7 Die Virtual-Reality-SDKs

Jede Virtual-Reality-Brille besitzt ihre eigenen Entwicklungswerkzeuge („SDK“ genannt), die für komplexere Projekte trotz der Unity-internen VR-Unterstützung benötigt werden. Diese Entwicklerwerkzeuge erweitern die von Unity mitgelieferten Funktionen um Fähigkeiten und Einstellungen, die nur für die einzelnen Headsets relevant sind. Egal für welche Brille du entwickelst, du verwendest also immer die interne VR-Unterstützung als Basis und kannst dann die jeweiligen SDKs zusätzlich verwenden, um noch mehr Einstellungsmöglichkeiten und Funktionen für die einzelnen Brillen zu haben, mit denen du dein Spiel für die jeweilige VR-Bille optimieren kannst.

Zunächst werden wir uns in diesem Kapitel die Funktionen und Einstellungen der Unity-internen VR-Unterstützung anschauen. Danach gehe ich auf die einzelnen SDKs und ihre Besonderheiten ein. Dieses Kapitel soll dir auch als Nachschlagewerk dienen, wenn du später einmal Fragen zu den einzelnen SDKs hast.

Bei den Entwicklungswerkzeugen, die für Unity meist in Form eines *Unity Packages* angeboten werden, sind in der Regel die darin enthaltenen *Prefabs* mit am wichtigsten. Sie demonstrieren dir nämlich, wie du die einzelnen Skripte aus dem Package verwenden und konfigurieren solltest. Aus diesem Grund werden wir uns die Prefabs im Detail ansehen und dabei auch etwas über die dazugehörigen Skripte erfahren. Ich werde dir in dem Zusammenhang nämlich auch alle relevanten Components der SDKs vorstellen und erklären, welche Funktionen und Einstellungen sie dir bieten. Bei den Eigenschaften der Components gibt es zwei Kategorien: *Inspector-Eigenschaften*, welche dir im Unity Inspector angezeigt werden, und *Code-Eigenschaften*, auf die du, wie auf Methoden, nur über Skripte zugreifen kannst. Bei den *Code-Eigenschaften* werde ich mich auf die wichtigsten beschränken, für eine vollständige Liste kannst du in die Skripte der jeweiligen Components oder in die offizielle Dokumentation hineinschauen.

7.1 Unity-interne VR-Unterstützung

Die in Unity integrierte VR-Unterstützung bietet dir die Möglichkeit, einige Funktionen, die alle unterstützten Headsets gemeinsam haben, zentral über bestimmte Unity-Klassen zu steuern. Dadurch wird dein Code unabhängig von der VR-Brille, die verwendet wird. Das ist sehr hilfreich, wenn du eine Anwendung schreibst, die später auf mehr als nur einer VR-Brille laufen soll. Außerdem ist dieser Ansatz auch für dich als Anfänger hilfreich: Dadurch, dass die Funktionen vereinheitlicht wurden, musst du bei Problemen nicht immer eine Lösung für dein spezielles Headset finden, sondern es reicht häufig, wenn du eine Lösung für *Unitys VR-Unterstützung* findest.

7.1.1 Unitys interne VR-Unterstützung aktivieren

Wie du Unitys interne VR-Unterstützung aktivierst, hast du bereits in dem Quickstart-Kapitel gelernt und sogar selber angewendet. Hier schauen wir uns die Funktion aber noch ein wenig genauer an.

Um die VR-Unterstützung zu aktivieren, musst du zuerst die *Player Settings* (**Edit/Project Settings/Player**) öffnen. Du solltest dann im Inspector unter anderem das Menü aus Bild 7.1 sehen. Hier musst du darauf achten, dass der korrekte Reiter für deine Zielplattform ausgewählt ist. In Bild 7.1 ist zum Beispiel der *PC*-Reiter aktiv. Wenn du ein Android-Spiel entwickelst, muss der *Android*-Reiter rechts aktiv sein.

Die Virtual Reality-Unterstützung aktivierst du anschließend, indem du, in der Kategorie **Other Settings**, die Checkbox **Virtual Reality Supported** aktivierst.

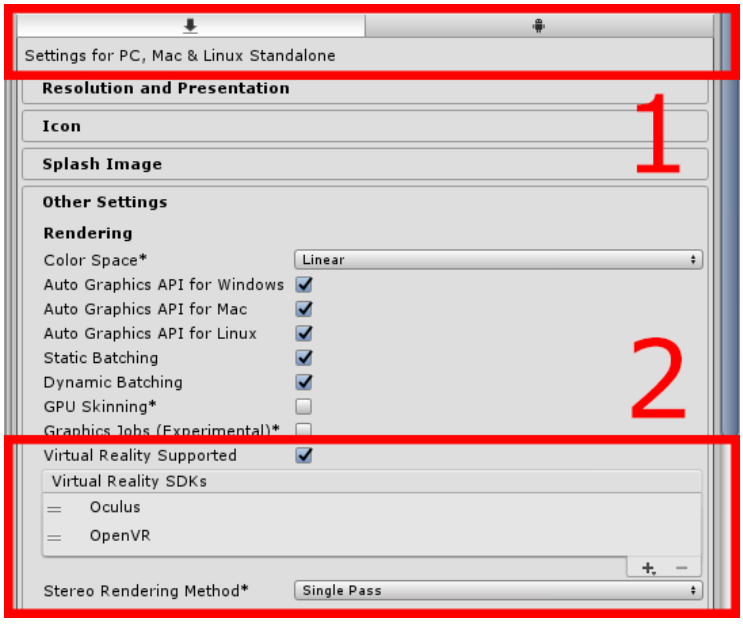

Bild 7.1 Die Player Settings. 1) Zielplattform-Reiter, 2) VR-Einstellungen

Durch das Aktivieren der Checkbox erscheinen mehrere VR-spezifische Optionen, die du in Bild 7.1 unten siehst. Als Erstes findest du dort die Liste der Virtual Reality SDKs, die du in deinem Spiel verwenden möchtest. Enthält die Liste, wie in Bild 7.1, mehrere Einträge, versucht das Spiel automatisch zu erkennen, welche VR-Brille verwendet wird, und aktiviert dann automatisch das passende SDK. Die Liste wird von oben nach unten abgearbeitet, das bedeutet, es wird bei der Einstellung in Bild 7.1 zuerst geprüft, ob eine *Oculus SDK*-kompatible Brille angeschlossen ist. Wird keine kompatible Brille gefunden, wird überprüft, ob eine kompatible Brille für den nächsten Eintrag, hier *OpenVR*, gefunden wird.

OpenVR immer als letzten Eintrag

Wenn eine Oculus Rift angeschlossen ist und „OpenVR“ an erster Stelle vor dem „Oculus“-SDK steht, würde automatisch das OpenVR-SDK verwendet werden, da die Rift auch von dem OpenVR-SDK unterstützt wird. „OpenVR“ sollte deswegen immer unter „Oculus“ in der Liste stehen, wenn du das „Oculus SDK“ für die Rift verwenden möchtest.

Bei der Überprüfung für das OpenVR-SDK startet die SteamVR-Software automatisch, auch wenn keine passende VR-Brille angeschlossen ist.

Im Zweifelsfall nur ein SDK angeben

Zwar funktioniert die automatische Erkennung in der Regel ohne Probleme, jedoch muss teilweise auch der Rest deines Spiels für das jeweilige SDK angepasst werden; zum Beispiel, wenn du die SDK-Packages verwendest, die wir nachfolgend besprechen. Für den Anfang ist es deswegen häufig sinnvoll, erst nur ein SDK in der Liste zu aktivieren und das Spiel zunächst nur für diese Brille zu entwickeln.

Ist diese VR-Option aktiviert und es wird eine VR-Brille an deinem PC erkannt, wird jedes *Camera*-Component in deiner Scene automatisch zu einer *Camera*, die ein stereoskopisches Bild für deine VR-Brille berechnet. Außerdem setzt die Camera bereits automatisch die Head-Tracking-Daten um, das bedeutet, sowohl Drehungen als auch Bewegungen[1] deiner VR-Brille werden automatisch auf die Camera übertragen.

Über die Option *Stereo Rendering Mode* kannst du zwischen dem Standard *Multi Pass* und dem optimierten *Single Pass*-Modus wählen. Der Single-Pass-Modus wurde entwickelt, um Performance zu sparen, indem er versucht, die Elemente für das linke und das rechte Auge in einem einzigen *SetPass*-Call zu zeichnen anstelle von zwei oder mehr. Diese Funktion soll verhindern, dass sich die *SetPass*-Calls und somit die benötigte Rechenpower verdoppeln, wenn man ein Bild in stereoskopischem 3D berechnet, wie es für VR-Brillen notwendig ist. Diese Option solltest du immer aktivieren, um Rechenleistung zu sparen.

[1] Nur wenn deine VR-Brille auch Positional-Tracking unterstützt

7.1.2 Interne VR-Unterstützung in der Scripting-API

Zusätzlich zu den Optionen im Unity Editor stehen dir auch noch einige Optionen und Einstellungen zur Verfügung, die du nur mittels eines Skripts steuern kannst. Alle VR-bezogenen Klassen für die integrierte VR-Unterstützung findest du in dem *Namespace* `UnityEngine.VR`.

Die wichtigsten dieser Klassen, Methoden und Variablen werde ich dir jetzt vorstellen. Ich werde dabei nur auf die Elemente eingehen, die man häufiger in einem Spiel verwendet. Alles, mit dem du wahrscheinlich nie selbst in Berührung kommen wirst, überspringe ich an dieser Stelle. Für einen umfassenden Einblick kannst du dir alle Klassen des Namespaces in der Unity-Dokumentation ansehen.[2]

Die Variablen und Methoden für die VR-Unterstützung sind in der Regel *statisch*. Das bedeutet, dass sie direkt über den Klassennamen aufgerufen werden müssen; zum Beispiel: `UnityEngine.VR.InputTracking.Recenter();`

7.1.3 InputTracking

Die *InputTracking*-Klasse enthält *statische* Methoden, die sich mit dem Positions- und Rotations-Tracking beschäftigen.

- **Recenter()** : `void`

  Diese Methode ist wahrscheinlich die wichtigste dieser Klasse. Durch die `Recenter`-Methode werden alle Tracking-Daten zurückgesetzt und auf diese Weise die aktuelle Position und Rotation als neue Ausgangsposition festgelegt. Diese Methode ist vor allem für Spiele, die im Sitzen gespielt werden, wichtig. Für gewöhnlich bittet man den Spieler, sich gemütlich hinzusetzen und eine bestimmte Taste zu drücken. Bei diesem Tastendruck wird die Recenter-Methode aufgerufen und seine aktuelle Sitzposition als Standardposition festgelegt.

  Beispiel: `UnityEngine.VR.InputTracking.Recenter ();`

- **disablePositionalTracking** : `bool`

  Mit dieser Variablen kannst du das Positional-Tracking deaktivieren, wenn es nötig ist. Wie im „Dos and Don'ts"-Kapitel beschrieben, ist das jedoch nicht empfehlenswert. Wenn deine Brille Positional-Tracking unterstützt, solltest du es unbedingt auch verwenden.

  Neben den hier vorgestellten Elementen existieren auch Methoden, um die aktuelle Position eines getrackten Elements über ein Skript auszulesen. Standardmäßig übernehmen das aber die Skripte aus den SDK-Packages für dich, sodass du diese Methoden in der Regel selbst nicht benötigst.

[2] https://docs.unity3d.com/ScriptReference/VR.InputTracking.html

7.1.4 VRDevice

Über die *VRDevice*-Klasse kannst du in deinem Skript Informationen über die VR-Brille erhalten, die der Spieler derzeit verwendet. Das kann zum Beispiel hilfreich sein, wenn du dein Spiel für bestimmte VR-Brillen besonders anpassen musst, damit es perfekt funktioniert.

Die Klasse enthält diese **statischen Variablen**:

- **isPresent** : `bool`

  Diese Variable gibt an, ob eine VR-Brille erfolgreich erkannt wurde und sofort verwendet werden kann.

- **model** : `string`

  Diese Variable enthält den Namen der verwendeten VR-Brille. Hierüber kannst du auch zwei Brillen unterscheiden, die dasselbe SDK verwenden.

- **refreshRate** : `float`

  Über diese Variable kannst du erfahren, mit welcher Bildwiederholungsrate die derzeit angeschlossene VR-Brille arbeitet. (Zum Beispiel 90, bei Oculus und HTC Vive oder 60 bei GearVR und Daydream)

Zudem enthält sie diese **statischen Methoden:**

- **DisableAutoVRCameraTracking (** *camera* : `Camera`**,** *disabled* : `bool`**)** : `void`

  Wie erwähnt werden standardmäßig alle *Cameras*, die sich in deiner Scene befinden, in einen VR-Modus umgeschaltet, wenn du das Spiel mit einer VR-Brille startest. Über diese Methode kannst du diesen VR-Modus für einzelne *Cameras* deaktivieren. Das kann zum Beispiel hilfreich sein, wenn du eine Überwachsungskamera in deinem Spiel integrieren möchtest, deren Bild auf einem Monitor im Spiel angezeigt werden soll, anstelle es direkt an das Headset zu schicken.

- **GetTrackingSpaceType ( )** : `TrackingSpaceType`

  Mit dieser Methode kannst du herausfinden, ob die VR-Brille deines Spielers derzeit für *Room-Scale* oder *Stationär* (stehend/sitzend) konfiguriert wird. Das kannst du zum Beispiel nutzen, um deine Spielmechaniken automatisch entsprechend anzupassen.

  Die Methode gibt dir entweder den Wert `TrackingSpaceType.Stationary` oder `TrackingSpaceType.RoomScale` zurück.

- **SetTrackingSpaceType (** *trackingSpaceType* : `TrackingSpaceType`**)** : `bool`

  Mit dieser Variablen kannst du die *Tracking Space*-Art deines Spielers für dein Spiel ändern. Das ist dann hilfreich, wenn dein Spiel nur eine Art von Tracking Space unterstützt. Bei Renn- und Cockpit-Spielen ist es zum Beispiel sinnvoll, die *Tracking Space*-Art auf *stationär* umzustellen, damit die Kamera an der korrekten Position platziert werden kann. Der Rückgabewert verrät dir, ob das Umstellen erfolgreich war. Unterstützt die verwendete VR-Brille kein Room-Scale, schlägt der Aufruf beispielsweise fehl, wenn du versuchst, den Room-Scale-Modus zu aktivieren.

In Listing 7.1 findest du ein kurzes Beispiel für das Umstellen der *Tracking Space*-Art, wie man es in einem Cockpit-Spiel verwenden könnte:

Listing 7.1 So könnte eine Recenter-Methode in einem Cockpit-Spiel aussehen. Steht der Tracking Space derzeit auf Room-Scale, wird er automatisch auf stationär umgestellt.

```
private void Recenter() // Setzt die Position und "Nach vorne"-Richtung zurück.
{
    if (VRDevice.GetTrackingSpaceType() != TrackingSpaceType.Stationary) {
        VRDevice.SetTrackingSpaceType(TrackingSpaceType.Stationary);
    }
    InputTracking.Recenter();
}
```

7.1.5 VRSettings

Diese Klasse enthält diverse Einstellungen für Unitys VR-Unterstützung. Du kannst hier die VR-Unterstützung unabhängig von den Einstellungen in den *Player Settings* ein- und ausschalten und auch gezielt ein bestimmtes SDK laden.

- **enabled** : `bool`

  Diese Variable aktiviert oder deaktiviert die Virtual-Reality-Unterstützung. Sie ist identisch mit der Checkbox *Virtual Reality Supported* in den Player Settings. Du kannst sie zum Beispiel verwenden, um den VR-Modus, während dein Spiel läuft, zu deaktivieren.

- **isDeviceActive** : `bool`

  Über diese Variable kannst du erfahren, ob derzeit irgendeine VR-Brille angeschlossen und verwendet wird. Ist dieser Wert `false`, obwohl das korrekte SDK geladen und die Brille angeschlossen ist, gibt es vielleicht ein Treiber-Problem.

