5 Die Spielentwicklung vorbereiten

Bevor wir unseren Prototyp weiter ausbauen und ihn dann zu einem richtigen Spiel weiterentwickeln können, gehen wir also einen Schritt zur Seite und verschaffen uns einen Überblick, was »Spielentwicklung« eigentlich genau umfasst. Tatsächlich umfasst diese Tätigkeit nämlich eine Vielzahl teilweise völlig unterschiedlicher Disziplinen: Game-Design, Programmierung, Grafikdesign, 3D-Modelling und -Animation, Musikkomposition und -produktion sowie Sounddesign, um nur die offensichtlichsten zu erwähnen.

Einerseits macht das die Spielentwicklung sehr interessant. Andererseits stellt es uns als Spielentwickler aber auch vor besondere Herausforderungen – vor allem, wenn wir versuchen, ein komplettes Spiel allein zu entwickeln. Neben den Talenten in den verschiedenen Disziplinen brauchen wir auch eine Vielzahl unterschiedlicher Werkzeuge, mit denen wir diese Talente auf eine Weise zum Ausdruck bringen können, die dann auch in Spielen verwendbar ist.

Der Auswahl der Werkzeuge sollten wir in jedem Fall ausreichend Zeit widmen, da die Entscheidung für ein bestimmtes Werkzeug neben den meistens anfallenden Lizenzkosten vor allem oft einen erheblichen Einarbeitungsaufwand mit sich bringt, der je nach Wert der eigenen Zeit die Lizenzkosten häufig um ein Vielfaches übersteigt. Hinzu kommt, dass Werkzeuge in der gleichen Gattung oft nur teilweise zueinander kompatibel sind, was den Wechsel zu einem späteren Zeitpunkt weiter erschwert. Daher ist es auch empfehlenswert, schon bei der Auswahl der Werkzeuge auf die Kompatibilität mit konkurrierenden Werkzeugen zu achten.

Dieses Kapitel soll Ihnen zunächst einen oberflächlichen, aber umfassenden Eindruck von den verschiedenen Aufgaben in der Spielentwicklung vermitteln. Zu jedem Bereich werden dann die Klassen von Werkzeugen sowie konkrete Beispiele kurz vorgestellt. Damit erhalten Sie auch Einstiegspunkte für eigene tiefergehende Recherchen.

Link auf unity-buch.de

Da es zu jedem Abschnitt in diesem Buch gleich eine Vielzahl von Links auf der Website zum Buch gibt, verwende ich hier nur Fußnoten mit entsprechendem Vermerk, ohne zusätzlich am Seitenrand auf die Links aufmerksam zu machen.

5.1 Regeln: Spielmechanik und Programmierung

Zunächst hat jedes Spiel eine *Spielmechanik*, also eine Menge von Regeln, die bestimmen, welche Aktionen des Spielers welche Konsequenzen innerhalb der Spielwelt haben, die aber durchaus auch die Eigendynamik einer Spielwelt beschreiben können. Dazu gehört selbstverständlich auch, wie das Spiel beginnt und endet sowie unter welchen Umständen der Spieler das Spiel gewinnt oder verliert. Zunächst müssen diese Spielmechaniken definiert werden, was einen wesentlichen Teil des Design-Dokuments ausmacht. Zur Erstellung des Design-Dokuments gibt es verschiedenste Vorlagen und sogar einige spezielle Tools. Man kann aber auch einfach eine beliebige Textverarbeitung verwenden.

Zur konkreten Umsetzung der Spielmechaniken verwenden wir immer irgendeine Art von Programmierung, und dazu brauchen wir eine Programmiersprache. Idealerweise kann man die gleiche Sprache dann auch für alle anderen Programmieraufgaben verwenden, die bei der Spielentwicklung anfallen, egal ob man nun die Logik der Benutzerschnittstelle umsetzt, die Kommunikation mit einem Webserver implementiert, um Highscores abzuspeichern und für alle Spieler verfügbar zu machen, oder ob man gar eine Netzwerkschicht für Multiplayer-Spiele entwickelt.

Auch wenn man bei dem Stichwort »Programmierung« meistens an textuellen Programmcode denkt, gibt es durchaus visuelle Programmiersprachen – auch im Unity-Umfeld.[1] Wir beschränken uns hier aber auf die Sprachen, die Unity direkt mitliefert, also *C#, JavaScript* bzw. *UnityScript* und *Boo*.

Auswahl der Programmiersprache

Die Sprache *Boo* ist eine Programmiersprache mit Python-ähnlicher Syntax und daher sicherlich für all diejenigen besonders interessant, die bereits Erfahrung mit Python haben. Inzwischen kann Boo auch für Android- und iOS-Spiele eingesetzt werden (das war am Anfang nicht so), führt aber dennoch eher ein Schattendasein.[2]

---

1 Im Unity Asset Store gibt es dazu eine eigene Kategorie: *Editor Extensions/Visual Scripting*.

2 Link auf *unity-buch.de:* Falls Sie Boo lernen möchten, hilft Ihnen vielleicht das Thema *Official Boo Scripting Resource Thread* im Unity-Forum, das ich von der Website zum Buch aus verlinkt habe.

JavaScript in Unity sollte eigentlich eher *UnityScript* heißen, und tatsächlich heißt die Sprache auch so,[3] wurde aber lange Zeit von Unity Technologies offiziell als JavaScript bezeichnet. Es handelt sich auch um eine zumindest von der Syntax her an JavaScript angelehnte Sprache, die vor allem Programmierern aus dem Webbereich einen einfachen Zugang in die Programmierung in Unity ermöglichen soll. Da hat die durchaus vorhandene Ähnlichkeit mit dem bekannten JavaScript aus der Webentwicklung und ActionScript aus der Flash-Entwicklung natürlich Vorteile. Ein weiterer Vorteil von JavaScript in Unity ist eine im Vergleich zu C# kompaktere Schreibweise, da beispielsweise auf die Klassendeklaration und Angabe der Vererbung von `MonoBehaviour` verzichtet werden kann. Auch einige Namespaces stehen hier automatisch zur Verfügung und müssen nicht explizit angegeben werden. Genau diese kompaktere Schreibweise ist jedoch aus meiner Perspektive auch der größte Nachteil.

Wenn man nämlich in Unity einsteigt und noch nicht mit Unity vertraut ist, gibt JavaScript-Code dem Einsteiger meistens relativ wenige Anhaltspunkte, um zu verstehen, welche Klassen genau verwendet werden. Mit dieser Information könnte man deutlich leichter in der *Scripting API* nachschlagen, um seine Kenntnisse zu vertiefen.

Ein weiteres Problem für den Programmieranfänger ist, dass JavaScript in Unity eben nicht wirklich JavaScript ist. Daraus resultiert, dass die meisten JavaScript-Tutorials eher Verwirrung stiften, weil sie sich auf die üblicherweise auf Webseiten verwendete Version von JavaScript beziehen, die zwar ähnlich ist, aber doch verschieden. Ist man aber bereits mit JavaScript aus dem Webbereich (oder ActionScript) und mit Unity vertraut, ist JavaScript eine gute Wahl.[4] Früher war ein großer Teil der Standard Assets in JavaScript implementiert, und auch viele Beispiele in der Dokumentation gab es nur in JavaScript. Das hat sich in den letzten Jahren aber zugunsten von C# verändert.[5]

Bei *C#* steht in Unity im Gegensatz zu JavaScript/UnityScript nicht nur C# drauf, sondern es ist auch wirklich C# drin, und zwar genau so, wie man es aus der.NET- und Mono-Welt kennt. Somit kann man sich auch für Unity auf die offizielle Sprachspezifikation beziehen und findet auch ein entsprechendes Programmierhandbuch.[6] Außerdem gibt es eine Vielzahl

3 Der Name »UnityScript« wird für die Projekte auf GitHub und *code.google.com* verwendet, und man findet auch innerhalb des mit Unity installierten Mono-Frameworks die Dateien *UnityScript.dll* und *UnityScript.Lang.dll*.

4 Link auf *unity-buch.de:* Im *Unify Community Wiki* gibt es einen Artikel, der die beiden Dialekte UnityScript (also JavaScript in Unity) und JavaScript (also den Web-Dialekt) direkt gegenüberstellt und vergleicht. Für den Einstieg dürfte dieser Artikel sehr hilfreich sein. Er ist unter dem Titel *UnityScript versus JavaScript* verlinkt.

5 Link auf *unity-buch.de:* Dazu gab es am 03.09.2014 auch ein offizielles Statement von Unity Technologies: *Documentation, Unity scripting languages and you*

6 Link auf *unity-buch.de:* Siehe auch den Link *C#-Sprachspezifikation*. Von dort aus sind das Programmierhandbuch und die Referenz verlinkt.

von Tutorials und anderen Möglichkeiten, die Sprache zu lernen; teilweise auch im Unity-Umfeld.[7]

Obwohl man früher teilweise den Eindruck gewinnen konnte, dass JavaScript die »Unity-Sprache« ist, hat schon 2012 eine informelle Umfrage im Unity-Forum ergeben, dass C# bei den Forumsteilnehmern tatsächlich etwas populärer ist als JavaScript. Von den 482 Teilnehmern an der Umfrage[8] haben 34,4 % angegeben, dass sie ausschließlich C# verwenden, 29,8 %, dass sie ausschließlich JavaScript verwenden, und 28,4 %, dass sie sowohl JavaScript als auch C# verwenden. Es ist zwar möglich, beide Sprachen in einem Projekt zu verwenden, dabei muss man aber aufgrund der Kompilierung in mehreren Schritten aufpassen, welche Klassen von C# aus auf JavaScript-Klassen zugreifen können und anders herum.[9]

Die Popularität der Sprache ist insofern relevant, als man bei einer populäreren Sprache Fragen leichter beantwortet bekommt und man auch leichter Team-Kollegen findet, wenn man die Programmierung nicht komplett allein erledigen möchte. Falls Sie die Programmiersprache Java kennen, ist der Umstieg auf C# auch sehr einfach, da die Sprachen sich sehr ähnlich sind.[10]

Diese Vorteile und das bei JavaScript höhere Risiko, Code zu produzieren, der zwar problemlos kompiliert, aber zur Laufzeit Probleme bereitet, haben mich dazu bewogen, selbst ausschließlich C# zu verwenden und auch als Implementierungssprache in diesem Buch durchgängig C# zu nutzen.

Auswahl der Programmierumgebung

Theoretisch könnte man auch heute noch mit einem einfachen Texteditor den Programmcode schreiben. Puristen mögen mir verzeihen, aber das halte ich für keine gute Idee: Moderne Entwicklungsumgebungen vereinfachen und beschleunigen den Prozess unter anderem durch Hilfsmittel, wie die automatische Vervollständigung des Codes[11], die sofortige Markierung von Syntax-Fehlern und die elegante Einbindung der API-Dokumentation. Eine solche Entwicklungsumgebung, die bei Unity mitgeliefert wird, haben wir bereits kennengelernt. Sie ist jedoch eine eigenständige und von Unity unabhängige Anwendung: *MonoDevelop*.[12]

---

7 Link auf *unity-buch.de:* Zum Beispiel das *CSharp Unity Tutorial* im Unify Community Wiki.

8 Link auf *unity-buch.de:* Siehe auch *Boo, C# and JavaScript in Unity – Experiences and Opinions*. Abgerufen am 21.01.2012.

9 Link auf *unity-buch.de:* Siehe *Special Folders and Script Compilation Order*.

10 Interessanterweise sind C# und Java sich viel ähnlicher als JavaScript und Java.

11 »Code completion« wäre hier auch in einem deutschen Umfeld der gängigere Begriff; IntelliSense ist Microsofts Variante davon.

12 Allerdings ist die mit Unity ausgelieferte Version von MonoDevelop speziell für Unity angepasst.

MonoDevelop hat einige Vorteile. Es unterstützt alle von Unity unterstützten Programmiersprachen und Plattformen. Man kann mit MonoDevelop also Scripts in C#, JavaScript und Boo bearbeiten, und zwar sowohl unter Mac OS X als auch unter Windows.[13]

Ich persönlich bevorzuge – obwohl ich selbst an einem Mac arbeite[14] – *Microsoft Visual Studio*. Die wesentlichen Nachteile sind hier, dass die Verwendung auf *Mac OS* etwas umständlich ist und Visual Studio zwar JavaScript unterstützt, aber nicht den in Unity verwendeten Dialekt; von Boo ganz zu schweigen. Das spielt aber natürlich nur dann eine Rolle, wenn man auch mit diesem Betriebssystem oder diesen Sprachen arbeiten möchte. Mit Visual Studio Express gibt es zwar eine kostenfreie Lizenz, diese ist aber im Vergleich zu den kostenpflichtigen Versionen deutlich eingeschränkt.

Der wesentliche Vorteil von Visual Studio besteht darin, dass dieses Werkzeug schon seit vielen Jahren entwickelt und konsequent verbessert wird und auch durch Plug-ins von Drittherstellern erweiterbar ist. Bei MonoDevelop hat man demgegenüber oft das Gefühl, noch mit Kinderkrankheiten konfrontiert zu werden – was aber natürlich mit der Zeit ein immer kleineres Problem wird. Praktischerweise erzeugt Unity für MonoDevelop auch mit Visual Studio kompatible *Solutions*.[15] Das heißt, zumindest technisch spricht nichts dagegen, beide Entwicklungsumgebungen parallel zu verwenden.

Falls Sie doch lieber einen reinen Texteditor verwenden, wäre *Sublime Text* vielleicht eine interessante Alternative. Da gibt es noch viele Editoren mehr, und Sie können natürlich auch Ihren Lieblingseditor weiterverwenden. Letztlich müssen wir nur einen Code-Editor bzw. eine Entwicklungsumgebung installieren und verwenden.

So ausgerüstet, können wir dann schon die meisten Herausforderungen bei der Programmierung angehen.[16] Würden Computerspiele allerdings nur aus der programmierbaren Logik ihrer Spielmechaniken bestehen, wären sie wohl kaum so populär, wie sie es tatsächlich sind. Tatsächlich sprechen

---

13 Außerdem läuft MonoDevelop unter Linux – tatsächlich lief es dort sogar zuerst und wird dort am besten unterstützt. Da aber Unity nicht unter Linux läuft, ist das in diesem Kontext nur eine Fußnote wert.

14 Ich verwende VMware Fusion, um Visual Studio als Windows-Anwendung neben meinen Mac-Anwendungen laufen zu lassen. Wie man das einrichtet, ist im Unify Community Wiki beschrieben: *Setting up Visual Studio for Unity on Mac,* verlinkt von *http://unity-buch.de*.

15 Eine Solution in Visual Studio oder auch in MonoDevelop sind mehrere Projekte, die in einer übergreifenden Datei zusammengefasst sind. Sie wird von Unity erzeugt, wenn Sie auf eine Scriptdatei doppelklicken oder auch auf eine Konsolenmeldung, die sich auf ein Script bezieht – dabei wird dann normalerweise auch MonoDevelop gestartet, außer Sie haben eine andere Entwicklungsumgebung in den Preferences eingestellt. Sie können aber auch über das Menü *Assets/Synchronize MonoDevelop Project* die Solution erzeugen. Sie wird dann auch bei jeder Änderung automatisch synchronisiert.

16 Für Softwaredesign bzw. Softwarearchitektur gibt es außerdem noch UML-Tools, und bei größeren Spielprojekten würde ich so etwas auch verwenden.

Computerspiele auf direkte Art und Weise die Sinne Hören und Sehen an, auf indirekte Weise über die Erfahrung auch das Fühlen (üblicherweise natürlich nicht im Sinne von haptischer Wahrnehmung, sondern im Sinne von Gefühlen und Emotionen wie Freude oder Trauer).

5.2 Spiel an Augen: Bildschirmdarstellung

Was wir am Bildschirm sehen, lässt sich zunächst in Spielwelt und virtuelle Benutzerschnittstelle unterteilen.[17]

5.2.1 Virtuelle Benutzerschnittstelle: Pixel- und Vektorgrafik

Die virtuelle Benutzerschnittstelle umfasst Bedienelemente (wie beispielsweise Buttons, die der Spieler anklicken kann) sowie die Darstellung von Aspekten des Spielzustands (wie z. B. die Punktezahl oder Lebensenergie des Spielers). Üblicherweise bieten Spiele auch ein *Hauptmenü,* aus dem heraus das Spiel gestartet werden kann und über das ggf. auch Einstellungen vorgenommen werden können. Um die virtuelle Benutzerschnittstelle umzusetzen, benötigen wir Grafiken und meistens auch Texte, die über die Verwendung von Schriftarten letztlich auch in Grafiken umgewandelt werden. Generell sind grafische Darstellungen, z. B. Icons und Fortschrittsbalken, nach Möglichkeit gegenüber Texten zu bevorzugen, weil sie weniger Aufmerksamkeit zum Verständnis erfordern und somit weniger vom eigentlichen Spielgeschehen ablenken. Außerdem sind Symbole meistens international verständlich und müssen nicht internationalisiert werden. (Mehr zu diesem Thema folgt in Abschnitt *8.1.2, Über Internationalisierung und Lokalisierung.*)

Bei der Gestaltung von Benutzerschnittstellen ist es oft vorteilhaft, zunächst die Anordnung der verschiedenen Bedienelemente abstrakt vorzunehmen. Diesen Prozess bezeichnet man als *Screendesign*. Zu diesem Zweck gibt es verschiedene Werkzeuge. Ich habe sehr gute Erfahrungen mit *Balsamiq Mockups* gemacht.

Grafiken können prinzipiell pixelbasiert oder vektorbasiert sein. Vektorgrafik hat den Vorteil beliebiger Skalierbarkeit und ist dabei meistens auch speichereffizienter – wird aber von Unity derzeit zumindest direkt noch nicht unterstützt.[18] Zur Erstellung von Pixelgrafiken verwendet man Bildbearbeitungsprogramme.[19] Kostenfrei und auf allen Plattformen verfügbar ist

17 Vgl. Schell, Jesse: The Art of Game Design, 2008. S. 223 ff.

18 Link auf *unity-buch.de:* Eine erwähnenswerte Erweiterung, die über den Unity Asset Store verfügbar ist, ist *RageSpline* von Juhakiili Oy. Eine komplett vektorbasierte GUI-Lösung wäre *NoesisGUI*.

19 Link auf *unity-buch.de:* Eine Auswahl von Vektorgrafik-Programmen finden Sie auf der Website zum Buch.

Gimp. *Pixelmator* – eine echte Mac-Anwendung – ist zwar nicht kostenfrei, aber auch recht erschwinglich. Der große (und teure) Platzhirsch ist in diesem Bereich nach wie vor *Adobe Photoshop,* und bei der Verwendung mit Unity hat Photoshop vor allem den Vorteil, dass das Dateiformat PSD direkt in Unity unterstützt wird.[20] Photoshop gibt es für Windows und Mac OS X.

