

6

Erweiterte Unity- Einführung

Nachdem du die letzten Kapitel durchgearbeitet hast, solltest du jetzt bereits ein Grundverständnis für Unity haben und dich ganz gut zurechtfinden. Du kennst jetzt den Inspector, kannst einfache Scenes erstellen, mit GameObject und Prefabs arbeiten, Components zu GameObjects hinzufügen, sie konfigurieren und sogar eigene Components erstellen. In diesem Kapitel gehen wir nochmals im Detail auf einige wichtige Unity-Funktionen ein, die du auf jeden Fall in deinem Spiel brauchen wirst. Einige der Funktionen wurden in den vergangenen Kapiteln bereits angeschnitten, hier folgen aber ein paar mehr Informationen, die du benötigen wirst, sobald du später selbstständig an Spielen arbeitest und dich nicht mehr an Anleitungen entlanghangelst. Dieses Kapitel soll dir, wie die meisten Kapitel in diesem Buch, auch als Nachschlagewerk dienen. Falls du Fragen bezüglich einer Unity-Funktion hast, kannst du also zu diesen Kapiteln zurückkommen. Für die Beispielprojekte am Ende des Buches erwarte ich nicht, dass du jede in diesem Kapitel besprochene Funktion auswendig kennst, jedoch werde ich in diesem Kapitel beschriebene Funktionen als bekannt voraussetzen, da du ja jederzeit hierher zurückkommen kannst, um Dinge, an die du dich nicht mehr genau erinnern kannst, nachzulesen.

■ 6.1 Licht, Schatten und Lightmapping

Wenn dein Spiel optisch beeindrucken soll, solltest du möglichst früh einige wichtige Entscheidungen bezüglich der Lichter in deiner Scene treffen, die den Look und die Performance deines Spiels stark beeinflussen können. In diesem Kapitel erfährst du, welche das sind und was du beachten solltest.

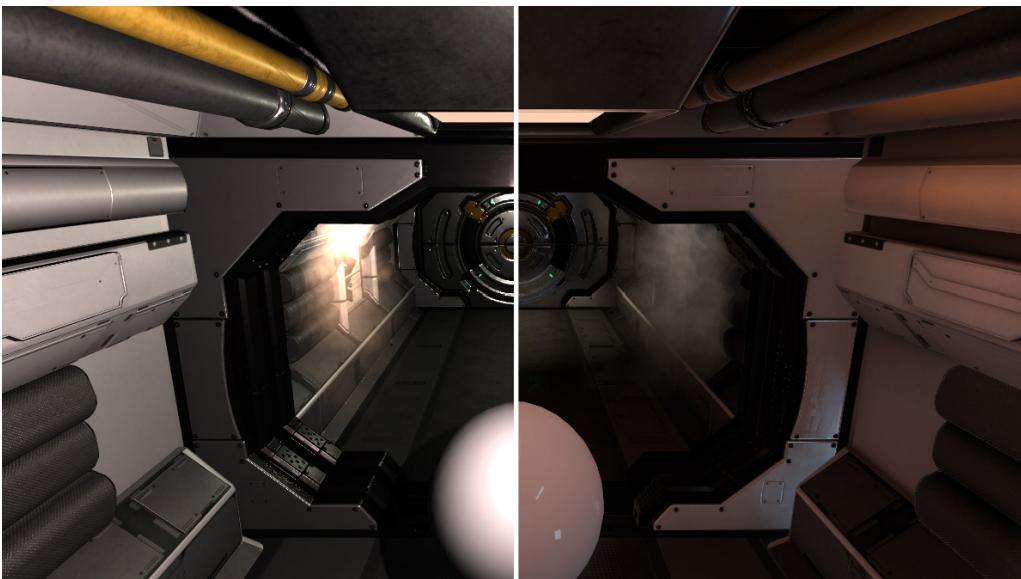


Bild 6.1 Links eine einfache Beleuchtung ohne die in diesem Kapitel besprochenen Techniken, rechts die Optik mit den Techniken

6.1.1 Global Illumination

Moderne Spiele-Engines erreichen ihre gute Optik mit einer Reihe von Techniken, welche unter dem Begriff *Global Illumination* oder kurz *GI* zusammengefasst werden. Ziel dieses Verfahrens ist es, das komplexe Verhalten von Licht realistisch zu berechnen. Dazu gehört zum Beispiel, wie das Licht von Objekten reflektiert oder absorbiert wird. *Global Illumination* akkurat zu berechnen, benötigt sehr viel Rechenpower, weshalb dies nicht ohne Weiteres in Echtzeit möglich ist. Die Spiele-Engines versuchen daher, über verschiedene Ansätze möglichst viele dieser Berechnungen im Vorfeld beim Erstellen des Spiels durchzuführen und nicht erst während das Spiel gespielt wird.

Bild 6.2 zeigt dieselbe Scene ohne Lichtberechnung, nur mit direktem Licht und mit *Global Illumination*. Neben dem indirekten Licht lässt sich gut sehen, wie das von der roten Kugel reflektierte Licht den Raum rötlich färbt.

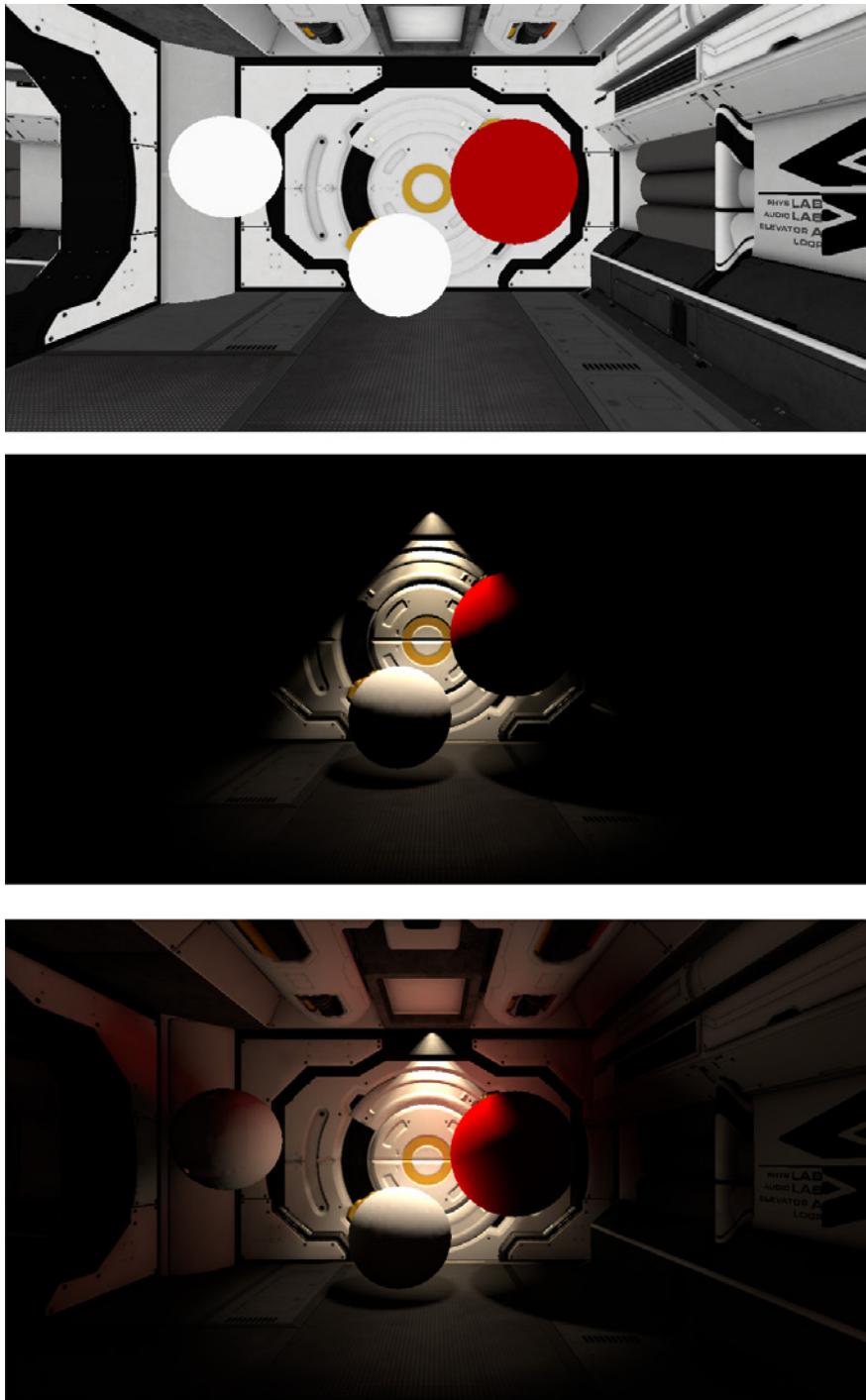


Bild 6.2 Dieselbe Scene ohne Lichtberechnung (oben), nur mit direktem Licht (mittig) und mit *Global Illumination* (unten)

Unity unterstützt verschiedene Arten der Lichtberechnung. An dieser Stelle möchte ich dir die reine Echtzeit-Lichtberechnung ohne Global Illumination (*Realtime*), die vorberechnete Lichtberechnung mit Global Illumination (*Mixed Lighting mit Baked GI*) und die Echtzeit-Lichtberechnung mit vorberechneter Global Illumination (*Realtime GI*) vorstellen.

6.1.1.1 Realtime Lightning ohne GI

Wenn du neue Lichter erzeugst und keine weiteren Einstellungen an diesen vornimmst, werden die meisten davon als *Realtime*-Lichter erzeugt. Das bedeutet, das Licht, das diese Lichter in die Scene werfen, wird für jeden Frame neu berechnet. Wenn sich *GameObjects* oder Lichter bewegen, wird die Lichtsituation sofort („in Echtzeit“) aktualisiert.

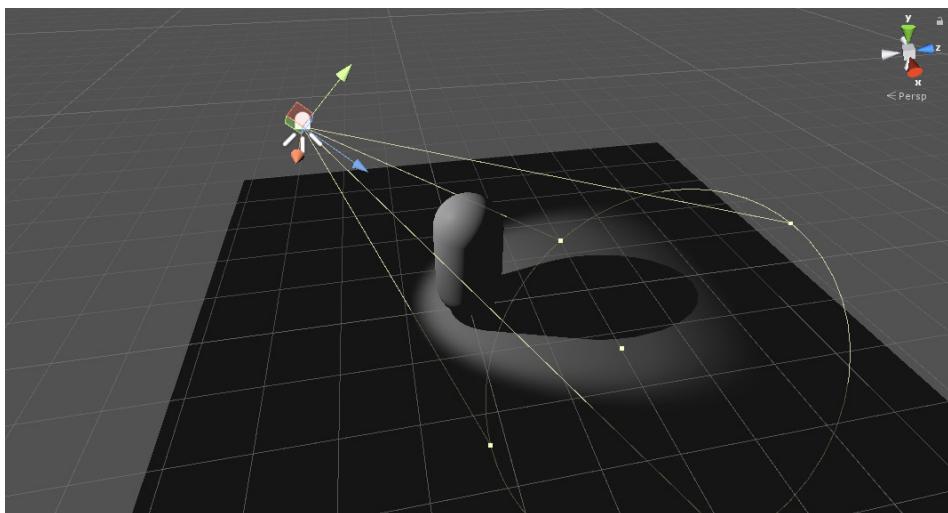


Bild 6.3 Realtime-Lichter erzeugen harte, sehr dunkle Schatten.

Realtime Lighting ist die einfachste Möglichkeit, Objekte in der Scene zu erleuchten, und ist praktisch, um zum Beispiel den Spieler oder andere bewegliche Objekte zu erhellen.

Bei Realtime-Lichtern gibt es jedoch keine „Bounces“. Das bedeutet, das Licht wird nicht von erleuchteten Objekten auf andere Objekte reflektiert, wie es in der Realität der Fall ist. Es werden also nur Objekte erleuchtet, die direkt von dem Licht angestrahlt werden. Aus diesem Grund sind Schatten von Echtzeit-Lichtern immer vollständig schwarz, wenn keine andere Lichtquelle ebenfalls auf die Stelle des Schattenwurfes strahlt.

Um durch *Global Illumination* eine realistischere Lichtstimmung zu erzeugen, musst du zusätzlich *Realtime GI* aktivieren, welches ich dir noch vorstellen werde.

6.1.1.2 Baked Global Illumination

Baked GI nutzt für die Darstellung des Lichtes in der Scene sogenannte *Lightmaps*. Bei der Berechnung einer *Lightmap* wird die Auswirkung von allen Lichtern auf alle statischen Objekte in der Scene berechnet und die Ergebnisse werden in Texturen (den Lightmaps) gespeichert, welche anschließend über die Geometrie der Scene gelegt werden. Auf diese

Weise wird die Scene erleuchtet, ohne dass Lichtberechnungen während des eigentlichen Spielens notwendig sind.

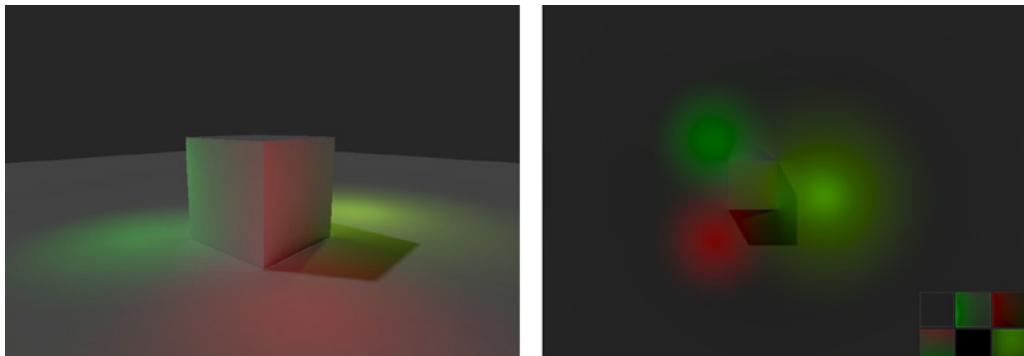


Bild 6.4 Links: Eine einfache Scene mit *Lightmap*, rechts: die dazugehörige *Lightmap*. Sowohl Licht als auch Schatten sind in derselben Grafik gespeichert.

Lightmaps können sowohl direktes Licht als auch indirektes Licht, das von anderen Objekten reflektiert wurde, speichern. Da das Licht in die *Lightmap* „gebacken“ wurde, aktualisiert es sich nicht, wenn sich zur Laufzeit Objekte oder Lichter bewegen, sind sie statisch. Es werden deshalb nur *GameObjects* in die *Lightmap* aufgenommen, die als *Static* markiert wurden. Würden auch dynamische Objekte in die *Lightmap* aufgenommen werden, würden ihre Schatten dauerhaft an derselben Stelle angezeigt werden, auch wenn das 3D-Modell gar nicht mehr da ist.

Du kannst zusätzlich zu den *Lightmaps* Realtime-Lichter verwenden, deren Licht über das vorberechnete Licht fällt („Mixed Lighting“).

6.1.1.3 Realtime Global Illumination

Während traditionelle *Lightmaps* sich nicht an ändernde Lichtbedingungen anpassen können, bietet *Realtime Global Illumination* eine Technik, mit der auch komplexe Lichtsituationen interaktiv verändert werden können. Diese Technik erlaubt es dir also, deine *Scene* mit *Global Illumination* und reflektiertem Licht zu beleuchten und trotzdem noch die Lichter zu bewegen oder ihre Leuchteigenschaften wie Reichweite und Farbe zur Laufzeit zu verändern.

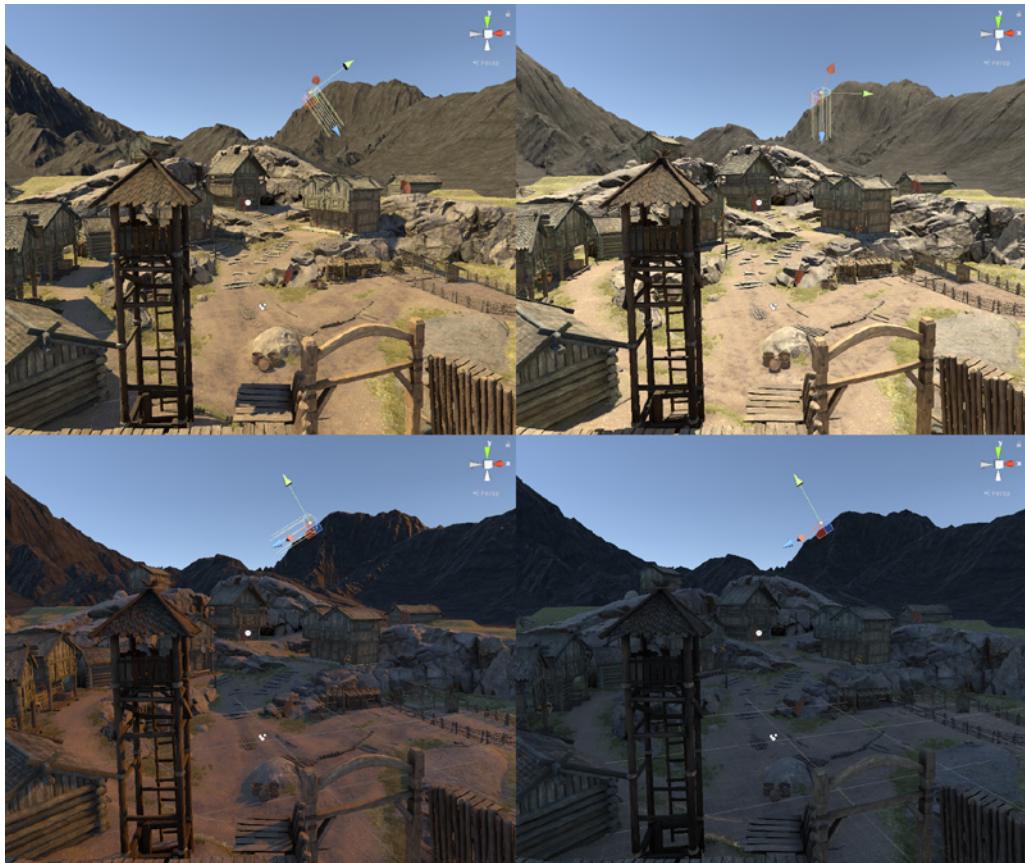


Bild 6.5 Ein Tag-und-Nacht-Zyklus, realisiert mit Realtime GI. Die Sonne bewegt sich flüssig über das Dorf und verändert dabei ihre Farbe. Die Lichtstimmung und die Schatten passen sich dynamisch an, ohne dass eine andere Scene geladen werden muss.

Aber wie kann diese Methode *Global Illumination* in Echtzeit berechnen, wenn wir doch in den Kapiteln zuvor gelernt haben, dass die Berechnungen dafür zu aufwendig sind? Die einfache Antwort ist, dass „Realtime Global Illumination“ in Wirklichkeit nicht vollständig *Realtime* ist, sondern einige aufwendige Berechnungen im Vorfeld ausgeführt werden. In älteren Unity-Versionen nannte sich diese Variante daher noch *Precomputed Realtime GI*. Da also einige Berechnungen im Vorfeld vorgenommen werden müssen, kann *Realtime GI*, genauso wie *Baked GI*, die Globale Illumination nur für *GameObjects* berechnen, die als *Static* markiert wurden. Dynamische Objekte werden deswegen zum Beispiel nicht von reflektiertem Licht erhellt, sondern nur von dem direkten Licht der Lichtquelle.

Bei *Realtime Global Illumination* wird das direkte Licht weiterhin ganz normal berechnet und nur für das indirekte Licht werden einige Optimierungen vorgenommen, damit dieses überhaupt angezeigt werden kann. Da indirektes Licht im Gegensatz zu direktem Licht kaum harte Kanten aufweist, sondern eher verwaschene, weiche Übergänge besitzt, kann hier die Auflösung für die Berechnung, im Vergleich zum direkten Licht, deutlich verringert werden. Beim Berechnen der *Realtime GI* erzeugt Unity eine niedrig aufgelöste Version der

Geometrie in deiner Scene und teilt sie in viele kleine Bereiche, welche sich „Clusters“ nennen.

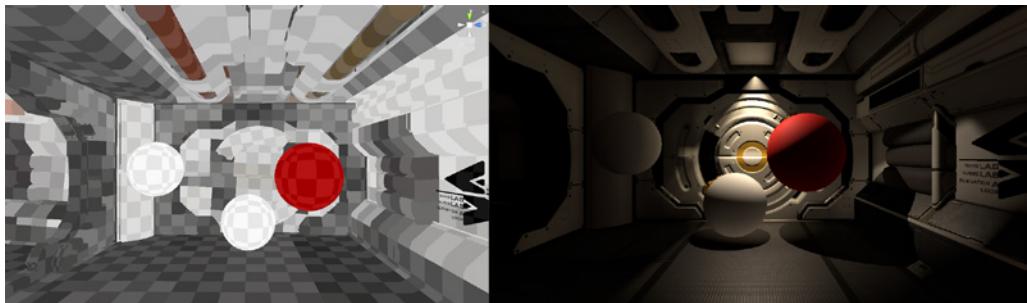


Bild 6.6 Links: Stellt man die *Scene View* auf *Realtime GI Albedo*, sieht man die generierten *Texels*, welche grob den Clustern entsprechen. Rechts: die gleiche Scene mit *Realtime GI*, wie sie im Spiel sichtbar wäre.

Bei der traditionellen Berechnung von indirektem Licht werden (Licht-)Strahlen in die Scene geschossen und ausgerechnet, wo sie wie auf welches Objekt treffen und wie sie dann auf welche Objekte reflektiert werden. Diese Verfolgung der reflektierten Strahlen ist, was die Berechnung des indirekten Lichts so aufwendig macht. Bei Realtime Global Illumination wird diese Strahlenverfolgung für alle Cluster ebenfalls im Vorfeld berechnet, ohne dabei konkrete Lichtwerte auszurechnen. Anstelle von einer Lightmap entsteht hier ein großes Netz, in dem die Cluster, die Licht aufeinander strahlen können, miteinander verbunden sind. Vereinfacht dargestellt: Wird *Cluster A* von direktem Licht getroffen, kann durch das Netz schnell herausgefunden werden, dass *Cluster B* zu 50% von indirektem Licht getroffen wird und *Cluster C* zu 25% und so weiter. Es muss also nicht jeder Lichtstrahl aufwendig verfolgt werden, um herauszufinden, welche Objekte Licht auf welche anderen Objekte reflektieren, weil diese Informationen bereits berechnet wurden.

„Echtzeit“ gibt es allerdings nicht umsonst: Das Ergebnis ist nicht identisch mit dem von traditionellem *Baked GI*, dessen Berechnung sehr lange dauern kann. Es fehlt zum Beispiel die *Final Gather*-Funktion, welche die optische Qualität nochmals deutlich erhöht. Bild 6.7 zeigt die beiden Varianten im Vergleich.

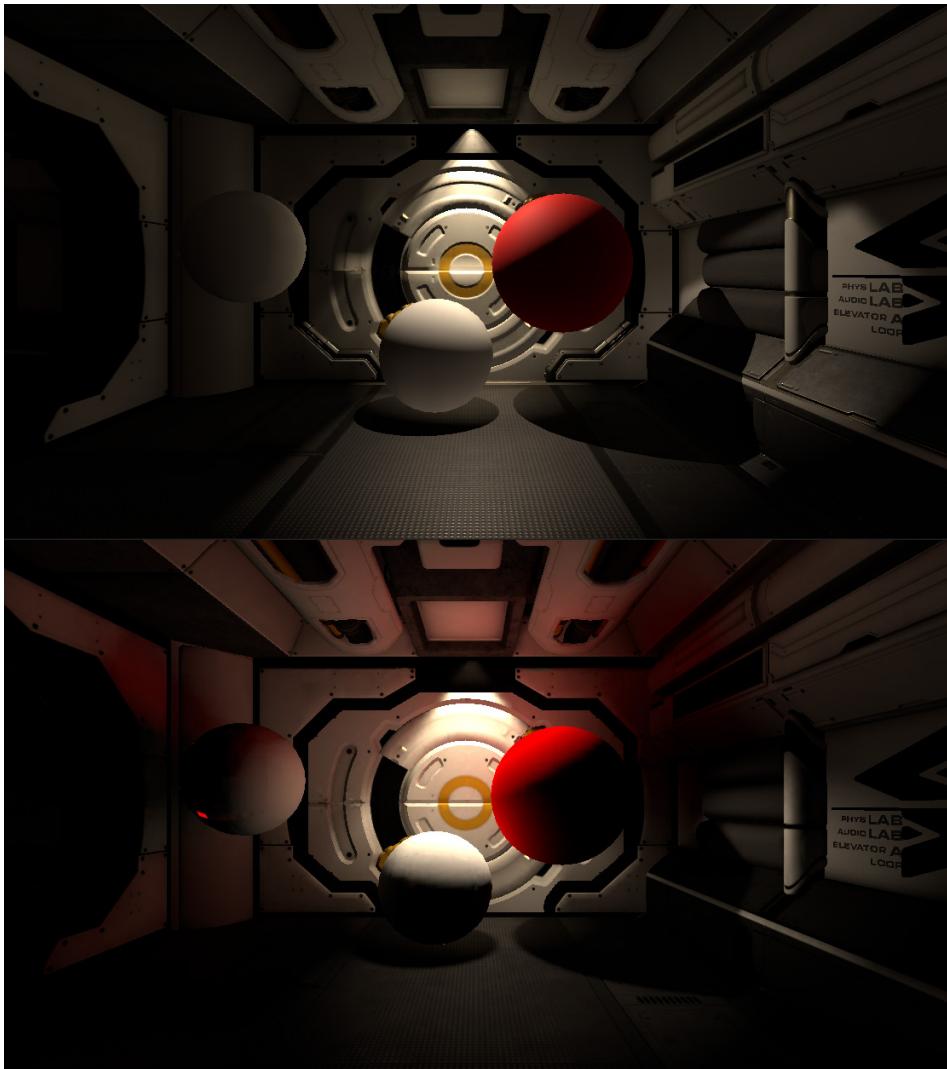


Bild 6.7 *Realtime GI* (oben) und *Baked GI* mit *Final Gather* (unten)

In der Praxis fällt der Unterschied zwischen den beiden nicht unbedingt so stark auf wie in dieser Testscene, welche gezielt dafür erstellt wurde, die Unterschiede hervorzuheben. Bei einer Außenszene, wo alle Objekte ohnehin eine ähnliche Farbgebung haben, ist der Unterschied zum Beispiel kaum zu sehen. Durch Optimierung der Einstellungen und der Scene lässt sich zudem auch stets noch mehr aus beiden Techniken herausholen.

6.1.1.4 Welche Technik ist die beste?

Auf diese Frage gibt es keine universale Antwort. Welche Technik du für dein Spiel verwenden solltest, hängt von verschiedenen Faktoren ab. Wenn du ein Spiel für eine mobile Plattform entwickelst, ist es wahrscheinlich am besten, ausschließlich *Baked GI* zu verwenden,

da dies die performanteste Lösung ist. Ist deine Zielhardware ein moderner PC, empfiehlt es sich, je nach Spiel, *Realtime GI* zu verwenden. Da das Ergebnis allerdings nicht perfekt ist, gibt es auch die Möglichkeit, eine Kombination aus *Realtime GI* und *Baked GI* zu verwenden. Beide Systeme gleichzeitig zu verwenden, kostet jedoch viel Performance und auch viel Arbeitsspeicher, da die *Lightmap*-Daten für beide Systeme jederzeit verfügbar sein müssen. Gerade wenn du ein VR-Spiel entwickelst, wo Performance sehr kritisch ist, solltest du also gut abwägen, ob du wirklich beide Systeme gleichzeitig benötigst.

Neben der Performance hängt es natürlich auch von deiner Spielidee und der vorhandenen Lichtsituation ab, welche Technik die geeigneter ist. Für dunkle Horror-Spiele würde sich *Realtime GI* zum Beispiel besser eignen, da hier flackernde Lichter und eine Taschenlampe gut realisiert werden können. *Realtime GI* eignet sich auch sehr gut, um einen Tag-Nacht-Wechsel leicht zu realisieren. Ist die Umgebung in deinem Spiel wiederum gut ausgeleuchtet und die Lichtsituation bleibt ohnehin die ganze Zeit über gleich, empfiehlt es sich, *Baked GI* zu verwenden und die gesparte Performance an anderer Stelle zu investieren.

6.1.1.5 Baked GI oder Realtime GI aktivieren

Bei einem neuen Unity-Projekt ist sowohl *Baked GI* als auch *Realtime GI* aktiviert. In diesem Fall werden abhängig vom Lichttyp (Echtzeit, vorberechnet oder gemischt) beide Arten der *Global Illumination* berechnet. Da beide Modi gleichzeitig jedoch unnötig Rechenleistung kosten, solltest du dich in den meisten Fällen für eine der beiden *Global-Illumination*-Varianten entscheiden und die andere in den *Lighting*-Einstellungen deaktivieren.

In dem *Lighting*-Fenster unter **WINDOW/LIGHTING/SETTINGS** findest du die jeweiligen Optionen in den Kategorien *Realtime Lighting* und *Mixed Lighting*. Wenn du *Baked GI* verwendest, kannst du weiterhin auch Echtzeit und gemischte Lichter einsetzen, allerdings wird für die Echtzeit-Anteile kein Global Illumination berechnet. Diese Mischung von „Baked“ und „Realtime“ nennt sich *Mixed Lighting*. Wenn du davon Gebrauch machen möchtest, solltest du einen für deine Zielplattform passenden *Lighting Mode* auswählen. Die einzelnen Modi stelle ich dir in dem folgenden Abschnitt vor.

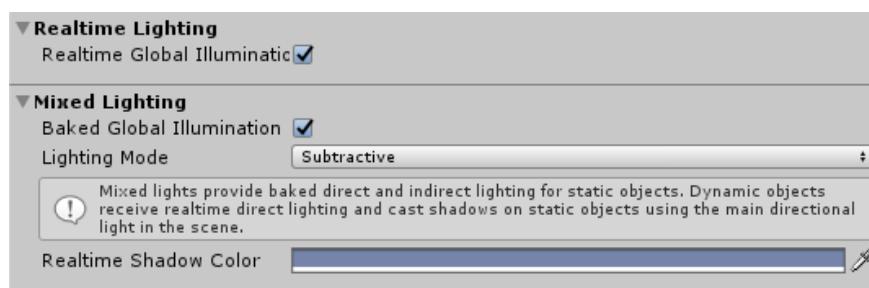


Bild 6.8 Über die Checkboxen kannst du die verschiedenen Global-Illumination-Varianten aktivieren und deaktivieren.

6.1.1.5.1 Mixed Lighting – Lighting Mode wählen

Im Grunde bestimmen die Lighting Modes, welche Techniken verwendet werden, um das Licht der Echtzeit-Lichtquellen und der vorberechneten Lichtquellen zu einem Gesamt-

ergebnis zusammenzurechnen. Der *Lighting Mode* bestimmt auch, wie viele Daten vorberechnet und wie viele in Echtzeit berechnet werden.

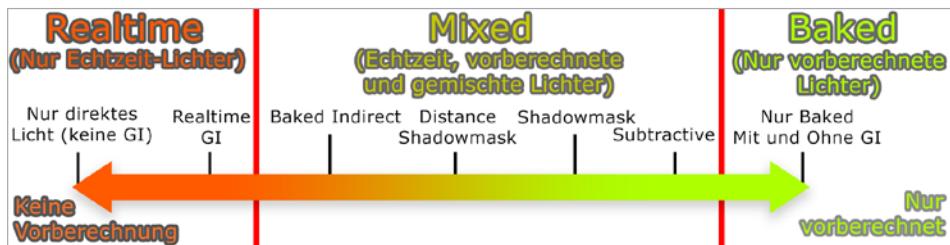


Bild 6.9 Die einzelnen Lighting-Optionen und -Modi sortiert nach ihrem Anteil an Echtzeit-Berechnungen

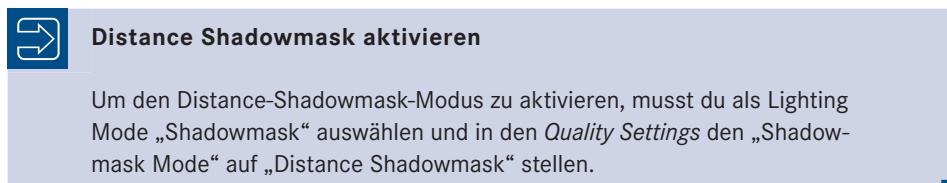
Jeder der *Mixed Lighting*-Varianten werde ich dir nun kurz vorstellen. Für eine exakte Auf-listung aller Vorteile und Einschränkungen der einzelnen Möglichkeiten empfehle ich einen Blick in die Unity-Dokumentation¹.

- **Baked Indirect:** Indirektes Licht wird vorberechnet, alle Schatten werden innerhalb der *Shadow Distance*² vollständig in Echtzeit berechnet und angezeigt. Außerhalb der *Shadow Distance* werden keine Schatten dargestellt.
 - **Hardware-Anforderung:** Medium (PC, High-End Mobile)
 - **Beispielanwendung:** Spiele/Level, die innerhalb von Gebäuden spielen, da hier die Sichtweite durch Wände etc. gering gehalten wird und somit die fehlenden Schatten hinter der *Shadow Distance* nicht auffallen
- **Shadowmask:** Für statische Objekte werden die Schatten in einer *Shadowmask* vorberechnet und gespeichert. Diese statischen Schatten sind bis zum Horizont sichtbar. Bewegliche Objekte werfen innerhalb der *Shadow Distance* Schatten auf andere bewegliche Objekte und statische Objekte. Statische Objekte werfen allerdings *keine* Schatten auf dynamische Objekte – hier muss mit *Light Probes* (Kapitel 6.1.6) nachgeholfen werden.
 - **Hardware-Anforderung:** Medium (PC, High-End Mobile)
 - **Beispielanwendung:** Weitläufige Welten, bei denen Schatten bis zum Horizont angezeigt werden sollen und bei denen kein Tag-Nacht-Wechsel stattfindet. Alternativ auch Scenes, die zu einem sehr großen Teil statisch sind und über detaillierte Texturen verfügen.
- **Distance Shadowmask:** Dieser Modus erweitert den einfachen *Shadowmask*-Modus mit der Funktionalität, dass auch ohne *Light Probes* statische Objekte Schatten auf dynamische Objekte werfen können, zumindest innerhalb der angegebenen *Shadow Distance*. Darüber hinaus funktioniert dieser Modus genauso wie *Shadowmask*. Dieser Modus ist der schönste Modus, benötigt aber auch die stärkste Hardware. Beachte den Hinweis-Kasten unten, wenn du diesen Modus verwenden möchtest.