- **loadedDeviceName** : `string`

  Über diese Variable kannst du den Namen des derzeit aktiven *SDK* herausfinden. Das ist zum Beispiel hilfreich, wenn du mehrere SDKs in den Player Settings angegeben hast und herausfinden möchtest, welches nun verwendet wird. Um den Namen der jeweiligen VR-Brille und nicht des SDK herauszufinden, musst du `VRDevice.model` verwenden. Über diese Variable kann das aktivierte SDK nicht geändert werden, dafür musst du die Methode *VRSettings.LoadDeviceByName* verwenden.

- **renderScale** : `float`

  Mit dieser Variablen kannst du die Auflösung verändern, mit der die Frames für die VR-Brille berechnet werden. Die native Auflösung des Headsets wird mit diesem Faktor multipliziert, um die Auflösung für die Berechnung der Frames zu erhalten. Durch diese Technik erhältst du ein sichtbar schärferes Bild auf Kosten der Performance deines Spiels. Wenn du den Wert dieser Variablen änderst, kann es zu einem kurzen Ruckler kommen. Der Ruckler entsteht dadurch, dass in deinem Speicher Platz für die Frames mit der neuen Auflösung geschaffen wird. In Kapitel 7.1.5.1 findest du noch einmal mehr Details zum *RenderScale*.

- **renderViewportScale** : `float`

  Diese Variable erlaubt es dir, den *renderScale* ohne den erwähnten Ruckler zu *verringern*. Mit einem Wert zwischen 0 und 1 kannst du also die Auswirkung der `renderScale`-Vari-

ablen anpassen. Ist dieser Wert 0, wird die `renderScale`-Variable nicht verwendet. Ist dieser Wert 1, wird der über die `renderScale`-Variable angegebene Wert verwendet. Bei Zwischenwerten wird ein Wert zwischen 1 und dem `renderScale`-Wert verwendet. Bei dieser Option wird, im Gegensatz zu *VRSettings.renderScale*, derselbe Speicherbereich weiterverwendet, weshalb kein Ruckler entsteht. Diese Variable sollte deshalb allerdings auch nur für vorübergehende Änderungen am *Render Scale* verwendet werden.

- **showDeviceView** : `bool`

  Mit dieser Variablen kannst du bestimmen, ob auf dem normalen Monitor des Computers (wenn vorhanden) die Sicht des Spielers angezeigt werden soll. Wenn du diese Option deaktivierst, wird auf dem Monitor nur ein schwarzes Fenster angezeigt. Zusätzlich kannst du dann über die *TargetEye*-Eigenschaft eines *Camera*-Components festlegen, dass die Sicht dieser *Camera* auf dem Monitor angezeigt werden soll.

- **supportedDevices** : `string[]`

  Diese Variable enthält einen String-Array (*string[]*) mit allen Virtual Reality SDKs, die in deinem Spiel enthalten sind. Die Liste ist identisch mit der Liste, die du in den *Player Settings* angelegt hast.

Außerdem enthält diese Klasse noch eine **statische Methode**:

- **LoadDeviceByName (** *sdkName* : `string`**)**

  LoadDeviceByName(*sdkNamensListe* : `string[]`)

  Diese Methode erlaubt es dir, ein bestimmtes SDK über ein Skript zu aktivieren. Das funktioniert allerdings nur mit SDKs, die du auch in den *Player Settings* angegeben hast bzw. die in der Variablen *VRSettings.supportedDevices* enthalten sind. Alternativ kannst du der Methode auch eine *Liste* (String-Array) von SDK-Namen übergeben, die dann, wie bei der automatischen Auswahl beim Spielstart, von oben nach unten durchlaufen wird, bis eine passende VR-Brille gefunden wird.

7.1.5.1 Render Scale

Die Auflösung der VR-Brille ist abhängig von der Hardware und kann nicht über die Software verändert werden. Es ist jedoch möglich, die Auflösung zu verändern, mit der die Grafikkarte die Bilder (*Frames*) für die Brille berechnet. Die einzelnen Frames werden durch die höhere Auflösung mit mehr Details berechnet und im Speicher abgelegt. Über verschiedene Algorithmen wird das hochaufgelöste Bild anschließend auf die native Auflösung der VR-Brille heruntergerechnet. Effektiv entsteht dadurch ein schärferes Bild mit schönen, nicht stufigen Kanten und mehr sichtbaren Details. Diese Berechnung erfordert jedoch auch deutlich mehr Rechenleistung.

Andersherum ist es auch möglich, die Auflösung für die berechneten Frames geringer zu wählen als die native Auflösung der Brille. Dadurch wird das Bild unschärfer, allerdings spart man auch einiges an Rechenaufwand.

Bei der Erhöhung des Render Scales solltest du vorsichtig vorgehen und es als optionale Funktion in dein Spiel integrieren, da es schnell zu starken Einbußen bei der Performance deines Spiels kommen kann. Hat die VR-Brille eine native Auflösung von 2160 x 1200 Pixeln, wird mit einem *Render Scale* von beispielsweise *1.5* jeder Frame mit einer Auflösung von 3240 x 1800 Pixeln berechnet, und das, je nach Brille, 90 Mal pro Sekunde. Der per-

fekte *Render Scale*-Wert hängt sowohl von deinem Spiel als auch von deinem Level und am meisten von der Hardware, die der Spieler verwendet, ab.

Da das Spiel ausschließlich nicht nur auf deinem Computer oder Smartphone laufen soll, sondern auch auf der ggf. schwächeren Hardware deiner Spieler, solltest du diesen Wert am besten über ein Menü einstellbar machen oder von Anfang an lieber etwas geringer wählen. 1.2 ist in der Regel ein guter Wert und Kompromiss zwischen benötigter Rechenleistung und Optik.

Wenn du den *Render Scale* über die Variable `VRSettings.renderScale` änderst, wird der Bereich im Grafikspeicher, wo die Frames zwischengespeichert werden, für die Frames mit der neuen Auflösung angepasst. Das kann in deinem Spiel zu einem kurzen Ruckler führen, weshalb du den *Render Scale* über diese Variable am besten nur einmalig beim Laden einer Scene oder in einem Menü ändern solltest.

Über die Variable `VRSettings.renderViewportScale` kannst du den von dir angegebenen `VRSettings.renderScale` vorübergehend korrigieren oder außer Kraft setzen. Hier wird der verwendete Speicherbereich nicht angepasst, wodurch keine Ruckler beim Ändern dieses Wertes entstehen.

7.2 Oculus Integration for Unity (Rift und Gear)

Die Entwicklungswerkzeuge von Oculus benötigst du, wenn du ein Spiel für die *Oculus Rift* oder die *Samsung GearVR* entwickeln möchtest. Die Components der Integration sind so geschrieben, dass sie automatisch erkennen, ob du gerade in einem Oculus Rift- oder GearVR-Projekt arbeitest. Das von Oculus angebotene *Unity Package* wird relativ selten upgedatet, da das Oculus SDK so geschrieben ist, dass die meiste Logik in der *Runtime* (also *Oculus Home bzw. GearVR Service*) liegt und nicht in den Spielen selbst.

Wenn Oculus ein Update für ihre Software rausbringt, musst du also in den meisten Fällen dein Spiel nicht updaten, da die Integration weiterhin kompatibel ist. Diese Vorgehensweise erlaubt es Oculus, dass sie häufiger Updates für die Software herausbringen können, ohne dass bereits existierende Spiel inkompatibel werden und die Entwickler sie ständig updaten müssen.

7.2.1 Download und Import

In dem Kasten unten findest du den Download-Link zu der *Oculus Unity Integration*-Version, die wir in diesem Buch besprechen werden. Außerdem findest du dort auch einen Link zu der *Oculus Unity Integration* im *Unity Asset Store*, wo du immer die neuste Version herunterladen kannst. Als dritte Möglichkeit findest du in dem Kasten auch die *Oculus-Entwickler-Webseite*, wo du zusätzlich auch noch Beta-, also Testversionen herunterladen kannst. Diese Versionen können sich jedoch eventuell von der hier besprochenen Version unterscheiden.

Oculus Integration

Die im Buch beschriebene Version findest du auf der begleitenden Webseite (Version: 1.18.1):

http://www.VRSpieleEntwickeln.de/zusatz/

Unter *Alle SDKs*, dann *OculusUtilities_1.18.1.unitypackage*

Asset Store:

https://www.assetstore.unity3d.com/en/#!/content/82022

Oculus-Entwickler-Downloads:

https://developer.oculus.com/downloads/

Wenn du das Package von der offiziellen Oculus-Seite heruntergeladen hast, musst du das heruntergeladene Zip-Archiv zunächst entpacken. In dem Ordner *OculusUtilities* findest du dann das *Unity Package*. Von der begleitenden Webseite lädst du direkt das Package und musst es nicht entpacken.

Als letzten Schritt kannst du jetzt das Package in dein Projekt importieren. Nachdem der Import-Vorgang abgeschlossen ist, solltest du in deinem Project Browser, wie in Bild 7.2, einen *OVR*-Ordner mit diversen Unterordnern und einen *Plugins*-Ordner sehen.

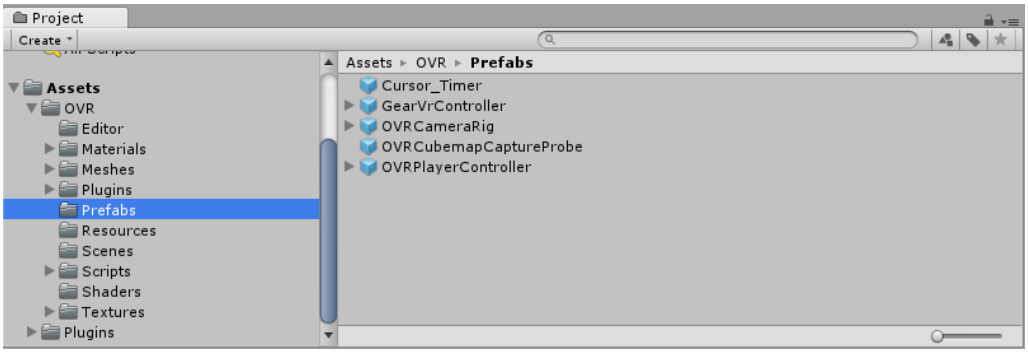

Bild 7.2 Nach dem Import solltest du in deinem *Project Browser* einen „OVR"-Ordner sehen.

Unter Umständen erscheint beim Importieren des *Oculus Utilities Packages* eine Meldung, die dich darüber informiert, dass eine neue Version des *OVRPlugins* verfügbar ist, und dich fragt, ob du diese Version aktivieren möchtest (siehe Bild 7.3). Wenn dies passiert, bestätige die Meldung mit **YES**.

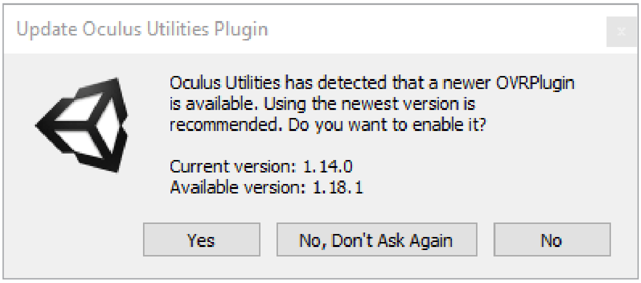

Bild 7.3 Wenn diese Meldung angezeigt wird, bestätige sie mit „Yes". Die Versionsnummern weichen ggf. ab.

Unity wird dann das interne Oculus-Plug-in updaten und anschließend dich dazu auffordern, den Unity Editor neu zu starten. Bestätige das Fenster dann mit **Restart** und warte den Neustart des Unity Editors ab.

7.2.1.1 Fehlermeldung in Console: „Multiple plugins…"

Nach dem Neustart kann es vorkommen, dass in der *Console* folgende Fehlermeldung erscheint: *Multiple plugins with the same name 'ovrplugin'…*, die du auch in Bild 7.4 siehst. Um die Ursache für die Fehlermeldung zu beseitigen, musst du im *Project Browser* den *OVR/Plugins* -Ordner löschen. (*Nicht* den „Plugins"-Ordner, der sich direkt in dem „Assets" Ordner befindet!) In Bild 7.4 ist der zu löschende Ordner nochmals im *Project Browser* markiert.

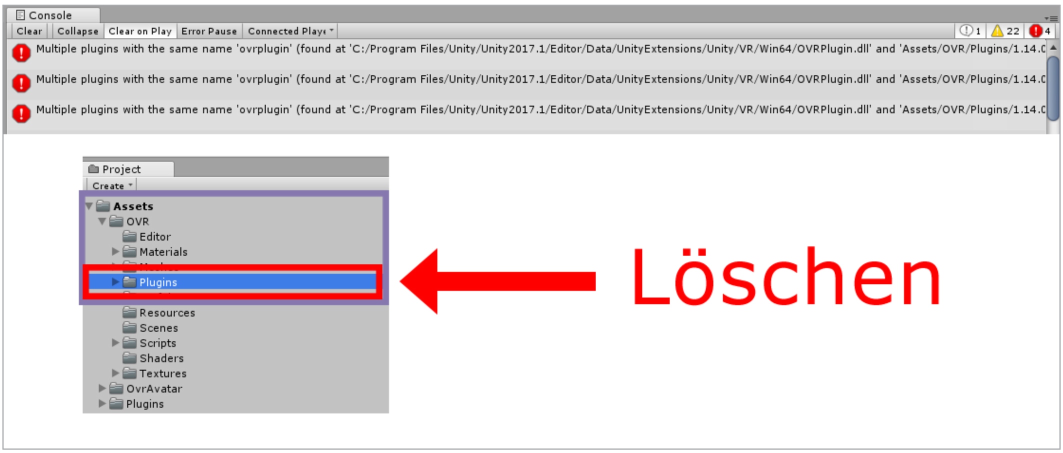

Bild 7.4 Wenn du die Fehlermeldung erhältst, musst du den „OVR/Plugins"-Ordner löschen.

7.2.2 Prefabs

In dem *OVR/Prefabs*-Ordner der *Oculus Integration* findest du fünf *Prefabs*. Für deine Projekte sind jedoch zuerst nur drei davon interessant, weshalb wir uns diese im Detail ansehen werden.

7.2.2.1 OVRCameraRig-Prefab

Das OVRCameraRig-Prefab ersetzt in Oculus Rift und GearVR-Projekten, die dieses SDK verwenden, die gewöhnliche *Camera* in der Scene. Das bedeutet, du musst die standardmäßig angelegte *Main Camera* aus der Scene löschen und ziehst dieses Prefab hinein.

In der Scene bzw. *Hierarchy* wird dann ein GameObject mit dem in Bild 7.5 dargestellten Aufbau angelegt.

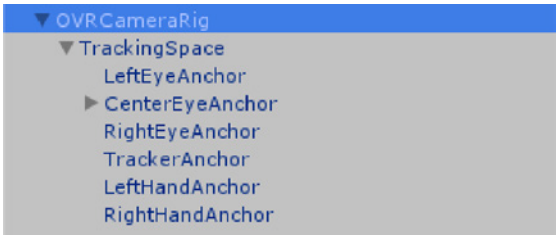

Bild 7.5 Der Aufbau des OVRCameraRigs in der Hierarchy

Alle Bewegungen, die du normalerweise mit dem *Main Camera*-GameObject gemacht hättest, musst du jetzt mit dem *OVRCameraRig* machen, nicht mit einem der Kind-Objekte.

Innerhalb des „TrackingSpace"-GameObjects, welches das Zentrum des realen Spielbereiches des Spielers darstellt, findest du sechs *...Anchor*-GameObjects. Jedes dieser GameObjects gehört zu einem getrackten Punkt deiner VR-Brille bzw. ihrer Controller. Zur Laufzeit werden die GameObjects, basierend auf den Tracking-Daten der Hardware, automatisch bewegt, sodass ihre Position und Rotation im *TrackingSpace* mit denen deiner Brille und den Controllern übereinstimmt.