5.2.2 Spielwelt: 2D, 3D, Modelling, Texturing und Animation

Während die virtuelle Benutzerschnittstelle zumindest im Prinzip bei allen Spielen ähnlich ist, gibt es völlig unterschiedliche Spielwelten: Vergleichen Sie nur beispielsweise 2D-Spiele mit einer eher abstrakten Welt, die auf einen Bildschirm passt (z. B. Pac Man, Tetris oder auch Schach), mit 2D-Spielen mit größeren Spielwelten, die Scrolling benötigen (z. B. Super Mario Bros), und diese wiederum mit 3D-Spielen, bei denen die Spielwelten zumindest vom räumlichen Empfinden her unserer physischen Umgebung entsprechen. Je nach Art der Spielwelt kommen auch völlig unterschiedliche Technologien zum Einsatz, um die jeweilige Spielwelt zu erschaffen.

2D-Spielwelten – Bitmaps oder Vektorgrafiken

Wie wir bereits gesehen haben, unterstützt Unity auch die Entwicklung von 2D-Spielen. Dazu brauchen wir Bitmap-Grafiken, die wir in Unity als Sprites verwenden können. Im Hinblick auf die Wiederverwendbarkeit dieser Assets kann es sinnvoll sein, vektorbasierte Tools zu verwenden und die entsprechenden Grafiken dann lediglich für Unity zu rastern (also aus der Vektorgrafik eine Bitmap mit einer spezifischen Auflösung zu erstellen).

Neben den bereits behandelten Bildbearbeitungsprogrammen gibt es auch spezielle Werkzeuge, mit denen Sprites und Sprite-Animationen komfortabel erstellt werden können.[21]

3D-Spielwelten – Modelling, Texturing und Animation

Ursprünglich war Unity ja eine 3D-Game-Engine, und von daher werden wir mit Unity auch oft Spiele entwickeln, die über 3D-Spielwelten verfügen. Eine sehr einfache und kleine 3D-Spielwelt haben wir ja sogar schon direkt in Unity erschaffen – und sind dabei aber recht schnell an die Grenzen dessen gestoßen, was mit Unity möglich ist. Hierzu ist es wichtig zu verstehen, dass Unity in keiner Weise als Modelling-Tool gedacht ist. Die *Scene View* und *Hierarchy View* sind eher als Level-Editor zu verstehen, mit dem

20 Prinzipiell unterstützen auch andere Bildbearbeitungsprogramme (wie Gimp oder Pixelmator) das PSD-Format. Da es sich aber um ein proprietäres Format von Adobe handelt, ist die Unterstützung hier nie optimal.

21 Link auf *unity-buch.de:* Einen kurzen Einstieg in diese Thematik sowie Beispiele für Tools finden Sie in dem Artikel *2D Game Sprite and Animation Software*.

man in anderen Tools erstellte Assets platzieren kann. Eine nennenswerte Ausnahme ist die in Unity integrierte Terrain Engine, mit der man Landschaften modellieren kann. Sie wird in Kapitel *15, Terrain Engine: Eine Landschaft bauen,* behandelt. Um herauszufinden, welche Werkzeuge wir für die Erstellung von 3D-Spielwelten brauchen, sollten wir uns erst einmal klar machen, was eine Spielwelt in einem 3D-Spiel eigentlich ausmacht.

Begriffe aus der Welt der 3D-Spiele

Grundsätzlich besteht eine 3D-Welt aus dreidimensionalen Formen, die wir als *3D-Modelle* bzw. verkürzt einfach nur als *Modelle* bezeichnen. Auch die Bezeichnung *Mesh* haben Sie dafür schon kennengelernt. Diese Modelle haben Oberflächen, deren Eigenschaften über *Materialien* definiert sind, die in den allermeisten Fällen über farbige Oberflächenstrukturen verfügen, die wir als *Texturen* bezeichnen.

Wie diese Texturen – bei denen es sich letztlich um einfache Pixelgrafiken handelt – auf die Oberfläche der Modelle gelegt werden, bestimmt die sogenannte *UV-Map,* die einfach gesagt die 2D-Koordinaten in der Textur-Grafik auf die 3D-Koordinaten des Modells abbildet. Neben den Farben für die Oberfläche gibt es auch verschiedene andere Eigenschaften der Oberflächen, die über Texturen definiert werden können. Beispielsweise gibt es *Specular Maps,* bei denen jeder Pixel in der Textur bestimmt, wie stark das Licht an dieser Stelle auf dem Modell reflektiert wird, oder *Bump Maps,* mit denen über die Textur feine Erhöhungen und Vertiefungen auf das eigentliche Modell gelegt werden können, um beispielsweise eine Riffelung darzustellen, ohne die einzelnen Riffeln modellieren zu müssen. Im Kontext von *physikbasierten Shadern* gibt es auch Texturen, die bestimmen, wie sehr die Oberfläche an bestimmten Stellen eher einem Metall oder einem Nichtleiter ähnelt oder wie glatt oder rau die Oberfläche ist – oder auch, wie sehr die Oberfläche selbst Licht abgibt.

Wie die verschiedenen Texturen eingesetzt werden können, bestimmen die schon oben erwähnten *Materialien*[22], die Sie ja auch schon als Eigenschaft der `MeshRenderer`-Komponente in Unity kennengelernt haben und die ihrerseits wiederum durch Shader ihre Eigenschaften erhalten. Über Materialien bzw. Shader wird auch bestimmt, wie sich Licht auf die Modelle auswirkt: Die Modelle können damit transparent gemacht werden, und sogar beliebige Deformationen sind möglich. *Bump Maps* sind hier ein einfaches Beispiel. Häufiger kommen *Normal-Maps* zum Einsatz, bei denen die Farbwerte R, G, B als Richtungsvektoren für die *Normale*[23] an dieser Stelle der Oberfläche interpretiert werden.

---

22 Siehe Abschnitt *6.1, Eigene Materialien erstellen und verwenden*, ab Seite 129.

23 Die Normale bzw. der Normalenvektor ist der Vektor, der von einer Fläche (wenn wir im dreidimensionalen Raum sind) orthogonal »nach außen« zeigt.

Eine spezielle Art von Texturen sind sogenannte *Skyboxes*. In Unity handelt es sich dabei üblicherweise um sechs Texturen für die verschiedenen Richtungen (vorne, hinten, rechts, links, oben und unten), die praktisch von innen auf einen immer maximal weit entfernten Würfel projiziert werden. Seit Unity 5.0 können Skyboxes auch in verschiedenen für Skyboxes üblichen Formaten (Cubic, Cylindrical, Spheremap) direkt in Unity importiert werden. Damit kann ressourcensparend der Himmel oder auch ein Gebirge am Horizont dargestellt werden, ohne dass man diese tatsächlich als 3D-Modelle in die Spielwelt einfügen müsste. Seit Unity 5 können Skyboxes auch im Kontext von *Global Illumination* dazu verwendet werden, die Szene realistisch auszuleuchten.

Mit den soeben im Schnelldurchlauf zusammengefassten Konzepten könnten wir eine statische Spielwelt erstellen, in der sich nichts bewegt. Das wäre für ein Spiel aber ziemlich langweilig. Bewegung kommt durch *Animation* ins Spiel – und das ist ein weiteres weites Feld, das hier nur kurz angerissen werden kann: Bereits gesehen haben Sie, dass sich Objekte als solche in der Spielwelt bewegen können – wie der Tracer in unserem Prototyp. Diese Art von Animation kann also durch ein wenig Scripting und die Verwendung der Physik-Engine bereits leicht in Unity umgesetzt werden. Sie haben in Abschnitt *2.3.4, Einfache 2D-Animationen mit Mecanim und Dope Sheet erstellen,* auch schon den in Unity eingebauten Animationseditor kennengelernt, mit dem einfache Animationen direkt in Unity umgesetzt werden können. Sie können damit sogar Animationen aus externen Tools bearbeiten. Später, in Abschnitt *11.2, Das Startmenü aufhübschen,* werden Sie dieses Unity-Tool noch besser kennenlernen.

Möchten wir aber menschen- oder tierähnliche Charaktere in unserer Spielwelt auftreten lassen, so brauchen wir *Character Animation,* also die Möglichkeit, diese Gestalten mit Persönlichkeit auf realistische Art und Weise zu bewegen. Ein wesentlicher Arbeitsschritt hierzu ist das *Rigging,* bei dem ein bestehendes Modell, beispielsweise von einem Menschen, um ein Skelett erweitert wird, das bestimmt, wie die verschiedenen Gliedmaßen sich zueinander bewegen können. Damit können zu dem Modell *Animationen* erzeugt werden, also Bewegungsabläufe wie Gehen, Springen oder Tanzen. Häufig werden solche Bewegungsabläufe über *Keyframe Animation* (zu Deutsch: Schlüsselbildanimation) erfasst, d. h., verschiedene Schlüsselstellungen werden nacheinander gespeichert und die Bewegung dazwischen wird dann interpoliert. Eine andere Möglichkeit, die Bewegungsabläufe direkt aus der realen Welt zu erfassen, ist *Motion Capture*. Dabei werden je nach Verfahren die Bewegungen entweder mit mehreren Kameras erfasst und durch die Verwendung von Markern die Berechnung der Bewegung im Raum ermöglicht, aus der dann wiederum eine Animation erstellt werden kann; oder die Bewegungen werden durch Verwendung spezieller Anzüge direkt erfasst. Mit *Mecanim* können dann wiederum in Unity verschiedene

solche Animationen in komplexen Zustandsgraphen organisiert und anhand beliebiger Parameter ineinander überblendet werden.

Character Animation geht schließlich bis hin zur *Gesichtsanimation* (Facial Animation), die die Darstellung von Mimik und lippensynchronisierter Sprache ermöglicht. Zur Gesichtsanimation kann einerseits ebenfalls eine Art Skelett verwendet werden, ein sogenanntes *Facial Rig,* häufig werden aber auch sogenannte *Morph Targets* oder *Blend Shapes* verwendet. Dabei werden direkt im Modell verschiedene Versionen gespeichert, beispielsweise das Gesicht mit einer hochgezogenen Augenbraue oder der Mund in der Stellung bei dem Phonem »O«. Diese werden seit Unity 4.3 sogar nativ unterstützt, also ohne dass dazu zusätzliche Add-ons notwendig wären.[24]

Werkzeuge zur Erstellung von 3D-Spielwelten

Es gibt eine Vielzahl von Modelling-Tools, die zur Erstellung von Modellen und Animationen für Unity verwendet werden können. Ein kostenfreies und dennoch sehr mächtiges Werkzeug ist *Blender,* zu dem es auch ein empfehlenswertes Buch im dpunkt.verlag gibt.[25] Am anderen Ende der Preisskala befinden sich Tools wie *Maya* und *3D Studio Max,* die auch in vielen großen Spielproduktionen zum Einsatz kommen. Für Mac-User, die in die Welt des Modelling einsteigen, ist *Cheetah 3D* zu empfehlen, da es gleichzeitig einen günstigen Preis hat und relativ einfach zu erlernen ist. Ein Werkzeug, das speziell auf das Erstellen von Character-Animationen zugeschnitten ist, wäre *MotionBuilder*. Eine relativ erschwingliche Möglichkeit, unter Verwendung von zwei Microsoft-Kinects Animationen mittels Motion Capturing zu erstellen, ist *iPi Desktop Motion Capture*. Gesichtsanimation mit Lippensynchronisation zu Audiodateien ist beispielsweise mit *FaceFX* möglich.

Einige Modelling-Tools (z. B. auch *modo*) ermöglichen es, Texturen direkt auf das 3D-Modell zu »malen«. Texturen können aber natürlich auch in Bildbearbeitungsprogrammen erstellt werden, und häufig werden Bildbearbeitungsprogramme verwendet, um Texturen aus verschiedenen Quellen zu optimieren oder Fehler darin zu korrigieren. Als Grundlage für Texturen kann man – je nach visuellem Stil der 3D-Spielwelt – auch Fotos verwenden. Wenn sich die Textur wiederholen soll, z. B. für Kacheln oder eine Wand aus Ziegeln, ist es wichtig, dass die Ränder sich nahtlos aneinanderfügen (Stichwort: *Seamless Textures*). Dazu gibt es unter anderem eine prozedurale Textur (*Substance*) von Allegorithmic: *Bitmap2material*. Unity unterstützt Substances, und wenn Sie diese selbst erstellen wollen, können Sie dazu *Substance Designer* verwenden. So ist es prinzipiell möglich, auch ohne Bildbearbeitungsprogramm an hochwertige Texturen zu kommen, die perfekt an die jeweilige Spielwelt angepasst werden können und zusätzlich

24 Link auf *unity-buch.de: Animation Blend Shapes* im Unity Manual.
25 Wartmann, Carsten: Das Blender-Buch, 5. Auflage, 2014.

auch noch deutlich weniger Speicherplatz verbrauchen. Ein relativ neues Tool ist der *Substance Painter,* mit dem Substances und andere Materialien direkt auf ein 3D-Modell aufgetragen werden können. Hier können mit einer Partikel-Engine auch Abnutzungseffekte sehr einfach simuliert werden. Außerdem unterstützt Substance Painter die verschiedenen Workflows für physikbasiertes Shading sehr gut.

Skyboxes kann man theoretisch auch mit einem Fotoapparat erstellen. Einfacher ist aber auch hier die Verwendung von Software. Zwei Werkzeuge, die sehr gut zum Berechnen von Skyboxes für typische Himmel geeignet sind, wären *Terragen 2* und *Vue*. Wer Skyboxes für Weltraumspiele selbst erzeugen möchte, findet in *Spacescape* einen verlässlichen Verbündeten.

5.3 Spiel an Ohren: Musik und Soundeffekte

Die Atmosphäre in einem Spiel wird vor allem durch musikalische Untermalung und Soundeffekte erzeugt. Daher ist der Audiobereich keinesfalls weniger wichtig als der visuelle Bereich oder die Programmierung. Jesse Schell, ein bekannter Game-Designer und Buchautor, berichtet sogar von einer Studie, in der Spieler zwei Versionen des gleichen Spiels bezüglich der visuellen Qualität beurteilen sollten. Der einzige Unterschied zwischen den beiden Versionen war, dass die eine über bessere Audioqualität verfügte als die andere. Interessanterweise bewerteten die Spieler die visuelle Qualität entsprechend der Audioqualität. Das heißt, das Spiel mit dem schlechteren Ton »sah« für die Spieler schlechter aus. Daher empfiehlt Jesse Schell sogar, die Musik für das Spiel so früh wie möglich im Entwicklungsprozess auszuwählen, idealerweise sogar ganz am Anfang.[26] Dieser Empfehlung schließe ich mich voll und ganz an.

Im Prinzip ist Audio in Computerspielen vergleichbar mit Audio in Film und Fernsehen, und Sie können den Effekt von Audio leicht selbst erfahren, indem Sie sich verschiedene herausragende Filmszenen einmal ohne Ton ansehen und danach mit Ton. Am besten geeignet für solch ein Experiment sind natürlich Action-Szenen sowie Szenen, in denen Spannung aufgebaut wird – aber auch eine romantische Szene wird ohne Ton in den allermeisten Fällen deutlich flacher wirken als mit Ton. Über den Hörsinn erreichen wir mit Spielen also auch das Fühlen – zumindest bezogen auf Gefühle und Emotionen. Den Tastsinn erreicht man nur, wenn der Spieler Musik und Effekte extrem laut aufdreht. ;-)

Ein wesentlicher Unterschied zwischen Audio in Film und Fernsehen gegenüber Audio in Computerspielen ist, dass Spiele interaktiv sind und daher die Ereignisse nicht oder nur in gewissem Rahmen vorhersehbar sind. Wenn beispielsweise in einer Spielszene durch musikalische Untermalung

26 Vgl. Schell, Jesse: The Art of Game Design, 2008. S. 351 f.

Spannung aufgebaut werden soll, hängt es meistens von Spieler-Aktionen ab, zu welchem Zeitpunkt diese Szene losgeht und wann sie endet.[27] Daher ist es bei Spielen wünschenswert, dass die Musik sich flexibel an die Spielsituation anpassen kann, was Komponisten und Musikproduzenten natürlich vor interessante Herausforderungen stellt.

Soundeffekte gibt es in Spielen überall dort, wo sie auch in Film und Fernsehen vorkommen: Ob es Schritte sind (die sich je nach Untergrund, Schuhwerk und Gangart anders anhören können), Explosionen und Schüsse, die Umgebungsgeräusche in einem Wald oder in einer Stadt – alles, was man am Bildschirm sieht und wovon der Spieler erwarten würde, dass es ein Geräusch macht, sollte auch ein Geräusch machen.

In Computerspielen gibt es aber noch weitere Ereignisse, die durch Soundeffekte wirkungsvoller dargestellt werden können: Ob das das Klicken auf einen Button ist, das Tippen eines Textes auf der Tastatur oder das Scrollen eines User-Interface-Elements – alles wirkt mit Ton dramatischer und wird den Spieler eher in die Spielwelt einbinden, wenn es auch entsprechend vertont ist.

Technisch kann man Audioeffekte unterteilen in andauernde Geräusche, wie z. B. ein Meeresrauschen, und solche, die einmal abgespielt werden, z. B. ein Schuss. Für andauernde Geräusche ist wichtig, dass diese ohne Knackser oder sonstige Unregelmäßigkeiten wiederholt werden können (Stichworte: *Loop* bzw. *Looping*).

Werkzeuge zur Musikproduktion für Spiele

Zur Produktion von Musik für Spiele können typische *DAWs* (*Digital Audio Workstations*) verwendet werden. Das sind Programme, die den vollständigen Produktionsprozess von Komposition bis Mastering unterstützen und somit auch als virtuelles Musikstudio bezeichnet werden könnten. Zwei Beispiele hierfür sind *Cubase* und *Logic Pro*. Beide unterstützen auch sogenannte *VST*-(*Virtual Studio Technology*-)Instrumente und -Effekte. Also vereinfacht gesagt Synthesizer, Sampler und Effektgeräte, die als Plug-ins auf dem Rechner laufen. Natürlich kann man auch Hardware-Synthesizer, -Sampler und -Effektgeräte verwenden. Dann benötigt man aber zusätzlich ein Mischpult, um die Audiosignale der verschiedenen Geräte wieder zusammenzumischen, und ein entsprechendes Audio-Interface, um die Musik wieder zurück in den Computer zu bekommen.

