Das Unity-Buch

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

von
Jashan Chittesh

1. Auflage

dpunkt.verlag 2015

Verlag C.H. Beck im Internet:
www.beck.de
ISBN 978 3 86490 232 1

Zu Inhaltsverzeichnis

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

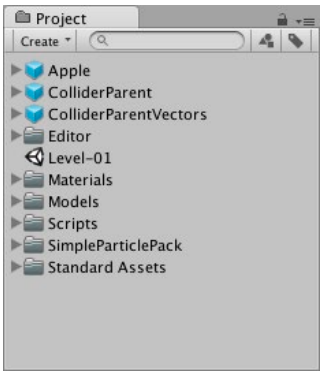

Abb. 7.1
Unsere organisch gewachsene Projektstruktur

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.

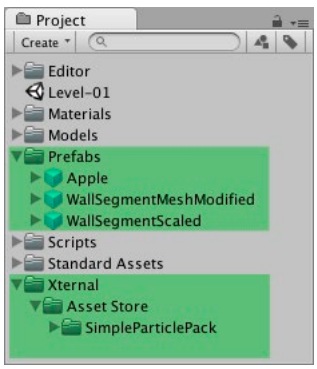

Abb. 7.2
Eine neue, aufgeräumte Projektstruktur

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

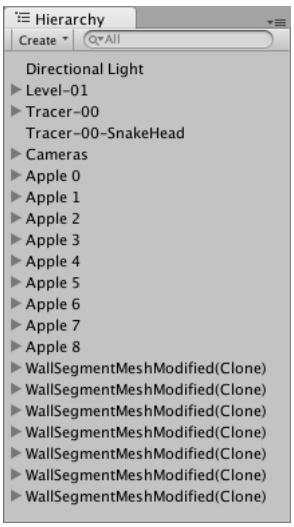

Abb. 7.3
Chaos in der Szenenhierarchie

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.

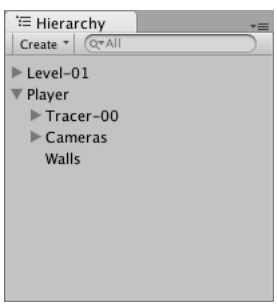

Abb. 7.5
Die Szenenhierarchie aufräumen, Teil 2

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.

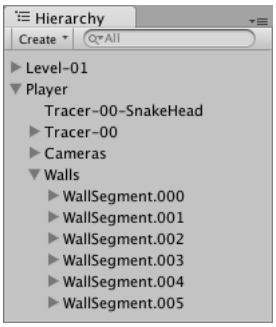

Abb. 7.6
Ein aufgeräumter Spielplatz: unsere Szenenhierarchie

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.

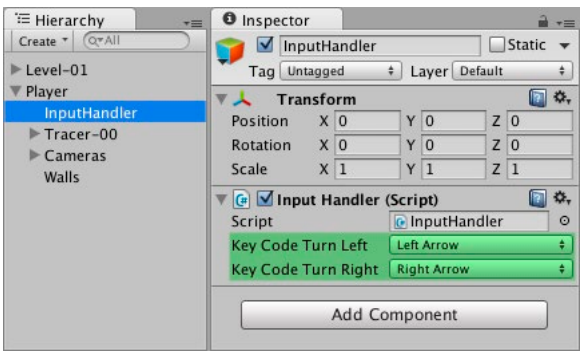

Abb. 7.7
Der InputHandler als eigenes Objekt in der Szenenhierarchie

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 abfragen

```
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`.

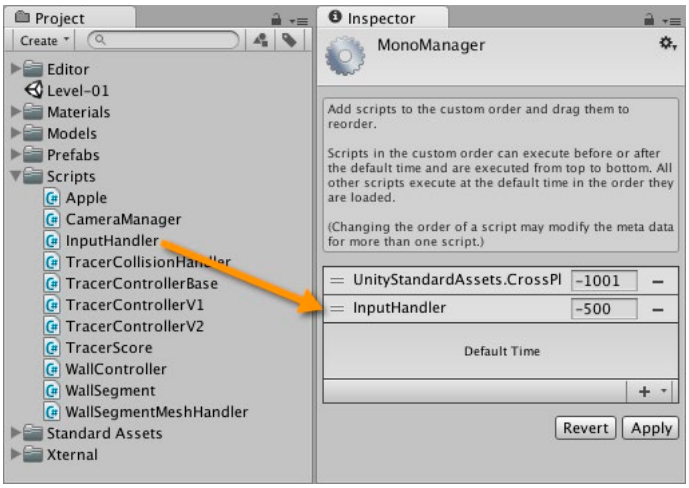

Abb. 7.8
Den InputHandler vor allen anderen Scripts ausführen

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?