- **LeftEyeAnchor , RightEyeAnchor:** Diese GameObjects werden an der Position des linken bzw. rechten Auges des Spielers positioniert. An diesen GameObjects findest du jeweils eine deaktivierte *Camera*. Diese *Cameras* werden standardmäßig nicht verwendet, können aber *anstelle* der *Camera* am *CenterEyeAnchor* aktiviert werden. Siehe 7.2.2.1.1 für mehr Infos zu.
- **CenterEyeAnchor:** Dieses GameObject wird mittig zwischen dem linken und rechten Auge des Spielers positioniert. An diesem GameObject findest du eine aktivierte *Camera*, welche standardmäßig für die Darstellung der Spielerperspektive verwendet wird.
- **TrackerAnchor:** Dieses GameObject wird automatisch an der Position einer deiner Oculus-Sensoren platziert.
- **LeftHandAnchor , RightHandAnchor:** Diese beiden Anchor werden an der Position des linken bzw. rechten *Oculus Touch*-Controllers positioniert, wenn sie vorhanden sind. Wenn ein GameObject der Hand des Spielers folgen soll, kannst du es beispielsweise als Kind dieser GameObjects setzen.

An dem *OVRCameraRig*-GameObject findest du zwei wichtige Components, nämlich das *OVRCameraRig*- und das *OVR Manager*-Component, mit denen du diverse Einstellungen bezüglich der Darstellung in der VR-Brille vornehmen kannst.

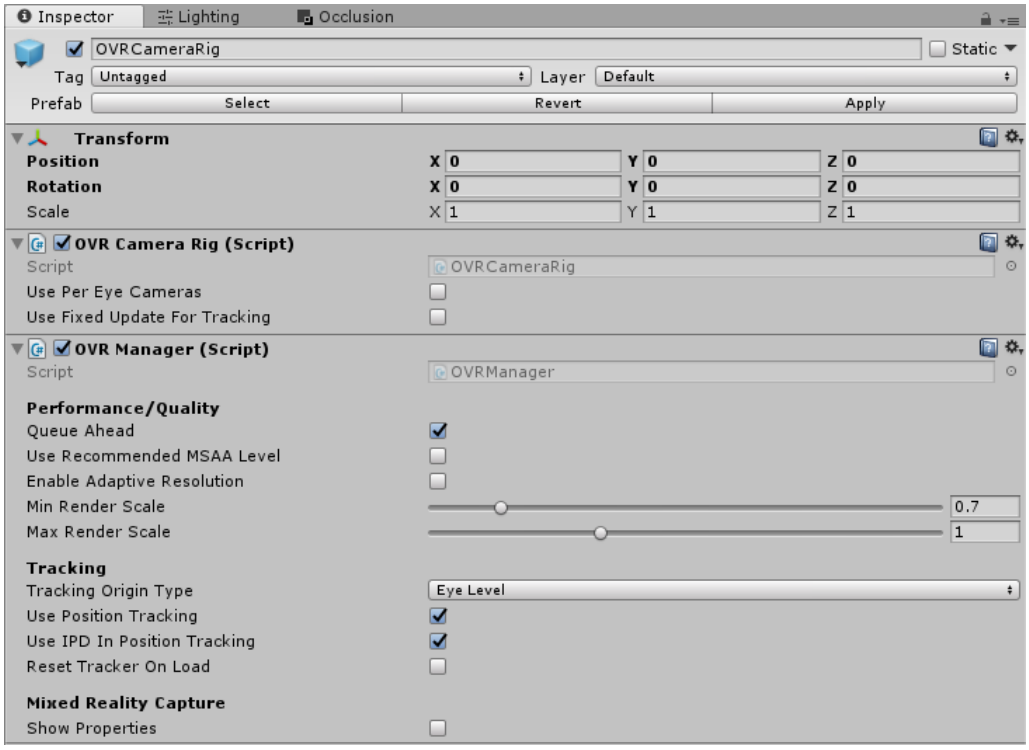

Bild 7.6 Die Components des OVRCameraRigs

Nachfolgend stelle ich dir die einzelnen Components *OVRCameraRigs*-Prefabs vor.

OVR Camera Rig-Component

Dieses Component aktualisiert die Position der einzelnen *...Anchor*-GameObjects.

- **Use per Eye Cameras:** Diese Option deaktiviert automatisch die *Camera* am *CenterEye Anchor* und nutzt die *Cameras* am *LeftEyeAnchor* und *RightEyeAnchor* an ihrer Stelle. Diese Option ist nur für besonders komplexe Spielmechaniken notwendig, bei denen die beiden *Cameras* beispielsweise unterschiedlich konfiguriert sein müssen.
- **Use Fixed Update for Tracking**: Wenn aktiviert, werden die einzelnen Anchors nicht in der *Update*-Methode, sondern in der *FixedUpdate*-Methode bewegt. Dies kann helfen, um Probleme bei der Interaktion mit physikalischen Gegenständen zu verhindern.

OVR Manager-Component

Dieses Component erlaubt es dir, einige Einstellungen bezüglich der Darstellung und des Trackings deiner Oculus Rift oder GearVR vorzunehmen. Die Standardeinstellungen sind für die meisten Projekte bereits in Ordnung, manchmal ist es aber sinnvoll, hier Änderungen vorzunehmen. Ich werde dir hier die wichtigsten Eigenschaften vorstellen, in diesem Component sind aber auch alle Eigenschaften mit Tooltipps versehen, die dir ebenfalls weiterhelfen können.

- **Use Recommended MSAA Level:** Wenn aktiviert, verwendet das Spiel automatisch die Anti-Aliasing-Einstellung, die für die VR-Brille und Hardware (Smartphone, PC) des Spielers von Oculus empfohlen werden.
- **Enable Adaptive Resolution:** Wenn aktiviert, passt das Oculus SDK den *Render Scale* dynamisch zwischen dem angegebenen *Min-* und *Max-* Wert an, während das Spiel läuft. Droht die Performance einzubrechen, wird der *Render Scale* reduziert und wieder erhöht, wenn wieder mehr Leistung zur Verfügung steht. Ähnlich wie bei einem gestreamten Video kann es jedoch sein, dass dem Spieler der Wechsel zwischen den Qualitätsstufen negativ auffällt.
- **Tracking Origin Type:** Hiermit bestimmst du, ob du das „OVRCameraRig" in deiner Scene auf Boden- oder Augenhöhe positioniert hast. In den meisten Fällen wirst du für sitzende Spiele *Eye Level* und für Room-Scale-Spiele *Floor Level* verwenden wollen.
- **Reset Tracker On Load:** Wenn aktiviert, wird beim Starten deines Spiels automatisch ein *InputTracking.Recenter ()* ausgeführt, der die Ausgangsposition und Rotation des Headsets zurücksetzt.
- **Mixed Reality Capture:** Diese Funktion erlaubt es dir, im Unity Editor *Mixed Reality*-Videos zu drehen, wobei du dein echtes Webcam-Bild über einen Greenscreen in die virtuelle Welt projizierst. Deinen Zuschauern kannst du so eine Third-Person-Perspektive auf dich bieten.[3]

  Neben den vorgestellten Einstellungen, die du im Inspector vornehmen kannst, stehen dir noch weitere Variablen und Methoden zur Verfügung, die du aus einem Skript heraus aufrufen kannst.

- **isUserPresent** : `bool`

  Über diese Variable kannst du prüfen, ob die VR-Brille derzeit von dem Spieler getragen wird (dies wird über den Näherungssensor der Rift und GearVR bestimmt).

- *static* **PlatformUIConfirmQuit ()** : `void`

  Öffnet den Oculus-Dialog zum Beenden des Spiels.

- *static* **void PlatformUIGlobalMenu ()** : `void`

  Öffnet das Oculus-InGame-Menü.

7.2.2.2 OVRPlayerController-Prefab

Das *OVRPlayerController*-Prefab ist nicht nur eine *Camera*, sondern bereits ein vollständig konfiguriertes Spieler-GameObject. Das bedeutet, du kannst das Prefab in die Scene ziehen und hast sofort ein GameObject, das du mit einem Gamepad oder mit Tastatur und Maus durch deine virtuelle Welt steuern kannst. Dieses Prefab ist jedoch nur mit klassischer Locomotion ausgelegt und nicht für Room-Scale-Spiele.

Wenn du dir den Aufbau in der *Hierarchy* anschaust, stellst du fest, dass das zuvor besprochene *OVRCameraRig* ein Teil des *OVRPlayerControllers* ist. Wenn du dieses Prefab also verwenden möchtest, musst du alle anderen *Cameras* in der Scene löschen oder deaktivieren.

[3] Mehr Details hierzu findest du in diesem Blog-Eintrag von Oculus: https://developer.oculus.com/blog/getting-started-with-mixed-reality-capture/

Bild 7.7 Der OVRPlayerController verwendet das OVRCameraRig und erweitert es um eine einfache „Laufen"-Funktion.

Das *OVRPlayerController*-GameObject ist das Haupt-GameObject, welches du auf Tastendruck bewegen kannst. Das GameObject enthält neben dem *OVRCameraRig* noch ein GameObject mit dem Namen *ForwardDirection*. Dieses GameObject zeigt stets in die aktuelle Vorwärts- bzw. Laufrichtung. Wenn du deinem Spieler ein 3D-Modell geben möchtest, das seinen virtuellen Körper darstellt, solltest du es zu diesem GameObject hinzufügen. Die Components am *OVRCameraRig*-GameObject sind identisch zu dem an dem gleichnamigen *Prefab* in Kapitel 7.2.2.1.

An dem *OVRPlayerController* findest du neben einem *Character Controller* drei weitere Components von Oculus: *OVR Player Controller*, *OVR Scene Sample Controller* und *OVR Debug Info*.

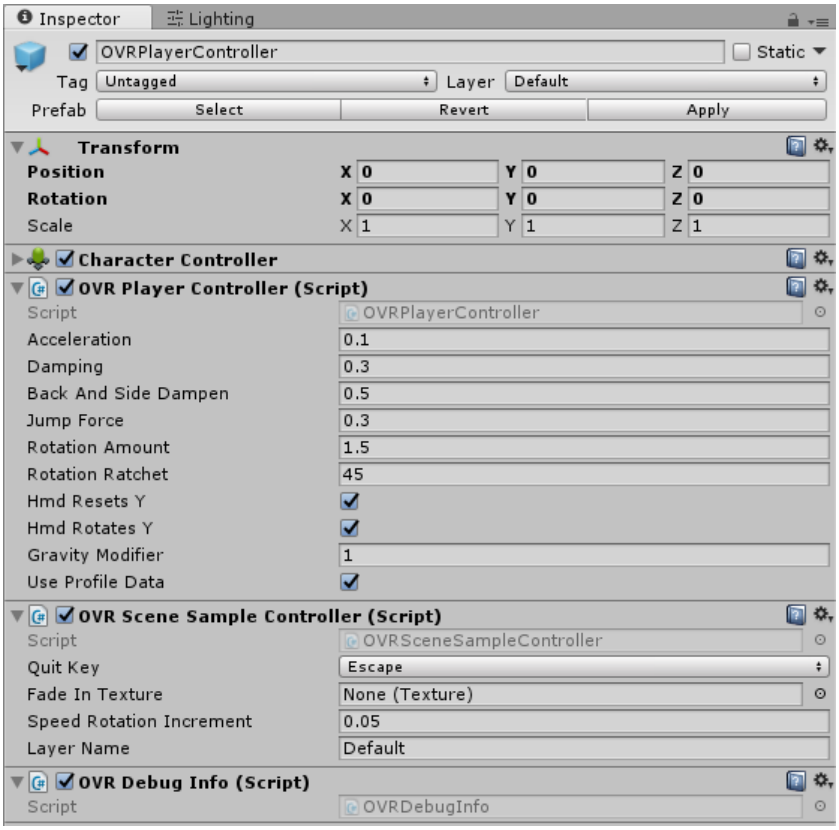

Bild 7.8 Die Components des OVR Player Controller-Prefabs

7.2.2.2.1 OVR Player Controller-Component

Dieses Component steuert die Bewegung der Spielfigur. Es setzt voraus, dass ein *Character Controller*-Component am selben GameObject vorhanden ist. Über die Inspector Eigenschaften kannst du beeinflussen, wie sich der Spieler bewegen soll.

- **Acceleration:** Bestimmt, wie schnell der Spieler beschleunigen soll, wenn er läuft.
- **Damping:** Bestimmt, wie schnell der Spieler wieder zum Stillstand kommt, wenn er keine Taste mehr drückt. (Zu geringes *Damping* kann zu einem erhöhten *Simulator Sickness*-Risiko führen.)
- **Back And Side Damping:** Wie *Damping*, allerdings für Seitwärts- und Rückwärtsbewegung.
- **Jump Force:** Wie hoch kann der Spieler springen?
- **Rotation Amount:** Wie schnell soll sich der Spieler mit einem Gamepad drehen können? (Zu schnelles Drehen kann ebenfalls zu einem erhöhten *Simulator Sickness*-Risiko führen.)

- **Rotation Ratchet:** Wenn du eine stufenweise Rotation, z. B. über die Tastatur, verwendest, kannst du hier angeben, wie groß eine Stufe ist, also um wie viel Grad sich der Spieler bei einem Tastendruck drehen soll.
- **HMD Resets Y:** Wenn aktiviert, dreht sich die Spielfigur beim Recentern ebenfalls zurück in ihre Ausgangsposition.
- **HMD Rotates Y:** Wenn aktiviert, entspricht die Blickrichtung mit dem Headset immer auch der Laufrichtung. Wenn diese Option deaktiviert ist, bestimmst du nur über den Controller oder die Tastatur und Maus die Laufrichtung und kannst dabei in jede beliebige Richtung mit der VR-Brille schauen.
- **Gravity Modifier:** Bestimmt, wie stark diese Figur von der Schwerkraft betroffen ist.
- **Use Profile Data:** Wenn aktiviert, entspricht die Größe dieses Spieler-Objektes immer der im Oculus-Profil angegebenen Körpergröße. Ansonsten haben alle Spieler dieselbe von dir einstellbare Größe.

7.2.2.2.2 OVR Scene Sample Controller-Component

Dies ist ein kleines Werkzeug, das von Oculus erstellt wurde, um dir beim Herumprobieren von verschiedenen Werten zu helfen, ohne dass du die VR-Brille abnehmen musst.

- Du kannst die Laufgeschwindigkeit über **die Zahlentasten 7 und 8** ändern.
- Die Rotationsgeschwindigkeit über **9 und 0**.
- **F2** ändert den sogenannten Vision Mode.
- **F11** wechselt zwischen Vollbild und Fenstermodus hin und her.
- **M** aktiviert und deaktiviert das Spiegeln der VR-Ansicht auf deinen Monitor.
- **Escape** beendet die Anwendung.
- Wenn du eine **Fade-in Texture** angibst, blendet sich die Sicht bei Spielstart ein.

Dieses Component solltest du entfernen, bevor du dein Spiel an andere Spieler schickst, da es nur für Entwickler vorgesehen ist.

7.2.2.2.3 OVR Debug Info-Component

Dieses Component zeigt dir einige Informationen zu deinem Headset in einem „Heads Up Display“ (kurz „HUD“) an, sodass du zum Beispiel jederzeit die FPS deines Spiels sehen kannst, ohne dass du die VR-Brille abnehmen musst.

- Über Leertaste kannst du die Einblendung ein- und ausschalten.

Dieses Component solltest du entfernen, bevor du dein Spiel an andere Spieler schickst, da es nur für Entwickler vorgesehen ist.

7.2.2.3 OVRCubemapCaptureProbe-Prefab

Dieses Prefab wurde extra dafür entworfen, 360°-Screenshots zu erstellen. Du kannst das Prefab zum Beispiel per Hand im Editor platzieren und dann über Tastendruck oder bei Spielstart auslösen. Du kannst das Prefab aber auch fest in dein Spiel einbauen, indem du es zum Beispiel zu deinem Spieler-GameObject hinzufügst, sodass es sich immer mitbewegt. Über eine voreingestellte Taste können dann deine Spieler jederzeit Screenshots

machen. Die Screenshots werden im *Cubemap*-Format gespeichert, das Oculus auch für die Screenshots in ihrem Store verwendet. Bild 7.9 zeigt, wie ein solcher Screenshot aussehen würde. Zusätzlich zu der Auslösung über eine Taste ist es auch möglich, die Aufnahme aus einem Skript heraus auszulösen.

Bild 7.9 So sieht zum Beispiel ein 360°-Screenshot im Cubemap-Format aus.

