconds(2);
    messageText.text = "";
}
```

Dieses Skript wird nun, wie bereits gesagt, dem „SpawnPoint“ angehängt. Im Abschnitt „Wurfstein entwickeln“ werden wir noch das *Prefab* erstellen, das wir dem Skript-Parameter *projectile* noch zuweisen müssen.

**21.4.2.1 PlayerController.cs**

Im Folgenden können Sie das komplette *PlayerController*-Skript sehen. Beachten Sie bitte hierbei, dass Sie für das Nutzen dieses Skriptes noch die virtuellen Buttons „Left“ und „Right“ anlegen müssen. Diese definieren Sie in den *Input Settings* und belegen diese am besten mit den Tasten **[Q]** (für „Left“) und **[E]** (für „Right“). Mehr Infos zum Erstellen von virtuellen Buttons und Achsen erhalten Sie im Kapitel „Maus, Tastatur, Touch“.

**Listing 21.42** PlayerController.cs

```
using UnityEngine;
using System.Collections;

public class PlayerController : MonoBehaviour {
    public float moveSpeed = 5.0F;
    public float rotationSpeed = 300.0F;
    [HideInInspector]
    public bool gameEnded = false;
    private Vector3 moveDirection = Vector3.zero;
    private CharacterController controller;
    private PlayerHealth playerHealth;
    private Quaternion destRotation;

    void Start() {
        playerHealth = GetComponent<PlayerHealth>();
        controller = GetComponent<CharacterController>();
        destRotation = transform.rotation;
    }
}
```

```

void Update() {
    if(playerHealth.health > 0 && !gameEnded)
    {
        moveDirection = new Vector3(Input.GetAxis("Horizontal"),
            0, Input.GetAxis("Vertical"));
        moveDirection = transform.TransformDirection(moveDirection);
        moveDirection = moveDirection * moveSpeed;
        controller.SimpleMove(moveDirection);

        if(Input.GetButtonDown("Left"))
            destRotation.eulerAngles = destRotation.eulerAngles -
                new Vector3(0,90,0);

        if(Input.GetButtonDown("Right"))
            destRotation.eulerAngles = destRotation.eulerAngles +
                new Vector3(0,90,0);

        float step = rotationSpeed * Time.deltaTime;
        transform.rotation = Quaternion.RotateTowards(
            transform.rotation, destRotation, step);
    }
    else
    {
        controller.SimpleMove(Vector3.zero);
    }
}

```

### 21.4.2.2 Footsteps.cs

Das für das Trittgeräusch zuständige Skript *Footsteps* können Sie dem Listing 21.43 entnehmen. Bitte fügen Sie dieses Skript ebenfalls dem „Player“ zu und weisen Sie der Variablen *audioClip* die Sounddatei „footstep“ zu. Da sich die durch das Skript temporär erzeugten Audioquellen nicht mit dem Spieler mitbewegen, sollten Sie bei dieser Datei in den *Import Settings* die Eigenschaft *3D Sound* deaktivieren.

Über *stepLength* definieren Sie das Intervall, in dem die Schritte zu hören sind. Je niedriger Sie diesen Wert setzen, desto häufiger sind die Schritte zu hören.

#### Listing 21.43 Footsteps.cs

```

using UnityEngine;
using System.Collections;

public class Footsteps : MonoBehaviour {
    public AudioClip audioClip;
    public float stepLength = 0.4F;
    public float volume = 0.7F;
    private CharacterController controller;
    private float delay = 0;
    void Start () {
        controller = GetComponent<CharacterController>();
    }

    void Update()
    {

```

```

        if (controller.velocity.sqrMagnitude > 0.2F && controller.isGrounded)
    {
        if (delay >= stepLength)
        {
            AudioSource.PlayClipAtPoint(audioClip, transform.position, volume);
            delay = 0;
        }
        delay += Time.deltaTime;
    }
}

```

### 21.4.2.3 Shooting.cs

Fügen Sie das folgende Skript dem „SpawnPoint“-Objekt zu. Später weisen Sie der Variablen `projectile` das Wurfstein-*Prefab* zu, das wir noch im Abschnitt „Wurfstein entwickeln“ erstellen werden. Je nach Vorliebe können Sie für das Werfen in den *Input Settings* noch eine andere Taste festlegen. Standardmäßig ist hier die Taste „left alt“ als *Positive Button* definiert. Sie könnten dies auf „space“ ändern und damit die Leertaste `[Space]` zum Werfen nutzen.

**Listing 21.44** Shooting.cs

```

using UnityEngine;
using System.Collections;

public class Shooting : MonoBehaviour {

    public GameObject projectile;
    private PlayerHealth playerHealth;
    private Inventory inventory;
    private GUIText messageText;
    private string info = "Zum Angreifen brauchst Du etwas zum Werfen";
    private bool messageShown = false;

    void Start () {
        playerHealth = transform.parent.GetComponent<PlayerHealth>();
        inventory = GameObject.FindGameObjectWithTag("Inventory").
            GetComponent<Inventory>();
        messageText = GameObject.FindGameObjectWithTag ("Message").
            GetComponent<GUIText>();
    }

    void Update () {
        if(playerHealth.health > 0)
        {
            if (Input.GetButtonDown ("Fire2"))
            {
                if (inventory.RemoveItem("Stone"))
                    Shoot();
                else if (!messageShown)
                    StartCoroutine>ShowInfoText());
            }
        }
    }
}

```

```

void Shoot()
{
    GameObject go = (GameObject) Instantiate(projectile,
        transform.position, transform.rotation);
    audio.Play ();
}

IEnumerator ShowInfoText()
{
    messageText.text = info;
    messageShown = true;
    yield return new WaitForSeconds(2);
    messageText.text = "";
}
}

```

### 21.4.3 Wurfstein entwickeln

Wir hatten ja bereits beim Erstellen der Inventar-Items ein *Prefab* für die Wurfsteine erstellt. Allerdings dient dieses lediglich dem Einsammeln der Wurfsteine. Als Waffe kann man dieses aber nicht nutzen, da das Objekt hierfür einige andere Komponenten und Einstellungen benötigt.

Aus diesem Grund erstellen wir als Nächstes ein weiteres Stein-*Prefab*, das wir dann „Throwing Stone“ nennen werden.

1. Ziehen Sie hierfür das Modell „stone\_01“ ein erneutes Mal in die Szene.
2. Benennen Sie das *GameObject* in der Szene in „Throwing Stone“ um.
3. Fügen Sie diesem einen *Sphere Collider* zu und aktivieren Sie *Is Trigger*.
4. Da die geworfenen Steine recht klein, aber auch schnell sein werden, vergrößern wir nun leicht den *Radius* des *Colliders* z.B. auf 0.12, um die Kollisionserkennung zu erleichtern.
5. Fügen Sie nun dem *GameObject* ein *Rigidbody* zu und aktivieren Sie *Is Kinematic*. Deaktivieren Sie stattdessen *Use Gravity*. Außerdem setzen Sie ebenfalls für eine bessere Kollisionserkennung den Parameter *Collision Detection* auf „Continue Dynamic“.
6. Ziehen Sie das *GameObject* „Throwing Stone“ in den „Prefab“-Ordner, um daraus ein *Prefab* zu erstellen.
7. Dieses *Prefab* weisen Sie nun dem *projectile*-Parameter des *Shooting*-Skriptes vom „SpawnPoint“ zu.

Wie Sie vielleicht schon bemerkt haben, wird der Wurfstein nun durch das *Shooting*-Skript erzeugt, aber er wird nicht bewegt. Die Fähigkeit, dass der Stein auch in die richtige Richtung fliegt, wollen wir deshalb jetzt in einem gesonderten Skript programmieren.

Dieses Skript wird sich aber nicht nur um die Bewegung kümmern, sondern auch den Schaden verursachen und auch den Stein selber zerstören, wenn dieser mit einem anderen Objekt kollidiert. Kurz gesagt: Dieses Skript wird sich um das gesamte Verhalten des Steins kümmern, weshalb wir es auch ganz einfach „StoneBehaviour“ nennen. Erzeugen Sie deshalb nun ein neues Skript „StoneBehaviour“ und fügen Sie dieses gleich dem *Prefab* „Thro-

wingStone“ zu. Selektieren Sie hierfür das *Prefab* im *Project Browser* und fügen Sie das Skript über den **Add Component**-Button im *Inspector* dem *Prefab* zu.

Um die beste Kontrolle über den Wurf bzw. die Flugbahn des Steins zu erhalten, werden wir den Stein über die *Translate*-Methode der *Transform*-Komponente bewegen. Bei der Erstellung des Wurfstein-*Prefabs* haben wir deshalb bereits den *Is Kinematic*-Parameter vom *Rigidbody* aktiviert. Die Geschwindigkeit des Steins steuern wir hierbei über eine Variable namens *speed*.

**Listing 21.45** Erzeugen der Bewegung des Wurfsteins

```
void Update()
{
    transform.Translate(transform.forward * speed * Time.deltaTime, Space.World);
}
```

Um den Spielspaß beim Kämpfen zu erhalten, sollte ein Zielen mit den Steinen relativ leicht sein. Hierfür ist es natürlich am einfachsten, wenn der *Spawn-Point* recht weit in der Mitte des Spielers liegt. Damit der Wurfstein aber nun nicht schon mit dem *Collider* des *Character Controllers* kollidiert, nutzen wir beim *Collider* des Steins den Parameter *Is Trigger*.

Das bedeutet aber auch, dass wir das Auswerten und Zufügen von Schaden in einer *OnTrigger*-Methode, vorzugshalber in der Methode *OnTriggerEnter*, ausführen müssen. Allerdings wird diese Methode auch ausgeführt, wenn das Objekt mit einem anderen *Trigger-Collider* kollidiert, z.B. mit dem unseres zukünftigen Questgebers (mehr dazu im Abschnitt „Questgeber erstellen“).

Deshalb wollen wir in diesem Fall mit einer *Layermask* arbeiten, über die wir festlegen, bei welchen *Layern* der folgende Code ausgeführt werden soll und bei welchen nicht. Hierfür erzeugen wir zunächst über **Edit/Project Settings/Tags and Layers** einen neuen *Layer* mit dem Namen „IgnoreStoneDetection“.

Nun deklarieren wir eine öffentliche *Layermask*-Variable mit dem Namen *acceptableLayers* genauso wie die bereits angesprochene *speed*-Variable und eine *damage*-Variable, die die Höhe des Schadens festlegt.

Außerdem soll noch ein kleiner Partikeleffekt erzeugt werden, wenn der Stein durch eine Kollision mit einem anderen *GameObject* kaputt geht. Hierfür benötigen wir noch eine *GameObject*-Variable mit dem Namen *destroyGo*. Den Partikeleffekt selber werden wir gleich im Anschluss erstellen.

**Listing 21.46** Variablen von StoneBehaviour

```
public LayerMask acceptableLayers;
public GameObject destroyedGo;
public float damage = 1;
public float speed = 10;
```

Beachten Sie nun, dass Sie im *Inspector* vom „ThrowingStone“ bei *acceptableLayers* der neue *Layer* „IgnoreStoneDetection“ deaktiviert ist und die restlichen angehakt sind.

In der Methode *OnTriggerEnter* werten wir nun im ersten Schritt den Layer aus, mit dem der Stein gerade kollidiert (siehe Abschnitt „Mit Layermasken arbeiten“ im Kapitel „Physik in Unity“). Ist der *Layer* des anderen Objektes in *acceptableLayers* vorhanden, so wird danach geprüft, ob das Objekt den *Tag* „Enemy“ besitzt. Wenn dies ebenfalls der Fall ist,

wird mit `SendMessage` die Methode `ApplyDamage` aufgerufen und die Variable `damage` übergeben.

Aber unabhängig davon, ob es nun ein „Enemy“ ist oder nicht, wenn das Objekt einen *Layer* besitzt, der sich in `acceptableLayers` befindet, wird zum Schluss das eigene *GameObject* zerstört und eine Instanz von `destroyGo` erzeugt. Allerdings wird auch dieses nach einer Sekunde gleich wieder zerstört. Hierfür wird `Instantiate` gleich im Rahmen eines zeitverzögerten `Destroy`-Befehls aufgerufen.

#### **Listing 21.47** OnTriggerEnter vom StoneBehaviour-Skript

```
void OnTriggerEnter(Collider other)
{
    if ((acceptableLayers.value & 1 << other.gameObject.layer) ==
        1 << other.gameObject.layer)
    {
        if (other.gameObject.CompareTag("Enemy"))
        {
            other.gameObject.SendMessage ("ApplyDamage",
                damage,SendMessageOptions.DontRequireReceiver);
        }
        Destroy(gameObject);
        Destroy(Instantiate(destroyedGo,transform.position,Quaternion.identity),1);
    }
}
```

Damit der Wurfstein nun nicht gleich beim Erzeugen mit *Collider* des *CharacterControllers* kollidiert, setzen wir gleich zu Beginn erst einmal den *Layer* des „Player“ auf „IgnoreStone Detection“. Bestätigen Sie hierbei die darauffolgende Abfrage, dass Sie das Setzen des *Layers* auch auf alle Kind-Objekte übernehmen wollen.

#### **21.4.3.1 StoneBehaviour.cs**

Das komplette *StoneBehaviour*-Skript, das Sie dem Wurfstein-Prefab „ThrowingStone“ zufügen, finden Sie im folgenden Listing.

#### **Listing 21.48** StoneBehaviour.cs

```
using UnityEngine;
using System.Collections;

public class StoneBehaviour : MonoBehaviour
{
    public LayerMask acceptableLayers;
    public GameObject destroyedGo;
    public float damage = 1;
    public float speed = 10;

    void Update()
    {
        transform.Translate(transform.forward * speed * Time.deltaTime,Space.World);
    }

    void OnTriggerEnter(Collider other)
    {
        //Wandelt per bitweiser Verschiebung einen Integer in eine Bitzahl um.
```

```

//Debug.Log(1 << other.gameObject.layer);
//Das Ergebnis des bitweisen Vergleichs (&),
//zwischen Layer und Maske, wird mit dem Layer als Bitzahl verglichen.
if ((acceptableLayers.value & 1 << other.gameObject.layer) ==
    1 << other.gameObject.layer)
{
    if (other.gameObject.CompareTag("Enemy"))
    {
        other.gameObject.SendMessage ("ApplyDamage",
            damage, SendMessageOptions.DontRequireReceiver);
    }
    Destroy(gameObject);
    Destroy(Instantiate(destroyedGo, transform.position, Quaternion.identity),1);
}
}
}

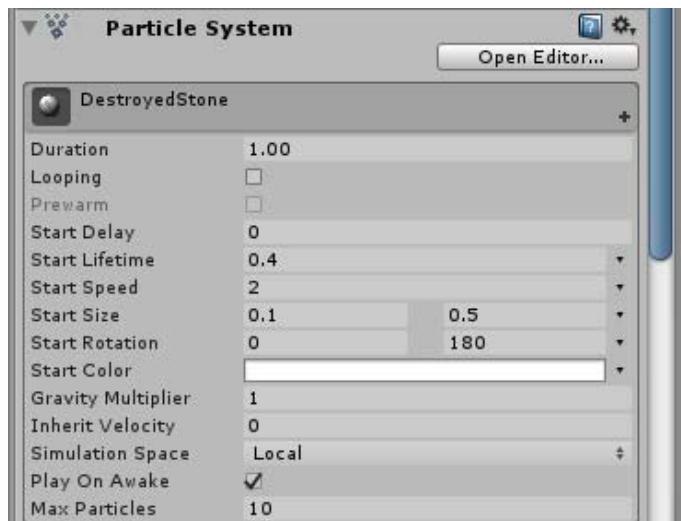
```

### 21.4.3.2 Zerberstenden Stein konfigurieren

Wenn ein Wurfstein gegen eine Wand oder ein anderes Objekt fliegt, soll dieser zerbersten. Hierfür wollen wir einen kleinen Partikeleffekt erstellen, der nicht herkömmliche Partikel erzeugt, sondern dreidimensionale Partikel in Form des Stein-Mesh.

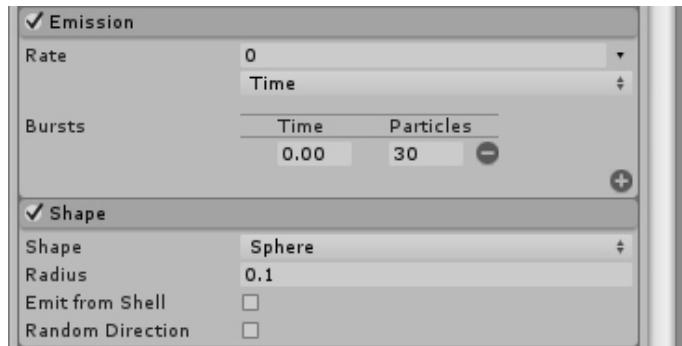
Erstellen Sie hierfür ein neues Partikelsystem und nennen Sie dieses „DestroyedStone“. Danach ziehen Sie es in den „Prefab“-Ordner, um daraus ein *Prefab* zu erstellen. Danach konfigurieren wir die einzelnen Module des Partikelsystems.

Zunächst parametrisieren wir das *Default-Modul* mit folgenden Einstellungen. Sowohl *Start Size* als auch *Start Rotation* sind hierbei auf *Random Between Two Constants* gestellt.



**Bild 21.16**  
Default-Modul des  
zerberstenden Steins

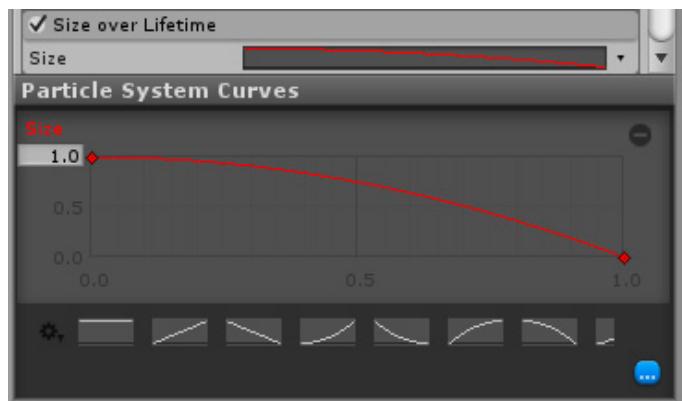
Danach folgen das *Emission-Modul* und das *Shape-Modul*. Da der Effekt nur einmal erscheinen soll, haben wir bereits oben *Looping* deaktiviert. Im *Emission-Modul* definieren wir nun einen einzelnen *Burst*. Durch den *Max Particles*-Wert von 10 im *Default-Modul* spielt es hier keine Rolle, ob wir den *Burst* nun auf 10, 30 oder 100 legen, es werden nur bis zu 10

**Bild 21.17**

Emission-Modul und Shape-Modul des zerberstenden Steins

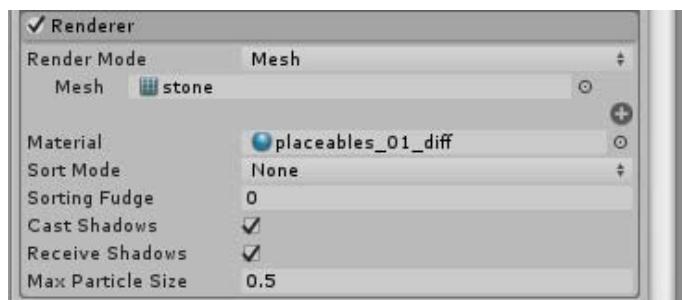
erzeugt. Als Form legen wir schließlich *Sphere* mit einem *Radius* von 0.1 fest, da die Partikel aus einem Punkt in alle Richtungen fliegen sollen.

Danach legen wir mithilfe vom *Size over Lifetime Modul* den Größenverlauf eines jeden Partikels fest. Hier sollen die Partikel zwar lange zu sehen sein, aber dann auch fließend verschwinden. Hierfür nutzen wir eine der Standardkurven, die Unity zur Verfügung stellt.

**Bild 21.18**

Size over Lifetime-Modul des zerberstenden Steins

Als Letztes folgt nun das *Renderer-Modul*, wo wir den *Render Mode* auf *Mesh* stellen. Nun wählen wir unter *Mesh* das *Mesh* des Stein-Modells aus („stone“). Wichtig ist nun noch, dass Sie auch das dazugehörige Material auswählen.

**Bild 21.19**

Renderer-Modul des zerberstenden Steins

Und schon ist der Partikeleffekt fertig. Jetzt brauchen Sie nur noch einmal mit **Apply** im *Inspector* des Partikelsystems die vorgenommenen Einstellungen auf das *Prefab* zu übertragen.

Abschließend soll der Stein nicht nur optisch zerbersten, sondern auch akustisch dies wiedergeben. Hierfür fügen Sie dem *GameObject* nun eine eigene  *AudioSource* zu, wo Sie *Play On Awake* aktivieren und *Loop* deaktivieren. Dieser weisen Sie nun den  *AudioClip* „stoneCrash“ zu.

Um das Partikelsystem-*GameObject* nun auch zu nutzen, weisen Sie dieses *Prefab* der  *destroyedGo*-Variablen vom *StoneBehaviour*-Skript des „ThrowingStone“-*Prefabs* zu.

## ■ 21.5 Quest erstellen

In diesem Abschnitt geht es um die eigentliche Aufgabe, die der Spieler lösen muss. In unserem Spiel soll unser Spieler einen Wasserbehälter finden, mit dem er heruntertropfendes Wasser auffangen kann. Um diese Aufgabe, auch Quest genannt, zu lösen, muss er noch Sub-Quests bewältigen und Gegner bekämpfen.

### 21.5.1 Erfahrungspunkte verwalten

Für das Überwältigen eines Gegners und das Lösen der Quest soll der Spieler am Ende auch Erfahrungspunkte bekommen. Deshalb wollen wir zuerst einmal ein kleines Skript namens *EPController* erstellen, das diese Erfahrungspunkte verwaltet und in der GUI anzeigt. Dieses *EPController*-Skript weisen Sie dem „Game Controller“ zu, der bereits den *LifePointController* besitzt.

Für die Funktionalitäten benötigen wir insgesamt zwei Variablen: eine Variable, die den EP-Wert speichert, und eine andere, die einen Verweis auf ein GUI-Objekt besitzt, um dort den Wert anzuzeigen.

**Listing 21.49** Variablen von *EPController*

```
public GUIText epText;
private int eps = 0;
```

Da uns für die GUI noch ein Objekt fehlt, wollen wir dieses nun als Nächstes erstellen.

1. Erzeugen Sie ein neues *GUIText*-Objekt und nennen Sie dieses „EpText“.
2. Fügen Sie „EpText“ dem bereits erstellten Container-Objekt „GUI“ als Kind-Objekt zu.
3. Positionieren Sie dieses auf (0,1,0) und einen *Pixel Offset* (70,-49).
4. Weisen Sie dieses Objekt der Skriptvariablen *epText* von *EPController* zu.

Für die eigentliche Logik brauchen wir wieder eine *UpdateView*-Methode, die das Anzeigen übernimmt, sowie eine Methode *AddPoints*, mit der wir von außen dem Skript neue EPs übergeben können. Die *UpdateView*-Methode wird zudem wieder in der *Start*-Methode das erste Mal aufgerufen, um der GUI einen Initialwert zuzuweisen.

**Listing 21.50** Methoden vom EPController

```
void Start()
{
    UpdateView();
}

public void AddPoints(int points)
{
    eps += points;
    UpdateView();
}

void UpdateView()
{
    epText.text = "EP: " + eps.ToString ();
}
```

Das war auch schon der komplette Inhalt von *EPController*. Jetzt brauchen Sie das Skript nur noch dem „Game Controller“ hinzuzufügen und das *GUIText*-Objekt „EpText“ der *epText*-Variablen zuzuweisen.

**21.5.1.1 EPController.cs**

Im folgenden Listing sehen Sie das fertige *EPController*-Skript, das die Erfahrungspunkte des Spielers verwaltet. Fügen Sie das fertige Skript dem „Game Controller“ zu.

**Listing 21.51** EPController.cs