¹ Hier findest du alle Details zu den einzelnen Mixed Lighting Modes: <https://docs.unity3d.com/Manual/LightMode-Mixed.html>

² Die *Shadow Distance* kann in den *Quality Settings* bestimmt werden und beschreibt die maximale Distanz zur Kamera, in der noch Echtzeit-Schatten angezeigt werden.

- **Hardware-Anforderung:** Hoch (High-End-PC, PlayStation 4, Xbox One etc.)
 - **Beispielanwendung:** Weitläufige Welten, bei denen Schatten bis zum Horizont angezeigt werden sollen und komplexe statische Objekte (Häuser, Berge, Bäume) Schatten auf bewegliche Spielfiguren und andere dynamische Objekte werfen sollen.
 - **Subtractive:** Dieser Modus ist sehr ressourcenschonend. Hier werden das indirekte Licht sowie statische Schatten in einer *Lightmap* gespeichert. Alle weiteren Informationen, die in anderen *Lighting Modes* verwendet werden, um nachträglich noch korrekte Echtzeit-Schatten oder Glanzeffekte auf diese statischen Objekte werfen zu können, werden verworfen. Durch das Verwerfen dieser Informationen wirken die Oberflächen häufig matt. Echtzeit-Schatten von dynamischen Objekten werden nur für ein einziges *Directional Light* berechnet. Statische Objekte werfen zudem keine Schatten auf dynamische Objekte, weshalb mit *Light Probes* (Kapitel 6.1.6) nachgeholfen werden sollte.
 - **Hardware-Anforderung:** Gering (Mobile)
 - **Beispielanwendung:** Spiele mit einfacher Optik und sehr wenigen dynamischen Objekten. Subtractive ist sehr gut geeignet für Spiele im Toon-Stil.



6.1.1.6 Der Lightmapping-Vorgang

Der *Lightmapping*-Vorgang, bei dem indirektes Licht, *Realtime GI* und *Baked GI* durch den Editor berechnet werden, startet standardmäßig automatisch und muss nicht per Hand gestartet werden. Bei komplexen Scenes kann dieser Vorgang sehr lange dauern. Während die Lichtberechnung läuft, siehst du rechts unter dem *Inspector*-Fenster eine Fortschrittsanzeige, die dich darüber informiert, wie weit die Berechnung fortgeschritten ist. Zusätzlich wird in dem Balken angezeigt, welcher Schritt gerade berechnet wird.

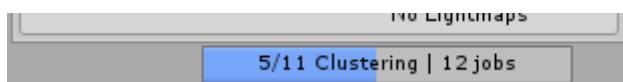


Bild 6.10 Unten rechts im Editor findest du die Fortschrittsanzeige für den Lightmapping-Vorgang.

Ob das *Lightmapping* automatisch gestartet wird, bestimmt die *Auto Generate*-Checkbox in den *Lighting*-Einstellungen ([WINDOWS/LIGHTING/SETTINGS](#)). Ist der automatische Modus aktiviert, erkennt Unity jede Änderung an statischen Objekten in der Scene und startet sofort einen neuen *Lightmapping*-Vorgang. Wenn bereits eine Berechnung im Gange ist, wird sie abgebrochen und eine neue gestartet. Da der *Lightmapping*-Vorgang einiges an Prozessorleistung und Arbeitsspeicher verbraucht, kann die Berechnung im Hintergrund störend sein und dich am Weiterarbeiten hindern. Um dies zu umgehen, kannst du das automatische Starten des *Lightmapping*-Vorgangs deaktivieren und ihn über die [GENERATE](#)

LIGHTING-Schaltfläche per Hand starten, wenn es dir gerade passt; zum Beispiel, wenn du ohnehin gerade eine Pause machst oder erst an einigen Scripten und nicht an der Scene arbeitest.

Name	CPU	Arbeits...	Datenträ...	Netzwerk
Unity Job Process	91,7%	1.155,2 MB	0,1 MB/s	0 MBit/s

Bild 6.11 Die Lichtberechnung läuft auf allen Kernen deines Prozessors und lastet ihn dadurch häufig vollkommen aus. Der Arbeitsspeicher-Verbrauch ist abhängig von der gewählten Auflösung für die Lightmaps (in diesem Fall „Very Low“).



Statische Geometrie

Da für das vorberechnete Licht nur statische Geometrie berücksichtigt wird, solltest du alle GameObjects, die sich nicht bewegen müssen, als *Static* markieren.

6.1.1.7 Global Illumination Cache

Um das Lightmapping zu beschleunigen, speichert Unity die Ergebnisse der *Baked-GI*- und *Realtime-GI*-Berechnungen im *GI Cache*. Wenn du etwas an deiner Scene änderst, muss mit der Berechnung dann nicht bei 0 angefangen werden, sondern viele Berechnungen für unveränderte Objekte können übersprungen werden, da die Ergebnisse bereits im *GI Cache* vorhanden sind und von dort gelesen werden können. Die Daten des *GI Caches* liegen nicht in deinem Projekt-Ordner, sondern standardmäßig in einem versteckten Verzeichnis auf deiner *Hauptfestplatte*. In den *GI-Cache*-Einstellungen kannst du jedoch auch einen anderen Speicherort angeben.

Die GI-Cache-Einstellungen findest du unter: **EDIT/PREFERENCES.../GI CACHE**. Dort kannst du die maximale Größe, die standardmäßig 10 GB beträgt, erhöhen oder verringern. Über die Schaltfläche **CLEAR CACHE** kannst du den Zwischenspeicher leeren. Erzeugt das Lightmapping immer wieder fehlerhafte Ergebnisse, ist vielleicht eine Datei im Cache beschädigt und ein Leeren das Caches könnte dieses Problem beheben.

6.1.2 Farträume

Unity unterstützt zwei unterschiedliche Farträume *Linear* und *Gamma*. Hier kann ausnahmsweise vollkommen objektiv geurteilt werden, dass der *Linear Color Space* der schöneren der beiden ist, da er realistischere Ergebnisse liefert als der *Gamma Color Space*.

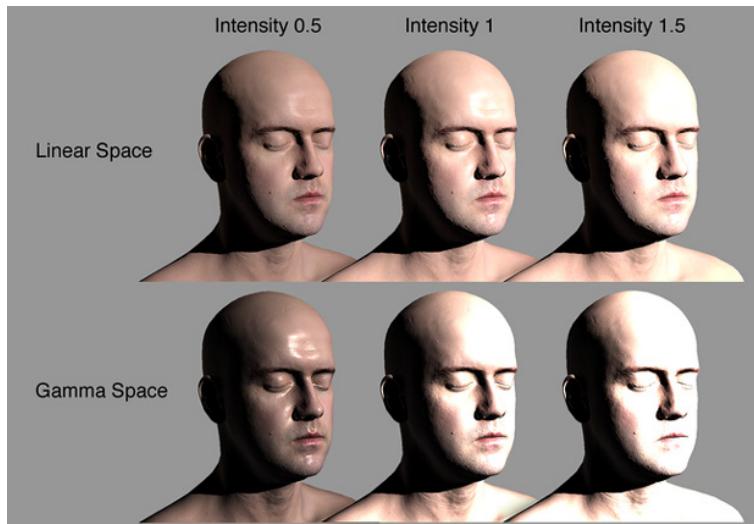


Bild 6.12 Der Linear und der Gamma Color Space im Vergleich (Quelle: Unity Technologies)

Wie in Bild 6.12 zu sehen, liefert der *Linear Color Space* ein schöneres und deutlich realistischeres Ergebnis, während die Farben beim *Gamma Color Space* sehr schnell zu einem einheitlichen Weiß werden.

Warum sollte man also überhaupt den Gamma Color Space verwenden?

Leider wird der *Linear Color Space* nicht auf jeder Hardware unterstützt, weshalb der *Gamma Color Space* auch die Standardeinstellung in einem neuen Unity-Projekt ist.

Am PC, auf neueren Smartphones und der aktuellen Konsolengeneration kannst du allerdings davon ausgehen, dass die Hardware diesen besseren Farbraum unterstützt. Wenn du ein VR-Spiel entwickelst, kannst du in der Regel, es sei denn, es ist ein Cardboard-Spiel, davon ausgehen, dass deine Ziel-Hardware den *Linear Color Space* unterstützt, und ich empfehle dir, ihn auch zu nutzen.

Den Farbraum kannst du in den *Player Settings* ändern: **EDIT/PROJECT SETTINGS/PLAYER/OTHER SETTINGS/COLOR SPACE**.

6.1.3 Environment Light

Ein wichtiges Element für den optischen Gesamteindruck deiner Scene ist das sogenannte *Environment Lighting* (dt. „Umgebungsbeleuchtung“). Die Option wird in älteren Unity-Versionen auch als „Ambient Lighting“ bezeichnet.). Dieses Licht bestimmt, wie hell oder dunkel die Scene ohne Lichtquellen bzw. an Stellen, wo Lichtquellen sie nicht erhellen, ist.

Für Scenes, bei denen der Himmel deutlich sichtbar ist, ist die Standardoption, welche die Helligkeit basierend auf der Skybox ausrechnet, häufig die optisch passendste Lösung. In Spielen, die hauptsächlich drinnen spielen und bei denen der Himmel nicht relevant ist, ist es in der Regel sinnvoll, eine eigene, dunklere Farbe zu wählen, sodass die Scene nur dort

hell ist, wo Lichtquellen sie erhellen. Bei einem Spiel mit einer Cartoon-Optik, wo sehr dunkle Schatten nicht erwünscht sind, könntest du auch eine hellere Farbe wählen.

Mit der *Environment-Light-Farbe* kannst du die gesamte Helligkeit und Farbe deiner Scene erhöhen oder verringern, ohne dass du die einzelnen Lichter anpassen musst. Bild 6.13 zeigt, um das zu demonstrieren, dieselbe Scene mit verschiedenen *Environment-Light-Farben*.

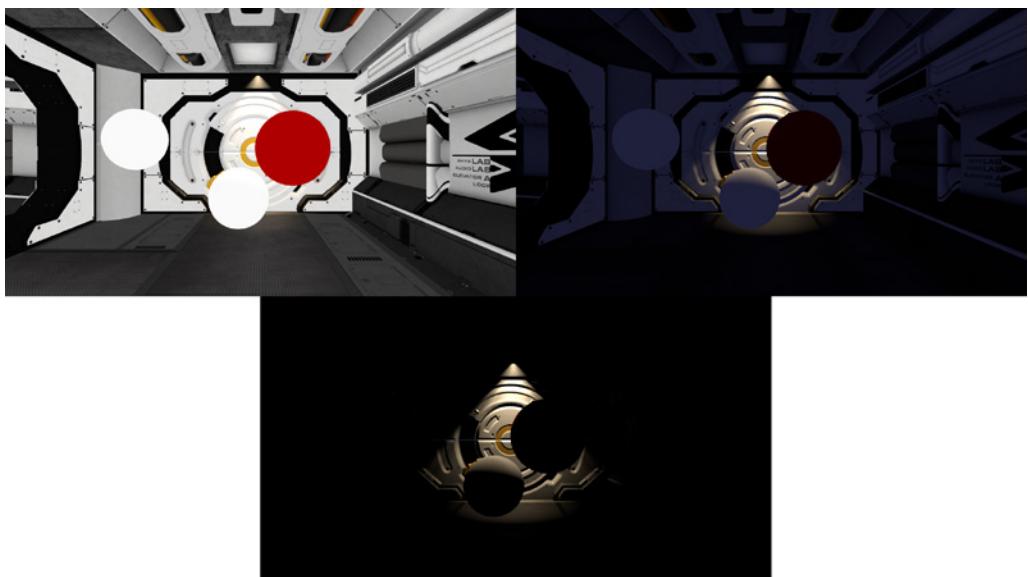


Bild 6.13 Dieselbe Scene mit weißem, dunkelblauem und komplett schwarzem Environment Light

Das *Environment Lighting* kann in den *Lighting*-Einstellungen geändert werden: **WINDOWS/LIGHTING/SETTINGS/ENVIRONMENT LIGHTING**. Standardmäßig ist dort der Wert *Skybox* ausgewählt. Alternativ kannst du einen Farbverlauf (*Gradient*) oder, wie in Bild 6.13, eine feste Farbe (*Color*) angeben.

6.1.4 Lichter in der Scene

An dieser Stelle haben wir nun die wichtigsten Elemente besprochen, über die du dir Gedanken machen musst, bevor du anfängst, deine Scenes in Unity auszuleuchten. Als Nächstes schauen wir uns die unterschiedlichen Lichttypen an, die du verwenden kannst, um deine Scenes auszuleuchten. In Unity kannst du zu jedem beliebigen *GameObject* ein *Light*-Component hinzufügen, um es leuchten zu lassen. In den meisten Fällen macht es aber Sinn, für die Lichter eigene *GameObjects* zu erzeugen, sodass du sie völlig frei bewegen kannst. Letzteres kannst du bequem über das *Create*-Menü in der *Hierarchy* erledigen: **CREATE/LIGHT/....**

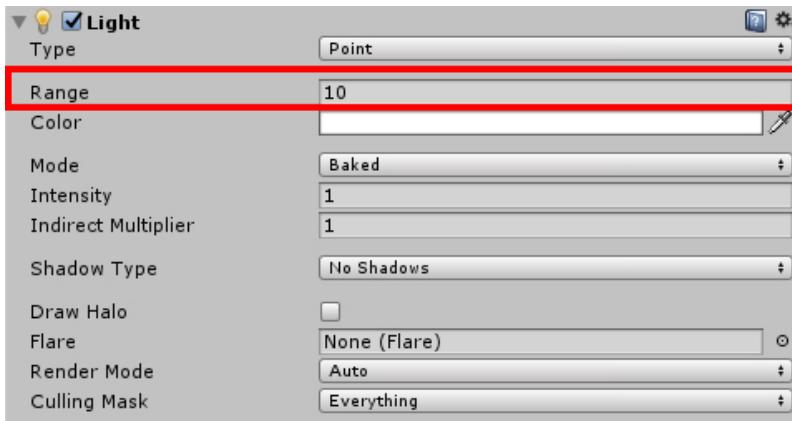


Bild 6.14 Ein Light-Component im Inspector. Der markierte Bereich enthält für jeden Lichttyp individuelle Parameter.

Das in Bild 6.14 zu sehende *Light*-Component ist universal einsetzbar, da es verschiedene Typen von Licht imitieren kann. Es ist über den Lichttyp zum Beispiel möglich, das Component wie eine *Sonne* zu konfigurieren, aber auch wie das *Spotlight* einer Taschenlampe. Jeder Lichttyp hat individuelle Einstellungsmöglichkeiten, es gibt jedoch auch einige Optionen, die für alle Lichttypen identisch sind. Diese schauen wir uns als Erstes an, danach gehen wir ins Detail mit den einzelnen Typen.

- **Type:** Bestimmt den Lichttyp.
- **Color:** Gibt die Farbe des Lichtes an.
- **Mode:** Gibt an, ob diese Lichtquelle in Echtzeit berechnet wird (*Realtime*) oder vorberechnet (*Baked*) ist. Bei vorberechneten Lichtern können zur Laufzeit weder die Eigenschaften noch die Position oder Rotation verändert werden. Zudem werden bei vorberechneten Lichtern auch keine Schatten für Objekte, die nicht als *Static* markiert wurden, angezeigt. Echtzeit-Lichter erhellen alle Objekte, kosten allerdings viel mehr Performance und das Ergebnis sieht häufig ein wenig schlechter aus als das von vorberechneten Lichtern. Die Option *Mixed* ist eine Mischung aus *Realtime* und *Baked*. Das Licht wird vorberechnet, bestrahlt aber über Echtzeitlicht-Berechnung auch Objekte, die nicht *Static* sind.
- **Intensity:** Gibt die Intensität des Lichtes an. Die Intensität bestimmt zusammen mit der Helligkeit der *Color*-Eigenschaft die Gesamthelligkeit des Lichtes.
- **Indirect Multiplier:** In Unity wird Licht von Objekten, auf die es trifft, reflektiert. Mit jeder Reflexion nimmt die Intensität des Lichtes ab. Über diesen Faktor kannst du das Abnehmen der Intensität beeinflussen. Steht dieser Wert auf 0, wird kein indirektes Licht reflektiert.
- **Shadow Type (nicht bei Area-Light):** Diese Eigenschaft bestimmt die Qualität der Echtzeitschatten. *Soft Shadows* sehen mit ihrem weichen Übergang realistischer aus. *Hard Shadows* sind jedoch weniger rechenaufwendig.
- **Draw Halo , Flare:** Diese beiden Optionen können Linseneffekte simulieren, wenn der Anwender in das Licht schaut. In Virtual Reality sollte man auf diese Effekte verzichten, da im Auge keine Linseneffekte auftreten.

- **Render Mode:** Zur Laufzeit optimiert Unity die *Realtime*- und *Mixed*-Lichtquellen, um Rechenleistung zu sparen. Durch das Optimieren kann es dazu kommen, dass die Lichtquelle kein Licht abstrahlt, obwohl sie im Bild ist. Mit dieser Option kann man angeben, dass es sich um ein wichtiges Licht (*Important Light*) handelt. Du solltest diese Option auf *Auto* stehen lassen, es sei denn, eine Lichtquelle bereitet dir Probleme.
- **Culling Mask:** Mit der Culling Mask kannst du dafür sorgen, dass diese Lichtquelle nur GameObjects auf einem bestimmten *Layer* bestrahlt. Dies hilft zum Beispiel, wenn du einzelne Objekte besonders hervorheben möchtest, indem sie heller leuchten als die Umgebung. Mit der Option *Everything* werden alle Objekte angestrahlt, was dem typischen Verhalten eines Lichtes entspricht.

6.1.4.1 Point Light

Ein Point Light wird typischerweise für Lampen und Kerzen verwendet, die nicht in eine bestimmte Richtung leuchten. Bei diesem Lichttyp strahlt nämlich von der Position des GameObjects Licht in alle Richtungen. Die Rotation des GameObjects ist daher unwichtig für diesen Lichttyp.

Individuelle Parameter:

- **Range:** Gibt die Reichweite der Lichtquelle an. Bis zu dem Endpunkt der Reichweite nimmt die Lichtstärke immer weiter ab. In der Scene View kannst du die Reichweite an dem gelben Kreis, der sich um das GameObject aufspannt, erkennen. Über diesen Helfer kannst du die Reichweite auch in der *Scene View* anpassen.

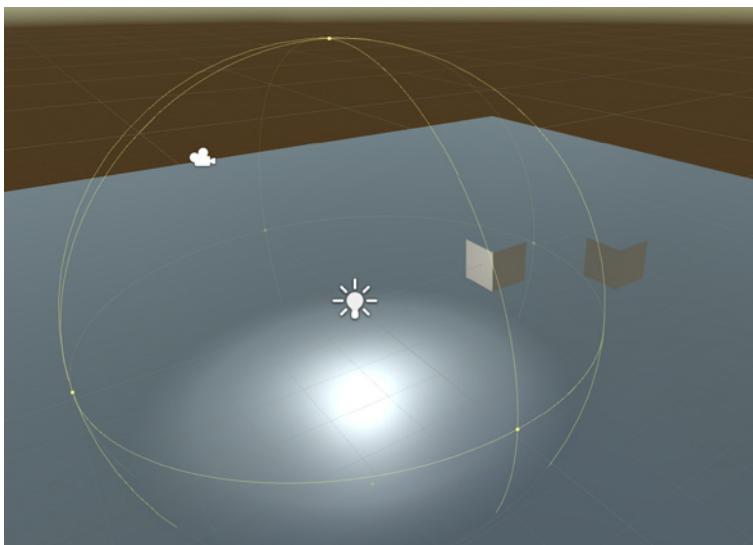


Bild 6.15 Ein Point-Light in der *Scene View*. Der gelbe Kreis ist der Helfer zum Anpassen der *Range*.

6.1.4.2 Directional Light

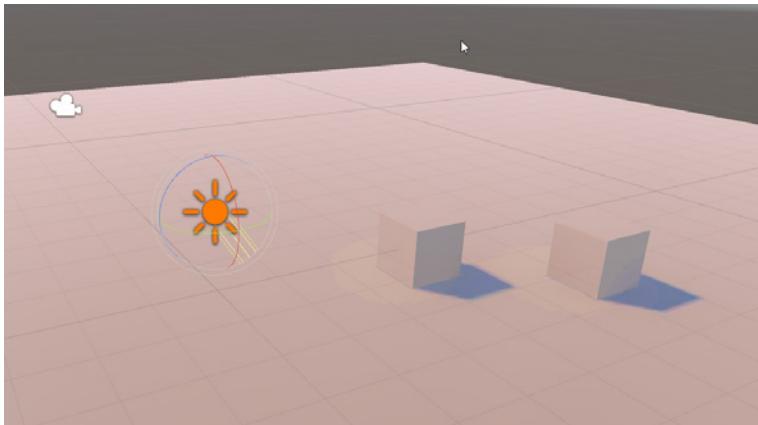


Bild 6.16 Directional Lights imitieren Sonnenlicht.

Dieser Lichttyp imitiert das Verhalten von Sonnenstrahlen. Bei den *Directional Lights* ist deshalb die Position des GameObjects unwichtig. Die Rotation bestimmt jedoch, aus welcher Richtung das Licht scheint. Für gewöhnlich hat man in einer Scene nur ein einziges *Directional Light*, das die Sonne simuliert. Grundsätzlich kann man aber beliebig viele *Directional Lights* in einer Scene verwenden, was interessant werden kann, wenn man besondere Lichtstimmungen erzeugen möchte. Bei dem *Directional Light* kannst du anhand der gelben Linien in der *Scene View* die Strahlrichtung des Components erkennen.

6.1.4.3 Spot Light

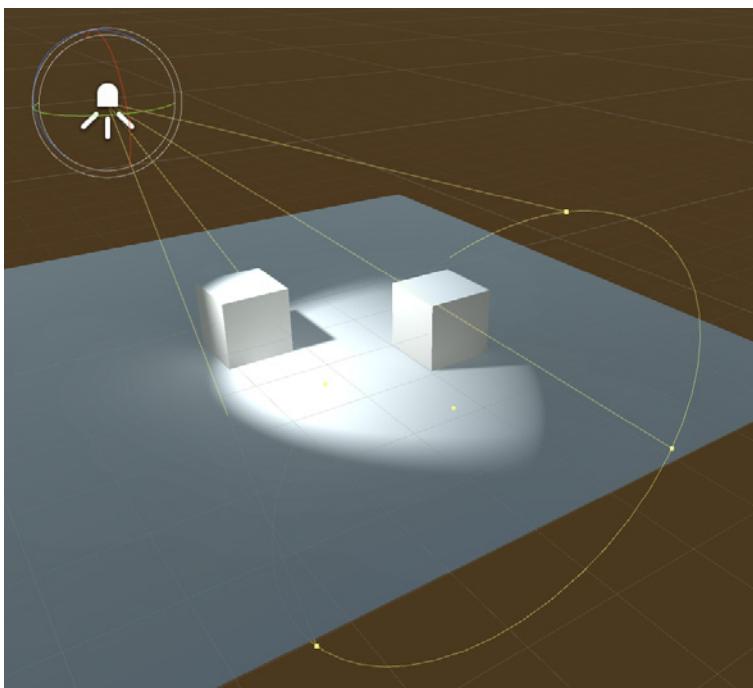


Bild 6.17 Das Spot-Light-Handle in der Scene View

Spot Lights erzeugen einen Lichtkegel in eine bestimmte Richtung, wie du es zum Beispiel von Taschenlampen kennst. Beim Erstellen eines *Spot Lights* sind also Rotation und Position wichtig.

Individuelle Parameter:

- **Range:** Gibt die Reichweite der Lichtquelle an. Bis zu dem Endpunkt der Reichweite nimmt die Lichtstärke immer weiter ab. In der *Scene View* kannst du die Reichweite an dem Boden des gelben Kegels erkennen. Greifst du den Boden des Kegels an dem gelben Punkt in der Mitte, kannst du die Reichweite auch in der *Scene View* anpassen.
- **Spot Angle:** Gibt den Öffnungswinkel des Kegels an. Ein größerer Öffnungswinkel sorgt für einen höheren Durchmesser des Lichtkegels, ohne dass die *Range* angepasst werden muss.

6.1.4.4 Area Light

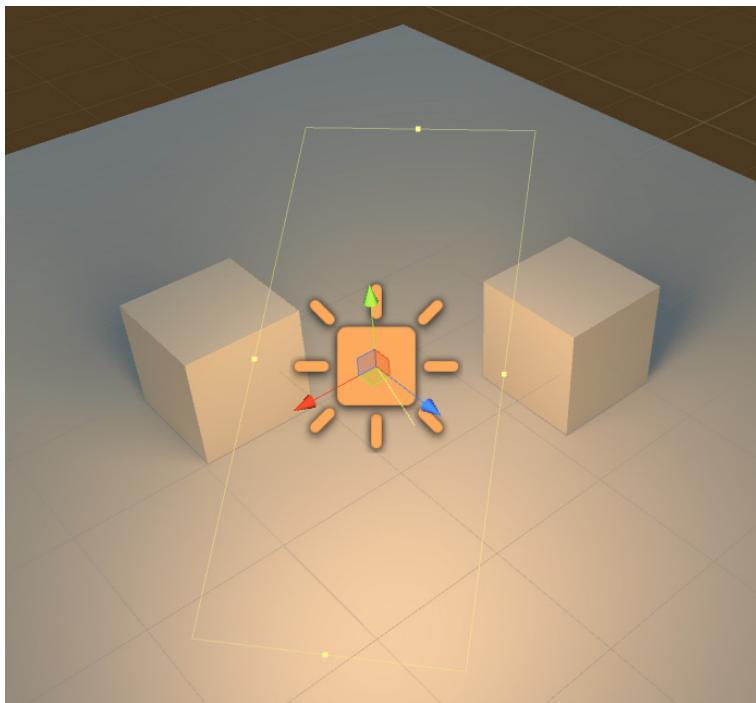


Bild 6.18 Ein *Area Light* in der Scene View

Ein *Area Light* strahlt wie das *Directional Light* paralleles Licht aus. Dieser Lichttyp erhellt allerdings nicht die komplette *Scene*, sondern nur einen angegebenen Bereich. *Area Lights* werden meist als Hilfsmittel benutzt, um bestimmte Bereiche einer großen Scene auszuleuchten, wenn sie durch das *Directional Light* nicht ausreichend erhellt werden. Auf diese Weise können Korrekturen an der Lichtstimmung für einzelne Bereiche vorgenommen werden, ohne dass diese den Rest der Scene beeinflussen. Für solche Fälle kann kein Spot oder Point Light verwendet werden, da die Schatten in diesen Fällen nicht parallel fallen würden, wie es bei Sonnenlicht der Fall ist. *Area Lights* können nur vorberechnet werden und bieten keine Echtzeit-Funktionen.

Individuelle Parameter:

- **Width:** Breite des Licht ausstrahlenden Bereiches. Dieser Wert kann in der Scene View an dem gelben Rechteck des Area Lights angepasst werden.
- **Height:** Höhe des Licht ausstrahlenden Bereiches. Dieser Wert kann in der Scene View an dem gelben Rechteck des Area Lights angepasst werden.

6.1.5 Emissive Materials

Emissive Materials sind speziell konfigurierte *Materials*, welche in der Lage sind, Licht abzustrahlen und so ihre Umwelt, wie in Bild 6.19, zu erhellen.

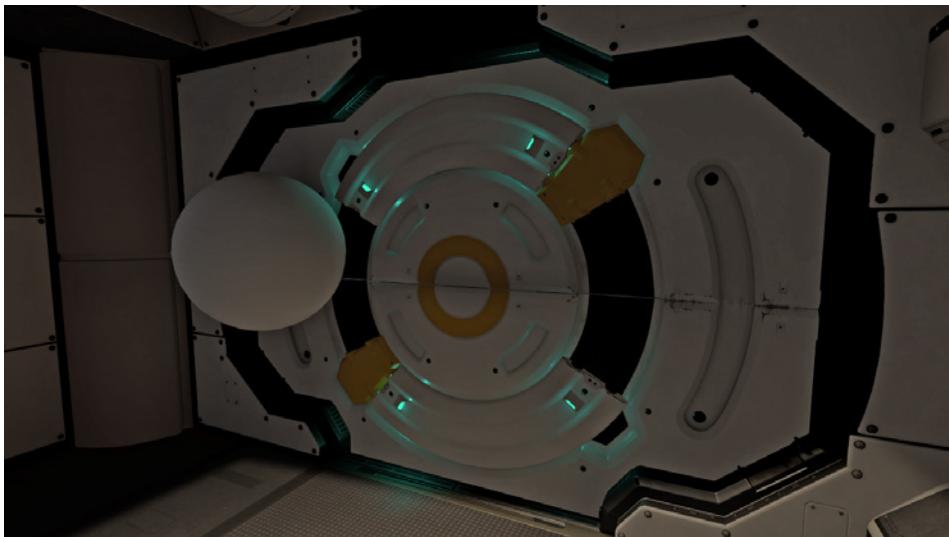


Bild 6.19 Lampen an einer Weltraumtür, realisiert mit einem Emissive Material

Emissive Materials sollten jedoch nicht als Ersatz für Lichter verwendet werden, sondern eher als Detail auf normalen Materials. Denkbar ist der Einsatz zum Beispiel bei einem Fernseher, bei dem die Vorderseite Licht abstrahlen soll, oder auch zum Beispiel für LEDs auf einem Steuerungspult.

Im *Standard-Shader* kannst du eine *Emission Map* angeben, welche bestimmt, an welchen Stellen das Material leuchten soll. Zusätzlich kannst du noch bestimmen, in welcher Farbe das Material leuchten soll und auch mit welcher Intensität.

Damit das Licht des Materials sichtbar auf die Umgebung abstrahlt, muss häufig ein höherer Intensitätswert gewählt werden. Das beeinflusst jedoch auch das Aussehen der leuchtenden Stellen. In einigen Fällen ist es praktischer, die Intensität im Material geringer zu wählen und das Licht auf den umliegenden Objekten mit einem *Point-* oder *Spot-Light* zu realisieren.

6.1.6 Light Probes

Dynamische Objekte werden bei der Berechnung des indirekten Lichtes (*Global Illumination*) nicht berücksichtigt. Wenn du in deiner Scene vollständig auf Realtime- und Mixed-Lights verzichtest, werden sie sogar gar nicht erleuchtet und werden, je nach *Environment Light*, vollkommen schwarz dargestellt. Um dieses Problem zu umgehen und deine beweglichen Objekte nahtlos in eine vorberechnete Lichtwelt einzufügen, verwendet man *Light Probes*.

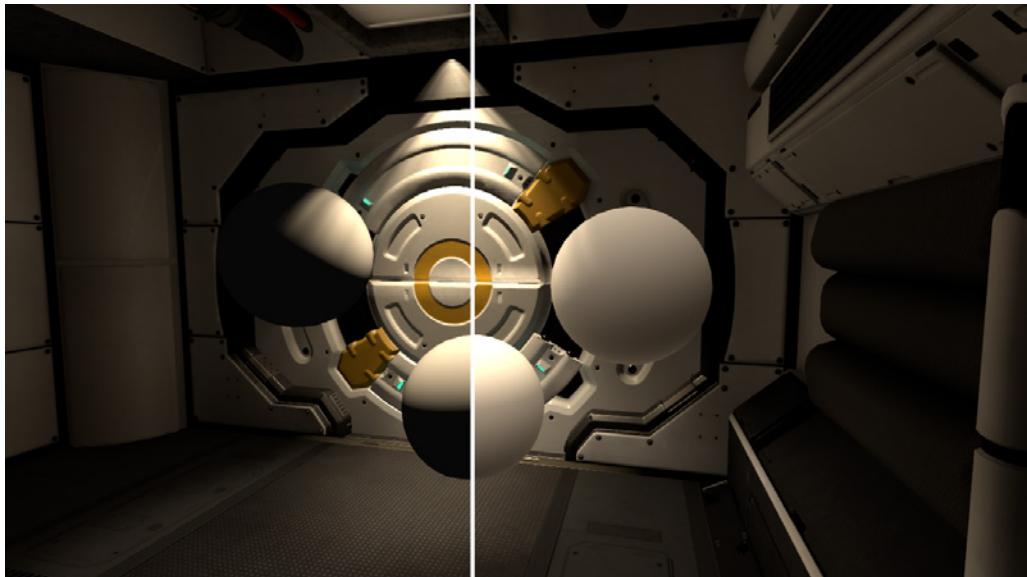


Bild 6.20 Eine Scene mit drei nicht statischen Kugeln und Mixed-Lights. Links ohne Light Probes und somit nur mit direktem Licht, rechts mit Light Probes, also direkt und indirektem Licht.

Indirektes Licht für dynamische Objekte in Echtzeit zu berechnen ist, wie du nun schon gelernt hast, viel zu rechenaufwendig. Also, wieso funktioniert das mit *Light Probes*?

Light Probes verwenden nur Annäherungswerte statt perfekten Berechnungen. Eine *Light Probe* ist ein einzelner Messpunkt. Für diesen Messpunkt wird während der Lichtberechnung ein Farb- und Helligkeitswert, inklusive direktem und indirektem Licht, berechnet. Befinden sich mehrere *Light Probes* in der Scene, bilden sie eine *Light Probe Group*, deren Messdaten miteinander verrechnet werden können. Befindet sich ein dynamisches *GameObject* in der *Scene*, wird der Lichtwert für dieses GameObject, wie in Bild 6.21 zu sehen, basierend auf den darum herumliegenden *Light Probes* ausgerechnet und das *GameObject* plausibel erhellt.