- **Triggered By Key:** Hier kannst du eine Taste festlegen, über die ein 360°-Screenshot aufgenommen werden soll.
- **Path Name:** Hier kannst du einen Pfad angeben, wo die 360°-Screenshots gespeichert werden sollen. Standardmäßig werden die Screenshots in einem versteckten Verzeichnis gespeichert. Du solltest hier deshalb einen eigenen Pfad angeben, zum Beispiel *Screenshots*. Dann werden die Screenshots in einem entsprechenden Unterordner in deinem Projekt bzw. Build-Ordner gespeichert. Auf der GearVR ist das Aufnehmen von *Cube Maps* nicht möglich.

7.2.3 Gamepad-, Headset- und Touch-Eingaben lesen

Oculus stellt eine spezielle Klasse bereit, um Eingaben von Headset, Gamepads oder auch den *Touch-Controllern* zu lesen. Anstelle von Unitys-eigener *Input*-Klasse verwendest du für Oculus-Projekte die *OVRInput*-Klasse, welche speziell für die Oculus-Hardware geschrieben wurde. In diesem Abschnitt schauen wir uns an, wie du mit dieser Klasse Eingaben von den Controllern lesen kannst.

Im Kern besteht die Klasse aus drei wichtigen Methoden:

- *OVRInput*.**Get (** *button* : `OVRInput.Button`**)** : `bool`

  OVRInput.**Get (** *axis* : `OVRInput.Axis1D`**)** : `float`

  OVRInput.**Get (** *axis2D* : `OVRInput.Axis2D`**)** : `Vector2`

  Gibt den aktuellen Zustand (`true` oder `false`) der im Parameter übergebenen Taste oder den aktuellen Wert der jeweiligen Achse (*Float* oder *Vector2*) zurück (vergleichbar mit `Input.GetButton(…)` bzw. `Input.GetAxis(…)`).
- *OVRInput*.**GetDown (** *button* : `OVRInput.Button`**)** : `bool`

  Gibt zurück, ob die im Parameter übergebene Taste in diesem Frame gedrückt wurde (wie `Input.GetButtonDown(…)`).
- *OVRInput*.**GetUp (** *button* : `OVRInput.Button`**)** : `bool`

  Gibt zurück, ob die im Parameter übergebene Taste in diesem Frame losgelassen wurde (wie `Input.GetButtonUp(…)`).

Wie bei Unitys Input-Klasse ist der größte Unterschied zwischen `Get` und `GetDown`, dass `Get` kontinuierlich *true* zurückgibt, solange die Taste gehalten wird, `GetDown` und `GetUp` wiederum nur exakt in dem Frame/Update, wo die Situation eingetreten ist. Tastenabfragen werden deshalb, wie in Listing 7.2, für gewöhnlich in der Update-Methode eingefügt, da diese für jeden Frame ausgeführt wird und man so keinen Zustand verpasst.

Listing 7.2 Beispiele für Get, GetDown und GetUp

```
void Update()
{
    if (OVRInput.GetDown(OVRInput.Button.One)) {
        Debug.Log("1 Taste wurde gerade gedrückt!");
    }
    if (OVRInput.Get(OVRInput.Button.One)) {
        Debug.Log("1 Taste wird immer noch gedrückt!");
    }
    if (OVRInput.GetDown(OVRInput.Button.One)) {
        Debug.Log("1 Taste wurde gerade losgelassen!");
    }
    float triggerLeft = OVRInput.Get(OVRInput.Axis1D.PrimaryIndexTrigger);
    Vector2 stickLeft = OVRInput.Get(OVRInput.Axis2D.PrimaryThumbstick);
}
```

Innerhalb der *OVRInput*-Klasse werden die lesbaren Eigenschaften der Controller in verschiedenen Kategorien (bzw. Typen) gruppiert. Elemente der einzelnen Typen kannst du als Parameter an die *Get*-Methode übergeben. Abhängig von dem Typ gibt die *Get*-Methode entweder einen Boolean (bei Buttons), einen Float (bei Achsen) oder einen Vector2 (bei 2D-Achsen) zurück. Die Methoden *GetDown* und *GetUp* können nur mit Buttons aufgerufen werden. Nachfolgend erkläre ich dir die einzelnen Typen:

- **OVRInput.Button:** Diese Kategorie enthält alle Tasten, die auf einem traditionellen Gamepad, auf Oculus Touch-Controllern oder auf der Samsung GearVR zu finden sind. Die Namen der Buttons entsprechen denen auf den *Oculus Touch-Controllern* bzw. dem offiziellen *Samsung Controller für GearVR*. Wenn du Werte von einem anderen Controller (z. B. Xbox-Controller) lesen möchtest, musst du die Namen der Buttons an derselben Position verwenden.
- **OVRInput.Axis1D:** Diese Kategorie enthält alle Eingabemöglichkeiten, welche als Status eine Kommazahl zurückgeben, wie zum Beispiel Trigger. Der zurückgegebene Wert liegt zwischen 0 (Ausgangsposition) und 1 (vollständig gedrückt).
- **OVRInput.Axis2D:** Diese Kategorie enthält alle Eingabemöglichkeiten, welche als Status einen zweidimensionalen Vektor zurückgeben. Dazu zählen zum Beispiel die Thumbsticks der Gamepads.

Um herauszufinden, wie eine bestimmte Taste oder Achse heißt, empfiehlt es sich, einen Blick in die offizielle *Oculus-Dokumentation* für die *OVR Input*-Klasse zu werfen. Dort findest du erklärende Grafiken für alle unterstützten Eingabegeräte. Einen Link dorthin habe ich dir in dem Kasten unten eingefügt.

Welcher Button befindet sich wo auf welchem Eingabegerät?

Die Antwort findest du in der Oculus-Online-Dokumentation:

https://developer.oculus.com/documentation/unity/latest/concepts/unity-ovrinput/#unity-ovrinput-touch

7.2.3.1 Touch und Näherungssensoren der Touch-Controller auslesen

Die Tasten der *Oculus Touch*-Controller haben Näherungssensoren eingebaut. Das erlaubt, dass du nicht nur prüfen kannst, ob eine Taste gedrückt wird, sondern auch, auf welchen Tasten der Spieler derzeit seine Finger liegen hat oder welche Tasten er fast berührt. Für diese Funktion werden in der *OVRInput*-Klasse noch zwei weitere Typen definiert, die alle Tasten mit solchen Sensoren enthalten. Die beiden Typen kannst du sowohl der *Get*- als auch der *GetDown*- bzw. *GetUp*-Methode übergeben. Als Rückgabewert *erhältst* du immer einen *Boolean* (`true` oder `false`).

- **OVRInput.Touch:** Diese Kategorie enthält alle kapazitiven Oberflächen der Oculus Touch-Controller. Diese Flächen werden verwendet, um die Fingerpositionen zu erkennen, und befinden sich unter anderem auf den Triggern und den Zahlentasten.
- **OVRInput.NearTouch:** Diese Kategorie enthält alle Flächen mit einem Näherungssensor auf den Oculus Touch-Controllern. Diese Sensoren werden verwendet, um die Fingerposition zu bestimmen, auch wenn die Finger die Oberfläche selbst nicht berührt.

Listing 7.3 zeigt dir beispielhaft, wie du den *Touch*- und *NearTouch*-Status erkennen kannst.

Listing 7.3 So erkennst du, ob eine der Spieler den Finger auf einer Taste liegen hat.

```
if (OVRInput.GetDown(OVRInput.NearTouch.PrimaryIndexTrigger)) {
    Debug.Log("Finger ist jetzt ÜBER linkem Trigger!");
}
if (OVRInput.GetDown(OVRInput.Touch.PrimaryIndexTrigger)) {
    Debug.Log("Linker Trigger wird jetzt berührt");
}
```

7.2.4 Haptisches Feedback

Die meisten Controller bieten die Möglichkeit, dem Spieler haptisches Feedback über eingebaute Vibrationsmotoren zu geben. Um dieses Vibrations-Feedback zu steuern, gibt es im Oculus SDK zwei verschiedene Ansätze: einen einfacheren Ansatz für normale Gamepads wie den *Xbox Controller für Windows* und einen komplexeren Ansatz für die *Oculus Touch*-Controller.

7.2.4.1.1 Feedback für traditionelle Controller

Zu den traditionellen Controllern gehören zum Beispiel Android Gamepads, Xbox Controller und ähnliche Eingabegeräte mit Vibrationsmotoren. Für diese Geräte kann über die

OVRInput-Klasse der Vibrationsmotor über die `SetControllerVibration(…)`-Methode aktiviert werden. Als Parameter benötigt die Methode eine Frequenz und eine Amplitude, welche das Gefühl des Feedbacks bestimmen.

Die **Frequenz** bestimmt die Geschwindigkeit des haptischen Feedbacks, also wie schnell die Vibrationen erfolgen sollen. Die Methode erwartet Werte zwischen 0 und 1.

Die **Amplitude** bestimmt die Stärke des haptischen Feedbacks, also wie kraftvoll die Vibrationen wirken sollen. Die Methode erwartet Werte zwischen 0 und 1.

Um den Vibrationsmotor wieder zu stoppen, rufst du die Methode einfach mit einer Frequenz und Amplitude von 0 auf.

Du solltest die `SetControllerVibration`-Methode nicht in jedem Frame (bzw. Update) ausführen, da es dann dazu kommen kann, dass der Controller nicht mehr aufhört zu vibrieren. In den meisten Fällen empfiehlt es sich, diese Methode, wie in Listing 7.4, innerhalb einer Coroutine zu verwenden. Um die perfekten Werte für Frequenz und Amplitude zu finden, musst du ein wenig herumprobieren und Erfahrungen mit der Methode sammeln.

Listing 7.4 Eine Coroutine, die den Controller drei Sekunden vibrieren lässt

```
IEnumerator vibrateForSeconds(){
    OVRInput.SetControllerVibration(0.5f, 0.8f); // Turn feedback on
    yield return new WaitForSeconds(3.0f);
    OVRInput.SetControllerVibration(0,0); // Turn feedback off
}
```

7.2.4.1.2 Haptisches Feedback für Oculus Touch-Controller

Das Feedback-System für die *Oculus Touch*-Controller funktioniert ein wenig anders als das für traditionelle Controller. Es arbeitet mit sogenannten *Haptics Clips*, welche jeweils ein bestimmtes Vibrationsmuster beschreiben. Diese Clips können entweder per Hand geschrieben werden oder auch aus *AudioClips*, also Soundeffekten, generiert werden.

Haptics Clip anlegen

Haptics Clips werden mit 320 Hz verarbeitet, das bedeutet, ein *Haptics Clip*, der aus 320 Samples besteht, würde *eine Sekunde* lang vibrieren. Je nach Vibrationseffekt werden also sehr viele Samples benötigt, was das Anlegen per Hand sehr aufwendig gestaltet. Aus diesem Grund werden wir uns an dieser Stelle den deutlich einfacheren Weg ansehen: das Erzeugen von *Haptics Clips* aus existierenden Sound-Dateien (*Audio Clips*).

Für diese Variante kannst du jede beliebige Sound-Datei aus deinem Projekt verwenden und daraus ein Vibrationsmuster basierend auf der Lautstärke des Clips generieren lassen. Alles, was du dafür tun musst, ist, einen neuen *OVRHapticsClip* mit dem `new`-Schlüssel anzulegen und ihm einen *AudioClip* als Parameter zu übergeben.

Listing 7.5 Beispiel, um einen Haptics Clip aus einem Audio Clip zu erzeugen

```
public AudioClip sfxCanon; // Über Inspector zuweisen
void Start () {
    OVRHapticsClip hapticsClip = new OVRHapticsClip(sfxCanon);
}
```

Haptics Clip auf Controller abspielen

Wenn du einen *Haptics Clip* erstellt hast, kannst du ihn auf einen der beiden Touch-Controller über den Vibrationsmotor abspielen lassen. Die Vibrationen für den linken und rechten Controller sind vollkommen unabhängig voneinander. Sollen beide gleichzeitig vibrieren, musst du deshalb denselben *Haptics Clip* an beide Controller schicken.

Die *OVRHaptics*-Klasse besitzt zwei statische Eigenschaften *LeftChannel* und *RightChannel*, welche für den linken und den rechten *Touch-Controller* stehen. *LeftChannel* und *RightChannel* stellen wiederum Methoden zur Verfügung, mit denen du die *Haptics Clips* auf dem jeweiligen Controller abspielen kannst. Die Methoden unterscheiden sich in erster Linie dadurch, wie vorgegangen wird, wenn zu diesem Zeitpunkt bereits ein anderer *Haptics Clip* abgespielt wird:

- *OVRHaptics.LeftChannel* .**Queue (** *clip* : `OVRHapticsClip`**)** : `void`

  Reiht den neuen Clip in die Warteschlange für diesen Controller ein. Er wird dann abgespielt, nachdem zuvor eingereihte Clips zu Ende gespielt wurden.

- *OVRHaptics.LeftChannel.***Clear()** : `void`

  Entfernt alle Clips aus der Warteschlange für diesen Controller und stoppt die Wiedergabe des aktuellen Clips.

- *OVRHaptics.LeftChannel.***Preempt (** *clip* : `OVRHapticsClip`**)** : `void`

  Entfernt alle vorhandenen Clips aus der Warteschlange und spielt den neuen Clip sofort ab. (Entspricht einem Aufruf von `Clear`, gefolgt von `Queue`)

- *OVRHaptics.LeftChannel.***Mix(** *clip* : `OVRHapticsClip`**)** : `void`

  Spielt den neuen Clip parallel zu den bereits in der Warteschlange existierenden Clips ab. Dies kann zum Beispiel verwendet werden, wenn der Spieler sich in einem einstürzenden Gebäude befindet und gleichzeitig eine Waffe abfeuert.

Listing 7.6 zeigt, wie du einen neuen *Haptics Clip* aus einer *AudioClip* erzeugst und anschließend auf beiden *Touch-Controllern* gleichzeitig abspielst.

Listing 7.6 Beispiel für das Abspielen von haptischem Feedback für die Oculus Touch-Controller

```
public void playHapticFeedback(AudioClip sfxForHaptics){
       OVRHapticsClip myHapticsClip = new OVRHapticsClip(sfxForHaptics);
       OVRHaptics.LeftChannel.Preempt(myHapticsClip); // linker Touch
       OVRHaptics.RightChannel.Preempt(myHapticsClip); // rechter Touch
}
```

7.2.5 Oculus Avatar SDK

Das Oculus Avatar SDK ist ein spezielles SDK, welches du verwenden kannst, um die Oculus Avatare, die sich jeder Spieler in dem Oculus-Home-Menü selber zusammenbauen kann, in dein Spiel zu integrieren. Das kann auch für Singleplayer-Spiele Sinn machen, da dir das SDK erlaubt, die aus *Oculus Home* und der *Oculus Touch-Einführung* bekannten Hände in dein Spiel zu integrieren, die immer die korrekte Position der Finger anzeigen.

Oculus Avatar SDK

Die im Buch beschriebene Version findest du auf der begleitenden Webseite (Version: 1.16.0_2017Fix):

http://www.VRSpieleEntwickeln.de/zusatz/

Unter *Alle SDKs*, dann *OvrAvatar_1.16.0_2017Fix.unityPackage*

Asset Store:

https://www.assetstore.unity3d.com/en/#!/content/82022

(Das Avatar SDK ist im Asset Store Package bereits enthalten)

Oculus-Entwickler-Downloads:

https://developer.oculus.com/downloads/package/oculus-avatar-sdk/

Um den Oculus Avatar verwenden zu können, musst du einfach das *LocalAvatar*-Prefab, welches du in dem Ordner *OvrAvatar/Content/Prefabs* findest, als Kind in das *OVRCameraRig*-GameObject in deiner Scene einfügen. Das *LocalAvatar*-GameObject sollte danach auf einer Ebene mit dem *TrackingSpace*-GameObject liegen. Bild 7.10 zeigt, wie die GameObjects nach dem Einfügen des Prefabs in der *Hierarchy* aussehen sollten und wie das Ergebnis dann im Spiel aussehen kann.