```
using UnityEngine;
using System.Collections;

public class EPController : MonoBehaviour {
    public GUIText epText;
    private int eps = 0;

    void Start()
    {
        UpdateView();
    }

    public void AddPoints(int points)
    {
        eps += points;
        UpdateView();
    }

    void UpdateView()
    {
        epText.text = "EP: " + eps.ToString ();
    }
}
```

## 21.5.2 Questgeber erstellen

Kommen wir nun zur eigentlichen Quest, die ich einfach mal Water-Quest nennen will. Diese soll, wie bereits eingangs erwähnt, darin bestehen, einen Gegenstand zu finden, um Wasser aufzufangen.

Im Endeffekt spielt es überhaupt keine Rolle, welchen Gegenstand er suchen soll. Die Aufgabe könnte genauso gut darin bestehen, eine Zauberspruchrolle, ein magisches Schwert oder ein Goldstück zu suchen. Das Prinzip ist immer das gleiche: Sobald der Spieler in einen *Trigger-Collider* eintritt, wird überprüft, ob dieser den gesuchten Gegenstand besitzt. Hat er diesen in seinem Inventarsystem, wird er dort entfernt und die Quest wird als „Erfüllt“ gekennzeichnet. Hat er diesen Gegenstand nicht, wird der Spieler über seine Aufgabe informiert.



### Alternative Questgeber-Texte

Texte von Questgebern werden häufig so aufgebaut, dass bei mehrmaligem Aufsuchen des Questgebers unterschiedliche Texte erscheinen. Hierfür brauchen Sie einfach nur eine boolesche Variable, die auf TRUE gesetzt wird, sobald der Spieler seine Aufgabe erfährt. Ist diese Variable nun beim Betreten des Trigger-Colliders TRUE, bedeutet dies, dass der Spieler bereits schon einmal da war, und es kann ein anderer Text eingeblendet werden.

So ein *Trigger-Collider* wird meistens bei einer Person, einem Gegenstand oder an einem Ort positioniert, der zu der jeweiligen Aufgabe thematisch passt. Zusammen wird das Gesamtkonstrukt auch als „Questgeber“ bezeichnet, da diese Person/Gegenstand/Ort dem Helden diese Aufgabe mitteilt.

### Questgeber-Konfiguration

Als Questgeber möchte ich das Partikelsystem „Waterdrops“ nutzen, das ich bereits im Abschnitt 11.10, „Wassertropfen erstellen“, des Kapitels „Partikeleffekte mit Shuriken“ vorgestellt habe.

Damit der Spieler diesem Partikelsystem auch tatsächlich Aufmerksamkeit schenkt, fügen wir diesem auch gleich noch das Skript *WaterdropSound* zu, das wir ebenfalls in „Wassertropfen erstellen“ entwickelt haben. Als *AudioClip* nutzen Sie die Datei „waterdrop“. Zusätzlich können Sie auch die Menge der Wassertropfen erhöhen, die durch das Partikelsystem erzeugt werden. Setzen Sie hierfür den *Rate*-Wert im *Emission-Modul* leicht höher, z. B. auf 0.8.

Positionieren Sie dieses Partikelsystem in einer Ecke des Dungeons an der Decke. Am besten platzieren Sie es an einer Stelle, die der Spieler gleich zu Anfang sieht, wenn das Spiel startet. Hierdurch wird der Spieler sofort zu der Quest geführt und erfährt dort auch gleich, was er zu tun hat.

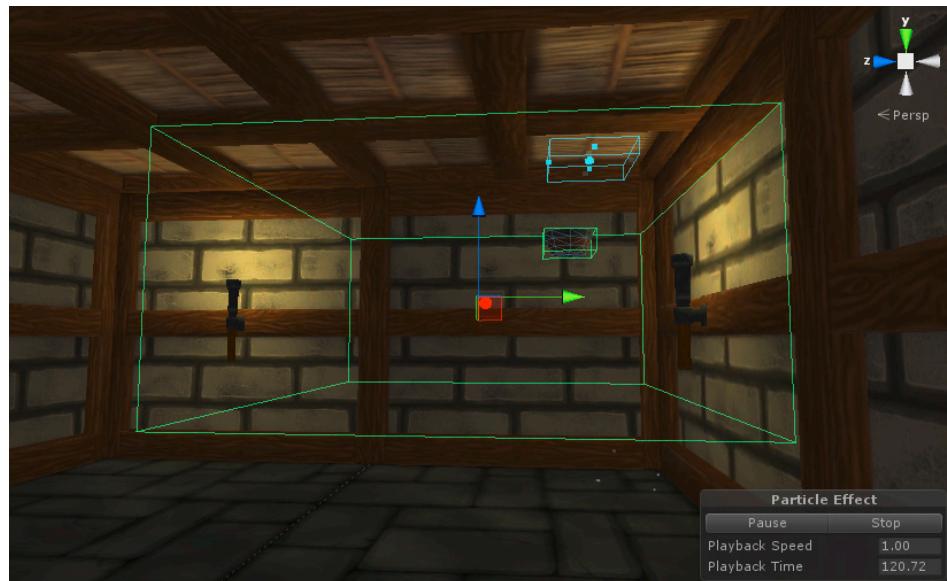
Unter diesem Partikelsystem platzieren wir nun unseren *Trigger-Collider* mit dem folgenden Quest-Skript, das wir noch entwickeln werden.

1. Erzeugen Sie hierfür einen Cube, den Sie „QuestTrigger“ nennen.
2. Skalieren Sie den Cube auf (2,1,2) und positionieren Sie ihn direkt unter dem Wassertropfen-Partikelsystem auf Spielerhöhe.

3. Legen Sie den *Layer* von „QuestTrigger“ auf „IgnoreStoneDetection“.
4. Deaktivieren Sie den *Mesh Renderer* des Cubes.
5. Aktivieren Sie den Parameter *Is Trigger* vom *Box Collider*
6. Erzeugen Sie nun ein neues Skript mit dem Namen „WaterQuest“ und fügen Sie dieses ebenfalls dem Cube zu.

Da der Spieler einen Wasserbehälter suchen und hierherbringen soll, wollen wir das erfolgreiche Abliefern des Behälters durch das Erscheinen und Herunterfallen eines Behälters symbolisieren.

Fügen Sie deshalb eine Instanz des „BucketEmpty“-*Prefabs* der Szene hinzu und platzieren Sie dieses direkt unter dem Wassertropfen-Partikelsystem in Höhe der Kamera.



**Bild 21.20** Questgeber mit Partikelsystem, Trigger und Wasserbehälter

## WaterQuest

Für die eigentliche Quest benötigen wir nun das bereits erstellte Skript *WaterQuest* mit den folgenden Variablen, die ich Ihnen gleich noch näher erläutern werde.

### Listing 21.52 Variablen des WaterQuest-Skriptes

```
public int eps = 20;
public GameObject finalBucket;
private GUIText messageText;
private string questMessage = "Suche einen Behälter, " +
    "um dieses Wasser zu trinken.";
private string questMessage2 = "Du brauchst einen leeren Behälter.";
private string questEndedMessage = "<size=20>Gratulation!</size>\n" +
    "Du hast die Aufgabe gelöst.";
private bool gotQuest = false;
```

```
private GameObject player;
private Inventory inventory;
private PlayerController playerController;
private EPController epController;
```

Die Variable `eps` dient später dem Zufügen von Erfahrungspunkten, wenn die Quest gelöst wurde. Der *GameObject*-Variablen `finalBucket` weisen wir die „BucketEmpty“-Instanz zu, die wir bereits eben der Szene zugefügt haben.

Mit der booleschen Variablen `gotQuest` merken wir uns, wenn der Spieler die Quest erhalten hat, um dann beim zweiten Mal den Alternativtext `questMessage2` anzuzeigen. Die `string`-Variablen `questEndedMessage`, `questMessage` und `questMessage2` dienen dabei als Speicher für die anzuzeigenden Texte. Der Befehl „\n“ erzeugt hierbei einen Zeilenumbruch innerhalb des *Strings*.

Beachten Sie außerdem die HTML-Tags im Inhalt von `questEndedMessage`, mit denen wir die Größe eines einzelnen Wortes hervorheben wollen. Unity unterstützt hier einige Format-Tags von HTML, die Sie hervorragend zum Hervorheben bestimmter Begriffe innerhalb eines *Strings* nutzen können.

Als Nächstes folgt nun die `Start`-Methode, in der wir die restlichen Variablen auf die entsprechenden Komponenten verweisen lassen. Außerdem deaktivieren wir das *GameObject* `finalBucket`, damit wir es später bei Erfolg wieder aktivieren können. In diesem Fall erscheint dann das Objekt und es fällt, dank der bereits zugewiesenen *Rigidbody*-Komponente, auf den Boden.

#### **Listing 21.53** Start-Methode von WaterQuest

```
void Start () {
    finalBucket.SetActive(false);
    player = GameObject.FindGameObjectWithTag ("Player");
    playerController = player.GetComponent<PlayerController>();
    inventory = GameObject.FindGameObjectWithTag("Inventory").
        GetComponent<Inventory>();
    messageText = GameObject.FindGameObjectWithTag ("Message").
        GetComponent<GUIText>();
    epController = GameObject.FindGameObjectWithTag("GameController").
        GetComponent<EPController>();
}
```

Beachten Sie hier wieder die Variable `messageText`, die wieder auf das *GUIText-GameObject* mit dem Tag „Message“ verweist. Hier nutzen wir also das gleiche Objekt, das wir bereits schon im *PlayerHealth*-Skript zum Anzeigen des „Game Over“-Textes genutzt haben.

Als Nächstes wollen wir nun den Kern dieses Questgebers programmieren: die `OnTriggerEnter`-Methode. Diese wird ausgeführt, sobald ein anderes *GameObject* mit einem *Rigidbody* und einem *Collider* in diesen *Trigger-Collider* hineintritt.

Hierfür kontrollieren wir zunächst, ob es sich bei dem Objekt um den Spieler handelt. Ist dies der Fall, versuchen wir ein Item mit der ID „Bucket“ aus dem Inventarsystem zu holen. Bekommen wir nun eines zurück, aktivieren wir `finalBucket`, womit das *GameObject* sichtbar wird, zeigen `questEndedMessage` in der GUI an und weisen schließlich die EPs dem *EPController* zu. Da unser Spiel zudem nur aus einer einzigen Quest besteht, setzen wir schließlich auch die Variable `gameEnded` vom *PlayerController* auf TRUE.

Besitzt der Spieler allerdings kein Objekt mit der ID „Bucket“, wird der Quest-Text angezeigt. Sollte der Spieler diesen bereits schon mal gelesen haben und diesen Trigger wiederholt auslösen, wird der Alternativtext `questMessage2` angezeigt.

#### **Listing 21.54** OnTriggerEnter-Methode von WaterQuest

```
void OnTriggerEnter(Collider other)
{
    if(other.gameObject == player)
    {
        if(inventory.RemoveItem("Bucket"))
        {
            finalBucket.SetActive(true);
            playerController.gameEnded = true;
            messageText.text = questEndedMessage;
            epController.AddPoints (eps);
        }
        else
        {
            if(gotQuest)
                messageText.text = questMessage2;
            else
                messageText.text = questMessage;
            gotQuest = true;
        }
    }
}
```

Als Letztes wollen wir jetzt noch dafür sorgen, dass die `questMessage` bzw. `questMessage2` auch wieder verschwindet, sobald der Held den *Trigger* wieder verlässt. Hierfür nutzen wir die Methode `OnTriggerExit`.

#### **Listing 21.55** OnTriggerExit von WaterQuest

```
void OnTriggerExit(Collider other)
{
    if(other.gameObject == player)
    {
        messageText.text = "";
    }
}
```

### InGameMenu

Damit hätten wir die eigentliche Quest komplett. Was soll aber nun passieren, wenn der Spieler das Spiel nun durchgespielt hat? Momentan ist es so, dass der Held nicht mehr steuerbar ist, und es wird der Text von `questEndedMessage` in der GUI angezeigt. Damit es aber irgendwie weitergeht, müsste der Spieler nun die Möglichkeit bekommen, das Spiel neu zu starten bzw. in ein Start-Menü zu gelangen.

Hierfür wollen wir abschließend noch ein kleines Skript namens „InGameMenu“ erstellen, das wir ebenfalls dem „Game Controller“ zufügen wollen. Dieses benötigt nun zunächst Referenzen auf die Komponenten `PlayerController` und `LifePointController`.

**Listing 21.56** Deklaration und Zuweisung der Variablen von InGameMenu

```
private PlayerController playerController;
private LifePointController lifePointController;
void Start () {
    playerController = GameObject.FindGameObjectWithTag("Player").
        GetComponent<PlayerController>();
    lifePointController = GameObject.FindGameObjectWithTag("GameController").
        GetComponent<LifePointController>();
}
```

Nun überprüfen wir in jedem Frame, ob der Held gestorben oder das Spiel zu Ende ist. Wenn eines von beiden der Fall ist, wird die Eröffnungsszene des Games (Index 0) gestartet, sobald die Taste `Escape`, `Space` oder `Return` gedrückt wurde. Sie könnten natürlich auch alternativ über `Input.anyKeyDown` alle Tasten auf einmal überprüfen.

**Listing 21.57** Rückkehr zur Startszene per Tastendruck

```
void Update()
{
    if(lifePointController.LifePoints == 0 || playerController.gameEnded)
    {
        if(Input.GetKeyDown(KeyCode.Escape) ||
            Input.GetKeyDown(KeyCode.Space) ||
            Input.GetKeyDown(KeyCode.Return))
        {
            Application.LoadLevel(0);
        }
    }
}
```

**21.5.2.1 WaterQuest.cs**

Das `WaterQuest`-Skript besitzt die Logik der Quest und wird direkt einem `GameObject` mit einem `Trigger-Collider` zugefügt.

**Listing 21.58** WaterQuest.cs

```
using UnityEngine;
using System.Collections;

public class WaterQuest : MonoBehaviour {

    public int eps = 20;
    public GameObject finalBucket;
    private GUIText messageText;
    private string questMessage = "Suche einen Behälter, " +
        "um dieses Wasser zu trinken.";
    private string questMessage2 = "Du brauchst einen leeren Behälter.";
    private string questEndedMessage = "<size=20>Gratulation!</size>\n" +
        "Du hast die Aufgabe gelöst.";
    private bool gotQuest = false;
    private GameObject player;
    private Inventory inventory;
    private PlayerController playerController;
    private EPController epController;
```

```

void Start () {
    finalBucket.SetActive(false);
    player = GameObject.FindGameObjectWithTag ("Player");
    playerController = player.GetComponent<PlayerController>();
    inventory = GameObject.FindGameObjectWithTag("Inventory").
        GetComponent<Inventory>();
    messageText = GameObject.FindGameObjectWithTag ("Message").
        GetComponent<GUIText>();
    epController = GameObject.FindGameObjectWithTag("GameController").
        GetComponent<EPController>();
}

void OnTriggerEnter(Collider other)
{
    if(other.gameObject == player)
    {
        if(inventory.RemoveItem("Bucket"))
        {
            finalBucket.SetActive(true);
            playerController.gameEnded = true;
            messageText.text = questEndedMessage;
            epController.AddPoints (eps);
        }
        else
        {
            if(gotQuest)
                messageText.text = questMessage2;
            else
                messageText.text = questMessage;
            gotQuest = true;
        }
    }
}

void OnTriggerExit(Collider other)
{
    if(other.gameObject == player)
    {
        messageText.text = "";
    }
}
}

```

### 21.5.2.2 WaterdropSound.cs

Dieses Skript erzeugt einen Sound, sobald ein Partikel eines Partikelsystems kollidiert. Fügen Sie dieses Skript dem „Waterdrops“-Partikelsystem hinzu, das das heruntertropfende Wasser erzeugt. Weisen Sie der *AudioClip*-Variablen *clip* die Datei „waterdrop“ zu.

#### **Listing 21.59** WaterdropSound.cs

```

using UnityEngine;
using System.Collections;

public class WaterdropSound : MonoBehaviour {
    public AudioClip clip;
}

```

```
void OnParticleCollision(GameObject other) {
    ParticleSystem.CollisionEvent[] collisionEvents =
        new ParticleSystem.CollisionEvent[5];
    int quantityCollisionEvents =
        particleSystem.GetCollisionEvents(other, collisionEvents);
    int i = 0;
    while (i < quantityCollisionEvents) {
        Vector3 pos = collisionEvents[i].intersection;
        AudioSource.PlayClipAtPoint(clip, pos, 0.3F);
        i++;
    }
}
```

### 21.5.2.3 InGameMenu.cs

Das *InGameMenu*-Skript dient dem Beenden des Spiels bzw. dessen Neustart. Fügen Sie das Skript aus dem folgenden Listing dem „Game Controller“ zu.

**Listing 21.60** InGameMenu.cs

```
using UnityEngine;
using System.Collections;

public class InGameMenu : MonoBehaviour {

    private PlayerController playerController;
    private LifePointController lifePointController;
    // Use this for initialization
    void Start () {
        playerController = GameObject.FindGameObjectWithTag("Player").
            GetComponent<PlayerController>();

        lifePointController = GameObject.FindGameObjectWithTag("GameController").
            GetComponent<LifePointController>();
    }

    void Update()
    {
        if(lifePointController.LifePoints == 0 || playerController.gameEnded)
        {
            if(Input.GetKeyDown(KeyCode.Escape) ||
                Input.GetKeyDown(KeyCode.Space) || Input.GetKeyDown(KeyCode.Return))
            {
                Application.LoadLevel(0);
            }
        }
    }
}
```

### 21.5.3 Sub-Quest erstellen

Damit der Spieler seine Quest lösen kann, muss eine *Prefab*-Instanz des Behälters mit der jeweiligen Inventar-ID im Spiel platziert werden, die er auch finden kann. Allerdings macht es wenig Sinn, wenn die Lösung der Quest gleich neben dem Questgeber zu finden ist und der Spieler diesen Gegenstand einfach aufzunehmen braucht. Um dem Spieler das Lösen der Aufgabe etwas schwieriger zu machen, gibt es sogenannte Sub-Quests, die zuvor gelöst werden müssen. Dies können Rätsel sein oder auch einfach Gegner, die bezwungen werden müssen.

Wir wollen in unserem Spiel beides einsetzen und beginnen damit, dem Spieler eine kleine Suchaufgabe zu stellen. Und zwar hatten Sie bereits im Kapitel „Animationen“ ein animiertes Fallgatter kennengelernt, das wir nun in unserem Spiel nutzen wollen. Dieses kann z.B. einen Gang sperren, an dessen Ende der Wasserbehälter zu finden ist. Damit wir hier aber auch eine echte Aufgabe erhalten, soll das Tor verschlossen sein, sodass der Spieler zuerst den passenden Schlüssel finden muss.

1. Erstellen Sie also nun wie in dem Kapitel „Animation“ beschrieben dieses Fallgatter mit seinen Animationen, dem *Animator Controller* und dem Controller-Skript inklusive der *Animation Events* und dem *DoorMovingState*-Skript.
2. Legen Sie einen neuen Tag „Door“ an und weisen Sie diesen dem Kind-Objekt „gate\_01“ zu.
3. Fügen Sie dem Kind-Objekt „gate\_01“ nun noch das *HoverEffects*-Skript mit den beiden Texturen „icon\_01\_32x32“ und „icon\_02\_32x32“ zu (siehe „Inventar-Items“). Als *tooltipText* weisen Sie „Eisentor“ zu.
4. Ziehen Sie danach das gesamte Fallgatter-*GameObject* in den „Prefab“-Ordner des *Project Browsers*, um ein *Prefab* mit dem Namen „Gate“ zu erzeugen.

### Tor abschließen

Um nun das Tor „zu verschließen“, brauchen Sie nichts anderes zu machen, als das *Animate Door*-Skript um eine boolesche Variable zu ergänzen, die das Schloss symbolisiert. Ist das Tor geöffnet, ist diese Variable TRUE, ist das Tor verschlossen, so ist sie FALSE. Deswegen nennen wir diese Variable auch einfach *isLocked*.

Außerdem soll eine Reaktion kommen, wenn das Tor verschlossen ist. Anstatt wieder eine Textnachricht anzuzeigen, wollen wir unseren Helden etwas sagen lassen, das dem Spieler den Hinweis gibt, einen Schlüssel zu suchen. Hierfür benötigen wir eine zweite *AudioClip*-Variable *lockedDoorMonologueClip*. Dieser *AudioClip* soll wieder über die eigene Player- *AudioSource* abgespielt werden, weshalb wir auch eine Variable für den Player benötigen. Außerdem brauchen wir noch eine Referenz auf unser *Inventory*, um von dort den Schlüssel zu holen.

**Listing 21.61** Variablen und Start-Methode vom *AnimateDoor*-Skript

```
public AudioClip doorClip;
public AudioClip lockedDoorMonologueClip;
public bool isLocked = true;

private Animator anim;
```

```

private int switchTrigger;
private DoorMovingState doorMovingState;
private GameObject player;
private Inventory inventory;

void Start () {
    switchTrigger = Animator.StringToHash ("Switch");
    anim = transform.parent.GetComponent<Animator>();
    doorMovingState = transform.parent.GetComponent<DoorMovingState>();
    player = GameObject.FindGameObjectWithTag("Player");
    inventory = GameObject.FindGameObjectWithTag("Inventory").
        GetComponent<Inventory>();
}

```

Da wir die Variable `isLocked` als *public* deklarieren, können wir dieses Skript für Tore nutzen, die nicht verschlossen sind. Sie brauchen einfach den Wert auf TRUE zu setzen.

Nun müssen wir noch den Lock-Mechanismus in die `OnMouseDown`-Methode implementieren. Hierfür überprüfen wir zunächst, ob das Tor verschlossen ist. Ist das der Fall, versuchen wir, aus dem Inventarsystem ein Item mit der ID „Key“ zu holen. Sollte das mit Erfolg gekrönt sein, setzen wir `isLocked` auf TRUE.

#### **Listing 21.62** Key-Überprüfung im AnimateDoor-Skript

```

if (isLocked)
{
    if(inventory.RemoveItem("Key"))
    {
        isLocked = false;
    }
}

```

Als Nächstes überprüfen wir, ob `isLocked` den Wert FALSE hat. Ist das der Fall, führen wir den bereits entwickelten Code zum Animieren des Tores aus dem Kapitel „Animationen“ aus. Ist der Wert aber noch TRUE, so geben wir den `AudioClip` `lockedDoorMonologueClip` über die  `AudioSource` des Spielers aus.

```

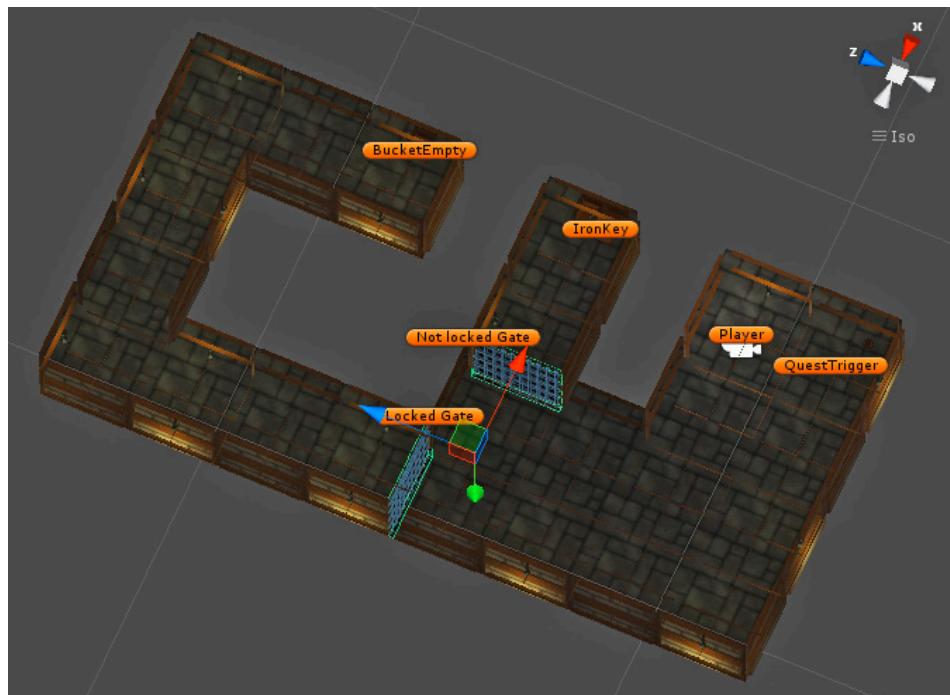
if (!isLocked)
{
    if (!doorMovingState.isMoving)
    {
        anim.SetTrigger (switchTrigger);
        AudioSource.PlayClipAtPoint(doorClip,transform.position);
    }
}
else
{
    player.audio.clip = lockedDoorMonologueClip;
    player.audio.Play ();
}

```

Weisen Sie nun im *Inspector* der Variablen `doorClip` die Datei „chains“ und `lockedDoorMonologueClip` den `AudioClip` „locked-door-monologue“ zu und drücken Sie **Apply**, damit diese Anpassungen auf das *Prefab* übertragen werden.

## Objekte platzieren und konfigurieren

Nun können Sie in Ihrem Dungeon den Startpunkt des Helden festlegen und alle Objekte der Quest und Sub-Quest so platzieren, dass sich daraus ein spielbarer Handlungsstrang ergibt (siehe Bild 21.21). Hierzu gehört neben dem Wasserbehälter-Prefab „BucketEmpty“ auch das Schlüssel-Prefab „IronKey“. Beide sollten bereits die notwendigen *InventoryItem*-Skripte mit den jeweiligen IDs besitzen. Außerdem brauchen Sie noch eine Instanz des „Gate“-Prefabs, um das verschlossene Tor darzustellen (*isLocked* ist TRUE). Sie können auch mehrere Tore in Ihrem Dungeon platzieren, die z. B. auch mal nicht verschlossen sind. Setzen Sie bei diesen Gates *isLocked* auf FALSE.



**Bild 21.21** Platzierung der Quest-Items im Dungeon

Wenn Sie dies gemacht haben, können Sie nun Ihr Spiel starten und testen, ob alle Objekte auch tatsächlich so reagieren, wie Sie sich das wünschen.

Achten Sie darauf, dass die spielrelevanten Objekte alle gut sichtbar und anklickbar im Spiel integriert sind. Bei einer starren Kamera, wie wir sie hier einsetzen, ist es am besten, solche Objekte auf Augenhöhe bzw. Kamerahöhe zu positionieren. Nutzen Sie hierfür Dekorationsgegenstände wie Holzkisten oder Fässer, um auf diesen die spielrelevanten Objekte zu platzieren.

Auch können Sie diese Objekte auffälliger gestalten, damit diese eher gesehen werden. So können Sie z. B. durch den Einsatz besonderer *Shader* oder auch geschickt platzierte Lichtquellen die Objekte präsenter und auffälliger darstellen.



**Bild 21.22** Sichtbarkeitsvergleich: Schlüssel mit Point Light und ohne Point Light

In Bild 21.23 wurde aus diesem Grund auf dem linken Motiv dem Schlüssel ein *Point Light* als Kind-Objekt zugefügt, das kurz oberhalb des Schlüssels positioniert wurde. Damit dieses die Umgebung nicht zu stark erhellt, wurde der *Radius* auf 0.1 reduziert und die *Intensity* auf 8 hochgesetzt. Auf dem rechten Motiv sehen Sie den gleichen Schlüssel ohne das *Point Light*.

Je nach Anspruch des Spiels können Sie so dem Spieler unter die Arme greifen und helfen, Aufgaben schneller zu lösen. Am Ende sollten Sie so etwas immer abhängig von der Zielgruppe machen und mithilfe von Testspielern noch einmal prüfen. Nur so kann man am Ende auch wirklich herausfinden, ob die jeweiligen Aufgaben realistisch, zu schwer oder vielleicht auch zu einfach sind.

### 21.5.3.1 AnimateDoor.cs

Das *AnimatorDoor*-Skript wird dem „gate\_01“-Objekt zugefügt, das ein Kind-Objekt von „Gate“ ist. Es steuert das Öffnen und Schließen des Fallgatters und die Schließlogik.

**Listing 21.63** AnimateDoor.cs