Unity bietet seit der Version 3.0 auch die Unterstützung von sogenannten *Tracker-Formaten*. Dabei handelt es sich um eine Kombination aus Audiosamples, die als Instrumente dienen, und aus der Sequenz, also der Abfolge, in der diese Samples in verschiedenen Tonhöhen abgespielt werden. Auf

27 Ein kleiner Trick, der in manchen Spielen funktioniert und dieses Problem elegant löst, besteht darin, die spannende Situation zeitlich zu beschränken: Wenn die Zeit abläuft, ist die Situation zu Ende. Dann kennen wir den Zeitpunkt im Voraus.

diese Art und Weise lassen sich ebenfalls Musikstücke in hoher Qualität produzieren, die aber nicht nur einen deutlich geringeren Speicherplatzverbrauch haben, was z. B. für Webspiele und auf mobilen Geräten eine Rolle spielt, sondern die prinzipiell auch zur Laufzeit, also während der Spieler spielt, geändert werden können. Somit können sich Musikstücke in Tracker-Formaten dynamisch der Spielsituation anpassen. Unity bietet dazu leider noch keine entsprechende API an. Musikstücke in Tracker-Formaten können Sie beispielsweise mit *MilkyTracker* (Mac OS X und Windows) oder *OpenMPT* (nur Windows) erstellen.

Seit Unity 5 gibt es in Unity einen mächtigen Audiomixer, der auch Effekte unterstützt. Den lernen Sie in Abschitt 8.4.6, *Audio-Engineering: Musik und Effekte aufeinander abstimmen* kennen.

Werkzeuge zum Erstellen von Audioeffekten

Eine naheliegende Möglichkeit, Audioeffekte zu erzeugen, besteht darin, die Geräusche mit einem Mikrofon aufzunehmen und über ein entsprechendes Audio-Interface in den Computer zu übertragen. Die so erzeugten Audiodateien können Sie dann mit einem beliebigen Audioeditor bearbeiten, z. B. mit *Audacity* (kostenfrei, Mac OS X und Windows) oder *WaveLab* (kommerziell, ebenfalls Mac OS X und Windows).

Alternativ können Sie auch die verschiedensten elektronischen Klangerzeuger (wie Synthesizer und VST-Instrumente) verwenden, um die Klänge ganz nach Ihren Vorstellungen zu designen. Zwei nennenswerte Hersteller von VST-Instrumenten und Effekten sind *Steinberg* und *Native Instruments*. Falls Sie Retro-Spiele mit dem typischen »8-Bit-Sound« entwickeln wollen, ist hierzu auch das bereits erwähnte *SFXR* verwendbar.

Falls Ihnen die Komposition von Musik oder das Sounddesign nicht liegt oder Sie nicht die entsprechenden Fähigkeiten mitbringen, können Sie natürlich auch bestehende Musik- oder Effektbibliotheken verwenden. Und diese Möglichkeit gibt es natürlich auch für 2D- und 3D-Assets und sogar – zumindest in einem gewissen Rahmen – für Scripts. Auf eine sehr einfache Möglichkeit, sich der Talente anderer zu bedienen, oder auch die eigenen Talente anderen anzubieten gehe ich im nächsten Abschnitt ein.

5.4 Teamwork für Einzelspieler: Unity Asset Store

Es liegt nahe, dass ein einzelner Entwickler kaum in allen Disziplinen der Spielentwicklung die entsprechenden Talente mitbringt. Und selbst in einem kleinen Team kann möglicherweise nicht jeder Bereich abgedeckt werden. Diese Problematik hat Unity Technologies erkannt und daher den *Unity Asset Store* eingeführt, den ich bereits mehrfach erwähnt habe. Die Idee ist dabei einerseits, jedem Unity-Spielentwickler eine Vielzahl von Assets

aus allen Bereichen zur Verfügung zu stellen, sodass man nicht jedes einzelne Asset selbst erstellen muss, das man zur Umsetzung seiner Spielidee braucht. Andererseits bietet der Asset Store natürlich denjenigen, die ihre Assets anderen anbieten möchten, einen lukrativen Marktplatz.

Auch bei über 20.000 Paketen im Januar 2015 ist es zwar nicht unbedingt so, dass man für jeden Bedarf etwas findet, aber zumindest findet man für praktisch alle Bereiche zumindest eine Arbeitsgrundlage: Ob das nun Texturen sind, Modelle, um einen Raum zu füllen, animierte Charaktere, verschiedenste Musikstücke zur stimmungsvollen Untermalung oder Soundeffekte. Außerdem gibt es eine Vielzahl von Erweiterungen des Unity Editors, die nicht nur die tägliche Arbeit mit Unity erleichtern, sondern teilweise Dinge ermöglichen, die man sonst erst mit hohem Aufwand selbst entwickeln müsste.

Natürlich ist der Asset Store nicht die einzige Quelle für Game Assets. Aber da die Pakete erst nach einer Prüfung durch Mitarbeiter von Unity Technologies veröffentlicht werden, entsprechen die Assets einem gewissen Standard. Und da es der Unity Asset Store ist, sind sie speziell für die Unity Engine optimiert und funktionieren dort auch reibungslos.

Gleiches gilt natürlich auch für die Assets, die wir im Folgenden zur Entwicklung unseres Buch-Spiels verwenden. Es würde den Rahmen sprengen, hier die konkrete Bedienung der vielfältigen Werkzeuge zu beschreiben oder gar die Künste lehren zu wollen, denen diese Werkzeuge dienen. Auf dieser Reise kann ich Sie leider nicht begleiten.

Daher biete ich im Folgenden die jeweils benötigten Assets zum Download an und beschränke mich auf deren Anwendung in Unity. Dies soll den geneigten und entsprechend talentierten und mit den richtigen Werkzeugen ausgestatteten Leser natürlich nicht davon abhalten, seine ganz eigene Version des Spiels mit komplett eigenen Assets umzusetzen.

5.5 Klassisches Teamwork oder Backup: Versionsverwaltung

Sind Sie in der glücklichen Situation, in einem kleinen Team arbeiten zu können, in dem die verschiedenen zur Spielentwicklung notwendigen Talente vereint sind? Dann stellt sich natürlich die Frage, wie mehrere Leute möglichst reibungslos und am besten gleichzeitig an einem Projekt arbeiten können. Die Standardantwort darauf kommt aus der Softwareentwicklung und heißt *Versionsverwaltung*. Und das ist gleichzeitig durchaus auch für Einzelentwickler eine sehr nützliche Sache, wenn auch aus anderen Gründen:

Die von einer Versionsverwaltung gehaltene Versionshistorie ermöglicht es nämlich, jede wesentliche Veränderung auch im Nachhinein zu überprüfen und rückgängig zu machen, wenn das mal notwendig wird – beispiels-

weise, wenn man feststellt, dass eine Funktionalität, die früher einwandfrei gearbeitet hat, nach verschiedenen Änderungen plötzlich nicht mehr das tut, was man von ihr erwartet.

Vereinfacht gesagt bestehen Systeme zur Versionsverwaltung aus einem zentralen Repository, in dem der aktuelle Projektstand sowie die gesamte Versionshistorie abgelegt sind, so wie einer Software, die den Zugriff auf dieses Repository ermöglicht. Dabei arbeitet man mit einer lokalen *Arbeitskopie* des Projekts, die sich vom aktuellen Stand des Repository unterscheidet, aber mit verschiedenen Aktionen synchronisiert werden kann: Jederzeit können Änderungen von anderen Entwicklern im Repository auf die aktuelle Arbeitskopie übertragen werden. Dieser Vorgang wird meistens mit *Checkout, Aktualisieren* oder *Get Latest Version* bezeichnet. Und Änderungen in der lokalen Arbeitskopie können – wenn sie fertig sind – jederzeit in das Repository eingespielt werden, was meist mit *Checkin* bezeichnet wird.

Beispiele für konkrete Lösungen zur Versionsverwaltung mit direkter Integration in Unity wären der *Unity Asset Server, Perforce* und *Plastic SCM*. Weitere Lösungen wären z. B. *Microsoft Team Foundation Server, Subversion, Bazaar* oder *Git*.[28]

Eine wichtige Besonderheit für die Versionsverwaltung im Zusammenhang mit Unity ist, dass wir es hier häufig mit vielen, teilweise sehr großen Binärdateien zu tun haben, was im Softwareentwicklungsalltag, für den die meisten Versionsverwaltungen ursprünglich entwickelt wurden, eher untypisch ist.[29] Mit den direkt in Unity integrierten oder spezifisch für die Spielentwicklung entworfenen Lösungen sollte es damit aber keine Schwierigkeiten geben.

Prinzipiell würde ich die Verwendung einer geeigneten Versionsverwaltung auf jeden Fall empfehlen, und falls Sie im Team arbeiten und ihr Spiel dennoch ohne Versionsverwaltung entwickeln wollen, müssen Sie sich sehr eng koordinieren und die Arbeitsabläufe entsprechend koordinieren. Leider würden alle dazugehörigen Details aber hier den Rahmen sprengen. Daher belasse ich es bei dieser Empfehlung. Denn: Ein unvollständiger Prototyp wartet darauf, zu einem vollwertigen Spiel zu werden – gehen wir es an!

---

28 Link auf *unity-buch.de:* Details zum Einrichten der verschiedenen Versionsverwaltungen würden hier den Rahmen sprengen, sind aber von *http://unity-buch.de* aus verlinkt: *Version Control Integration* und *Using External Version Control Systems with Unity*.

29 Link auf *unity-buch.de:* Zu diesem Thema gibt es auch Fragen mit entsprechenden Antworten im Internet, z. B. *Is there a distributed VCS that can manage large files?*

7 Projekt-Polishing – Iteration 1

Grundsätzlich sind Prototypen nicht unbedingt dazu gedacht, weiterverwendet zu werden. Bei manchen Prototypen wäre das auch gar nicht umsetzbar: Tracy Fullerton empfiehlt in einem absolut lesenswerten Buch sogar die konsequente Verwendung von physischen Prototypen in frühen Phasen des Game-Designs.[1] Da stellt sich die Frage der weiteren Verwendung erst gar nicht. Das wesentliche Ziel bei der Arbeit mit einem Prototyp ist ja, mit möglichst geringem Aufwand und Risiko bestimmte Ideen ausprobieren zu können, diese Ideen ggf. zu überarbeiten oder ganz fallen zu lassen und dann mit einem anderen Prototyp neue Ideen bzw. mit einem überarbeiteten Prototyp die modifizierten Ideen auszuprobieren.

Ein Spielprojekt bzw. ein Software-Projekt sauber aufzusetzen kostet aber normalerweise mehr Zeit, daher wird man bei Prototypen oft eher den »Hacker-Stil« verfolgen: Hauptsache, es funktioniert, egal wie.

Wenn aber– wie in unserem Fall – ein Prototyp gezeigt hat, dass unsere Ideen grundsätzlich funktionieren, und wir darauf aufbauend ein richtiges Spiel entwickeln wollen, dann spricht nichts dagegen, den Prototyp iterativ zu einem sauber aufgesetzten Spiel zu erweitern.

Solche *Polishing-Phasen* lohnen sich auch immer dann, wenn wir im Eifer des Gefechts aus irgendwelchen Gründen »mal eben schnell« das Spiel um Features erweitert haben, ohne dabei Zeit für eine saubere Integration dieser Features in das bestehende Projekt aufgewendet zu haben. Ob der Grund also ein kurzfristiger Release-Termin für eine Demo oder Abgabe bei einem Spielentwicklungswettbewerb ist oder eine prototypartige Erweiterung eines bestehenden Spiels oder ein Prototyp an sich, ist da eher nebensächlich. Das Prinzip, um das es hier geht, ist, dass es gute Gründe gibt, im Verlauf der Entwicklung eines Spiels immer mal wieder einen Schritt zurück zu tun und sich zu fragen:

An welchen Stellen fühlt sich mein Projekt gerade »unsauber« an, und wie kann ich vorgehen, um diese Stellen aufzuräumen?

1 Vgl. Tracy Fullerton, Game Design Workshop – A Playcentric Approach to Creating Innovative Games, S. 175 ff.

Und dann sollten Sie auch aufräumen. Diesen Vorgang nennen wir *Polishing*[2], und er bezieht sich keineswegs nur auf die Implementierungsebene (also unseren Programmcode), sondern meistens auf Aspekte des Spiels, die der Spieler sieht (z. B. GUI, Handling der Steuerung, Animationen von Charakteren usw.).

7.1 Die Projektstruktur optimieren

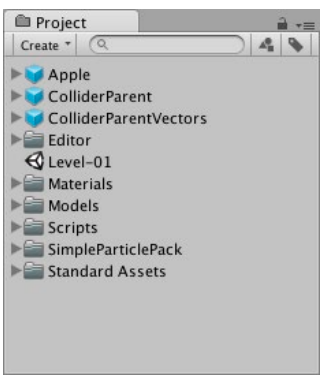

Abb. 7.1
Unsere organisch gewachsene Projektstruktur

Sehen wir uns unser Projekt also nun aus dieser Perspektive an: Da wir uns noch im Prototyp-Stadium befinden, lassen wir Usability-Aspekte sowie Aspekte der Optik außen vor und betrachten nur die Ebenen des Projekts innerhalb von Unity sowie die Implementierungsebene. Werfen wir also zunächst einen Blick auf die Projektstruktur (siehe Abb. 7.1).

Was fällt Ihnen hier auf? Nehmen Sie sich ruhig einen Moment Zeit, und machen Sie sich ein paar Notizen, was Sie hier verwirrend finden, wo potenziell Probleme auftreten könnten, wenn das Projekt weiterwächst, und wie Sie vorgehen würden, um die Struktur hier zu verbessern. Lesen Sie am besten erst danach weiter …

Mir fallen folgende Punkte auf:

- Unsere Prefabs liegen auf der Hauptebene. Zufällig werden sie aufgrund der alphabetischen Sortierung gerade zusammen aufgelistet. Später werden sie aber wahrscheinlich wirr über das Projekt verstreut. Außerdem wird das Projekt bei mehreren Prefabs wahrscheinlich schnell unübersichtlich, wenn wir so weitermachen.
- Die beiden Prefabs `ColliderParent` und `ColliderParentVectors` haben Namen, bei denen ich schon jetzt nachdenken muss, wozu die eigentlich gut sind.
- Es gibt zwei Verzeichnisse mit Assets, die von externer Quelle kommen (`Standard Assets` und `SimpleParticlePack`). Diese sind mit unseren eigenen Assets vermischt. Später wissen wir vielleicht nicht mehr, was wir selbst erstellt haben und was wir aus externen Quellen bezogen haben, und insbesondere werden wir uns nicht mehr erinnern, wie die externen Assets lizenziert sind. Daraus können uns im schlimmsten Falle Probleme mit anderer Leute Urheberrecht entstehen. Der Ordner `Standard Assets` ist hier allerdings speziell, da er von Unity vorgegeben ist.

Grundsätzlich gibt es unterschiedliche Herangehensweisen, wie man ein Projekt strukturieren kann, und die jeweils günstigste Struktur hängt oft auch vom jeweiligen Projekt ab: Während es manchmal günstig ist, alle Assets für einen Level zusammen zu organisieren, ist es in anderen Fällen

2 »Polishing« ist ein Begriff, den man zwar übersetzen könnte (mit »Polieren«), aber es ist unwahrscheinlich, dass das auch irgendjemand auf Deutsch so sagt.

vorteilhafter, die Assets grob nach Typen oder auch Arbeitsschritten zu organisieren (z. B. »Assets, die zum Level-Design benötigt werden an einen Platz; Assets, die mit Charakteren zu tun haben, an einen anderen«). Oder es gibt allgemeine und Level-spezifische Assets: Die allgemeinen werden dann in einem entsprechenden Ordner untergebracht, die spezifischen jeweils bei dem Level. Weiterhin haben wir gerade gesehen, dass manchmal auch die Herkunft von Assets eine sinnvolle Struktur ergibt (z. B. »Assets aus dem Asset Store«, »Assets von 3D-Freelancer Hans Mustermann«).

Angenehmerweise macht Unity es sehr einfach, die Projektstruktur jederzeit zu verändern, wenn man feststellt, dass die aktuelle Struktur nicht mehr passt. An sich könnten Sie das Projekt jetzt nach Ihrem Geschmack strukturieren. Außerdem haben Sie über Labels eine zusätzliche Möglichkeit, ihre Assets zu verschiedenen Gruppen zusammenzufassen.[3] Das hätte nur den erheblichen Nachteil, dass es deutlich schwieriger wäre, den späteren Abschnitten dieses Buches zu folgen. Daher empfehle, dass wir folgende Veränderungen gemeinsam durchführen:

1. Erzeugen Sie ein neues Verzeichnis `Prefabs`, und ziehen Sie die drei Prefabs `Apple`, `ColliderParent` und `ColliderParentVectors` in dieses Verzeichnis.
2. Benennen Sie `ColliderParent` in `WallSegmentScaled` um und `ColliderParentVectors` in `WallSegmentMeshModified`.
3. Erzeugen Sie ein neues Verzeichnis `Xternal` und in diesem Verzeichnis ein Unterverzeichnis `Asset Store`. Ziehen Sie nun `SimpleParticlePack` in das Verzeichnis `Asset Store`. Das Verzeichnis `Standard Assets` lassen wir, wo es ist, da es sich um einen speziellen Ordner handelt. Der Name `Xternal` hat den Vorteil, dass X meistens ans Ende sortiert wird und wir damit eine gute Trennung der externen Assets von unserem restlichen Projekt haben.

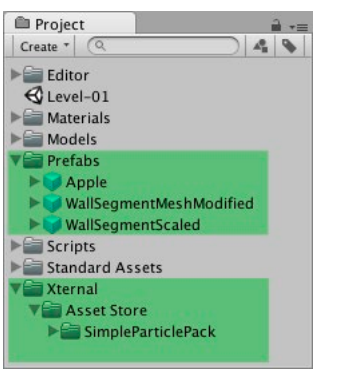

Abb. 7.2
Eine neue, aufgeräumte Projektstruktur

Auf der Website zum Buch sind die beiden Einträge *Special Folder Names* und *Special Folders and Script Compilation Order* im Unity Manual verlinkt – dort können Sie alles über diese speziellen Ordner erfahren, was es zu wissen gibt.

Link auf unity-buch.de

Ihre Projektstruktur sollte nun aussehen wie in Abb. 7.2.

3 Labels hatten Sie am Anfang unserer Reise kennengelernt, in Abschnitt *2.2.8, In Project, Hierarchy und Scene View suchen,* ab Seite 39. Wie Sie Labels hinzufügen, sehen Sie in Abb. 2.35 auf Seite 42.