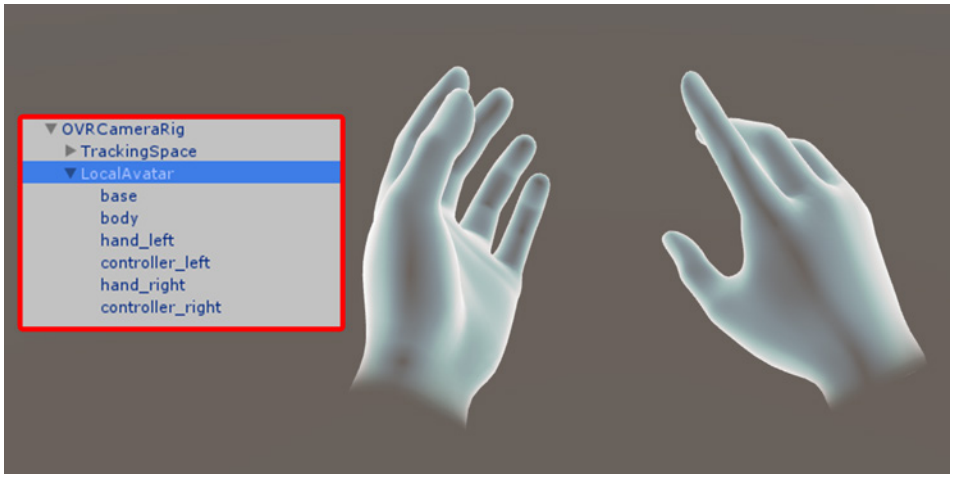

Bild 7.10 So sehen die Avatar-Hände dann in der Hierarchy (links) und Game View (rechts) aus.

Import kann einige Minuten in Anspruch nehmen

Beim Importieren des Avatar-Packages kann es vorkommen, dass Unity einige Minuten benötigt, um die Datei *AvatarSurfaceShaderSelfOccluding.shader* zu verarbeiten und das Unity-Fenster in der Zeit nicht auf Eingaben reagiert.

7.2.6 Mehr Infos und Hilfe

Für den Anfang hast du jetzt schon alle wichtigen Prefabs und Components der Oculus-Integration kennengelernt. In dem Ordner *Scenes* findest du noch ein paar einfache Beispiele, die du dir ansehen kannst. Während des Beispielprojektes werden wir uns dann ansehen, wie man mit der Integration in der Praxis arbeitet.

Ich habe dir in dieser Einführung nicht jede einzelne Funktion und jede einzelne Klasse vorgestellt, sondern nur das, was du in der Praxis in den meisten Fällen auch tatsächlich brauchst. Nachfolgend habe ich dir aber noch eine Sammlung von Links mit erweiterten Informationen und Details zusammengestellt, welche dir bei Problemen helfen sollten.

Mehr Infos und Online-Dokumentation

Offizielle, allgemeine Einführung in die Unity-Werkzeuge:

https://developer.oculus.com/documentation/unity/latest/concepts/book-unity-gsg/

Die Oculus-Übersicht zu den Unity-Werkzeugen:

https://developer.oculus.com/documentation/unity/latest/concepts/unity-utilities-overview/

Häufige Fragen und Antworten zu den Werkzeugen:

https://developer.oculus.com/documentation/unity/latest/concepts/unity-faq/

Mehr Details zu OVRInput:

https://developer.oculus.com/documentation/unity/latest/concepts/unity-ovrinput/#unity-ovrinput

Mehr Details zu OVRHaptics:

https://developer.oculus.com/documentation/unity/latest/concepts/unity-ovrhaptics/

Die Code-Dokumentation mit allen Klassen, Methoden und Variablen:

https://developer3.oculus.com/doc/1.6-unity/index.html

7.3 SteamVR (u. a. HTC Vive)

Die Entwicklungswerkzeuge für SteamVR benötigst du, wenn du ein Spiel für eine *SteamVR*-kompatible Brille wie die *HTC Vive* entwickeln möchtest. Das SteamVR-Plug-in unterstützt nämlich neben der Vive auch einige andere VR-Brillen wie die *Oculus Rift inkl. Oculus Touch* sowie zum Beispiel das *OSVR-Entwickler-Kit* von Razer. Die Schnittstelle von SteamVR wird deshalb auch *OpenVR* genannt.

Das SteamVR-SDK ist so geschrieben, dass die meiste Logik in der *Runtime* (also der *SteamVR*-Anwendung) liegt und nicht in den Spielen selbst. Du implementierst lediglich die Schnittstelle zu SteamVR, die SteamVR-Anwendung enthält dann die Logik, die benötigt wird, um die einzelnen unterstützten Brillen anzusprechen. Das hat den Vorteil, dass du dir in den meisten Fällen keine Gedanken machen musst, für welche der unterstützten Brillen du entwickelst.

SteamVR unterscheidet zwischen *Tracking Space* und *Play Area*. Der Tracking Space ist der komplette dir zur Verfügung stehende Bereich, an dessen Grenzen das *Chaperone*[4] eingeblendet wird, damit du nicht gegen eine Wand oder einen Tisch läufst. Der Tracking Space muss nicht rechteckig sein, sondern kann exakt dem dir zur Verfügung stehenden Platz entsprechen. Wenn du ein Spiel entwickelst, wäre es allerdings sehr umständlich, jede beliebige *Tracking Space*-Form zu berücksichtigen. Aus diesem Grund gibt es zusätzlich noch die *Play Area*. Die *Play Area* ist, wie in Bild 7.11 zu sehen, das größtmögliche Rechteck, das in dein *Tracking Space* passt. Wenn du ein Spiel entwickelst, kannst du also stets davon ausgehen, dass der Spieler jeden Punkt innerhalb dieses rechteckigen Bereiches erreichen kann, und musst dir nicht um anders geformte Spielbereiche Gedanken machen.

Bild 7.11 Hellblau: der gesamte Tracking Space, grün: die Play Area, auf die sich die meisten Spiele beschränken werden

[4] So heißt das Sicherheitssystem von SteamVR, welches dir in der virtuellen Welt die Grenzen deines Tracking Space anzeigt.

7.3.1 Download und Import

Im Kasten unten findest du den Download-Link zu der *SteamVR SDK*-Version, die wir in diesem Buch besprechen werden. Außerdem findest du dort auch einen Link zu dem *SteamVR Plugin* im Unity Asset Store, wo du immer die neuste Version herunterladen kannst. Diese Version kann sich eventuell von der hier besprochenen Version unterscheiden.

SteamVR

Die im Buch beschriebene Version findest du auf der begleitenden Webseite (Version: 1.2.2):

http://www.VRSpieleEntwickeln.de/zusatz/

Unter *Alle SDKs*, dann *SteamVR Plugin_1_2_2.unitypackage*

Neuste Version im Unity Asset Store:

https://www.assetstore.unity3d.com/en/#!/content/32647

Nachdem du das Package heruntergeladen hast, kannst du es, wie im „Quickstart"-Kapitel beschrieben, importieren.

Nach dem Import-Vorgang öffnet sich das Fenster aus Bild 7.12 mit einigen Optimierungsvorschlägen für dein Projekt. Lese dir die Änderungen, die das Plug-in empfiehlt, aufmerksam durch und wähle dann Accept All, um alle Änderungen automatisch vornehmen zu lassen.

Wenn du die *Personal*-Edition von Unity nutzt, wird sich das Fenster wahrscheinlich bald nochmals öffnen und dir erneut vorschlagen, den *Splash Screen* zu deaktivieren. Da dies eine der wenigen Funktionen ist, die in der kostenlosen Unity-Version eingeschränkt sind, solltest du hier auf *Ignore* klicken, um diese Empfehlung dauerhaft auszublenden.

Bild 7.12 Das SteamVR-Plug-in erkennt automatisch, welche Änderungen in deinem Projekt vorgenommen werden sollten.

Nachdem du das alles gemacht hast, solltest du in deinem Project Browser, wie in Bild 7.13, einen *SteamVR*-Ordner sowie diverse Unterordner sehen. Wenn alles so weit in Ordnung aussieht, bist du bereit, weiter zu machen.

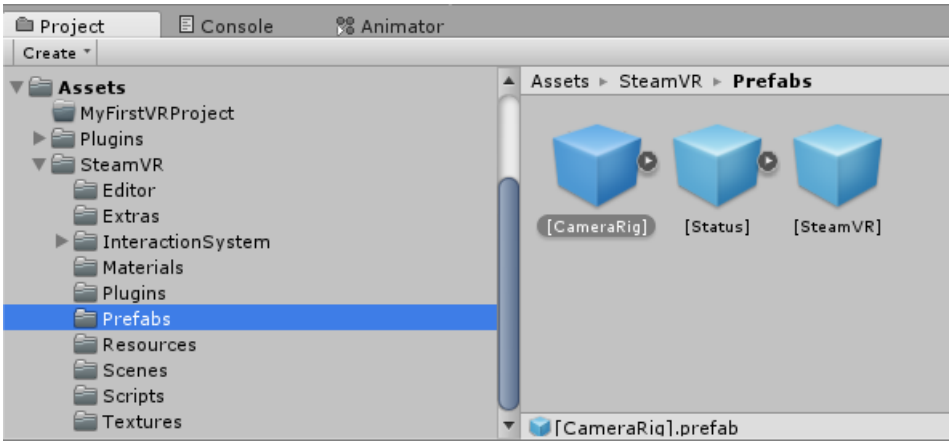

Bild 7.13 Nach dem Import solltest du in deinem *Project Browser* einen „SteamVR“-Ordner sehen.

7.3.2 Prefabs

Das *SteamVR Plugin* stellt dir drei *Prefabs* zur Verfügung, welche die Grundfunktionalität des SteamVR-Plug-ins darstellen. Zusätzlich enthält das Package noch einige weitere Prefabs mit Beispielen für verschiedene Interaktionen, diese werde ich dir in dem nachfolgenden Kapitel 7.3.4 vorstellen.

7.3.2.1 SteamVR-Prefab

Dieses Prefab stellt den Kern der SteamVR-Funktionalität dar und muss in jeder Scene vorhanden sein. Legst du es nicht selber an, wird es automatisch angelegt, sobald es benötigt wird, allerdings mit den Standardeinstellungen. Das *SteamVR_Renderer*-Component an diesem Prefab sorgt dafür, dass alle *SteamVR_Cameras* ein korrektes Bild für die VR-Brillen berechnen.

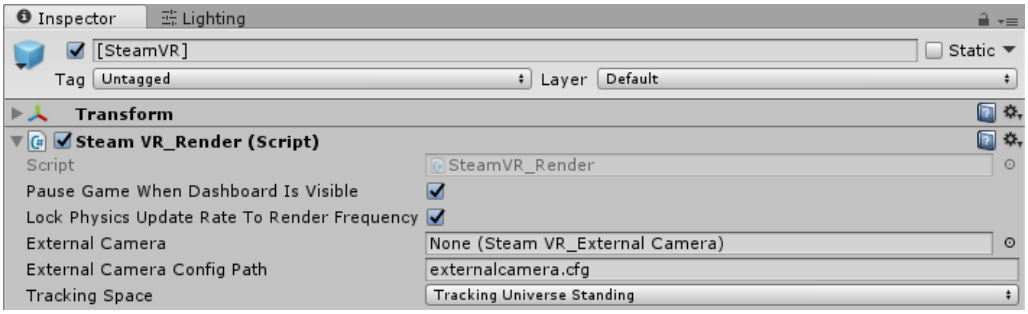

Bild 7.14 Die Components des SteamVR-Prefabs

7.3.2.1.1 SteamVR_Renderer-Component

Dieses Component rendert vollkommen automatisch alle *SteamVR_Cameras*. Es bietet außerdem die Möglichkeit, einen Mixed-Reality-Modus zu aktivieren, bei dem eine Aufnahme des realen Spielers mit der virtuellen Welt überlagert wird (z. B. für Livestreams oder Trailer). Mehr Infos hierzu findest du in dem Kasten unten.

Im *Inspector* bietet das Component folgende Optionen:

- **Pause Game When Dashboard Is Visible:** Wenn es aktiviert ist, pausiert dein Spiel automatisch, sobald der Spieler die *Steam Overlay* öffnet. Wenn du ein Multiplayer-Spiel mit Online-Funktionalität erstellst, solltest du diesen Punkt deaktivieren, da es sonst zu Verbindungsabbrüchen kommen kann.
- **Lock Physics Update Rate To Render Frequency:** Wenn aktiviert, nutzt die Physik-Engine das gleiche Timing wie die Darstellung in der VR-Brille. Diese Option verhindert ruckelige oder fehlerhafte Physik-Darstellung
- **External Camera:** Hier gibst du ein konfiguriertes *SteamVR_External_Camera*-GameObject an, wenn du ein Mixed-Reality-Video drehen möchtest.

- **External Camera Config Path:** Hier gibst du den Pfad zu einer externalcamera.cfg an, welche die Mixed-Reality-Darstellung konfiguriert. Mit dem Standardwert liegt sie im Stammverzeichnis deines Projektes.
- **Tracking Space:** Hier kannst du eine bestimmte *Tracking Space*-Art erzwingen. Wenn dein Spiel zum Beispiel eine sitzende Experience ist, kannst du das hier einstellen, um das *Recentern* der Kamera zu ermöglichen.

Mixed-Reality-Videos in SteamVR

Um ein Mixed-Reality-Video zu drehen, benötigst du einen SteamVR-Tracker, eine Videokamera und im Idealfall einen Greenscreen-Würfel, mindestens aber eine Greenscreen-Wand. Falls du mehr wissen möchtest, kannst du dir folgende Links genauer ansehen.

So funktionieren Mixed-Reality-Videos in SteamVR:

https://www.youtube.com/watch?v=IPTazOkOLRw

Anleitung, um Mixed-Reality-Videos aufzunehmen (Englisch):

https://steamcommunity.com/app/358720/discussions/0/405694031549662100/?l=german

7.3.2.2 Camera Rig-Prefab

Dieses Prefab ersetzt in SteamVR-Projekten die gewöhnliche *Camera* in der Scene. Das bedeutet, du musst die standardmäßig angelegte *Main Camera* aus der Scene löschen und ziehst anschließend dieses Prefab in deine Scene.

In der *Hierarchy* sieht das GameObject dann wie in Bild 7.15 aus.

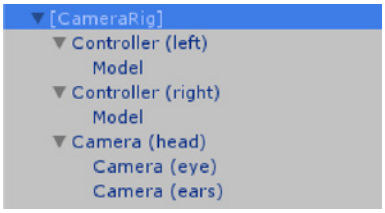

Bild 7.15 Der Aufbau des OVRCameraRigs in der Hierarchy

Alle Bewegungen, die du normalerweise mit dem *Main Camera*-GameObject gemacht hättest, musst du jetzt mit dem *[CameraRig]* machen, nicht mit einem der Kind-Objekte.

Die Position des *CameraRigs* entspricht immer dem Zentrum deines realen *Tracking Space* bzw. deiner realen *Play Area*. An dem GameObject kannst du zwei SteamVR-Components finden: *SteamVR_ControllerManager* (siehe Kapitel 7.3.1.2.1) und *SteamVR_PlayArea* (siehe Kapitel 7.3.1.2.2).

Das GameObject hat drei direkte Kinder:

- **Controller (left) und Controller (right)**

  Die beiden Controller-GameObjects werden durch das *SteamVR_TrackedObject*-Component automatisch entsprechend der Position der realen Controller positioniert (siehe Kapitel 7.3.1.2.3). Jeder der Controller hat ein „Model"-GameObject als Kind, welche jeweils ein *SteamVR_RenderModel*-Component besitzen. Dieses Component kann verwendet werden, um automatisch 3D-Modelle der Controller an der korrekten Position anzuzeigen (siehe Kapitel 7.3.1.2.4).

- **Controller (head)**

  Dieses GameObject wird durch das *SteamVR_TrackedObject*-Component automatisch an der Position der VR-Brille innerhalb deines Tracking-Bereiches positioniert. Das GameObject hat zwei Kinder. *Camera (eye)* enthält die *Camera*, welche das Bild für die VR-Brille rendert, und *Camera (ears)* einen *Audio Listener*, der die Ohren des Spielers darstellt. Beide GameObjects haben zusätzlich noch Components des SteamVR-Plug-ins, die das Verhalten der *Camera* und des *Audio Listeners* steuern (*SteamVR_Camera* und *SteamVR_Ears*, siehe dazu Kapitel 7.3.1.2.5 und Kapitel 7.3.1.2.6). Das *Controller (head)*-GameObject selbst besitzt ebenfalls eine *Camera*, welche für das gespiegelte Bild auf dem Monitor verwendet wird.