```
using UnityEngine;
using System.Collections;

public class AnimateDoor : MonoBehaviour {
    public AudioClip doorClip;
    public AudioClip lockedDoorMonologueClip;
    public bool isLocked = true;

    private Animator anim;
    private int switchTrigger;
    private DoorMovingState doorMovingState;
    private GameObject player;
    private Inventory inventory;

    void Start () {
        switchTrigger = Animator.StringToHash ("Switch");
        anim = transform.parent.GetComponent<Animator>();
        doorMovingState = transform.parent.GetComponent<DoorMovingState>();
```

```

player = GameObject.FindGameObjectWithTag("Player");
inventory = GameObject.FindGameObjectWithTag("Inventory").
    GetComponent<Inventory>();
}

void OnMouseDown() {
    if (isLocked)
    {
        if(inventory.RemoveItem("Key"))
        {
            isLocked = false;
        }
    }

    if (!isLocked)
    {
        if (!doorMovingState.isMoving)
        {
            anim.SetTrigger (switchTrigger);
            AudioSource.PlayClipAtPoint(doorClip,transform.position);
        }
    }
    else
    {
        player.audio.clip = lockedDoorMonologueClip;
        player.audio.Play ();
    }
}
}

```

### 21.5.3.2 DoorMovingState.cs

Das Skript wurde bereits im Kapitel „Animationen“ entwickelt und dient dem Anzeigen, ob sich das Fallgatter gerade rauf bzw. runter bewegt. Dieses Skript muss dem *GameObject* „Gate“ zugefügt werden.

**Listing 21.64** DoorMovingState.cs

```

using UnityEngine;
using System.Collections;

public class DoorMovingState : MonoBehaviour {
    public bool isMoving=false;

    void SwitchMovingState(int id)
    {
        isMoving = !isMoving;
        //Debug.Log (id);
    }
}

```

## ■ 21.6 Gegner erstellen

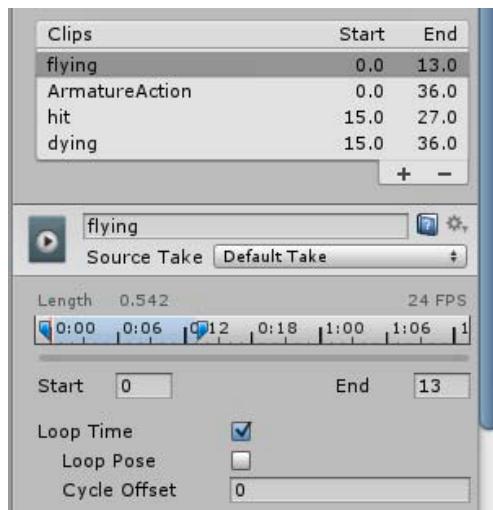
Zu einem echten Abenteuerspiel gehören neben Rätseln natürlich auch Gegner, die den Helden angreifen. In unserem Fall wollen wir Fledermäuse in unser Dungeon integrieren, die den Spieler attackieren sollen.

Hierfür benötigen wir aber nicht nur ein Fledermaus-Modell, sondern auch Animationen, die das Modell entsprechend bewegen lassen. Außerdem benötigen wir eine künstliche Intelligenz, die die Fledermaus mithilfe der bereits vorgestellten *Pathfinding*-Funktionen steuert (siehe Kapitel „Künstliche Intelligenz“) und unseren Helden auch angreifen lässt.

### 21.6.1 Model-, Rig- und Animationsimport

Da das Fledermaus-Modell „bat\_03“ bereits eigene Animationen mitbringt, brauchen wir uns um das Erstellen der Animationen nicht mehr zu kümmern. Allerdings werden diese Animationen häufig beim Export aus dem eigentlichen Modelling-Programm zu einer einzigen großen Animation zusammengepackt. Auch in unserem Fall ist dies so, weshalb wir nun erst einmal in den *Import Settings* diese wieder auseinandernehmen und auf verschiedene *Animation-Clips* verteilen müssen.

Gehen Sie hierfür in den *Import Settings* des Fledermaus-Modells „bat\_03“ auf den *Animations*-Reiter. Mithilfe des Pluszeichens unterhalb der *Animation-Clips*-Liste können Sie nun neue Clips der Liste hinzufügen. Den Ausschnitt der hierzugehörigen Animation legen Sie dann über die Zeitleiste oder die beiden unteren Eingabefelder fest. Den Namen können Sie direkt in das Textfeld eingeben und den vorgeschlagenen Namen überschreiben. Insgesamt erstellen wir auf diese Weise nun drei Animationen: „flying“, „hit“ und „dying“ (siehe Bild 21.24).



**Bild 21.23**

Animationsimport des Fledermaus-Modells

Beachten Sie hier noch den Parameter *Loop Time*, der für das Wiederholen der *Animation-Clips* zuständig ist. Aktivieren Sie diesen Parameter nur bei „flying“, bei den anderen *Animation-Clips* bleibt dieser deaktiviert.

Weiter sollten Sie noch auf dem Reiter *Model* der *Import Settings* den *Scale Factor* auf 0.1 stellen und auf dem Reiter *Rig* kontrollieren, dass auch der *Animation Type* „Generic“ gewählt wurde.

Alle Anpassungen müssen natürlich mit **Apply** bestätigt und auf das Modell übertragen werden.

## 21.6.2 Komponenten und Prefab konfigurieren

Zunächst wollen wir das *GameObject* für unseren Gegner mit allen notwendigen Komponenten ausstatten, die dieses benötigt.

1. Ziehen Sie das Objekt „bat\_03“ in die Szene und nennen es in „Bat“ um.
2. Erzeugen Sie einen neuen *Tag* „Enemy“ und weisen Sie diesen dem *GameObject* zu.
3. Fügen Sie dem *GameObject* nun ein *Rigidbody* zu. Weil wir das *GameObject* über einen *NavMeshAgent* steuern werden, deaktivieren Sie *Use Gravity* und aktivieren Sie *Is Kinematic*.
4. Geben Sie „Bat“ einen *Sphere-Collider* mit einem *Radius* von 0.5 und aktivieren Sie den *Is Trigger*-Parameter.
5. Geben Sie dem *GameObject* eine  *AudioSource*, bei der Sie *Loop* und *Play On Awake* deaktivieren. Weisen Sie der  *AudioSource* die Datei „batCry“ als  *AudioClip* zu.
6. Für das *Pathfinding* fügen wir „Bat“ schließlich über **Component/Navigation/Nav Mesh Agent** eine *NavMeshAgent*-Komponente zu. Die Einstellungen dieser Komponente übernehmen Sie bitte aus Bild 21.24.
7. Abschließend ziehen Sie das gesamte *GameObject* in den „Prefabs“-Ordner, um ein *Prefab* daraus zu erstellen.

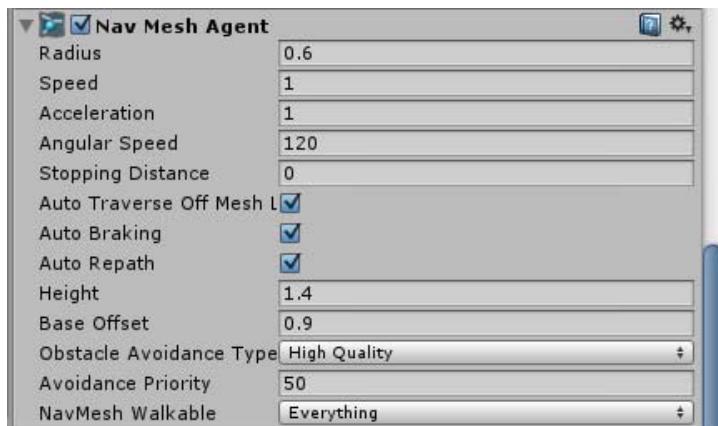
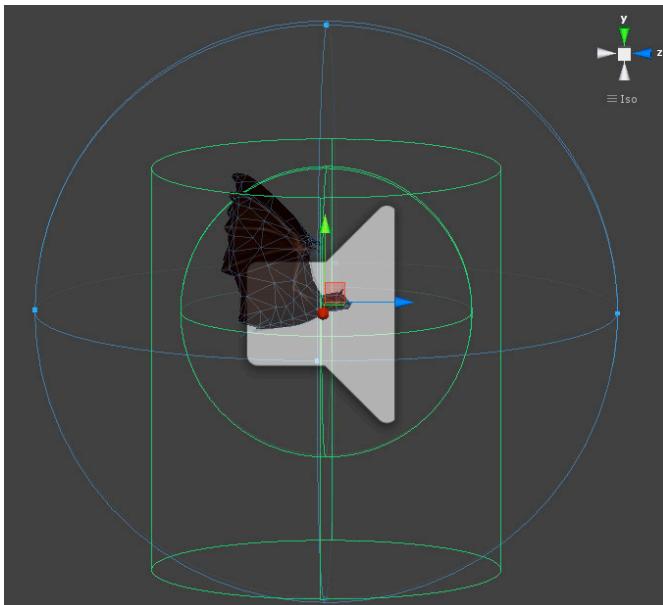


Bild 21.24 NavMeshAgent-Einstellungen der Fledermaus

Bei der Konfiguration des *NavMeshAgents* müssen wir vor allem bedenken, dass unser NPC fliegt und sich nicht auf dem Boden bewegt. Dabei wollen wir es aber nicht zu kompliziert machen und verschieben das Objekt der Fledermaus innerhalb des *NavMeshAgent* einfach an das obere Ende mit einem *Base Offset* und reduzieren die Höhe auf das obere Ende der Fledermaus bzw. deren Flügelschlag.

Da unsere Fledermaus zunächst nur wie eine Art Wache lediglich eine vordefinierte Strecke langsam abfliegen soll, legen wir die Parameter *Speed* und *Acceleration* zunächst auf recht niedrige Werte. Später werden wir beim Angreifen *Speed* noch via Skript hochsetzen.



**Bild 21.25**  
Fledermaus mit  
NavMeshAgent, Collider  
und AudioSource

### 21.6.3 Animator Controller erstellen

Damit wir unsere Fledermaus nun animieren können, benötigen wir als Erstes einen *Animator Controller*. Selektieren Sie hierfür den „Animations“-Ordner im *Project Browser* und erstellen z. B. über **Assets/Create/Animator Controller** einen *Animator Controller* mit dem Namen „BatController“.

Nun können wir uns um die verschiedenen *Animation States* kümmern, die unser NPC besitzen soll. Öffnen Sie hierfür das *Animator*-Fenster über **Window/Animator** und selektieren Sie den gerade erstellten „BatController“.

Unsere Fledermaus wird insgesamt vier unterschiedliche *Animation States* besitzen, die Sie über das Kontextmenü des *Animator*-Fensters mit **Create State/Empty** erstellen können (erreichbar über die rechte Maustaste). Diese benennen und konfigurieren Sie nun wie folgt:

- **Flying** wird mit „Set as Default“ als *Default Animation State* festgelegt und soll während des normalen Fliegens abgespielt werden. Weisen Sie der *Motion*-Eigenschaft den *Animati-*

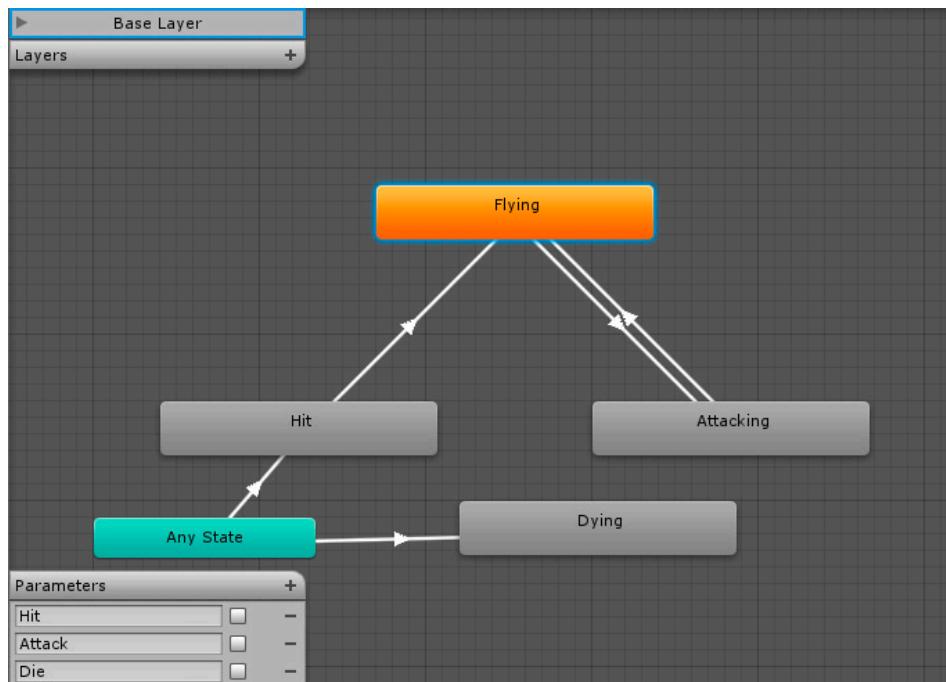
*tion-Clip „flying“ zu. Sollten Sie den Clip nicht in der Auswahl finden, achten Sie darauf, dass Sie in dem Auswahlfenster im Bereich „Assets“ suchen, nicht im Reiter „Scene“.*

- **Attacking** wird aktiv sein, wenn die Fledermaus dem Helden Schaden zufügt. Weisen Sie auch hier den normalen „flying“ *Animation-Clip* zu. Allerdings soll dieser mit der fünf-fachen Geschwindigkeit abgespielt werden. Deshalb setzen Sie *Speed* auf 5.
- **Hit** weisen Sie den *Animation-Clip* „hit“ zu. Dieser soll abgespielt werden, wenn der Fledermaus Schaden zugefügt wird.
- **Dying** weisen Sie den *Animation-Clip* „dying“ zu. Dieser *Animation State* wird dann aktiv, wenn die Fledermaus stirbt.

Zum Steuern dieser *Animation States* benötigen wir nun insgesamt drei verschiedene Parameter, die Sie in der *Parameters*-Liste anlegen müssen:

- **Hit** als *Trigger*-Parameter
- **Attack** als *Bool*-Parameter
- **Die** als *Bool*-Parameter

Nun können Sie die *Transitions* definieren, die für das Wechseln zwischen den verschiedenen *Animation States* zuständig sind (siehe Bild 21.26). Klicken Sie hierfür mit der rechten Maustaste auf den Start-*Animation State* und wählen **Make Transition** aus. Danach klicken Sie mit der linken Maustaste auf den Ziel-*Animation State*.



**Bild 21.26** Animator Controller der Fledermaus

Im *Inspector* der jeweiligen *Transition* werden nun die *Conditions* hinterlegt, bei welchen erfüllten Bedingungen von einem *State* in den anderen gewechselt werden soll:

- **Any State -> Hit:** Hit; Die = FALSE.
- **Any State -> Dying:** Hit; Die = TRUE.
- **Hit -> Flying:** Exit Time = 0.80.
- **Flying -> Attacking:** Attack = TRUE.
- **Attacking -> Flying:** Attack = FALSE.

Nun können Sie den *Animator Controller* der *Animator*-Komponente unseres „Bat“-Prefabs als *Controller* zuweisen. Sie können hierfür die „Bat“-Instanz in der Szene nutzen und nach dem Zuweisen des Controllers mit **Apply** diese Anpassungen wieder auf das *Prefab* übertragen. Da die Animationen keine Bewegungen auf das Modell übertragen, können Sie dort auch gleich noch *Apply Root Motion* deaktivieren.

#### 21.6.4 NavMesh erstellen

Als Nächstes wollen wir nun das *NavMesh* für die *Pathfinding*-Funktion erstellen. Öffnen Sie hierfür über **Window/Navigation** das Navigation-Fenster.

Markieren Sie nun alle statischen Objekte Ihrer Szene mit dem Kennzeichen *Navigation Static* im *Navigation*-Fenster. Oder setzen Sie gleich komplett den *Static*-Haken im *Inspector* der jeweiligen *GameObjects*. In unserem Fall sollten das alle „Wall“-, „Floor“- und „Ceiling“-Objekte sowie die Dekorationsgegenstände „Create“ und „Barrel“ sein.



Bild 21.27 Dungeon mit NavMesh

Achten Sie darauf, dass alle „Gate“-Objekte und deren Kind-Objekte nicht als *Static* deklariert werden. Ansonsten werden diese immer als Hindernisse in die Wegberechnungen einfließen, unabhängig davon, ob das Tor offen oder geschlossen ist.

Nun müssen wir noch die Einstellungen im *Bake*-Reiter des *Navigation*-Fensters auf den *NavMeshAgent* anpassen.

- Stellen Sie *Radius* auf 0.6.
- Setzen Sie *Height* auf 1.4.

Nun können Sie das *NavMesh* über den **Bake**-Button erstellen. Achten Sie darauf, dass Sie nach dem Verschieben, Entfernen oder Zufügen neuer Dekorationsgegenstände das *Baken* erneut starten, damit das *NavMesh* ebenfalls neuberechnet wird.

Nach dem Erstellen sehen Sie blau dargestellt das *NavMesh*. Nun können Sie überprüfen, ob Sie vielleicht vergessen haben, ein Hindernis als *Static* zu bezeichnen oder irrtümlicherweise ein bewegliches Teil als solches zu definieren.

## 21.6.5 Umgebung und Feinde erkennen

Sobald die Fledermaus unseren Held bemerkt, soll diese ihn angreifen. Hierfür wollen wir ein Skript namens *EnemySonar* entwickeln, das nach dem Spieler Ausschau hält.

Zusätzlich soll dieses Skript aber auch dafür sorgen, dass die Fledermaus nicht gegen das Fallgatter fliegt. Schließlich haben wir das Fallgatter („Gate“) nicht als *Static* bezeichnet, wodurch das gesamte Objekt für die Wegberechnung nicht als Hindernis berücksichtigt wird. Nutzen Sie Unity Pro, können Sie das Kind-Objekt „gate\_01“ einfach mit einer *NavMeshObstacle*-Komponente ausstatten und diesen Teil des Skriptes vernachlässigen. Unity kümmert sich dann selbstständig um das Beachten des Fallgatters. Wenn Sie aber die kostenlose Version einsetzen, zeige ich Ihnen, wie Sie dies auch ohne diese Komponente umsetzen können.

Für dieses Skript benötigen wir nun vier Variablen. Die erste Variable *fieldOfView* gibt das Sichtfeld des NPC an, die zweite *obstacleDetected* gibt an, ob ein Hindernis erkannt wurde. Die dritte Variable *playerDetected* speichert, ob der Spieler entdeckt wurde, und als Letztes benötigen wir noch eine Variable, in der wir das „Player“-*GameObject* speichern. Schließlich soll das Skript ja genau dieses Objekt suchen.

### **Listing 21.65** Variablen von *EnemySonar*

```
public float fieldOfView = 120;
public bool obstacleDetected = false;
public bool playerDetected = false;
private GameObject player;
```

Für unseren Suchvorgang benötigen wir genau eine Methode, die wir *Searching* nennen wollen. In dieser wollen wir mithilfe eines *Raycasts* nach dem Spieler suchen. Da dies aber sehr rechenintensiv ist und bei vielen Gegnern in einem Spiel doch irgendwann auf die Performance drücken könnte, wollen wir diesen Vorgang nur jede halbe Sekunde ausführen.

Hierfür starten wir diese Methode in der Start-Methode mit dem Befehl `InvokeRepeating`, wo wir auch den Wiederholungsrhythmus festlegen. Möchten Sie das Intervall ändern, können Sie das hier sehr leicht einstellen.

#### **Listing 21.66** Start-Methode von EnemySonar

```
void Start () {
    player = GameObject.FindGameObjectWithTag ("Player");
    InvokeRepeating("Searching",0.5F,0.5F);
}
```

In der `Searching`-Methode schicken wir nun von der eigenen Position zu der Position des Spielers (die Variablenzuweisung hatten wir bereits in der `Start`-Methode gemacht) einen `Raycast`. Wenn sich kein Hindernis dazwischen befindet, überprüfen wir nun, ob sich der Winkel in dem Rahmen von `fieldOfView` bewegt.

Wenn das der Fall ist, rufen wir die Methode `StopSearching` auf, die die Variable `playerDetected` auf `TRUE` setzt und die Suche-Methode `Searching` stoppt. Denn sobald der Gegner den „Player“ entdeckt hat, soll er diesem folgen, auch wenn dieser vielleicht mal hinter einer Ecke verschwindet und nicht mehr vom NPC gesehen wird.

Damit wir den Suchvorgang auch von einem anderen Skript aus unterbrechen können, erstellen wir `StopSearching` gleich als öffentlich sichtbare Methode.

#### **Listing 21.67** Searching und StopSearching von EnemySonar

```
void Searching()
{
    Vector3 direction = player.transform.position - transform.position;
    Ray ray = new Ray(transform.position,direction.normalized);
    RaycastHit hit;
    if (Physics.Raycast(ray,out hit,5))
    {
        if(hit.collider.gameObject == player)
        {
            Vector3 dir = hit.transform.position - transform.position;
            float angle = Vector3.Angle(dir, transform.forward);
            if(angle < fieldOfView * 0.5f)
                StopSearching ();
        }
    }
}

public void StopSearching ()
{
    playerDetected = true;
    CancelInvoke ("Searching");
}
```

Kommen wir nun zu dem Teil, den wir für das Erkennen des geschlossenen Fallgatters benötigen. Da wir den *Sphere-Collider* als *Trigger-Collider* definiert haben, überprüfen wir in der `OnTriggerEnter`-Methode, ob wir mit dem Fallgatter kollidieren. Ist das der Fall, dann setzen wir `obstacleDetected` auf `TRUE`.

Wenn wir nun aus einem *Collider* mit dem Tag „Door“ heraustreten, z.B. weil das Tor hochfährt, setzen wir die Variable auf `FALSE`. Dies machen wir mit der `OnTriggerExit`-Methode.

```

void OnTriggerEnter(Collider other)
{
    if (other.gameObject.CompareTag("Door"))
    {
        obstacleDetected = true;
    }
}

void OnTriggerExit(Collider other)
{
    if (other.gameObject.CompareTag("Door"))
    {
        obstacleDetected = false;
    }
}

```

Natürlich hat diese Lösung auch einige Schwächen, aber für unseren Zweck sollte es erst einmal reichen. Am elegantesten ist es natürlich, Unity Pro mit der *NavMeshObstacle*-Komponente oder aber ein anderes/eigenes *Pathfinding*-System zu nutzen. Dies ist natürlich auch möglich, aber eher weniger für Einsteiger geeignet.

Wenn Sie das Skript nun fertig haben, fügen Sie es dem „Bat“-*Prefab* unserer Fledermaus zu.

### 21.6.5.1 EnemySonar.cs

Das folgende Listing zeigt das komplette *EnemySonar*-Skript, das Sie dem Fledermaus-*Prefab* zuweisen müssen.

#### **Listing 21.68** EnemySonar.cs

```

using UnityEngine;
using System.Collections;

public class EnemySonar : MonoBehaviour {
    public float fieldOfView = 120;
    public bool obstacleDetected = false;
    public bool playerDetected = false;
    private GameObject player;
    void Start () {
        player = GameObject.FindGameObjectWithTag ("Player");
        InvokeRepeating("Searching",0.5F,0.5F);
    }

    void Searching()
    {
        Vector3 direction = player.transform.position - transform.position;
        Ray ray = new Ray(transform.position,direction.normalized);
        RaycastHit hit;
        if (Physics.Raycast(ray,out hit,5))
        {
            if(hit.collider.gameObject == player)
            {
                Vector3 dir = hit.transform.position - transform.position;
                float angle = Vector3.Angle(dir, transform.forward);
            }
        }
    }
}

```

```

        if(angle < fieldOfView * 0.5f)
            StopSearching ();

    }

}

public void StopSearching ()
{
    playerDetected = true;
    CancelInvoke ("Searching");
}

void OnTriggerEnter(Collider other)
{
    if (other.gameObject.CompareTag("Door"))
    {
        obstacleDetected = true;
    }
}

void OnTriggerExit(Collider other)
{
    if (other.gameObject.CompareTag("Door"))
    {
        obstacleDetected = false;
    }
}
}

```

## 21.6.6 Gesundheitszustand verwalten

Als Nächstes wollen wir uns um die Lebensverwaltung bzw. den Gesundheitszustand unseres NPC kümmern. Wie das *PlayerHealth*-Skript soll auch dieses Skript auf der *HealthController*-Klasse aufbauen.

Also erstellen wir ein neues Skript namens *EnemyHealth*, das von *HealthController* erbt und folgende Variablen besitzt (siehe Listing 21.69):

**Listing 21.69** Variablen und Start-Methode von *EnemyHealth*

```

using UnityEngine;
using System.Collections;

public class EnemyHealth : HealthController {
    public bool isShocked = false;
    public float shockedTime = 0.5F;
    public int eps = 1;
    private EnemySonar enemySonar;
    private EPController epController;
    private Animator anim;
    private int hitTrigger;
    private int dieBool;
    void Start()
    {

```

```

enemySonar = GetComponent<EnemySonar>();
epController = GameObject.FindGameObjectWithTag("GameController").
    GetComponent<EPController>();
hitTrigger = Animator.StringToHash ("Hit");
dieBool = Animator.StringToHash ("Die");
anim = transform.GetComponent<Animator>();
}

```

Wie Sie dem Listing entnehmen können, benötigen wir hier Referenzen auf das vorher programmierte *EnemySonar*-Skript, auf das *EPController*-Skript des „Game Controllers“ sowie auf die eigene *Animator*-Komponente. Außerdem wollen wir später noch die „Hit“ und „Die“-Variablen aus dem *Animator* ansprechen, weshalb wir deren Hash-Werte in Variablen speichern. Erfahrungspunkte soll es für das Töten eines Gegners natürlich auch geben, weshalb wir noch eine Variable *eps* brauchen. Zum Schluss brauchen wir noch zwei Variablen für den Schockzustand, in den der Gegner verfallen soll, wenn er verletzt wurde. Hierfür brauchen wir eine boolesche Variable, die den Status speichert, und eine Variable, die die Dauer dieses Zustands vorgibt.

Im nächsten Schritt wollen wir die beiden überschreibbaren Methoden *Damaging* und *Dying* ausprogrammieren, die in der Basisklasse *HealthController* leer sind.

Als Erstes rufen wir in der *Damaging*-Methode den Trigger-Parameter „Hit“ auf, damit der *Animator Controller* in den „Hit“-State wechselt. Zusätzlich soll noch der Sound der eigenen  *AudioSource* abgespielt und die *isShocked*-Variable auf TRUE gesetzt werden. Diese soll natürlich nach einer durch die Variable *shockedTime* vorgegebenen Zeit wieder zurückgesetzt werden. Dies machen wir in einer gesonderten Methode *ResetShocked*.

#### **Listing 21.70** Damaging und ResetShocked vom EnemyHealth-Skript