7.2 Die Szenenhierarchie übersichtlicher gestalten

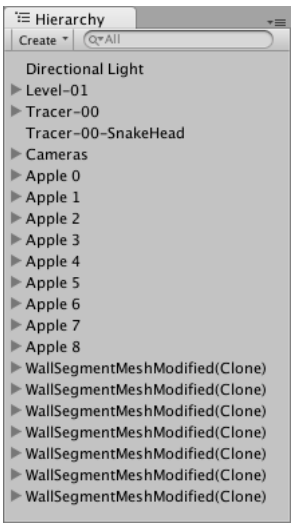

Abb. 7.3
Chaos in der Szenenhierarchie

Betrachten wir nun die Szenenhierarchie. Und zwar am besten, nachdem wir das Spiel gestartet und einige Drehungen durchgeführt haben. Je nachdem, wie viele Äpfel Sie in Ihrem Level platziert haben und wie viele Drehungen Sie in der aktuellen Session durchgeführt haben, sieht das dann in etwa so aus wie in Abb. 7.3.

Das ist das gleiche Spiel wie vorher: Was fällt Ihnen auf? Wie könnte man die Szenenhierarchie übersichtlicher gestalten? Vielleicht müssen wir zur Umsetzung der einen oder anderen Optimierungsidee auch den Code ändern – aber das sollte uns nicht davon abhalten, die Szene zu jedem Zeitpunkt möglichst übersichtlich zu halten! Vergessen Sie nicht: In Unity verbringen wir viel Zeit damit, das Spiel zu spielen, zu inspizieren und Veränderungen darin auszuprobieren. Manchmal »greifen« wir Objekte am einfachsten über die *Scene View* – aber oft ist es am schnellsten, wenn wir uns die Objekte in der *Hierarchy View* heraussuchen und dann über Doppelklick oder Selektieren und Druck auf F das Objekt in der Szene anfokussieren. Wenn unsere Szene aber ein chaotisches Durcheinander ist, funktioniert das natürlich nicht.

Haben Sie sich ein paar Gedanken gemacht? Tragen wir unsere Ideen nun zusammen. Hier ist meine Liste:

- Die `Apple`-Objekte gehören zum Level, sollten ein gemeinsames Elternobjekt haben und Namen, die ungefähr die Position im Level erkennen lassen.
- `Directional Light` gehört ebenfalls zum Level. Hier wäre ein Elternobjekt zwar nicht unbedingt notwendig, aber es schadet auch nicht – insbesondere, falls wir den Level später besser ausleuchten wollen und dazu mehrere Lichtquellen einsetzen.
- Die Kameras (`Cameras` und alles, was darunter liegt) folgt automatisch unserem Tracer, also gehört sie letztlich zum Spieler.
- `Tracer-00` und `Tracer-00-SnakeHead` gehören unbedingt zusammen und gehören ebenfalls zum Spieler. Eigentlich hätten wir `Tracer-00-SnakeHead` gerne unter `Tracer-00`. Das geht aber technisch nicht, weil sie sich getrennt bewegen müssen. Wir können aber `Tracer-00-SnakeHead` dynamisch erzeugen, sobald das Spiel startet. Somit stört das Objekt nicht in der Szene, solange wir nicht spielen; und vor allem verhindern wir auf diese Weise, dass jemand z. B. versehentlich die Position ändert und das Spiel deswegen nicht mehr funktioniert.
- Die Liste der Wandsegmente gehört zum Spieler und sollte fortlaufend durchnummeriert sein, damit wir jederzeit leicht erkennen könnten, welches das Segment direkt hinter dem Spieler ist und welches das Segment ist, das der Spieler zuerst erzeugt hatte. Außerdem sollten sie unter einem Elternobjekt `Walls` zusammengefasst sein.

Vergessen Sie jetzt auf keinen Fall, das Spiel zu stoppen – sonst verlieren Sie später alle Änderungen! Wahrscheinlich gab es in Ihrer Szene seit dem letzten Speichern keine Veränderungen, aber sicher ist sicher: ***Vor solchen Veränderungen ist Speichern angesagt!*** Falls wir versehentlich die Szene kaputt machen, schließen wir dann einfach die Szene, ohne zu speichern, und beim erneuten Öffnen ist alles wieder gut.

Pro-Tipp

Sie können durch Doppelklick auf eine bereits geöffnete Szene in der Project View diese auch einfach erneut laden.

Wenn Sie ganz sichergehen wollen, können Sie sogar eine Kopie der alten Szene anlegen. Wichtig ist dabei, dass Sie auch dabei nicht vergessen, vorher abzuspeichern, weil Sie sonst möglicherweise eine Kopie eines veralteten Stands der Szene hätten. Duplikate beliebiger Assets im Projekt – also auch von Szenen – erstellen Sie mit der Tastenkombination ⌘+D bzw. Strg+D. Am sichersten ist natürlich die Verwendung einer Versionsverwaltung.[4]

Aber nun zu unserer Aufräumaktion. Zunächst die einfachen Schritte:

1. Legen Sie zunächst drei leere GameObjects an, nennen Sie sie `Apples`, `Geometry` und `Lights`, und setzen Sie bei allen drei Objekten `Position = (0, 0, 0)`. Dies geht am elegantesten, indem Sie alle drei Objekte gleichzeitig selektieren und dann für alle auf einmal über das Kontextmenü-Rädchen `Reset` durchführen.
2. Ziehen Sie als Nächstes alle Objekte unter `Level-01` in das neue Objekt `Geometry` (also `BottomPlate`, `WallEast` usw.). Ziehen Sie dann `Geometry` unter `Level-01`.
3. Alle Äpfel gehören unter `Apples`, und `Directional Light` gehört unter `Lights`. Jetzt können Sie `Apples` und `Lights` ebenfalls unter `Level-01` ziehen. Als Reihenfolge unter Level-01 würde ich vorschlagen: `Geometry`, `Apples` und dann `Lights`. Hintergrund dieser Wahl ist, dass die Positionierungen der Äpfel von der Geometrie abhängen und die Lichtquellen so gewählt werden können, dass Sie Levelgeometrie und Äpfel optimal ausleuchten, also von den beiden vorigen Elementen abhängig sind. Aber so genau muss man es nicht nehmen.

1. Bei der Benennung der Äpfel ist Kreativität gefragt: Falls Ihre Äpfel unterschiedliche Punktezahlen vergeben, ist die Punktezahl sicher ein sinnvoller Bestandteil des Namens. Wie bei den Seitenwänden können Sie auch die Himmelsrichtungen als Postfixe verwenden (North = Z+[5], South = Z-, East = X+, West = X-), ggf. auch nur die ersten Buchstaben

4 Siehe Kapitel *5.5, Klassisches Teamwork oder Backup: Versionsverwaltung*.

5 Z+ bedeutet: »auf der Z-Achse in positiver Richtung«. Das ist natürlich beliebig, aber es ist durchaus hilfreich, um eine einheitliche Vorstellung davon zu haben, wie in unserer virtuellen 3D-Welt die Achsen und die uns bekannten Himmelsrichtungen zusammenhängen.

Abb. 7.4
Die Szenenhierarchie aufräumen, Teil 1

und diese auch in Kombination (z. B. NEE für Nord-Ost-Ost, also ein positiver Wert auf der Z-Achse und im Vergleich zu anderen Äpfeln ein höherer positiver Wert auf der X-Achse). Center bzw. C könnte hier auch sinnvoll sein. Ebenso könnten Äpfel, die in einer Ecke oder ganz nah an der Wand liegen, z. B. »Corner« oder »Wall« im Namen tragen. Für diese Operation ist auf jeden Fall die Top-Down-Ansicht in der *Scene View* hilfreich, die Sie leicht durch Klick auf die Y-Achse im Scene Gizmo erhalten.

2. Erzeugen Sie weiterhin auf der höchsten Ebene der Hierarchie (also neben `Level-01`, `Cameras` usw.) ein neues, leeres GameObject `Player` sowie ein neues Objekt `Walls`, beide mit `Position = (0, 0, 0)`. Ziehen Sie `Cameras`, `Tracer-00` und `Tracer-00-SnakeHead` sowie unser neues Objekt `Walls` unter `Player`.

Die neue Szenenhierarchie sollte in etwa so aussehen wie in Abb. 7.4, wobei Ihre Äpfel wahrscheinlich anders positioniert sind und daher auch andere Namen tragen (und ggf. haben Sie mehr oder weniger Äpfel in Ihrer Szene).

SnakeHead dynamisch erzeugen

Die wesentlichen Methoden zum dynamischen Erzeugen von SnakeHead haben Sie bereits gelernt: `Tracer-00-SnakeHead` muss ein Prefab sein, das ähnlich wie das Prefab für die Wandsegmente im `WallController` zur Verfügung steht und dort über `Instantiate()` erzeugt wird. Natürlich muss unser SnakeHead im `TracerControllerV2` erzeugt werden, und zwar beim Initialisieren, also in der Methode `Awake()`. Die Position und Rotation des SnakeHead muss dem Transform `pointFront` entsprechen, auf das `TracerControllerV2` bereits eine Referenz hat.

Wissen Sie noch, wie Sie Prefabs aus bestehenden Objekten aus der Hierarchie erzeugen? Ziehen Sie einfach `Tracer-00-SnakeHead` aus der *Hierarchy View* in unser neues Verzeichnis `Prefabs` im *Project Browser*. So liegt es gleich an der richtigen Stelle. Aus der Szene können Sie das Objekt dann direkt löschen. Ändern Sie dann bitte noch den Namen des Prefabs in `SnakeHead`.

Wir wollen außerdem, dass unser neu instanziiertes Objekt immer genau vor unserem `Tracer-00` liegt, also unter dem Objekt `Player` an erster Position und mit dem Namen des Objektes (also `Tracer-00`) als Präfix, und dann feststehend als Postfix `-SnakeHead`. Das ist neu – Listing 7.1 zeigt alle notwendigen Erweiterungen in `TracerControllerV2`.

Listing 7.1
Erweiterungen in TracerControllerV2

```
public Transform pointBack;
public Transform pointFront;
public Transform rotateBody;

public Rigidbody snakeHeadPrefab;
```

```
private Rigidbody snakeHead;

private WallController wallController = null;
private Queue<Turn> turns = new Queue<Turn>();

void Awake() {
   wallController = GetComponent<WallController>();

   snakeHead = (Rigidbody) Instantiate(snakeHeadPrefab,
                   pointFront.position, pointFront.rotation);
   snakeHead.transform.parent = this.transform.parent;
   snakeHead.name = string.Format("{0}-SnakeHead", this.name);
   snakeHead.transform.SetAsFirstSibling();
}
```

Wenn Sie das Spiel jetzt starten und das Fahrzeug nicht mehr steuern können, sollten Sie einen Blick in die Konsole werfen!

Dort sehen Sie zuerst eine `UnassignedReferenceException` mit einer genauen Erklärung, was zu tun ist, und danach eine Reihe »Folgefehler«: `NullReferenceExceptions`, weil die Variable `snakeHead` nicht initialisiert wurde.

Die Ursache wäre also, dass Sie unser neues Prefab `SnakeHead` nicht auf den Slot der neuen Variable `snakeHeadPrefab` am Objekt `Tracer-00` in der Komponente `TracerControllerV2` gezogen haben. Das kann passieren, zumal ich solche naheliegenden Schritte nicht mehr immer dazuschreibe. Wichtig ist mir aber natürlich, dass Sie sich dann zu helfen wissen. Schließlich sollen Sie am Ende des Buches selbst Ihr Spiel entwickeln können![6]

Die Szenenhierarchie sieht nun deutlich übersichtlicher aus. Vor allem kann niemand mehr `Tracer-00-SnakeHead` an eine falsche Position verschieben. Das Objekt gibt es nämlich in unserer Szene nicht mehr, wie Abb. 7.5 zeigt.

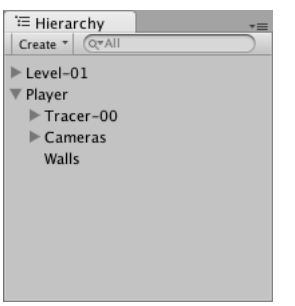

Abb. 7.5
Die Szenenhierarchie aufräumen, Teil 2

Die Wandsegmente benennen und einordnen

Übung macht den Meister

Soeben haben wir gelernt, wie wir neu instanziierte Objekte unter ein bestehendes Objekt hängen können sowie das Objekt zu benennen. Insofern sollte es Ihnen jetzt möglich sein, die neu erzeugten Wandsegmente unter das Objekt Walls zu hängen und sinnvoll zu benennen. Versuchen Sie das ruhig als Übung selbst! Verantwortlich für das Erzeugen der Wände ist ja unser `WallController`, d. h., die Erweiterung ist in jedem Fall dort vorzunehmen.

6 Falls Sie tatsächlich mal an einer Stelle nicht mehr weiterkommen sollten: Sehen Sie bitte zuerst auf der Website zum Buch im Bereich Errata zu dem entsprechenden Kapitel nach. Vielleicht hat sich tatsächlich ein Fehler eingeschlichen. Falls Sie dort nicht die Lösung finden: Auf der Website gibt es genau dafür auch das Fragen-Forum.

Die Lösung finden Sie im fertigen Projekt `Traces_Prototype_120.zip`, das Sie wie üblich von der Website zum Buch herunterladen können.

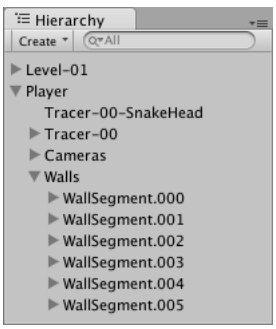

Abb. 7.6
Ein aufgeräumter Spielplatz: unsere Szenenhierarchie

Jetzt ist die Szenenhierarchie auch dann aufgeräumt, wenn wir das Spiel spielen, denn alles ist am rechten Fleck, wie Abb. 7.6 illustriert.

7.3 Den Code für Erweiterungen vorbereiten

Wir haben jetzt eine verbesserte Projektstruktur sowie eine übersichtlichere Szenenhierarchie. Im nächsten Schritt soll der Programmcode verbessert werden. Da wir unsere Scripts von Anfang an sehr modular aufgebaut und uns im Zweifelsfall für die »Softwareentwickler-Variante« entschieden haben statt für die »Hacker-Variante«, ist hier nicht viel zu tun.

Wahrscheinlich fällt Ihnen aber zumindest ein Punkt auf, an dem es Handlungsbedarf gibt. Ein Implementierungsdetail, das nämlich offensichtlich unschön ist, sind die beiden Klassen `TracerController` und `TracerControllerV2`. Diese Scripts sollen ja jederzeit in unserem `Tracer-00` ausgetauscht werden können, die Implementierungen unterscheiden sich aber recht deutlich. Damit der Austausch der Scripts dennoch problemlos möglich ist, sollten zumindest die öffentlichen Membervariablen für den Editor jederzeit identisch sein. Dafür bietet sich klassisch aus der objektorientierten Programmierung die Einführung einer gemeinsamen Elternklasse an, um die öffentlichen Variablen einfach erben zu können.

Die zweite Änderung ist wahrscheinlich nicht so offensichtlich. Daher hierzu ein Szenario: Stellen Sie sich vor, Sie haben das Spiel praktisch fertig entwickelt und mit verschiedensten Tastaturkommandos (links, rechts, springen, schießen, Zeit verlangsamen usw.) versehen und stellen jetzt fest, dass die von Ihnen gewählten Tasten zwar auf einer deutschen Tastatur gut funktionieren, nicht aber auf einer englischen. Oder Sie möchten das Spiel als Nächstes auf mobile Geräte portieren. Mobile Geräte haben normalerweise gar keine Tastatur, d. h., Sie müssen die Steuerung von Tastaturkommandos auf das Tippen auf Flächen am Display umstellen. Im Moment haben wir die Tastaturkommandos im `TracerController` bzw. `TracerControllerV2` und die Kommandos für PowerUps hätten wir wahrscheinlich in einem `PowerUpHandler`.

Da wäre es doch angenehmer, wenn wir von vornherein die Steuerung in eine eigene Komponente `InputHandler` auslagern, in der die eigentlichen Befehle des Spiels (Drehung links, Drehung rechts) von den konkreten Steuerbefehlen (Taste links, Taste rechts) abstrahiert sind, sodass man später z. B. sehr einfach »Steuerfläche links«, »Steuerfläche rechts« implementieren kann. Das wird also unsere zweite Änderung auf der Ebene des Programmcodes.

Aber ein Schritt nach dem anderen – führen wir zunächst eine saubere Lösung für den bzw. die TracerController ein.

7.3.1 TracerController-Varianten über Vererbung umsetzen

Zunächst ist es etwas unglücklich, dass wir einen `TracerController` haben und einen `TracerControllerV2`. So etwas passiert während einer heißen Entwicklungsphase leicht und ist in Unity auch relativ einfach aufzulösen, sofern man nicht auf die Idee kommt, die Klasse in der Entwicklungsumgebung oder im Dateisystem umzubenennen. Das Problem dabei wäre, dass Unity die umbenannte Klasse als neues Script betrachten würde. Damit würden alle Referenzen verloren gehen. Da bedeutet, an jedem GameObject, an dem wir das umbenannte Script verwenden, würde im Script-Slot »`Missing (Mono Script)`« stehen. In der Konsole würden wir beim Start des Spiels die Fehlermeldung »*The referenced script on this Behaviour is missing!*« erhalten, und das Spiel würde wahrscheinlich nicht mehr funktionieren oder schlimmer: Bestimmte Features, die wir vielleicht bei den ersten Tests gar nicht bemerken, könnten nicht mehr funktionieren.

Pro-Tipp

Führen Sie Änderungen an Dateien im Projekt am besten niemals über den Finder bzw. den Windows Explorer durch, sondern immer über Unity. Dies gilt auch für Änderungen an Dateinamen von Scripts, die Sie nicht über die externe Entwicklungsumgebung (Visual Studio oder MonoDevelop) durchführen sollten, sondern ebenfalls direkt in Unity.

Also benennen wir `TracerController` direkt in Unity um, und zwar im *Project Browser*. Wir nennen ihn jetzt `TracerControllerV1`. Im nächsten Schritt ändern wir dann den Klassennamen in unserer Entwicklungsumgebung (Visual Studio oder Mono Develop). Dazu gibt es üblicherweise entsprechende Refactoring-Befehle, die dafür sorgen, dass alle Referenzen innerhalb des Programmcodes ebenfalls korrekt aktualisiert werden. Achten Sie aber bei Verwendung solcher Refactoring-Befehle aus oben genannten Gründen darauf, dass Sie nicht die Verzeichnisstruktur oder Dateinamen ändern!

Der nächste Schritt ist, eine neue Klasse `TracerControllerBase` im Unity Projekt einzuführen und die beiden Klassen `TracerControllerV1` und `TracerControllerV2` davon erben zu lassen. Listing 7.2 zeigt die neue Klassendeklaration von `TracerControllerV1`; `TracerControllerV2` funktioniert analog.

