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

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

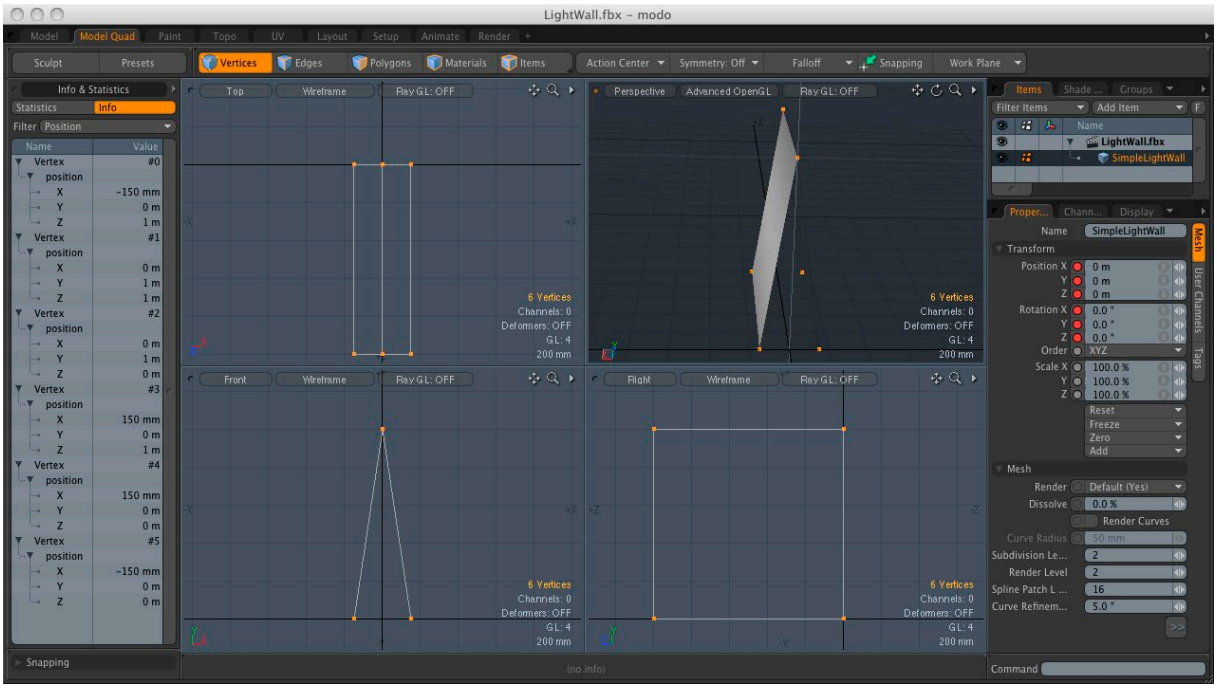


Abb. 6.58
Ein einfaches Modell für die Wände in modo

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.

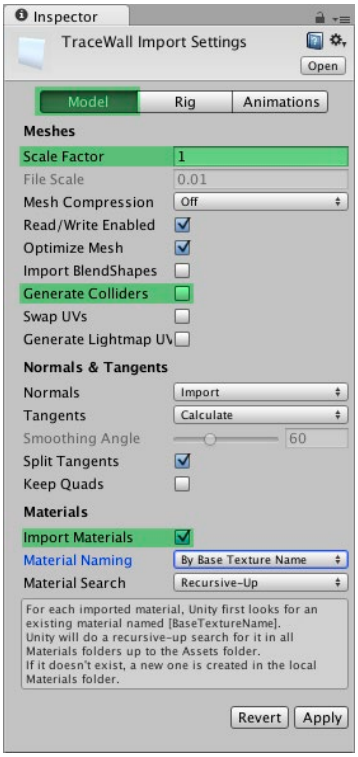


Abb. 6.59
Import-Einstellungen für 3D-Modelle

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.

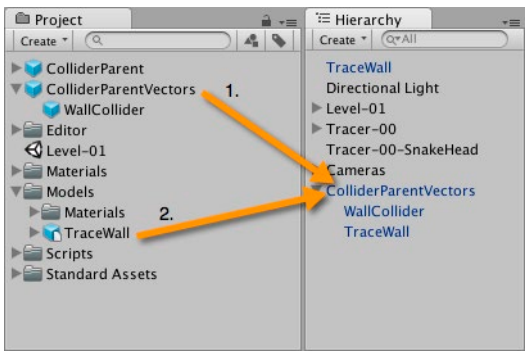
6.2.11 Ein neues Prefab für die Wände erstellen

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



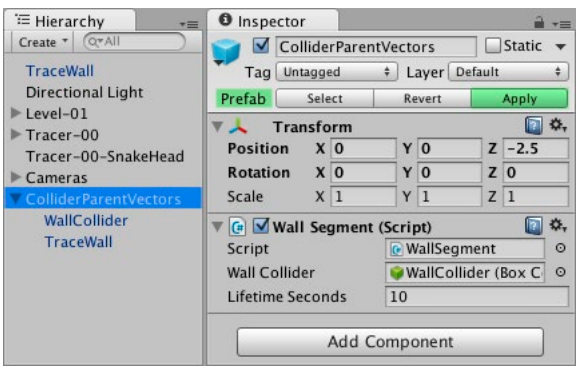
Den `MeshRenderer` des GameObjects `WallCollider` unter `ColliderParentVectors` müssen wir natürlich deaktivieren, damit unser neues, höchst elegantes Wandmodell zum Vorschein kommt.