```

public override void Damaging ()
{
    anim.SetTrigger(hitTrigger);
    audio.Play ();
    isShocked = true;

    if (!enemySonar.playerDetected)
        enemySonar.StopSearching();

    Invoke("ResetShocked", shockedTime);
}

void ResetShocked()
{
    isShocked = false;
}

```

Außerdem wollen wir, dass der Spieler als entdeckt gilt, sobald dieser den Gegner verletzt. Deshalb rufen wir zusätzlich auch *StopSearching* vom *EnemySonar*-Skript auf, wenn dieser bisher noch nicht entdeckt wurde.

In der *Dying*-Methode setzen wir neben dem „Hit“-Trigger den „Die“-Parameter auf TRUE. Auch hier spielen wir den  *AudioClip* ab und setzen *isShocked* auf TRUE. Allerdings rufen wir nun stattdessen mit einer Sekunde Verzögerung die Methode *DestroyMe* auf, die das komplette *GameObject* zerstört und dem *EPController* die Erfahrungspunkte zuweist.

**Listing 21.71** Dying und DestroyMe vom EnemyHealth-Skript

```
public override void Dying ()
{
    anim.SetBool(dieBool,true);
    anim.SetTrigger(hitTrigger);
    audio.Play ();
    isShocked = true;
    Invoke ("DestroyMe",1);
}

void DestroyMe()
{
    Destroy(gameObject);
    epController.AddPoints (eps);
}
```

Weisen Sie auch dieses Skript nun dem „Bat“-*Prefab* zu.

Eigentlich ist es damit auch getan. Achten Sie nur darauf, dass Sie in dem Dungeon ausreichend „Stone“-Items platzieren, damit der Spieler die Fledermäuse später auch töten kann.



**Bild 21.28** Beispielplatzierung von Stone-Objekten

Sollten Sie hierbei einen Aufbau wie in Bild 21.28 wählen wollen, beachten Sie, dass das *Prefab* kein *Rigidbody* besitzt. Nimmt der Spieler nun zuerst einen unteren Stein weg, bleiben die anderen in der Luft stehen.

In diesem Fall sollten Sie für alle „Stone“-Instanzen, die sich nicht ganz unten befinden, folgende Änderungen in der Szene vornehmen:

- Markieren Sie diese *GameObjects* in der Szene.
- Fügen Sie diesen ein *Rigidbody* zu.
- Aktivieren Sie *Freeze Rotation* für *X*, *Y* und *Z*.

Hierdurch bleibt die Steinpyramide zwar stehen, aber sobald die unteren Steine entfernt werden, fallen die oberen herunter.

### 21.6.6.1 EnemyHealth.cs

Das folgende *EnemyHealth*-Skript verwaltet den Gesundheitszustand eines gegnerischen NPC. Da es auf das vorherige *EnemySonar*-Skript zugreift, benötigen Sie beide Skripte, wenn Sie dieses hier dem *GameObject* zuweisen wollen.

**Listing 21.72** EnemyHealth.cs

```
using UnityEngine;
using System.Collections;

public class EnemyHealth : HealthController {
    public bool isShocked = false;
    public float shockedTime = 0.5F;
    public int eps = 1;
    private EnemySonar enemySonar;
    private EPController epController;
    private Animator anim;
    private int hitTrigger;
    private int dieBool;
    void Start()
    {
        enemySonar = GetComponent<EnemySonar>();
        epController = GameObject.FindGameObjectWithTag("GameController").
            GetComponent<EPController>();
        hitTrigger = Animator.StringToHash ("Hit");
        dieBool = Animator.StringToHash ("Die");
        anim = transform.GetComponent<Animator>();
    }

    public override void Damaging ()
    {
        anim.SetTrigger(hitTrigger);
        audio.Play ();
        isShocked = true;

        if(!enemySonar.playerDetected)
            enemySonar.StopSearching();

        Invoke("ResetShocked",shockedTime);
    }

    public override void Dying ()
    {
        anim.SetBool(dieBool,true);
        anim.SetTrigger(hitTrigger);
        audio.Play ();
        isShocked = true;
        Invoke ("DestroyMe",1);
    }

    void ResetShocked()
    {
        isShocked = false;
    }

    void DestroyMe()
    {
```

```

        Destroy(gameObject);
        epController.AddPoints (eps);
    }
}

```

## 21.6.7 Künstliche Intelligenz entwickeln

Kommen wir nun zum eigentlichen Kern des NPC, zu der künstlichen Intelligenz, kurz KI. Natürlich gehören zu dieser KI auch das bereits konfigurierte *Pathfinding* oder auch die Feinderkennung, die wir schon programmiert haben. Aber diese verschiedenen Elemente müssen nun noch zusammengeführt und in eine gemeinsame logische Struktur gepackt werden. Hierfür erstellen wir jetzt ein Skript mit dem Namen *EnemyAI*, das Sie auch gleich dem „Bat“-*Prefab* zufügen sollten.

Unser feindlicher NPC kennt nun drei unterschiedliche Zustände, um die wir uns nun kümmern müssen:

- einen Standardzustand, in dem der NPC wie eine Wache eine vordefinierte Strecke abfliegt und nach dem Spieler Ausschau hält,
- einen Angriffsztand, in dem er den Spieler attackiert,
- einen Ausnahmeztand, in dem sich der NPC nicht mehr fortbewegt, weil er sich z.B. in einem Schockzustand befindet oder der Spieler tot ist.

Zum Angreifen benötigen wir nun eine Variable *damageValue*, der wir den Schadenswert zuweisen, die der NPC dem Spieler zufügen kann. Zusätzlich brauchen wir eine Variable *attackingSpeed*, die die Geschwindigkeit festlegt, mit der die Fledermaus den Spieler angreift. Schließlich soll sie beim Angreifen schneller fliegen als beim Patrouillieren. Außerdem brauchen wir noch eine boolesche Variable, die angibt, ob die Fledermaus aktuell überhaupt in der Lage ist, zu attackieren bzw. Schaden zuzuführen.

**Listing 21.73** Variablen für das Angreifen der Fledermaus

```

public float damageValue = 1;
public float attackingSpeed = 4;
private bool readyToHit = true;

```

Für das Patrouillieren benötigen wir ein *Transform*-Array, in dem wir die Wegpunkte speichern, zwischen denen der NPC hin und her fliegen soll, sowie eine Indexvariable, die den aktuellen Index speichert. Weiter brauchen wir eine Zeitvariable *waypointPauseTime*, die angibt, wie lange die Fledermaus an einem Wegpunkt warten soll, bevor sie zum nächsten weiterfliegt.

**Listing 21.74** Variablen für das Patrouillieren der Fledermaus

```

public Transform[] waypoints;
public float waypointPauseTime = 2;
private int currentWaypointIndex;

```

Schlussendlich benötigen wir wieder Referenzen auf den Player und dessen *PlayerHealth*-Skript, auf die eigene *NavMeshAgent*-Komponente sowie das *EnemySonar*- und das *EnemyHealth*-Skript. Außerdem brauchen wir Zugriff auf den eigenen *Animator* und den Hash-Wert des „Attack“-Parameters. Hierfür definieren wir weitere Variablen und weisen diesen die Werte in der Start-Methode zu.

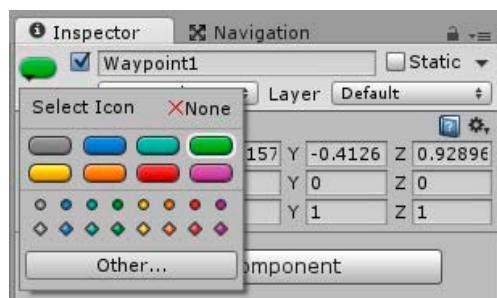
**Listing 21.75** Weitere Variablen und Start-Methode von EnemyAI

```
private GameObject player;
private PlayerHealth playerHealth;
private NavMeshAgent agent;
private EnemySonar enemySonar;
private EnemyHealth enemyHealth;
private Animator anim;
private int attackBool;
void Start ()
{
    agent = GetComponent<NavMeshAgent>();
    player = GameObject.FindGameObjectWithTag ("Player");
    playerHealth = player.GetComponent<PlayerHealth>();
    enemySonar = GetComponent<EnemySonar>();
    enemyHealth = GetComponent<EnemyHealth>();
    anim = transform.GetComponent<Animator>();
    attackBool = Animator.StringToHash ("Attack");
}
```

Als Erstes wollen wir uns um das Patrouillieren kümmern. Die Hauptaufgabe, das Steuern und Fortbewegen des NPC, wird hierbei von Unitys *Pathfinding*-Modul bereits übernommen. Wir müssen lediglich die Zielposition vorgeben, wo dieser als Nächstes hinfliegen soll.

Im Folgenden wollen wir nun immer zwei Wegpunkte (Waypoints) pro Fledermaus setzen, zwischen denen der NPC nun hin und her fliegen soll. Wenn Sie möchten, können Sie natürlich auch mehrere nehmen. Die Positionen legen wir dabei über *Empty GameObjects* fest, die wir dann dem *waypoints*-Array zuweisen.

1. Fügen Sie hierfür am besten ein neues, leeres *GameObject* der Szene zu und nennen dieses „Waypoint1“.
2. Weisen Sie diesem nun im *Inspector* ein eigenes Icon zu.



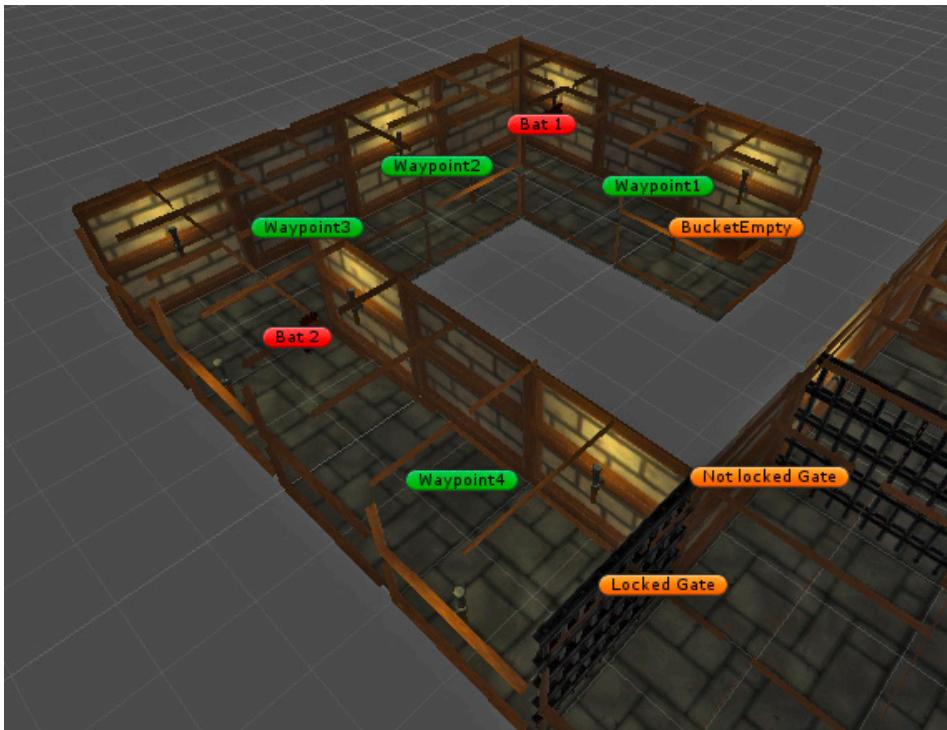
**Bild 21.29**

Icon-Zuweisung für einen Waypoint

3. Platzieren Sie den Waypoint an einer Endposition der Wegstrecke, die die Fledermaus abfliegen soll, z.B. auf *Position* (7,1,10).
4. Kopieren Sie den Waypoint und benennen Sie ihn in „Waypoint2“ um.

5. Positionieren Sie „Waypoint2“ an der zweiten der beiden Positionen, zwischen denen die Fledermaus hin- und herfliegen soll, z.B. auf *Position* (5,1,13).
6. Ziehen Sie nun eine Bat-Instanz in Ihr Spiel und platzieren diese irgendwo zwischen den beiden Waypoints und ziehen beide Waypoints auf das *waypoints*-Array.

Möchten Sie noch eine zweite oder dritte Fledermaus im Dungeon platzieren, wiederholen Sie das Vorgehen.



**Bild 21.30** Zwei platzierte Fledermäuse mit je zwei Waypoints

Kommen wir nun zu der eigentlichen Logik, die das Hin- und Herfliegen steuert. Dieses programmieren wir in einer Methode namens *WaypointWalk*. In dieser überprüfen wir, ob die Distanz zum Ziel kleiner bzw. gleich der Distanz ist, bei der wir anhalten wollen. Ist dies der Fall, verändern wir das Ziel, indem wir *currentWaypointIndex* hochzählen oder, wenn wir bereits beim höchsten Index sind, auf 0 setzen. Danach starten wir die Methode *WaypointPause*, die für eine kurze Zeit die *speed*-Eigenschaft des *NavMeshAgent* heruntergesetzt, damit der NPC nicht gleich zum nächsten Ziel fliegt.

#### **Listing 21.76** Patrouille laufen

```
void WaypointWalk ()
{
    if(agent.remainingDistance <= agent.stoppingDistance)
    {
        if(currentWaypointIndex == waypoints.Length - 1)
            currentWaypointIndex = 0;
    }
}
```

```

        else
            currentWaypointIndex++;

        StartCoroutine("WaypointPause", waypointPauseTime);
    }
    agent.SetDestination (waypoints[currentWaypointIndex].position);
}

IEnumerator WaypointPause(float seconds)
{
    float oldSpeed = agent.speed;
    agent.speed = 0.1F;
    yield return new WaitForSeconds(seconds);
    agent.speed = oldSpeed;
}

```

Im nächsten Schritt wollen wir uns nun um das Schadenzufügen kümmern. Dies wollen wir recht einfach umsetzen. Und zwar fügt die Fledermaus dem Player sofort Schaden zu, wenn diese miteinander kollidieren.

Im Abschnitt „OnCollision-Methoden“ des Kapitels „Physik in Unity“ hatte ich erläutert, dass diese Methoden nur dann ausgelöst werden, wenn mindestens eines der beiden aufeinander treffenden *Collider*-Objekte ein *Rigidbody* mit deaktiviertem *Is Kinematic*-Parameter besitzt. Da leider weder die Fledermaus (aufgrund des *NavMeshAgents*) noch unser „Player“ oder das bewegliche „Gate“ diesen besitzt, können wir hiermit nicht arbeiten. Alternativ können wir aber auf die *OnTriggerStay*-Methode, die dieses nicht voraussetzt, ausweichen.

In dieser überprüfen wir nun beim Aufruf von *OnTriggerStay*, ob zunächst einmal der NPC noch am Leben und ob das kollidierende Objekt das „Player“-Objekt ist, das natürlich auch noch am Leben sein sollte. Wenn das alles der Fall ist, wird nun der Schaden zugefügt.

Allerdings wird diese Methode in jedem Physikzyklus aufgerufen, solange sich ein anderer *Collider* in diesem befindet. Damit der Spieler nun aber nicht gleich nach einer Sekunde Aufenthalt in dem *Collider* tot ist, führen wir noch die *readyToHit*-Variable ein, die das Zufügen von Schaden nur einmal pro Sekunde zulässt.

Während die Variable sofort auf FALSE gesetzt wird, sobald Schaden zugeführt wird, wird diese mithilfe der Methode *SetReadyToHit* erst zeitverzögert wieder auf TRUE gesetzt. Zudem setzen wir in dieser Methode auch den booleschen Parameter „Attack“ des Animators, um für eine halbe Sekunde in den „Attacking“-Status zu wechseln.

#### **Listing 21.77** Schaden zufügen

```

void OnTriggerEnter(Collider other)
{
    if (enemyHealth.health > 0)
    {
        if (other.gameObject == player && playerHealth.health > 0)
        {
            if (readyToHit)
            {
                readyToHit = false;
                if (!enemySonar.playerDetected)
                    enemySonar.StopSearching();
                other.gameObject.SendMessage ("ApplyDamage", damageValue,
                    SendMessageOptions.DontRequireReceiver);
            }
        }
    }
}

```

```

        StartCoroutine("SetReadyToHit");
    }
}
}

IEnumerator SetReadyToHit()
{
    anim.SetBool(attackBool,true);
    yield return new WaitForSeconds(0.5F);
    anim.SetBool(attackBool,false);
    yield return new WaitForSeconds(0.5F);
    readyToHit = true;
}

```

Beachten Sie außerdem den Aufruf von `StopSearching` beim Zufügen des Schadens. Sollte nämlich der Spieler den NPC von hinten „anrempeln“, ist es natürlich mehr als logisch, dass die Fledermaus spätestens hierdurch den Helden bemerkt und diesen dann auch angreift.

Kommen wir nun zu der eigentlichen Logik, die in der `Update`-Methode für das Ausführen und das Wechseln zwischen den verschiedenen Status zuständig ist. Solange nun der Spieler nicht entdeckt ist, führen wir die Methode `WaypointWalk` aus. Ist das nicht der Fall, steht nun die Frage im Raum, ob sie den Player nun attackiert oder sich in dem Ausnahmezustand befindet, wo sie sich gar nicht fortbewegt.

Hierfür überprüfen wir nun, ob

- die Fledermaus noch lebt
- der Spieler noch lebt
- ein Hindernis (Fallgatter) registriert wurde
- die Fledermaus sich im Schockzustand befindet
- die Fledermaus wieder bereit zum Zuschlagen ist

Wenn alles okay ist, wird der `speed`-Parameter des `NavMeshAgents` auf die Angriffs geschwindigkeit gesetzt, die Beschleunigung wird auf zwei hochgesetzt und als Ziel die Position des Helden zugewiesen. Sollte sich aber z.B. die Fledermaus im Schockzustand befinden oder der Spieler tot sein, wird die Methode `Stop` des `NavMeshAgents` aufgerufen und der NPC bleibt auf der Stelle stehen.

#### **Listing 21.78** Update-Methode des EnemyAI-Skriptes

```

void Update ()
{
    if (!enemySonar.playerDetected)
    {
        WaypointWalk ();
    }
    else if (enemyHealth.health > 0 && playerHealth.health > 0 &&
              !enemySonar.obstacleDetected && !enemyHealth.isShocked && readyToHit)
    {
        agent.speed = attackingSpeed;
        agent.acceleration = 2;
        agent.SetDestination (player.transform.position);
    }
    else

```

```
{
    agent.Stop (true);
}
}
```

Haben Sie nun das Skript ausprogrammiert und dem „Bat“-Prefab zugefügt, sind wir mit unserem Gegner endlich fertig.

### 21.6.7.1 EnemyAI.cs

Das Skript *EnemyAI* wird dem „Bat“-*Prefab* angehängt und führt alle Bausteine der künstlichen Intelligenz zusammen.

Zu Beginn fliegt der NPC eine vordefinierte Strecke entlang, die er überwachen soll. Sobald der NPC den Player entdeckt, wird er diesen angreifen. Für das Festlegen dieser Überwachungsstrecke platzieren Sie *Empty GameObjects* an den Stellen, wo der NPC längs laufen/fliegen soll, und fügen Sie diese dem waypoints-Array zu.

**Listing 21.79** EnemyAI.cs

```
using UnityEngine;
using System.Collections;

public class EnemyAI : MonoBehaviour {
    public float damageValue = 1;
    public float attackingSpeed = 4;
    public Transform[] waypoints;
    public float waypointPauseTime = 2;

    private GameObject player;
    private NavMeshAgent agent;
    private Animator anim;
    private EnemySonar enemySonar;
    private PlayerHealth playerHealth;
    private EnemyHealth enemyHealth;
    private int currentWaypointIndex;
    private bool readyToHit = true;
    private int attackBool;

    void Start ()
    {
        agent = GetComponent<NavMeshAgent>();
        player = GameObject.FindGameObjectWithTag ("Player");
        playerHealth = player.GetComponent<PlayerHealth>();
        enemySonar = GetComponent<EnemySonar>();
        enemyHealth = GetComponent<EnemyHealth>();
        anim = transform.GetComponent<Animator>();
        attackBool = Animator.StringToHash ("Attack");
    }

    void Update ()
    {
        if (!enemySonar.playerDetected)
        {
            WaypointWalk ();
        }
        else if (enemyHealth.health > 0 && playerHealth.health > 0 &&
```

```

        !enemySonar.obstacleDetected && !enemyHealth.isShocked &&
        readyToHit)
    {
        agent.speed = attackingSpeed;
        agent.acceleration = 2;
        agent.SetDestination (player.transform.position);
    }
    else
    {
        agent.Stop (true);
    }
}

void WaypointWalk ()
{
    if(agent.remainingDistance <= agent.stoppingDistance)
    {
        if(currentWaypointIndex == waypoints.Length - 1)
            currentWaypointIndex = 0;
        else
            currentWaypointIndex++;
        StartCoroutine("WaypointPause", waypointPauseTime);
    }
    agent.SetDestination (waypoints[currentWaypointIndex].position);
}

void OnTriggerStay(Collider other)
{
    if (enemyHealth.health > 0)
    {
        if (other.gameObject == player && playerHealth.health > 0)
        {
            if (readyToHit)
            {
                readyToHit = false;
                //sollte der Spieler in den Enemy laufen
                //bevor dieser vom Enemy entdeckt wurde.
                if(!enemySonar.playerDetected)
                    enemySonar.StopSearching();

                other.gameObject.SendMessage (
                    "ApplyDamage", damageValue,
                    SendMessageOptions.DontRequireReceiver);
                StartCoroutine("SetReadyToHit");
            }
        }
    }
}

IEnumerator SetReadyToHit()
{
    anim.SetBool(attackBool,true);
    yield return new WaitForSeconds(0.5F);
    anim.SetBool(attackBool,false);
    yield return new WaitForSeconds(0.5F);
    readyToHit = true;
}

```

```
IEnumerator WaypointPause(float seconds)
{
    float oldSpeed = agent.speed;
    agent.speed = 0.1F;
    yield return new WaitForSeconds(seconds);
    agent.speed = oldSpeed;
}
```

## ■ 21.7 Eröffnungsszene

Für ein richtiges Spiel benötigen wir natürlich nicht nur das eigentliche Game und dessen Level, sondern auch eine Eröffnungsszene mit einem Startmenü.

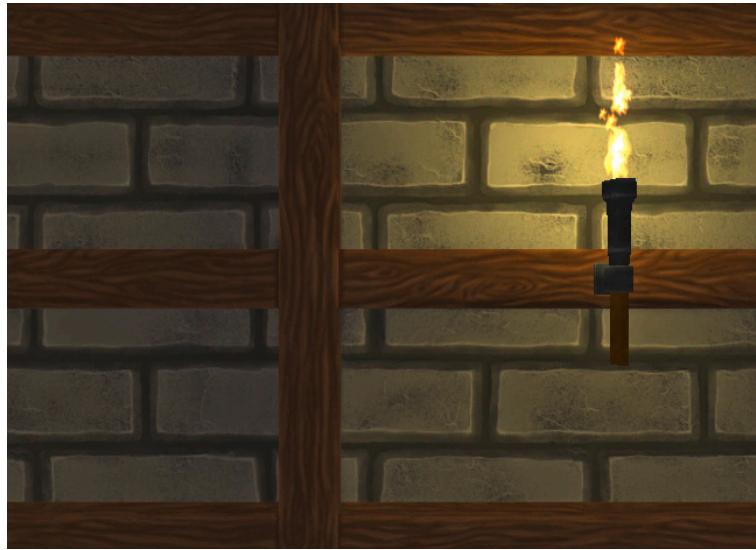
So eine Startszene dient aber nicht nur dem eigentlichen Starten des Spiels. Häufig werden hier auch alte Spielstände geladen oder Variablen Default-Startwerte zugewiesen. Auch in unserer Eröffnungsszene nutzen wir diese, um die zur Verfügung stehenden Lebenspunkte unseres Spielers zuzuweisen.

### 21.7.1 Startszene erstellen

Für eine solche Eröffnungsszene brauchen wir natürlich auch eine eigene Szene, die wir nun auf einfache Weise schnell erstellen können.

1. Öffnen Sie die Dungeon-Szene und speichern Sie diese mit **File/Save Scene as** unter dem neuen Namen „Startmenu“ im Ordner „Scenes“.
2. Ziehen Sie die „Main Camera“ aus dem „Player“ und löschen Sie den Rest des Helden, also das komplette „Player“-Objekt.
3. Positionieren Sie das „Main Camera“-*GameObject* an einem geeigneten Platz, z.B. mit Blick auf eine Wand mit Fackeln.
4. Entfernen Sie alle Skriptkomponenten vom „Game Controller“.
5. Fügen Sie der  *AudioSource* des „Game Controllers“ wieder „introMusic“ zu und aktivieren Sie *Loop* und *Play On Awake*.
6. Entfernen Sie alle Kind-Objekte vom „GUI“-Container außer „MessageText“.
7. Löschen Sie alle anderen Elemente, die nicht zum eigentlichen Dungeon gehören, also alle „Bat“-Instanzen, alle „Stone“-Instanzen, alle „BucketEmpty“-Instanzen, alle „Gate“-Instanzen, den „IronKey“, den „QuestTrigger“, das „Waterdrop“-Partikelsystem, das „Inventory“-Objekt und das „TooltipText“-Objekt.

Am Ende sollten Sie jetzt eine Szene erhalten, die Sie starten können, ohne dass großartig noch was passiert.



**Bild 21.31** Startmenü-Szenenbild

8. Wenn dies nun einwandfrei funktioniert, fügen Sie nun beide Szenen den *Build Settings* zu. Starten Sie hierfür jede Szene und führen dann dort in den *Build Settings Add Current* aus.
9. Schieben Sie als Letztes in den *Build Settings* die neue Szene „Startmenu“ nach oben, damit diese den Index 0 besitzt und die Szene „Dungeon“ den Index 1 hat.

## 21.7.2 Startmenü erstellen

Nun wollen wir noch ein kleines Menü-Skript erstellen, über das wir das Spiel starten und auch beenden können. Außerdem wollen wir über einen dritten Button einen kleinen Beschreibungstext anzeigen lassen, der Informationen zum Spiel darstellt. Hierfür nutzen wir wieder das „MessageText“-Objekt, das wir in der Dungeon-Szene schon des Öfteren genutzt haben.

Erstellen Sie hierfür ein Skript mit dem Namen *MainMenu*, das Sie dann dem „Game Controller“ zufügen.

### 21.7.2.1 Startmenü mit OnGUI

Zum Definieren der GUI-Objekte benötigen wir in der *OnGUI*-Variante noch eine Variable vom Typ *Rect*.

In der Start-Methode initialisieren wir dann die Variablen bzw. bauen die Referenzen zu diesen auf. Möchten Sie dabei die Buttons in einer anderen Größe haben, brauchen Sie hier einfach nur den Width- und/oder den Height-Wert entsprechend anzupassen.

**Listing 21.80** Variablen von MainMenu