Listing 7.2
Von TracerControllerBase erben

```
public class TracerControllerV1 : TracerControllerBase {
```

Die Klassen erben jetzt also nicht mehr direkt von `MonoBehaviour`, sondern von `TracerControllerBase`. Dann können wir die öffentlichen Membervariablen `baseVelocity`, `pointBack`, `pointFront`, `rotateBody` und `snakeHeadPrefab` aus `TracerControllerV2` in `TracerControllerBase` verschieben und `baseVelocity`, `pointBack`, `pointFront` sowie `rotateBody` aus `TracerControllerV1` löschen. Weiterhin können wir auch `myRigidbody` und `wallController` in die Vaterklasse ziehen (also per Ausschneiden und Ein-

fügen). Dabei dürfen wir aber nicht vergessen, dass diese Membervariablen jetzt nicht mehr `private` sein dürfen, sondern `protected` sein sollten.

Außerdem habe ich der Klasse `TracerControllerBase` die beiden Methoden `InitComponents()` und `StartMoving()` spendiert, die jeweils von `Awake()` bzw. `Start()` in den Kindklassen aufgerufen werden und den Code enthalten, der in beiden `Awake()`- bzw. `Start()`-Methoden identisch war.

7.3.2 Den InputHandler zur Behandlung von Tastaturabfragen erstellen

Unser `InputHandler` soll im Wesentlichen dafür sorgen, dass wir im `TracerController` auf Spielkommandos prüfen können statt wie bisher auf Tastaturkommandos. Bei der Gelegenheit können wir auch gleich die Tastaturbefehle konfigurierbar machen. Das hat nicht nur den Vorteil, dass wir die Steuerung leicht an verschiedene Tastaturlayouts oder Spielerpräferenzen anpassen können: Mit einer anpassbaren Steuerung können wir auch sehr leicht einen einfachen Mehrspieler-Modus umsetzen, bei dem beide Spieler eine Tastatur einsetzen, aber Spieler 1 einfach andere Tasten zur Steuerung benutzt als Spieler 2. Das machen wir auch, und zwar in Kapitel *14, Ein minimales Multiplayer-Spiel*.

Die komplette Klasse `InputHandler` finden Sie in Listing 7.3.

Listing 7.3
Die neue Klasse InputHandler.cs

```
using UnityEngine;
using System.Collections;

public enum TurnCommand { None, Left, Right }

public class InputHandler : MonoBehaviour {

   public KeyCode keyCodeTurnLeft = KeyCode.LeftArrow;
   public KeyCode keyCodeTurnRight = KeyCode.RightArrow;

   private TurnCommand currentTurnCommand = TurnCommand.None;

   public bool HasTurnCommand(TurnCommand cmd) {
       return currentTurnCommand == cmd;
   }

   public void Update() {
       currentTurnCommand = TurnCommand.None;

       TestKey(keyCodeTurnLeft, TurnCommand.Left);
       TestKey(keyCodeTurnRight, TurnCommand.Right);
   }

   private void TestKey(KeyCode keyCode, TurnCommand command) {
       if (Input.GetKeyDown(keyCode)) {
           currentTurnCommand = command;
       }
   }

}
```

Unsere Spielkommandos setzen wir über den selbst definierten *Enumerationstyp*[7] `TurnCommand` um. Der Einsatz von *Enumerationstypen,* also selbst definierten Listen von Konstanten, bietet sich für diesen Einsatzzweck sehr an, da sie gut lesbar und leicht erweiterbar sind. Um den Zugriff auf `TurnCommand` von anderen Klassen aus möglichst einfach zu halten, haben wir den *Enumerationstyp* außerhalb der Klasse InputHandler deklariert; andernfalls müssten wir z. B. im TracerController `InputHandler.TurnCommand.Left` schreiben.

In der `Update()`-Methode setzen wir zuerst `currentTurnCommand` auf `TurnCommand.None`, d. h., jeder Spielbefehl steht jeweils nur im aktuellen Frame zur Verfügung, und zwar nach dem Aufruf der `Update()`-Methode durch die Engine. Er kann dann über unser öffentliches Property *CurrentTurnCommand* von überall aus abgefragt werden, wenn wir eine Referenz auf `InputHandler` haben.

Die brauchen wir also in jedem Fall noch im `TracerController`. Praktischerweise haben wir ja jetzt eine Basisklasse, d. h., wir müssen lediglich `TracerControllerBase` erweitern, wie in Listing 7.4 gezeigt.

```
public class TracerControllerBase : MonoBehaviour {

  public InputHandler inputHandler;
  public float baseVelocity = 5F;
  public Transform pointBack;
```

Listing 7.4
TracerControllerBase um inputHandler erweitern

In der Szenenhierarchie brauchen wir jetzt noch ein neues GameObject auf `Position = (0, 0, 0)`, das den Namen `InputHandler` bekommt. Dieses setzen wir in der Hierarchie unter `Player`, wie Sie in Abb. 7.7 sehen.

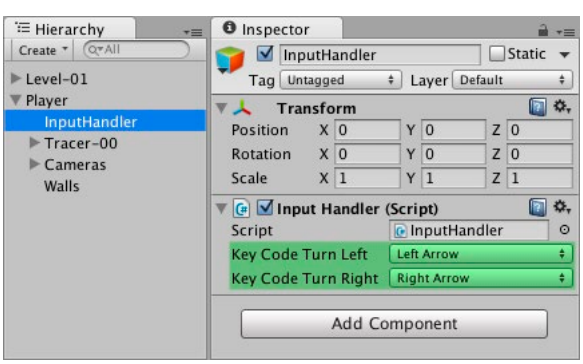

Abb. 7.7
Der InputHandler als eigenes Objekt in der Szenenhierarchie

So können wir den `InputHandler` dem neuen Slot im `TracerController` zuweisen und dann in unseren beiden `TracerControllern` (V1 und V2) den Code zur Abfrage der Tastaturkommandos so ändern wie in Listing 7.5. Das ist die Variante für V1, ändern Sie an der entsprechenden Stelle V2 analog.

7 Link auf *unity-buch.de:* Für eine Erklärung von Enumerationstypen (Enums) verweise ich auf den Artikel *Enumerationstypen (C#-Programmierhandbuch),* der von der Website zum Buch aus verlinkt ist.

Listing 7.5
Für Drehungen unseren eigenen InputHandler abfrage

```
if (inputHandler.HasTurnCommand(TurnCommand.Left)) {
    rotation = -90F;
} else if (inputHandler.HasTurnCommand(TurnCommand.Right)) {
    rotation = 90F;
}
```

Das funktioniert jetzt schon – es stellt sich aber die Frage: Wann wird innerhalb eines Frame die `Update()`-Methode von `InputHandler` aufgerufen und wann die `Update()`-Methode des `TracerControllers`, die ja dann auf den Spielbefehl zugreift, der durch den letzten Aufruf der `Update()`-Methode in `InputHandler` gespeichert wurde.

Im Moment lautet die Antwort: Wir wissen es nicht. Entweder vorher oder nachher. Irgendwann. Bis Unity 3.5 war das übrigens die endgültige Antwort. Inzwischen können wir die Reihenfolge aber bestimmen.

7.4 Die Reihenfolge der Scriptaufrufe bestimmen

Normalerweise ist uns die Reihenfolge, in der Unity die Methoden in unterschiedlichen Scripts aufruft, egal – und sie sollte es auch sein. Falls Sie in Ihrem Script voraussetzen, dass z. B. die `Update()`-Methode eines bestimmten Scripts vor allen anderen oder nach allen anderen oder zu einem ganz bestimmten Zeitpunkt zwischen zwei anderen Scripts aufgerufen wird, dann sollten Sie sich zuerst fragen: Ist das wirklich notwendig? Gibt es nicht eine Lösung, die allgemeiner funktioniert?

Es gibt bestimmte Operationen (z. B. das Ausrichten einer Kamera auf ein bestimmtes Spielobjekt in Bewegung), die immer nach allen Update-Aufrufen durchgeführt werden sollen. Für diesen Fall empfiehlt sich die Implementierung von `LateUpdate()` statt `Update()`. Das heißt, für diesen Fall gibt es eine andere Lösung, die Sie bevorzugen sollten.

In unserem Fall funktioniert es so oder so: Wenn zuerst die `Update()`-Methode von `TracerController` aufgerufen wird und erst dann die `Update()`-Methode von `InputHandler`, dann wird das Kommando eben im nächsten Frame ausgewertet. Wirklich problematisch wäre nur, wenn sich die Reihenfolge in jedem Frame ändert. Aber das passiert nicht. Es funktioniert aber etwas besser bzw. schneller, wenn wir dafür sorgen, dass die Methoden von `InputHandler` vor allen anderen aufgerufen werden. So können wir sicherstellen, dass der Tastaturbefehl immer im gleichen Frame ausgewertet wird und nicht vielleicht erst einen Frame später.

Rufen Sie dazu über das Menü *Edit/Project Settings/Script Execution Order* den MonoManager im *Inspector* auf. Nun können Sie das Script `InputHandler` aus dem Project Browser direkt über `Default Time` ziehen, wie in Abb. 7.8 illustriert. Klicken Sie dann auf `Apply`.

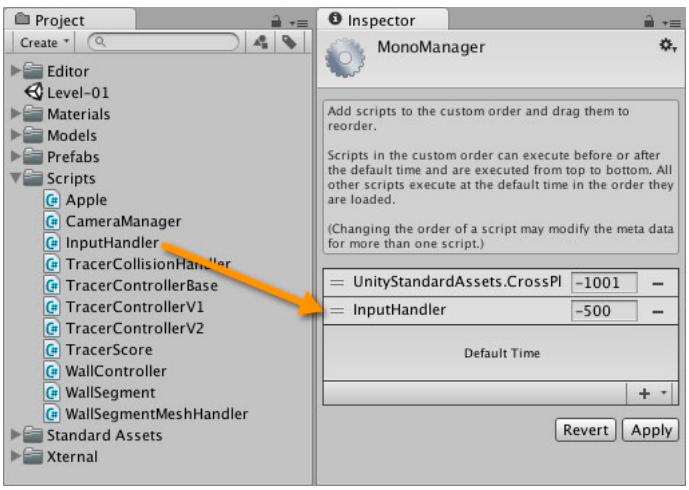

Abb. 7.8
Den InputHandler vor allen anderen Scripts ausführen

Sie können alle hier gelisteten Scripts unter Verwendung des *Sort-Icons* über oder unter die `Default Time`, also die Standardzeit ziehen, und natürlich auch die Reihenfolge von mehreren Scripts untereinander bestimmen. Alle Scripts, die nicht in der Liste stehen, werden in beliebiger Reihenfolge zur `Default Time` ausgeführt. Falls Sie aber irgendwann mehr Zeit mit dem Finden einer optimalen Ausführungsreihenfolge der Scripts verbringen als mit dem Entwickeln Ihrer Spielideen, dann denken Sie nochmals an die beiden Fragen:

1. Ist das wirklich notwendig?
2. Gibt es nicht eine Lösung, die allgemeiner funktioniert?

Die komplett aufgeräumte Version des Projekts finden Sie auf der Website zum Buch. Das ist die Datei `0090_Prototyp_Polishing.zip`.

Download von unity-buch.de

Wir haben jetzt einen aufgeräumten Prototyp und können uns im nächsten Schritt darum kümmern, dass aus dem Prototyp ein richtiges Spiel wird, das wir im kleinen Kreis veröffentlichen können, um erstes Feedback von Spielern zu bekommen. Sind Sie bereit?

6.2.9 Erweiterung zu Lösung C: Vektortransformation

Für die Collider brauchen wir ja auch bei Lösung C alles, was wir für Lösung B benötigt hatten. Die visuelle Darstellung der Wand soll jetzt aber über ein eigenes Modell erfolgen, dessen einzelne Punkte wir prozedural verändern. Damit gehen wir gewissermaßen »unter die Motorhaube« und eröffnen uns ganz neue Möglichkeiten: Sie könnten sogar so weit gehen, basierend auf Unity ein Modelling-Tool zu entwickeln. Oder Spiele, bei denen die Spieler ihre Avatare komplett selbst gestalten können. Oder Spiele mit von Grund auf prozedural erzeugter Geometrie. Oder … Bevor wir uns im *Feature Creep*[43] verlieren, kommen wir wieder zurück zum Thema: Für unsere Lösung C verwenden wir das simple Modell `TraceWall.fbx`, das aus zwei quadratischen Flächen besteht, die über jeweils vier Punkte definiert sind.

Sie können sich dieses Modell von *unity-buch.de* herunterladen. Dort finden Sie es unter dem Namen `TraceWall_Model.zip`.

Download von unity-buch.de

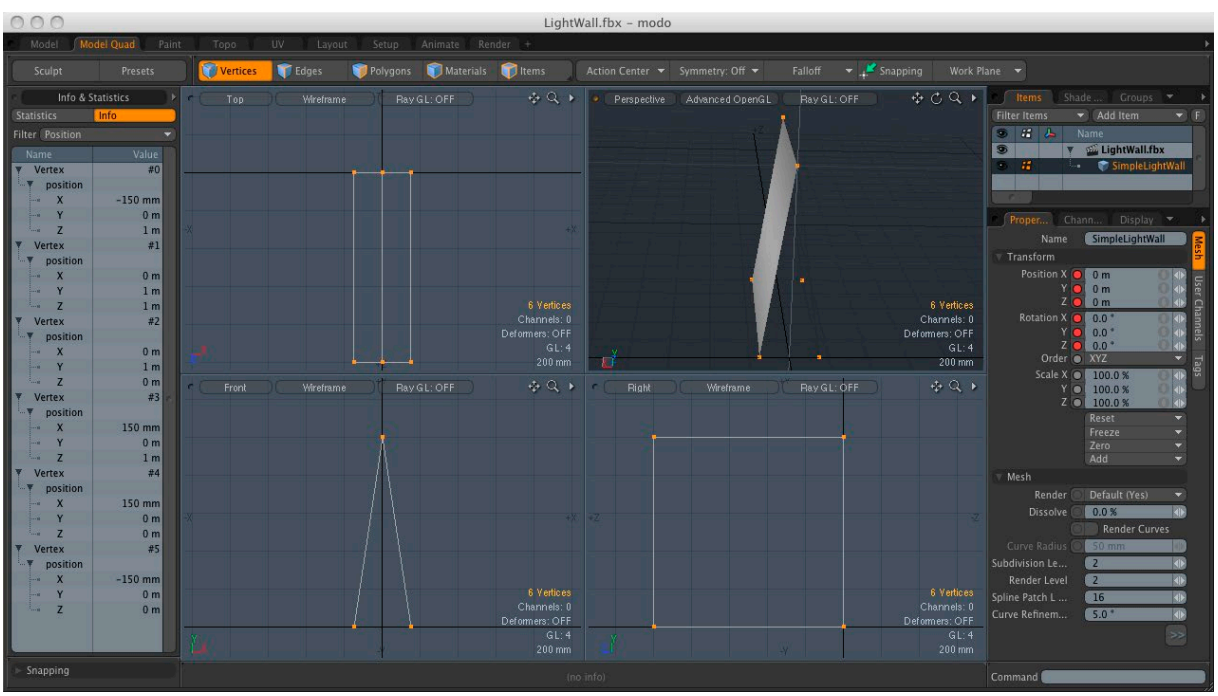

Abb. 6.58
Ein einfaches Modell für die Wände in modo

Oder Sie legen sich mit einem Modelling-Tool Ihrer Wahl ein eigenes Modell an. Achten Sie dabei bitte auf die präzisen Koordinaten der 6 bzw. 8 Punkte (die oberen beiden Punkte der Quadrate liegen aufeinander, daher kommt man auch mit 6 Punkten aus). Die unteren Punkte liegen auf `X = -0.15`

43 Link auf *unity-buch.de:* Feature Creep ist, wenn man mitten im Projekt ein Feature nach dem anderen neu dazuerfindet und auf diese Weise nie fertig wird. Einige Artikel dazu finden Sie von der Website zum Buch aus verlinkt.

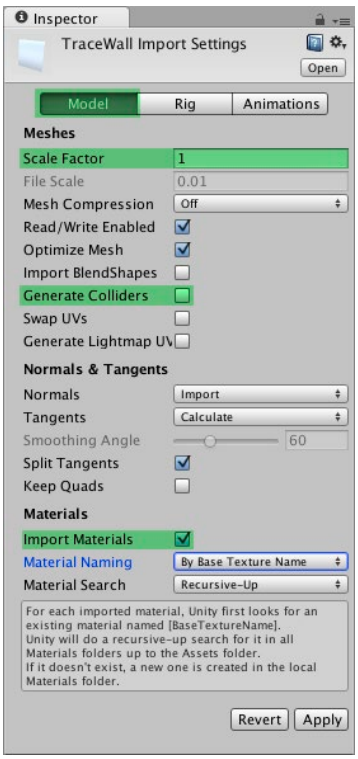

Abb. 6.59
Import-Einstellungen für 3D-Modelle

bzw. `X = 0.15`, `Y = 0` und `Z = 0` bzw. `Z = 1`. Die oberen Punkte liegen auf `X = 0`, `Y = 1` und `Z = 0` bzw. `Z = 1`. Etwas anschaulicher ist das in Abb. 6.58. Die beiden Quadrate ergeben sich letztlich aus einem extrudierten Dreieck, und entscheidend ist dabei, dass das erste Dreieck auf `Z = 0` liegt und nach `Z = 1` extrudiert wird.

6.2.10 Das Modell für die Wand in Unity importieren

Für die 3D-Modelle legen wir in unserem Unity-Projekt das neue Verzeichnis `Models` an und ziehen die Datei `TraceWall.fbx` einfach in dieses Verzeichnis. Unity legt dabei automatisch ein Unterverzeichnis `Materials` an, in dem es das automatisch importierte Material `TraceWallMaterial` ablegt. Wenn Sie das Modell `TraceWall` im *Project Browser* selektieren, sehen Sie im *Inspector* die Import-Einstellungen (siehe Abb. 6.59). Für uns sind im Moment im Bereich `Model` nur `Scale Factor`, `Generate Colliders` und `Import Materials` wichtig. Außerdem deaktivieren wir in `Animations` und `Rig` gleich noch alles, was mit Animationen zu tun hat. Die Erklärungen zu den anderen Eigenschaften können Sie nachlesen, wenn Sie das Hilfe-Icon anklicken.

`Scale Factor` sollte so eingestellt sein, dass die Einheiten in Ihrem Modelling-Programm den Einheiten von Unity entsprechen. Bei Dateien im FBX-Format steht hier inzwischen normalerweise 1. Das war vor Unity 5 anders: Da musste bei FBX-Dateien `0.01` stehen, um die korrekte Skalierung zu erhalten (das passiert jetzt mit `File Scale`).