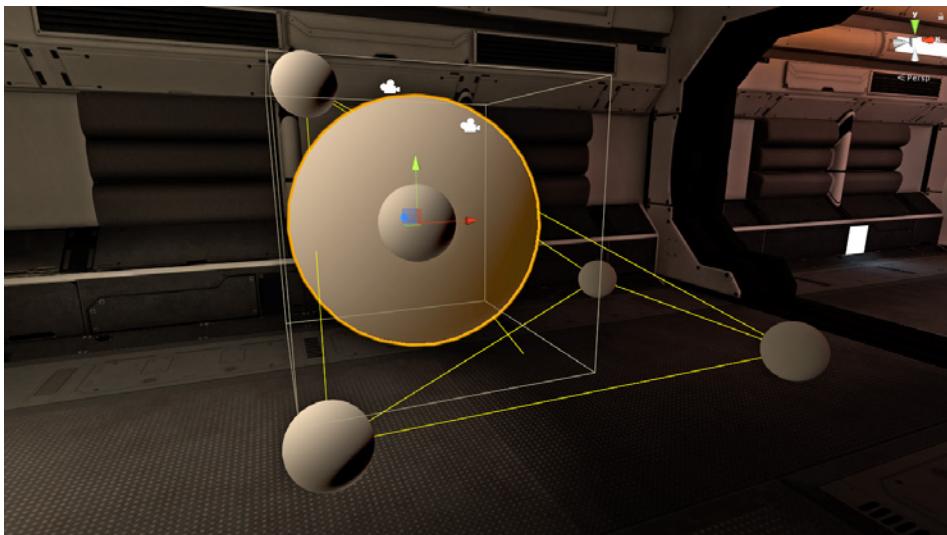


Bild 6.21 Durch Light Probes kann indirektes Licht auch für dynamische Objekte annäherungsweise berechnet werden.

6.1.6.1 Light Probes zur Scene hinzufügen

Light Probes kannst du über das Create-Menü in der *Hierarchy* zu deiner Scene hinzufügen: (**CREATE/LIGHT/LIGHT PROBE GROUP**). Dadurch wird ein neues GameObject mit einem *LightProbeGroup*-Component erzeugt, welches beliebig viele *Light Probes* verwalten kann.

Über die Schaltfläche **EDIT LIGHT PROBES** im *Inspector* kannst du den Bearbeitungsmodus aktivieren. In diesem Modus kannst du eine oder mehrere der *Light Probes* in der Scene View auswählen und sie verschieben oder entfernen. Über die Schaltfläche **DUPLICATE SELECTED** kannst du weitere *Light Probes* hinzufügen, indem du die aktuelle Auswahl duplizierst. Alternativ kannst du einzelne *Light Probes* über die Schaltfläche **ADD LIGHT PROBE** hinzufügen.

Wie viele *Light Probes* du platzieren solltest, ist davon abhängig, wie groß die einzelnen Bereiche deiner Scene sind und wie stark sich die Lichtsituation verändert. In Bereichen mit gleichbleibender Lichtstimmung reicht es, wenige *Light Probes* zu platzieren. In den Bereichen, wo sich das Licht stark verändert (hell zu dunkel oder Farbe), solltest du, wie in Bild 6.22, mehr *Light Probes* platzieren.

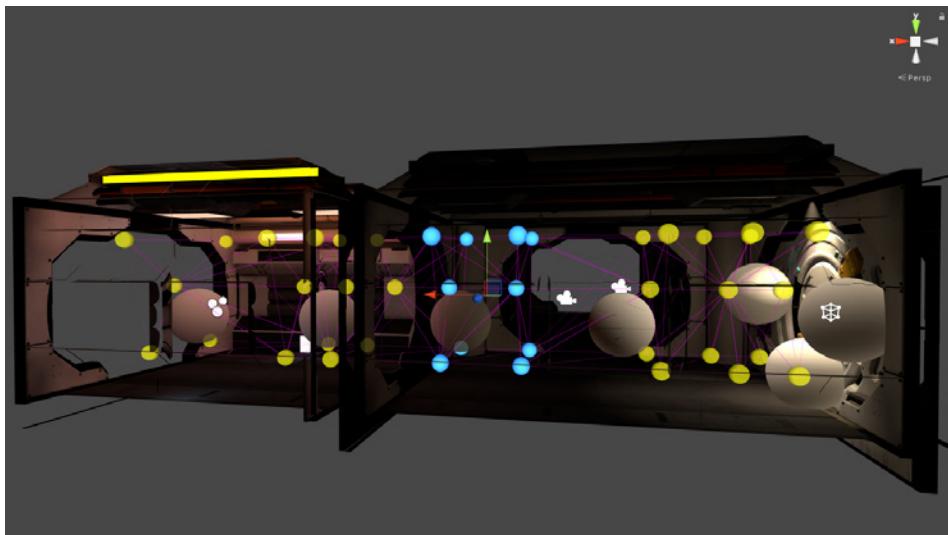


Bild 6.22 Ein Level mit Light Probes; beachte die höhere Dichte an Messpunkten, an Stellen, wo sich die Lichtsituation stark ändert.

Grundsätzlich empfiehlt es sich, die *Light Probes* gleichmäßig zu verteilen, Ecken, in denen der Spieler aber nur selten hinkommt, solltest du mit weniger Messpunkten versehen als Bereiche, wo sich der Spieler ständig aufhält. Zu viele *Light Probes* können auch zu Performance-Einbußen führen.



Light Probes dürfen nicht in statischer Geometrie stecken

Beim Platzieren der Light Probes musst du darauf achten, dass keine Light Probe mit einem statischen Objekt kollidiert. Ansonsten wird diese Light Probe nach der Lichtberechnung komplett schwarz sein und Objekte in ihrer Nähe dunkel einfärben.

6.1.7 Reflexionen

Standardmäßig nutzen Modelle in Unity den sogenannten „Standard“-Shader. Dieser Shader ist ein sogenannter *Physically Based Shader* (kurz PBS). Das bedeutet, er wurde dafür entworfen, das Verhalten von Licht auf unterschiedlichen Materialien möglichst akkurat darzustellen. Dazu imitiert der *Shader* die physikalischen Eigenschaften des gewünschten Materials, also zum Beispiel auch wie stark die Oberfläche reflektiert.

Selbst wenn ein *Material* die Welt nicht buchstäblich spiegelt, sondern einfach nur ein Teil des Lichtes reflektiert, wie zum Beispiel viele metallene Oberflächen, macht dieser Effekt einen großen Unterschied in der Optik, wie du in Bild 6.23 sehen kannst.

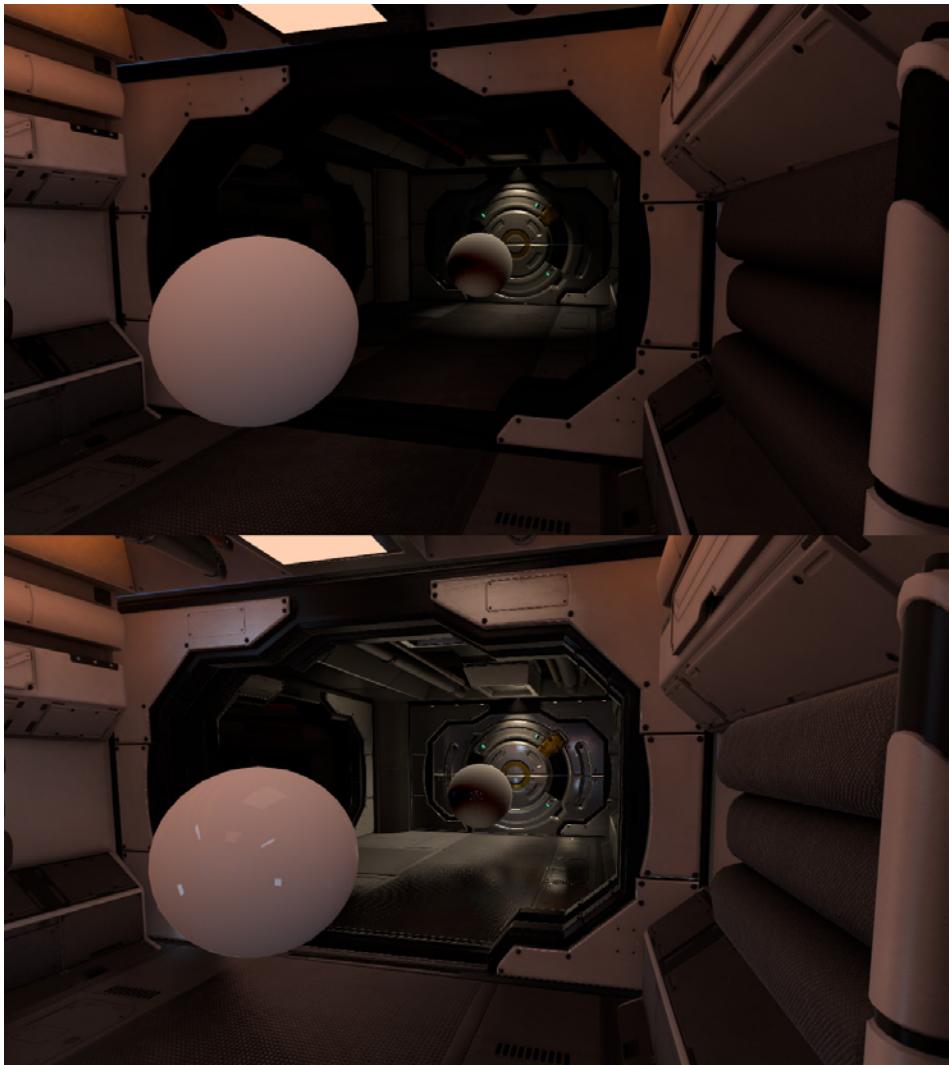


Bild 6.23 Ohne Reflexionen oben, mit Reflexionen unten. Neben der sichtbaren Reflexion auf der Oberfläche wird die Reflexion auch in die Lichtberechnung aufgenommen, wodurch der Raum heller wird.

6.1.7.1 Reflexionen zur Scene hinzufügen

Physikalisch korrekte Reflexionen für jeden beliebigen Punkt auf einer Oberfläche zu berechnen, ist selbst mit einem starken Gaming-PC nicht in Echtzeit möglich. Aus diesem Grund bietet Unity unterschiedliche Varianten an, um die Reflexionen im Vorfeld zu berechnen:

Standardmäßig reflektieren alle Modelle, deren *Shader* entsprechend konfiguriert ist, die *Skybox* deiner *Scene*. Alternativ kannst du in den *Lighting*-Einstellungen auch eine feste

Grafik (*Cubemap*) angeben, welche für die Reflexion verwendet werden soll (**WINDOWS/LIGHTING/SETTINGS/ENVIRONMENT REFLECTIONS**).

Exaktere Reflexionen kannst du mit sogenannten *Reflection Probes* erreichen. *Reflection Probes* werden in der Scene platziert. Jede einzelne berechnet dann die Reflexion für ihre Position und speichert sie in einer Textur. Für jede *Reflection Probe* kannst du dann einen Bereich definieren, in dem die Reflexion dieser Probe verwendet werden soll. Bild 6.24 zeigt beispielhaft, wie ein Level mit einer *Reflection Probe* pro Raum aussehen könnte.

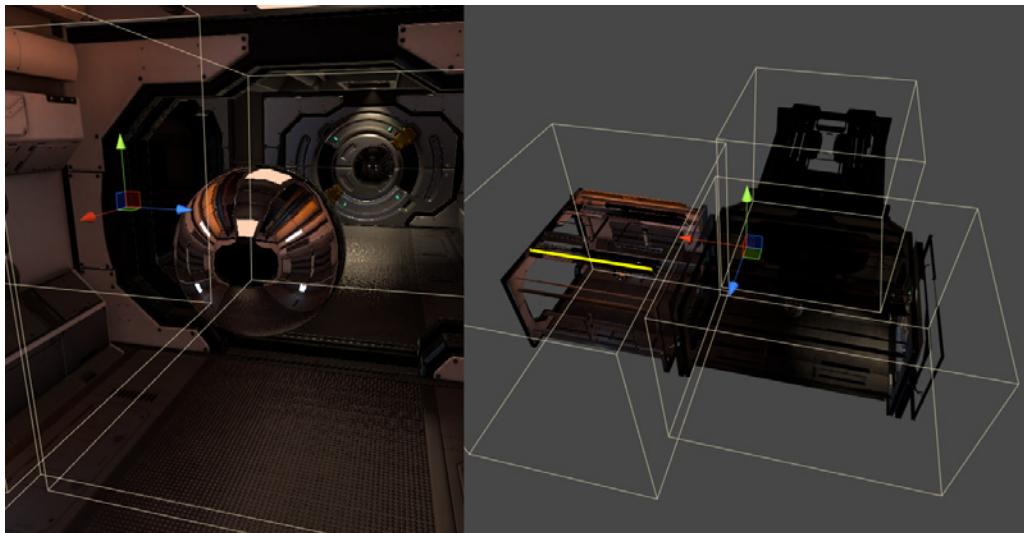


Bild 6.24 Für jeden Raum wurde eine eigene relevante *Reflection Probe* hinzugefügt, damit jeweils die richtige Lichtstimmung reflektiert wird.

Reflection Probes kannst du, wie die meisten Elemente, entweder als Component zu einem bestehenden *GameObject* hinzufügen oder über das *Create*-Menü in der *Hierarchy* erzeugen: **CREATE/LIGHT/REFLECTION PROBE**.

Über die *Type*-Einstellung im *Inspector* des Components kannst du bestimmen, wann und wie die *Reflection Probe* berechnet werden soll. *Reflection Probes* können entweder während der Lichtberechnung gerendert werden (*Baked*) oder auch erst während dein Spiel läuft (*Realtime*). In letzterem Fall kannst du noch zusätzlich über die *Refresh Mode*-Einstellung unterscheiden, ob dies nur einmalig beim Laden der Scene passieren soll (*On Awake*), in jedem Frame (*Every Frame*) oder ob das Rendering durch ein Script gesteuert werden soll (*Via scripting*). Da das Berechnen der *Reflection Probes* sehr rechenintensiv sein kann, kannst du die Berechnung über die Einstellung *Time Slicing* auf mehrere Frames verteilen, anstelle die komplette Textur in nur einem einzigen Frame berechnen zu lassen.

Aufgrund der deutlich besseren Performance solltest du, wenn möglich, immer vorberechnete (*Baked*) *Reflection Probes* verwenden. Diese haben jedoch den Nachteil, dass sie nicht dynamisch auf wechselnde Lichtsituationen eingehen und nichtstatische Objekte in der Reflexion nicht sichtbar sind. Letzteres ist allerdings eher nur für stark reflektierende Flächen wie Spiegel relevant. Das Reagieren auf wechselnde Lichtsituationen ist hingegen ein häufiges auftretendes Problem. Kann man in deinem Spiel beispielsweise das Licht in einem

Raum ein- und ausschalten, sind vorberechnete *Reflection Probes* nur bedingt gut geeignet. In diesen Fällen solltest du *nicht* den *Refresh Mode* auf *Every Frame* stellen, auch wenn dies der einfachste Weg ist. Der bessere Weg wäre, die Berechnung der *Reflection Probes* über ein Script zu steuern, sodass sie einmalig neu berechnet werden, wenn das Licht ein- oder ausgeschaltet wird.

In den *Cubemap Capture Settings* solltest du die Auflösung (*Resolution*) so gering wie möglich, aber so groß wie nötig wählen. Wenn du stark reflektierende beziehungsweise spiegelnde Flächen in deiner Scene hast, ist eine höhere Auflösung häufig sinnvoll. Wenn auf der Oberfläche des Materials die Reflexion ohnehin nur verschwommen, unscharf oder gar nicht zu sehen ist (dies ist abhängig von den Einstellungen im Shader), kannst du problemlos eine kleinere Auflösung wählen. Je geringer die Auflösung, desto besser die *Performance*. Dieser Tipp ist vor allem bei Echtzeit-*Reflection-Probes* wichtig.



Plausiblere Reflexionen auf Oberflächen wie Böden und Wänden

Solltest du in deiner Scene, wie in Bild 6.24, eine *Reflection Probe* pro Raum verwenden, um Reflexionen für die Wände, Decken und Böden deiner Räume zu berechnen, erhältst du einen optisch realistischeren Effekt, wenn du die *Option Box Projection* in den *Runtime Settings* der jeweiligen *Reflection Probes* aktivierst.

6.1.8 Lighting-Fenster

Das *Lighting*-Fenster enthält Einstellungen für die *Skybox*, *Global Illumination* und Lichtberechnung im Allgemeinen. Außerdem findest du dort noch weitere Einstellungen, die sich mit der Darstellung deiner Scene beschäftigen. Eine exakte Beschreibung zu jedem der Parameter findest du in den Tooltips und in der Dokumentation. An dieser Stelle werde ich dir die wichtigsten Optionen vorstellen.

Environment

- **Skybox Material:** Hier kannst du die Skybox ändern.
- **Sun Source:** Hier kannst du ein *Directional Light* angeben, das von der Skybox als Sonne interpretiert wird. Bei einer prozeduralen *Skybox* wird die Sonne dann automatisch an einer plausiblen Position im Himmel angezeigt.
- **Environment Lighting:** Hier kannst du das Environment Light konfigurieren (siehe Kapitel 6.1.3).
- **Environment Reflections:** Hier kannst du die globalen Umgebungsreflexionen konfigurieren (siehe Kapitel 6.1.7).

Realtime Lighting

- **Realtime Global Illumination:** Hier kannst du Realtime GI aktivieren (siehe Kapitel 6.1.1.3).

Mixed Lighting

- **Baked Global Illumination:** Aktiviert die vorberechnete Global Illumination (siehe Kapitel 6.1.1.2).
- **Lighting Mode:** Hier kannst du konfigurieren, wie das Licht für die gemischte Licht-Berechnung berechnet werden soll (siehe Kapitel 6.1.1.5.1).

Lightmapping Settings

Hier kannst du allgemeine Einstellungen für die Qualität der Lichtberechnung vornehmen.

- **Lightmapper:** *Enlighten* ist der Standard-Lightmapper seit Unity 5. Neu dazugekommen ist der *Progressive Lightmapper*, welcher zuerst den in der Scene View sichtbaren Bereich berechnet und während der Berechnung den Fortschritt anzeigt. Du erhältst mit ihm also sehr schnell eine Vorschau von den Teilen deiner Scene, die du dir gerade ansiehst. Aufgrund einiger Einschränkungen solltest du die finalen Light Maps derzeit jedoch mit *Enlighten* berechnen.
- **Indirect Resolution:** Bestimmt die Auflösung für die Berechnung des indirekten Lichts. Ein höherer Wert bedeutet ein exakteres Ergebnis, allerdings auch deutlich mehr Rechenzeit.
- **Lightmap Resolution:** Bestimmt die Auflösung für die Berechnung der *Lightmap*.
- **Lightmap Padding:** Bestimmt den Abstand zwischen Objekten auf der *Lightmap*. Diesen Wert musst du nur ändern, wenn das Licht von mehreren Objekten ineinander überläuft.
- **Lightmap Size:** Die Dateiauflösung der gespeicherten Lightmap. Je höher die Auflösung, umso höher ist die Qualität der gespeicherten Lightmap. Allerdings wird auch mehr RAM für die Berechnung benötigt. Für den Wert 4096 empfiehlt sich, mindestens 16 GB RAM zu haben, und ggf. solltest du andere Anwendungen bei der Berechnung schließen.
- **Compress Lightmaps:** Reduziert die Dateigröße der gespeicherten Lightmaps, reduziert jedoch auch die Qualität. Wenn du Artefakte (wie bei stark komprimierten JPEGs) in deiner Scene siehst, deaktiviere diese Option.
- **Ambient Occlusion:** Erzeugt kleine, weiche Schatten in Ritzen und Spalten von Objekten in deiner Scene. Hierdurch erhöht sich die Rechenzeit der Lightmap, dafür wirkt sie deutlich realistischer. Wenn du diese Option aktivierst, stehen dir weitere Parameter zur Verfügung, um das Aussehen der Schatten genauer anzupassen.
- **Final Gather:** Wenn diese Option aktiviert ist, wird der letzte Durchlauf der Lightmap-Berechnung mit erhöhter Auflösung durchgeführt. Dies erhöht die optische Qualität enorm, erhöht jedoch auch die Rechenzeit. Wenn du diese Option aktivierst, stehen dir weitere Parameter zur Verfügung, um den *Final-Gather*-Prozess genauer anzupassen.
- **Directional Mode:** Die Option *Directional* verwendet Informationen über die Oberfläche des Materials, um ein besseres Ergebnis zu erzielen. Auf älteren Smartphones wird dies jedoch ggf. nicht unterstützt.

- **Indirect Intensity:** Hier kannst du bestimmen, wie stark das indirekte Licht leuchten soll. Manchmal macht es Sinn, hier ein wenig herum zu probieren, um den perfekten Wert für die eigene Scene zu finden.
- **Albedo Boost:** Erhellst die Oberfläche von Materialien. Der Standardwert ist realistisch und muss für gewöhnlich nicht verändert werden.
- **Lightmap Parameters:** Einige weitere Einstellungen werden unter diesen *Lightmap Parameters* zusammengefasst. Dir stehen hier verschiedene Presets zur Verfügung. Die Presets mit höherer Qualität sehen besser aus, dauern aber auch länger zu berechnen.

Other Settings

- **Fog:** Diese Option aktiviert einen Nebel, der entfernte Geometrie ausblendet und so die Sichtweite reduziert. In älteren Videospiele auf schwächerer Hardware wurde dies häufig verwendet. Der Nebel sieht nicht realistisch aus, weswegen du ihn wahrscheinlich nicht als Nebel-Effekt verwenden möchtest. (Das würde man eher mit einem Partikelsystem lösen, sieh dich mal im Asset Store um.)
- **Halo und Flare:** Dies sind beides Effekte, welche bestimmte Linseneffekte mit einfachen Mitteln darstellen. Ich empfehle, sie für VR-Spiele nicht zu verwenden, da sie häufig veraltet wirken.
- **Spot Cookie:** Diese Option ändert das Aussehen der Lichtkegel von *SpotLights*, zum Beispiel um sie mehr wie Taschenlampen aussehen zu lassen.

6.1.9 Light Explorer

Das „Light Explorer“-Fenster findest du in der Toolbar unter **WINDOW/LIGHTING/LIGHT EXPLORER**. Dieses Fenster ist sehr praktisch, um die Lichtsituation in Scenes mit vielen Lichtquellen zu verwalten. Wie du in Bild 6.25 sehen kannst, stellt dir das Fenster nämlich eine Auflistung aller vorhandenen Lichtquellen zur Verfügung und erlaubt es dir, alle an einer zentralen Stelle zu verwalten. Spätestens für das Feintuning der Scene ist dieses Fenster praktisch, wenn du mit den Farbwerten und den Intensitäten ein wenig herumspielen möchtest. Der *Light Explorer* ist nicht begrenzt auf *Light*-Components, über die Reiter *Reflection Probes*, *Light Probes* und *Static Emissive*s kannst du jeweils andere Components verwalten und musst nicht die *Hierarchy* oder die *Scene View* nach den jeweiligen *GameObjects* durchsuchen.

Name	On	Type	Mode	Color	Intensity	Indirect Multiplier	Shadow Type
Light	<input checked="" type="checkbox"/>	Point	Mixed	[Color Swatch]	2	1	No Shadows
Light	<input checked="" type="checkbox"/>	Point	Mixed	[Color Swatch]	2	1	No Shadows
Point light (3)	<input checked="" type="checkbox"/>	Point	Baked	[Color Swatch]	0.5	1	Soft Shadows
HallwayWallSpot	<input checked="" type="checkbox"/>	Spot	Mixed	[Color Swatch]	1.7	1.25	No Shadows
HallwaySpotlightEnd	<input checked="" type="checkbox"/>	Spot	Mixed	[Color Swatch]	2.5	1.65	No Shadows
HallwayWallSpot	<input type="checkbox"/>	Spot	Mixed	[Color Swatch]	4	1	No Shadows
HallwayWallSpot	<input type="checkbox"/>	Spot	Mixed	[Color Swatch]	1.7	1.25	No Shadows
HallwaySpotlightEnd	<input type="checkbox"/>	Spot	Mixed	[Color Swatch]	2.5	1.65	No Shadows
HallwaySpotlightEnd	<input checked="" type="checkbox"/>	Spot	Mixed	[Color Swatch]	2.5	1.65	No Shadows
HallwayWallSpot	<input checked="" type="checkbox"/>	Spot	Mixed	[Color Swatch]	2.43	1	No Shadows
HallwayWallSpot	<input checked="" type="checkbox"/>	Spot	Mixed	[Color Swatch]	1.7	1.25	No Shadows
HallwayDownlighterSpot	<input type="checkbox"/>	Spot	Mixed	[Color Swatch]	3.25	1	Soft Shadows

Bild 6.25 Der Light Explorer zeigt alle Elemente in der Scene, welche die Lichtstimmung aktiv beeinflussen.

■ 6.2 Shader

Shader sind kleine Programme, die auf der GPU, also deiner Grafikkarte, ausgeführt werden. Im Kern bestimmen sie, wie ein Material aussieht. Die ganzen Einstellungsmöglichkeiten, die du für jedes Material hast, sind technisch nur Parameter, die zur Berechnung des Materials an das jeweilige Shader-Programm weitergegeben werden.

Grundsätzlich ist es so, dass komplexe Shader häufig schönere und realistischere Ergebnisse liefern können, einfache Shader aber deutlich schneller zu berechnen sind. Universale Shader, wie der *Standard Shader* von Unity, sind sehr einfach zu verwenden und für jeden Typ von Material einsetzbar, egal ob Glasscheibe oder Metalltür. Die gute Anpassbarkeit erhält man jedoch nur zu Lasten der Performance, weshalb es für schwächere Hardware sogenannte *Mobile Shader* gibt. Diese haben meist weniger Einstellungsmöglichkeiten, dafür lasten sie die GPU aber auch weniger aus.

Wegen der besseren Performance, die uns in VR sehr wichtig ist, ist es häufig sinnvoll, für Daydream, GearVR- und Cardboard-Spiele die *Mobile Shader* anstelle des *Standard Shaders* zu verwenden. Nachfolgend werde ich die beiden Shader-Arten einmal genauer vorstellen.

Wenn du ein neues Material anlegst, verwendet es standardmäßig immer den *Standard Shader*. Du kannst den Shader für jedes Material einzeln über das **SHADER**-Menü des jeweiligen Materials ändern, wie du in Bild 6.26 sehen kannst.

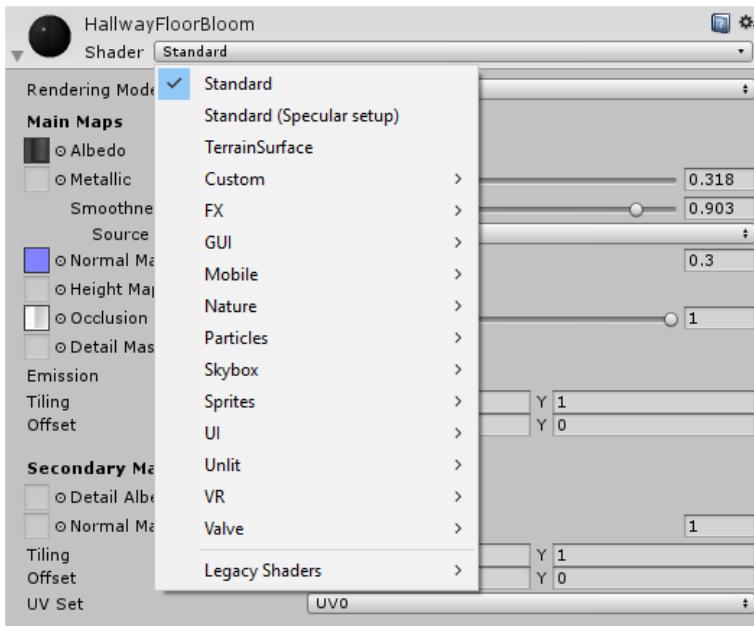


Bild 6.26 Das Shader-Menü findest du ganz oben im Inspector-Bildschirm des jeweiligen Materials.

6.2.1 Der Standard Shader

Der Unity Standard Shader ist ein sehr mächtiger, universaler Shader, der für jede erdenkliche Art von Materialien (Stein, Holz, Glass, Plastik etc.) verwendet werden kann. Der Shader ist, wie in Kapitel 6.1.7 bereits kurz erwähnt, ein *Physically Based Shader* (PBS). Er kann also verwendet werden, um eine Technik namens *Physically Based Rendering* (PBR) zu verwenden. Bei dieser Technik imitiert der Shader die physikalischen Eigenschaften der gewünschten Oberfläche, um Dinge wie Reflexion und Streuung des reflektierten Lichtes zu berechnen.

Im Bereich des *Physically Based Renderings* gibt es zwei gängige Möglichkeiten, eine Materialoberfläche zu definieren: entweder darüber, wie metallisch (*Metallic*) oder wie spiegelnd (*Specular*) sie aussieht. Unity unterstützt beide Techniken. Der normale *Standard Shader* verwendet die *Metallic*-Variante und der Shader *Standard (Specular Setup)* verwendet die *Specular*-Variante. Die naheliegende Vermutung, dass der Shader Standard nur für metallische Oberflächen und der *Standard (Specular Setup)* nur für spiegelnde Flächen geeignet sei, ist falsch. Beide Shader-Varianten können benutzt werden, um jede beliebige Art von Oberfläche für das *Physically Based Rendering* zu definieren. Da beide Varianten gängig sind, haben die Unity-Entwickler die Engine nicht auf eine der beiden limitiert, sondern stellen dir beide zur Verfügung.

Wir schauen uns jetzt einmal die einzelnen Funktionen des *Standard Shaders* an. Die Funktionen des Shaders sind in vier Bereiche eingeteilt, wie du in Bild 6.27 sehen kannst. Der Shader unterstützt eine große Anzahl von verschiedenen Effekten, das bedeutet allerdings

nicht, dass du auch jeden Effekt verwenden musst. Du musst also nicht jedem Feld des Shaders einen Wert zuweisen. Lässt du ein Feld frei, wie zum Beispiel die Height und Detail Map in Bild 6.27, wird die jeweilige Funktion automatisch deaktiviert und nicht verwendet.

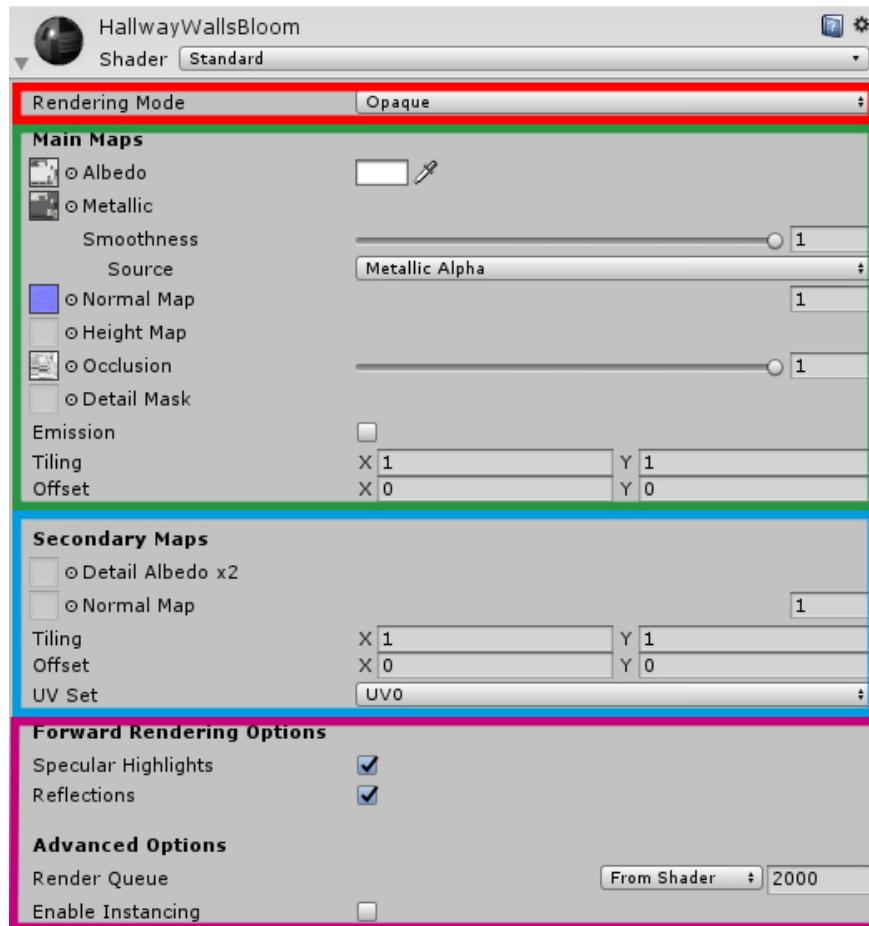


Bild 6.27 Rendering Mode (rot), Main Maps (grün), Secondary Maps (blau), Erweiterte Optionen (lila)

Rendering Mode

Der *Rendering Mode* beschreibt, wie das Material grundsätzlich dargestellt werden soll:

- **Opaque:** Dies ist der Standardwert, der auch für die meisten Materials geeignet ist. Diese Option wird für Materials verwendet, die keine Transparenz aufweisen und durch die man nicht hindurchsehen können soll.
- **Transparent:** Für durchsichtige Materials wie Glas solltest du diese Option verwenden. Hier bestimmt der Alpha-Wert der *Albedo Map* (auch *Diffuse Map* genannt) die Transparenz des Objektes. Wenn die Grafik selbst keine Transparenz aufweist, kannst du über die daneben liegende *Color*-Eigenschaft einen Alpha-Wert („A“) bestimmen.

- **Cutout:** Bei dieser Option musst du eine *Albedo Map* mit Transparenz verwenden. *Cutout* erlaubt keine halbdurchlässigen Bereiche, es gibt nur „sichtbar“ und „unsichtbar“. Über den *Alpha Cutoff*-Regler kannst du bestimmen, ab welchem Alpha-Schwellwert ein Pixel der *Texture* im Material angezeigt werden soll.
- **Fade:** Diese Option ist ähnlich wie *Transparent*, allerdings sollte diese Option nicht für durchsichtige *Materials* wie Glas verwendet werden, sondern für *Materials* von *GameObjects*, die man im Spiel mit einem Übergang aus- und einblenden möchte. Während bei *Transparent* die Spiegelung immer sichtbar ist (wie bei Glas), wird sie im *Fade*-Modus mit ausgeblendet.