7.3.2.2.1 SteamVR_ControllerManager-Component

Bild 7.16 Das SteamVR Controller Manager-Component im Inspector

Das *Controller Manager*-Component aktiviert und deaktiviert die GameObjects *Controller (left)* und *Controller (right)*, wenn ein Controller mit SteamVR verbunden beziehungsweise die Verbindung getrennt wird. Da beliebig viele Controller und Tracker mit SteamVR verbunden werden können, können in dem *Objects*-Array noch beliebig viele weitere *GameObjects* hinzugefügt werden, welche nach und nach für jeden weiteren Controller oder Tracker aktiviert bzw. deaktiviert werden. Diese GameObjects sollten ein konfiguriertes *SteamVR_TrackedObject*-Component besitzen, damit ihre Position auch entsprechend dem zusätzlichen Controller oder Tracker angepasst wird.

Wenn du deine Controller einschaltest, wird von SteamVR basierend auf ihrer Position (links oder rechts vom Headset) bestimmt, ob sich der Controller gerade in deiner linken oder rechten Hand befindet. Im Normalfall werden die entsprechenden GameObjects erst nach dieser Erkennung aktiviert. Aktivierst du die Option *Assign All Before Identified*, wird anstelle der Position die Reihenfolge des Einschaltens verwendet, um den Controller einer Rolle zuzuordnen. Der erste Controller, der eingeschaltet wird, ist dann der rechte und der zweite der linke.

7.3.2.2.2 SteamVR_PlayArea-Component

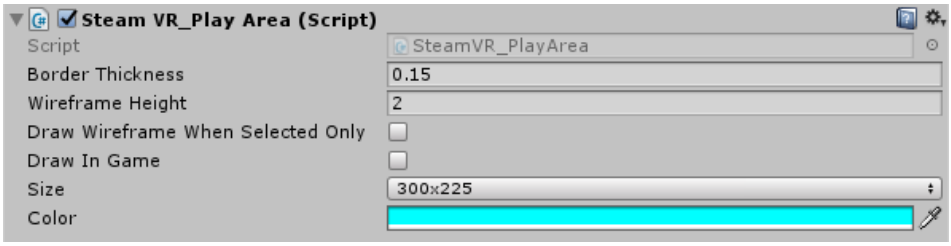

Bild 7.17 Das SteamVR Play Area-Component im Inspector

Das *Play Area*-Component zeichnet eine Vorschau der verschiedenen *Play Area*-Größen in Form eines Rechtecks in der *Scene View* und optional auch im eigentlichen Spiel. Diese Anzeige kannst du nutzen, um dein Level-Design perfekt auf die unterschiedlichen *Play Area*-Größen anzupassen. Zusätzlich kannst du diese Anzeige auch in deinem Spiel aktivieren, damit die Begrenzung dauerhaft angezeigt wird. Dieses Component ist nur für die optionale Darstellung der *Play Area* zuständig, nicht für das Chaperone, das die Grenzen von deinem *Tracking Space* anzeigt.

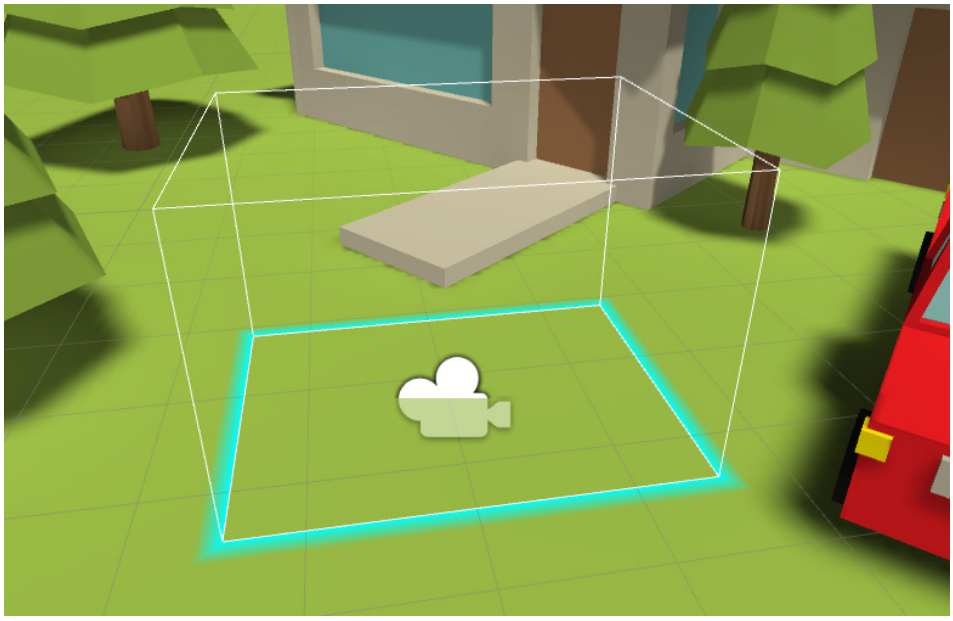

Bild 7.18 So wird die Play Area in der Scene View dargestellt.

Das Component bietet dir im *Inspector* folgende Einstellungsmöglichkeiten:

- **Border Thickness:** Gibt die Dicke der dargestellten Begrenzung auf dem Boden an.
- **Wireframe Height:** Gibt die Höhe des in der *Scene View* dargestellten Quaders an, der dir helfen soll, die Größe des Spielers einzuschätzen.

- **Draw Wireframe When Selected Only:** Wenn aktiviert, wird der Quader zur Einschätzung der Höhe nur angezeigt, wenn das GameObject in der Scene bzw. Hierarchy markiert ist.
- **Draw In Game:** Bestimmt, ob die Bodenbegrenzung auch im Spiel angezeigt wird oder nur in der *Scene View* des Editors.
- **Size:** Hier kannst du die dargestellte Play-Area-Größe ändern und so unterschiedliche Varianten ausprobieren. Wählst du die Option *Calibrated*, wird die *Play Area*-Größe aus der SteamVR-Konfiguration übernommen. (Diese Option ändert jedoch nicht deine SteamVR-Konfiguration, sondern nur die Vorschau dieses Components im Editor.)
- **Color:** Gibt die Farbe der Begrenzung auf dem Boden an.

Dieses *Play Area*-Component ist in erster Linie als Entwicklungswerkzeug gedacht, weshalb du in den meisten Fällen wahrscheinlich die Option *Draw in Game* deaktivieren möchtest, bevor du dein Spiel veröffentlichst. Die meisten Entwickler gehen so vor, dass sie ihr Level für die größtmögliche *Play Area* designen.

Mithilfe eines Skripts kannst du von diesem Component auch die individuelle Größe der Play Area deines Spielers auslesen. Diesen Wert kannst du dann zum Beispiel verwenden, um das Level dynamisch an den Platz, der dem Spieler zur Verfügung steht, anzupassen.

7.3.2.2.3 SteamVR_TrackedObject-Component

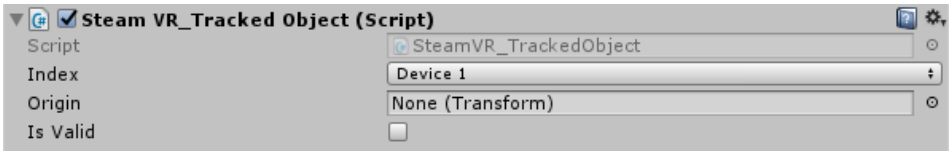

Bild 7.19 Das SteamVR Tracked Object-Component im Editor

Das *Tracked Object*-Component ist dafür zuständig, dass sich GameObjects synchron zu einem getrackten Gegenstand in der echten Welt (z. B. Headset, Controller oder SteamVR Tracker) bewegen.

- **Index:** Dieser Index bestimmt, zu welchem getrackten Objekt dieses GameObject gehört. *Hmd* ist die VR-Brille und *Device 1* bis *Device 4* sind für die beiden Controller sowie die beiden Lighthouses reserviert. Einem dritten Controller würdest du also den Index *Device 5* geben. Du kannst bis zu elf zusätzliche Controller oder Tracker einbinden.

In dem Ordner *Scenes* findest du eine „example"-Scene, welche eine Konfiguration für die maximale Anzahl an trackbaren Geräten enthält (Brille, zwei Lighthouses, zwei Controller und elf zusätzliche Tracker oder Controller).

7.3.2.2.4 SteamVR_RenderModel-Component

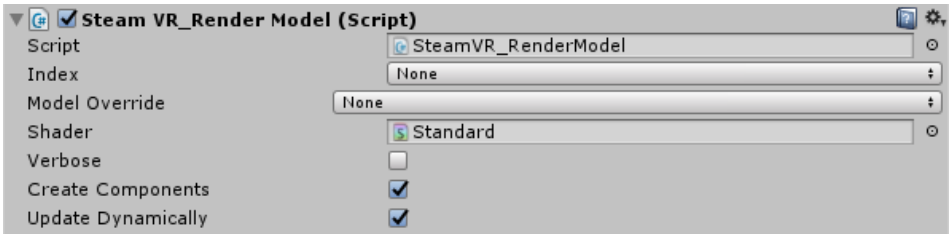

Bild 7.20 Das SteamVR Render Model-Component im Inspector

Das *Render Model*-Component lädt automatisch ein Modell, das zu der Hardware des jeweiligen getrackten Gegenstands mit dem angegebenen Index passt.

Wird der Index eines Controllers angegebenen und der Spieler verwendet zum Beispiel *HTC Vive*-Controller, werden ihm, wie in Bild 7.21, *Vive*-Controller angezeigt. Verwendet er *Oculus Touch*-Controller über SteamVR, werden ihm die *Oculus Touch*-Controller angezeigt. Ist der ausgewählte Index eine *Lighthouse-Station*, wird ein *Lighthouse*-Modell geladen und so weiter. Auch bei zusätzlichen Controllern oder Trackern wird automatisch erkannt, um welche Hardware es sich handelt, und ein entsprechendes Modell geladen, wenn die Hardware unterstützt wird.

Bild 7.21 Sobald du die Anwendung startest, lädt das Skript automatisch das entsprechende Modell.

Die Modelle, die von diesem Component geladen werden, liegen nicht in deinem Projektverzeichnis, sondern sind Teil der *SteamVR Runtime*.

Eigene Modelle statt der Standardmodelle

Wenn du eigene 3D-Modelle, z. B. Hände oder Waffen, anstelle der Controller-Modelle verwenden möchtest, deaktiviere dieses Component oder lösche das *Model*-GameObject aus den Controller-GameObjects.

7.3.2.2.5 SteamVR_Camera-Component

Bild 7.22 Das SteamVR Camera-Component im Inspector

Das *Camera*-Component des SteamVR-Plug-ins wird in erster Linie dafür verwendet, um eine gewöhnliche Unity-*Camera* als eine für SteamVR relevante *Camera* zu markieren und diese leichter in der *Scene* zu finden.

Der *Collapse*-Button ist für dich nicht relevant, wenn du das Prefab verwendest. Wenn du das SteamVR_Camera-Component zu einer beliebigen *Camera* hinzufügst, siehst du an dieser Stelle einen *Expand*-Button, der aus der normalen *Camera* automatisch ein einfaches Camera-Rig generiert. Der *Collapse*-Button dient dazu, diesen Vorgang wieder rückgängig zu machen.

7.3.2.2.6 SteamVR_Ears-Component

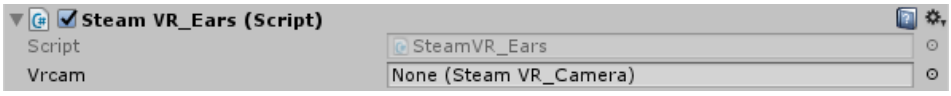

Bild 7.23 Das SteamVR Ears-Component im Inspector

Das *Ears*-Component sorgt dafür, dass der Sound immer aus der richtigen Richtung kommt, wenn du Lautsprecher oder eine Surround-Anlage anstelle von Kopfhörern verwendest. Ohne dieses Component wären zum Beispiel links und rechts vertauscht, wenn du dich um 180° drehst. Verwendest du keine Kopfhörer, hat dieses Skript keine Auswirkung auf den Sound.

7.3.3 Eingaben der Controller lesen

Valve stellt eine spezielle Klasse bereit, um Eingaben von den SteamVR-Controllern zu lesen. Anstelle von Unitys *Input*-Klasse musst du in SteamVR-Projekten die *SteamVR_Controller*-Klasse verwenden.

Der Aufbau ist etwas anders als bei Unitys eigener Input-Klasse. Du benötigst zunächst die interne ID des Controllers, von dem du Eingaben lesen möchtest. Diese ID kannst du auf verschiedene Arten und Weisen erhalten. Eine einfache Variante ist, den Index über das *SteamVR_ControllerManager*-Component auszulesen. Die Umsetzung in Listing 7.7 funktioniert zum Beispiel vollständig script-basiert. Der Code zum Auslesen des Index befindet sich in der *Update*-Methode, damit der Index auch dann automatisch ausgelesen wird, wenn der Controller erst nach Spielstart eingeschaltet wird. Dadurch befindet sich zwar ein `FindObjectOfType` im Update Loop, es wird allerdings nur so lange ausgeführt, bis beide Controller verbunden wurden und der Index ausgelesen wurde.

Listing 7.7 Beispiel: IDs des linken und rechten Controllers erhalten

```
private int leftControllerId = -1;
private int rightControllerId = -1;
void Update(){
    if (leftControllerId == -1) {
        leftControllerId = (int)FindObjectOfType<SteamVR_ControllerManager>()
                                .left.GetComponent<SteamVR_TrackedObject>().index;
    }
    if (rightControllerId == -1) {
        rightControllerId = (int)FindObjectOfType<SteamVR_ControllerManager>()
                                .right.GetComponent<SteamVR_TrackedObject>().index;
    }
}
```

Im Kern besteht die Klasse aus folgenden Methoden:

- **SteamVR_Controller.Input(*id*).GetPress (*button* : `ButtonMask`) : `bool`**

  Gibt den aktuellen Zustand (*true* oder *false*) eines bestimmten Buttons auf dem Controller zurück (wie `Input.GetButton(…)`).

- **SteamVR_Controller.Input(*id*).GetPressDown(*button* : `ButtonMask`) : `bool`**

  Gibt zurück, ob ein bestimmter Button in diesem Frame gedrückt wurde (wie `Input.GetButtonDown(…)`).

- **SteamVR_Controller.Input(*id*).GetPressUp(*button* : `ButtonMask`) : `bool`**

  Gibt zurück, ob ein bestimmter Button in diesem Frame losgelassen wurde (wie `Input.GetButtonUp(…)`).

- **SteamVR_Controller.Input(*id*).GetTouch (*touchpad* : `ButtonMask`) : `bool`**
  SteamVR_Controller.Input(*id*).GetTouchDown(*touchpad*: `ButtonMask`) : `bool`
  SteamVR_Controller.Input(*id*).GetTouchUp(*touchpad*: `ButtonMask`) : `bool`

  Diese drei Methoden verhalten sich genauso wie ihre *GetPress*-Pendants, nur dass sie verwendet werden können, um zu erkennen, ob ein Touchpad berührt bzw. nicht berührt wird. Diese Methode gibt nur zurück, ob das jeweilige Touchpad berührt wird. Um die Position des Fingers auf dem Pad zu erhalten, muss du die Methode *GetAxis* verwenden.

- **SteamVR_Controller.Input(*id*).GetAxis (*axis* : `EVRButtonId`) : `Vector2`**

  Gibt einen 2D-Vektor zurück, der den Wert der jeweiligen Achse beschreibt. Diese Methode kannst du zum Beispiel verwenden, um die Touch-Position auf dem Touchpad oder die Trigger-Position auszulesen.

Als Parameter musst du jeweils eine Button-ID aus der `ButtonMask`-Liste übergeben. Wie schon bei der Unity *Input*-Klasse implementierst du auch diese Abfragen üblicherweise in der `Update`-Methode.