```
private GUIText messageText;
private Rect rect;
void Start () {
    messageText = GameObject.FindGameObjectWithTag("Message").guiText;
    messageText.text = "";
    rect = new Rect(50,50,200,30);
}
```

Zunächst definieren wir in der OnGUI-Methode die drei Buttons und rufen bei jedem eine eigene Methode auf, in der wir dann den dazugehörigen Code schreiben, der ausgeführt werden soll.

Für die Positionierung der Controls weisen wir beim ersten Button die fixe Y-Position 50 zu und erhöhen diese nun bei jedem weiteren Button um ebenfalls 50, damit diese am Ende untereinander dargestellt werden.

**Listing 21.81** OnGUI-Methode von MainMenu

```
void OnGUI()
{
    rect.y = 50;
    if (GUI.Button(rect,"Start"))
    {
        StartGame ();
    }

    rect.y += 50;
    if (GUI.Button(rect,"Steuerung und Info"))
    {
        ShowInfo ();
    }

    rect.y += 50;
    if (GUI.Button(rect,"Beenden"))
    {
        CloseGame ();
    }
}
```

In der Methode StartGame starten wir nun mit LoadLevel die Szene 1. Außerdem schreiben wir mit der *PlayerPrefs*-Klasse den Startwert für den Parameter „LPs“ weg, den wir im *LifePointController* der Dungeon-Szene für die Lebenspunkte nutzen. Außerdem löschen wir den Inhalt von messageText, falls dort Inhalt angezeigt wird.

**Listing 21.82** StartGame-Methode von MainMenu

```
void StartGame()
{
    messageText.text = "";
    PlayerPrefs.SetInt("LPs",3); //Startwert
    Application.LoadLevel(1);
}
```

In der ShowInfo-Methode, die durch den „Steuerung und Info“-Button aufgerufen wird, weisen wir lediglich einen Infotext der messageText-Variablen zu.

**Listing 21.83** ShowInfo-Methode von MainMenu

```
void ShowInfo ()
{
    messageText.text = "Du wurdest in ein Kellergewölbe verbannt.\n" +
        "Monster und Rätsel warten dort auf Dich.\n\n" +
        "Zur Steuerung:\n" +
        "A/D: Links/Rechts gehen\n" +
        "W/S: Vorwärts/Rückwärts gehen\n" +
        "Q/E: Links/Rechts drehen\n" +
        "Linke Maustaste: Gegenstände aufnehmen\n" +
        "Rechte Maustaste/Leertaste: Angreifen";
}
```

Als Letztes kommt die CloseGame-Methode, die die Methode Quit der Application-Klasse aufruft und damit das Spiel schließt.

**Listing 21.84** CloseGame-Methode von MainMenu

```
void CloseGame ()
{
    Application.Quit();
}
```

Beachten Sie, dass die Quit-Methode nicht im Editor getestet werden kann. Zum Testen dieser Funktionalität sollten Sie das Game als Stand-alone *builden* bzw. erstellen.



**Bild 21.32** Startmenü

Starten Sie nun das Spiel mit der Szene „Startmenu“, sollten Sie das Menü sehen, aus dem Sie dann auch das Spiel starten können. Wenn der Spieler nun stirbt und alle Leben verloren hat, gelangt er nach dem Drücken auf die **[Space]**-Taste (oder **[Escape]** oder **[Return]**) wieder in das Menü.

### 21.7.2.2 MainMenu.cs

Das komplette *MainMenu*-Skript, das Sie dem „Game Controller“ zufügen sollten, sieht am Ende wie im folgenden Listing aus.

**Listing 21.85** MainMenu.cs

```
using UnityEngine;
using System.Collections;

public class MainMenu : MonoBehaviour {

    private GUIText messageText;
    private Rect rect;
    // Use this for initialization
    void Start () {
        messageText = GameObject.FindGameObjectWithTag("Message").guiText;
        messageText.text = "";
        rect = new Rect(50,50,200,30);
    }

    void OnGUI()
    {
        rect.y = 50;
        if (GUI.Button(rect,"Start"))
        {
            StartGame ();
        }

        rect.y += 50;
        if (GUI.Button(rect,"Steuerung und Info"))
        {
            ShowInfo ();
        }

        rect.y += 50;
        if (GUI.Button(rect,"Beenden"))
        {
            CloseGame ();
        }
    }

    void StartGame ()
    {
        messageText.text = "";
        PlayerPrefs.SetInt("LPs",3); //Startwert
        Application.LoadLevel(1);
    }

    void ShowInfo ()
    {
```

```

messageText.text = "Du wurdest in ein Kellergewölbe verbannt.\n" +
    "Monster und Rätsel warten dort auf Dich.\n\n" +
    "Zur Steuerung:\n" +
    "A/D: Links/Rechts gehen\n" +
    "W/S: Vorwärts/Rückwärts gehen\n" +
    "Q/E: Links/Rechts drehen\n" +
    "Linke Maustaste: Gegenstände aufnehmen\n" +
    "Rechte Maustaste/Leertaste: Angreifen";
}

void CloseGame ()
{
    messageText.text = "";
    if (Application.isWebPlayer)
        messageText.text = "Diese Funktion wird " + " " +
            "im Webplayer nicht unterstützt";
    else
        Application.Quit();
}

```

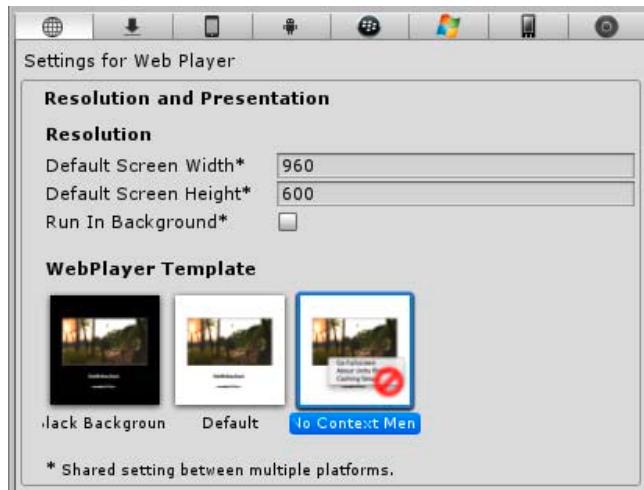
## ■ 21.8 Web-Player-Anpassungen

Wollen Sie nun das Spiel veröffentlichen, dann ist natürlich zuerst einmal wichtig, dass Sie alle zum Spiel gehörenden Szenen der „Scenes in Build“-Liste zufügen. In unserem Fall gibt es aber gerade beim Web-Player-Build noch zwei weitere Besonderheiten, die Sie beachten und entsprechend anpassen sollten.

### 21.8.1 Web-Player-Template ändern

Die erste Besonderheit betrifft das Attackieren. Da der Spieler das Werfen der Steine nicht nur über die **[Space]**-Taste ausführen kann, sondern auch über die rechte Maustaste, kommt es hier zu einem Konflikt mit einer Standardfunktion von Unity. Über die rechte Maustaste ruft der Spieler im Webplayer normalerweise ein Kontextmenü auf, mit dem er unter anderem in die Vollansicht wechseln kann.

Damit dieses Kontextmenü nun nicht jedes Mal aufpoppt, wenn der Spieler einen Stein werfen will, können wir nun dieses Kontextmenü unterdrücken. Gehen Sie hierfür über **Edit/Project Settings/Player** in die *Player Settings* des Web-Players und wählen dort im Bereich *Webplayer Template* „No Context Menu“ aus.

**Bild 21.33**

„No Context Menu“–  
Einstellung in den Player  
Settings

## 21.8.2 Quit-Methode im Web-Player abfangen

Als zweiten Punkt sollten Sie beachten, dass die `Quit`-Methode der `Application`-Klasse nicht im Web-Player funktioniert. Entweder Sie entfernen den kompletten Button aus dem `MainMenu`-Skript oder Sie führen in dem Fall alternativen Code aus. Dabei können Sie mithilfe der `isWebPlayer`-Eigenschaft ermitteln, ob das Game aktuell im Web-Player ausgeführt wird oder woanders.

**Listing 21.86** Web-Player-Abfrage im Beenden-Button

```
if (GUI.Button(rect,"Beenden"))
{
    messageText.text = "";
    if (Application.isWebPlayer)
        messageText.text = "Diese Funktion wird " +
            "im Webplayer nicht unterstützt";
    else
        Application.Quit();
}
```

# ■ 21.9 Umstellung auf uGUI

Wenn Sie Unity in der Version 4.6 oder höher besitzen, werden Sie bereits das neue *uGUI*-System nutzen können. Da sich dieses aber noch im Beta-Stadium befand, als dieses Buch geschrieben wurde, bin ich in den vorangegangenen Abschnitten aus Kompatibilitätsgründen zunächst von den herkömmlichen GUI-Möglichkeiten ausgegangen, sprich von den *GUI-Elements* und der *OnGUI*-Programmierung.

Möchten Sie das Spiel nun auf *uGUI* umstellen, müssen Sie natürlich zunächst die *UIElements* aus den beiden Szenen entfernen und diese durch *uGUI*-Controls ersetzen. Außerdem müssen die Zugriffe in den Skripten entsprechend angepasst werden. Im Folgenden wollen wir diese anzupassenden Dinge einmal durchgehen.



### Download des Beispiel-Games mit einer uGUI-Oberfläche

Auf meiner Website <http://www.hummelwalker.de/buch-zusatzmaterial/> können Sie das komplette Beispiel-Game, umgestellt auf *uGUI*, herunterladen. Das Passwort finden Sie im Kapitel 1 („Einleitung“) dieses Buches.

#### 21.9.1 Skriptanpassungen

Ersetzen Sie als Erstes in den Skripten *Inventory*, *InventoryItem*, *ItemProperties*, *MainMenu* und *PlayerHealth*, *HoverEffects*, *EPController*, *LifePointController*, *Shooting*, *WaterQuest* die folgenden Typen, wozu natürlich auch die Übergabeveriablen der Methoden gehören:

- *GUIText* durch *Text*
- *GUITexture* durch *Image*
- *Texture2D* durch *Sprite* (außer im *HoverEffects*-Skript)

Damit die Typen *Text* und *Image* überhaupt in den Skripten zur Verfügung stehen, müssen Sie vorher noch den Namespace *UnityEngine.UI* mit *using* in diese Skripte einbinden.

Beachten Sie auch die *GetComponent*-Zuweisungen, die Sie mit diesen Datentypen in den Skripten machen. Diese müssen Sie natürlich ebenfalls umstellen.

##### **Listing 21.87** Beispiel: Ersetzen der GetComponent-Zuweisungen

```
//messageText = GameObject.FindGameObjectWithTag ("Message").
//    GetComponent<GUIText>(); //UIElement
messageText = GameObject.FindGameObjectWithTag ("Message").
    GetComponent<Text>(); //uGUI
```

#### Inventory-Anpassungen

Nach diesen allgemeinen Änderungen müssen Sie im *Inventory*-Skript die Zugriffe auf die Variablen auf die neuen Typen anpassen.

Vor allem bei der *Image*-Komponente unterscheiden sich die Eigenschaften von der *GUITexture*-Komponente. Aufgrund der etwas anderen Reaktion dieser Komponente werden wir deshalb in der *UpdateView*-Methode zuerst alle Grafikelemente ausblenden. Danach aktivieren wir nur die, die auch Inhalte (Grafiken) haben sollen, und weisen diese entsprechend zu.

##### **Listing 21.88** uGUI-Variante von *UpdateView* aus dem *Inventory*-Skript

```
void UpdateView()
{
    int index = 0;
```

```

int guiCount = guiItemTextures.Length;

for(int i = 0; i < guiCount; i++)
{
    //guiItemTextures[i].texture = null; //GUIElement
    guiItemTextures[i].enabled = false; //uGUI
    guiItemQuantity[i].text = "";
}
foreach(KeyValuePair<string,ItemProperties> current in items)
{
    //guiItemTextures[index].texture = current.Value.texture;//GUIElement
    guiItemTextures[index].sprite = current.Value.texture; //uGUI
    guiItemTextures[index].enabled = true;
    guiItemQuantity[index].text = current.Value.quantity.ToString();
    index++;
}
}

```

## PlayerHealth-Anpassungen

Als Nächstes werden wir die Darstellung des Gesundheitsbalkens im *PlayerHealth*-Skript auf die neuen Komponentenmöglichkeiten umstellen. Hier bietet uns die *Image*-Komponente ganz andere Möglichkeiten als noch die *GUITexture*-Komponente, auf die wir aber noch im Abschnitt „Controls platzieren“ zu sprechen kommen.

In dem *PlayerHealth*-Skript können wir die gesamten Berechnungen für die Länge des Balkens löschen bzw. auskommentieren (siehe Listing 21.89) und stattdessen in der *UpdateView*-Methode den Gesundheitszustand als Prozentwert der Variablen *fillAmount* zuweisen (siehe Listing 21.90). Den Rest macht die *Image*-Komponente mithilfe des *Image Type* „Filled“.

### Listing 21.89 uGUI-Variante der Start-Methode des PlayerHealth-Skripts

```

void Start ()
{
    //messageText = GameObject.FindGameObjectWithTag ("Message").
    // GetComponent<GUIText>(); //GUIElement
    messageText = GameObject.FindGameObjectWithTag ("Message").
        GetComponent<Text>(); //uGUI
    lifePointController = GameObject.FindGameObjectWithTag("GameController").
        GetComponent<LifePointController>();
    //guiRect = new Rect (healthGui.pixelInset); //GUIElement
    //guiMaxWidth = guiRect.width; //GUIElement
    maxHealth = health;
    UpdateView();
}

```

### Listing 21.90 uGUI-Variante von UpdateView des PlayerHealth-Skripts

```

void UpdateView()
{
    if(healthGui != null) {
        //guiRect.width = guiMaxWidth * health/maxHealth; //GUIElement
        //healthGui.pixelInset = guiRect; //GUIElement
        healthGui.fillAmount = health/maxHealth; //uGUI
    }
}

```

## HoverEffects-Anpassungen

Für unsere „Dungeon“-Szene müssen wir jetzt nur noch das *HoverEffects*-Skript anpassen.

Während bei der *GUITexture*-Komponente die Position über den *pixelOffset* zugewiesen wurde, setzen wir die *Image*-Komponente über die Position des *RectTransforms*. Da diese vom Typ *Vector3* ist, müssen wir auch den Datentyp ändern und die Z-Position zuweisen. Damit sieht die *OnMouseEnter*-Methode im *HoverEffects*-Skript wie folgt aus:

```
void OnMouseEnter()
{
    if (clickableCursor != null)
        Cursor.SetCursor(clickableCursor,hotSpot,CursorMode.Auto);
    msgBox.text = tooltipText;
    /* GUIElements -B
    */
    Vector2 pos;
    pos.x = Input.mousePosition.x ;
    pos.y = Input.mousePosition.y + 20; // um Text etwas hoher anzuzuzeigen.
    msgBox.pixelOffset = pos;
    */
    /* GUIElements -E

    //uGUI -B
    Vector3 pos;
    pos.x = Input.mousePosition.x ;
    pos.y = Input.mousePosition.y + 20; // um Text etwas hoher anzuzuzeigen
    pos.z = 0;
    msgBox.rectTransform.position = pos;
    //uGUI -E
}
```

## MainMenu-Anpassungen

Die Anpassungen in der „Startmenu“-Szene sind gegenüber der Dungeon-Szene kinderleicht.

Im *MainMenu*-Skript müssen Sie lediglich alle Zugriffe von *GUIText* auf *Text* umstellen (den Namespace *UnityEngine.UI* nicht vergessen!) und die *rect*-Variable löschen. Diese brauchen wir nicht mehr in der *OnGUI*-Methode, da auch diese nicht mehr benötigt wird und komplett gelöscht werden kann. Alternativ können Sie die *OnGUI*-Methode wie auch den restlichen nicht mehr benötigten Code auch einfach auskommentieren.

### Listing 21.91 Anpassungen im MainMenu-Skript

```
//private GUIText messageText; //GUIElements
private Text messageText; //uGUI
//private Rect rect; //OnGUI/GUIElements