In Bild 6.28 siehst du die vier Rendering Modes und ihre Unterschiede nebeneinander.

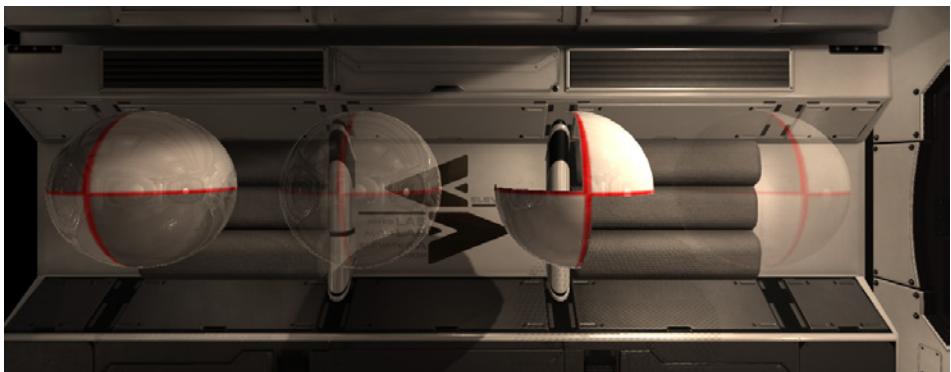


Bild 6.28 Die Render Modes von links nach rechts: Opaque, Transparent, Cutout, Fade. Man kann gut sehen, wie bei Transparent die Reflexion bestehen bleibt, während bei Fade das gesamte Material ausgeblendet wird.

Main Maps

Jede der Eigenschaften in dem *Main-Maps*-Bereich kontrolliert einen optischen Effekt des Materials:

- **Albedo:** Diese Eigenschaft kombiniert eine optionale *Texture* und den angegebenen Farbwert, um das grundlegende Aussehen des Materials zu bestimmen. Der Farbwert färbt die *Texture*. Wird der Farbwert auf Weiß gesetzt, wird die *Texture* ohne Änderungen verwendet. Über den Alpha-Wert der Farbe und den Alpha-Wert der *Texture* kann, je nach *Render Mode*, die Transparenz des Materials bestimmt werden.
- **Metallic:** Diese Eigenschaft kann entweder über eine *Texture* oder über einen Wert zwischen 0 und 1 angegeben werden. Dieser Wert bestimmt, wie metallisch die Oberfläche sein soll. Über den dazugehörigen *Smoothness*-Wert kann zudem noch bestimmt werden, wie glatt oder rau die Oberfläche sein soll. Je rauer die Oberfläche ist, desto diffuser wirken die Reflexionen auf der Oberfläche. Wenn du eine *Texture* verwendest, kann der Alpha-Wert der *Texture* verwendet werden, um den *Smoothness*-Wert des Materials zu bestimmen.
- **Normal Map:** Die *Normal Map* bestimmt, wie uneben das Material ist. Diese Informationen werden bei der Lichtberechnung berücksichtigt, um herausstehende Teile heller und vertiefte Bereiche dunkler darzustellen. Hierdurch entsteht ein 3D-Effekt, der die Oberflä-

che realistischer wirken lässt. Die Farben der *Normal Map* beschreiben den Normalen-Vektor der Oberfläche an der jeweiligen Stelle.

- **Height Map:** Diese Texture teilt dem Shader mit, welche Elemente des Materials sichtbar hervorstehen sollen. Über das sogenannte *Bump-Mapping* erhält die Oberfläche des Materials dann einen 3D-Effekt, als würde es sich bei Oberfläche um ein 3D-Modell handeln.
- **Occlusion Map:** Hier kannst du eine in Graustufen erstellte *Texture* angeben, welche Informationen für das *Ambient Occlusion* (dt. „Umgebungsverdeckung“) enthält. *Praxisbeispiel:* Die durch die *Height Map* erzeugten Erhöhungen werfen keine Schatten auf dem Material. Über eine *Occlusion Map* kannst du diese per Hand hinzufügen. Die *Occlusion Map* verhindert zudem auch Reflexionen und Spiegelungen in schattierten Bereichen, wodurch ein realistischeres Aussehen erreicht wird. Siehe Bild 6.29 für einen Vergleich.
- **Detail Mask:** Hier kannst du eine in Graustufen gehaltene *Texture* einfügen, die bestimmt, an welchen Stellen die *Secondary Maps* verwendet werden sollen. *Secondary Maps* erlauben es dir, bestimmte Bereiche deines Materials mit besonders hohem Detailgrad darzustellen.
- **Emission:** Mit der *Emission*-Checkbox aktivierst du das *Emission*-Feature des Shaders, welches in Kapitel 6.1.5 bereits vorgestellt wurde. Du kannst eine Emission Map angeben, welche bis auf die Stellen des Materials, die leuchten sollen, vollständig schwarz ist. Ist deine Emission-Map bunt und nicht nur in Graustufen gehalten, leuchtet das Material in der jeweiligen Farbe. Über das Farbauswahlfeld kann die Emission-Map zudem noch nachträglich gefärbt werden. Der Zahlenwert rechts daneben bestimmt die Intensität des Lichtes. Über die Auswahlliste *Global Illumination* kannst du auswählen, ob das abgestrahlte Licht in *Lightmaps* gespeichert oder in *Realtime* berechnet werden soll.

Bild 6.29 zeigt dir dieselbe Albedo-Texture in vier verschiedenen Materials. Das erste Material verwendet nur die Albedo-Texture selbst. Nach und nach verwenden die weiteren Materials mehr Features und erzeugen so, Stück für Stück, eine realistischere Darstellung der Oberfläche. In dem letzten Bild ist gut der 3D-Effekt erkennbar, der durch das Zusammenspiel von Normal, Height und Occlusion Map entsteht.



Bild 6.29 Die vier wichtigsten Main Maps im Vergleich. Am besten sieht man den optischen Unterschied jedoch in der Engine selbst.

Secondary Maps

Secondary Maps werden dazu benutzt, bestimmte Bereiche deines Materials mit extra vielen Details anzuzeigen, für den Fall, dass der Spieler ganz nah herangeht. Über die *Detail Mask*-Eigenschaft in den *Main Maps* bestimmst du, wo die *Secondary Maps* angewendet werden sollen. Anschließend kannst du eine nahtlos wiederholbare *Texture* und optional auch eine dazu passende *Normal Map* als *Secondary Maps* angeben. Die *Secondary Map* wird dann mit einem hohen Detailgrad in den Bereichen, die in der *Detail Mask* markiert wurden, wiederholend angezeigt. Ein solcher Detailgrad wäre ohne *Secondary Maps* nur möglich, wenn du *Main Maps* mit einer sehr hohen Auflösung oder viele einzelne Materials verwenden würdest. Diese Alternativen würden allerdings viel mehr Arbeitsspeicher und auch Performance kosten. Die *Secondary-Map*-Variante erlaubt also, mit zwei niedriger aufgelösten *Textures* denselben Effekt mit einem einzigen Material zu erreichen. Ein weiterer Vorteil ist, dass du den Detailgrad der *Secondary Maps* bequem über den *Inspector* verändern kannst.

Bild 6.30 zeigt einen Wachturm ohne und mit *Secondary Maps* für die Details der Holztextur.



Bild 6.30 Dieser Wachturm nutzt eine einzige Textur für alle Teilelemente des gesamten Turms (Main Albedo, links im Bild). Da die Texturen für alle Elemente auf der einen Grafik Platz finden müssen, ist für jedes einzelne Element nicht viel Platz. Deshalb sieht das Ergebnis sehr verschwommen aus (linke Bildhälfte). Mithilfe einer *Secondary Map*, welche zu der bestehenden Textur hinzugefügt wird, kann der Detailgrad zurückgewonnen werden (rechte Bildhälfte).

Erweiterte Optionen

Als Letztes folgen nun noch die erweiterten Einstellungen, die sich in *Forward Rendering Options* und *Advanced Options* unterteilen. In den meisten Fällen musst du dich mit beiden Optionengruppen nicht näher beschäftigen.

Die *Forward Rendering Options* funktionieren, wie der Name schon sagt, nur mit dem *Forward Rendering Path* und nicht mit aktiviertem *Deferred Rendering*. Sie erlauben dir in erster Linie, die Reflexion von Licht (*Specular Highlights*) und die Spiegelung der Scene auf der Oberfläche des Materials (*Reflections*) getrennt voneinander zu deaktivieren. Für gewöhnlich sind diese Funktionen, wie im echten Leben, aneinandergekoppelt.

Die *Advanced Options* bietet dir zwei Optionen: *Rendering Queue* erlaubt dir den Zeitpunkt, wann dieses Material gerendert wird, anzupassen. *Enable Instancing* erlaubt dir, das sogenannte *GPU Instancing* für dieses Material zu aktivieren. An diesen Optionen solltest du nur etwas ändern, wenn dir die Begriffe bereits etwas sagen.

6.2.2 Mobile Shader

Mobile Shader sind anders als der *Standard Shader* keine universalen Shader, sondern stark optimierte Shader, die jeweils immer nur eine bestimmte Aufgabe haben. Diese Shader sind speziell für mobile Geräte optimiert, die weniger Leistung haben. Bei einem modernen Smartphone muss man diese jedoch nicht mehr zwingend verwenden, da die Smartphones immer stärker werden. Da wir jedoch Virtual-Reality-Spiele entwickeln, die auch moderne Smartphones an ihre Grenze bringen können, ist es bei GearVR- und GoogleVR-Spielen

trotzdem sinnvoll, diese Shader anstelle des *Standard Shaders* zu verwenden. Hin und wieder liest man sogar die Empfehlung, die *Mobile Shader* auch für PC-VR-Spiele zu verwenden, da sie auch dort deutlich an Performance sparen.

In manchen Fällen ist der Unterschied zum *Standard Shader* kaum sichtbar, zum Beispiel, wenn du ohnehin kaum die besonderen Funktionen des *Standard Shaders* nutzt. Wenn du einen *Mobile Shader* verwendest, bedeutet das nicht, dass dein Spiel hässlich wird, auch mit den mobilen Shadern können gute Ergebnisse erzielt werden, wie dir Bild 6.31 zeigt. Es benötigt teilweise lediglich ein wenig mehr Arbeit, da du hochwertigere Assets verwenden solltest.

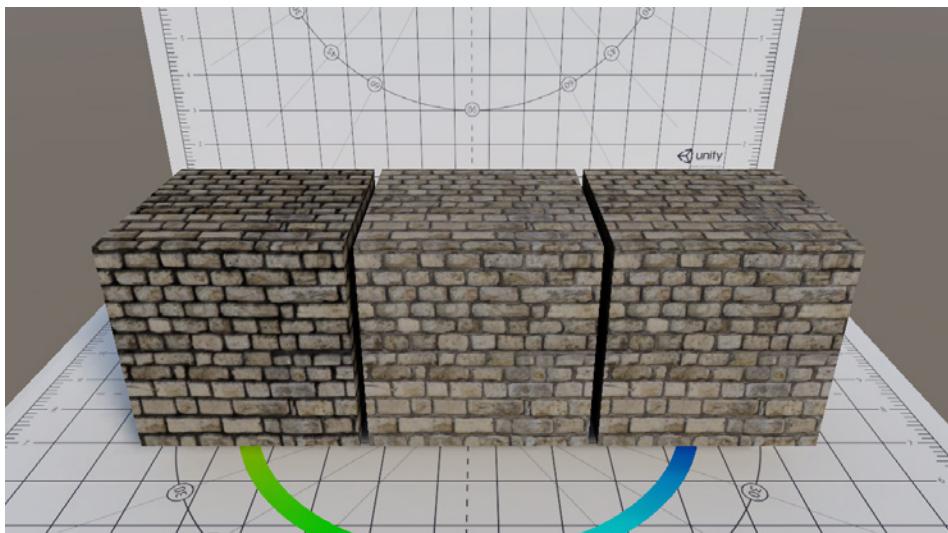


Bild 6.31 Standard Shader und Mobile Shader im Vergleich.
Links: Standard Shader mit allen Main Maps, wie in Bild 6.29.
Mitte: Standard Shader nur mit Albedo und Normal Map.
Rechts: Mobile Shader Albedo und Normal Map

Unitys mitgelieferte *Mobile Shader* unterstützen unter anderem folgende Funktionen nicht, die der *Standard Shader* unterstützt: *Height Maps*, *Occlusion Maps*, *Reflections*, *Emission*, *Secondary Maps*. Im nächsten Abschnitt stelle ich dir die gebräuchlichsten Mobile Shader kurz vor.

6.2.2.1 Standard-Mobile-Shader

Die drei Mobile Shader *Mobile Diffuse*, *Mobile Bumped Diffuse* und *Bumped Specular* ersetzen den *Standard Shader* im Segment der Mobile Shader. Welchen der drei du verwenden solltest, hängt davon ab, welche Textures („Maps“) du für dein Material verwenden möchtest.

- Der *Diffuse* Shader unterstützt ausschließlich eine *Base Texture* (nur „Albedo“).
- Der *Bumped Diffuse* Shader unterstützt eine *Base Texture* und eine *Normal Map*.

- Der *Bumped Specular* Shader unterstützt neben der *Base Texture* und einer *Normal Map* auch noch eine *Gloss Map*, diese muss allerdings in die *Base Texture* über *Alpha-Kanal* integriert werden. Zusätzlich kann über den *Shininess-Slider* noch bestimmt werden, wie glänzend die Oberfläche ist.

Du solltest immer den kleinstmöglichen Shader verwenden, um eine optimale Performance zu erreichen.

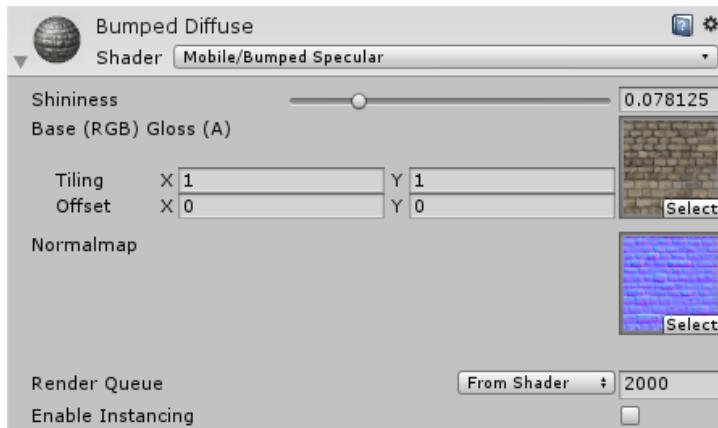


Bild 6.32 Eine beispielhafte Konfiguration des Mobile Bumped Specular Shaders

6.2.2.2 Mobile Unlit (Supports Lightmap)

Der performanteste und daher empfehlenswerteste Shader ist jedoch keiner der zuvor vorgestellten, sondern der *Mobile Unlit (Supports Lightmap)* Shader. Dieser Shader unterstützt überhaupt keine Echtzeit-Lichter und nur eine *Base Map*, keine Normal- oder Specular-Werte. Dieser Shader eignet sich gut für Projekte mit einem minimalistischen Look oder Projekte mit sehr hochwertigen Texturen. Je weniger detailliert deine Texturgrafik ist, desto stärker fällt es auf, wenn du einfache Shaders verwendest. Bild 6.33 zeigt drei *Materials*, die mit dem *Mobile Unlit (Supports Lightmap)* Shader erstellt wurden. Der Unterschied zu den anderen Mobile Shadern ist nicht erkennbar, solange du keine Echtzeit-Lichteffekte benötigst.

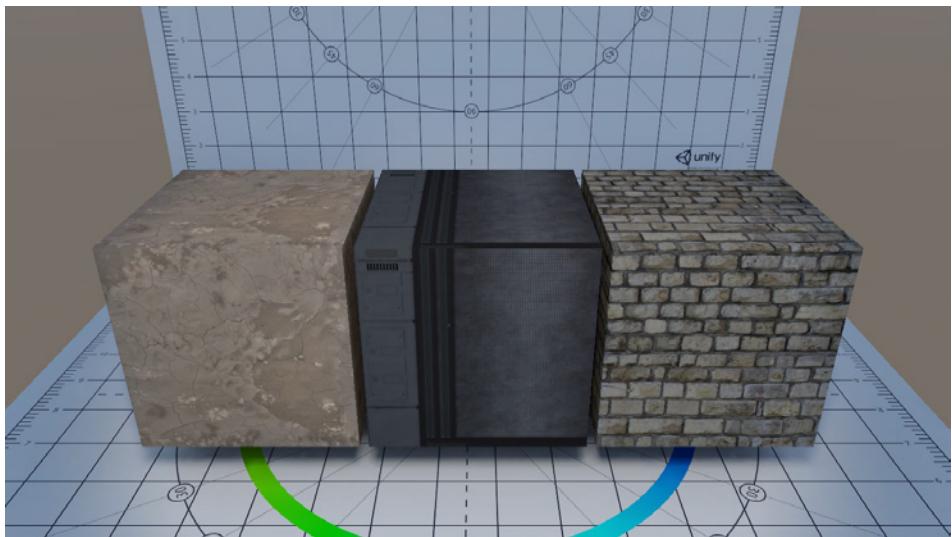


Bild 6.33 Diese Materials nutzen alle den Mobile Unit (Supports Lightmap) Shader.

■ 6.3 Audio

Wie du in den Kapiteln zuvor schon erfahren hast, ist gutes Audiodesign ein sehr wichtiges Thema in Videospielen und trägt stark zur Immersion bei. Deshalb sollten Audioeffekte niemals als optional betrachtet werden. Häufig müssen die Sounds nicht 100% akkurat wie im echten Leben klingen, damit sie das Spielerlebnis verbessern. Kein Sound ist stets schlechter als ein nicht 100% akkurate Sound. Wenn der Soundeffekt immer derselbe ist, egal ob du mit der Brechstange gegen eine Steinwand oder einen Holzschränk schlägst, fällt es vielen Spielern erst einmal nicht auf. Für jedes Material unterschiedliche Sounds zu haben, wäre natürlich besser, benötigt aber auch eine entsprechende Sound-Datenbank. Gerade als Hobby- oder Indie-Entwickler mit sehr geringem Budget ist es häufig viel Aufwand, eine umfassende Sound-Bibliothek aufzubauen.

Das Unity-Audio-System besteht im Kern aus folgenden Bestandteilen:

- **AudioClip:** Das ist Unitys Bezeichnung für Audio-Assets (mp3, wav, ogg etc.). Mehr Infos dazu findest du im Quickstart-Guide unter „Assets“ am Anfang des Buches.
- **AudioSource:** Dieses Component wird verwendet, um *AudioClips* abzuspielen, und kann sowohl für 2D- als auch für 3D-Sounds konfiguriert werden.
- **AudioMixer:** *AudioMixer* erlauben es dir, mehrere *AudioSources* in Gruppen zusammenzufassen und die Gruppen dann an einer zentralen Stelle im Unity-Editor abzustimmen (Laustärke, Effekte, Verzerrung etc.).
- **AudioListener:** Das sind die Ohren des Spielers. Pro Scene darf es nur einen einzigen *AudioListener* geben. Dieses Component wurde ebenfalls bereits im Quickstart-Guide des Buches angesprochen.

Während du die drei Bestandteile *AudioClip*, *AudioSource* und *AudioListener* immer benötigst, ist der *AudioMixer* optional. Der *AudioMixer* ist jedoch ein sehr mächtiges Werkzeug, welches es dir erlaubt, die gesamte Soundkulisse deines Spiels an einer zentralen Stelle zu verwalten.

Wenn du später ein Audio-SDK für *Spatial-Audio* verwenden möchtest, macht es Sinn, dass du dich zunächst mit dem Unity-internen System vertraut machst, da die Audio-SDKs in der Regel Erweiterungen für Unitys interne Sound-Engine sind.

6.3.1 AudioSources

Die *AudioSource* ist Unitys universales Sound-Component für Soundeffekte, Hintergrundmusik und auch allen anderen Arten von Geräuschen. *AudioSource*s können, wie fast alle Components, sowohl über den *Inspector* als auch via Code konfiguriert und bedient werden. Das ist besonders für die Fälle interessant, wo Soundeffekte nur bei bestimmten Ereignissen abgespielt werden sollen und nicht wie Hintergrundmusik die ganze Zeit über. Um eine *AudioSource* anzulegen, kannst du entweder ein *AudioSource*-Component zu einem existierenden *GameObject* hinzufügen oder in der *Hierarchy* über das Create-Menü ein neues *GameObject* mit einer *AudioSource* erstellen: **CREATE/AUDIO/AUDIO-SOURCE**.

Zunächst betrachten wir einmal die Konfigurationsmöglichkeiten im Unity Inspector: Bild 6.34 zeigt dir eine neu erstellte *AudioSource*. Alle Optionen stehen dir mit dem gleichen Namen auch im Code zur Verfügung, aber dazu kommen wir anschließend.

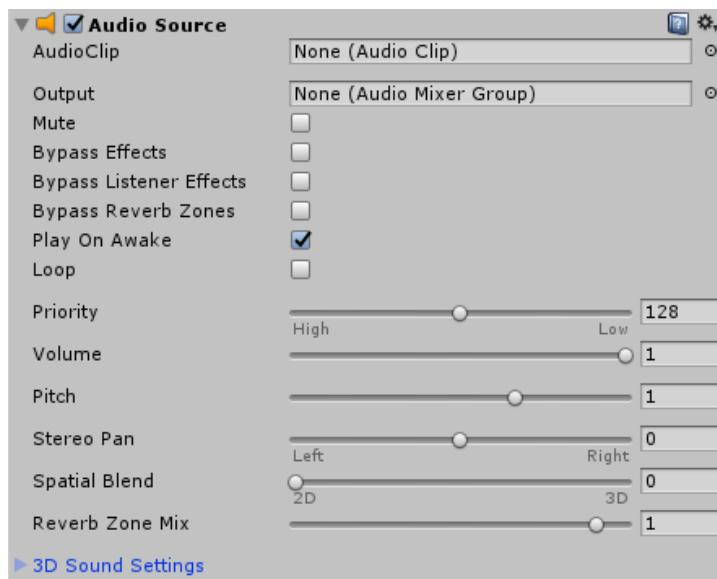


Bild 6.34 Der Inspector-Editor einer AudioSource

Audio Sources (dt. „Tonquellen“) können sowohl positionsabhängiges 3D-Audio abspielen als auch globale Hintergrund-Sounds. Diese sind unabhängig von der Position des Spielers immer gleich laut.

Nachfolgend erkläre ich dir kurz die Optionen des Components:

- **AudioClip:** Hier musst du den Soundeffekt angeben, der abgespielt werden soll.
- **Output:** Hier kannst du eine *AudioMixerGroup* angeben, welche wiederum zu einem *AudioMixer* gehört. Dadurch ist es dir möglich, alle *AudioSources* einer Gruppe über ein zentrales Mischpult zu steuern.
- **Mute:** Über diese Option kannst du den Sound stumm stellen. Der Sound wird dann trotzdem weiter abgespielt, er ist nur nicht mehr hörbar.
- **Bypass Effects:** Mit dieser Checkbox kannst du Filtereffekte, die sich auf diese Soundquelle beziehen, kurzfristig deaktivieren und wieder aktivieren.
- **Bypass Listener Effects:** Mit dieser Option kannst du Effekte, die sich auf den *AudioListener* beziehen, für diese AudioSource kurzfristig ein- oder ausschalten.
- **Bypass Reverb Zones:** Hiermit kannst du kurzfristig *Reverb Zones* für diese Audio-Quelle deaktivieren.
- **Play On Awake:** Wenn du diese Option aktivierst, wird der *AudioClip* dieser *AudioSource* direkt abgespielt, wenn das dazugehörige GameObject aktiviert wird.
- **Loop:** Hier stellst du ein, ob der *AudioClip* in einer Endlosschleife abgespielt werden soll.
- **Priority:** Die Priorität einer Audio-Quelle gibt an, wie wichtig sie in der Scene ist. Für Hintergrundmusik solltest du „0“ (am wichtigsten) angeben, für sonstige Sound-Effekte kannst du den Standardwert stehen lassen.
- **Volume:** Hier kannst du die Laustärke der Audio Source festlegen.
- **Pitch:** An dieser Stelle kannst du die Tonhöhe für diesen Clip angeben, mit der Tonhöhe ändert sich auch die Abspielgeschwindigkeit.
- **Stereo Pan:** Hier kannst du für 2D-Sound-Effekte die Position (links oder rechts) angeben.
- **Spatial Blend:** Hier stellst du ein, ob es sich bei dieser *AudioSource* um eine 2D- oder 3D-Soundquelle handelt. Eine *2D-Soundquelle* spielt den Sound unabhängig von der Position der Soundquelle und des Spielers ab. Die Laustärke für jedes Ohr hängt ausschließlich von den Werten *Volume* und *Stereo Pan* ab. In den meisten Fällen wird dies für Hintergrundmusik verwendet. Bei einer *3D-Soundquelle* wird die Lautstärke für jedes Ohr abhängig von der Position der *AudioSource* und des *AudioListeners* (Spielers) berechnet. Man bekommt also den Eindruck, dass sich die Soundquelle in der 3D-Welt befindet. In den meisten Fällen wirst du diese Einstellung verwenden wollen.
- **Reverb Zone Mix:** Hier kannst du einstellen, wie stark diese *AudioSource* durch *Reverb Zones* beeinflusst wird.

3D Sound Settings

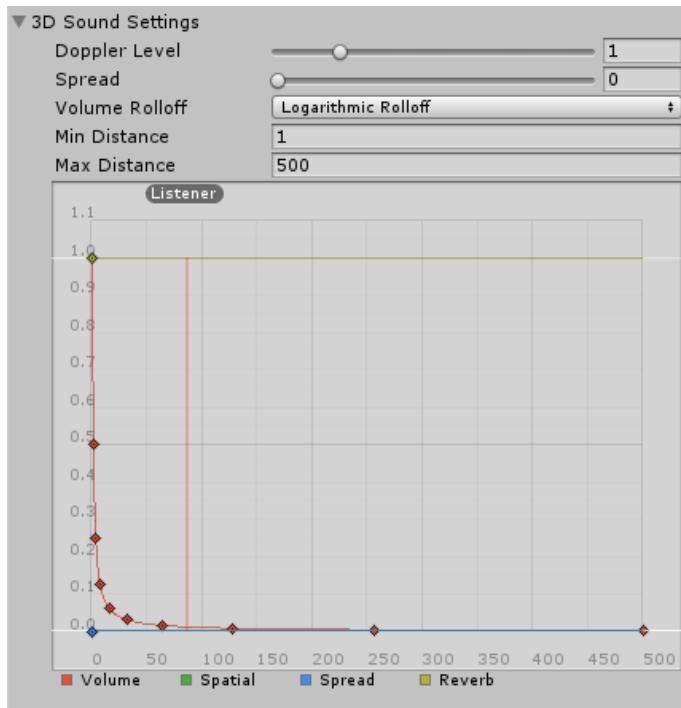


Bild 6.35 Die 3D-Sound-Einstellungen kannst du über den Pfeil daneben ausklappen.

Diese Einstellungen sind nur für *AudioSources* relevant, deren *Spatial Blend* auf *3D* steht. Hier kannst du zum Beispiel einstellen, wie weit diese *AudioSource* hörbar sein soll und ob das Geräusch eher aus einer sehr bestimmten Richtung oder einer groben Richtung kommen soll. Die Unity Sound Engine bezieht Wände und andere Hindernisse übrigens nicht automatisch in die Berechnung der Laustärke mit ein. Damit der *AudioClip* in verwinkelten Gebäuden trotzdem nicht durch alle Wände hindurch hörbar ist, kannst du eine detaillierte Einstellung der Hörweite vornehmen.

- **Doppler Level:** Hier kannst du angeben, wie stark der Doppler-Effekt auf diese *AudioSource* angewendet werden soll.
- **Spread:** Mit diesem Attribut kannst du einstellen, wie präzise der Sound dieser *AudioSource* in einer Audio-Anlage mit mehreren Kanälen (z. B. 5.1-Systeme) sein soll. Je kleiner diese Zahl, desto präziser kann die Quelle geortet werden. Je größer dieser Wert, desto mehr verteilt sich der Sound auch auf die angrenzenden Kanäle. Für weit entfernte, laute Geräusche, wie Explosionen, kann es zum Beispiel sinnvoll sein, eine höhere Streuung zu verwenden.
- **Min Distance:** Solange der Abstand des Spielers zu dieser *AudioSource* kleiner ist als die *Min Distance*, wird der *AudioClip* mit voller Lautstärke abgespielt. Erst danach beginnt der sogenannte *Rolloff* (Abfall der Lautstärke).
- **Max Distance:** Hier kannst du die maximale Hörweite dieser *AudioSource* angeben.

- **Rolloff Mode:** Bewegt sich der Spieler zwischen *Min Distance* und *Max Distance*, wird die *Audio Source* stetig leiser. Mit dieser Option kannst du angeben, wie der Abfall aussehen soll. Der Standardwert ist *Logarithmic Rolloff*, bei dem der Sound sehr schnell leiser wird. *Linear Rolloff* beschreibt einen gleichmäßigen Abfall der Lautstärke mit der Distanz. Alternativ zu diesen Optionen kannst du den Laustärkeabfall auch manuell in dem Diagramm unter dieser Option festlegen. In verwinkelten Gebäuden mit dicken Wänden ergibt es eventuell Sinn, dass ein Geräusch bei geringer Distanz sehr laut ist, aber auch schnell leiser wird, wenn man sich von ihm entfernt. Auf einem weiten Feld wären Geräusche, im Gegensatz dazu, auch bei großer Distanz noch gut hörbar.

Die Reichweite einer *AudioSource* kannst du auch in der *Scene View* ändern. Wenn ein *GameObject* mit einem *AudioSource*-Component in der *Hierarchy* ausgewählt ist, wird um das *GameObject* herum eine Drahtgitter-Kugel angezeigt. Diese Kugel symbolisiert die maximale Reichweite der *AudioSource*. Durch Ziehen an den kleinen Würfeln kannst du die Reichweite bequem in der *Scene View* anpassen.



Bild 6.36 Die Kugel symbolisiert die maximale Hörweite der ausgewählten *AudioSource*.



Hinweis zu diesen Optionen in Spatial-Audio-SDKs:

Diese 3D-Sound-Optionen werden teilweise nicht verwendet oder automatisch bestimmt, wenn du ein Audio-SDK für Spatial-Sound verwendest.

6.3.2 *AudioSource* via Script steuern

Im Code stehen dir, wie schon erwähnt, alle Eigenschaften des Components zur Verfügung, die du auch im Inspector sehen kannst. Dazu kommen noch einige Methoden, mit denen du das Abspielverhalten kontrollieren kannst. Die wichtigsten davon stelle ich dir jetzt einmal vor:

■ Play() : void

Spielt den aktuellen `clip` von Anfang an ab. Wird der Clip bereits wiedergegeben, wird die Wiedergabe abgebrochen und am Anfang neu begonnen.

■ Pause() : void**UnPause() : void**

Pausiert den aktuellen `clip` an der aktuellen `time` bzw. setzt die Wiedergabe fort.

■ Stop() : void

Stoppt das Abspielen des aktuellen `clip` und setzt die `time` auf 0.

■ PlayDelayed(delay : double) : void

Ruft `Play()` mit einer Verzögerung auf. Der Parameter gibt an, wie viele *Sekunden* gewartet werden soll, bevor der aktuelle `clip` gestartet wird.

■ PlayOneShot(otherClip : AudioClip [,volume : float]) : void

Normalerweise kann eine `AudioSource` immer nur einen einzigen `AudioClip` gleichzeitig abspielen. `PlayOneShot(...)` stellt jedoch eine Ausnahme dar. Über diese Methode kannst du zusätzlich zum aktuellen `clip` noch beliebig viele andere `AudioClips` abspielen. Im Gegensatz zur normalen `Play()`-Methode wird die aktuelle Wiedergabe nicht unterbrochen, sondern der `otherClip` wird gleichzeitig abgespielt. Optional kann eine von der Einstellung der `AudioSource` unabhängige Laustärke für den Clip übergeben werden. In der Praxis wird diese Methode zum Beispiel für den Feuer-Sound einer Waffe verwendet, deren Feuerrate schneller ist als die Länge des Schuss-Sounds.

Um eine `AudioSource` verwenden zu können, brauchst du zunächst einen Verweis auf eine solche. Diesen erhältst du zum Beispiel mittels `GetComponent` oder über eine `public`-Variable im `Inspector`. Außerdem benötigst du natürlich Verweise auf die `AudioClips`, die du abspielen möchtest. Diese holst du dir am besten über eine `public`-Variable, welche du dann im `Inspector` mit dem gewünschten `AudioClip` ausstattest (siehe Listing 6.1).

Listing 6.1 Ein Beispiel für einen Background-Sound und ein paar Soundeffekte, die mit einem Script gesteuert werden

```
public AudioSource audioSource;
public AudioClip backgroundMusic;
public AudioClip shootSFX;
public AudioClip reloadSFX;

public void PlayBackgroundMusic(){
    audioSource.clip = backgroundMusic;
    audioSource.loop = true;
    audioSource.volume = 0.8f;
    audioSource.Play();
}

public void Fire(){
    audioSource.PlayOneShot(shootSFX);
}

public void Reload(){
    audioSource.PlayOneShot(reloadSFX, 0.5f);
}
```

6.3.3 Temporäre AudioSource via Script

Manchmal braucht man mal eben schnell eine *3D-AudioSource* an einer bestimmten Stelle und dann nie wieder. Anstelle eine *AudioSource* an die gewünschte Position zu verschieben und den Sound abzuspielen, kannst du auch die statische Methode *AudioSource.PlayClipAtPoint(...)* verwenden.