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.

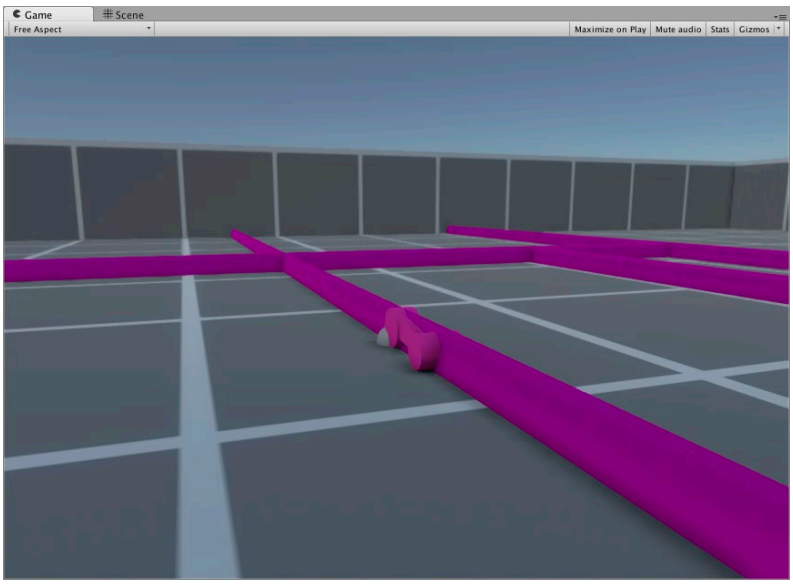


Abb. 6.62
Mit den Wänden ist etwas schiefgelaufen.

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

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.

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`.

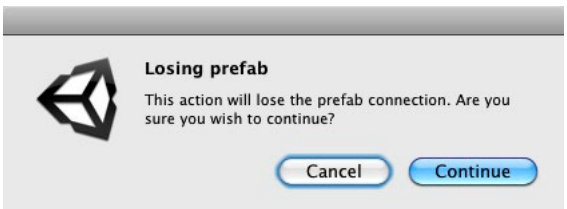


Abb. 6.63
Losing Prefab – das macht aber nichts.

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.

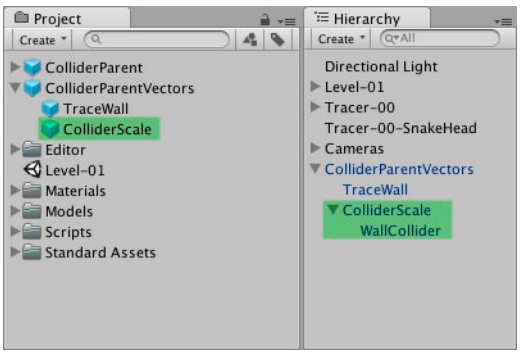


Abb. 6.64
Prefabs im Projekt und ihre Instanzen in der Szene

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.

Listing 6.32
Erweiterung von WallSegmentMeshHandler.LateUpdate()

```
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();
}
```

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.

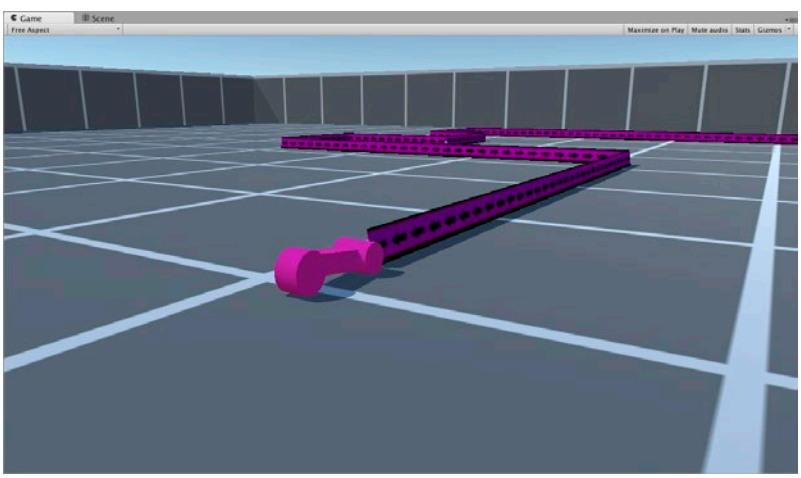


Abb. 6.65
Ganz schön was geschafft!

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