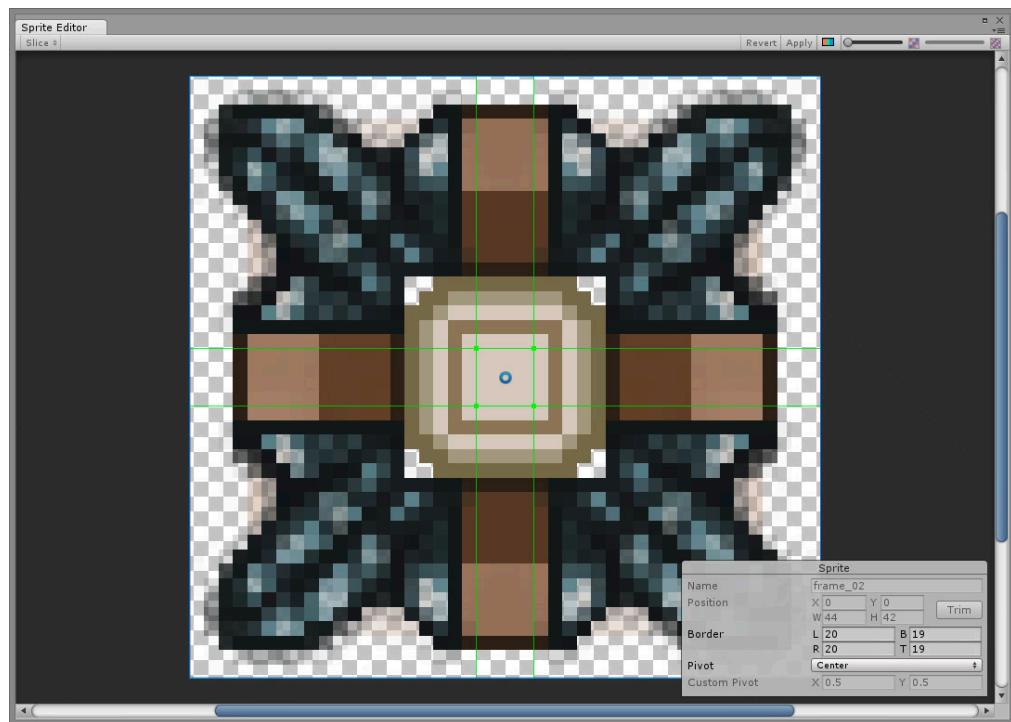
void Start () {
    //messageText = GameObject.FindGameObjectWithTag("Message").
    //             GetComponent<Text>(); //uGUI
    messageText.text = "";
    //rect = new Rect(50,50,200,30); //OnGUI/GUIElements
}
```

## 21.9.2 Controls erstellen

Nachdem wir die Skripte angepasst haben, können wir uns nun um die Controls kümmern. Da uGUI im Gegensatz zu den anderen Systemen Sprites nutzt, müssen wir nur vor dem Platzieren und Parametrisieren der Controls erst einmal alle Grafiken in *Sprites* umwandeln.

Hierfür weisen wir zunächst den betroffenen Grafiken in den *Import Settings* den *Texture Type* „Sprite“ sowie den *Sprite Mode* „Single“ zu. Dies sind die Grafiken „bucket“, „frame\_02“, „healthbar“, „inventory-tile“, „key“, „lifepointHeart“ und „stone“.

Eine weitere *Sprite*-Editierung benötigt lediglich die Grafik „frame\_02“. Öffnen Sie hierfür den *Sprite Editor* in den *Import Settings* und setzen Sie die *Border*-Parameter (im Bild grün dargestellt) auf folgende Werte: L 20, R 20, B 19 und T 19.



**Bild 21.34** Sprite-Border-Einstellungen bei frame\_02

Anschließend müssen Sie sich um die Inventar-*Prefabs* kümmern und den *texture*-Variablen der *InventoryItem*-Komponenten die neuen *Sprite*-Elemente zuweisen. Achten Sie darauf, dass Sie immer die aufgeklappten Unterelemente zuweisen, nie die Haupt-Assets selber.

### 21.9.2.1 InGame-Controls mit uGUI

Jetzt kümmern wir uns um die Controls unserer Benutzerschnittstelle, die während des Spiels angezeigt werden sollen.

### TooltipText

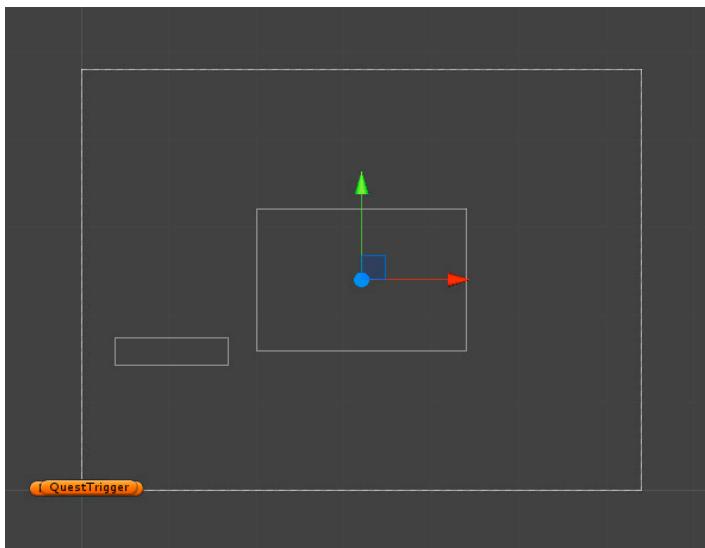
Öffnen Sie die Szene „Dungeon“ und fügen Sie der Szene über **GameObject/Create UI/Text** ein *Text-GameObject* zu.

- Geben Sie dem *GameObject* den Namen „TooltipText“.
- Weisen Sie den *Tag* „Tooltip“ zu.
- Geben Sie ihm die *Anchors* (center, middle), wobei die erste Angabe für die horizontale Achse gilt und die zweite für die vertikale.
- Löschen Sie den *Default-Text* der *Text*-Komponente.
- Übernehmen Sie den *Font Style* „Normal“.
- Weisen Sie *Font Size* den Wert 14 zu.
- Weisen Sie beiden *Alignment*-Werten die Center-Positionen zu.
- *Color* weisen Sie (251,251,251,255) zu und fügen Sie diese den *Presets* im Color-Dialog zu.
- *Effect Style* geben Sie „Outline“.
- *Effect Color* weisen Sie (0,0,0,255) zu.
- *Effect Distance* erhält (1,1).

### MessageText

Im Weiteren kopieren Sie dieses *Text-Control* und verändern folgende Dinge:

- Nennen Sie das Control in „MessageText“ um.
- Weisen Sie den *Tag* „Message“ zu.
- Positionieren Sie die *Anchors* (stretch, stretch).
- Weisen Sie im *RectTransform* *Left* 200.5, *Right* 200.5, *Top* 159 und *Bottom* 159 zu.
- Nutzen Sie den *Font Style* „Bold“.
- Nehmen Sie die *Font Size* 16.



**Bild 21.35**  
Die Text-Controls  
MessageText und  
TooltipText

## Spielerwerte-Panel

Im nächsten Schritt wollen wir nun die Spielerwerte darstellen, sprich Leben, Gesundheit und die Erfahrungspunkte. Im Gegensatz zur *UIElements*-Version wollen wir dieses Mal mit einem zusätzlichen Panel arbeiten, das Controls sowohl logisch als auch visuell bündeln soll. Fügen Sie hierfür der Szene ein *Panel-GameObject* zu und führen folgende Änderungen durch:

- Nennen Sie das *GameObject* in „PropertyPanel“ um.
- Geben Sie ihm die *Anchors* (left, top), wobei die erste Angabe für die horizontale Achse gilt und die zweite für die vertikale.
- Weisen Sie im *RectTransform Pos X* 130, *Pos Y* -84, *Width* 197 und *Height* 105 zu.
- Übergeben Sie der *Source Image*-Eigenschaft den *Sprite* „frame\_02“.

Da wir ein solches Panel auch für unser Inventar nutzen wollen, kopieren wir dieses, bevor wir weitermachen, und verändern bei der Kopie folgende Werte:

- Geben Sie der Kopie den Namen „InventoryPanel“.
- Geben Sie ihm die *Anchors* (left, bottom).
- Weisen Sie im *RectTransform Pos X* 130, *Pos Y* 84, *Width* 197 und *Height* 105 zu.

Nun wenden wir uns aber wieder den Spielerwerten zu. Erzeugen Sie für den Gesundheitsbalken ein *Image-GameObject* und fügen es dem „PropertyPanel“ als Kind-Objekt zu.

- Nennen Sie das *Image-GameObject* in „HealthBar“ um.
- Geben Sie ihm die *Anchors* (left, top).
- Weisen Sie im *RectTransform Pos X* 96.3, *Pos Y* -29.6, *Width* 128 und *Height* 16 zu.
- Übergeben Sie der *Source Image*-Eigenschaft den *Sprite* „healthbar“.
- Als *Image Type* nehmen Sie „Filled“. Hierzu nehmen Sie als *Fill Method* „Horizontal“ und *Fill Origin* „Right“.
- Das fertige Control weisen Sie nun der *HealthGui*-Variablen des *PlayerHealth*-Skriptes zu.



**Bild 21.36**

PropertyPanel mit HealthBar

Im Weiteren wollen wir uns nun um die Anzeige der Lebensanzahl kümmern. Hierfür erzeugen Sie zunächst ein *Image-GameObject* für die Hintergrundgrafik und fügen es ebenfalls dem „PropertyPanel“ als Kind-Objekt zu.

- Nennen Sie das *GameObject* in „LPs“ um.
- Geben Sie ihm die *Anchors* (left, top).

- Weisen Sie im *RectTransform Pos X* 48.3, *Pos Y* -67.56, *Width* 32 und *Height* 32 zu.
- Übergeben Sie der *Source Image*-Eigenschaft den *Sprite* „lifepointHeart“.
- Als *Image Type* wählen Sie „Simple“.

Nun erzeugen Sie ein neues *Text-GameObject* und fügen dieses dem obigen „LPs“-*GameObject* als Kind-Objekt zu.

- Nennen Sie das *GameObject* in „LPText“ um.
- Weisen Sie die *Anchors* (stretch, stretch) zu.
- Übergeben Sie im *RectTransform Left* 0, *Right* 0, *Top* 0 und *Bottom* 3.
- Nutzen Sie den *Font Style* „Bold“.
- Nehmen Sie die *Font Size* 18.
- Weisen Sie beiden *Alignment*-Werten die Center-Positionen zu.
- *Color* weisen Sie (251,251,251,255) zu. Nutzen Sie hierfür am einfachsten den *Presets*-Wert im Color-Dialog, den wir am Anfang erstellt haben.
- *Effect Style* weisen Sie „Outline“ zu.
- *Effect Color* weisen Sie (0,0,0,255) zu.
- *Effect Distance* weisen Sie (1,1) zu.
- Weisen Sie „LPText“ der Variablen lpText vom *LifePointController*-Skript zu, das sich am „Game Controller“ befindet.

Zum Schluss benötigen wir jetzt noch ein *Text-Control*, um die Erfahrungspunkte anzuziegen. Da wir fast die identischen Einstellungen benötigen wie beim vorherigen *Text-Control*, kopieren wir dieses einfach und ziehen es aus dem „LPs“-Objekt heraus, damit es sich auf der gleichen Hierarchieebene wie „LPs“ unter dem „PropertyPanel“ befindet.

- Nennen Sie das *GameObject* in „EPText“ um.
- Weisen Sie die *Anchors* (left, top) zu.
- *Alignments* weisen Sie Left und Center zu, sodass der Text linksbündig ausgerichtet wird.
- Weisen Sie im *RectTransform Pos X* 131.37, *Pos Y* -65.9, *Width* 58.2 und *Height* 28.8 zu.
- „EPText“ weisen Sie der Variablen epText vom *EPController*-Skript zu, das sich ebenfalls am „Game Controller“ befindet.



**Bild 21.37** Fertiges PropertyPanel mit Objektstruktur

## Inventory-Panel

Im nächsten Schritt wollen wir uns um das „InventoryPanel“ kümmern, das wir bereits weiter oben erstellt haben. Hierfür erzeugen wir aber zunächst ein neues *Image-GameObject*, fügen es aber erst einmal noch nicht dem Panel zu.

- Nennen Sie das *GameObject* in „InventoryTile“ um.
- Geben Sie ihm die *Anchors* (left, bottom).
- Weisen Sie der *Source Image*-Eigenschaft den *Sprite* „inventory-tile“ zu.
- Als *Image Type* wählen Sie „Simple“ und drücken Sie den **Set Native Size**-Button, um die Größe des *RectTransforms* auf die des *Sprites* zu setzen.

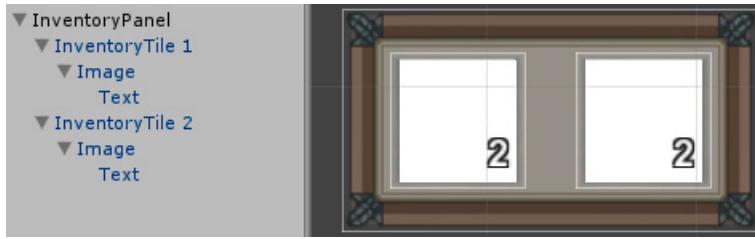
Fügen Sie „InventoryTile“ ein neues *Image*-Objekt als Kind-Objekt zu und weisen Sie diesem folgende Eigenschaften zu:

- Geben Sie ihm die *Anchors* (stretch, stretch).
- Weisen Sie im *RectTransform* *Left* 4.6, *Right* 4.6, *Top* 3.2 und *Bottom* 3.2 zu.
- Als *Image Type* wählen Sie „Simple“.
- Weisen Sie der *Source Image*-Variablen keine Grafik zu.

Nun erzeugen Sie noch ein *Text-GameObject* und fügen dieses dem *Image-GameObject* als Kind-Objekt zu.

- Weisen Sie die *Anchors* (left, top) zu.
- Übergeben Sie im *RectTransform* *Pos X* 54.7, *Pos Y*-69, *Width* 24.6 und *Height* 25.4.
- Nutzen Sie den *Font Style* „Bold“.
- Nehmen Sie die *Font Size* 18.
- Weisen Sie den *Alignment*-Werten *right* und *bottom* zu, damit dieses unten rechts platziert wird.
- *Color* weisen Sie (251,251,251,255) zu. Nutzen Sie auch hier wieder am einfachsten den *Presets*-Wert im Color-Dialog, den wir zu Anfang erstellt haben.
- *Effect Style* weisen Sie „Outline“ zu.
- *Effect Color* legen Sie mit (0,0,0,255) fest.
- *Effect Distance* weisen Sie (1,1) zu.
- Ziehen Sie das *GameObject* „InventoryTile“ in den „Prefabs“-Ordner, um ein Prefab zu erstellen.
- Benennen Sie das Objekt in „InventoryTile 1“ um.
- Ziehen Sie das *GameObject* auf das „InventoryPanel“, damit es ein Kind-Objekt von diesem wird.
- Weisen Sie im *RectTransform* *Pos X* 54.8, *Pos Y* 52.5, *Width* 64 und *Height* 64 zu.
- Danach kopieren Sie das Objekt und nennen die Kopie in „InventoryTile 2“ um.
- Weisen Sie dem *RectTransform* von „InventoryTile 2“ *Pos X* 143, *Pos Y* 52.5, *Width* 64 und *Height* 64 zu.

Nachdem Sie nun das grafische Inventarsystem erstellt haben, weisen Sie nun die beiden *Text-GameObjects* sowie die beiden *Image*-Objekte der *InventoryTiles* den beiden Arrays



**Bild 21.38** Fertiges InventoryPanel mit Objektstruktur

guiItemTextures und guiItemQuantity vom *Inventory*-Skript zu. Achten Sie darauf, dass Sie dem „Element 0“ der Arrays immer das jeweilige Objekt von „InventoryTile 1“ zuweisen und „Element 1“ immer das andere.

Übrigens, wenn Sie das Inventar vergrößern möchten, brauchen Sie lediglich „InventoryPanel“ größer zu ziehen und das „InventoryTile“ entsprechend häufig diesem zuzufügen. Fügen Sie nun noch die *Text*- und *Image*-Komponenten den beiden obigen Arrays hinzu, schon sind Sie fertig.

### 21.9.2.2 Hauptmenü-Controls mit uGUI

Für die Startszene benötigen wir insgesamt drei Buttons und ein *Text-GameObject*.

#### MessageText

Zuerst kümmern wir uns um das *Text*-Objekt. Dafür fügen wir der Szene ein neues hinzu und geben ihm folgende Eigenschaften:

- Zunächst benennen wir das Objekt um in „MessageText“.
- Danach geben wird dem Objekt den Tag „Message“.
- Als Anchors nutzen wir (center, middle).
- Weiter weisen wir dem RectTransform Pos X 0, Pos Y 0, Pos Z 0, Width 259 und Height 261 zu.
- Als *Font Style* nehmen wir „Bold“.
- *Font Size* weisen wir 16 zu.
- Als *Alignment* bestimmen wir bei beide Center.
- *Color* weisen Sie wieder (251,251,251,255) zu.
- *Effect Style* legen Sie auf „Outline“ fest.
- *Effect Color* weisen Sie (0,0,0,255) zu.
- *Effect Distance* legen Sie auf (1,1).

#### Buttons

Danach wollen wir uns um die Buttons kümmern. Hierfür fügen wir einen einzelnen Button hinzu, aus dem wir später wieder ein Prefab erstellen. An diesem Button machen wir nun folgende Anpassungen:

- Wir benennen den Button in „MenuButton“ um.

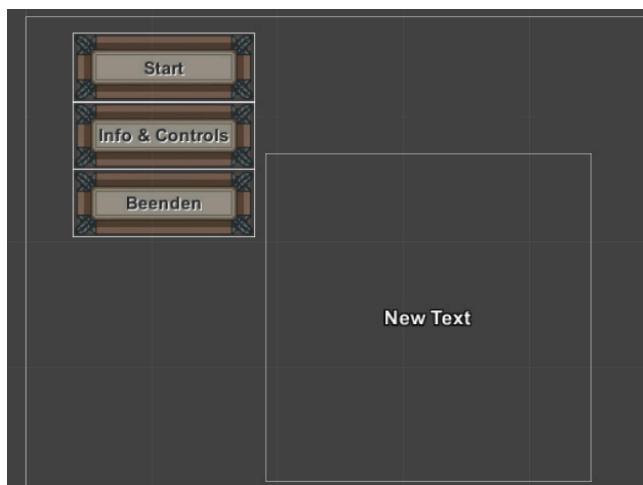
- Als *Anchors* nutzen wir (left, top).
- Weiter weisen wir dem *RectTransform Pos X 37, Pos Y -13.5, Pos Z 0, Width 145* und *Height 55* zu.
- Als Source Image nehmen wir den Sprite „frame\_02“.
- *Color* belassen wir auf dem halbtransparenten Weiß.
- Als *Image Type* nehmen wir *Sliced*.
- Als *Alignment* weisen wir beidem Center zu.
- Als *Transition* nutzen wir „ColorTint“.
- *Normal Color* weisen Sie wieder (255,255,255,255) zu.
- *Highlighted Color* bekommt die Werte (255,235,191,255).
- *Pressed Color* weisen Sie wieder (210,172,86,255) zu.
- *Disabled Color* belassen wir auf (128,128,128,255).

Nun gehen Sie auf das „Label“-Kind-Objekt und passen auch dessen Eigenschaften an.

- Als *Font Style* nehmen wir „Bold“.
- *Font Size* weisen wir 14 zu.
- Als *Alignment* bestimmen wir bei beiden Center.
- *Color* weisen Sie wieder (43,43,43,255) zu.
- *Effect Style* erhält „Shadow“.
- *Effect Color* weisen Sie (0,0,0,255) zu.
- *Effect Distance* legen Sie mit (1,1) fest.

Danach erzeugen Sie ein *Prefab* aus dem Control, in dem Sie es in den „Prefabs“-Ordner ziehen. Jetzt können Sie das Control zwei Mal kopieren und untereinander platzieren. Hierbei sollte Unity mit dem Snapping-Verhalten der Controls unterstützen.

Danach können Sie die *Text*-Eigenschaften der „Label“-Kind-Objekte der Buttons ändern und den Labels die Textinhalte „Start“, „Info & Controls“ und „Beenden“ zuweisen.



**Bild 21.39**  
uGUI-Controls der  
Startmenü-Szene

Zum Schluss wollen wir den Buttons noch Funktionalität verpassen. Hierfür gibt es zwei unterschiedliche Möglichkeiten, die ich Ihnen beide kurz vorstellen möchte.

1. Sie deklarieren die drei Methoden `StartGame`, `ShowInfo` und `CloseGame` im `MainMenu`-Skript als *public*-Methoden. Nun weisen Sie wieder den „`onClick`“-Events den „Game Controller“ zu, wo Sie aber nun direkt auf die Methoden verweisen können, z.B. `Main.Menu.StartGame()`.
2. Sie lassen die Methoden *private* und weisen den „`onClick`“-Events der Buttons den „Game Controller“ zu, wo Sie dann `MainMenu.SendMessage` stattdessen auswählen. Im rechten Feld geben Sie nun noch, je nach Button, entweder die Methode `StartGame`, `ShowInfo` oder `CloseGame` manuell ein.

Egal für welche Variante Sie sich entscheiden, beide sind nicht sonderlich aufwendig. Danach können Sie das Spiel starten und testen.

## ■ 21.10 So könnte es weitergehen

Da dieses Spiel natürlich keinen kompletten Dungeon Crawler darstellt und das Erstellen eines kompletten einfach den Rahmen sprengen würde, endet an dieser Stelle die Entwicklung des Beispiel-Games. Nichtsdestotrotz können Sie es jetzt noch weiter entwickeln oder nach Belieben umbauen. Sowohl in dem Beispiel wie auch im restlichen Buch haben Sie alle notwendigen Basics kennengelernt, um aus diesem kleinen Spiel ein spannendes Adventure-Game zu entwickeln, das den Spieler auch für eine längere Zeit fesselt und begeistert.

Zunächst sollten Sie sich hierfür eine echte Hauptaufgabe für den Helden überlegen, die ihn durch das gesamte Spiel leiten sollte. Dies kann z.B. das Finden des Dungeon-Ausgangs sein oder ein Hauptgegner, der getötet werden soll. Danach können Sie das Level um weitere Gänge und Hallen erweitern und dieses um weitere Quests und Gegner ergänzen. Und auch weitere Fallgatter, sowohl verschlossene als auch offene, dürfen natürlich nicht fehlen.

Bei den Rätseln können Sie sich am Anfang an der Logik der *WaterQuest* orientieren und den Wasserbehälter durch andere Items ersetzen. Sie können sich aber auch komplett eigene ausdenken. Sollten Sie sich entscheiden, neben den bereits bekannten Fledermäusen noch weitere Gegner dem zuzufügen, benötigen Sie hierfür natürlich eigene 3D-Modelle sowie Animationen und vielleicht auch andere KI-Skripte. Hierfür benötigen Sie dann Know-how im 3D-Modelling. Alternativ können Sie sich aber auch z.B. im Asset Store umschauen, wo es hierfür ebenfalls eine große Auswahl an Modellen (mit Animationen) gibt.

Nicht zuletzt können Sie natürlich auch zusätzliche Szenen anlegen, in die der Spieler wechselt kann. Das Springen in eine andere Szene ist hierbei genauso einfach umzusetzen wie das Auslösen des Questgeber-Dialogs: Sie positionieren einfach in der Szene einen Trigger-Collider, z.B. vor einem Tor, einer Leiter oder einem ähnlichen Objekt, der dann ein Skript besitzt, das in einer `OnTriggerEnter`-Methode die nächste Szene startet. Für den Spieler wirkt es dann so, als wenn das Objekt, z.B. die Leiter, zur nächsten Ebene führt.

Allerdings müssen Sie bei solchen Szenenwechseln beachten, dass dann auch die Inhalte des Inventarsystems und die Erfahrungspunkte szeneübergreifend erhalten bleiben. Ansonsten ist das Inventar nach dem Wechsel in eine neue Szene komplett leer, was sicherlich nicht gewünscht sein dürfte. Einige Konzepte hierzu hatte ich Ihnen bereits im Kapitel „Skript-Programmierung“ vorgestellt. Für die Erfahrungspunkte eignet sich z.B. das gleiche *PlayerPrefs*-Verfahren wie bei den Lebenspunkten. Für das Inventarsystem empfehle ich da eher ein Verfahren mit der *DontDestroyOnLoad*-Methode. Das benötigt also schon die eine oder andere Anpassung an den bestehenden Skripten.

Aber auch das Umbauen des gesamten Spiels zu einem *Third Person Game* ist genauso denkbar wie das Verändern der Kulissen und der Spielatmosphäre zu einem Science-Fiction- oder Horror-Game. Ihrer Kreativität sind da keine Grenzen gesetzt.

Lizenziert für gregwyss@hotmail.com.

© 2014 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

# 22

# Der Produktionsprozess in der Spieleentwicklung

Nachdem Sie nun das Handwerkszeug der Spieleentwicklung mit Unity kennengelernt haben, wollen wir uns am Schluss noch einmal dem grundsätzlichen Vorgehen beim Entwickeln eines eigenen Spiels widmen, dem sogenannten Produktionsprozess.

Die Entwicklung eines Spiels ist aufgrund der verschiedenen, aber miteinander verknüpften Tätigkeiten (Programmieren, Modellieren, Animieren, Texturieren) ein recht komplexes Unterfangen. Dementsprechend ist auch ein strukturiertes Vorgehen bei der Spieleentwicklung sehr wichtig, die im Vorwege auch geplant werden sollte. Dabei gilt der Grundsatz: Je mehr Personen an der Entwicklung beteiligt sind, desto detaillierter sollte auch eine Planung sein, an die sich alle halten.

Möchten Sie auf Hobbybasis für sich ein kleines Spiel entwickeln, kann sich die Vorbereitung vielleicht auf ein paar Notizen und Skizzen beschränken. Sollten Sie aber eine Auftragsarbeit machen, bei der Sie sich an Termine halten müssen, steigt sofort der Umfang der Vorbereitung an. Aber auch die Zusammensetzung des Teams spielt eine Rolle. Entwickeln Sie z.B. zu zweit ein Spiel, wo der bzw. die andere Ihnen gegenübersteht, können Entscheidungen leichter kurzfristig getroffen werden, als wenn sich die andere Person auf einem anderen Kontinent in einer anderen Zeitzone befindet.

## ■ 22.1 Die Produktionsphasen

Auch wenn die Vorgehensweisen in der Praxis sehr unterschiedlich sein können, kann man den gesamten Prozess bei der Spieleentwicklung in einer ganz allgemeinen Struktur darstellen. Diese kann man wiederum in einzelne Phasen aufteilen, die ich kurz erläutern möchte. Möchten Sie mehr zur Planung und Entstehung von Spielen erfahren, kann ich Ihnen das Buch „Game Design und Produktion“ von Gunther Rehfeld empfehlen, das einen sehr guten Einblick in diese Thematik bietet.

### 22.1.1 Ideen- und Konzeptionsphase

Am Anfang eines Spiels steht immer eine Spielidee bzw. die Ideenfindung hierzu. Die Spielidee kann ein grobes Konzept sein, es kann aber auch aus einer einfachen Spielmechanik oder einem Genre bestehen, für das man ein Spiel machen möchte.

Als Nächstes wird diese Basisidee ausgearbeitet und zu einem echten Spiel geformt, häufig auch mithilfe von Prototypen, um einzelne Ideen zu testen. Bei der professionellen Spieleentwicklung werden dabei alle Punkte des Spiels genau ausgearbeitet und in einem Game-Design-Dokument zusammengefasst.

Dieses Dokument stellt dann den Kern der weiteren Spieleentwicklung dar und sollte alle relevanten Fragen beantworten (siehe weiter unten), die die verschiedenen Mitglieder des Entwicklerteams benötigen, um das Spiel umzusetzen. Ein Game-Design-Dokument ist dabei vergleichbar mit dem Drehbuch eines Films, in dem die gesamte Handlung beschrieben wird.

Aber auch wenn dort alles genau erläutert wird, ist so ein Game-Design-Dokument noch lange kein starres Instrument, an das man sich mit Biegen und Brechen halten sollte. Sie werden immer wieder auf Dinge stoßen, die sich vielleicht nicht umsetzen lassen wie anfangs gedacht, oder sie erweisen sich nicht als praxistauglich. Deshalb lebt ein solches Dokument und sollte im Laufe der Spieldesignerproduktion immer wieder den Änderungen und Erweiterungen entsprechend angepasst werden.

Bei Hobbyentwicklern ist so eine schriftliche Ausarbeitung natürlich ebenso wichtig. Allerdings besteht diese hier in der Praxis meistens eher aus Skizzen und Ideenlisten, die mehr einen roten Faden darstellen als eine detaillierte Vorgabe für den/die Entwickler. Nichtsdestotrotz sollte man sich natürlich auch hier später an die eigenen Vorgaben halten.

### 22.1.2 Planungsphase

Nachdem nun das gesamte Spiel in dem Game-Design-Dokument theoretisch ausgearbeitet vorliegt, wird darauf aufbauend ein Projektplan erarbeitet, der festlegt, welche Person wann etwas aus dem Design-Dokument umsetzt. Je größer das Team ist, desto wichtiger ist natürlich auch hier eine detaillierte Planung.

Aber auch die Komplexität des Games spielt bei der Ausarbeitung der Planung eine wichtige Rolle. Selbst wenn Sie alleine ein Spiel entwickeln möchten, kann es dann ohne Planung schnell vorkommen, dass Sie sich in Details verzetteln und den Faden verlieren.

### 22.1.3 Entwicklungsphase

In der eigentlichen Entwicklungsphase geht es jetzt darum, das im Game-Design-Dokument Niedergeschriebene in die Praxis umzusetzen. Hierzu gehören das Erstellen der Skripte, das Modellieren der 3D-Modelle, das Gestalten der Levels, das Komponieren der Musik, das Riggen und Animieren, das Texturieren usw. Dabei sollten Sie natürlich stets den Projektplan beachten und diesen anpassen, wenn sich dieser ändert.

## 22.1.4 Testphase

Die Testphase wird meistens in Alpha- und Beta-Phase unterschieden. Während in der Alpha-Phase das Spiel bereits spielbar ist und die grundlegenden Spielfunktionen implementiert sind, kann es immer noch zu kleinen Änderungen und Anpassungen kommen, die sich durch das Testen ergeben. So wird z.B. in dieser Phase noch viel am Spielfluss und am Balancing des Spiels gearbeitet, damit das Game am Ende auch ausgewogen und spielbar ist. In der Beta-Phase werden dann keine Funktionalitäten mehr zugefügt oder geändert (auch „Code-Freeze“ genannt). Stattdessen werden nur noch Fehler behoben, die sich beim Testen ergeben.

## 22.1.5 Veröffentlichung und Postproduktion

Am Ende sollte nun ein fertiges Spiel stehen, das auslieferbereit ist. Dieses kann dann entsprechend im App-Store hochgeladen oder woanders bereitgestellt werden.