Diese Methode erzeugt an der angegebenen Position eine *AudioSource*, spielt den gewünschten *AudioClip* mit der angegebenen Lautstärke einmalig ab und zerstört die *AudioSource* danach automatisch wieder. Du musst dich also um nichts kümmern. Aus Performance-Gründen solltest du diese Methode jedoch nicht für Sounds verwenden, die regelmäßig abgespielt werden, z.B. der Schuss-Sound einer Waffe.

Es ist immer dann sinnvoll, eine temporäre *AudioSource* zu verwenden, wenn du nur einmalig einen Sound an der Stelle abspielen möchtest und sich nicht ohnehin schon ein *GameObject* an der Stelle befindet. In der Praxis kann das zum Beispiel das Geräusch eines Kugeleinschlags auf einer Wand sein.

Da die Methode *static* (statisch) ist, benötigst du keinen Verweis auf eine bestimmte *AudioSource*, um sie aufrufen zu können, wie Listing 6.2 zeigt.

Listing 6.2 Beispiel-Script für eine temporäre *AudioSource*

```
public AudioClip hitWallSFX;
public void playHitWallSoundAtPosition(Vector3 hitPosition){
    AudioSource.PlayClipAtPoint(hitWallSFX, hitPosition, 1f);
}
```

6.3.4 Reverb Zones

Reverb Zones sind Bereiche, die du in deiner Scene definieren kannst. Alle Sounds, die innerhalb einer *Reverb Zone* abgespielt werden, erhalten einen konfigurierbaren *Halleffekt* (*Reverb*). Sie können also beispielsweise genutzt werden, um in Höhlen oder großen Hallen eine passende akustische Stimmung zu erzeugen. Um eine *Reverb Zone* anzulegen, musst du das entsprechende Component zu einem *GameObject* hinzufügen. Die Position des *GameObjects* ist dann das Zentrum der angelegten *Reverb Zone*. Du findest das Component unter: **ADD COMPONENT/AUDIO/AUDIO REVERB ZONE**.

Bei *Reverb Zones* wird, anders als man es erwartet, nicht geprüft, ob sich die Soundquelle in der *Reverb Zone* befindet, sondern ob sich der Spieler (genauer der *AudioListener*) in einer *Reverb Zone* befindet. Ist das der Fall, erhalten alle abgespielten Sounds, die er hört, einen Hall, auch wenn die Soundquellen außerhalb der *Reverb Zone* liegen.

Reverb Zones können sehr komplex konfiguriert werden, wir werden uns an dieser Stelle jedoch auf die wichtigsten Einstellungen beschränken:

- **Min Distance:** Diese Option legt die Größe der eigentlichen *Reverb Zone* fest, hier ist der Halleffekt in voller Laustärke hörbar. *Reverb Zones* sind stets kugelförmig.
- **Max Distance:** Häufig möchtest du nicht, dass beim Verlassen der *Reverb Zone* der Hall schlagartig aufhört, sondern man möchte einen weichen Übergang schaffen. Dieser Para-

meter bestimmt einen zweiten, größeren kugelförmigen Bereich. In dem Bereich zwischen *Min Distance* und *Max Distance* nimmt der Hall immer stärker ab, sodass man ihn an der Grenze der äußeren Kugel kaum noch hören kann.

- **Reverb Preset:** Hier kannst du aus diversen Voreinstellungen wählen, wie sich der Hall anhören soll: Soll es sich eher anhören wie in einer Höhle oder wie in Häuserschluchten einer Großstadt? In den meisten Fällen ist es am einfachsten, verschiedene Optionen auszuprobieren, bis du eine gute gefunden hast. Über die Option *User* kannst du den Klang vollkommen frei konfigurieren.



Hinweis zu Reverb Zones in Spatial-Audio-SDKs:

Teilweise verwenden Spatial-Audio-SDKs eigene Alternativen zu Reverb Zones. Manche können die Hall- und Klang-Parameter sogar vollkommen automatisch, basierend auf deiner Scene und der gebauten Geometrie, berechnen. In den meisten Fällen funktionieren sie aber ähnlich wie Unitys eigene Reverb Zones und bieten lediglich ein paar mehr Parameter.

6.3.5 Audio Mixer

Der *AudioMixer* gibt dir eine deutlich bessere und einfachere Möglichkeit, den Klang in deiner Scene zu bestimmen, als es mit der direkten Konfiguration von *AudioSources* allein möglich wäre. Einen neuen *AudioMixer* legst über das Kontextmenü im *Project Browser* an. Klicke mit der rechten Maustaste in den Ordner, wo der *AudioMixer* abgelegt werden soll, und wähle **CREATE/AUDIO MIXER**. Mit einem Doppelklick auf den angelegten *AudioMixer* öffnest du das „*Audio Mixer*“-Fenster, wo du deinen *AudioMixer* konfigurieren kannst. Bild 6.37 zeigt, wie das „*Audio Mixer*“-Fenster im Editor aussieht.

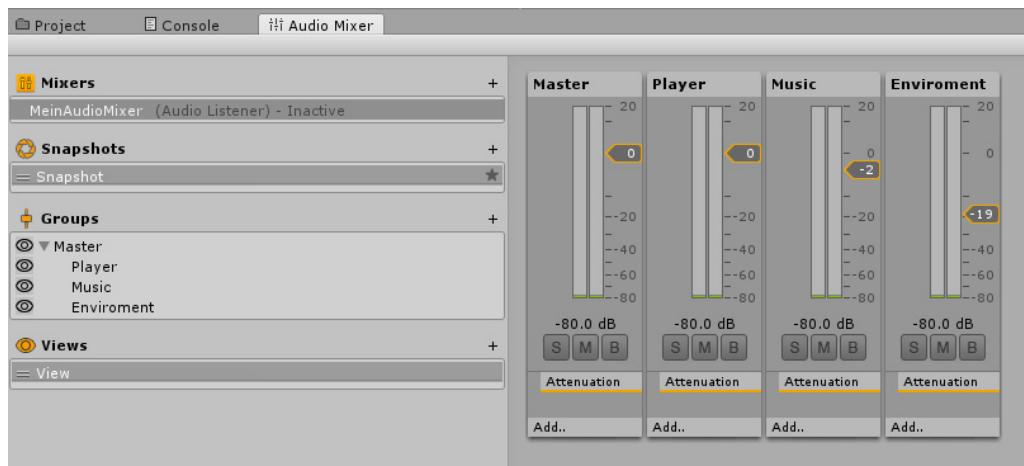


Bild 6.37 Ein Audio Mixer

Standardmäßig hat ein *AudioMixer* nur eine „Master“-Gruppe, welche alle Audiosignale steuert, die durch den Mixer geleitet werden. Unter der Master-Gruppe kannst du beliebig viele Untergruppen anlegen und sogar auch Untergruppen für Untergruppen, wenn du das möchtest. Bei diesem hierarchischen Aufbau kontrolliert die übergeordnete Gruppe immer auch alle Audiosignale ihrer Untergruppen.

In Bild 6.37 habe ich zusätzlich zur Master-Gruppe noch drei weitere Gruppen angelegt, für Sounds vom Spieler, Musik und Umgebungsgeräusche. Jede dieser Gruppen (inkl. der Master-Gruppe) können beliebig viele *AudioSources* über ihre *Output*-Option im *Inspector* zugewiesen werden. Jede der drei zusätzlichen Gruppen kontrolliert die Laustärke ihrer Audioquellen unabhängig von den anderen. Die Master-Gruppe kontrolliert, wegen des hierarchischen Aufbaus, jedoch auch die Laustärke der drei Untergruppen.

Man könnte die *AudioSources* noch weiter unterscheiden und beispielsweise unter der „Player“-Gruppe noch eine Gruppe „Gun Sounds“ und „Voice“ erstellen, damit man alle Sounds der Waffe des Spielers unabhängig von seiner Stimme kontrolliert. Spätestens wenn du selber ein wenig herumprobierst, wirst du sehen, dass das System eigentlich sehr einfach und intuitiv funktioniert.

Eine neue Gruppe legst du an, indem du in der Übersicht, links, bei „Groups“ auf das + klickst. Es wird dann eine neue Untergruppe für die aktuell ausgewählte Gruppe angelegt. Solltest du dich vertun, kannst du die Gruppen einfach mittels *Drag & Drop* in dem Fenster umsortieren.



„Gruppen“ und „Kanäle“

Gruppen sind vergleichbar mit Channels (dt. „Kanäle“) in der modernen Musikbearbeitung. Während man in der Musikbearbeitung jedoch üblicherweise nur ein Instrument pro Kanal hat, kannst (und solltest) du in Unity mehrere zusammengehörige *AudioSources* in einer einzigen Gruppe zusammenfassen.

6.3.5.1 Gruppen-Lautstärke und Effekte

Die *Laustärke* jeder Gruppe kannst du anpassen, indem du den gelben Pfeil an der Pegelanzeige bewegst. Das Abmixen erledigst du am besten, während du das Spiel im Editor testest, da du dann live hören kannst, wie sich dein aktueller Mix anhört. Der *AudioMixer* gehört zu den wenigen Elementen in Unity, dessen Änderungen auch im *Play-Mode* gespeichert werden.

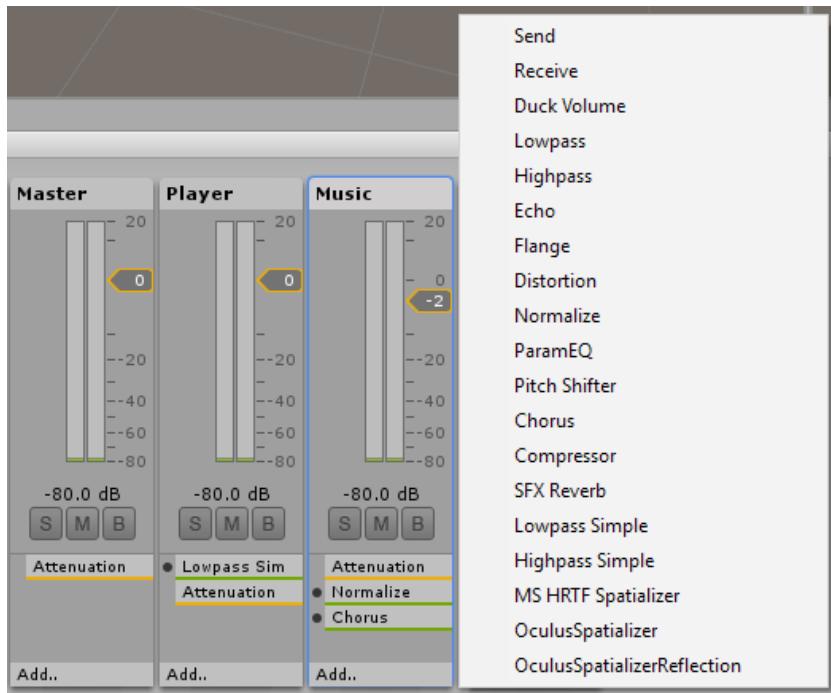


Bild 6.38 Für jede Gruppe kannst du getrennt Laustärke und diverse Effekte einstellen.

Neben der Lautstärke kannst du auch diverse unterschiedliche Effekte zu den einzelnen Gruppen hinzufügen. Hier ist, wie auch bei der Lautstärke, die Hierarchie zu beachten: Fügst du einen Effekt zu einer Untergruppe hinzu, wird dieser nur auf ihr zugeordneten *AudioSources* angewendet. Fügst du einen Effekt zu einer höher geordneten Gruppe (z.B. Master) hinzu, wird der Effekt auch für alle Audioquellen in den jeweiligen Untergruppen angewendet.

Einen Effekt kannst du über die **Add...**-Schaltfläche, unten in jeder Gruppe, hinzufügen. Wenn du mehrere Effekte zu einer Gruppe hinzufügst, solltest du, je nach Effekt, auf die Reihenfolge der Effekte achten. Du kannst die Effekte mittels *Drag & Drop* in der Effekt-Übersicht sortieren. Über die kleinen Kreise neben den Effektnamen kannst du einzelne Effekte vorübergehend deaktivieren und so unterschiedliche Konfigurationen ausprobieren. Wenn du einen Effekt anklickst, stehen dir im *Inspector* Anpassungsmöglichkeiten für den jeweiligen Effekt zur Verfügung.

6.3.5.2 Solo, Mute und Bypass

In jeder Gruppe stehen dir die drei Schaltflächen **S**, **M** und **B** zur Verfügung. Die drei Buchstaben stehen für *Solo*, *Mute* und *Bypass* und ihre Schaltflächen erlauben dir, für Testzwecke das Verhalten der jeweiligen Gruppe zu verändern.

- **Solo:** Die „Solo“-Funktion stellt alle Gruppen, bei denen nicht auch Solo aktiviert ist, stumm. Das erlaubt es dir, für den perfekten Mix nur einzelne Gruppen zu aktivieren, sodass du nicht von den Sounds anderer Gruppen abgelenkt wirst. Du kannst *Solo* bei

mehreren Gruppen aktivieren, es werden dann alle Gruppen stumm geschaltet, bei denen es nicht aktiv ist.

- **Mute:** Die „Mute“-Funktion stellt die ausgewählte Gruppe und dadurch auch alle ihre Kinder stumm. Mutest du die Master-Gruppe, würdest du zum Beispiel gar nichts mehr hören, auch wenn die Kinder nicht gemutet sind.
- **Bypass:** Die „Bypass“-Funktion erlaubt es dir, mit einem Klick alle Effekte an der jeweiligen Gruppe zu deaktivieren. Anders als *Solo* und *Mute* werden nicht automatisch auch alle Effekte von Untergruppen deaktiviert, sondern wirklich nur die Effekte der Gruppe, bei der du die „Bypass“-Funktion aktiviert hast.

6.3.5.3 Snapshots

Snapshots machen aus dem praktischen *AudioMixer* ein noch mächtigeres Werkzeug. Der Grundgedanke dahinter ist, dass du mehrere Konfigurationen deines *AudioMixers* unter verschiedenen Namen abspeichern kannst. Danach kannst du mit einem Script zwischen den einzelnen gespeicherten *Snapshots* hin und her wechseln. Ein naheliegendes Beispiel wäre, dass sich alle Umgebungsgeräusche automatisch leiser schalten und die Hintergrundmusik ein wenig lauter wird, wenn der Spieler das Pausenmenü des Spiels öffnet. Anschließend sollen die Effekte wieder zurückgesetzt werden, wenn er das Menü verlässt. Ein anderes Beispiel wäre das Simulieren von Taubheit nach einer lauten Explosion im Spiel. Hier würde man dann alle Sounds sehr leise machen und ggf. ein wenig filtern, damit sie sich weit weg anhören. Nach ein paar Sekunden wechselt man dann wieder zu der „normalen“ Konfiguration. Die Einsatzmöglichkeiten sind sehr groß und *Snapshots* sind ein wichtiges Element für gutes Audiodesign in einem Spiel.

Wenn du mehrere *Snapshots* (über das + bei der *Snapshot*-Liste) angelegt hast, kannst du ein Script mit `public`-Variablen vom Typ *AudioMixerSnapshot* erstellen und dann zwischen den einzelnen *Snapshots* hin und her wechseln, indem du die Methode `TransitionTo(...)` auf dem Ziel-Snapshot aufrufst. Die Methode `TransitionTo(...)` akzeptiert einen Parameter vom Typ `float`. Mit diesem kannst du bestimmen, wie lange die Überblendung zwischen dem aktuellem und dem neuen Snapshot dauern soll.

Listing 6.3 Beispiel für das Aktivieren von Snapshots via Code

```
public AudioMixerSnapshot pauseMenuSnapshot;
public AudioMixerSnapshot gameplaySnapshot;
private bool currentlyPaused;
void Update(){
    if(Input.GetKeyDown(KeyCode.Escape)){
        if(currentlyPaused){
            currentlyPaused = false;
            gameplaySnapshot.TransitionTo(0.5f);
        }else{
            currentlyPaused = true;
            pauseMenuSnapshot.TransitionTo(1);
        }
    }
}
```

Listing 6.3 zeigt dir ein Beispiel für das Verwenden von Snapshots in einem Script. Wenn du das Script zu einem beliebigen *GameObject* hinzufügst, musst du im *Inspector* noch die *AudioMixerS snapshots* zuweisen, welche du vorher im *AudioMixer* anlegen musst.

6.3.5.4 Views

Im *AudioMixer*-Fenster kannst du außerdem noch sogenannte „Views“ anlegen. Anders als man denken könnte, haben sie nichts mit unterschiedlichen Perspektiven im Spiel zu tun (das würde man über *Snapshots* regeln). Im *AudioMixer*-Fenster kannst du Gruppen ausblenden, indem du auf das Augen-Icon in der *Groups*-Liste klickst. Blendest du eine Gruppe aus, werden Gruppen, die ihr untergeordnet sind, nicht automatisch auch ausgeblendet. Dies scheint zunächst nicht intuitiv, erlaubt es aber, die Ansicht noch genauer anzupassen.

Wenn du einen sehr komplexen *AudioMixer* erstellt hast, hilft diese Ein- und Ausblend-Funktion, die Übersicht zu behalten. Damit du in diesem Fall nicht immer wieder viele Gruppen ein- und ausschalten musst, kannst du den jeweils aktuellen Zustand in einer *View* speichern und so schnell zwischen unterschiedlichen Ansichten hin und her wechseln.

Bei einem komplexen *AudioMixer* könnte man beispielsweise eine *View* nur für die Gruppen anlegen, die sich mit Umgebungsgeräuschen (Vögel, Bäume, Regen) beschäftigen, und dann eine andere für diverse Audio-Gruppen des Spielers (Stimme, Schritte, Items, Waffe) und so weiter.

6.3.5.5 Mehrere AudioMixer

Für sehr komplexe Projekte, die selbst mit der Verwendung von *Views* unübersichtlich werden, können außerdem auch mehrere *AudioMixer* angelegt werden, welche sich jeweils um bestimmte Bereiche der Soundkulisse kümmern. Jeder Mixer hat dann seine eigenen Gruppen, Snapshots und Views, sodass sie zwar gleichzeitig, aber vollkommen unabhängig voneinander agieren können.

Zwischen deinen einzelnen *AudioMixern* deines Projektes kannst du in der „Mixers“-Liste des *AudioMixer*-Fensters hin und her schalten.

■ 6.4 Tastatur und Gamepad-Eingaben lesen

Für ein richtiges Videospiel reicht das bloße Umsehen mit der VR-Brille meistens nicht aus – auch wenn es tatsächlich manche VR-Erfahrungen gibt, die sich zwecks Einfachheit darauf beschränken. Deswegen bietet dir Unity eine einfache Möglichkeit an, Eingaben von Maus, Tastatur und Gamepads zu lesen.

In Unity hast du zwei Möglichkeiten, eine bestimmte Taste abzufragen: Entweder du verwendest in deinem Script sogenannte *KeyCodes*, welche jeweils für eine bestimmte Taste auf der Tastatur oder dem Gamepad stehen, oder du verwendest Unitys *Input-Manager*. Während die Tasten bei Ersterem fest in dein Script eingebunden sind (*hardcoded*), gibt es bei letzterer Variante die Möglichkeit, dass die Tasten, ohne Änderungen an den Scripten, nachträglich geändert werden können. Für Nicht-VR-Spiele oder VR-Spiele, die man mit Tastatur

& Maus spielt, solltest du immer den *Input-Manager* verwenden, damit der Spieler die Steuerung nach seinen Vorlieben selber anpassen kann.

Leider unterstützt der *Input-Manager* nicht die individuellen Controller der VR-Brillen, weswegen wir uns nur kurz ansehen werden, wie man den Input-Manager konfiguriert.

Danach schauen wir uns an, wie du in Unity Eingaben über die Input-Klasse lesen kannst. Die jeweiligen VR-SDKs haben zwar ihre eigenen Klassen zum Lesen der Eingaben, sie nutzen jedoch dasselbe Prinzip. Das Wissen über Unitys Input-Klasse stellt deswegen eine Grundlage für das Verständnis der SDKs dar.



Die Virtual-Reality-SDKs haben teilweise ihre eigenen Input-Klassen

Die Input-Klasse von Unity musst du für Virtual-Reality-Spiele in der Regel nur in Ausnahmefällen verwenden, da die VR-SDKs ihre eigenen Klassen für das Lesen der Eingaben von ihren jeweiligen Controllern haben. Diese individuellen Klassen sind meist ähnlich aufgebaut wie die Unity-Klasse, weshalb du diesen Abschnitt trotzdem lesen solltest. Mehr Details zu den jeweiligen Input-Klassen findest du in den Kapiteln zu den Virtual Reality SDKs.

6.4.1 Der Input-Manager

Den *Input-Manager* findest du unter **EDIT/PROJECT SETTINGS/INPUT**. In diesem Fenster kannst du beliebig viele virtuelle Tasten und Achsen definieren, welche du später in deinem Script über ihren *Namen* abfragen kannst. Dabei können mehrere Tasten oder Achsen auch denselben Namen haben, sodass mehrere Tasten dieselbe Funktion auslösen.

Wenn mehrere virtuelle Tasten den gleichen Namen im *Input-Manager* haben, werden alle angegebenen Tasten geprüft und die virtuelle Taste gilt als gedrückt, wenn mindestens eine der angegebenen Tasten gedrückt wurde. Du musst in deinem Script beispielsweise nur den Zustand der virtuellen Taste „Fire1“ abfragen, und der *Input Manager* überprüft automatisch, ob irgendeine der definierten „Fire1“-Tasten gedrückt wird.

6.4.2 Virtuelle Tasten und Achsen

Der Unterschied zwischen einer *Taste* und einer *Achse* ist, dass Tasten nur zwei Zustände kennen: gedrückt und nicht gedrückt. Achsen hingegen können einen beliebigen Wert zwischen -1 und 1 annehmen, wobei 0 die Ausgangsstellung ist. Wenn du es dir mit einem Joystick vorstellst, entspricht ein Achsenwert von 0.5 zum Beispiel, dass der Joystick zur Hälfte nach rechts eingeschlagen ist. -1 hingegen würde dann bedeuten, der Joystick ist vollkommen nach links eingeschlagen. Im Gegensatz zu Tasten sind hier beliebig viele Zwischenzustände möglich.

In Unitys *Input-Manager* sind allerdings auch die „virtuellen Tasten“ als *virtuelle Achsen* hinterlegt. Drückst du eine Taste auf der Tastatur, schlägt die virtuelle Achse komplett aus und kehrt zu 0 zurück, wenn du sie wieder loslässt.

Diese „Alles ist eine Achse“-Variante hat den Vorteil, dass man, ohne Anpassungen im Script, dieselbe virtuelle Achse sowohl über eine Taste auf der Tastatur als auch über den Thumstick eines Gamepads steuern kann. Wenn du beispielsweise die Steuerung der Spielfigur mit virtuellen Achsen programmierst, musst du in deinem Script keine Unterscheidung für Tastatur, Joystick oder Gamepad einfügen, da beide Eingabemethoden dieselbe virtuelle Achse beeinflussen.

Im *Input-Manager* findest du eine solche Konfiguration für die Achsen *Horizontal* und *Vertical*, welche in Nicht-VR-Spielen typischerweise zum Steuern der Spielfigur verwendet werden.

6.4.3 Konfiguration einer Achse

Da der *Input-Manager* ein wichtiges Element in Unity ist und er in Zukunft wahrscheinlich auch einen Teil der Eingaben von VR-Controllern erfassen können wird, folgt hier nochmals eine Übersicht der einzelnen Einstellungsmöglichkeiten für jede im Input-Manager definierte Achse.

Zum Anlegen einer neuen virtuellen Achse (oder Taste) duplizierst du einfach eine der existierenden Achsen, die im Idealfall bereits ähnlich funktioniert wie die, die du anlegen möchtest. Klicke dazu mit der rechten Maustaste auf das ausklappbare Array-Element und wähle **Duplicate Array Element**.

Anschließend änderst du den Namen und passt die Konfiguration deines neuen Elementes deinen Wünschen nach an.

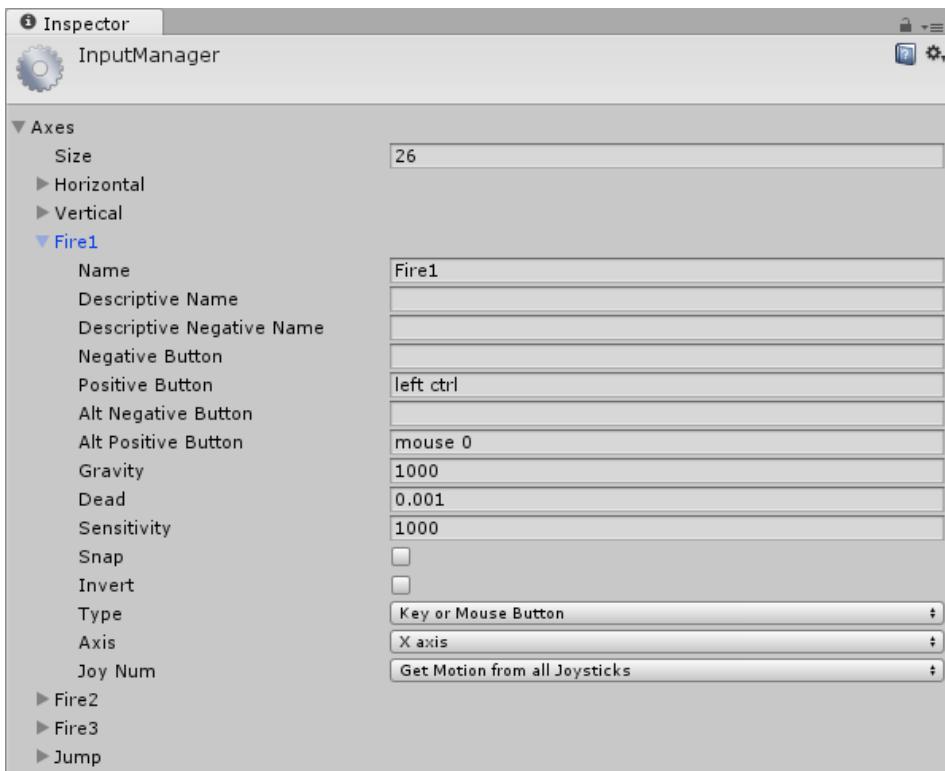


Bild 6.39 Der Input-Manager und die virtuelle Achse „Fire1“, welche hier für die linke Maustaste und die linke Strg-Taste konfiguriert wurde

- **Name:** Dieses Feld legt den internen Namen der virtuellen Taste fest. Über diesen Namen wirst du die Achse in deinem Code ansprechen können, deswegen sollte er möglichst eindeutig und gut lesbar sein. In den meisten Fällen macht es Sinn, die Achsen nach ihrer Funktion zu benennen, z.B. „Primary Fire“, „Menu“, „Jump“.
- **Descriptive Name:** Bei PC-Spielen kann der Spieler die Tastenbelegung anpassen, ohne dass du dafür etwas programmieren musst. Im Bildschirm für die Anpassung wird dieser „ausdrucksvolle Name“ anstelle des internen Namens angezeigt. Gibst du keinen *Descriptive Name* an, wird der interne Name angezeigt. Hier kannst du einen leicht verständlichen Namen angeben, wenn der interne Name ggf. zu kryptisch sein sollte.
- **Descriptive Negative Name:** Identisch zu *Descriptive Name*, nur für die negative Richtung der Achse, z.B. „Rückwärts laufen“
- **Negative Button:** Diese Taste lässt die Achse in die negative Richtung ausschlagen.
- **Positive Button:** Diese Taste lässt die Achse in die positive Richtung ausschlagen.
- **Alt Negative Button:** Alternative Taste für einen Ausschlag in die negative Richtung
- **Alt Positive Button:** Alternative Taste für einen Ausschlag in die positive Richtung
- **Gravity:** Geschwindigkeit in Einheiten pro Sekunde, mit der die Achse wieder auf 0 zurückfällt, wenn dazugehörige Tasten losgelassen wurden

- **Dead:** Die sogenannte „Dead-Zone“ ignoriert minimale Veränderungen des Joysticks/Thumbsticks. Je höher der Wert, umso weiter muss der Stick sich aus seinem Zentrum bewegen, damit die Achse anfängt, ihren Wert zu ändern.
- **Sensitivity:** Geschwindigkeit in Einheiten pro Sekunde, mit der die Achse ihren Zielwert erreicht, wenn eine Taste gedrückt wurde
- **Snap:** Wenn aktiv, fällt der Achsenwert sofort auf 0, wenn die Taste für die jeweils andere Richtung gedrückt wird, nachdem die Taste für eine Richtung losgelassen wurde
- **Invert:** Wenn aktiv, werden alle positiven und negativen Werte vertauscht
- **Type:** Bestimmt, ob diese virtuelle Achse durch eine Tastatur-, Gamepad- oder Maustaste, durch Mausbewegung oder einer Joystickachse (Joystickachsen beinhalten auch Thumbsticks etc.) verändert wird
- **Axis:** Wenn der *Type* eine Joystickachse ist, kann hier bestimmt werden, welche Achse es ist
- **Joy Num:** Wenn mehrere Gamepads oder Joysticks mit dem Computer des Spielers verbunden sind, kannst du hier bestimmen, welcher verwendet werden soll. (Bei den meisten Singleplayer-Spielen wirst du hier *Get Motion from all Joysticks* auswählen wollen.)



Weiterführende Links

Mehr Details darüber, wie du die Buttons angeben musst, erfährst du in der offiziellen Doku ganz unten:

<https://docs.unity3d.com/Manual/ConventionalGameInput.html>

Welche Taste/Achse deines Gamepads welche Nummer hat, erfährst du am Beispiel des Xbox360 Controllers für Windows im Unity-Wiki:

<http://wiki.unity3d.com/index.php?title=Xbox360Controller>

6.4.4 Gamepad- und Tastatureingaben erfassen

Jetzt kommen wir dazu, wie du Tasteneingaben tatsächlich lesen kannst. Im Kern funktioniert das über mehrere statische Methoden der *Input*-Klasse, wie die *GetKey*-Methoden, die ich dir nachfolgend vorstelle. Die meisten VR-SDKs besitzen ein Äquivalent für die drei Methoden, sodass es auch für die VR-Entwicklung wichtig ist, diese Methoden zu kennen.

- **GetKey(key : KeyCode) : bool**

Diese Abfrage gibt zurück, ob die angegebene Taste zu diesem Zeitpunkt gerade gedrückt wird. Diese Abfrage ist also so lange true, wie die Taste gedrückt gehalten wird.

- **GetKeyDown(key: KeyCode) : bool**

Diese Abfrage funktioniert wie *GetKey*, allerdings ergibt sie nur dann ein positives Ergebnis, wenn die Taste *in diesem Frame* gedrückt wurde. Im Gegensatz zu der normalen *GetKey*-Methode erhältst du also nur *einmalig* ein positives Ergebnis, wenn die Taste gedrückt gehalten wird.

■ **GetKeyUp(key: KeyCode) : bool**

Diese Abfrage funktioniert wie GetKeyDown, allerdings ergibt sie nur dann, *einmalig*, ein positives Ergebnis, wenn die Taste *in diesem Frame* losgelassen wurde.

Am wichtigsten in diesem Abschnitt ist, dass du die Unterschiede zwischen der „normalen“, der „Down“- und der „Up“-Variante verstehst; denn dieses System wirst du auch in den Input-Klassen des VR-SDK wiederfinden.

Tasten- und Achsenabfragen schreibst du typischerweise in die Update-Methode deines Components, da sie ja die ganze Zeit abgefragt werden sollen. Listing 6.4 zeigt dir, wie eine solche Update-Methode aussehen könnte.

Listing 6.4 Die Verwendung von GetKey, GetKeyDown und GetKeyUp

```
void Update(){
    if(Input.GetKeyDown(KeyCode.Space)){
        Debug.Log("##### Leertaste wurde in diesem Moment GEDRÜCKT");
    }
    if(Input.GetKey(KeyCode.Space)){
        Debug.Log("Leertaste wird gedrückt GEHALTEN");
    }
    if(Input.GetKeyUp(KeyCode.Space)){
        Debug.Log("##### Leertaste wurde in diesem Moment LOSGELASSEN");
    }
}
```

6.4.4.1 Tasten mittels Input-Manager auslesen

Die drei Methoden GetButton, GetButtonDown und GetButtonUp funktionieren genauso wie ihre GetKey-Pendants, allerdings übergibst du ihnen als Parameter den Namen einer virtuellen Achse des *Input-Managers*. Die Abfragen geben immer dann `true` zurück, wenn einer der Buttons, die in der Achse definiert wurden (*Positive, Negative, Alt Positive oder Alt Negative*), gedrückt wurde. Wenn du unterscheiden möchtest, ob ein positiver oder negativer Button gedrückt wurde, musst du zusätzlich den Achsenwert mit GetAxis auslesen. Listing 6.5 demonstriert die Verwendung der Methode.

Listing 6.5 Beispiel für GetButton

```
void Update()
{
    if (Input.GetButton("Fire1")) {
        Debug.Log("[Linke Maustaste] oder [Linkes STRG] ist gedrückt");
    }
    if (Input.GetButtonDown("Horizontal")) {
        if (Input.GetAxis("Horizontal") > 0) {
            Debug.Log("[Pfeil rechts] oder [D] wurde gedrückt");
        } else {
            Debug.Log("[Pfeil links] oder [A] wurde gedrückt");
        }
    }
}
```

6.4.4.2 Achsenwerte auslesen

Das Auslesen von Achsenwerten erfolgt über die Methode `GetAxis` der `Input`-Klasse. Als Parameter übergibst du ihr den Namen der gewünschten virtuellen Achse aus dem `InputManager` und zurück erhältst du eine Zahl zwischen -1 und 1. Wobei 0 die Ausgangsposition der Achse ist. Listing 6.6 demonstriert das Auslesen einer virtuellen Achse mit `GetAxis`.