Listing 7.8 Beispiel für GetPress-, GetTouch- und GetAxis-Methoden

```
if(leftControllerId != -1){
    if(SteamVR_Controller.Input(leftControllerId)
        .GetPressDown(SteamVR_Controller.ButtonMask.Trigger))
    {
        Debug.Log("Linker Trigger wurde gerade gedrückt!");
    }
}
if(rightControllerId != -1){
    if(SteamVR_Controller.Input(rightControllerId)
        .GetTouch (SteamVR_Controller.ButtonMask.Touchpad))
    {
        Vector2 pos = SteamVR_Controller.Input(rightControllerId)
                            .GetAxis(EVRButtonId.k_EButton_SteamVR_Touchpad);
        Debug.Log("Rechtes Touchpad berührt: " + pos);
    }
}
```

In dem SteamVR-Beispielprojekt (SciFi-Stealth-Shooter) findest du ein weiteres praktisches Beispiel für das Auslesen von Tasten über das *SteamVR SDK*.

7.3.4 Haptisches Feedback

Die SteamVR-Integration bietet dir eine einfache Methode, um dein Spiel mit haptischem (Vibrations-)Feedback auszustatten. Die Methode dafür findest du ebenfalls in der *SteamVR_Controller*-Klasse. Wie schon für die Controller-Eingaben, benötigst du deshalb auch hier zunächst den Index des Controllers, der das haptische Feedback wiedergeben soll. Wenn du diesen kennst, kannst du die `TriggerHapticPulse`-Methode verwenden, um einen Vibrationsimpuls an den jeweiligen Controller zu schicken. Listing 7.9 demonstriert das an einem Beispiel.

Listing 7.9 Beispiel für die Methode TriggerHapticPulse

```
if(SteamVR_Controller.Input(leftControllerId)
       .GetPress(SteamVR_Controller.ButtonMask.Trigger))
{
       SteamVR_Controller.Input(leftControllerId).TriggerHapticPulse(800);
}
```

Der Parameter gibt die Impulsdauer in *Mikrosekunden* an. Je höher die Zahl, desto stärker wird die Vibration empfunden. Ein einziger Aufruf der Methode ergibt nur eine sehr kurze

Vibration. Für komplexe Vibrationsmuster, zum Beispiel für das Stampfen eines Dinosauriers, musst du mehrere Aufrufe dieser Methode kombinieren. Dafür bieten sich zum Beispiel *Coroutines* an.

7.3.5 Interaktionsbeispiel aus The Lab (u. a. Longbow, Teleport)

In dem Unterordner *InteractionSystem* findest du den Quellcode, Modelle, Sounds und Materials von einigen der besten Interaktionselementen aus der Valve Experience *The Lab*! Dazu zählen unter anderem der *Bogen* und die *Teleport*-Mechanik.

Falls du die kostenlose *The Lab*-Experience von Valve noch nicht ausprobiert hast, solltest du das auf jeden Fall noch tun. Es ist eine gute Erfahrung, welche die verschiedenen Spielkonzepte und Interaktionsmöglichkeiten demonstriert.

„The Lab“ im Steam Store:

http://store.steampowered.com/app/450390/

Die Komponenten, die du in diesem Ordner findest, sind extra so geschrieben, dass sie flexibel auch in andere Projekte eingebaut und nach Belieben verändert oder erweitert werden können. Innerhalb des Ordners findest du eine *InteractionSystem.pdf*-Datei, welche eine ausführliche Einführung und Dokumentation des *Interaction Systems* enthält. Nachdem du dir die Beispielprojekte dieses Buches angeschaut hast, solltest du dort auf jeden Fall auch hineinsehen. Durch das Wissen aus diesem Buch, zusammen mit ein wenig Englisch-Kenntnissen, solltest du dich auch ohne eine detaillierte Anleitung in das *Interaction System* einarbeiten können. Für Spiele, die nur das *SteamVR-SDK* verwenden und nicht mit weiteren Plattformen kompatibel sein sollen, ist das System ein sehr guter Start.

In dem Ordner *InteractionSystem/Samples/Scenes* findest du eine *Interactions_Example*-Scene, welche alle Funktionalitäten des Systems demonstriert. Wenn du anfängst, dich mit dem System zu beschäftigen, sollte dies dein erster Anlaufpunkt sein.

7.3.6 Mehr Infos und Hilfe

Für das SteamVR-Plug-in *für Unity* gibt es leider keine offizielle Dokumentation, sondern lediglich für die darunterliegende Schnittstelle. Die beste Anlaufstelle für Probleme ist deswegen das SteamVR Developers-Forum oder Reddit. Ich habe dir die wichtigsten Links in dem Kasten unten zusammengestellt. Zusätzlich findest du im Stammverzeichnis des Unity-Plug-ins eine *readme.txt* und eine *quickstart.pdf*, welche ebenfalls ein paar Tipps und Tricks enthalten.

SteamVR: wichtige Links und Dokumentation:

https://steamcommunity.com/app/358720/discussions/0/613956964584902849/

SteamVR Developers Forum:

https://steamcommunity.com/app/358720/discussions/

SteamVR Subreddit:

https://www.reddit.com/r/SteamVR/

SteamVR – Valve Developers Community:

https://developer.valvesoftware.com/wiki/SteamVR

7.4 GoogleVR SDK

Die Entwicklungswerkzeuge für GoogleVR benötigst du, wenn du ein Spiel für Google Daydream oder Cardboard entwickeln möchtest. Das GoogleVR Package erweitert die native Funktionalität unter anderem um Components, die es ermöglichen, den Daydream Controller zu verwenden. Außerdem bietet es, neben einige erweiterten Einstellungsmöglichkeiten, auch bessere Test- und Debug-Features.

7.4.1 Download und Import

In dem Kasten unten findest du den Download-Link zu der *GoogleVR SDK*-Version, die wir in diesem Buch besprechen werden. Außerdem findest du dort auch einen Link zu der offiziellen GoogleVR-Entwickler-Webseite, wo du immer die neuste Version herunterladen kannst, die sich aber eventuell von der hier besprochenen Version unterscheidet.

GoogleVR

Die im Buch beschriebene Version findest du auf der begleitenden Webseite (Version 1.60.0):

http://www.VRSpieleEntwickeln.de/zusatz/

Unter *Alle SDKs*, dann *GoogleVRForUnity_1.60.0.unitypackage*

Neuste Version von der Google-Developers-Seite:

https://developers.google.com/vr/unity/download

Lade das *Google VR Unity Package* herunter. Nach dem Download kannst du das *Unity Package*, wie im Quickstart-Kapitel beschrieben, in dein Projekt importieren.

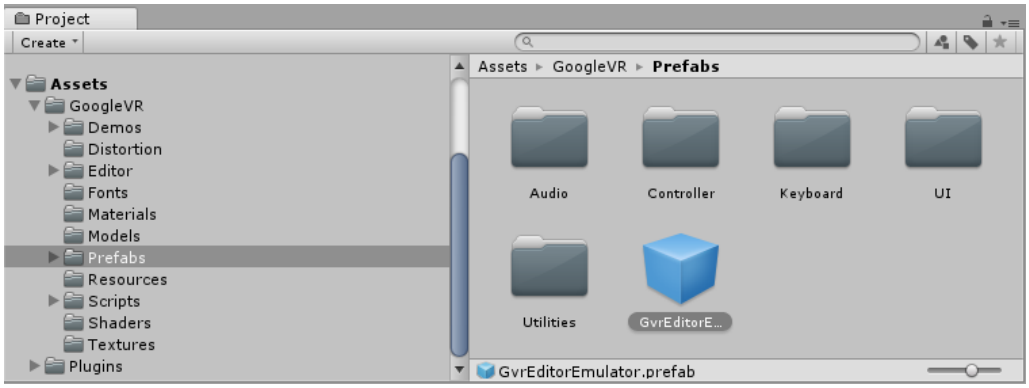

Bild 7.24 Nach dem Import solltest du in deinem *Project Browser* einen GoogleVR-Ordner sehen.

7.4.2 Prefabs

Das *GoogleVR Unity Package* enthält viele verschiedene *Prefabs*, von denen ich dir die wichtigsten gleich im Detail vorstellen werde. Die meiste Funktionalität wurde bereits in die native Unity-VR-Integration übertragen, weshalb für den Anfang nur wenige der Prefabs für uns interessant sind. Die meisten enthaltenen Prefabs stellen einheitliche Lösungen für verbreitete Probleme wie Audio und UI-Interaktion dar. Da diese Lösungen jedoch GoogleVR-spezifisch sind und sich eher an fortgeschrittene Entwickler richten, werde ich auf sie in diesem Buch nicht im Detail eingehen. Wenn du dich dafür interessierst, solltest du dir im Anschluss an dieses Buch die Beispiele im *Demos*-Ordner des GoogleVR-Packages genauer ansehen. In dem *Prefabs*-Ordner sind diese *Prefabs* nach verschiedenen Kategorien sortiert:

- **Audio** enthält die *Spatial Audio Engine* von Google.
- **Controller** enthält ein Prefab zur Verwendung des Controllers.
- **Keyboard** enthält Prefabs für eine VR-Bildschirmtastatur.
- **UI** enthält Prefabs, die es erlauben, Unitys UI-System mit dem Daydream Controller zu bedienen.
- **Utilities** enthält ein Prefab für eine FPS-Anzeige.

7.4.2.1 GvrControllerMain-Prefab

Dieses Prefab findest du in dem Unterordner *Prefabs/Controller*. Das *GvrControllerMain*-Prefab enthält Components, die sich mit dem Daydream Controller beschäftigen.

Die Components an diesem Prefab stellen zunächst nur alle Informationen zur Verfügung, die man benötigt, um beispielsweise ein 3D-Modell an der Position des realen Controllers darzustellen. Das Auslesen der Werte und die eigentliche Darstellung erfolgen jedoch über andere Components, wie zum Beispiel den in Kapitel 7.4.2.1.1 beschriebenen *GvrArmModelOffset*.

Zur Berechnung der Position wird standardmäßig ein *„Arm-Modell* verwendet. Das Arm-Modell simuliert einen unsichtbaren Arm und berechnet anhand des Winkels, in dem der reale Controller gehalten wird, eine plausible Position für die virtuelle Schulter, den virtuellen Ellenbogen und das virtuelle Handgelenk.

Das *Arm-Modell* wird benötigt, damit der Controller gut funktioniert, weil der Daydream Controller nur einen Beschleunigungssensor zur Positionserkennung nutzt. Er kann also Bewegungen nur anhand von Beschleunigungen und nicht über ein absolutes Tracking, wie bei der Oculus Rift oder HTC Vive, erkennen. Diese Art der Bewegungserkennung ist sehr ungenau, weshalb alternativ das *Arm-Modell* verwendet wird, um gute Ergebnisse erzielen zu können. Die Position des virtuellen Arms wird in erster Linie durch den Winkel des Controllers und die Parametern des Arm-Modells bestimmt. Die Werte des Beschleunigungssensors fließen standardmäßig gar nicht in die Berechnung mit ein, dies kann aber optional aktiviert werden.

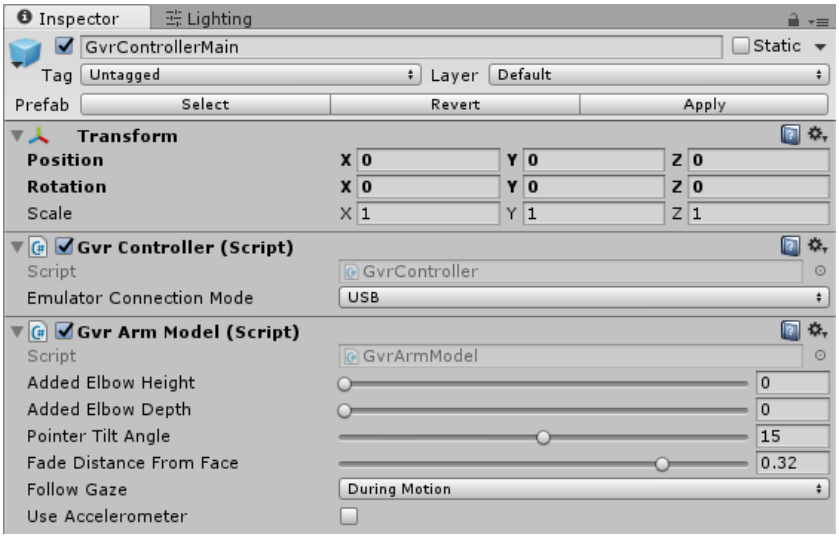

Bild 7.25 Das GvrControllerMain-Prefab im Inspector

Das in Bild 7.25 zu sehende *GvrController*-Component schauen wir uns in Kapiteln *7.4.3*, „Eingaben des Daydream Controllers lesen“, und *7.4.4.1*, „Controller Emulator für den Editor“, noch im Detail an.

Das ebenfalls sichtbare *GvrArmModel*-Component bietet dir einige Optionen, um das *Arm-Modell* deinen Wünschen nach anzupassen. Hier findest nun kurze Erklärungen zu den wichtigsten Optionen:

- **Added Elbow Height:** Hier kannst du die Höhe des simulierten Ellenbogens anpassen. Ein höherer virtueller Ellenbogen bewirkt einen höher gehaltenen virtuellen Controller.
- **Added Elbow Depth:** Hier kannst du den Abstand des simulierten Ellenbogens zum virtuellen Körper beeinflussen, dies resultiert darin, dass der virtuelle Controller weiter weggehalten wird.

- **Pointer Tilt Angle:** Der virtuelle Controller hat eine bestimmte Zeigerichtung. In der Standardkonfiguration weicht diese um 15° von der tatsächlichen „Nach-vorne-Richtung" des Controllers ab, weil sich das beim Bedienen von Menüs natürlicher anfühlt. Über diesen Parameter kannst du die Abweichung anpassen.
- **Follow Gaze:** Sollen sich die Schultern immer mit der Blickrichtung mitdrehen? *Never* sorgt dafür, dass sich die simulierten Schultern nie mit der Blickrichtung mit bewegen. *During Motion* bewirkt, dass sich die Schultern immer dann mitbewegen, wenn der Spieler läuft. *Always* sorgt dafür, dass sich die Schultern immer mitdrehen.

  Für dein Spiel empfiehlt es sich, die unterschiedlichen Varianten zu testen und die passendste zu wählen.
- **Use Accelerometer:** Hier kannst du aktivieren, dass auch die Daten des Beschleunigungssensors bei der Berechnung der Armposition berücksichtigt werden.

7.4.2.1.1 GvrArmModelOffsets-Component

Das *GvrArmModelOffsets*-Component ist ein einfaches Skript, um ein beliebiges GameObject an einer Position des Arm-Modells zu platzieren. Mögliche Punkte sind: Schulter, Ellenbogen, Handgelenk und Spitze des Controllers. Du kannst das Component einfach zu einem existierenden GameObject hinzufügen, damit es sich automatisch mit der jeweiligen angegebenen Position mitbewegt.

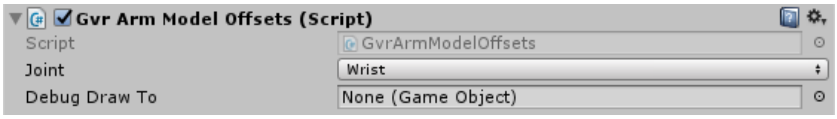

Bild 7.26 Das GvrArmModelOffset-Component im Inspector

Die Konfiguration des GameObjects ist sehr einfach, es gibt nur zwei Optionen:

- **Joint:** Hier kannst du auswählen, welchem Gelenk das GameObject folgen soll:

  Shoulder (dt. Schulter), *Elbow* (dt. Ellenbogen), *Wrist* (dt. Handgelenk) oder *Pointer* (Spitze des Controllers)
- **Debug Draw To:** Hier kannst du ein beliebiges GameObject angeben, zu dem in der *Scene View* eine Linie gezeichnet wird. Damit dieses Feature funktioniert, musst du zusätzlich ein *LineRenderer*-Component zu dem GameObject mit dem *GvrArmModelOffsets*-Component hinzufügen.

Zur Demonstration habe ich in Bild 7.27 vier Würfel angelegt und alle in dasselbe *Parent*-GameObject eingefügt. Jeder Würfel verfügt über ein *GvrArmModelOffsets*-Component und ein *LineRenderer*-Component, welche jeweils für eine der möglichen Positionen konfiguriert wurden.