Während die Veröffentlichung an sich nur ein Ereignis darstellt, schließt sich diesem aber meistens noch eine echte Phase an, die sogenannte Postproduktion. In dieser geht es schließlich darum, nachträglich gefundene Fehler zu beheben und Bug-Fixes bereitzustellen sowie, je nach Spielkonzept, auch Updates mit weiteren Features zu veröffentlichen. Die Weiterentwicklung eines Spiels kann dabei so umfangreich sein, dass der gesamte Prozess wieder von vorne beginnt.

# ■ 22.2 Das Game-Design-Dokument

Im Rahmen der Spieleentwicklung wird normalerweise immer ein Game-Design-Dokument erstellt, in dem das Spiel beschrieben wird. Das Ziel dieses Dokumentes ist es, sich im Vorwege detailliert Gedanken darüber zu machen, wie das Spiel werden soll, und dieses dann auch schriftlich zu fixieren.

Dabei gibt es keine echten Vorschriften, wie so ein Dokument auszusehen hat oder wie die Informationen erfasst werden. Je nach Information kann dies z.B. in Textform, als Skizze oder auch als Diagramm in das Dokument eingebracht werden. Am Ende muss es zweckdienlich sein, schließlich soll es Ihnen und den anderen Entwicklern später helfen, das Spiel umzusetzen. Und umso mehr Details bereits hier geklärt werden, desto weniger Fragen treten später auf.

Auch wenn es keine echten Vorgaben für ein Game-Design-Dokument gibt, möchte ich Ihnen hier einige Fragen nennen, die in so einem Dokument erklärt werden könnten.

- Welchem Genre entspricht das Spiel?
- Was für ein Grafikstil soll genutzt werden?
- Welche Zielplattformen sollen unterstützt werden (Gerätetyp, Betriebssystem, technische Anforderungen, usw.)?

- Wer ist der Held, wie soll er aussehen und was soll er können?
- Was für eine Hintergrundgeschichte hat das Spiel und wie wird sie im Spiel fortgeführt?
- Wie funktioniert die Spielesteuerung?
- Wie sehen die Levels aus? Was für Interaktionsmöglichkeiten gibt es? Wie sind die Lösungswege?
- Was für Gegner gibt es und was können diese?
- Welche Menüs sind vorgesehen und wie sind diese aufgebaut?
- Was für Audiodateien werden benötigt (Musik, FX Sounds, was für eine Art Sounds etc.)?

Diese und alle anderen Punkte, die am Ende in dem Game-Design-Dokument festgehalten werden, sollten während des Produktionsprozesses auch kontrolliert und, sollten sich Änderungen ergeben, angepasst werden.

# Schlusswort

Wenn Sie diese Zeilen lesen, sind Sie bereits am Ende des Buches angekommen. Sie haben gelernt, wie Sie Unity nutzen können, um eigene Spiele für den Computer und für das Smartphone zu entwickeln. Natürlich werden Sie jetzt nicht gleich den nächsten MMORPG aus dem Ärmel schütteln können, auch wenn Sie viele Grundlagen dafür nun kennen. Im Gegenteil, Sie sollten eher klein beginnen.

Sollten Sie jetzt anfangen, Ihr erstes eigenes Spiel zu entwickeln, starten Sie zunächst mit einem ganz kleinen Spiel mit nur wenigen Features. Entwickeln Sie z.B. ein virtuelles Dosenwerfen oder einen Pong-Klon. Wichtig ist hierbei, dass Sie das Spiel komplett fertig machen. Denn zum einen ist es immer ein Erfolgserlebnis und damit motivierend, ein Projekt abzuschließen und es seinen Freunden, Bekannten oder seiner Community vorzustellen. Zum anderen sollten Sie beachten, dass gerade das Abrunden eines Spiels, wie z.B. das Erstellen von Menüs, Speicher- und Ladefunktionen usw., häufig mehr Arbeit bedeutet, als Sie vielleicht anfangs denken.

Auch möchte ich Ihnen an dieser Stelle sagen, dass noch nie ein Meister vom Himmel gefallen ist, schon gar nicht in der Spieleentwicklung. Je mehr Spiele Sie im Laufe der Zeit entwickeln werden, desto mehr werden Sie natürlich lernen. Erst durch die Praxis werden Sie ein Gespür dafür bekommen, wie Sie am besten eine Landschaft gestalten, wann Sie wie eine Programmierklasse aufbauen sollten oder auf welche Weise Sie einen ausgedachten Effekt am besten mit einem Partikelsystem umsetzen können. Geben Sie sich also Zeit und fangen Sie klein an. Mit Unity haben Sie jedenfalls ein passendes Tool an der Seite, mit dem Sie Ihre kreativen Spieleideen verwirklichen können.

Lizenziert für gregwyss@hotmail.com.

© 2014 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

# Index

## Symbolen

2D 95, 209, 278, 319  
 2D-Button 8, 10  
 3D-Koordinatensystem 95  
 3D Sound 209  
 3DText 271

## A

A\*-Algorithmus 345  
 Accelerometer 197  
 Accessoren 55  
 activeTerrain 259  
 AddComponent 48, 80  
 AddExplosionForce 152  
 AddField 305  
 AddForce 151  
 AddForceAtPosition 152  
 AddRelativeForce 152  
 AddRelativeTorque 152  
 AddTorque 152  
 Advanced 106  
 Ambient Light 129  
 Anchors 283  
 Anchor-Handle 285  
 Anchor Presets 283  
 Angular Drag 148  
 Animation 295, 315, 326  
 Animation-Clip 326, 330  
 Animation Events 341  
 Animation State 330, 432  
 Animation Types 323  
 Animation View 317  
 Animator 334  
 Animator Controller 329, 432  
 Anti-Roll Bars 174  
 anyKey 191  
 Any State 331  
 Application 88  
 Area Light 133  
 Array 37ff., 54, 63  
 Asset 6f., 14, 17ff., 22f., 26ff.

Asset Store 29, 114, 250, 260

Atomic 332  
 Audio 203  
 AudioClip 209  
 Audio-Filter 211  
 AudioListener 117, 203  
 AudioSource 204  
 Ausführungsreihenfolge 93  
 Avatar 323  
 Avatar Body Mask 333  
 Avatar Definition 325  
 Awake 73f.

## B

Background 115  
 Backup 370  
 Balancing 471  
 Basisklasse 56, 68  
 Bedingte Operatoren 43  
 BeginHorizontal 276  
 BeginVertical 276  
 Behaviour 70  
 Beschleunigungssensor 197  
 Betrag 97  
 Bit-Feld 162  
 Blend Tree 332  
 Blending 334  
 Blueprint 282  
 Bone-Mapping 324  
 Boo 67  
 Box Collider 154  
 Brush 245  
 Build-In-Shader 105  
 Button 273, 291  
 Byte Order 248

## C

C# 33, 67  
 Camel Case 35  
 Camera 115, 117, 280  
 CancelInvoke 84

Canvas 278

Capsule 111  
 Capsule Collider 154  
 centerOfMass 150  
 Character Controller 178, 183, 403  
 Clear Flags 115  
 Clipping Planes 116  
 Cluster 256  
 Collider 80, 154  
 Collision 157  
 Collision Detection 149, 160  
 CollisionEvent 228  
 Component 15, 24, 69, 75, 78  
 Conditions 330  
 Console 6, 20, 66, 91  
 ConstantForce 153  
 Constraints 149  
 Cookie 106  
 Coroutine 82, 303  
 Create from Grayscale 108  
 CreateInstance 69  
 CREATE TABLE 307  
 Create Tree Collider 253  
 CSV 308  
 Cube 111  
 Cubemap 136  
 Culling Mask 115, 131  
 Cursor 106, 193  
 CursorMode 194  
 Curves 318  
 Cylinder 111

## D

Datenbank 303, 306  
 Datenbanksprache 306  
 Datentypen 36  
 Debug 91  
 Default Cursor 389  
 Default Time 93  
 Deferred Lighting 145  
 deltaTime 72  
 Depth 116, 123

Destroy 69, 78, 81  
 Dictionary 63, 380  
 Directional Light 131, 134  
 DontDestroyOnLoad 69, 90  
 DontRequireReceiver 80  
 Dope Sheet 318  
 do-Schleife 46  
 Drag 148  
 DVD 3

## E

Editor 92  
 Editor Extensions 29, 113  
 Eigenschaftsmethode 55  
 Eltern-Objekt 102  
 Emissive Materials 142  
 Empty GameObject 24, 101, 172  
 enableEmission 227  
 EndHorizontal 276  
 EndVertical 276  
 Enumeration 39  
 eulerAngles 103f., 181  
 Event 292, 296, 328  
 Event Camera 281  
 Event-Delegates 290  
 Event-Methoden 72, 270  
 Event Trigger 296

## F

FBX 112, 323  
 Field of View 116  
 Find 77  
 FindGameObjectsWithTag 77  
 FindWithTag 74, 76  
 First Person Controller 181  
 fixedDeltaTime 147  
 Fixed Joint 176  
 Fixed Timestep 147f., 159, 174  
 FixedUpdate 73, 147, 151  
 Flare Layer 117  
 Flatten 246  
 fontSize 298  
 ForceMode 153  
 foreach-Schleife 46  
 for-Schleife 45  
 Forward Friction 165  
 Forward Rendering 144  
 Frame 30, 72, 82, 168

## G

Game Controller 393  
 Game-Design-Dokument 470  
 GameObject 7f., 14ff., 24f., 30f.,  
 68f., 74, 76ff., 80, 88  
 Game View 6, 10f., 13, 31  
 Generate Colliders 339

Generic - Animation Type 324  
 GET 305  
 GetAxis 189  
 GetButton 190  
 GetButtonDown 181  
 GetCollisionEvents 227f.  
 GetComponent 79  
 GetCurrentAnimatorStateInfo 338  
 GetKey 191  
 GetMouseButton 192  
 GetMouseButtonDown 153  
 GetQualityLevel 259  
 GetTouch 196  
 Gizmo 9ff., 204  
 Gizmo Display Toggles 12  
 Graphical User Interface 267  
 Grass Lighting 255  
 Gravitation 148  
 Grayscale 105, 108, 136, 247  
 GUI 106, 267, 270  
 GUIElement 196, 268  
 GUI-Klasse 273  
 GUILayer 117  
 GUILayout-Klasse 275  
 GUISkin 276  
 GUIStyle 276  
 GUIText 268  
 GUITexture 268

## H

Hand-Tool 12  
 Hard Shadows 134  
 Hash-Wert 337  
 Heightmap 105, 108, 247f.  
 Hierarchy 6, 9, 13ff., 30f.  
 Highscore 306  
 Hinge Joint 177  
 HitTest 197  
 HTML 305  
 Humanoid - Animation Type 324

## I

Icon 15  
 identity 104  
 IEnumerator 83  
 if-Anweisung 41  
 Image 287, 295  
 Image Effects 125  
 Initialisierung 36  
 Input 153, 189, 194  
 InputField 292  
 Input-Manager 185, 188, 200  
 Input Settings 183  
 INSERT INTO 307  
 insideUnitSphere 82  
 Inspector 15, 55, 76  
 Instantiate 104, 300

Instanziieren 37, 47  
 Instanzmember 52  
 Interface 59  
 Interpolate 149  
 Intersection 228  
 Inventarsystem 380  
 Invoke 84  
 InvokeRepeat 84  
 IsInvoking 84  
 Is Kinematic 148, 157, 198  
 Is Trigger 158  
 isWebPlayer 455

## J

JavaScript 67  
 Joints 176  
 Joystick 188

## K

Kamera 115  
 KeyCode 191  
 Keyframe 317  
 Kl 345, 442  
 Kind-Objekt 102  
 Klasse 47, 56, 65, 70  
 Klassendiagramm 68  
 Klassenmember 52  
 Kollisionen 80, 154  
 Kollisionserkennung 148, 154, 158  
 Komplizierungsreihenfolge 92  
 Komponente 15, 17, 24, 47f., 68f.,  
 75, 78, 81  
 Konsole 91  
 Konstanten 39  
 Künstliche Intelligenz 345, 430,  
 442

## L

Label 273  
 LateUpdate 75, 82  
 Layer 13, 16, 26  
 Layer-Based Collision Detection  
 161  
 Layer Collision Matrix 161  
 LayerMask 161, 412  
 Lebensverwaltung 395  
 Lens Flares 139  
 Level Index 88, 90  
 Light 129  
 Light Cookies 135  
 Light Halos 138  
 Lightmapping 129, 131, 133, 141  
 linkshändiges Koordinatensystem  
 95  
 List 63  
 Lizenzmodelle 2

localEulerAngles 103, 170  
 localPosition 103  
 localRotation 103  
 LookAt 103  
 loop match 327

**M**

magnitude 97  
 Main Camera 115, 119, 203  
 Manual 7  
 Mask 328  
 Mass 148  
 Mass Place Trees 253  
 Masseschwerpunkt 150  
 Material 104  
 Mathf 52, 168  
 Mehrspieler-Games 199  
 Mesh 98, 154  
 Mesh Collider 154f., 161  
 MeshFilter 100  
 MeshRenderer 100, 135  
 Methode 48  
 Minimap 123  
 Missing (MonoBehaviour) 71  
 MoCap 316  
 Model Import Settings 113  
 MonoBehaviour 48, 56, 68, 70, 72,  
   88  
 MonoDevelop 2, 65f., 71  
 Mono-Framework 33  
 Motion Capturing 316  
 mousePosition 192  
 Move 180  
 Muscle 324  
 mySQL 306  
 mysql\_query 307

**N**

nameHash 338  
 Namespace 61, 68, 72  
 Navigation 290  
 Navigation Layer 350  
 NavigationMesh 346f.  
 NavMesh 347, 434  
 NavMeshAgent 346, 432  
 NavMeshObstacle 350, 435  
 Negationsoperator 81  
 Netzwerkspiele 200  
 Noise 199  
 Normale 99  
 Normalenvektor 99  
 Normalisieren 98  
 normalized 98  
 Normalized View Port Rect 116, 122  
 Normalmap 105f., 108  
 null 78

**O**

Object 68f.  
 Objektorientierte Programmierung  
   47, 56, 62  
 OnCollision 157  
 OnControllerColliderHit 181  
 OnDestroy 88  
 OnGUI 74, 272  
 OnLevelWasLoaded 90  
 OnMouseDown 152, 270  
 OnMouseEnter 270  
 OnMouseExit 270  
 OnMouseOver 153, 270  
 OnMouseUp 270  
 OnMouseUpAsButton 270  
 OnParticleCollision 222, 227, 237,  
   241  
 Orientierung 197  
 Orthogonal 10  
 Orthographic 21, 124  
 Ortsvektor 97  
 out 53

**P**

Paint Details-Tool 254  
 Paint Height-Tool 246  
 Paint Texture-Tool 249  
 Parametermodifizierer 53  
 Parenting 14, 102, 118  
 parsen 309  
 Particle Effect Control 215  
 ParticleSystem 213, 263  
 Partikeleffekte 213  
 Partikelsysteme 213, 264  
 Pathfinding 345, 430  
 Perspective 21  
 Perspektivisch 10  
 PHP 303, 305f.  
 phpMyAdmin 307  
 Physics Materials 164, 175  
 Physics Settings 161  
 Physik-Engine 73, 147, 154  
 Pixelkoordinaten 192  
 Pixelkoordinatensystem 192  
 Pixel Lighting 144  
 Place Tree-Tool 252  
 Plane 112  
 Plane Distance 281  
 PlayClipAtPoint 208, 241  
 Play Controls 13  
 PlayerPrefs 85f., 88  
 Point Light 132  
 Polygon 98  
 Polygontext 98  
 POST 305  
 Postprocessing-Effekte 125  
 Prefabs 78, 299  
 Primitives 111, 157

Primitive Collider 154ff., 160  
 Produktionsphasen 469  
 Produktionsprozess 469  
 Project Browser 6, 8, 14ff., 22, 27f.,  
   30  
 Projection 116, 124  
 Projector 140  
 Project Wizard 5, 20ff.  
 Projekt Browser 71  
 Pro Standard Assets 92  
 Prozedurale Mesh-Generierung 114

**Q**

Quad 111  
 QualitySettings 259  
 Quaternion 82, 104, 181  
 Quelltext 34  
 Quest 416  
 Questgeber 418

**R**

Raise/Lower-Tool 245  
 Random 81  
 Raw Image 288  
 Raycast 54, 98, 160, 177  
 RaycastHit 178  
 Raycasting 177  
 Rect-Handle 281f.  
 Rect-Tool 12, 281f.  
 RectTransform 70, 281  
 ref 53  
 Reflection 106  
 Reflection Cubemap 105  
 Rename 71  
 Rendering Path 117, 144  
 Render Mode 130, 279  
 Render Settings 127, 129  
 Render Texture 125, 261  
 RepeatButton 273  
 Reset 102  
 Reverb Zone 210  
 Richtungsvektor 97  
 Rig 323  
 Rigidbody 148, 198  
 Root 102  
 Root Motion 328  
 Rotate 103  
 Rotate-Tool 12  
 RotateTowards 182

**S**

Sandbox 303  
 Scale-Tool 12  
 Scene Gizmo 9, 10  
 Scene View 6, 8f., 11ff., 30f. 95,  
   157  
 Schatten 134

Schnittstellen 59  
 Schriftgröße 298  
 Screen 297  
 ScreenPointToRay 120, 193  
 Screen Space – Camera 280, 295  
 Screen Space – Overlay 279  
 ScriptableObject 69  
 Script Execution Order Settings 93  
 Scripting Reference 3, 7, 66  
 Scrollbar 293  
 SELECT 308  
 Selection List 294  
 Self-Illuminated-Shader 142  
 Send Collision Messages 227  
 SendMessage 79  
 SendMessageOptions 79  
 SetActive 77  
 SetCursor 193  
 SetDestination 347  
 SetQualityLevel 259  
 Shader 104  
 Shadow Type 130  
 Shuriken 213  
 Sichtbarkeit 51, 57  
 Sideways Friction 165  
 SimpleMove 179  
 Singleton 90  
 Size 116  
 SkinnedMeshRenderer 100  
 Skript 65, 70, 75, 92  
 Skybox 127  
 Slider 293  
 Smartphones 194  
 Smooth Height-Tool 247  
 Smooth Sphere Collisions 155  
 Snapping-Modus 282  
 Soft Shadows 134  
 Source-Code 34  
 Spawn-Point 406  
 Sphere 111  
 Sphere Collider 154  
 Spherical Harmonics Lights 145  
 Split 309  
 Split-Screen 122, 200  
 Spot Light 132  
 sprachübergreifende Zugriff 92  
 Spring Joint 176  
 Sprite 13, 106, 285, 288, 319  
 Sprite Animation 223, 319  
 Sprite Editor 285, 319  
 SQL 306  
 sqrMagnitude 97  
 Stand-alone 85, 186  
 Standard Assets 92  
 Start 74

State Machine 329  
 Static 16  
 Static Collider 159  
 Stiffness 165  
 StringToHash 337  
 Subfenster 275  
 Sub-Quest 425  
 Superglobals 305  
 switch-Anweisung 44  
 Syntax 34  
 Szene 23, 88

**T**

Tablets 194  
 Tabs 5  
 Tag 25, 76  
 Target Texture 117, 125  
 Terrain 161, 243, 263  
 Terrain Collider 161, 244, 253  
 Terrain Settings 247, 248  
 Text 286, 295  
 TextArea 274  
 TextField 274  
 Texture 106  
 Texture Type 270, 285, 319  
 Tiepass-Filter 199  
 Tiling 107  
 Time 72, 147  
 Time Manager 147  
 Toggle 292  
 Touch 194  
 Touch-Controls 196  
 TouchCount 196  
 Transform 14, 70, 73, 101  
 Transform-Tools 8f., 12, 30, 101, 281  
 Transition 289, 330  
 Translate 103, 198  
 Translate-Tool 12  
 Tree 252f., 263  
 Triangle 98, 110  
 Trigger 158  
 Trigger-Collider 158, 206

**U**

uGUI 278, 455  
 UIRenderer 278  
 UnityEngine 68, 72  
 UnityEngine.UI 278  
 UnityGUI 272  
 UnityPackage 27f.  
 UnityScript 68  
 Update 72, 82  
 Upgrading 22

Use Gravity 148  
 UV Mapping 110

**V**

var 37  
 Variablenzugriff 80  
 Vector3 96, 104  
 Vektor 96  
 Vererbung 56  
 Vergleichsoperatoren 42  
 Versiegeln 58  
 Version Control System 370  
 Versionsverwaltung 370  
 Vertex Lighting 144, 255  
 Vertex Lit 145  
 Vertex-Snapping 12  
 Vertices 98  
 Viewport Space 268  
 Virtuelle Achsen 185, 187  
 Virtuelle Tasten 187

**W**

WaitForSeconds 83  
 WASD-Tastensteuerung 181, 403  
 Wasser 261  
 Water 261  
 Webplayer 1, 85, 303  
 Web-Player 454  
 WebPlayer Template 92, 454  
 Webspace 306  
 Wegfindung 345  
 Wheel Collider 163  
 Wheel Friction Curve 164, 174  
 while-Schleife 46  
 Wind Zones 252, 260, 263  
 World Space 281  
 Wrap Mode 107  
 WWWForm 305  
 WWW-Klasse 303

**X**

XML 308

**Y**

yield 83, 303

**Z**

Zeilenumbrüche 287  
 Zufallswerte 81  
 Zugriffsmodifizierer 60, 78

TOM DEMARCO



Als auf der Welt  
**DAS LICHT**  
ausging

Ein Wissenschafts-Thriller

HANSER

Lizenziert für gregwyss@hotmail.com.

© 2014 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

**»Der Weltuntergang steht bevor,  
aber nicht so, wie Sie denken.  
Dieser Krieg jagt nicht alles in die Luft,  
sondern schaltet alles ab.«**

Tom DeMarco  
Als auf der Welt das Licht ausging

ca. 560 Seiten. Hardcover  
ca. € 19,99 [D]/€ 20,60 [A]/sFr 28,90  
ISBN 978-3-446-43960-3 · WG 121  
Erscheint im November 2014



Sie möchten mehr über Tom DeMarco und  
seine Bücher erfahren.  
Einfach Code scannen oder reinklicken unter  
[www.hanser-fachbuch.de/special/demarco](http://www.hanser-fachbuch.de/special/demarco)

Lizenziert für gregwyss@hotmail.com.  
© 2014 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder  
Vervielfältigung.

**TOM DEMARCO**

**Als auf der Welt  
DAS LICHT  
ausging**

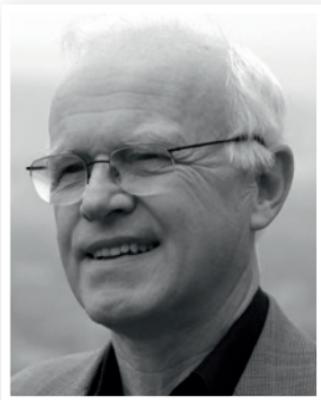
Aus dem Amerikanischen von Andreas Brandhorst

**Leseprobe**

Das Buch erscheint im November 2014.

## Tom DeMarco

ist Projektmanagement-Experte, vielgefragter Berater und Autor zahlreicher im Carl Hanser Verlag erschienener Bestseller wie »Der Termin« oder zuletzt »Wien wartet auf Dich«. Er ist Partner der Atlantic Systems Guild, einer Beratergruppe, die sich auf die komplexen Prozesse der Systementwicklung spezialisiert hat, mit besonderem Augenmerk auf die menschliche Dimension.



## Andreas Brandhorst

Andreas Brandhorst (geboren 1956 in Norddeutschland) hat mit dem Schreiben sehr früh angefangen und wenige Jahre später wurde aus dem Hobby ein Beruf, zu dem nicht nur das Schreiben eigener Texte gehörte, sondern auch das Übersetzen (u.a. der Werke von Terry Pratchett). Er ist

Autor der bekannten Kantaki-Romane (2 Trilogien), der Science-Fiction-Romane »Kinder der Ewigkeit«, »Das Artefakt« (ausgezeichnet mit dem Deutschen Science-Fiction-Preis als bester deutscher SF-Roman 2013), »Der letzte Regent« und »Das Kosmotop« (erscheint Juni 2014). Außerdem gehen die Thriller »Äon«, »Die Stadt« und »Seelenfänger« auf sein Konto.



## Was vorher geschah:

Fanatiker in der US-Regierung glauben sich hinter ihrem Raketenabwehrschild sicher und wollen Amerikas Gegner überall auf der Welt mit einem nuklearen Erstschlag auslöschen. Doch eine kleine Gruppe von Forschern hat einen Apparat entwickelt, der weltweit Atomexplosionen verhindern kann. Die Aktivierung dieses Apparats würde zwar einen Atomkrieg verhindern, aber der Menschheit auch die Elektrizität nehmen

...

## DER MANN, DER EINGRIFF

Homer hatte sie alle zu Bett geschickt. Während der vergangenen Nacht waren sie auf den Beinen gewesen, sagte er, und sie mussten frisch und ausgeruht sein für das, was geschehen würde. Doch zumindest für Loren und Edward war an Schlaf nicht zu denken. Sie saßen am Fenster von Edwards Zimmer und blickten über die Stadt. Ed hatte das Licht ausgeschaltet, dann wieder ein und noch einmal aus. Eine ganz einfache Sache, über die man gar nicht nachdachte, aber würde sie am kommenden Tag noch möglich sein? Er durchquerte das dunkle Zimmer und setzte sich auf den leeren Stuhl Loren gegenüber.

»Ich glaube, ich weiß, was passieren wird«, sagte er.  
»Kelly hat es bereits gesagt. Es scheint vorherbestimmt zu sein.«

Loren nickte betrübt.

»Die Offshore-Kabaner werden sich genau so verhalten, wie es Simula-7 vorhergesagt hat. Sie sind vollkommen berechenbar. Sie werden St. Louis angreifen, wie in der Simulation. Sie werden es sich nicht anders überlegen. Bestimmt gehen sie davon aus, dass unsere Behörden eine Evakuierung der Stadt veranlassen. Und bestimmt reden sie sich ein, dass wir eine Eskalation vermeiden wollen. Es wäre dumm von uns, alles auf die Spitze zu treiben – nach dem, was wir auf Kuba angerichtet haben, ist die Zerstörung einer leeren Stadt durch

eine einzelne Rakete nicht mehr als eine kleine Verwarnung.«

Loren nickte erneut. »Dumm«, wiederholte er.

»Sie werden glauben, dass wir nicht zurückschlagen, aber da irren sie sich. Wir wissen es besser. Stell dir vor, was passiert, wenn sie ihre eine Rakete auf St. Louis abfeuern. Das modifizierte Revelation 13 erledigt sie vielleicht, oder auch nicht. Möglicherweise ist das System nicht in der Lage, eine einzelne Rakete abzufangen. Angenommen, es ist dazu imstande. Was machen wir dann?«

Loren dachte darüber nach. »Die logische Sache wäre, nichts weiter zu tun. Die Welt könnte glauben, dass Amerikas Raketenabwehr schild funktioniert, dass wir unangreifbar sind. Es wäre eine sehr starke Position für uns.«

»Aber wir gehen nicht logisch vor. Wir sind Fanatiker.«

»Ja. Also regen wir uns mächtig auf. Man hat uns angegriffen; wir müssen es den Angreifern zeigen. Es ist eine Frage verletzten Männerstolzes.«

»Also schicken wir unsere Raketen los. Wir radieren Iran, Nordkorea, Pakistan und alle anderen Länder aus, die uns ein Dorn im Auge sind. Genau das passiert, wenn der Schild hält. Wenn nicht ... Dann sitzen wir in den Trümmern von St. Louis. Es wird viele Opfer geben, weil wir die Stadt nicht evakuiert haben. Was machen wir?«

»Vergeltung. Wir suchen einen Sündenbock und starten unsere Raketen.«

»Wir starten sie auf jeden Fall.«

»Ja, auf jeden Fall.«

Sie schauten in die Nacht hinaus und beobachteten die Lichter der Stadt. Nach einer Weile fuhr Edward fort: »In der griechischen Tragödie gibt es einen Moment des Übergangs, direkt nach dem Höhepunkt. Vorher haben Menschen die Ereignisse kontrolliert und nachher kontrollieren die Ereignisse die Menschen. Ich habe den Eindruck, dass dieser Moment heute Morgen verstrichen ist. Jetzt erwartet uns das düstere Ende

der griechischen Tragödie; die Akteure werden zu Zuschauern.«

»Das gilt nicht für uns«, sagte Loren. »Bei uns sieht es anders aus. Wir können eingreifen. Wir können handeln, den Effektor einschalten.«