Listing 6.6 Verwendung von GetAxis zur Berechnung der Laufgeschwindigkeit

```
float maxWalkSpeed = 5; // Einheiten (Meter) pro Sekunde
float currentWalkSpeed = 0;
void Update() {
    float axisValue = Input.GetAxis("Vertical");
    currentWalkSpeed = maxWalkSpeed * axisValue * Time.deltaTime;
    Debug.Log("Achsenwert: " + axisValue);
    Debug.Log("Laufgeschwindigkeit: " + currentWalkSpeed + " Einheiten pro Sek");
}
```

■ 6.5 Physik

In Virtual Reality ist plausibles physikalisches Verhalten sehr wichtig. Viel mehr als in gewöhnlichen Spielen verbringen Spieler Zeit damit, Objekte durch die Gegend zu werfen und mit verschiedenen Spielereien die Grenzen des Spiels auszutesten. Damit das Verhalten der Objekte plausibel wirkt, müssen sie zum Beispiel durch Kollisionen, Schwerkraft und andere Kräfte korrekt beeinflusst werden. Diese komplizierten Berechnungen musst du nicht selber programmieren, Unity stellt dir dafür eine sogenannte *Physics Engine* zur Verfügung.

Diese Engine erlaubt es dir, durch die Konfiguration von ein paar Parametern deinen Game-Objects ein möglichst realistisches physikalisches Verhalten zu verleihen. Zusätzlich hast du die Möglichkeit, auf die Physik einzelner Objekte mittels Scripten Einfluss zu nehmen. Du kannst sie beschleunigen, abbremsen und noch viel mehr.

Die Unity Engine besitzt zwei unabhängige *Physics Engines*, eine für 3D-Berechnungen und eine für 2D-Berechnungen. Wir werden uns in diesem Buch ausschließlich der 3D-Variante widmen, da die 2D-Variante nur für 2D-Spiele interessant ist.

6.5.1 Collider

Collider werden von der *Physics Engine* verwendet, um Kollisionen zu berechnen. Der *Collider* selbst ist unsichtbar und muss nicht exakt die Form des 3D-Models haben. In der Praxis ist eine grobe Annäherung in der Regel sogar effizienter und im Spiel nicht von einem perfekt akkurate *Collider* unterscheidbar.

Die einfachsten und performantesten *Collider* sind die sogenannten *primitiven Collider*. Zu diesen zählen folgende drei *Collider*, welche jeweils als ein eigenes Component realisiert sind.

- **Box Collider:** Besitzt eine Würfelform
- **Sphere Collider:** Besitzt eine Kugelform
- **Capsule Collider:** In Kapsel-(Tabletten-)Form

Fügst du zu einem *GameObject*, das bereits einen *MeshRenderer* besitzt, einen primitiven *Collider* hinzu, wird dieser automatisch so positioniert und skaliert, dass er zu dem *Mesh* des *MeshRenderers* passt. Du kannst zu einem einzelnen *GameObject* mehrere *Collider* hinzufügen und so einen kombinierten *Collider* für das *GameObject* erstellen. Durch geschickte Anpassung der Position können solche kombinierten *Collider* die Form eines Objektes bereits sehr gut nachformen, ohne dass der *Collider* zu detailliert wird. Zusätzlich können auch noch weitere *Collider* zu den Kindern deines *GameObjects* hinzugefügt werden. Dies ist zum Beispiel praktisch, damit die Rotation eines *Box Colliders* unabhängig von der Rotation des Haupt-*GameObjects* geändert werden kann. Wie in Bild 6.40 zu sehen, werden die *Collider* durch grüne Linien an dem *GameObject* dargestellt, wenn das *GameObject* in der *Hierarchy* aktiviert ist.



Bild 6.40 Links ein einfacher Box Collider, rechts ein kombinierter Collider, bestehend aus mehreren Box Collidern

Es gibt jedoch auch Fälle, wo diese kombinierten *Collider* zu ungenau sind, um die Form des Objektes gut nachzubilden. Für diese Fälle gibt es das *Mesh Collider*-Component.

Diese komplexen *Collider* entsprechen exakt der Form des jeweiligen *Meshes* (des 3D-Modells). Diese *Collider* benötigen jedoch *deutlich* mehr Prozessorleistung als die primitiven Varianten. *MeshCollider* haben deshalb auch ein paar Einschränkungen, die du beachten solltest: Objekte, die einen *MeshCollider* verwenden, können zum Beispiel nur passiv in die Physik-Berechnung aufgenommen werden. Das bedeutet, andere Objekte, die durch die *Physics Engine* bewegt werden, können zwar mit ihnen kollidieren, sie selbst können aber nicht

durch die *Physics Engine* bewegt werden. Ein GameObject mit einem Standard-*MeshCollider* kann also nur für Umgebungskollisionen verwendet werden.

Es gibt jedoch eine Ausnahme: Ein GameObject, das einen *MeshCollider* verwendet, kann aktiv durch die *Physics Engine* bewegt werden, wenn du aus dem *Mesh* eine konvexe Hülle (*Convex Hull*) generieren lässt. Das machst du, indem du an dem *MeshCollider* die Checkbox **CONVEX** aktivierst.

Die konvexe Hülle ist ein *Collider*, der die Form des ursprünglichen *Meshes* besitzt, allerdings ohne sämtliche Aussparungen (Löcher, Einschnitte etc.). Ob das für dein Modell eine mögliche Lösung ist, hängt von dem Modell ab. Je nach Modell ist das Ergebnis ungenauer als ein kombinierter *Collider*, den du selber zusammenbaust. Bild 6.41 zeigt den Standard-*MeshCollider* und die konvexe Hülle eines Schubwagens miteinander. Hier wäre der kombinierte Collider aus Bild 6.40 zum Beispiel genauer als die konvexe Hülle.

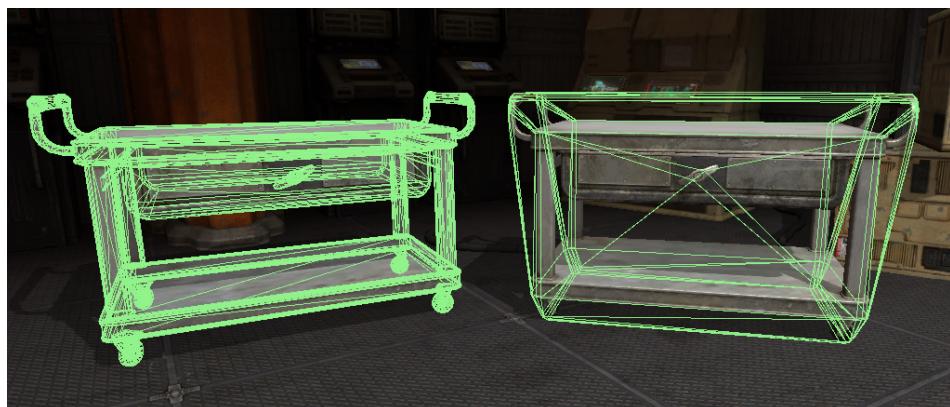


Bild 6.41 Links der sehr komplexe Standard-*MeshCollider*, rechts ein *MeshCollider* mit aktivierter „Convex“-Funktion

6.5.1.1 Primitive Collider Components im Inspector

Die Components der unterschiedlichen *Collider* sind alle gleich aufgebaut. In Bild 6.42 siehst du beispielhaft, wie ein *Box Collider* im *Inspector* aussieht.

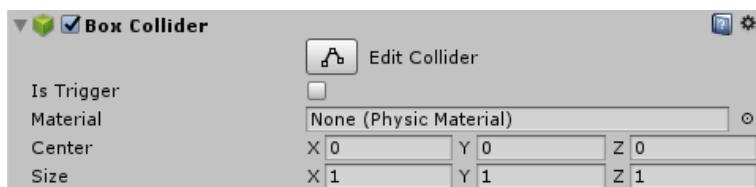


Bild 6.42 Ein *Box Collider* Component im Editor

Ganz oben findest du immer die **EDIT COLLIDER**-Funktion, die es dir erlaubt, die Größe und Position des *Colliders* in der *Scene View* zu bearbeiten.

Die Checkbox *Is Trigger* bestimmt, ob dieser *Collider* nicht für Kollisionen, sondern zur Erkennung von Objekten in der Nähe verwendet werden soll. Ist die Checkbox aktiv, kann

über die Methoden `OnTriggerStay`, `OnTriggerEnter` und `OnTriggerExit` erkannt werden, ob ein `GameObject` sich in der Nähe des Trigger-Colliders befindet, ihn betritt oder verlässt. Mehr dazu folgt in Kapitel 6.5.5.

Über das Feld `Material` gibst du kein normales `Material` an, sondern ein `Physic Material`, welches nicht das Aussehen des Modells beschreibt, sondern die physikalischen Eigenschaften der Oberfläche (glatt, rau, weich, hart).

Die restlichen Optionen sind abhängig von dem Collider-Typen und bestimmen die Größe und Position des Colliders.

6.5.2 Rigidbody

In diesem Abschnitt widmen wir uns einem der Kerne der Physics-Engine, den `Rigidbody`-Components.

`Rigidbody`-Components sorgen dafür, dass ein `GameObject` in die Physik-Berechnung der `Physics Engine` aufgenommen wird. Das bedeutet, `GameObjects` mit einem `Rigidbody` werden von physikalischen Kräften wie der Schwerkraft beeinflusst und fallen dementsprechend, wie in Bild 6.43 zu sehen, Richtung Boden. Damit das `Rigidbody`-Component etwas bewirkt, muss das `GameObject` zusätzlich auch über einen primitiven `Collider` oder einen konvexen `MeshCollider` verfügen.

`GameObjects`, die über ein `Rigidbody`-Component verfügen, werden häufig als `Rigidbody Objects` bezeichnet.

Wie jedes Component, kannst du das `Rigidbody`-Component über das `Add Component`-Menü hinzufügen: **ADD COMPONENT/PHYSICS/RIGIDBODY**.



Bild 6.43 Fügst du zu einem `GameObject` ein `Rigidbody` hinzu, wird es unter anderem von der Schwerkraft beeinflusst.

Im *Inspector* stehen dir verschiedene Konfigurationsmöglichkeiten zur Verfügung, wie du in Bild 6.44 sehen kannst.

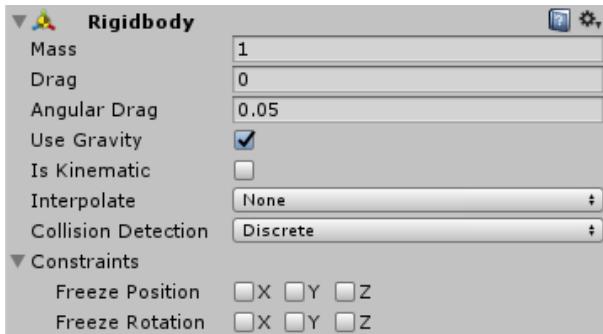


Bild 6.44 Der Inspector-Bereich des Rigidbody-Components

- **Mass:** Das Gewicht dieses GameObjects.
- **Drag:** Reibungs- bzw. Luftwiderstand von diesem GameObject, wenn es sich vorwärtsbewegt. 0 bedeutet kein Widerstand, sodass das Objekt, ohne äußere Einflüsse, niemals stehen bleiben würde, wenn es einmal in Bewegung gebracht wird.
- **Angular Drag:** Reibungs- bzw. Luftwiderstand von diesem GameObjects für Rotationen. Dieser Wert gibt an, wie schnell dieses Objekt, ohne äußere Einflüsse, aufhört zu rotieren.
- **Use Gravity:** Gibt an, ob dieses Objekt von der Schwerkraft beeinflusst wird.
- **Is Kinematic:** Diese Option deaktiviert, dass dieses Objekt durch die Physics Engine bewegt wird.

Eine solche Funktion wird zum Beispiel benötigt, wenn das *GameObject* mit einem anderen *GameObject* *physikalisch* verbunden werden soll, es aber nicht selbst durch die *Physics Engine* bewegt werden darf. Dies wird in Kapitel 6.5.4 über *Physics Joints* noch im Beispiel erklärt.

- **Interpolate:** Durch Interpolation kann das Bewegungsverhalten eines durch die Physik-Engine bewegten Objekts verschönert werden, falls es sich zur Laufzeit ruckartig bewegen sollte.
 - **None:** Keine Interpolation.
 - **Interpolate:** Die Bewegung wird auf Basis der Position im letzten Frame verfeinert.
 - **Extrapolate:** Die Bewegung wird auf Basis der zu erwartenden Position im nächsten Frame verfeinert.
- **Collision Detection:** Wenn sich ein Objekt sehr schnell bewegt, kann es sein, dass es sich durch ein anderes Objekt mit *Collider* hindurchbewegt, obwohl dies nicht passieren sollte. Das hängt mit verschiedenen Optimierungen während der Kollisionserkennung zusammen. Solltest du dieses Problem bemerken, kannst du die Kollisionserkennung mit diesem Parameter anpassen.

Discrete ist der Standardwert. Diesen solltest du aus Performance-Gründen auch möglichst immer verwenden. Bei Problemen kannst du *Continuous Dynamic* auswählen. Sollte dies dein Problem nicht beheben, wähle die Option *Continuous*. Diese Option kostet allerdings die meiste Performance.

- **Constrains:** Mit dieser Option verhinderst du, dass sich der *Rigidbody* auf der globalen x-, y-, z-Achse bewegen bzw. auf der lokalen x-, y-, z-Achse drehen kann.

6.5.2.1 Schwerkraft global verändern

In dem *Rigidbody*-Component selbst kannst du nur bestimmen, ob ein einzelnes *GameObject* von der Schwerkraft beeinflusst werden soll oder nicht. Du kannst jedoch auch an einer zentralen Stelle die Schwerkraft für alle *Rigidbody Objects* in der aktuellen *Scene* anpassen.

Die Option findest du unter **EDIT/PROJECT SETTINGS/PHYSICS/GRAVITY**.

Alternativ kannst du Schwerkraft auch mit in einem Script über die Variable `Physics.gravity` manipulieren.

6.5.3 Kräfte und Beschleunigung

Rigidbodys können nicht nur benutzt werden, damit Objekte physikalisch korrekt auf den Boden fallen oder einen Berg runterrollen. Du kannst Rigidbodys auch mittels eines Scripts beeinflussen, indem du Kräfte und Beschleunigungen über folgende Methoden simulierst.

Dafür verwendest du die beiden Methoden `AddForce` und `AddTorque`:

- **AddForce (force : Vector3 [, mode : ForceMode]) : void**

Mit dieser Methode kannst du Kräfte und Beschleunigungen auf ein *Rigidbody Object* ausüben. Der `force`-Parameter gibt die Richtung und die Stärke der Kraft an. Je höher die Werte, desto stärker also die Kraft. Der Parameter `mode` ist optional und bestimmt, welche Art von Kraft ausgeübt werden soll. Mit dem Standardwert `ForceMode .Force` wird beim Anwenden der Kraft die Masse des Objektes berücksichtigt, bei `ForceMode.Acceleration` wird die Masse hingegen ignoriert.

`ForceMode.Impulse` und `ForceMode.VelocityChange` ändern die Kraft des *GameObjects* sofort auf den angegebenen Wert und ignorieren dabei, dass sich das *GameObject* eventuell bereits in Bewegung befindet, und werden ungeschwächt angewendet.³ `Impulse` berücksichtigt dabei wie `Force` die Masse des *GameObjects* und `VelocityChange` berücksichtigt diese nicht.

- **AddTorque (torque : Vector3 [, mode : ForceMode]) : void**

Mit dieser Methode kannst du ein *Rigidbody Object* in Drehung versetzen. Über den Parameter `torque` bestimmst du, wie viel Drehmoment in welche Richtung angewendet werden soll. Je größer die Werte, desto stärker das Drehmoment in die jeweilige Richtung. Der optionale Parameter `mode` bestimmt wie schon bei `AddForce`, ob es sich um eine kontinuierliche Kraft handelt oder nicht und ob die Masse des betroffenen Objektes berücksichtigt werden soll.

In Listing 6.7 siehst du als einfaches Beispiel ein Script, welches du zu einem *GameObject* mit einem *Trigger Collider* hinzufügen kannst. Das Script erzeugt dann ein Anti-Schwerkraft-Feld, welches Objekte schweben lässt. Bild 6.45 zeigt das Script im Einsatz.

³ Bewegt sich ein Objekt in eine bestimmte Richtung, muss es zunächst abgebremst werden, damit es sich in die Gegenrichtung bewegen kann. Dieses Abbremsen wird hier ignoriert.

Listing 6.7 Dieses AntiGravity-Script erzeugt innerhalb eines Trigger Colliders, der sich am selben GameObject befinden muss, ein Anti-Schwerkraft-Feld, welches Objekte schweben lässt.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class AntiGravity : MonoBehaviour {
    public float hoverForce = 13;
    void OnTriggerEnter(Collider other)
    {
        other.GetComponent<Rigidbody>().AddForce(Vector3.up * hoverForce,
                                                     ForceMode.Acceleration);
        other.GetComponent<Rigidbody>().AddTorque(Vector3.right * 2);
    }
}
```

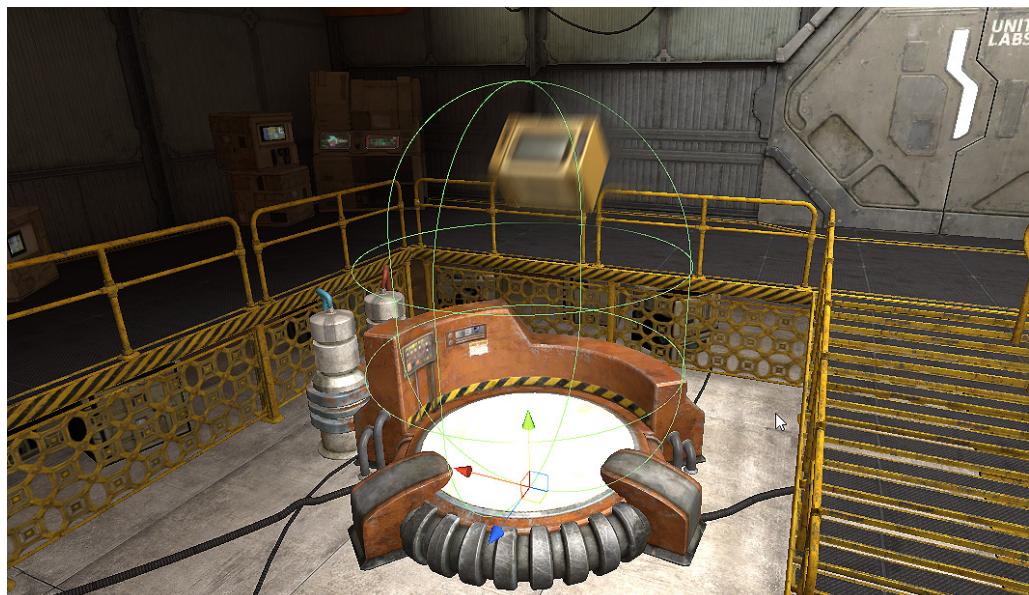


Bild 6.45 Ein Capsule Collider als Trigger lässt in diesem Beispiel Rigidbody Objects schweben.

6.5.4 Physics Joints – Verknüpfungen

Du kannst ein *Rigidbody Object* mit einem anderen *Rigidbody Object* oder zu einem festen Punkt in der Scene mit einem sogenannten *Joint* verbinden. In Unity stehen dir dafür verschiedene *Joint*-Arten zur Verfügung, die die betroffenen *Rigidbody Objects* jeweils auf verschiedene Arten miteinander verbinden und ihnen unterschiedlich viel Freiraum bieten.

Ein *Hinge Joint* erlaubt zum Beispiel, dass sich ein verbundenes Objekt noch weiterhin, wie eine Tür, drehen darf. Ein *Spring Joint* simulierte hingegen, dass ein Objekt quasi mit einer *Sprungfeder* an einem bestimmten Punkt oder einem anderen Objekt befestigt wurde.

Dabei ist es wichtig zu wissen, dass *Physics Joints* nicht sichtbar sind. Das bedeutet, wenn du zum Beispiel einen *Spring Joint* verwendest, wird nicht tatsächlich eine Sprungfeder zwischen den beiden Punkten dargestellt, die Objekte verhalten sich nur als wären sie durch eine Feder verbunden.

Joints sind komplex konfigurierbar, so kannst du zum Beispiel auch einstellen, dass sie kaputtgehen, wenn die auf sie wirkende Kraft zu groß wird; also zum Beispiel wenn der Spieler zu fest an einem bestimmten Objekt zieht. Die beiden zuvor verbundenen Objekte sind wieder vollkommen eigenständig.

Joints sind in Unity als *Components* realisiert, die du über den *Inspector* zu *GameObjects*, die ein *Rigidbody Component* besitzen, hinzufügen kannst: **ADD COMPONENT/PHYSICS/...** Wenn du die Suche des „Add Component“-Menüs verwendest, solltest du darauf achten, dass du nicht versehentlich einen der *2D Joints* verwendest. Diese sind ausschließlich für 2D-Spiele vorgesehen, welche die *2D-Physics Engine* von Unity verwenden.

Der in Bild 6.46 dargestellte *Fixed Joint* ist die einfachste Möglichkeit, zwei *Rigidbody Objects* physikalisch miteinander zu verbinden. Er verbindet die beiden Objekte nämlich starr miteinander, sodass sich die Objekte nicht mehr unabhängig voneinander bewegen können. In dem Bild sind die Eigenschaften sichtbar, die jede der Joint-Arten besitzt. Die komplexeren Varianten besitzen zudem noch individuelle Eigenschaften, die wir uns anschließend ansehen werden.

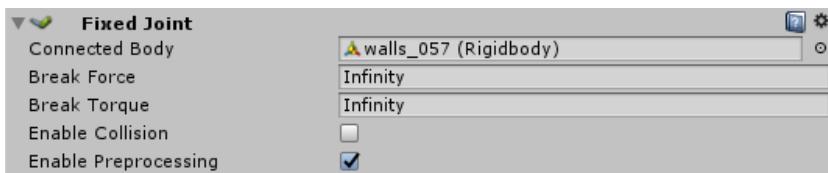


Bild 6.46 Der Fixed Joint enthält die minimale Konfiguration für Joints.

- **Connected Body:** Hier kannst du ein anderes *Rigidbody Object* angeben, mit dem der Besitzer dieses Components verbunden werden soll. Gibst du hier kein anderes Objekt an, wird der Besitzer dieses Components mit einem festen Punkt in der Scene verbunden (im Falle des *Fixed Joints* mit seiner aktuellen Position).
- **Break Force:** Hier kannst du bestimmen, wie viel Kraft auf den Joint wirken muss, damit er bricht und sich durch ihn verbundene Elemente wieder frei bewegen können. Der Standardwert ist *Infinity*, dieser sorgt dafür, dass selbst bei unendlich großer Kraft der Joint nicht kaputtgeht.
- **Break Torque:** Wie *Break Force* nur für Drehmoment, also drehende Kräfte.
- **Enable Collision:** Standardmäßig können zwei *Rigidbody Objects*, die miteinander verbunden wurden, nicht miteinander kollidieren. Auf diese Weise werden unter anderem sogenannte „Physic Glitches“⁴ verhindert. Über diese Checkbox kannst du die Kollision aktivieren, falls das für dein Vorhaben notwendig ist.

⁴ Bei einem Physic Glitch stoßen sich zum Beispiel zwei Objekte gegenseitig ab, sind aber gleichzeitig durch einen Joint miteinander verbunden. Das kann bewirken, dass sie unkontrolliert durch die Scene fliegen.

- **Enable Preprocessing:** Wenn diese Option aktiviert ist, ignoriert die Physics Engine Einschränkungen an dem Joint, wenn dies sehr starke Impulse auf die betroffenen Game-Objects zur Folge hätte. Diese Option kann die Genauigkeit des Joints beeinträchtigen, dafür liefert es in der Regel ein plausibleres Ergebnis.

Anchor und Connected Anchor

Die meisten Joints besitzen zudem noch die Möglichkeiten, einen *Anchor* und einen *Connected Anchor* anzugeben. Damit gibst du die Punkte an, an denen die Objekte durch den Joint miteinander verbunden werden sollen. Der *Anchor* bezieht sich dabei auf den Punkt an dem *GameObject*, das über das *Joint-Component* verfügt, und der *Connected Anchor* bestimmt den Punkt an dem angegebenen *Connected Body*. Ist kein anderes Objekt zugewiesen, gibt der *Connected Anchor* einen Punkt im Weltkoordinatensystem an. Der *Connected Anchor* wird relativ zu der Position des verbundenen *GameObjects* angegeben; nicht wie der *Anchor*, relativ zu der Position des *GameObjects*, der den *Joint* besitzt. Weil die Konfiguration des *Connected Anchor* deshalb schnell etwas umständlich wird, versucht Unity dir die Arbeit abzunehmen: Wenn du die Checkbox *Auto Configure Connected Anchor* anklickst, ermittelt Unity automatisch einen passenden Punkt, der zu dem von dir bestimmten *Anchor* passt.

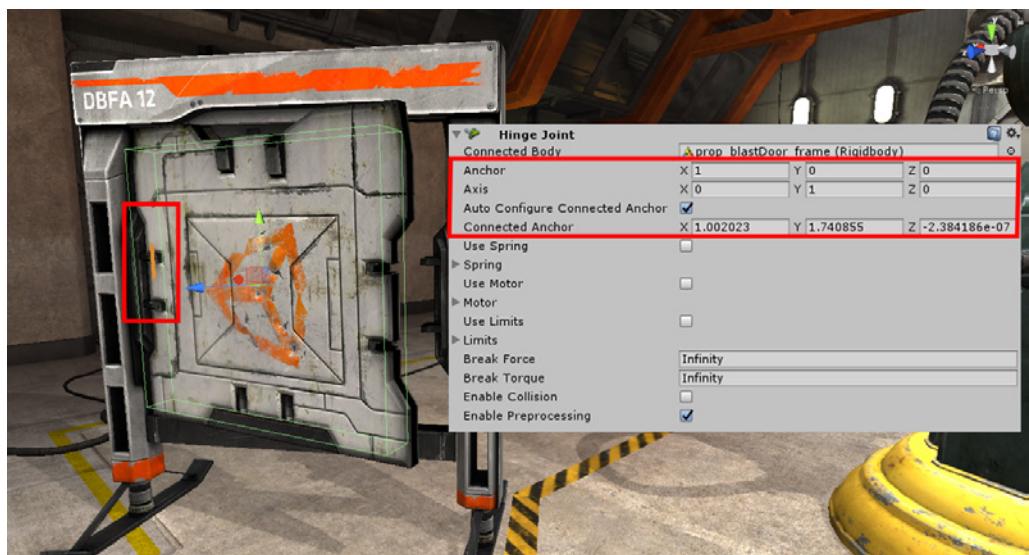


Bild 6.47 Rechts der Konfigurationsbereich für die beiden Anchor-Punkte, links markiert der orange Strich, wo sich die Punkte in der Scene View befinden.

Besonderheiten bei der Konfiguration der einzelnen Joint-Arten

Neben dem starren *Fixed Joint* stehen dir in Unity folgende *Joint*-Arten zur Verfügung:

- **Spring Joint:** Dieser *Joint* verbindet das betroffene *Rigidbody Object* mittels einer flexiblen Verbindung mit einem anderen *Rigidbody Object* oder einem festen Punkt in der Welt. Der Spring Joint kann entweder wie eine Art Sprungfeder oder Gummiseil konfiguriert werden. Über die Werte *Min Distance* und *Max Distance* kannst du einen Bereich definie-

ren, in dem keine Federkraft ausgeübt wird. Ist der Abstand kleiner als *Min Distance*, stößt die Feder das *Object* weg, ist der Abstand größer als *Max Distance*, zieht die Feder das *Object* näher heran. Sind beide Werte 0, versucht der *Spring Joint* die beiden *Anchor*-Punkte stets so nah wie möglich aneinander zu halten. Wie groß dieser Abstand tatsächlich ist, hängt von der Federkraft des Joints (*Spring*) und dem Gewicht (*Mass*) der angehängten Objekte ab. Der *Damper*-Wert bestimmt, wie stark die Federkraft gedämpft wird. Mit diesem Wert kannst du zum Beispiel verhindern, dass die verbundenen Objekte stark hin und her springen.

- **Hinge Joint:** Diesen *Joint* kannst du dir am besten wie eine Türangel vorstellen. Er befestigt das jeweilige *Rigidbody Object* mit einem bestimmten Punkt an einem *Anchor*-Punkt und verhindert, dass sich die beiden Punkte voneinander entfernen. Das *GameObject* kann sich jedoch weiterhin, wie eine Tür, um den *Anchor*-Punkt herumdrehen. Mit der *Axis*-Einstellung kannst du bestimmen, um welche Achse sich das *GameObject* drehen darf. Mit den Optionen der Kategorie *Limits* kannst du die Drehung auf der jeweiligen Achse weiter einschränken (z. B. maximaler Öffnungswinkel).
- **Character Joint:** Dieser komplexe *Joint* hat die Aufgabe, ein menschliches Gelenk zu imitieren, und stellt dir Optionen für das Konfigurieren entsprechender Bedingungen zur Verfügung.
- **Configurable Joint:** Dieser *Joint* ist eine vollkommen frei konfigurierbare Verbindung. Er stellt dir sehr viele Einstellungsmöglichkeiten zur Verfügung, was ihn sehr mächtig, aber auch sehr komplex macht. Für die meisten Szenarios wirst du diesen Joint nicht benötigen, da einfache Joints bereits alle typischen Einsatzmöglichkeiten abdecken.



Bild 6.48 Eine Abrissbirne, basierend auf mehreren *Rigidbody Objects* und *Joints*

Bild 6.48 zeigt beispielhaft eine Abrissbirne, die aus mehreren *Rigidbody Objects* und *Joints* gebaut wurde. Das oberste Rohr wurde mithilfe eines *Fixed Joints* fest in der *Scene* verankert. Die beiden nachfolgenden Rohre und die Kugel selbst wurden mithilfe von *Spring Joints* jeweils an dem vorhergehenden Teil befestigt. Die *Anchor*-Punkte wurden, wie in Bild

6.49 zu sehen, so gewählt, dass sie möglichst nah beieinanderliegen. Die *Spring Joints* wurden zudem so konfiguriert, dass sie stark genug sind, die einzelnen *Rigidbody Objects* eng beieinander zu halten.

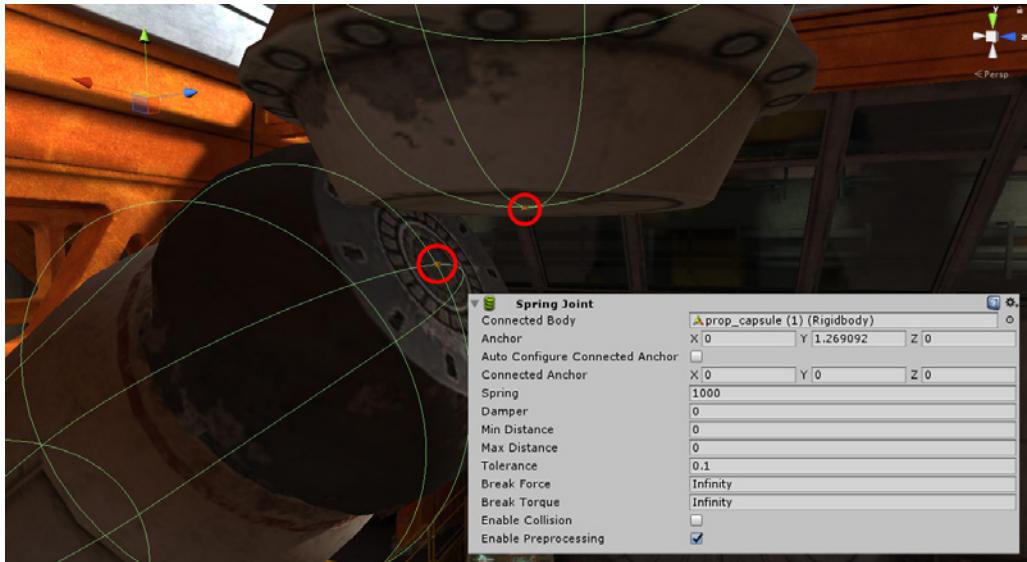


Bild 6.49 Die Konfiguration und Anchor-Punkte (rot markiert) eines der Elemente

6.5.5 Kollisionen und Trigger erkennen

Wie zuvor schon erwähnt, ist es in Unity möglich, Kollisionen zwischen zwei Objekten innerhalb eines *Scripts*, das mindestens an einem der beiden *GameObjects* vorhanden sein muss, zu erkennen. In diesem Zusammenhang ist es auch möglich, einen *Collider* so zu konfigurieren, dass er selbst nicht mit anderen Objekten kollidiert, sondern nur dazu dient, andere *Collidern* zu erkennen. Letzteres nennt man einen *Trigger-Collider*. Mit einem Trigger-Collider kannst du zum Beispiel erkennen, ob der Spieler einen bestimmten Bereich betritt, ihn verlässt oder sich darin aufhält. Damit die Erkennung funktioniert, muss mindestens eines der beiden *GameObjects*, jedoch zusätzlich ein *Rigidbody*-Component besitzen. Es ist aber egal, ob das Component als *Kinematic* konfiguriert ist oder nicht.

Sowohl das Erkennen von Kollisionen als auch die Verwendung von Trigger-Collidern werden wir uns jetzt einmal ansehen.

6.5.5.1 Kollisionen erkennen

Zunächst widmen wir uns den Kollisionen. Das Erkennen von Kollisionen mittels Script kann zum Beispiel dazu verwendet werden, passende Kollisionssoundeffekte abzuspielen. Ein solches Feature ist vor allem in Virtual Reality wichtig, wo Spieler gerne Objekte durch die Gegend werfen.

Zur Kollisionserkennung stellt dir Unity drei Event-Methoden zur Verfügung, die du einfach in deinem Script einbauen kannst:

■ **OnCollisionEnter** (*collisionInfo: Collision*) : void

Diese Methode wird ausgeführt, wenn der *Collider*, der sich am selben *GameObject* wie dieses Script befindet, mit einem anderen *Collider* kollidiert.

■ **OnCollisionStay** (*collisionInfo: Collision*) : void

Diese Methode wird in jedem Update ausgeführt, solange die Kollision andauert.

■ **OnCollisionExit** (*collisionInfo: Collision*) : void

Diese Methode wird ausgeführt, wenn dieses *GameObject* zuvor mit einem anderen *GameObject* kollidiert ist und den anderen *Collider* jetzt nicht mehr berührt.

Als Parameter übergeben alle drei Methoden ein *Collision*-Objekt, welches diverse Informationen über die Kollision enthält: Dazu gehört unter anderem ein Verweis auf den jeweils anderen *Collider* (*collisionInfo.collider*) oder auch die Punkte, an denen sich die beiden *Collider* berühren (*collisionInfo.contacts*).

6.5.5.1.1 Beispiel: Kollisionssoundeffekte

In Listing 6.8 siehst ein einfaches *Script*, welches im Falle einer Kollision einen Soundeffekt an der jeweiligen Stelle abspielt. Das *Script* verwendet dafür immer den ersten Kontakt-Punkt der Kollision. Anschließend erzeugt das *Script* eine temporäre *AudioSource*, wie sie in Kapitel 6.3.3 beschrieben wurde, um an der jeweiligen Stelle einen Soundeffekt abzuspielen.

Listing 6.8 Ein Beispiel-*Script* für Kollisionssoundeffekte