Pro-Tipp

> Ein kleiner Trick, um die korrekte Skalierung zu überprüfen, besteht darin, das Modell in die Szene zu ziehen und neben einen Einheitswürfel zu positionieren. Wenn Sie beispielsweise das Modell eines Menschen in Unity importieren, der 1,70m groß sein soll, können Sie bei dem Einheitswürfel `Scale = (1, 1.7, 1)` setzen. Der importierte Mensch sollte dann ungefähr so hoch sein wie der Würfel.

Mit `Generate Colliders` können Sie automatisch sogenannte `MeshCollider` erzeugen. Diese sind praktisch, weil sie automatisch exakt der Form des Modells entsprechen, haben aber gravierende Einschränkungen, wie z. B. dass Kollisionen zwischen zwei `MeshCollidern` nicht erkannt werden, sondern nur zwischen primitiven Collidern (Box, Sphere usw.) und `MeshCollidern`. Außerdem kann die Verwendung von `MeshCollidern` zu Performanceproblemen führen, vor allem auf mobilen Geräten. Da wir uns um den Collider unserer Wand schon gekümmert haben, können wir hier sowieso ohne Bedenken auf das automatische Erzeugen eines `MeshColliders` verzichten.

`Import Materials` bestimmt – wie der Name schon sagt –, ob Unity automatisch Materialien erzeugen soll oder nicht. Das ist meistens sehr praktisch, daher lassen wir das auch aktiv. Manchmal möchte man aber für

mehrere Modelle ein einziges Material verwenden, und dann stören die von Unity automatisch erzeugten Materialien meistens.

Pro-Tipp, Link auf unity-buch.de

Wenn Sie sich bereits beim Modellieren Ihrer 3D-Assets an entsprechende Namenskonventionen für Texturen bzw. Materialien halten, können Sie mit den Einstellungen unter `Material Naming` (sichtbar wenn `Import Materials` aktiv ist) dafür sorgen, dass Unity Ihren Modellen immer die richtigen Materialien zuweist. Bei entsprechenden Einstellungen unter `Material Search` klappt das sogar, wenn die Materialien an ganz anderer Stelle im Projekt abgelegt sind als die Modelle. Durch Implementierung eines eigenen `AssetPostprocessor` können Sie hier nach dem Import sogar beliebig komplexe Logiken für die Zuweisung Ihrer Materialien (und anderer Import-Einstellungen) vornehmen. Den Einstieg dazu finden Sie auf der Website zum Buch unter *AssetPostprocessor* verlinkt.

Das Importieren von Animationen können wir uns bei unserem Modell sparen, weil es gar keine Animationen hat. Unity erzeugt aber eine Animationskomponente, wenn wir das nicht abschalten. Gehen Sie dazu am besten folgendermaßen vor:

1. Wählen Sie zuerst `Animations`, und deaktivieren Sie die Checkbox `Import Animation`.
2. Wechseln Sie dann zu `Rig`, und wählen Sie bei `Animation Type` den Wert `None`.

An sich könnten Sie auch einfach nur bei `Rig` den `Animation Type` setzen. So können Sie aber unter `Animations` die Checkbox nicht mehr deaktivieren und erhalten einen Text *No animation data available in this model*, was zwar nicht wirklich stört, aber einfach überflüssig ist.

6.2.11 Ein neues Prefab für die Wände erstellen

Ähnlich wie bei unseren beiden Varianten zur Steuerung des Fahrzeugs (`TracerController` und `TracerControllerV2`) wollen wir bei den Wänden auch die Möglichkeit haben, Lösung B und Lösung C jederzeit auszutauschen. Daher legen wir dieses Mal von dem kompletten Prefab `ColliderParent` ein Duplikat an (⌘+D bzw. Strg+D) und nennen es `ColliderParentVectors`.

An dem neuen Prefab müssen wir einige strukturelle Änderungen durchführen. Das geht am einfachsten, wenn wir es vom Projekt in die Szene ziehen und dann in der Szene bearbeiten. Ziehen Sie also `ColliderParentVectors` aus dem *Project Browser* in die *Hierarchy View* (1). Ziehen Sie als Nächstes `TraceWall` aus dem *Project Browser* direkt auf `ColliderParentVectors` in der *Hierarchy View* (2). Die beiden Drag&Drop-Vorgänge sind in Abb. 6.60 veranschaulicht, wobei Sie `ColliderParentVectors` in

einen leeren Bereich in der *Hierarchy View* ziehen sollten, um ihn nicht versehentlich zu einem Kind eines anderen Objekts zu machen (was wir beim Ziehen von `TraceWall` auf `ColliderParentVectors` ja wollen).

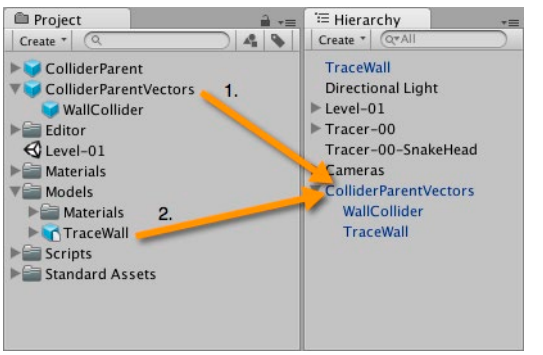

Abb. 6.60
Prefab in die Szene und TraceWall auf Prefab ziehen

Den `MeshRenderer` des GameObjects `WallCollider` unter `ColliderParentVectors` müssen wir natürlich deaktivieren, damit unser neues, höchst elegantes Wandmodell zum Vorschein kommt.

6.2.12 Änderungen an Prefabs von der Szene ins Projekt zurückschreiben

Wahrscheinlich fällt Ihnen auf, dass sich an unserem Prefab im *Project Browser* noch nichts geändert hat. Änderungen an Prefabs innerhalb der Szene betreffen generell zunächst nur die Instanz des Prefabs in der Szene. Wenn wir wollen, dass die Änderungen für das Prefab selbst gelten, müssen wir diese Unity explizit mitteilen, indem wir im *Inspector* den `Apply`-Button drücken.

Normalerweise muss dazu bei hierarchischen Prefabs nur irgendein GameObject in der Hierarchie des Prefabs selektiert sein. Mit Selektion auf `WallCollider` würde das auch jetzt funktionieren. Wenn Sie aber zufällig `TraceWall` selektiert hätten, würde es nicht funktionieren, weil `TraceWall` ja noch nicht offiziell ein Teil unseres Prefabs ist. Erkennen können Menschen mit Adleraugen den Unterschied daran, dass bei TraceWall im Moment statt »Prefab« noch »Model« steht und statt dem `Apply`-Button ein `Open`-Button erscheint. Die richtige Selektion mit dem richtigen Button zeigt Abb. 6.61. *Neo, ich kann dir nur die Tür, äh, den Button zeigen. Klicken musst du ihn selbst!*

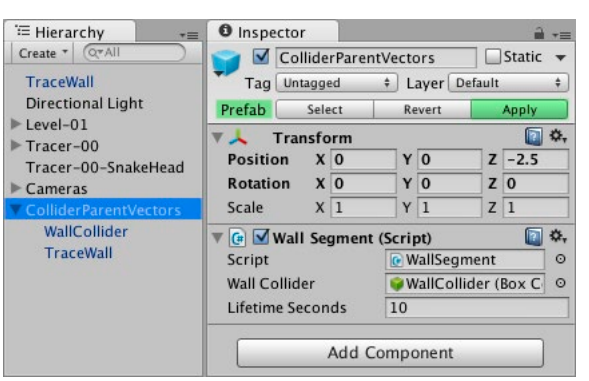

Abb. 6.61
Ein Prefab an der Schwelle, aus der Szene in das Projekt zu transzendieren

Sobald Sie auf den Button geklickt haben und damit die `TraceWall` hochoffiziell in `ColliderParentVectors` assimiliert wurde, ist ihre Natur als Model vergessen und Sie könnten auch bei selektierter `TraceWall` die Änderungen am Prefab zurück in das Projekt teleportie-

ren. Sobald die Änderungen in das Projekt gesichert sind, können wir die Prefab-Instanz wieder aus der Szene löschen.

Wenden wir uns nun also wieder den Freuden der oldschool Spieleprogrammierung zu:

6.2.13 Prozedural das Modell-Mesh verändern

Für das Ändern des Meshes erstellen wir ein eigenes Script mit dem Namen `WallSegmentMeshHandler`. Den Code dazu finden Sie in Listing 6.24.

Listing 6.24
Das neue Script WallSegmentMeshHandler

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

[RequireComponent(typeof(WallSegment))]
public class WallSegmentMeshHandler : MonoBehaviour {

  public Transform meshFilter;

  private WallSegment wallSegment;
  private Mesh wallMesh;
  private Vector3[] vertices;
  private List<int> verticesToMoveIndexes = new List<int>();

  void Awake() {
    wallSegment = GetComponent<WallSegment>();
    wallMesh = meshFilter.GetComponent<MeshFilter>().mesh;
    vertices = wallMesh.vertices;
    for (int i=0; i < vertices.Length; i++) {
      if (vertices[i].z > 0) {
        verticesToMoveIndexes.Add(i);
      }
    }
  }
  void LateUpdate() {
    foreach (int vertexToMoveIndex in verticesToMoveIndexes) {
      vertices[vertexToMoveIndex].z = wallSegment.Distance;
    }
    wallMesh.vertices = vertices;
  }
}
```

Lassen Sie sich nicht davon verwirren, dass `WallSegment` im Moment noch kein öffentliches Property `Distance` hat und Unity dementsprechend mit einem Kompilierfehler meckert, falls wir ihm den Fokus geben. Darum kümmern wir uns gleich.

Das Erste, was Ihnen wahrscheinlich auffällt, ist, dass wir hier statt der Methode `Update()` die Methode `LateUpdate()` verwenden:

Die Methode `LateUpdate()` wird aufgerufen, nachdem alle `Update()`-Methoden aufgerufen wurden und bevor der Frame gerendert wird. Damit ist sichergestellt, dass alle Änderungen an der Szene bereits durchgeführt sind (zumindest wenn diese Änderungen in `Update()` und `FixedUpdate()` implementiert sind). `LateUpdate()` wird beispielsweise häufig für Kamerascripts verwendet, die Objekten in der Szene folgen.

Wir verwenden das hier deswegen, weil die Distanz in der Update()-Methode von WallSegment berechnet wird und diese Berechnung auf jeden Fall für den aktuellen Frame stattgefunden haben muss, bevor wir die Wand entsprechend anpassen. Weitere Möglichkeiten, die Reihenfolge von Scriptaufrufen zu beeinflussen, lernen Sie in Abschnitt *7.4, Die Reihenfolge der Scriptaufrufe bestimmen,* kennen.

Interessant ist in unserem neuen Script außerdem die Verwendung der Unity-Komponente `MeshFilter`, über die wir ja schon in Abschnitt *4.2, Level 01: Das Quadrat – Modeling in Unity,* ausführlich gesprochen haben. Diese bietet Zugriff auf eine Instanz der Klasse `Mesh`, die uns unter anderem ermöglicht, jeden einzelnen Punkt des Modells vom Programmcode aus zu modifizieren. Da `Vector3` wie oben erwähnt ein *Struct* ist, können wir die für uns relevanten Punkte nicht einfach in einer Liste speichern und davon ausgehen, dass Änderungen an den Punkten in der Liste eine Auswirkung auf das Modell haben (*by-value*). Stattdessen halten wir das komplette Array der Punkte vor und speichern die Indizes der relevanten Punkte in einer Liste. Relevant sind für uns die Punkte, deren Z-Wert größer als `0` ist.

Daher habe ich bei dem Modell auf die exakte Einhaltung der Koordinaten bestanden. Falls an dieser Stelle (oder an einer gleichartigen Stelle in einem zukünftigen Projekt) bei Ihnen etwas nicht funktionieren sollte, können Sie unter Verwendung von `Debug.Log()` die Koordinaten der Punkte in die Konsole ausgeben.

Nach der Änderung der Z-Koordinate der relevanten Punkte müssen wir natürlich das komplette Array wieder an das Mesh übergeben. Wie gesagt: Structs werden *by-value* gespeichert, nicht *by-reference*.

Erweitern wir nun die Klasse `WallSegment` wie in Listing 6.25 beschrieben.

Listing 6.25
Die Entfernung in WallSegment verfügbar machen

```
public IEnumerator DestroyAfterLifetime() {
   yield return new WaitForSeconds(lifetimeSeconds);
   Destroy(this.gameObject);
}

private float distance = 0;
public float Distance {
   get { return distance; }
}
```

```
void Update() {
   if (growing) {
       distance = Vector3.Distance(
             transform.position,
             tracerReferencePoint.position);
       Vector3 scale = transform.localScale;
       scale.z = distance;
       transform.localScale = scale;
   }
}
```

Ist Ihnen aufgefallen, dass vor `distance` in der `Update()`-Methode nicht mehr der Typ (`float`) steht? Falls Sie das übersehen, hätten wir eine lokale Variablendeklaration, die unsere neue private Membervariable `distance` verschattet, und dann würde unser neues Property `Distance` immer den initialen Wert 0 liefern. ***Solche Fehler können eine Menge Zeit kosten – also Vorsicht!***

Unity sollte nun die neue Klasse problemlos kompilieren, und Sie können das neue Script `WallSegmentMeshHandler` entweder auf das Prefab `ColliderParentVectors` im Projekt ziehen (Project Browser) oder auf die Instanz des Prefabs in der Szene (Hierarchy View), sofern Sie diese nicht oben gelöscht haben. Falls Sie Letzteres bevorzugen, dürfen Sie nur nicht vergessen, die Änderung mit `Apply` zurück ins Projekt zu speichern.

Auf den Slot `Mesh Filter` der Komponente `WallSegmentMeshHandler` ziehen Sie bitte das GameObject `TraceWall`. Das tun Sie wieder entweder direkt im Projekt oder in der Szene, und danach klicken Sie auf `Apply`. Und bei der Gelegenheit können wir auch mal wieder Szene und Projekt speichern.

Schließlich müssen wir unserem `Tracer-00` in der Szene das neue `Wall Prefab` aus dem Projekt zuweisen. Ziehen Sie dazu `ColliderParent-Vectors` aus dem Projekt in den Slot `Wall Prefab` von `Tracer-00`.

Jetzt können wir mal wieder ein wenig spielen und unser Meisterwerk bewundern. Immerhin: Die neue Wand verlängert sich. Nur sieht das nicht ganz so aus, wie wir uns das gedacht hatten, sondern so wie in Abb. 6.62.

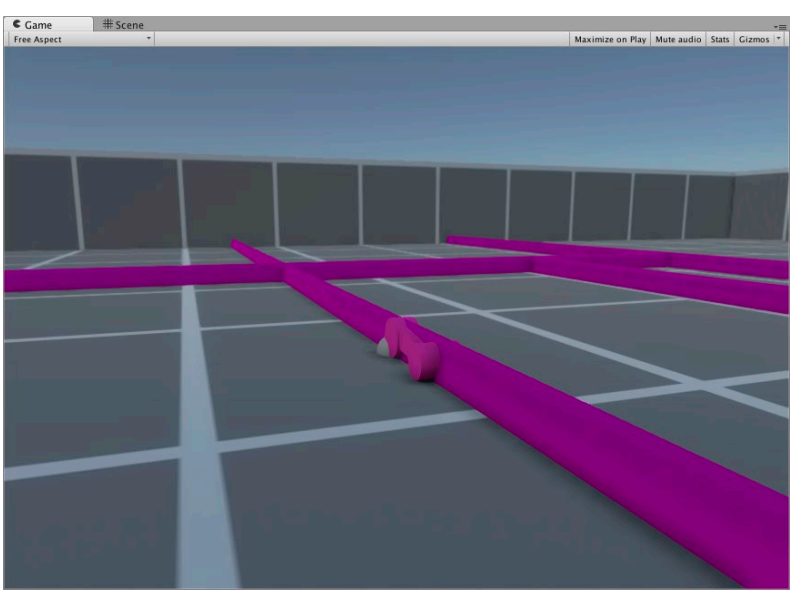

Abb. 6.62
Mit den Wänden ist etwas schiefgelaufen.

6.2.14 Den Fehler finden

Es gibt mehrere Kandidaten für solche Fehler. Ein Problem könnte sein, dass die Änderungen an den Punkten unseres Modells nicht korrekt sind. Beispielsweise wäre es theoretisch denkbar, dass Unity die Koordinaten intern anders handhabt. Das ist aber nicht der Fall, und wir können auch leicht prüfen, dass `TraceWall` immer noch `Scale = (1, 1, 1)` hat, also nicht eine Skalierung von `TraceWall` das Problem verursacht.

Da wir zum Skalieren unseres Colliders den exakt gleichen Faktor verwenden wie für unser Modell, ist es offenbar auch kein Fehler in der Berechnung der zurückgelegten Distanz. Aber was skalieren wir da eigentlich genau? Sehen wir uns doch noch mal den Code von `WallSegment` an (siehe Listing 6.26):

Listing 6.26
Welches Transform hätten Sie gern?