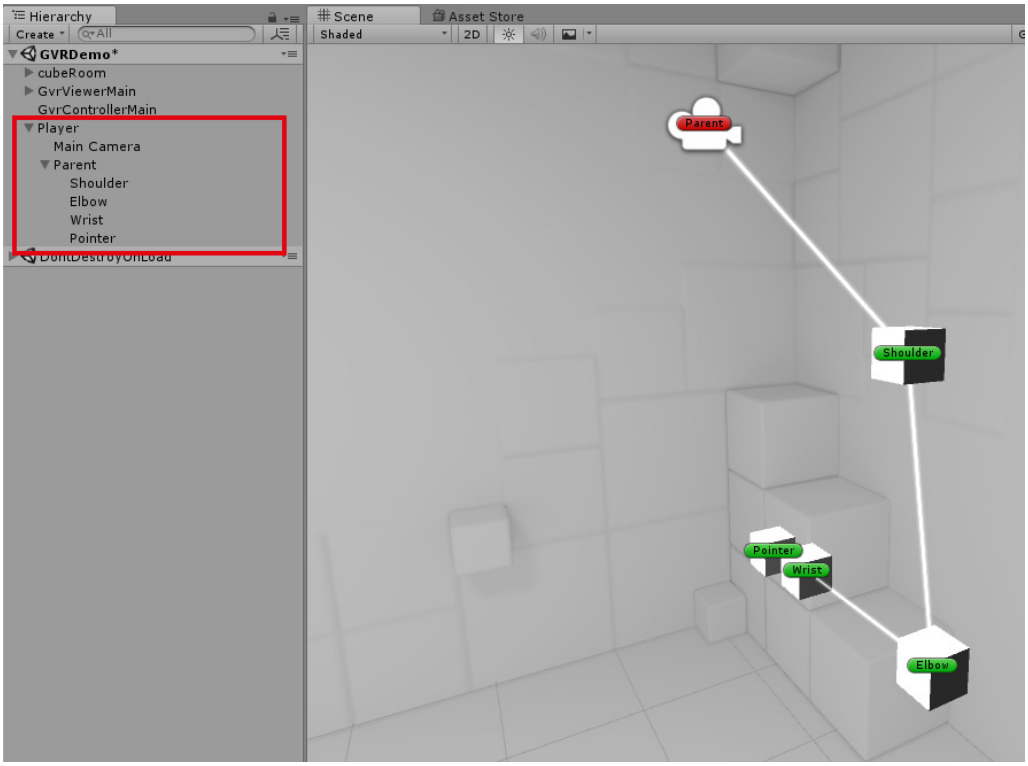

Bild 7.27 Ein Aufbau für ein Player-Objekt mit Camera und einem einfachen Arm

In dem Unterordner *Prefabs/Ui/* findest du ein Prefab mit dem Namen *GvrControllerPointer*. Dieses Beispiel-Prefab zeigt an der Position der virtuellen Hand ein 3D-Modell des Daydream Controllers. Außerdem schießt das Prefab einen Laserpointer, mit dem du mit Ui-Elementen interagieren kannst.

In dem Unterordner *Samples/Scenes/* findest du außerdem eine fertig konfigurierte Beispiel-Scene, welche auch das *GvrControllerPrefab* verwendet.

7.4.2.2 GvrFPSCanvas-Prefab

Dieses Prefab kannst du verwenden, um schnell eine FPS-Anzeige in dein GoogleVR-Spiel zu integrieren, um die Performance zu überwachen. Ein solches Prefab ist beim Entwickeln sehr hilfreich, weil du die Performance so stetig im Blick haben kannst, ohne die VR-Brille abnehmen zu müssen. Ziehe das Prefab, um die FPS-Anzeige zu verwenden, einfach in deine Scene, am besten als Kind der *Main Camera* oder deines Spieler-GameObjects. Das Prefab enthält eine Unity Ui. Achte darauf, dass die Anzeige so positioniert ist, dass du sie im Spiel sehen kannst, ohne dass sie zu sehr stört. Sobald du dein Spiel startest, zeigt der Text wie in Bild 7.28 die Frames per Second an. Darüber findest du außerdem die Zeit in Millisekunden, die benötigt wurde, um den letzten Frame zu berechnen.

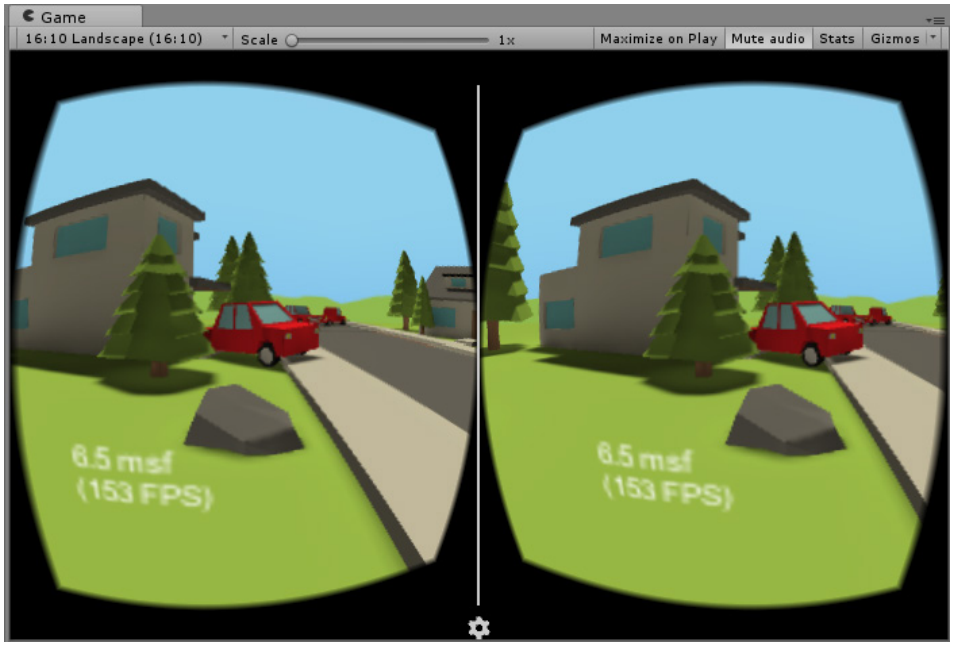

Bild 7.28 So könntest du die FPS-Anzeige zum Beispiel platzieren.

7.4.3 Eingaben des Daydream Controllers lesen

Der Daydream Controller ist einfach aufgebaut, und genauso einfach aufgebaut ist auch das Skript, um Eingaben von ihm lesen zu können. Google stellt dir dafür diverse *statische Eigenschaften* innerhalb der *GvrController*-Klasse zur Verfügung:

- **IsTouching** : `bool`

  TouchDown : `bool`

  TouchUp : `bool`

  Diese Eigenschaften funktionieren wie die *GetButton*-Methoden der Input-Klasse. `IsTouching` ist kontinuierlich `true`, solange der Spieler das Touchpad berührt. Die anderen Eigenschaften sind jeweils nur für einen Frame `true`, wenn das Touchpad zuerst berührt oder losgelassen wurde.

- **TouchPos** : `Vector2`

  Diese Eigenschaft gibt dir einen zweidimensionalen Vektor zurück, welcher die Touch-Position auf dem Touchpad beschreibt; wobei (0, 0) das Zentrum ist.

- **ClickButton** : `bool`

  ClickButtonDown : `bool`

  ClickButtonUp: `bool`

Mit diesen Eigenschaften kannst den aktuellen Zustand des „Click Buttons“ des Touchpads abfragen. Er wird aktiviert, wenn das Touchpad heruntergedrückt wird, anstelle es nur zu berühren.

- **AppButton** : `bool`

  AppButtonDown : `bool`

  AppButtonUp : `bool`

  Mit diesen Eigenschaften kannst du den aktuellen Zustand des „App Buttons“ abfragen. (Das ist der Button mit dem „Minus“-Zeichen).

- **Orientation**: `Quaternion`

  Diese Eigenschaft enthält die aktuelle Rotation des Controllers, welche über das eingebaute Gyroskop ausgelesen wurde.

- **Gyro** : `Vector3`

  Diese Eigenschaft enthält einen dreidimensionalen Vektor, der dir die direkten Daten des eingebauten *Gyroscopes* zur Verfügung stellt.

- **Accel** : `Vector3`

  Diese Eigenschaft enthält einen dreidimensionalen Vektor, der dir die Daten des *Beschleunigungssensors* zur Verfügung stellt.

Die Eigenschaften kannst du, wie in Listing 7.10 zu sehen, bequem aus einem Skript heraus abfragen, typischerweise erledigt man das auch bei dem Daydream Controller in der Update-Methode:

Listing 7.10 Beispiel zum Lesen von Eingaben des Daydream Controllers

```
void Update()
{
    if (GvrController.IsTouching) {
        Vector2 touchPosition = GvrController.TouchPos;
        Debug.Log("Berühre TouchPad bei: " + touchPosition);
    }
    if (GvrController.AppButtonDown) {
        Debug.Log("Aktuelle Gyro-Daten: " + GvrController.Gyro);
    }
}
```

7.4.4 Erweiterte Editorfunktionen fürs Testen

Eine Besonderheit bei der Entwicklung mit dem GoogleVR SDK ist, dass du die Apps zu einem gewissem Grad auch ohne eine VR-Brille im Editor testen kannst. Es sind dafür lediglich ein paar Hilfsobjekte notwendig:

Ziehst du zum Beispiel das Prefab *GvrEditorEmulator*, das du direkt im *Prefabs*-Ordner findest, in deine Scene, kannst du mit deiner Maus die Rotationen des Headsets simulieren und dich so in der virtuellen Welt umsehen. Die Position des GameObjects ist dabei egal, wichtig ist nur, dass das daran angehängte *GvrEditorEmulator*-Component irgendwo in der Scene vorhanden ist.

Sobald das Component in der Scene vorhanden ist, kannst du über die PLAY-Schaltfläche des Editors das Testen starten. Jetzt kannst du die ALT-TASTE auf deiner Tastatur gedrückt halten und dabei die Maus bewegen, um Bewegung des Headsets zu simulieren.

7.4.4.1 Controller Emulator für den Editor (Daydream)

Nun kannst du im Editor schon einen Teil deines Spiels testen. Damit du auch die Funktionen des Daydream Controllers im Editor testen kannst, stellt Google dir einen „Controller Emulator" bereit, der den Controller simuliert.

Leider hat Google diese Funktion nicht so realisiert, dass man einfach den Daydream Controller via USB anschließen und am PC verwenden kann. Stattdessen stellt Google eine spezielle *Controller Emulator-App* zur Verfügung, welche du auf deinem Smartphone installieren musst. Wenn sie auf deinem Smartphone läuft, kannst du es via WLAN oder USB mit dem Unity-Editor auf deinem Computer verbinden. Das Smartphone übernimmt dann die Rolle des Daydream Controllers. Die entsprechende App findest du im offiziellen GoogleVR GitHub Repository (siehe Kasten).

Die Daydream Controller Emulator App auf GitHub:

https://github.com/googlevr/gvr-android-sdk/blob/master/apks/controller_emulator.apk?raw=true

Rufe diesen Link direkt von deinem Smartphone auf oder kopiere die APK-Datei von deinem PC aus auf den Speicher deines Handys. Verwende dann einen Dateimanager, um die APK-Datei auf deinem Handy zu installieren.

Die einfachste Möglichkeit, dein Handy mit dem Editor zu verbinden, ist via USB. Achte darauf, dass an dem *GvrControllerMain*-GameObject in deiner Scene die Option *Emulator Connection Mode* auf *USB* steht. Starte dann die frisch installierte App auf deinem Smartphone und verbinde es via USB mit deinem Computer. Wenn du jetzt auf PLAY klickst, kannst du mit dem Lagesensor deines Smartphones und den auf dem Bildschirm dargestellten Tasten alle Funktionen des Controllers simulieren. Bild 7.29 zeigt, wie die Controller Emulator-App aussieht. Ganz oben findest du den aktuellen Verbindungsstatus.

Touchpad Click-Button verwenden

Um den Click-Button des Touchpads zu verwenden, musst du das Touchpad zweimal schnell hintereinander antippen.

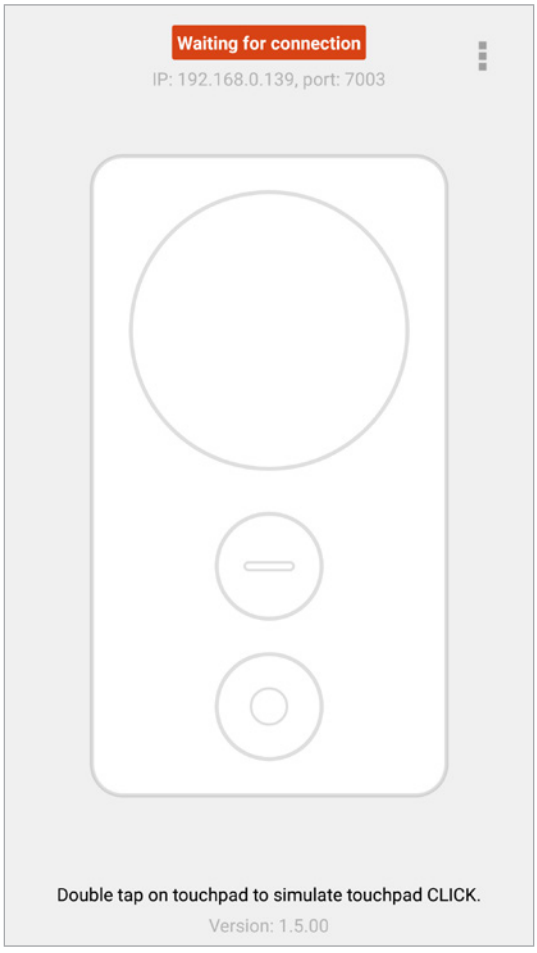

Bild 7.29 Über den Touchscreen deines Smartphones kannst du die Tasten und das Touchpad in der Controller Emulator-App bedienen.

Alternativ kannst du auch dein Smartphone über WLAN mit dem Editor verbinden. Stelle dazu den *Emulator Connection Mode* an dem *GvrControllerMain*-GameObject auf *WIFI*. Danach muss du das Skript *EmulatorConfig.cs* öffnen und dort ganz unten die Variable `WIFI_SERVER_IP` entsprechend der lokalen IP-Adresse deines Smartphones anpassen. Die benötigte Adresse wird praktischerweise in der *Controller Emulator*-App ganz oben unter dem aktuellen Verbindungsstatus angezeigt.

Als dritte und letzte Möglichkeit besteht noch die Option, den echten Daydream Controller mit deinem Smartphone zu verbinden und diesem zu sagen, er soll alle Eingaben an den Editor weiterleiten. Dazu musst du in der App zunächst oben rechts auf die DREI PUNKTE klicken und dann in dem Menü SWITCH TO REAL CONTROLLER auswählen. Auf deinem Smartphone werden nun entsprechende Hinweise zur Verwendung das echten Daydream Controllers angezeigt. Folge ihnen, um den Controller zu verbinden. Wenn alles geklappt hat, solltest du dann auf deinem Bildschirm den Inhalt von Bild 7.30 sehen. Lass das Fenster auf

deinem Smartphone geöffnet, solange du testen möchtest. Wenn du jetzt deine App im Editor testest, kannst du den echten Daydream Controller verwenden, um den virtuellen Controller zu steuern.

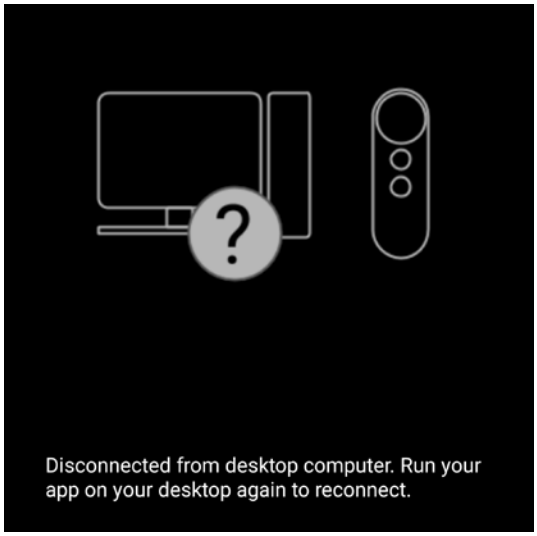

Bild 7.30 Wenn du diesen Bildschirm siehst, ist alles bereit zum Testen.