```
using UnityEngine;
public class CollisionSoundEffect : MonoBehaviour
{
    //Über den Inspector musst du diese Variablen einen AudioClip zuweisen
    public AudioClip collisionSFX;
    void OnCollisionEnter(Collision collisionInfo)
    {
        //Hole ersten Kollisionspunkt
        Vector3 collisionPoint = collisionInfo.contacts[0].point;
        //Erzeuge eine temporäre AudioSource an der Stelle
        AudioSource.PlayClipAtPoint(collisionSFX, collisionPoint, .7f);
    }
    // Die anderen beiden benötigen wir für dieses Beispiel nicht,
    // Du würdest sie jedoch genauso verwenden:
    void OnCollisionStay(Collision collisionInfo) { }
    void OnCollisionExit(Collision collisionInfo) { }
}
```

Das *Script* kannst du einfach zu einem *GameObject*, das über einen *Collider* und ein *Rigidbody*-Component verfügt, hinzufügen. Anschließend musst du im Inspector noch der Eigenschaft *collisionSFX* einen passenden *AudioClip* zuweisen. Das *GameObject* sollte dann aussehen wie in Bild 6.50. Sobald das *GameObject* jetzt mit einem anderen *Collider* zusammenstößt, wird der Soundeffekt abgespielt.

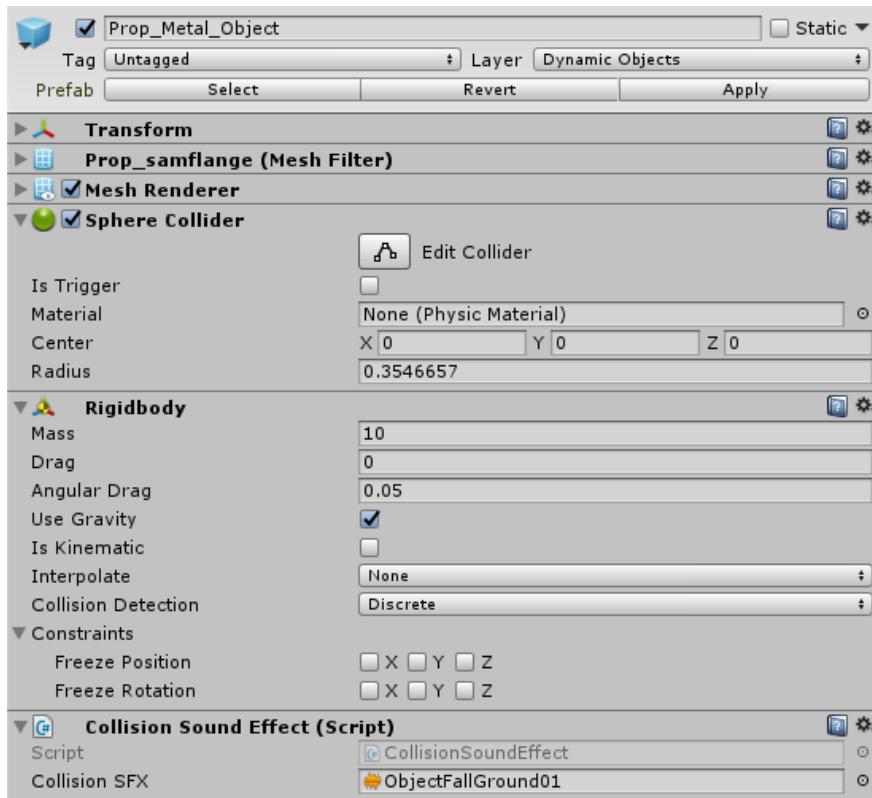


Bild 6.50 Konfiguration eines GameObjects mit einem *Collider*, einem *Rigidbody* und dem *CollisionSoundEffect*-Script

6.5.5.2 Trigger-Collider

Trigger haben eine große Anzahl an Einsatzmöglichkeiten. In vielen Fällen werden sie zum Beispiel verwendet, um zu erkennen, ob sich der Spieler derzeit in einem bestimmten Bereich der Scene befindet. Neben dem Spieler kann aber jedes GameObject, das über einen Collider verfügt, erkannt werden. Die einzige Bedingung ist: Entweder das GameObject mit dem Trigger-Collider *oder* das andere GameObject muss über ein Rigidbody-Component verfügen.

Wenn du zum Beispiel wie in Bild 6.51 einen *Trigger-Collider* an einem beweglichen Objekt, in diesem Fall ein Auto, befestigst, kannst du ihn nutzen, um Hindernisse frühzeitig zu erkennen und es automatisch bremsen oder ausweichen zu lassen.

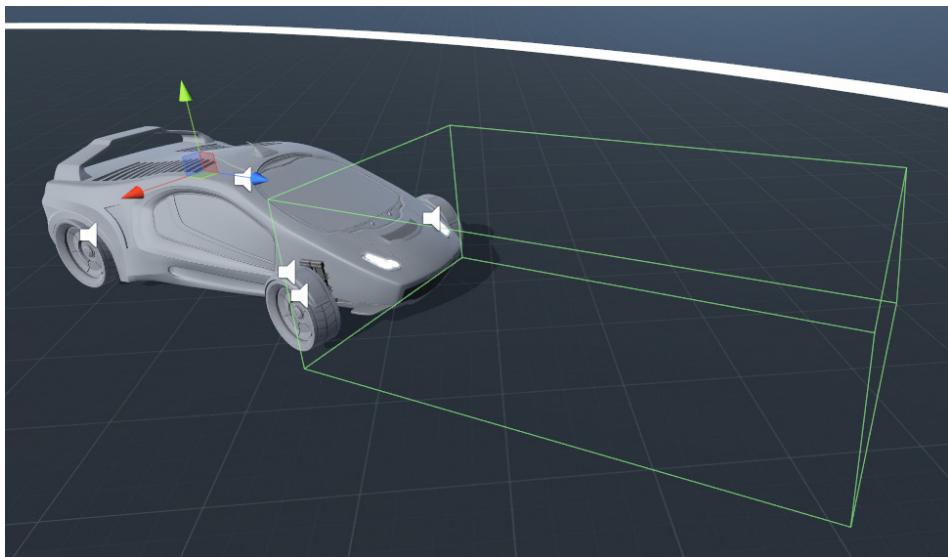


Bild 6.51 Ein Trigger-Collider vor einem Fahrzeug, um Hindernisse zu erkennen, bevor das Fahrzeug hineinfährt

Auch wenn die Einsatzmöglichkeiten vielseitig sein können, werden wir uns zunächst einmal auf den klassischen „Ist der Spieler in diesem Raum?“-Einsatzzweck beschränken, da dies in nahezu jedem Spiel relevant werden kann.

Ich habe, wie in Bild 6.52 zu sehen, zunächst einen Testraum erstellt. Ziel ist es zu erkennen, wann der Spieler diesen Bereich betritt. In unserem Beispiel soll in diesem Moment ein Licht angehen und ein Soundeffekt von einer *AudioSource* abgespielt werden. Der in dem Bild zu sehende *Collider* ist bereits über die Option *Is Trigger* als ein *Trigger-Collider* markiert. Die orange Astronautin ist unser Spieler, deshalb wurde ihr *GameObject* mit dem Tag „Player“ versehen. (Zur Erinnerung: Du kannst besondere *GameObjects* mittels *Tags* markieren, um sie von anderen *GameObjects* unterscheiden zu können. Den Tag für das derzeit ausgewählte *GameObject* kannst du ganz oben im *Inspector* auswählen.)

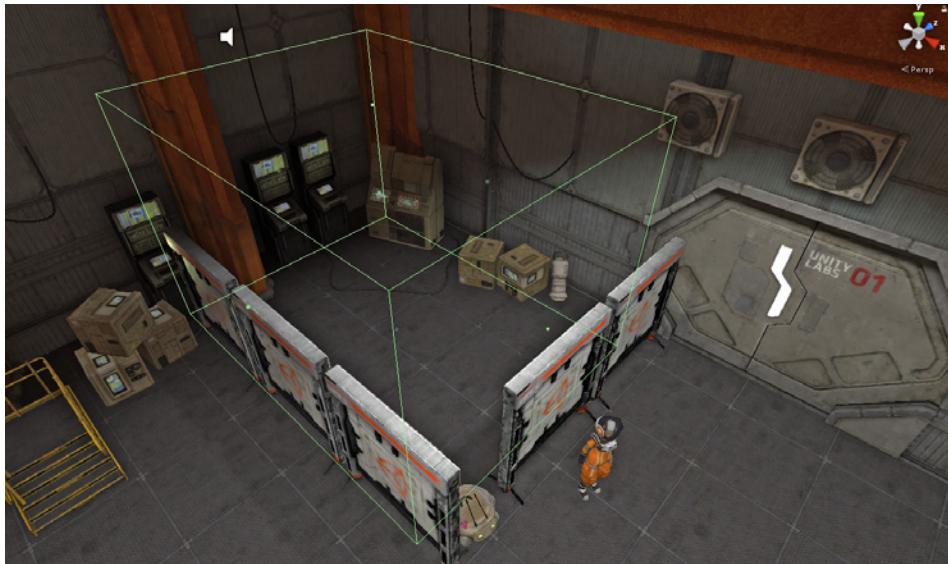


Bild 6.52 Der Trigger-Collider umfasst den abgesperrten Bereich.

Um nun zu erkennen, ob sich der Spieler innerhalb des von uns erstellten *Colliders* befindet, stehen uns drei Methoden zur Verfügung, welche den drei Methoden zur Kollisionserkennung stark ähneln. Im Gegensatz zu den *OnCollision*-Methoden übergeben die Trigger-Methoden allerdings direkt den *Collider* des jeweils anderen *GameObject*s.

- **OnTriggerEnter (*other: Collider*) : void**

Diese Methode wird für jedes *GameObject*, das einen *Collider* besitzt, ausgeführt, wenn es den Trigger-Bereich betritt. Betreten mehrere *Collider* gleichzeitig den Trigger-Bereich, wird diese Methode mehrmals hintereinander mit jeweils unterschiedlichen Parametern aufgerufen.

- **OnTriggerStay (*other: Collider*) : void**

Diese Methode wird in jedem Update, für jeden *Collider* ausgeführt, der sich innerhalb des Triggers befindet. Ein Beispiel für die Verwendung dieser Methode ist das Anti-Schwerkraft-Feld aus Kapitel 6.5.3.

- **OnTriggerExit (*other: Collider*) : void**

Wie *OnTriggerEnter* wird diese Methode für jedes *GameObject* ausgeführt, allerdings in diesem Fall, wenn es den Trigger-Bereich verlässt.

Wir fügen also zu dem *GameObject*, an dem sich auch der *Trigger-Collider* befindet, das Script aus Listing 6.9 hinzu, welches die drei *OnTrigger...-Methoden* implementiert:

Listing 6.9 Dieses Script zeigt dir beispielhaft, wie man die *OnTrigger*-Methoden verwenden kann, um zu prüfen, ob der Spieler einen Raum betreten hat.

```
using UnityEngine;
public class ActivateLightOnPlayerEnter : MonoBehaviour {
    public Light spotlight;
    public AudioSource audioSource;
```

```

public AudioClip sfxSwitchLight;
void OnTriggerEnter(Collider other)
{
    // Prüfe, ob der Collider, der gerade den Trigger betreten hat, den Tag
    // "Player" besitzt
    if (other.CompareTag("Player")){
        // Action
        Debug.Log("Spieler hat den Bereich betreten");
        spotlight.enabled = true;
        audioSource.PlayOneShot(sfxSwitchLight);
    }
}
void OnTriggerExit(Collider other)
{
    if (other.CompareTag("Player")){
        Debug.Log("Spieler hat den Bereich verlassen");
        spotlight.enabled = false;
        audioSource.PlayOneShot(sfxSwitchLight);
    }
}
// nicht benötigt:
void OnTriggerStay(Collider other) { }
}

```

Fügt man dieses Script zu dem *GameObject* des *Trigger-Colliders* hinzu und weist den Variablen eine passende Lichtquelle, Audioquelle und Soundeffekt zu, ist bereits alles erledigt. Wie in Bild 6.53 zu sehen, schaltet sich das Licht nun automatisch ein, wenn der Spieler den Bereich betritt. Zusätzlich wird das Licht auch automatisch wieder ausgeschaltet, wenn er ihn wieder verlässt.

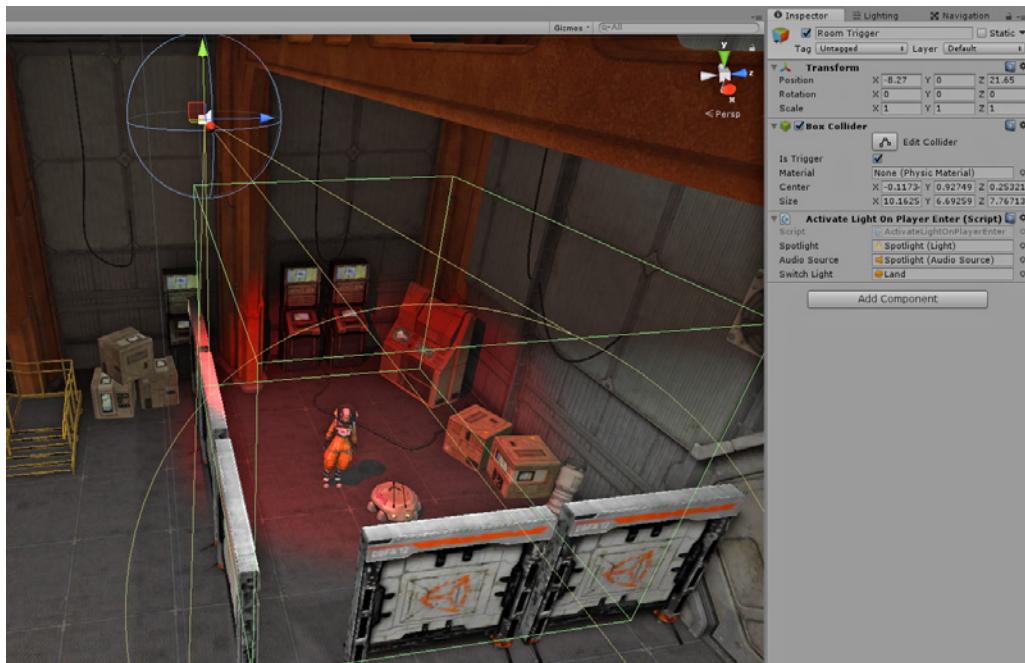


Bild 6.53 Das Licht und die Audioquelle werden automatisch durch das Script ausgelöst, wenn der Spieler den Trigger-Collider betritt.

6.5.6 Raycasts

Bei einem *Raycast* wird von einem bestimmten Punkt aus ein unsichtbarer Strahl in eine bestimmte Richtung geschossen, bis er auf das erste Hindernis trifft oder seine maximale Reichweite erreicht hat. Dadurch kann geprüft werden, ob sich in der jeweiligen Richtung ein Hindernis befindet oder nicht.

Raycasts werden zum Beispiel wie in Bild 6.54 verwendet, um zu prüfen, ob eine abgefeuerte Waffe ein Ziel trifft. Auch bei der künstlichen Intelligenz von Gegnern kommen Raycasts zum Einsatz, um beispielsweise zu testen, ob sie den Spieler sehen können oder die Sicht durch ein Hindernis blockiert wird.

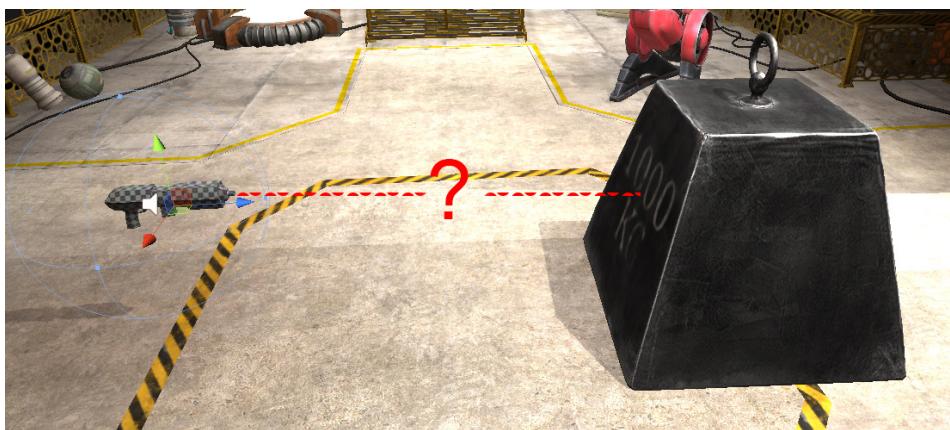


Bild 6.54 Mit Raycasts kannst du prüfen, ob eine Sichtlinie zwischen zwei Punkten besteht, oder auch herausfinden, was sich vor einem Objekt befindet.

Trifft ein Raycast auf ein Hindernis, kann anschließend genauer geprüft werden, ob es sich bei dem getroffenen *Collider* zum Beispiel um den Spieler oder eine Wand handelt.

Raycasts erzeugst du über die `Physics.Raycast`-Methode:

- `Raycast(origin: Vector3, direction: Vector3, out hitInfo: RaycastHit, [maxLength : float]) : bool`

Diese Methode schießt den Strahl durch deine Scene. Der Parameter `origin` bestimmt die Ausgangsposition, von wo der Strahl geschossen werden soll. Der Parameter `direction` ist ein Richtungsvektor, der die Schussrichtung bestimmt. In das `hitInfo`-Objekt speichert die Methode bei einem Treffer erweiterte Details zu dem Treffer. Darunter ist zum Beispiel auch ein Verweis auf den getroffenen *Collider* enthalten. Der optionale Parameter `maxLength` bestimmt, wie lang der Strahl maximal fliegen soll, wenn er auf kein Hindernis trifft.

Als kleines Beispiel siehst du in Listing 6.10 ein Script für eine *Physics Gun*. Diese Waffe schießt auf Tastendruck einen Raycast nach vorne und prüft, was der Raycast getroffen hat. Ist das getroffene Ziel ein *Rigidbody Object*, bei dem nicht die Option *Is Kinematic* aktiviert ist, übt die Waffe eine Kraft auf das Objekt aus und schleudert es nach hinten.

Listing 6.10 Dieses Script soll zu einem Waffen-Modell hinzugefügt werden und macht aus ihm eine Physics Gun.

```
using UnityEngine;
public class PhysicsGun : MonoBehaviour {
    public float gunImpulse = 20;
    public AudioSource audioSource;
    public AudioClip sfxGun;
    // Update is called once per frame
    void Update () {
        // Wenn Feuer Taste gedrückt
        if (Input.GetButtonDown("Fire1")){
            //Spiele Soundeffekt ab
            audioSource.PlayOneShot(sfxGun);
            // Erzeuge ein RaycastHit-Objekt für die Ergebnisse des Raycasts:
            RaycastHit hit;
            // Schieße einen Raycast von der aktuellen Position dieses GameObjects
            // nach "Vorne", mit einer maximalen Länge von 10 m
            if (Physics.Raycast(transform.position, transform.forward,
                out hit, 10f)) {
                //Wenn wir etwas treffen, prüfe, ob das Ziel einen Rigidbody
                //besitzt, der nicht Kinematic ist
                Rigidbody r = hit.collider.GetComponent<Rigidbody>();
                if(r != null && !r.isKinematic)
                {
                    // Wenn ja, füge dem GameObject einen Impuls zu
                    r.AddForce(transform.forward * gunImpulse,
                        ForceMode. VelocityChange);
                }
            }
        }
    }
}
```

6.5.7 Character Controller

Der *Character Controller* ist ein Component, das vor allem für das freie Herumlaufen in der Scene wichtig ist. Dazu zählen zum Beispiel alle *Locomotion*-Methoden, bei denen sich der Spieler nicht teleportiert.

Fügst du einen *Character Controller* zu einer Spielfigur hinzu, erhält sie automatisch einige wichtige Fähigkeiten, welche dir einige Arbeit abnehmen, wenn du eine Spielfigur frei herumlaufen lassen möchtest: Zunächst erhält das GameObject automatisch einen *Capsule Collider*, der verhindert, dass die Spielfigur durch Hindernisse hindurchlaufen kann. Dann bietet er verschiedene Methoden an, mit denen du die Spielfigur über ein Script in der Welt bewegen kannst. Das Besondere dabei ist, dass der *Character Controller* die Figur nicht nur bewegt, sondern dabei auch viele Sonderfälle berücksichtigt. Wenn du einen *Character Controller* verwendest, kann die Spielfigur nicht nur auf flachen Ebenen, sondern auch auf bergigen Untergründen laufen. Sogar Stufen können automatisch erklimmen werden, ohne dass du diese Funktionen selber programmieren musst. Das *Character Controller*-Component nimmt dir also einiges an Arbeit ab, wenn du deinen Spieler klassisch herumlaufen lassen möchtest.

Fügst du einen *Character Controller* zu einem *GameObject* hinzu, bieten sich dir im *Inspector* diverse Einstellungsmöglichkeiten, mit denen du seine Eigenschaften noch genauer anpassen kannst:

- **Slope Limit:** Die maximale Steigung in Grad, die dieser Charakter hochlaufen kann
- **Step Offset:** Die maximale Stufenhöhe, die dieser Charakter hochlaufen kann
- **Skin Width:** Dieser Wert legt eine gewisse Toleranz bei der Kollisionsabfrage fest, die verhindern soll, dass der Character Controller an jeder Ecke oder Kante hängen bleibt, wenn er sie streift. Falls du also häufig an Objekten hängen bleibst, erhöhe diesen Wert.
- **Min Move Distance:** Hier wird ein Mindestwert festgelegt, der erreicht werden muss, bevor sich der Charakter in Bewegung setzt. Dies kann zum Beispiel dazu dienen, leichte Joystick-/Thumbstick-Bewegungen zu ignorieren.
- **Center:** Legt das Zentrum des *Colliders* fest. Ändere diesen Wert, um ihn vertikal oder horizontal zu verschieben.
- **Radius:** Legt den Radius des *Colliders* fest. Dieser Wert sollte groß genug gewählt werden, damit der Spieler mit seinem Kopf nicht in Wänden stecken bleibt.
- **Height:** Legt die Höhe des *Colliders* und somit auch der Spielfigur fest.

Nur durch den *Character Controller* bewegt sich das jeweilige *GameObject* noch nicht, du benötigst noch ein Script, welches die Eingaben des Spielers liest und dadurch den *Character Controller* steuert. In dem Sci-Fi-Stealth-Shooter-Beispielprojekt werden wir uns mit dem Character Controller und seiner Steuerung nochmals detaillierter beschäftigen.

Wenn du aber schon ein wenig herumprobieren willst, findest du hier ein einfaches Script, das dir erlaubt, mit dem *Character Controller* durch eine beliebige *Scene* zu laufen. Mit dem Script kannst du immer nur in Blickrichtung laufen, umsehen musst du dich mit dem Headset.

Listing 6.11 Ein einfaches Script zur VR-tauglichen und einsteigerfreundlichen Kontrolle des Character Controllers

```
using UnityEngine;
public class SimpleVRCharacterController : MonoBehaviour {
    // Laufgeschwindigkeit
    public float walkSpeed = 1.5f;

    // Wenn aktiviert, läuft die Spielfigur von alleine,
    // ohne dass eine Taste gedrückt werden muss
    public bool autoWalk = false;

    // Verweis auf den zu verwendenden Character Controller
    private CharacterController characterController;

    void Start () {
        // Suche einen Character Controller am selben GameObject wie dieses Script
        characterController = GetComponent<CharacterController>();
    }
    void Update () {
        float currentWalkSpeed = 0;
        // Wenn versucht wird, mit Tastatur, Xbox-Game GearVR Touchpad zu laufen
        // laufe los. Daydream Nutzer müssen hier autoWalk verwenden, da ansonsten
```