```
void Update() {
   if (growing) {
       distance = Vector3.Distance(
             transform.position,
             tracerReferencePoint.position);
       Vector3 scale = transform.localScale;
       scale.z = distance;
       transform.localScale = scale;
   }
}
```

Die Komponente `WallSegment` haben wir auf dem GameObject `ColliderParentVectors`. Das heißt, wir skalieren `ColliderParentVectors`. `TraceWall` hängt unter `ColliderParentVectors`, also erbt es auch die Skalierung. So war das natürlich nicht gedacht.

Testen wir kurz die Hypothese: Kommentieren Sie die letzte Zeile aus, mit der die Skalierung in die Transform-Komponente geschrieben wird (Listing 6.27).

Listing 6.27
Ursachen-Hypothese in WallSegment testen

```
Vector3 scale = transform.localScale;
scale.z = distance;
//transform.localScale = scale;
```

Jetzt funktioniert zumindest dieser Teil wie gewünscht. Natürlich haben wir jetzt auch die Skalierung unseres Colliders deaktiviert, d. h., wir müssen die Zeile natürlich wieder einkommentieren. Aber wir wissen jetzt, dass hier der Fehler liegt.

Die Lösung ist dann recht einfach: Wir führen in WallSegment direkt unter der Zeile `public Collider wallCollider;` eine zusätzliche öffentliche Variable `public Transform colliderScale;` ein und schreiben die Update-Methode einfach um, wie in Listing 6.28.

Listing 6.28
Den Fehler korrigieren

```
Vector3 scale = colliderScale.localScale;
scale.z = distance;
colliderScale.localScale = scale;
```

In unserem alten Prefab `ColliderParent` können wir jetzt einfach `ColliderParent` auf den neuen Slot `Collider Scale` ziehen. Damit entspricht das Verhalten des alten Prefab auch mit der neuen Implementierung dem, das wir bereits getestet haben. Natürlich sollten Sie das verifizieren, indem wir das Prefab `ColliderParent` aus dem Projekt auf den Slot `Wall Prefab` im `WallController` von `Tracer-OO` ziehen und das Spiel kurz testen. Danach ziehen wir wieder `ColliderParentVectors` auf den Slot `Wall Prefab`, da wir jetzt wieder mit unserem neuen Prefab arbeiten.

In unserem neuen Prefab brauchen wir jetzt ein neues, leeres GameObject als Zwischenebene für die Skalierung. Diese Änderung müssen wir wieder in der Szene vornehmen. Ziehen Sie also `ColliderParentVectors` aus dem Projekt in die Szene. Erzeugen Sie ein neues, leeres GameObject `ColliderScale` unter `ColliderParentVectors`, und setzen Sie `Position = (0, 0, 0)`. Jetzt können wir `WallCollider` unter `ColliderScale` ziehen. Allerdings beschwert sich Unity mit der Meldung »*Losing Prefab. This action will lose the prefab connection. Are you sure you wish to continue?*« (siehe Abb. 6.63). Ja, wir sind sicher. Also klicken wir auf `Continue`.

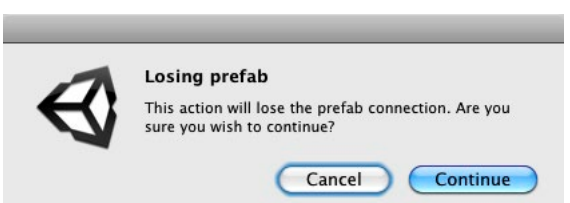

Abb. 6.63
Losing Prefab – das macht aber nichts.

Losing Prefab – nicht wirklich …

Diese Meldung erscheint immer dann, wenn wir die Struktur eines Prefabs grundsätzlich verändern. Praktischerweise ist die Fehlermeldung nicht ganz korrekt: Unity behält nämlich die Referenz auf das Prefab durchaus. Die Meldung bedeutet lediglich, dass ab diesem Zeitpunkt Änderungen im Prefab nicht mehr automatisch auf die Prefab-Instanz übertragen werden. Das erkennen Sie auch daran, dass die Instanz nicht mehr blau ist. Wir können aber immer noch den `Apply`-Button nutzen, um die Änderungen von der Instanz in das Prefab zurückzuschreiben. Damit stellen wir auch die vollständige Verbindung wieder her. Das bedeutet, die Prefab-Instanz erscheint in der Szene wieder blau und übernimmt automatisch alle Änderungen am Prefab selbst. Nur sollten Sie niemals vergessen, möglichst bald diese Verbindung wiederherzustellen – sonst bekommen Sie möglicherweise später ein Durcheinander in Ihrem Projekt. Klicken Sie also jetzt `Apply`!

Wir müssen jetzt noch unser neues Zwischenobjekt `ColliderScale` auf den dafür vorgesehenen Slot *Collider Scale* im Wallsegment ziehen. Sobald Sie alle Änderungen an der Prefab-Instanz mit dem `Apply`-Button in das Prefab im Projekt zurückgesichert haben, können wir dann auch die Prefab-Instanz in der Szene wieder löschen.

6.2.15 Beschränkung der Prefab-Ebenen im Project Browser

Vielleicht fällt Ihnen auf, dass im *Project Browser* nur die ersten beiden Hierarchie-Ebenen eines Prefabs angezeigt werden. `WallCollider` liegt jetzt bei `ColliderParentVectors` auf der dritten Ebene und ist damit im *Project Browser* nicht sichtbar oder änderbar. Abb. 6.64 veranschaulicht diesen Sachverhalt.

Aufgrund dieses Umstands sollten Sie tiefe hierarchische Strukturen in Prefabs nach Möglichkeit vermeiden und Komponenten, an denen Sie häufig Änderungen vornehmen, nur an GameObjects auf den ersten beiden Ebenen hängen. Am übersichtlichsten ist an sich, die relevanten Komponenten nur an das GameObject in der ersten Ebene zu hängen. In unserem Fall ist das aber schwer umsetzbar. Wollen wir also den `MeshRenderer` von `WallCollider` ein- oder ausschalten oder Veränderungen an der `BoxCollider`-Komponente vornehmen, kommen wir nicht daran vorbei, das Objekt in die Szene zu ziehen, die Änderungen dort vorzunehmen, die Änderungen mittels `Apply` zu speichern und das Objekt wieder aus der Szene zu löschen. Prefabs sind ein Bereich in Unity, an dem es in zukünftigen Versionen sicher noch einige Änderungen geben wird.

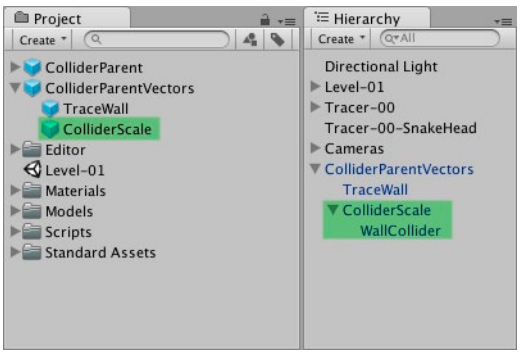

Abb. 6.64
Prefabs im Projekt und ihre Instanzen in der Szene

Anstatt das Zwischenobjekt `ColliderScale` zu verwenden, könnten wir auch den `BoxCollider` mithilfe der `Center`-Eigenschaft verschieben. Das hat aber den Nachteil, dass dann das `Cube`-Mesh nicht mehr identisch mit dem `BoxCollider` ist. Eine Möglichkeit, dieses Problem zu lösen, wäre das Erstellen eines eigenen Modells, das entsprechend im Raum positioniert ist. Diesen Aufwand wollen wir aber nicht betreiben, sondern nehmen stattdessen dieses Beispiel einfach als Ausnahme von der Regel. Solche Ausnahmen werden Ihnen noch öfter begegnen! Daher ist es gut, sich an die Vorgehensweise zu gewöhnen: Prefab in Szene ziehen, bearbeiten, Änderungen mit `Apply` speichern, Prefab-Instanz aus Szene löschen.

6.2.16 Wenn sichtbare Flächen unsichtbar werden

Sobald Sie das Spiel wieder testen,[44] werden Sie recht schnell feststellen, dass mit unserer aktuellen Lösung etwas nicht stimmt: Die Wand erscheint zwar und vergrößert sich auch wie gewünscht – aber sie verschwindet auf mysteriöse Weise auch immer wieder. Und je nach Perspektive erscheinen dann Wände, wo wir vorher keine gesehen hatten. **Woran könnte so etwas liegen?**

44 Natürlich erst, nachdem Sie alle Änderungen sauber abgespeichert haben!

Unity verwendet eine Technik namens *View Frustum Culling*[45], die dazu dient, dass nur diejenigen Objekte gezeichnet werden, die auch für die Kamera sichtbar sind. Dazu muss Unity aber die Ausmaße aller Objekte kennen. Normalerweise werden diese automatisch von Unity berechnet, und wir müssen uns um solche Feinheiten nicht kümmern. Aber so, wie es aussieht, benutzt Unity in unserem Fall noch die ursprünglichen Ausmaße unserer Wandsegmente. Unsere Wände sind nämlich genau dann sichtbar, wenn der Anfangspunkt der Wand sichtbar ist, und sie verschwinden, wenn dieser Anfangspunkt nicht sichtbar ist.

Wenn Sie sich die Methoden der Klasse `Mesh` in der Unity-Dokumentation ansehen, kommen Sie wahrscheinlich sehr schnell auf die Lösung: Es gibt nämlich eine Methode `Mesh.RecalculateBounds()`. Und da steht auch, dass wir diese Methode aufrufen müssen, nachdem wir Vektoren geändert haben. Also tun wir das auch, wie in Listing 6.29 beschrieben.

Listing 6.29 Erweiterung von Update in WallSegmentMeshHandler

```
void LateUpdate() {
   foreach (int vertexToMoveIndex in verticesToMoveIndexes) {
       vertices[vertexToMoveIndex].z = wallSegment.Distance;
   }
   wallMesh.vertices = vertices;
   wallMesh.RecalculateBounds();
}
```

Nach dieser Änderung tritt der Fehler nicht mehr auf, und wir sehen die Wände die ganze Zeit über.

6.2.17 UV-Map kontinuierlich anpassen

Einen Vorteil unserer Lösung C hatte ich wie folgt beschrieben: »Wir können die UV-Map kontinuierlich anpassen, sodass wir die Wände auch sauber texturieren können.« Natürlich brauchen wir für unseren Prototyp nicht wirklich Texturen auf den Wänden. Aber wenn wir diesen Schritt noch umsetzen, ist unsere Lösung C komplett.

Download von unity-buch.de

Zuerst brauchen wir natürlich eine geeignete Textur: Verwenden Sie dazu einfach einen Pfeil nach rechts. Eine Beispieltextur können Sie von der Website zum Buch herunterladen. Sie finden sie unter dem Namen `TraceWall_ArrowTexture.zip`.

Importieren Sie die Textur `Arrow.psd` in das Verzeichnis `Materials / Textures` in unserem Projekt, und legen Sie die Textur als `Albedo` auf das automatisch angelegte Material: `Models / Materials / BoxMat`.

45 »Frustum«: Pyramidenstumpf, »to cull«: herausfiltern. Gemeint ist also das Herausfiltern von Objekten, die außerhalb des View-Frustums liegen, also außerhalb des Pyramidenstumpfes, der durch den Sichtbereich der Kamera definiert ist.

Wenn Sie das Spiel nun testen, sehen Sie zwei Probleme: Zum einen wird die Textur mit der Wand skaliert, was sehr unschön aussieht. Zum anderen zeigt der Pfeil auf der rechten Seite der Wand in Fahrtrichtung und auf der linken Seite der Wand in die entgegengesetzte Richtung. Das zweite Problem würde man normalerweise einfach durch Korrektur der UV-Map im Modelling-Tool lösen. Wir können aber auch beide Probleme elegant im Code lösen:

Die **UV-Map** ist nichts weiter als ein Array von `Vector2`, also zweidimensionale Vektoren, die genau so angeordnet sind wie die Punkte im Raum (`vertices`). Dabei sind die Koordinaten normalisiert zwischen 0 und 1. Das heißt, `(0, 0)` ist der Punkt links unten in der Textur und `(1, 1)` der Punkt ganz rechts oben. Wir können also für jeden Punkt unseres 3D-Modells bestimmen, von welchen Koordinaten der Textur dieser Punkt seine Farbe erhält. Die Werte auf den Flächen zwischen den Punkten werden entsprechend interpoliert.

Also speichern wir die UV-Map analog unserer Punkte als neue Membervariablen in `WallSegmentMeshHandler` (siehe Listing 6.30).

Listing 6.30 Die neue Membervariable uvs in WallSegmentMeshHandler

```
private WallSegment wallSegment;
private Mesh wallMesh;
private Vector2[] uvs;
private Vector3[] vertices;
private List<int> verticesToMoveIndexes = new List<int>();
```

Die `Awake()`-Methode erweitern wir nun so, dass zunächst das Array der UV-Map aus `wallMesh` in unsere Membervariable übernommen wird und dann die X-Werte der UV-Punkte entsprechend der Z-Werte unserer Punkte im Raum gesetzt werden. Wir möchten, dass die Punkte, an denen die Wand startet, den Punkten ganz links in unserer Textur entsprechen. Daher setzen wir den X-Wert auf 0 (siehe Listing 6.31).

Listing 6.31 Erweiterung von WallSegmentMeshHandler.Awake()

```
void Awake() {
   wallSegment = GetComponent<WallSegment>();
   wallMesh = meshFilter.GetComponent<MeshFilter>().mesh;
   vertices = wallMesh.vertices;
   uvs = wallMesh.uv;
   for (int i=0; i < vertices.Length; i++) {
       if (vertices[i].z > 0) {
           verticesToMoveIndexes.Add(i);
       } else {
           uvs[i].x = 0;
       }
   }
}
```

Schließlich müssen wir den X-Wert derjenigen Punkte, die in jedem Frame neu positioniert werden, einfach auf die Distanz setzen, die sie tatsächlich vom Ursprungspunkt haben. Damit bekommen wir natürlich auf der X-Achse UV-Werte jenseits von 1 – aber genau das wollen wir auch, weil sich die Textur ja wiederholen soll. Im Code sieht das dann aus wie in Listing 6.32.