»Aber können wir frei entscheiden? Haben wir eine Wahl? Eigentlich nicht. Wir müssen den Effektor einschalten. Die Zahlen diktieren es, denn selbst ein begrenzter nuklearer Schlagabtausch würde weitaus mehr Opfer fordern. Wir müssten selbst dann eingreifen, wenn wir wüssten, dass weitere Eskalationen ausbleiben. Das ist der zweite Teil von dem, was meinem Gefühl nach heute Nacht geschehen wird: Wir schalten den Effektor ein. Wodurch die Welt zum Stillstand kommt. Die Menschen, die von den Atomraketen getötet worden wären, leben noch – wir haben sie gerettet. Aber jetzt funktioniert nichts mehr. Die abgefeuerten Raketen fallen zu Boden, ohne zu explodieren, doch das ist nur der Anfang. Motoren lassen sich nicht mehr starten, es gibt keinen elektrischen Strom, Flugzeuge stürzen ab ... Der wahre Albtraum beginnt.«

»Vielleicht können wir den Effektor später abschalten. Wenn die Krise überstanden ist.«

»Nein, nie. Es wird immer mehr Waffen geben, die auf ihre Chance warten.«

»Wir können den Effektor nach Belieben ein- und ausschalten.«

»Loren. Denk gründlich darüber nach. Wenn wir uns einen Spaß daraus machen, den Permanenten Effektor zu aktivieren und zu deaktivieren ... Wie lange dauert es dann wohl, bis die Leute merken, dass wir dahinterstecken?«

»Ich verstehe nicht ganz ...«

»Rupert Paule wendet sich an Armitage, einen Physiker von Weltklasse, und sagt: »Was zum Teufel ist hier los? Welche Kraft neutralisiert unsere Raketen, Motoren und Generatoren?« Armitage nimmt einige Untersuchungen vor, während der Effekt wirkt. Was kann er herausfinden?«

Loren überlegte. »Die potenzielle Energie in jeder brennbaren Substanz erscheint reduziert, wenn sich der Effekt auswirkt, und kehrt ohne den Effekt auf das normale Niveau zurück. Das böte einen Hinweis auf die ganze Theorie von T-Prime. Wozu wir Jahre gebraucht haben, um es zu verstehen ...«

»Armitage könnte viel schneller die richtigen Schlüsse ziehen, vielleicht in Wochen.«

»In Tagen«, sagte Loren.

»Wahrscheinlich. Und dann würde Paule fragen: »Wer tut uns dies an, Dr. Armitage?« Und für Lamar wäre es schon nach einer Sekunde klar: Homer Layton und sein Team. Homers Artikel in Science über die Pekuliarbewegung bietet einen eindeutigen Hinweis.«

»Sie würden über uns herfallen, uns den Effektor wegnehmen und ihn ausschalten.«

»Und manchmal würden sie ihn wieder einschalten.«

Loren begriff, was Edward meinte. »Oh.«

»Ja. Sie schalten ihn ein, wenn sie gegen sie gerichtete strategische Aktionen entdecken. Und sie schalten ihn aus, wenn sie selbst zuschlagen wollen. Einen besseren Abwehrschild gibt es nicht.«

»Vielleicht wäre es gar nicht so schlecht.«

»Es wäre furchtbar. Denn ihnen müsste klar sein, dass auch andere Länder Physiker haben. Mit dem Hinweis, den wir ihnen gegeben haben, kommen sie schnell hinter das Geheimnis von T-Prime. Was bedeutet, dass sie nach einigen Wochen ihren eigenen Effektor bauen können. Doch das würde den Eiferern und Fanatikern ganz und gar nicht gefallen. Sie müssten damit rechnen, dass ihr Vorteil nach wenigen Wochen dahin ist. Was würden sie tun?«

»Sie müssten handeln, um ihren Vorteil abzusichern. Sie würden vielleicht ...«

»Genau. Sie würden angreifen, solange sie die Gewiss-

heit haben, dass sich der Gegner nicht wirkungsvoll zur Wehr setzen kann. Davon müssen wir ausgehen.«

Es dauerte eine Weile, bis sich die Erkenntnis festsetzte. Die Entscheidung, den Effektor einzuschalten, war auch die Entscheidung, ihn für immer an zu lassen. »Vielleicht funktioniert der Effektor gar nicht«, sagte Loren.

»Das ist unsere optimistischste Hoffnung.« Edward lächelte bitter. »Es würde bedeuten, dass wir zusammen mit dem Rest der Welt in nuklearer Glut gebraten oder vom Fallout vergiftet werden. Wir wären tot, was wir eines Tages ohnehin sein werden. Aber zumindest müssten wir uns nicht vorwerfen, dass alles unsere Schuld ist.«

.

.

.

... Eine Atomrakete vom Typ SS-24 startete von einer Insel vor der Küste Ecuadors, gerichtet auf St. Louis, Missouri. Der Startzeitpunkt war so gewählt, dass die Rakete ihr Ziel genau um Mitternacht St. Louis-Zeit erreicht. Ein zweihundert Meilen westlich von San Diego stationierter Zerstörer der amerikanischen Marine ortete die Rakete und einige Sekunden später wurde ein Alarm ausgelöst. Da der Zerstörer auf eine solche Sichtung vorbereitet gewesen war, ging man gleich auf Alarmstufe Rot und schickte eine Nachricht ins StratCom-Netzwerk.

Albert döste mit dem Ohr am Empfänger. Die Mitteilung hätte Teil seines Traums sein können, denn in letzter Zeit träumte er oft von solchen Dingen. Er hob den Kopf und starrte auf das Gerät in seiner Hand, das den Alarm wiederholte. Er blickte zu Homer, der wach im Sessel neben ihm saß. Homer hatte alles gehört. Die Worte der Ankündigung schienen keine nennenswerte Wirkung auf ihn zu haben; sie waren mehr wie ein morgens klingelnder Wecker. In diesem Fall lautete die Botschaft des Klingelns: Es geht los.

Albert hielt das kleine Gerät wieder ans Ohr und sein

Blick kehrte zu Homer zurück. »Neunzehn Minuten, glauben sie«, sagte er und sah auf die Uhr. »Um ein Uhr unserer Zeit.«

Homer stand mühsam auf. Alte Leute sollten nicht in tiefen Sesseln sitzen, dachte er. Loren, der auf dem Boden neben ihm geschlafen hatte, war schon auf den Beinen. »Ich hole die anderen«, sagte er.

Edward hatte seine Tür einen Spaltbreit offen gelassen. Loren sah ins Zimmer und sagte: »Es ist so weit, Ed.« Es hätte das frühe Wecken für den Beginn eines Campingausflugs sein können. Er hörte Edwards Antwort aus dem dunklen Zimmer und ging weiter, zu Sonia gleich nebenan.

Loren klopfte an und wartete. Er hörte Bewegung im Zimmer, die Tür öffnete sich und Sonia blinzelte im Licht des Flurs. »Sonia.« Er wollte sie in die Arme nehmen, sie trösten, doch sie behielt die Tür zwischen ihnen.

»Ich bin im Schlafanzug«, sagte sie.

»Komm so schnell du kannst zu Homer.«

»Gib mir ein paar Minuten fürs Anziehen.« Sie schloss die Tür.

Weiter zu Kellys Zimmer. Loren klopfte an und hörte Geräusche, bevor sich die Tür öffnete. Kelly war hellwach. Sie trug ein weißes Nachthemd mit Rüschen an den Ärmeln. Hinter ihr brannte eine kleine Lampe.

»Es ist geschehen«, sagte Loren.

Kelly zog ihn herein. »Sieh nach Curtis«, sagte sie. »Ich ziehe mir schnell was über.«

Loren ging ins Nebenzimmer und spähte in die Dunkelheit. Er hörte das gleichmäßige Atmen des Kindes. Die Gestalt im Bett wirkte friedlich im Schlaf. Er kehrte in Kellys Raum zurück. Sie stand vor der Kommode, mit dem Rücken zu ihm, und zog eine Jeans unter ihrem Nachthemd hoch. Ihr Hintern zeigte sich kurz, als sie die Hose zurechtrückte. Das Nachthemd warf sie achtlos beiseite. Loren sah ihren langen, schmalen Rücken. Sie war größer als seine Schwestern, dach-

te er, ein bisschen größer. Kelly zog sich ein T-Shirt über den Kopf und drehte sich zu ihm um. »Fertig«, sagte sie und stand barfuß vor ihm. Keine Schuhe, keine Unterwäsche. Sie trafen noch vor Edward in Homers Suite ein.

Homer hatte Maria geweckt. Sie trat aus dem Schlafzimmer und zog den Gürtel eines Morgenmantels zu. Claymore kam von der anderen Seite herein. Sonia und Edward erschienen gleichzeitig. Noch elf Minuten bis zum Einschlag. Homer schloss die Tür, verriegelte sie und drehte sich ernst zu ihnen um.

»Gloria Verde hat eine Rakete auf St. Louis abgefeuert. Albert hat den Alarm vor einigen Minuten mit seinem StratCom-Apparat gehört. Die Rakete wird ihr Ziel um ein Uhr unserer Zeit erreichen. Uns bleiben nur wenige Minuten, um genau zu überlegen. Darauf kommt es jetzt an, dass wir genau nachdenken.

Es gibt einige Dinge, die wir Albert, Maria und Claymore erklären müssen, über unsere Vereinbarung in Bezug auf den Effektor, falls wir entscheiden, ihn einzuschalten. Hörst du zu, Clay?«

»Oh, klar.« Claymore hatte als einziger Platz genommen. Er saß auf der Couch, in einem pfirsichfarbenen Schlafanzug. Auf dem Tisch lag eine Hochglanzbroschüre über das Nachtleben von Fort Lauderdale. Er schlug sie auf. »Klar«, sagte er.

Homer wandte sich an Albert und Maria. »Ihr wisst, was es mit dem Effektor auf sich hat. Ich habe es euch erklärt. Ihr wisst auch, was wir heute Nacht tun könnten, was wir in Erwägung ziehen. Aber was auch immer hier geschieht, ihr seid dafür nicht verantwortlich. Das ist wichtig. Die Verantwortung tragen wir fünf.« Er sah die Mitglieder der Gruppe an. »Ich selbst, Edward, Sonia, Loren und Kelly. Nur wir fünf. Wir stimmen ab, bevor wir etwas unternehmen. Zuvor sind wir übereinkommen, dass die Entscheidung, den Effektor einzuschalten,

die Zustimmung von uns allen verlangt. Eine Nein-Stimme läuft auf ein Veto hinaus. Offenbar müssen wir heute Nacht abstimmen. Bald.

Noch hat eine Abstimmung darüber, ob wir den Effektor verwenden sollen, keinen Sinn, denn ich würde mit Nein stimmen. Wir können nicht einschreiten, um St. Louis zu retten. Es gibt noch immer die Möglichkeit, dass damit alles vorbei ist. Wenn Washington entscheidet, den Angriff auf St. Louis ohne Vergeltungsmaßnahmen hinzunehmen, brauchen wir den Effektor nicht einzuschalten. Das wäre eine große Erleichterung für uns alle. Auf diese Weise müssen wir es sehen. Wir warten bis nach der Explosion der Rakete. Wir warten und warten. Wenn Amerika protestiert, ohne einen Gegenangriff zu starten, brauchen wir nicht abzustimmen. In dem Fall muss niemand sagen, wie er oder sie gestimmt hätte. Dann können wir den Rest unseres Lebens mit ruhigem Gewissen verbringen, weil wir die Macht, die in unsere Hände fiel, unangetastet ließen, eine Macht, die die Welt in Dunkelheit stürzen kann. Dann werden wir uns immer fragen, was geschehen wäre, wenn wir ein paar Leben in einer Stadt des Mittelwestens gerettet, dafür aber die ganze Welt grundlegend verändert hätten. Wir könnten bei einem Bier in Cornell darüber reden.«

Ihm gingen die Worte aus. Er hätte überhaupt nichts sagen müssen, das wussten sie alle.

Für einen langen Moment herrschte Stille und dann raschelte es, als Claymore umblätterte.

Homer fiel noch etwas ein. »Wenn wir abstimmen müssen, und ich hoffe, das ist nicht der Fall, aber wenn uns die Umstände zu einer Abstimmung zwingen, so möchte ich fragen ...«

Albert hob die Hand. Er hatte das Ohr am Empfänger und sein Blick ging ins Leere. »Sie starten«, sagte er.

»Was?«, fragte Loren fassungslos. »Wer startet? Wir?«

»Der Präsident hat den Befehl gegeben. Amerika schlägt zu.«

»Aber das kann doch nicht sein! Sie müssen warten, bis die Rakete St. Louis trifft. Vielleicht hält der Abwehrschild. Oder die Kubaner überlegen es sich im letzten Moment anders und lassen die Rakete ins Meer stürzen. Oder sie explodiert überhaupt nicht. Es ist zu früh für eine Reaktion.«

Albert zuckte die Schultern.

Homer sah auf die Uhr. »Wir stimmen jetzt ab«, sagte er. »Es bleiben noch neun Minuten. Wenn wir alle mit Ja stimmen, können wir handeln, noch bevor die Rakete St. Louis erreicht. Dann retten wir auch das Leben der dortigen Menschen, was alles leichter macht.«

»Es wird gestartet«, sagte Albert. »StratCom bestätigt, dass sich die erste Rakete auf den Weg macht ... und jetzt die zweite, von einem U-Boot aus. Es hat begonnen. Weitere Starts werden gemeldet ...«

»Wir stimmen ab.« Homer und seine Gruppe wichen beiseite, weg von Albert und Maria. Eine symbolische Trennung. »Ja bedeutet, dass wir den Effektor einschalten. Nein bedeutet, dass wir nichts unternehmen. Ich stimme ...«

»Warte!«, sagte Loren. Er erinnerte sich an die letzte Abstimmung. Alle hatten sofort ihre Stimme abgegeben, mit Ausnahme von Sonia; letztendlich war es also ihre Stimme gewesen, die den Ausschlag gegeben hatte. Loren wollte nicht, dass sich so etwas wiederholte. »Kleine Zettel«, sagte er. »Wir schreiben unsere Stimme auf. Damit niemand der Letzte ist und den ganzen Druck fühlen muss.«

Auf dem Tisch lag ein Block mit gelben Haftzetteln. Loren riss einen für jeden von ihnen ab. Es gab Stifte und jeder nahm einen. Sonia holte einen aus ihrer Handtasche. Loren schrieb »Ja« auf seinen Zettel und sammelte dann die anderen ein. Er klebte sie an seinen Ärmel, in einer Reihe: alles Ja-Stimmen. Sonias Ja war so klein geschrieben, dass man genau hinsuchen musste, um es zu erkennen: zwei winzige Buchstaben, kaum einen halben Zentimeter groß.

»Alle haben mit Ja gestimmt«, sagte er.

Homer nickte. »Ich schalte den Effektor selbst ein.«

»Noch sieben Minuten«, sagte Albert.

Edward hatte den verzierten Eichenholzkasten mitgebracht. Er stellte ihn auf den Tisch, öffnete ihn und trat zurück. Stille herrschte. Homer ging allein zu dem Kasten und blickte darauf hinab.

»Es befindet sich ein Schiebeschalter an der Seite«, sagte Loren.

»Ich weiß, ich weiß.«

Alberts Stimme kam wie aus weiter Ferne. »Noch sechs Minuten«, sagte er. »Was nicht heißt, dass ich drängen möchte.«

»Ich weiß«, erwiderte Homer.

Es wäre Loren lieber gewesen, wenn Maria jetzt neben Homer gestanden hätte; er sollte jetzt nicht so allein sein. Doch Maria war tief in den weißen Sessel gesunken und hatte den Kopf zur Seite gedreht.

Kelly trat vor, griff mit beiden Händen nach Homers linker Hand und drückte ihre Wange an seine. Loren glaubte zu sehen, dass sie ihm etwas zuflüsterte, aber er hörte nichts. Homer nickte und streckte die rechte Hand nach dem Schalter aus. Loren reckte den Hals. Hatte er den Effektor eingeschaltet? Homer wirkte wie erstarrt.

»Wie viele Menschen leben in St. Louis?«, fragte Edward. »Drei Millionen? Homer, in den nächsten Minuten rettest du genug Menschen, um die Entscheidung zu rechtfertigen. Innerhalb der nächsten Stunde wirst du Dutzende von Millionen Leben gerettet haben, weitaus mehr, als durch den Effekt verlorengehen.«

»Ich weiß«, sagte Homer. »Also tue ich es.« Er betätigte den Schiebeschalter und trat zurück. Die anderen beugten sich vor. Der Schalter leitete Strom in den kleinen, einem Maser ähnelnden Generator und löste die mechanische Arretierung,

die das freie Schweben der Karte verhinderte. In der Mitte des Apparats glühte es rosarot. Die Karte begann sich zu drehen und suchte nach dem magnetischen Nordpol. Sie drehte sich über den Norden hinaus, kehrte dann quälend langsam zu ihm zurück und verharrte schließlich. Loren blickte aus dem Fenster. Nichts war geschehen.

»Vielleicht ist der Magnet ...«, begann er.

Das Licht im Zimmer wurde schwächer. Es ging nicht einfach aus, wie bei einem plötzlichen Stromausfall; es wirkte eher, als würde jemand einen Dimmer herunter drehen. Als es im Zimmer ganz dunkel geworden war, sahen sie aus dem Fenster. Auch in der Stadt breitete sich Dunkelheit aus – nach einigen Sekunden waren überhaupt keine elektrischen Lichter mehr zu sehen. Eine Zeit lang blieb es still, bis Albert das Schweigen brach. »Drei Minuten bis zum Einschlag der Rakete in St. Louis.« Er hielt sich noch immer den StratCom-Apparat ans Ohr. Das Gerät lief mit Batterie, war also nicht vom Effektor betroffen. Der StratCom-Sender befand sich in einem Satelliten, außerhalb des irdischen Magnetfelds.

Sie wandten sich alle dem Fenster zu. Claymore stand auf und kam näher. »Sieh nur«, sagte er und winkte Homer nach vorn. »Ich hab's dir ja gesagt. Es ist eine andere Farbe.«

Der Nachthimmel hatte einen Hauch von Rosarot. Es sah wie die Nordlichter aus, die Aurora Borealis, aber das schwache Leuchten zeigte sich im Süden.

»Es ist eine andere Farbe«, wiederholte Claymore.  
»Pink.«

»Ja, stimmt«, sagte Homer.

Loren holte tief Luft. »Es ist ein Uhr. Wird etwas durchgegeben?«

Alle Blicke richteten sich auf Albert. Er drückte sich den Empfänger noch etwas fester ans Ohr und schüttelte den Kopf. Dann starre er wieder ins Nichts. »Moment ... Es heißt, der Schild habe gehalten. Ja, der Schild habe gehalten und St.

Louis sei nicht zerstört. Es gibt Beobachter unweit der Stadt und sie melden keine Explosion.« Albert sah die anderen an. »Sie glauben, es liegt am Raketenabwehrschild.«

»Oh«, sagte Homer. »Ihnen dürfte bald klarwerden, was geschehen ist.« Er setzte sich auf die Armlehne von Marias Sessel. Sie sah noch immer zur Seite.

»Es werden die Namen der Personen genannt, die angeblich St. Louis gerettet haben«, sagte Albert. »Armitage und seine Leute ... und Curly Burlingame. Curly Burlingame?«

»Ein wahrer amerikanischer Held«, sagte Edward.

»Jetzt werden einige Stromausfälle in den Vereinigten Staaten gemeldet«, fuhr Albert fort. »Keine große Sache, heißt es. Die Rede ist von mutmaßlicher Sabotage, aber nur Einzelfälle.«

Homer lächelte grimmig. »Sabotage, ja. Einzelfälle, nein.«

»Stromausfälle auch in Europa. Sie wissen noch nicht, was sie davon halten sollen.«

Homer winkte geistesabwesend. »Schalt aus, Albert. Worauf es jetzt ankommt, passiert nicht dort draußen, sondern hier drinnen.«

Albert legte den StratCom-Empfänger auf den Couchtisch und sah wieder aus dem Fenster. Es gab überhaupt kein künstliches Licht mehr, nur Sterne und das fahle rosarote Leuchten, wie das schwache Licht etwa eine Stunde vor Sonnenaufgang. Aber es ließ sich in allen Richtungen beobachten und war am südlichen Horizont ein wenig stärker.

»Meine Güte«, sagte Albert. »Was haben wir getan?«

Homer saß in der Dunkelheit. »Was haben wir getan? Was habe ich getan? Wir haben etwa acht Millionen Menschen zum Tod verurteilt – sie werden im Lauf der nächsten Monate sterben. Acht Millionen.« Er sprach leise, schwieg einige Sekunden und fügte dann noch leiser hinzu: »Im Vergleich mit uns war Hitler ein Dilettant.«

Loren hielt den Atem an. Kelly beugte sich zu Homer

hinab, streckte die Hände nach seinen Seiten aus und ... kitzelte ihn. Homer war unglaublich kitzlig. Er zuckte heftig zusammen. »Dummer alter Kerl«, sagte Kelly. »Du hast gerade St. Louis gerettet und sechzig Millionen Menschen überall auf der Welt. Das geht aus unseren Berechnungen hervor. Du hast die Atmosphäre der Erde vor radioaktiver Verseuchung bewahrt. Vielleicht hast du sogar das ganze Leben auf diesem Planeten gerettet.«

»Es stimmt, Homer«, sagte Loren. »Du bist der größte Held aller Zeiten.«

»Aber all das Sterben, das jetzt beginnt ...«, wandte er ein.

»Daran ist jemand anderer schuld.« Edward legte Homer den Arm um die Schulter. »Rupert Paule. Er und General Simpson und all die anderen. Es ist ihre Schuld, Homer.«

Homer nickte, wirkte aber nicht sonderlich überzeugt.

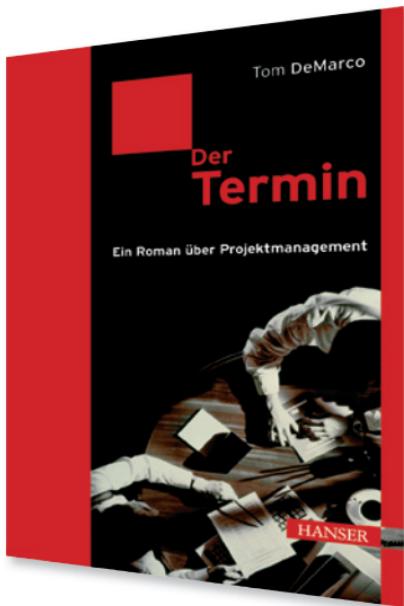
Loren löste die Batterie vom Effektor und sah seine Annahmen bestätigt, als das winzige rosarote Licht in der Kartenmitte blieb - es bezog seine Energie vom irdischen Magnetfeld. Der kleine Apparat auf der Karte war nötig für die Übertragung der Störung, die den Effekt erhielt. Solange er aktiv und ausgerichtet blieb, dauerte der Effekt an. Loren entfernte auch die Arretierung, damit sie nicht unabsichtlich ausgelöst werden konnte, schloss den Kasten und schloss ihn ab.

Edward verteilte Taschenlampen aus einer Box mit Vorräten, die sie Stunden zuvor hochgetragen hatten. Außerdem gab er jedem eine Liste mit detaillierten Anweisungen für die nächsten Schritte.

»Es wartet viel Arbeit auf uns, Leute, und wir haben nur ein paar Stunden Zeit, alles zu erledigen. Packen wir's an.«

(Ende 15. Kapitel)

## Weitere Bücher von Tom DeMarco



Tom DeMarco  
**Der Termin**  
ISBN 978-3-446-41439-6



Tom DeMarco, Timothy Lister  
**Wien wartet auf Dich!**  
ISBN 978-3-446-43895-8

Tom DeMarco, Tim Lister  
**Bärentango**  
ISBN 978-3-446-22333-2

Tom DeMarco  
**Spielräume**  
ISBN 978-3-446-21665-5

# »Der Weltuntergang steht bevor, aber nicht so, wie Sie denken. Dieser Krieg jagt nicht alles in die Luft, sondern schaltet alles ab.«

Im obersten Stock der Cornell University's Clark Hall stehen der Physiker Homer Layton und seine drei jungen Assistenten vor einem Durchbruch, der es ermöglicht, die Zeit etwas langsamer ablaufen zu lassen. Sie vermuten, dass der sogenannte Layton-Effekt keinen praktischen Nutzen haben wird, rechnen aber damit, dass die von ihnen geplante Abhandlung einem Paukenschlag in der Welt der theoretischen Physik gleichkommen wird. Doch dann bemerkt Loren Martine, jüngstes Mitglied von Homers Team, etwas Seltsames: Wird die Zeit verlangsamt, reicht die in Brennstoffen gespeicherte Energie nicht mehr für ein plötzliches Feuer. Dinge können noch immer brennen, wenn auch langsamer, aber nichts kann mehr explodieren. Die Wissenschaftler stellen sich eine Art Layton-Effekt-Taschenlampe vor, die das Abfeuern einer Waffe verhindert. Ihnen wird klar, dass man auch die Explosion einer Bombe oder gar einen ganzen Krieg verhindern könnte.



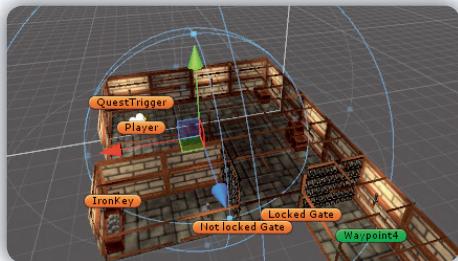
Sie möchten mehr über Tom DeMarco und  
seine Bücher erfahren.  
Einfach Code scannen oder reinklicken unter  
[www.hanser-fachbuch.de/special/demarco](http://www.hanser-fachbuch.de/special/demarco)

# SPIELE ENTWICKELN MIT UNITY //

- Für alle, die ihr eigenes Spiel entwickeln wollen; Vorkenntnisse sind nicht erforderlich
- Auf DVD: das Game aus dem Buch mit allen Ressourcen; weitere Beispiele; Videotutorials (Gesamtdauer: 65 Minuten)
- Im Internet: Zusatzmaterialien und Aktualisierungen



Entwerfen Sie komplexe 3D-Welten.  
(Szene aus einem der Video-Tutorials)



Kombinieren Sie die unterschiedlichen Tools und Techniken, die Unity bietet. (Grafik aus dem Buch)



Entwickeln Sie zusammen mit dem Autor ein komplettes Spiel. (Szene aus dem Beispiel-Game)



Dipl.-Ing. (FH)

Carsten

**SEIFERT** ist

Spiele- und Softwareentwickler, Blogger und YouTuber.

Der Unity-Community ist er mit seinen Tutorials bestens bekannt. Besuchen Sie ihn auf seiner Webseite [www.hummelwalker.de](http://www.hummelwalker.de).

## AUS DEM INHALT //

- Installation & Oberfläche von Unity 4.6
- Einführung in C#
- Skript-Programmierung
- Objekte in der dritten Dimension
- Kameras: die Augen des Spielers
- Licht und Schatten
- Physik-Engine
- Maus, Tastatur, Touch
- Audio
- Partikeleffekte mit Shuriken
- Landschaften gestalten
- Wind Zones
- GUI: klassische und neue (uGUI) Techniken
- Prefabs
- Internet & Datenbanken
- Animationen
- Künstliche Intelligenz
- Fehlersuche & Performance
- Spiele erstellen & publizieren
- Beispiel: Entwicklung eines 3D-Adventures
- Produktionsprozesse in der Spieleentwicklung

HANSER