```
void LateUpdate() {
   foreach (int vertexToMoveIndex in verticesToMoveIndexes) {
       vertices[vertexToMoveIndex].z = wallSegment.Distance;
       uvs[vertexToMoveIndex].x = vertices[vertexToMoveIndex].z;
   }
   wallMesh.vertices = vertices;
   wallMesh.uv = uvs;
   wallMesh.RecalculateBounds();
}
```

Listing 6.32
Erweiterung von WallSegmentMeshHandler.LateUpdate()

Wenn Sie das Spiel jetzt starten, sehen Sie eine vollständige Implementierung von Lösung C und haben im Verlauf dieses Kapitels die größte Herausforderung gemeistert, die es bei der Implementierung dieses Spiels gibt – und dabei auch einiges über die Möglichkeiten des Scripting in Unity gelernt. **Herzlichen Glückwunsch!**

Übrigens ist die Pfeil-Textur auf der Wand mehr als nur ein grafischer Effekt, mit dem wir die Korrektur der UV-Map per Script motivieren: In einem Multiplayer-Modus können Spieler so erkennen, in welche Richtung ein anderer Spieler gefahren ist – was normalerweise anhand der Wände nicht sichtbar ist.

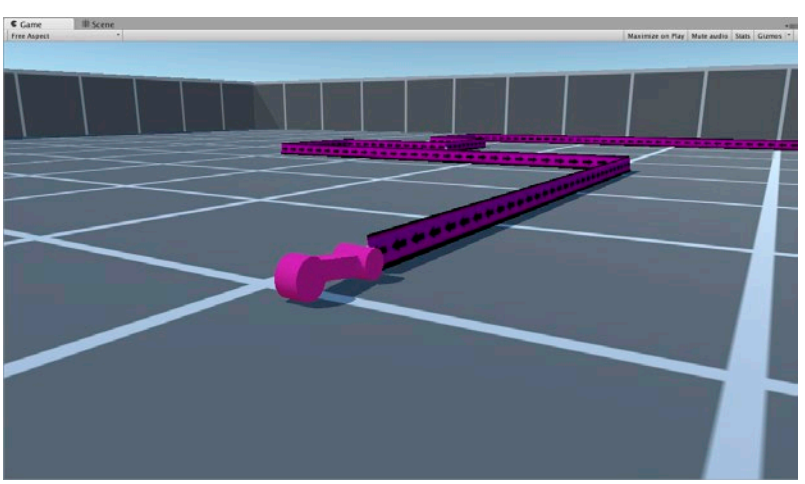

Abb. 6.65
Ganz schön was geschafft!

Wir sind jetzt mit unserem Prototyp ein entscheidendes Stück weiter gekommen. Vor allem haben wir ein gutes Stück Risiko bewältigt: Stellen Sie sich vor, Sie hätten schon 10 Level fertig modelliert und 15 verschiedene

Tracer, hätten Online-Highscore-Listen und verschiedenste PowerUps … kurz gesagt: Stellen Sie sich vor, Sie hätten schon enorm viel Zeit in das Projekt investiert und würden dann feststellen, dass das mit den Wänden und Drehungen nicht so funktioniert, wie Sie es sich gedacht hatten – im schlimmsten Fall würden Sie es dann erst merken, wenn die Spieler bereits genervte Reviews in den jeweiligen App Stores eingetragen haben.

Natürlich ist das unwahrscheinlich, da es von diesem Spielkonzept schon zig Varianten gibt, die das auch irgendwie hinbekommen haben. Aber ich hoffe natürlich, dass Sie sich an völlig unerforschte Spielkonzepte heranwagen und wirklich innovative Spielideen entwickeln. Und dafür sollten Sie aus diesem Abschnitt Folgendes mitnehmen:

Pro-Tipp

Implementieren Sie problematische Spielmechaniken möglichst früh in einem Prototyp, und finden Sie dabei heraus, ob Sie sie auch so implementieren können, dass sie wirklich Spaß machen.

Download von unity-buch.de, erste spielbare Version

Das Projekt, wie es nach diesem Kapitel aussehen sollte, finden Sie auf der Website zum Buch. Das ist die Datei `Traces_Prototype_090.zip`. Diesen Stand können Sie übrigens auch direkt vom Download-Bereich aus spielen! Folgen Sie einfach dem Link *Traces Prototype Walls!* Sie müssen allerdings hier noch einen Browser-Refresh durchführen, um das Spiel neu zu starten. Es ist also ein sehr rudimentärer Prototyp.

Für unseren Prototyp brauchen wir jetzt nur noch zwei Funktionalitäten, die dank der in Unity eingebauten Physik-Engine sehr einfach umzusetzen sind: Unser Tracer muss natürlich explodieren, wenn er gegen eine Wand fährt, und wir brauchen unsere »Äpfel«, also Items zum Einsammeln. So ähnlich wie das Einsammeln der Äpfel könnten Sie auch das Einsammeln von PowerUps implementieren.

6.3 Von Äpfeln und Explosionen, Triggern und Kollisionen

Vielleicht ist Ihnen beim Testen schon aufgefallen, dass unser Tracer sich nach der Kollision mit einer Wand etwas seltsam in irgendeiner Richtung bewegt. Das liegt daran, dass die Physik-Engine von Unity aufgrund der Kollision die Richtung und Geschwindigkeit des Rigidbody ändert.

Über Physik-Materialien (`PhysicMaterial`), die man den Collidern zuweisen kann, könnte man hier die konkreten Eigenschaften wie Reibung (`Dynamic Friction/Static Friction`) und Rückprallstärke (`Bounciness`) einstellen und auch festlegen, wie diese Werte von zwei an einer Kollision beteiligten Collidern kombiniert werden. An dieser Stelle brauchen wir das aber nicht. Ganz im Gegenteil sieht das derzeitige Verhalten eher wie ein

Jashan Chittesh ist Diplom-Informatiker und arbeitet freiberuflich als Softwareentwickler. Seit 2007 entwickelt er mit Unity Computerspiele und gründete 2011 dazu die Firma narayana games. Seine Kenntnisse gibt er in Form von Unity-Workshops unter anderem an der Filmakademie Ludwigsburg weiter.

Jashan Chittesh

Das Unity-Buch

2D- und 3D-Spiele entwickeln mit Unity 5

dpunkt.verlag

Jashan Chittesh
jashan@narayana-games.net

Lektorat: René Schönfeldt
Copy-Editing: Friederike Daenecke, Zülpich
Satz: Petra Strauch, Bonn
Herstellung: Frank Heidt
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: Stürtz GmbH, Würzburg

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.

ISBN:
Buch 978-3-86490-232-1
PDF 978-3-86491-679-3
ePub 978-3-86491-680-9

1. Auflage 2015

Wieblinger Weg 17
69123 Heidelberg

5 4 3 2 1 0

Vorwort

Unity ist eine Entwicklungsumgebung für Spiele und interaktive 3D-Anwendungen sowohl für Einsteiger als auch für Profis. Durch eine außergewöhnlich intuitive und einfache Benutzeroberfläche ist diese Game-Engine sehr gut zum Einstieg in die Spielentwicklung geeignet. Darüber hinaus bietet Unity auch dem fortgeschrittenen Spielentwickler beeindruckende und fast unbegrenzte kreative Möglichkeiten. Natürlich darf auch mit einem so komfortablen und doch mächtigen Werkzeug der Aufwand bei der Spielentwicklung nicht unterschätzt werden. Mit dem richtigen Know-how, etwas Talent und vor allem Beständigkeit kann mit Unity jedoch auch ein Einsteiger eine auf seine jeweiligen Fähigkeiten abgestimmte Spielidee in absehbarer Zeit in ein spielbares Produkt verwandeln. Sie werden nicht schon übermorgen das nächste große MMO auf dem Markt haben (der große grüne Button dazu fehlt Unity noch ;-)). Aber wenn Sie sich mit Unity intensiver befassen, haben Sie sicher in wenigen Monaten das eine oder andere kleine Spiel, das Ihnen, Ihren Freunden und Bekannten Freude bereitet. Und wer weiß: Manchmal haben auch kleine, einfache Spiele großen Erfolg. Dann erfreuen Ihre Spiele vielleicht sogar die ganze Welt! Und wenn Sie dranbleiben, haben Sie vielleicht eines Tages Ihr eigenes Spielentwicklungsstudio und produzieren richtig große Titel. Dieses Buch ist dazu vielleicht der Anfang.

Über dieses Buch

Das Unity-Buch hat vor allem die Absicht, das entsprechende Know-how auf allen relevanten Ebenen zu vermitteln und interessierten Anwendern mit grundsätzlichen Programmierkenntnissen einen möglichst mühelosen Einstieg in die Spielentwicklung mit Unity zu bieten.

Ich möchte Sie also auf eine Reise einladen, bei der wir gemeinsam auf spielerische Weise ein Spiel mit mehreren Variationen entwickeln. Dabei begegnen wir verschiedenen Bereichen von Unity, meistern Herausforderungen und entdecken in kleinen und größeren Schritten die unbegrenzten Möglichkeiten der Spielentwicklung. Gemeinsam betreten wir hier eine Welt, deren Horizont so weit reicht wie unsere Fantasie. Alles, was wir dann

noch brauchen, ist die Bereitschaft, unsere Zeit der Verwirklichung der Ideen zu schenken, die aus dieser Fantasie geboren wurden.

Voraussetzungen

Spielentwicklung ist ein weites Feld, bei dem man eine Vielzahl von Talenten und Fähigkeiten einbringen kann. Im Umkehrschluss bedeutet das auch, dass man mit einem einzigen Buch nicht alles lernen kann, was man für die Entwicklung aller möglichen Spiele braucht. Speziell im Bereich 3D-Modelling und Programmierung verweise ich daher auf weiterführende Literatur, falls Sie Ihre Fähigkeiten dazu auffrischen wollen. Die Programmierbeispiele in C# sind vor allem am Anfang so erklärt, dass minimale Programmierkenntnisse in beliebigen Programmiersprachen zum Verständnis ausreichen sollten. Wenn Sie C# oder eine ähnliche Programmiersprache wie Java bereits können ist das natürlich von Vorteil. Kenntnisse in anderen Sprachen wie JavaScript oder Python können Sie in Unity ebenfalls gut einsetzen.

Wie Sie dieses Buch am besten lesen

Dieses Buch soll bewusst nicht als Referenz oder Nachschlagewerk dienen. Dazu gibt es das *Unity Manual*, das automatisch immer auf dem aktuellsten Stand von Unity ist und somit natürlich viel besser mit der schnellen Entwicklung der Game-Engine Schritt halten kann als jedes gedruckte Werk. Vielmehr ist dieses Buch als Reiseführer zu verstehen, der Sie Schritt für Schritt durch die Höhen und Tiefen des Entwicklungsprozesses verschiedener Variationen einer Spielidee führt. Dieser Reiseführer gibt Ihnen an vielen Stellen auch Vorschläge für eigene Entdeckungsreisen bis in die tiefsten Tiefen des Kaninchenbaus: teilweise in Form von Stichworten und Links, die Ihnen gute Einstiegspunkte für eigenen Recherchen bieten …

Übung macht den Meister

… teilweise auch durch anspruchsvollere Übungen, zu deren Umsetzung Sie meistens die im Buch beschriebenen Features und Workflows nutzen können, sich manchmal aber auch selbstständig zusätzliches Wissen aneignen müssen. Sie erkennen die Übungen leicht an der Marginalie *Übung macht den Meister*.

Von Anfang bis Ende

Insofern empfiehlt es sich natürlich auch, das Buch von Anfang bis Ende durchzuarbeiten und lediglich die Abschnitte, deren Inhalt Ihnen bereits vertraut ist, nur kurz zu überfliegen, um zu sehen, ob es da nicht doch noch ein Detail gibt, das Ihnen unbekannt ist.

Nachschlagen

Vielleicht werden Sie manchmal Ihr Wissen zu einem bestimmten Thema schnell mithilfe dieses Buches auffrischen oder einen bestimmten Trick nachlesen wollen. Dazu dient das ausführliche Stichwortverzeichnis. Mein Anliegen beim Schreiben des Buches war es, jeden Begriff und dazu passende Synonyme zu erfassen, die für so eine Suche infrage kommen könnten. Ich hoffe, dass mir dies gelungen ist und dass das Stichwortver-

zeichnis für Sie genau diesen Zweck erfüllt! Falls Sie einmal etwas nicht finden sollten: Lassen Sie mich das bitte wissen!

Die Website zum Buch: unity-buch.de

Die beste Möglichkeit, um sowohl mit Ihren Reisegefährten während der Spielentwicklung als auch mit mir in Kontakt zu treten, ist die Website zum Buch: *unity-buch.de*. Website und Buch gehören zusammen und ergänzen sich: Wenn *Das Unity-Buch* Ihr Reiseführer ist, dann ist *unity-buch.de* Ihr Portal in die Welt, die wir jetzt gemeinsam bereisen. Gleichzeitig spannt die Website ein soziales Netz aller Leser dieses Buches auf, die sich auf diese Weise beteiligen und ihre Reise mit anderen teilen wollen.

Grundsätzlich orientieren sich die verschiedenen Bereiche der Website am Inhaltsverzeichnis des Buches: Sie können also nach der Lektüre eines Kapitels im Buch auf die Website gehen und finden dort leicht alle Ressourcen, die zu diesem Kapitel passen. Zusätzlich bietet die Website aber auch Ansichten anhand verschiedener thematischer Kriterien und erlaubt Ihnen so auch eine ganz individuelle Reiseroute.

Downloads von unity-buch.de

Im Bereich *Downloads* auf *unity-buch.de* finden Sie nicht nur jeden Entwicklungsschritt unserer Beispielprojekte[1], sondern auch Pakete mit rohen Beispielcodes, Bildern, Sounds und 3D-Modellen. Diese können Sie auch zur selbstständigen Umsetzung der Beispielprojekte oder neuen Varianten davon nutzen, falls Sie mangels Zeit oder entsprechend kultivierten Talents diese Dateien nicht selbst erstellen können oder wollen.

Beachten Sie bitte, dass diese Dateien lediglich Ihrem Lernen oder auch als Vorlage für eigene Dateien dienen und in Ihren eigenen Projekten nur nach schriftlicher Genehmigung verwendet werden dürfen.[2] Natürlich können Sie auch für unsere gemeinsamen Beispielprojekte Ihre eigenen Kreationen verwenden. Alle Stellen im Buch, die auf einen Download verweisen, erkennen Sie leicht an dem Text *Download von unity-buch.de* neben dem Haupttext.

Falls Sie eine langsame Internet-Verbindung haben oder viel unterwegs sind: Es gibt auch ein Komplettpaket zum Download mit allen Einzel-Downloads und sämtlichen Screencasts (s. u.). Das können Sie über eine schnelle Internet-Verbindung herunterladen und dann auf Ihren Rechner übertragen.

Screencasts auf unity-buch.de

Manche Dinge lassen sich mit Worten und selbst mit Bildern nur sehr schwer beschreiben. In diesen Fällen habe ich Screencasts erstellt, also kleine Tutorial-Videos, um den Sachverhalt für Sie leicht verständlich zu veranschaulichen. Auch auf diese Screencasts verweise ich an entsprechenden Stellen im Buch, jeweils mit dem Text *Screencast auf unity-buch.de*. Auf

1 Eine nützliche Orientierung, wenn Sie sich mal in Unity verlaufen haben

2 Zumindest, falls Sie vorhaben, diese Projekte auch zu veröffentlichen, und die Dateien nicht so stark verfremden, dass der Ursprung nicht mehr erkennbar ist

unity-buch.de finden Sie im Bereich *Screencasts* alle Videos, die im Buch erwähnt werden, sowie ergänzende Tutorials.

Links auf unity-buch.de

Neben den Dateien und Videos bietet die Website zum Buch auf *unity-buch.de* noch eine Vielzahl weiterführender Links, auf die häufig auch im Buchtext verwiesen wird. Das erkennen Sie auch am Text *Link auf unity-buch.de*. Natürlich müssen Sie nicht sofort alle von der Website aus verlinkten Artikel lesen, und die Artikel, die wiederum von dort aus verlinkt sind, und so weiter – bis Sie das gesamte Internet bereist und verspeist haben. Aber schrittweise können Sie über diese Links Ihren Horizont erweitern, tolle Gegenden im World Wide Web kennenlernen und Ihre Kenntnisse in allen für die Spielentwicklung relevanten Bereichen immer weiter vertiefen. Im Bereich *Links* auf *unity-buch.de* finden Sie alle Links zum Buch: sowohl nach Abschnitten sortiert, sodass Sie nach der Lektüre eines Abschnitts im Buch dessen Inhalte komfortabel auf der Website vertiefen können, als auch nach thematischen Kriterien. Außerdem merkt die Seite sich, welchen Links Sie bereits gefolgt sind[3] – so haben Sie jederzeit den Überblick über Ihren Fortschritt.

Das Fragen-Forum auf unity-buch.de

Link auf unity-buch.de

Im Bereich *Fragen-Forum* finden Sie ein deutsches Forum für Fragen im Stil von *StackOverflow* oder *Unity Answers* (siehe auch die Links auf der Website zum Buch). Bitte zögern Sie nicht, mich über dieses Forum zum Buch zu fragen. Gerne auch, falls Ihnen mal ein Stichwort im Stichwortverzeichnis fehlt: Zum einen kann ich Ihnen wahrscheinlich leicht helfen, die entsprechende Stelle im Buch zu finden; zum anderen gibt mir das auch die Möglichkeit, das Stichwortverzeichnis in der nächsten Auflage entsprechend zu erweitern und auch auf der Website zum Buch für andere Studienreisende in Sachen Spielentwicklung einen entsprechenden Vermerk im Bereich *Errata* zu hinterlassen.

Das Fragen-Forum arbeitet mit sogenannten Tags, wie *2D, 3D, C#, Physik, Unity UI, Asset Store, Stichwortverzeichnis* oder auch *Errata*. Jeder Frage können Sie ein bis fünf solcher Tags zuordnen, und natürlich können Sie über die Tags auch zu diesem Tag passende Fragen suchen und damit vielleicht sehr schnell eine Antwort finden.

Was Sie in diesem Buch lernen: more than just facts

Nach der Lektüre von *Das Unity-Buch* sollten Sie genügend über wesentliche Bereiche der aktuellsten Version von Unity wissen, um Ihre eigenen Spielideen erfolgreich umsetzen zu können. Es wird zwar nicht jedes Feature beschrieben, und vor allem nicht bis ins letzte Detail. Dafür werden Sie aber am Ende dieses Buches einen sehr guten Orientierungssinn entwickelt

3 Vorausgesetzt natürlich, Sie haben sich registriert und eingeloggt

haben: einerseits, um bei Bedarf mühelos bis ans letzte Detail aller Unity-Features zu gelangen, um sie bestmöglich zur Umsetzung Ihrer Spielidee zu nutzen; andererseits, um sich in dem manchmal sehr herausfordernden Prozess der Entwicklung von Spielen zurechtzufinden.

Das ist ein weiterer Grund, aus dem ich Ihnen mit dem *Fragen-Forum* auf der Website zum Buch anbiete, sich mit anderen Entwicklern zu vernetzen: Gerade in den schwierigen Phasen und vor allem am Anfang kann ein Austausch mit Reisegefährten den notwendigen Schub verleihen, um in den Hyperspace durchzubrechen, anstatt in Stagnation zu geraten und am Ende aufzugeben!

Spielentwicklung ist nicht nur ein enorm weites und gleichzeitig sehr tiefes, komplexes und herausforderndes Feld, sondern auch eines, in dem eine unglaublich schnelle Weiterentwicklung geschieht: Unity kenne ich persönlich seit Version 2.0 – das war Ende 2007. Inzwischen sind wir bei Version 5.0, und ganze Bereiche sind neu dazugekommen. Andere sind praktisch nicht mehr wiederzuerkennen, und manche Features sind auch weggefallen. Es macht großen Spaß, über immer modernere und mächtigere Werkzeuge zu verfügen und immer elegantere Workflows zu nutzen, die mit geringerem Aufwand hochwertigere Ergebnisse ermöglichen. Das bedeutet aber auch, dass man praktisch die ganze Zeit über Neues lernen muss, ohne sich dabei mit der technischen Entwicklung zu verzetteln. Genau dazu soll dieses Buch motivieren. Deswegen schreibe ich über einige Themen bzw. Features nur so viel, dass Sie das Feature gut in den Gesamtprozess der Spielentwicklung einordnen und sich die Details dann selbstständig aneignen können. Die Website *unity-buch.de* mit den weiterführenden Links dient als Einstiegspunkt für diese Vertiefung.

Link auf unity-buch.de

Leider sind viele für die Spielentwicklung relevante Artikel nur auf Englisch verfügbar. Das gilt auch für die Dokumentation von Unity. Dieses Buch ist in deutscher Sprache verfasst, daher gebe ich – wo möglich – Verweise auf deutsche Artikel. Es gibt auch bereits seit April 2010 *Unity Insider*, ein aktives deutsches Unity-Netzwerk mit Diskussionsforum.[4] Dort finden Sie auch eine Initiative für ein deutsches Dokumentations-Wiki.[5] Möglicherweise wird es hier später auch von Unity Technologies selbst etwas geben.[6] Microsoft hat praktisch das komplette Microsoft Developer Network (MSDN) und damit auch die Dokumentation von C# und vom. NET Framework auf Deutsch übersetzt[7] – und wie wir noch sehen werden, nutzt das auch uns als Unity-Entwickler.

4 Siehe auch den Link *Unity Insider – Deutschlands führendes Unity Netzwerk* auf der Website zum Buch.

5 Siehe auch den Link *Unity Insider Forum: Deutsche Doku-Wiki* [sic!]

6 Eine Initiative für Spanisch, Portugiesisch und Russisch wurde am 01.12.2014 gestartet, siehe die Link: *Translating Documentation, We Need You!*

7 Daher verlinke ich hier jeweils zuerst auf die deutsche Version, beispielsweise der *C#-Sprachspezifikation*. Für diejenigen, die lieber das englische Original lesen, ist natürlich auch immer der entsprechende Link dabei.

Dennoch lohnt es sich für jeden Spielentwickler, sich auch mit der englischen Sprache anzufreunden. Denn bei der Spielentwicklung machen gute Englischkenntnisse einem das Leben deutlich einfacher. Viele für uns Spielentwickler relevante Ressourcen sind in dieser Sprache verfasst, und je mehr Sie auf Englisch lesen, desto leichter wird es mit der Zeit. Auch hier gilt: *Übung macht den Meister!* Vielleicht reisen Sie ja auch mal in ein englischsprachiges Land.

Wir reisen jetzt erst mal gemeinsam in die wundervolle Welt der Spielentwicklung. Das Fahrzeug, mit dem wir diese Reise antreten, ist Unity – und das nächste Kapitel habe ich einer ersten Vorstellung dieses Fahrzeugs gewidmet. ***Let the game begin!***

Jashan Chittesh, 3. März 2015

Danksagungen

Zunächst ein riesiges Dankeschön an Joachim Ante, Nicholas Francis und David Helgason dafür, dass sie Unity zu einem Produkt entwickelt und der Welt zur Verfügung gestellt haben. Ohne diese drei Gründer von *Unity Technologies* (ursprünglich *OTEE – Over The Edge Entertainment*) würde es Unity und damit dieses Buch gar nicht geben. Dieses Dankeschön erstreckt sich natürlich auch weiter auf alle inzwischen über 500 Mitarbeiter von Unity Technologies – you girls and guys rock!

Dieses Buch würde es auch nicht geben ohne meinen Lektor René Schönfeldt, der den gesamten Prozess von Anfang bis Ende mit tollen Anregungen, geduldig und wohlwollend unterstützt hat. Dafür ein ganz herzliches Dankeschön!

Wertvolles Feedback habe ich auch von den Fachgutachtern erhalten, namentlich seien hier Martin Schulz, Marcus Ross und Julia Schmidt genannt.

Weiterhin möchte ich Friederike Daenecke für die wertvolle Rechtschreibkorrektur und Petra Strauch für den Satz dieses inzwischen doch recht umfangreichen Werkes danken!

Gestalterische Unterstützung erhielt ich von Nicole Delong und Sebastian Weidner. Hervorzuheben ist hier vor allem die neue 2D-Art für Snake sowie das GUI-Design von Nicole und die Profi-Tipps zum Postprocessing des Buchcovers von Sebastian. Danke auch für Eure Freundschaft!

Dankbarkeit aus tiefstem Herzen fühle ich natürlich auch für meine Lebensgefährtin Mirimah, die vor allem in den heißen Schreibphasen geduldig, verständnisvoll und liebevoll unterstützend die Erschaffung dieses Buches begleitet hat.