```

        if((Input.GetButton("Vertical") && Input.GetAxis("Vertical") > 0)
           || Input.GetButton("Fire1") || autoWalk)
    {
        currentWalkSpeed = walkSpeed;
    }
    else if (Input.GetButton("Vertical") && Input.GetAxis("Vertical") < 0)
    {
        currentWalkSpeed = walkSpeed * -0.75f;
    }
    Vector3 moveSpeed = transform.forward * currentWalkSpeed + Physics.gravity;
    // Der Befehl SimpleMove bewegt den Charakter in die angegebene Richtung
    characterController.SimpleMove(moveSpeed);
}
}

```

Damit das Script funktioniert, musst du es zusammen mit einem *Character Controller* (**ADD COMPONENT/PHYSICS/CHARACTER CONTROLLER**) zu deiner *Main Camera* hinzufügen. Konfiguriere den *Character Controller* anschließend, wie es in Bild 6.55 zu sehen ist. Deine *Camera* sollte sich danach am oberen Ende des *Colliders* befinden.

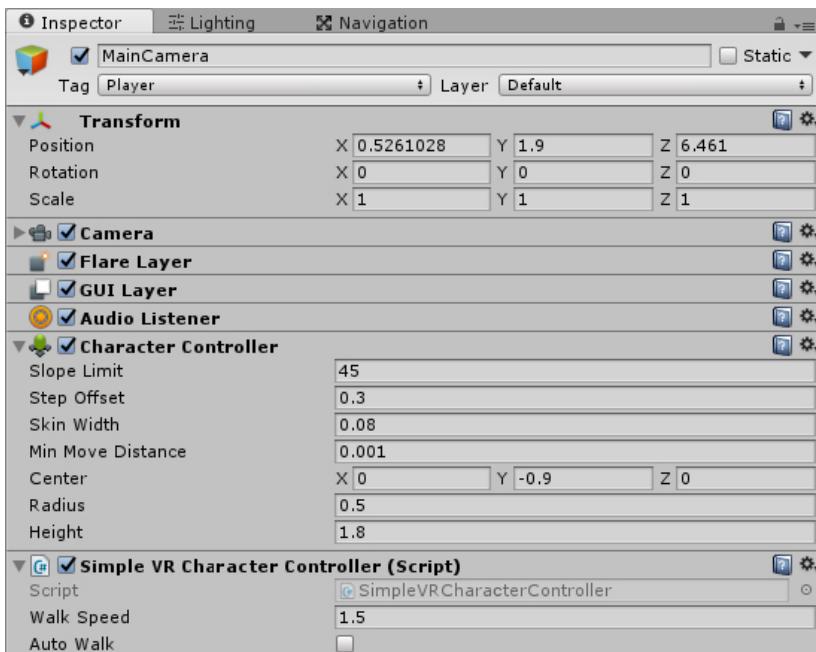


Bild 6.55 So solltest du deinen Character Controller an deinem Main-Camera-GameObject konfigurieren.

Das Script bietet dir folgende Steuerungsmöglichkeiten:

- **Oculus Rift & SteamVR:** Du kannst entweder über die Tasten **W** und **S** auf deiner Tastatur oder mit dem linken Stick eines Gamepads vorwärts oder rückwärts laufen. Oculus Touch und die SteamVR-Handcontroller schauen wir uns erst im nächsten Kapitel an. Damit dieses Beispiel auf deinem **SteamVR-Headset** funktioniert, musst du die *y-Position* der *Main Camera* auf 0 setzen.

- **GearVR:** Du kannst entweder auf das Touchpad deiner GearVR tippen oder den linken Stick eines Gamepads verwenden.
- **Daydream:** Hier musst du die Option *Auto Walk* aktivieren. Diese lässt dich dauerhaft laufen. Leider kann der Daydream-Handcontroller nicht ohne das SDK abgefragt werden, weshalb wir an dieser Stelle des Buches noch diesen Trick verwenden müssen.

■ 6.6 Partikeleffekte

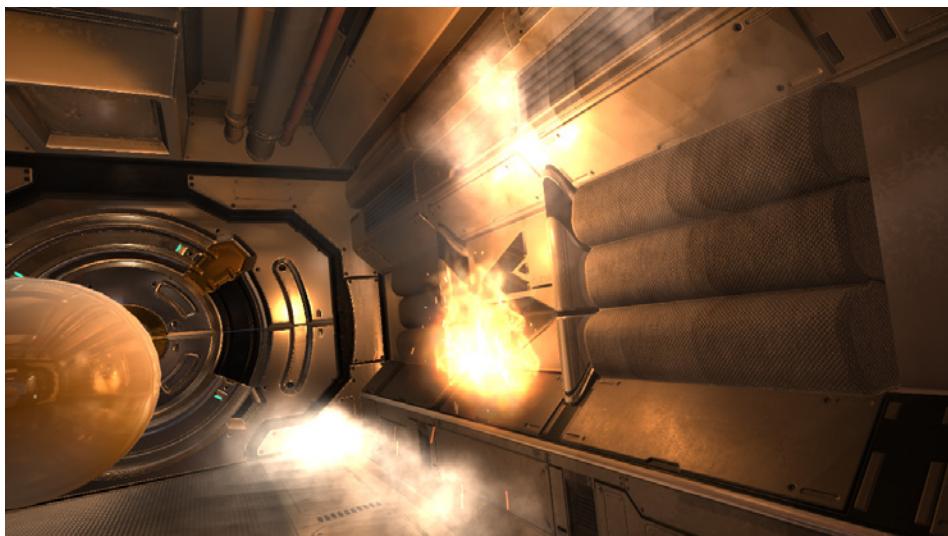


Bild 6.56 Beispiel-Szene mit Partikeleffekten

In Videospielen dienen Partikeleffekte häufig dem Zweck, die Spielwelt lebendiger und somit realer wirken zu lassen. Sie können zum Beispiel Rauch, Staub, Feuer, Wolken oder aber auch Explosionen, Zaubersprüche oder kleine Tiere wie Fliegen oder Schmetterlinge darstellen. Partikeleffekte bestehen entweder aus kleinen einfachen Bildern („Sprites“) oder 3D-Modellen („Meshes“), welche, ohne viel Performance zu kosten, in einer sehr großen Stückzahl gleichzeitig angezeigt werden können. Bei einem Partikeleffekt kommt es meist nicht auf einen einzelnen Partikel an, sondern der optische Effekt entsteht durch alle Partikel gemeinsam. Aus diesem Grund kannst du auch nicht jedes einzelne Partikel kontrollieren, sondern immer nur ein ganzes System. Du sagst einem *Partikelsystem*, wie der Effekt aussehen soll, und das Partikelsystem kümmert sich darum, die einzelnen Partikel zu bewegen.

Bei einem „Rauch“-Partikeleffekt würde das Partikelsystem zum Beispiel viele kleine, zweidimensionale Wolken-Sprites erstellen, welche sich mit einer zufälligen Geschwindigkeit, Größe und Richtung bewegen. Die einzelnen Sprites sehen nicht besonders beeindruckend aus, in hoher Stückzahl entsteht aber, wie in Bild 6.57 zu sehen, ein überzeugender Raucheffekt.

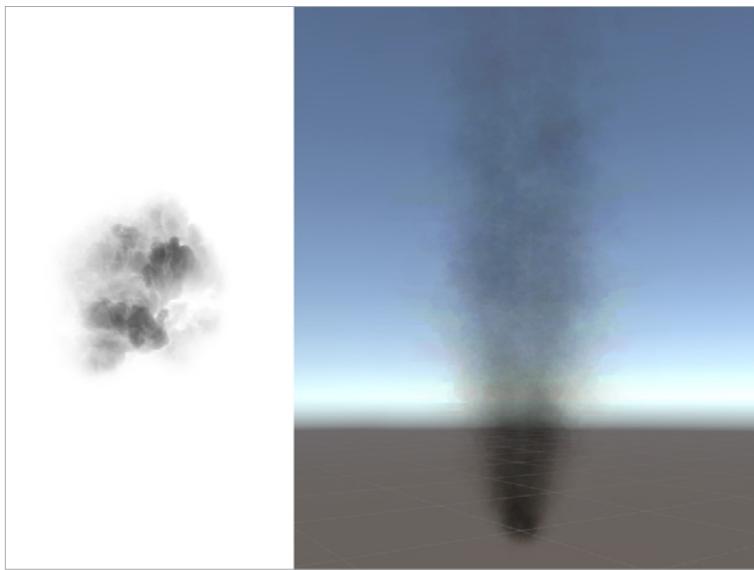


Bild 6.57 Links der Sprite eines einzelnen Partikels, rechts derselbe Sprite in einem Partikel-system, das für einen Raucheffekt konfiguriert ist

6.6.1 Partikelsysteme

Jedes Partikel hat eine vorbestimmte *Lifetime* (dt. Lebenszeit), in der es eine Vielzahl von Änderungen erleben kann. Am Anfang wird jedes Partikel von einem *Particle System*-Component generiert („emitted“). Wie viele Partikel ein *Particle System* pro Sekunde generiert, wird in erster Linie durch die *Emission Rate* bestimmt. Das *Particle System* bestimmt basierend auf seiner Konfiguration für jedes Partikel eine zufällige Startposition und einige andere Startparameter. Das Aussehen und Verhalten von jedem einzelnen Partikel ist dabei nicht von Anfang bis zum Ende gleich, sondern kann sich basierend auf verschiedenen Parametern wie Lebenszeit und Geschwindigkeit ändern. Nach welchen Regeln diese Änderungen passieren, legst du ebenfalls im *Particle System* fest.

Ein neues *Particle System* kannst du entweder über das *Create*-Menü in der *Hierarchy* erstellen (**CREATE/PARTICLE SYSTEM**) oder indem du das *Particle System*-Component zu einem beliebigen *GameObject* hinzufügst. Die Position des *GameObjects* entspricht dabei dem Zentrum des *Particle Systems*.

Bild 6.58 zeigt dir, wie das *Particle System*-Component im *Inspector* aussieht.

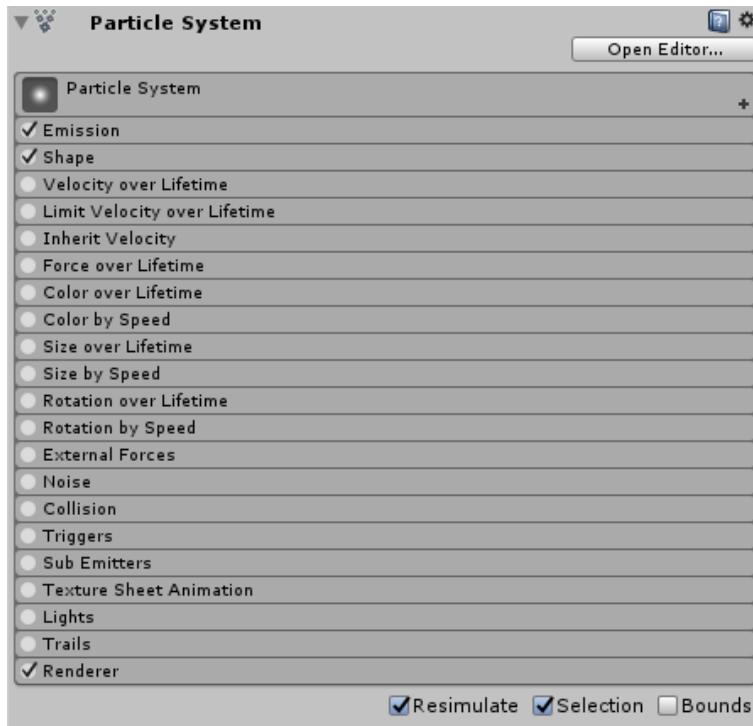


Bild 6.58 Das *Particle System*-Component mit allen Partikelmodulen, welche das Verhalten der Partikel bestimmen können

Ein *Particle System* besteht aus einem Hauptmodul, welches immer aktiv ist, und mehreren optionalen Partikelmodulen, die abhängig von der Geschwindigkeit, Lebenszeit sind oder auch zufällig das Aussehen von jedem einzelnen Partikel bestimmen können. Die einzelnen Module kannst du aus- und einklappen, indem du im Inspector auf ihre Namen klickst.

6.6.2 Particle-Effect-Steuerung

Wenn ein *GameObject*, das mit einem *Particle System*-Component ausgestattet ist, in der *Hierarchy* ausgewählt wird, erscheint in der *Scene View* die *Particle Effect*-Steuerung.

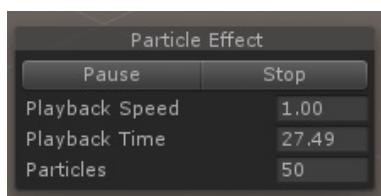


Bild 6.59 Die Particle-Effect-Steuerung

Die *Particle Effect*-Steuerung, welche du auch in Bild 6.59 sehen kannst, erlaubt es dir, das ausgewählte *Particle System* zu steuern. Über die **SIMULATE/PAUSE**-Schaltfläche kannst du das aktuelle *Particle System* pausieren und fortsetzen. Die **STOP**-Schaltfläche erlaubt dir, das *Particle System* in seinen Ausgangszustand zurückzusetzen. Sie ist vor allem für Effekte, die nur einmalig und nicht kontinuierlich abgespielt werden, wichtig. Ist ein System gestoppt, kannst du es über die **SIMULATE**-Schaltfläche erneut starten.

Zusätzlich kannst du in dem Feld folgende Eigenschaften ablesen und ändern:

- **Playback Speed:** Hier bestimmst du, wie schnell die Vorschau abgespielt werden soll. 1.00 ist die normale Abspielgeschwindigkeit.
- **Playback Time:** Hier wird angezeigt, wie viele Sekunden der *Particle Effect* bereits abgespielt wird. Du kannst hier auch eine Zeit eintragen, um zu einem beliebigen Zustand zu springen, ohne darauf warten zu müssen.
- **Particles:** Dieser Eintrag zeigt an, wie viele Partikel derzeit für diesen *Particle Effect* dargestellt werden. Die Zahl wird durch die Emissionsrate und die Lebenszeit der Partikel bestimmt und kann in diesem Fenster nur abgelesen werden.

6.6.3 Module des Particle Systems

In diesem Abschnitt werde ich dir zeigen, wie du die Module des *Particle System*-Components konfigurieren kannst, denn die Module bieten dir einige Konfigurationsmöglichkeiten, die so hauptsächlich hier auftreten. Außerdem werde ich dir die wichtigsten Module des *Particle System*-Components kurz vorstellen.

Da das *Particle System*-Component sehr komplex ist, empfiehlt es sich, für den Anfang fertig konfigurierte Partikelsysteme zu verwenden und diese anzupassen. Nach und nach lernst du dann, wie du bestimmte Effekte erreichen kannst.

Im *Inspector* kannst du mit einem Klick auf den jeweiligen Modulnamen die Parameter für das Modul auf- und zuklappen. Das Hauptmodul (in der Dokumentation *Particle Main module* genannt) kannst du im *Inspector* mit einem Klick auf *Particle System* auf- und zuklappen. Alle Module, bis auf das Hauptmodul, haben neben ihren Namen eine kleine Checkbox, mit denen du ihren Effekt jeweils aktivieren und deaktivieren kannst.

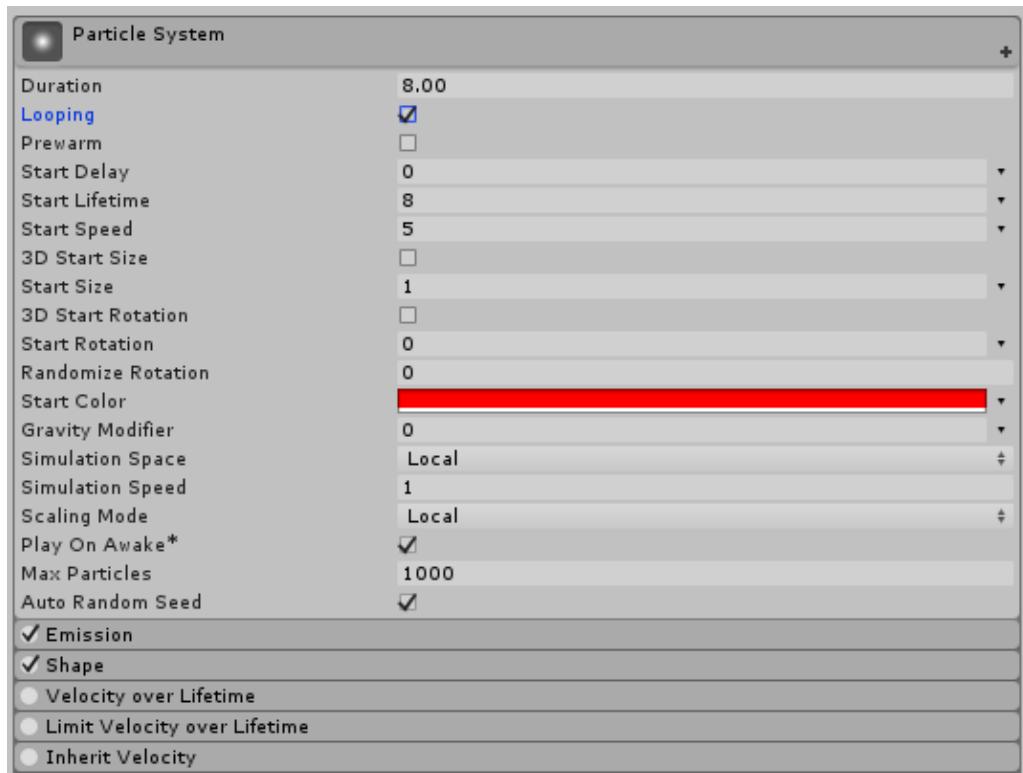


Bild 6.60 Ein Ausschnitt der Module des *Particle Systems*, das Hauptmodul ist aufgeklappt und seine Parameter sind sichtbar.

6.6.3.1 Besonderheiten bei den Partikelsystem-Eigenschaften

Beim *Particle System*-Component hast du für fast jeden Parameter die Möglichkeit, nicht nur einen festen Wert, sondern auch einen Wertebereich oder einen Wert in Abhängigkeit der verstrichenen Zeit anzugeben. Wie du das für Zahlenwerte und für Farben machen kannst und worin sich die Möglichkeiten unterscheiden, erkläre ich dir in den folgenden Abschnitten.

6.6.3.1.1 Numerische Parameter als Zahl, Kurve oder Bereich

Die meisten numerischen Parameter des *Particle System*-Components haben rechts neben dem Zahlenfeld einen kleinen, nach unten zeigenden Pfeil. Klickst du darauf, kannst du auswählen, dass du anstelle einer konstanten Zahl eine Kurve oder einen Bereich angeben möchtest.

Die unterschiedlichen Möglichkeiten sind:

- **Constant:** Diese Variante definiert einen festen Bereich, der über die gesamte Lebenszeit des Partikels gleichbleibt. Teilweise kann für die einzelnen Achsen jeweils ein eigener Wert angegeben werden.

- **Curve:** Hier kannst du eine Kurve (oder Funktion) angeben. Die Kurve definierst du in dem Kurveneditor unten im Inspector. Dort hast du die Möglichkeit, aus vordefinierten Kurven zu wählen oder sie manuell festzulegen. (Den Kurveneditor werden wir uns gleich auch noch im Detail anschauen.)
- **Random Between Two Constants:** Hier kannst du einen Min-/Max-Bereich angeben, aus dem für jedes Partikel ein zufälliger Wert bestimmt wird.
- **Random Between Two Curves:** Funktioniert genauso wie *Curve*, nur dass du hier zwei Kurven angeben kannst, welche einen Min-/Max-Bereich abhängig von einem weiteren Parameter (zum Beispiel Zeitachse) definieren. Für jedes Partikel wird dann ein zufälliger Wert aus dem Bereich bestimmt.

Der Kurveneditor

Den Kurveneditor findest du unter dem Namen *Particle System Curves* ganz unten im *Inspector*, wenn du einen der Werte auf „Curve“ umstellst und das Kurvenfeld einmal anklickst.

Unten in dem Editor stehen dir verschiedene vorgefertigte Kurven zur Wahl. Alternativ kannst du mit der linken Maustaste existierende Kurvenpunkte in dem Kurveneditor verschieben. Mit der rechten Maustaste kannst du über **ADD KEY** weitere Kurvenpunkte hinzufügen und so deine eigene Kurve bestimmen.

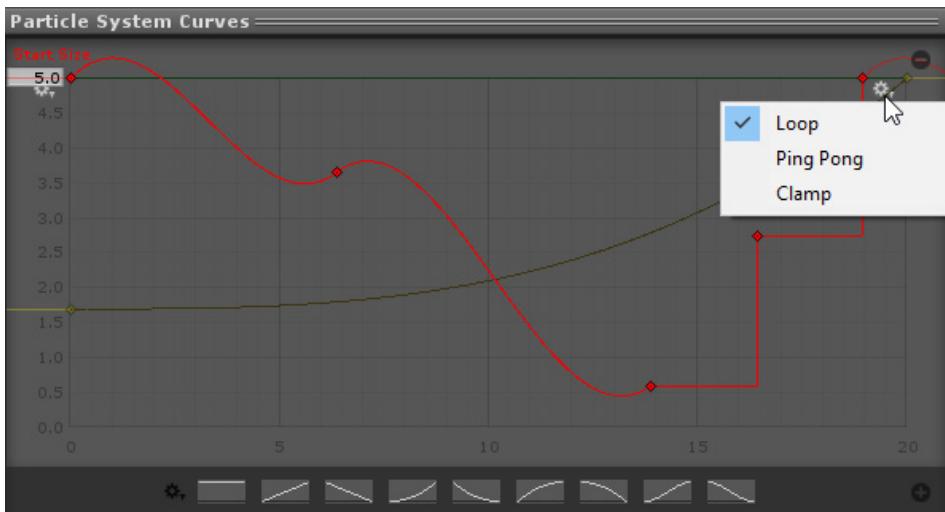


Bild 6.61 Mit dem Kurveneditor kannst du Kurven frei definieren.

Über das **KLEINE ZAHNRAD** kannst du, wie in Bild 6.61, wählen, was passieren soll, wenn das Ende der Kurve erreicht und der Effekt noch weiter ausgeführt wird (z. B. bei Endloseffekten).

- **Loop:** Wenn das Ende der Kurve erreicht wird, wird wieder am Anfang der Kurve begonnen.
- **Ping Pong:** Wenn das Ende der Kurve erreicht wird, wird die Kurve anschließend rückwärts ausgeführt, bis der Anfang erreicht wird. Danach wird sie wieder vorwärts ausgeführt und immer so weiter.

- **Clamp:** Wenn das Ende der Kurve erreicht wird, wird der letzte Wert der Kurve dauerhaft gehalten.

Außer bei den ... *over Lifetime*- und ... *by Speed*-Modulen bezieht sich die horizontale Achse der Kurven üblicherweise auf die Zeitachse des *Particle Systems*. Das bedeutet, je länger das *Particle System* läuft, desto weiter wird die Kurve nach rechts durchlaufen, bis die im *Hauptmodul* angegebene *Durration* (dt. „Dauer“) des Systems erreicht wird.

6.6.3.1.2 Farbparameter als Farbe, Verlauf oder Bereich

Ähnlich wie bei den numerischen Parametern hast du auch bei den Farbparametern die Möglichkeit, über den kleinen Pfeil, rechts neben der Farbe, zwischen unterschiedlichen Varianten auszuwählen, um die Farbe zu bestimmen.

- **Color:** Legt einen konstanten Farbwert fest.
- **Gradient:** Dies ist das Farbäquivalent zu einer Kurve: ein Farbverlauf. In dem *Gradient Editor* kannst du den Farbverlauf nach deinen Wünschen anpassen. Wie bei einer *Curve* wird basierend auf einem zweiten Wert (wie z. B. Zeitpunkt auf der Zeitachse) die entsprechende Farbe gewählt.
- **Random Between Two Colors:** Hier kannst du zwei Farben angeben und für jedes Partikel wird dann zufällig eine Farbe gewählt, die sich in dem Farbbereich zwischen diesen beiden Farben befindet.
- **Random Between Two Gradients:** Dies ist das Farbäquivalent von *Random Between Two Curves*. Du gibst zwei Farbverläufe an, welche einen Farbbereich definieren, der sich abhängig von einem weiteren Parameter verändert. Für jedes Partikel wird dann abhängig von diesem zweiten Parameter eine zufällige Farbe aus dem Farbbereich gewählt.

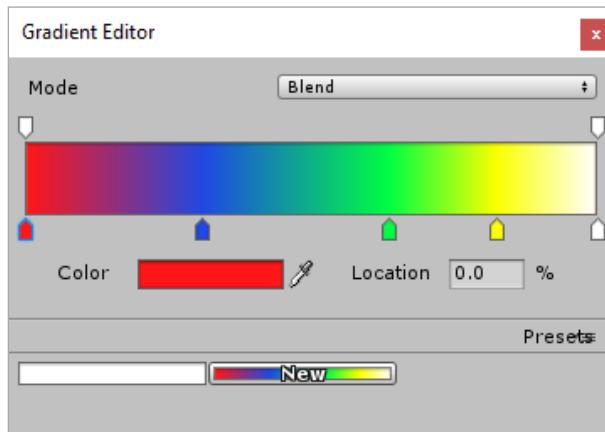


Bild 6.62 Mit dem „Gradient Editor“ kannst du Farbverläufe erzeugen und optional mit **NEW** abspeichern, um sie bequem wiederverwenden zu können.

6.6.3.2 Das Hauptmodul

Das Hauptmodul enthält wichtige allgemeine Einstellungen, welche das gesamte *Particle System* betreffen. Die meisten von den Parametern bestimmen den Ausgangszustand von

frisch erzeugten Partikeln, bevor ihr Aussehen und Verhalten von den anderen Modulen verändert wird.

Es stehen dir folgende Parameter zur Verfügung:

- **Duration:** Dieser Wert bestimmt, wie lange ein Durchlauf des *Particle Systems* dauert, also für wie lange das System neue Partikel erzeugen soll. Auch bei sich wiederholenden Effekten ist die Dauer wichtig, denn in Kurvenform angegebene Parameter sind meistens abhängig von dem Zeitpunkt in der Zeitachse des *Particle Systems*. (Eine Ausnahme stellen die Parameter in ... over Lifetime- und ... over Speed-Modulen dar.)
- **Looping:** Ist Looping aktiviert, beginnt das *Particle System* den *Particle-Effect* wieder von vorne abzuspielen, wenn die Duration überschritten wird. Es werden also endlos viele Partikel erzeugt.
- **Prewarm:** Wenn Looping und Prewarm aktiviert ist, wird das *Particle System* so gestartet, als würde es bereits eine Weile laufen. Das bedeutet, es befinden sich bereits mehrere Partikel in der Luft. Ohne Prewarm sind bei der Aktivierung noch keine Partikel vorhanden und man kann sehen, wie das *Particle System* nach und nach beginnt, Partikel zu erzeugen. Bei Effekten wie Regen, Staub oder ein Feuer, das schon beim Laden der Scene brennen soll, macht es Sinn, Prewarm zu aktivieren, damit es nicht so aussieht, als wäre das Feuer erst angegangen, als der Spieler die Scene betreten hat. Andersherum, wenn das Feuer erst durch den Spieler entfacht wird, sollte man Prewarm deaktivieren, damit die Flamme nicht plötzlich erscheint, sondern über einen kurzen Zeitraum größer wird.
- **Start Delay:** Bestimmt die Zeit, die das *Particle System* wartet, bevor es beginnt, Partikel zu erzeugen.
- **Start Lifetime:** Bestimmt die Lebenszeit eines frisch angelegten Partikels. Ist die Lebenszeit eines Partikels abgelaufen, verschwindet es.
- **Start Speed:** Bestimmt die Geschwindigkeit eines frisch angelegten Partikels.
- **3D Start Size:** Wenn es aktiviert ist, kannst du bei Start Size unterschiedliche Skalierungen für jede Achse angeben.
- **Start Size:** Bestimmt die Größe eines frisch angelegten Partikels.
- **3D Start Rotation:** Wenn aktiviert, kannst du die Start Rotation für jede Achse einzeln bestimmen.
- **Start Rotation:** Bestimmt den Drehmoment eines frisch angelegten Partikels.
- **Randomize Rotation:** Hier kannst du einen Wert zwischen 0 und 1 angeben, der die Wahrscheinlichkeit beeinflusst, dass sich ein Partikel in die entgegengesetzte Richtung dreht.
- **Start Color:** Die anfängliche Farbe neuer Partikel.
- **Gravity Modifier:** Gibt an, wie stark dieses Partikel von der Schwerkraft beeinflusst wird. Beispielsweise: 0 = keine Schwerkraft, 0.5 = halbe Schwerkraft, 1 = normale Schwerkraft, 2 = doppelte Schwerkraft usw.
- **Simulation Space:** Hier gibst du an, in welchem Koordinatensystem die Partikel berechnet werden, z. B. im lokalen Bereich des dazugehörigen *GameObjects*, im Weltkoordinaten- system oder im lokalen Bereich eines anderen Objektes.

Wenn sich das Particle System bewegt und du setzt den Wert auf „World“, entsteht ein Partikel-Schweif oder auch ein Trägheitseffekt, da sich bereits existierende Partikel nicht mit dem *GameObject* mitbewegen. Das kann z.B. bei Fackeln einen guten Feuer-Effekt ergeben. Im lokalen System sieht man es dem *Particle Effect* hingegen nicht an, ob es sich gerade bewegt oder nicht.

- **Simulation Speed:** Hier kannst du die Geschwindigkeit, mit der das *Particle System* abgespielt wird, dauerhaft verändern. (Während die „Playback Speed“-Option in der „Particle Effect“-Steuerung sich nur auf die Vorschau bezieht).
- **Scaling Mode:** Bestimmt, wie die Skalierung für das *Particle System* berechnet werden soll: Soll nur die lokale Skalierung des *GameObjects* verwendet oder sollen alle Skalierungen von übergeordneten *GameObjects* angewendet werden? Mit der Option *Shape* wird nur die Skalierung aus dem „Shape“-Modul verwendet und die Skalierung der *GameObjects* ist für die Größe des *Particle Systems* irrelevant.
- **Play On Awake:** Bestimmt, ob dieser *Particle Effect* sofort gestartet wird, wenn das *GameObject* aktiviert wird, oder ob du es per Code auslösen musst.
- **Max Particles:** Die Anzahl der gleichzeitig aktiven Partikel dieses *Particle Systems* ergibt sich aus der Emissionsrate und der Lebenszeit. Über diesen Parameter kannst du zusätzlich die maximale Anzahl an Partikeln begrenzen. Bei Erreichen der Grenze werden erst wieder neue Partikel erzeugt, wenn die Lebenszeit aktiver Partikel abgelaufen ist.
- **Auto Random Seed:** Wenn aktiviert, verhalten sich die Partikel bei jedem Abspielen ein wenig anders. Wenn du einen festen *Seed* angibst, verhalten sich die Partikel bei jedem Abspielen exakt gleich.

6.6.3.3 Module zum Erzeugen von Partikeln

Die Module in diesem Abschnitt beeinflussen, wann und wie neue Partikel von diesem *Particle System* erzeugt werden.

Emission-Modul

Das *Emission*-Modul bestimmt, wie viele Partikel zu welchem Zeitpunkt erzeugt werden. Ist das Emission-Modul deaktiviert, werden überhaupt keine Partikel erzeugt. Es gibt die Möglichkeit, die Emissionsrate pro Sekunde (*Rater over Time*) und pro zurückgelegter Unity-Einheit des *GameObjects* (*Rate over Distance*) anzugeben. Zusätzlich können auch sogenannte *Bursts* definiert werden, bei denen zum angegebenen Zeitpunkt (*Time*) eine zufällige Anzahl von Partikeln gleichzeitig erzeugt wird. Die Anzahl kann mit den Parametern *Min* und *Max* eingeschränkt werden.

Shape-Modul

Das Shape-Modul bestimmt, in welchem Bereich um die *GameObject*-Position herum Partikel erzeugt werden. Um einen natürlichen Effekt zu erzeugen, möchtest du in der Regel nicht, dass alle Partikel von exakt der gleichen Position ausgehen.

Mit dem *Shape*-Parameter kannst du eine Form bestimmen, an der sich das *Particle System* orientieren soll. Der Parameter bietet dir mehrere vordefinierte Formen an, alternativ kannst du aber auch ein eigenes Mesh angeben.

Die weiteren Parameter ergeben sich aus der jeweiligen gewählten Form und dienen fast alle dazu, die Größe und Ausrichtung der gewählten Form zu bestimmen. Einige Formen haben zudem die Option *Emit from*. Mit dieser Option kannst du bestimmen, ob die Partikel in dem gesamten Volumen oder nur an der Außenhülle erzeugt werden sollen.

6.6.3.4 Verändern von Partikeln basierend auf ihren Eigenschaften

Die Module zwischen dem Shape-Modul und dem Render-Modul dienen dazu, das Aussehen einzelner Partikel, basierend auf verschiedenen Eigenschaften, zu verändern. Zu den Eigenschaften zählen zum Beispiel Lebenszeit des Partikels (... *over Lifetime*) oder Geschwindigkeit (...*over Velocity*).

Damit sich die Werte abhängig von der Lebenszeit oder Geschwindigkeit etc. ändern, musst du in den Modulen Kurven oder Farbverläufe angeben. Andernfalls setzt du nur eine konstante Farbe. Je nach gewünschtem Effekt ist es häufig empfehlenswert, die *Random*-Varianten für die Parameter zu verwenden, damit sich die einzelnen Partikel zufällig und nicht wie geplant verhalten.

Eine Beschreibung für jedes einzelne Modul und auch jeden einzelnen Parameter der Module findest du in der Unity-Dokumentation, die ich dir in dem Kasten unten verlinkt habe.

 Die Online-Dokumentation für die Module des Unity Particle Systems:
<https://docs.unity3d.com/Manual/ParticleSystemModules.html>
Wähle in der Kapitelübersicht links das jeweilige Modul aus, über das du mehr erfahren möchtest.

6.6.3.5 Renderer-Modul

Das letzte Modul ist das Renderer-Modul. Dieses Modul bestimmt das grundlegende Aussehen der Partikel. Wenn es deaktiviert ist, sind die Partikel dieses Systems nicht mehr sichtbar. Innerhalb des Moduls kannst du angeben, ob die Partikel aus Sprites (*Billboard*-Grafiken) oder aus einfachen 3D-Modellen (*Mesh*) bestehen. 3D-Modelle dürfen allerdings maximal ein einziges *Sub-Mesh* verwenden, sie müssen also einfach aufgebaut sein.

Das *Material* bestimmt, welche Grafiken für die Partikel verwendet werden.

Mit den beiden Parametern *Cast Shadows* und *Receive Shadows* kannst du bestimmen, ob die einzelnen Partikel Schatten werfen bzw. empfangen sollen, jedoch können Schatten bei einer hohen Anzahl an Partikeln schnell zu Performance-Problemen führen.

Min und *Max Particle Size* bestimmen mit einer Zahl zwischen 0 und 1, wie viel Fläche des Bildschirms ein einzelnes Partikel maximal einnehmen darf.

6.6.4 Particle Effect Editor

Mit den Modulen können bereits interessante Partikeleffekte erstellt werden, für noch komplexere Effekte muss man aber häufig mehrere Partikelsysteme kombinieren. Bei einem Feuereffekt könnte zum Beispiel ein Partikelsystem nur für die Flammen und ein weiteres nur für den Rauch zuständig sein. Solche verschachtelten Effekte erstellst du am besten mit der Hilfe des *Particle Effect Editors*. Der Editor erlaubt dir, mehrere *Particle System*-Components gleichzeitig zu bearbeiten und in der *Scene View* als Vorschau anzusehen.

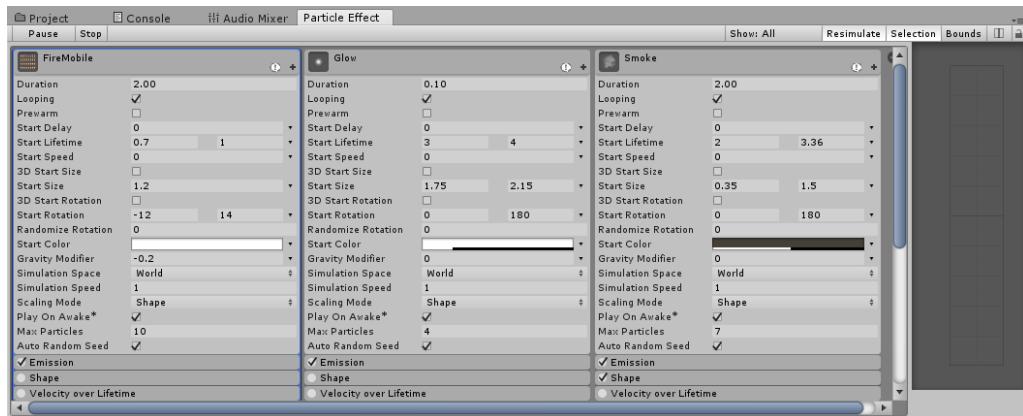


Bild 6.63 Im „Particle Effect“-Editor kannst du mehrere *Particle Systems* gleichzeitig bearbeiten.

6.6.5 Particle Systems via Script steuern

In vielen Fällen genügt es, Partikelsysteme nur über die zugehörigen *GameObjects* zu aktivieren und deaktivieren. Für komplexere Situationen kannst du das *Particle Systems*-Component aber auch über ein Script starten, pausieren, stoppen und die Parameter des Systems verändern. Außerdem kannst du Kollisionen von Partikeln mit der Umgebung erkennen.

Um ein Partikelsystem steuern zu können, benötigst du natürlich erst einen Verweis auf das jeweilige *Particle System*-Component. Diesen erhältst du am besten über eine `public`-Variable oder `GetComponent<ParticleSystem>()`, wenn sich das Component am gleichen *GameObject* befindet. Die Methoden `Play()`, `Pause()` und `Stop()` haben dieselbe Funktion wie die entsprechenden Tasten in der *Particle Effect*-Steuerung (siehe Kapitel 6.6.2). Listing 6.12 zeigt dir, wie du sie verwenden kannst.

Listing 6.12 ParticleSystem über Code steuern

```
void Update(){
    if(Input.GetKeyDown(KeyCode.Space)){
        particleSystem.Play();
    }
    if(Input.GetKeyDown(KeyCode.P)){
        particleSystem.Pause ();
    }
    if(Input.GetKeyDown(KeyCode.S)){
        particleSystem.Stop();
    }
}
```

```
        particleSystem.Stop();
    }
}
```

6.6.6 Standard-Partikeleffekte

Es ist nicht einfach, einen komplexen Partikeleffekt aus dem Nichts zu erstellen. Deshalb enthält Unity ein paar Beispiel-Partikeleffekte, welche du direkt benutzen kannst. Diese Effekte sind unterschiedlich komplex, sodass sie gut verwendet werden können, um sich in das Erstellen von Partikeleffekten einzuarbeiten.

Das *Unity Package* mit den Standardeffekten kannst du importieren, indem du in der Toolbar **ASSETS/IMPORT PACKAGE/PARTICLESYSTEMS** wählst.

■ 6.7 Landschaften mit Terrains erstellen

Die reale Welt ist keine flache Scheibe, deswegen sollte das deine Spielwelt auch nicht sein. Unitys *Terrain Engine* erlaubt es dir, große und abwechslungsreiche Landschaften für dein Spiel zu erzeugen. Der Untergrund für das Bergdorf aus Bild 6.64 wurde zum Beispiel mit ihr erstellt. Das Erstellen des Terrains erfolgt dabei über das *Terrain-Component*, welches dir einige einfach zu bedienende Werkzeuge zur Verfügung stellt.



Bild 6.64 Der unebene Untergrund sowie das Gras wurden mit Unitys Terrain Editor erstellt.

Ein neues Terrain legest du am besten über das *Create*-Menü in der *Hierarchy* an, da in diesem Fall automatisch der dazugehörige *Terrain Collider* mit angelegt wird. Dieser *Collider* passt sich automatisch der Form des von dir erstellten Untergrundes an und muss erst nicht weiter konfiguriert werden. Wähle also in der *Hierarchy*: **CREATE/3D OBJECT/TERRAIN**, wenn du ein neues Terrain anlegen möchtest.

Das *Terrain*-Component stellt dir diverse Werkzeuge zur Verfügung, um das Terrain zu bearbeiten. In Bild 6.65 ist die Werkzeugauswahl rot markiert. In dem Kasten darunter findest du immer eine kurze Beschreibung für das derzeit ausgewählte Werkzeug.

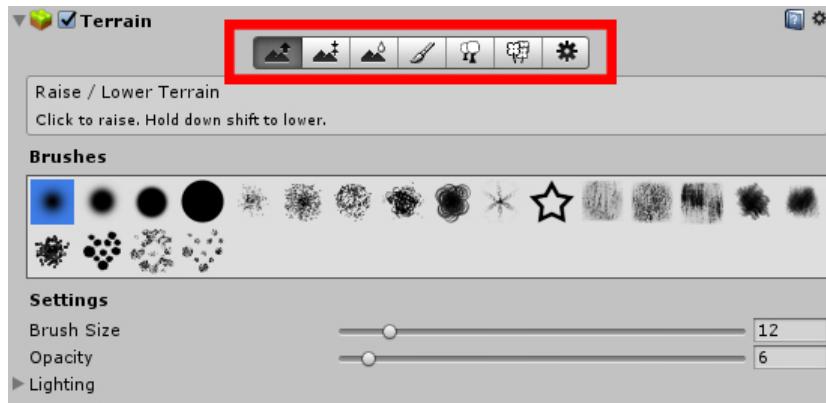


Bild 6.65 Das Terrain-Component im Inspector

Bis auf das Einstellungs-Panel ganz rechts besitzt jedes der Werkzeuge die Möglichkeit, einen *Brush* (dt. „Pinsel“) auszuwählen. Zusätzlich können stets auch noch die Größe des *Brushes* (*Brush Size*) und die Deckkraft (*Opacity*) bestimmt werden.

Es ist kein Zufall, dass diese Optionen an ein Bildbearbeitungsprogramm erinnern, denn genauso erstellst du auch dein Terrain: indem du mit den Pinseln in der *Scene View* direkt auf das *Terrain* malst. Die unterschiedlichen *Brushes* erlauben dir, interessante und abwechslungsreiche Details in das Terrain zu malen. Die echte Welt sieht schließlich auch nicht immer gleich aus.

Die Werkzeuge von links nach rechts:

1. **Umgebung erhöhen/vertiefen:** Mit diesem Werkzeug kannst du Berge und Täler in das Terrain malen. Wenn du beim Klicken die **SHIFT**-Taste gedrückt hältst, malst du Täler. Der *Opacity*-Parameter bestimmt, wie schnell sich die Landschaft ändert.
2. **Feste Höhe malen:** Bei diesem Werkzeug kannst du über den *Height*-Parameter (dt. „Höhe“) eine feste Höhe angeben, die bei Mausklick angewendet wird. Der *Opacity*-Parameter bestimmt, wie schnell sich die Landschaft zu der angegebenen Höhe ändert. Hältst du beim Klicken **SHIFT** gedrückt, wird die Höhe der angeklickten Position für den *Height*-Parameter übernommen.
3. **Höhe glätten:** Dieses Werkzeug glättet Kanten in der Landschaft, wenn du sie anklickst. Der *Opacity*-Parameter bestimmt, wie stark die Landschaft geglättet werden soll.
4. **Texture malen:** Mit diesem Werkzeug kannst du das Terrain buchstäblich anmalen. Wähle eine *Texture* aus und male sie auf das Terrain. Der *Opacity*-Parameter bestimmt die Deckkraft der *Texture*, sodass du auch mehrere *Textures* mischen kannst. *Textures* müssen zunächst über die Schaltfläche **EDIT TEXTURES...** angelegt werden.
5. **Bäume platzieren:** Dieses Werkzeug erlaubt dir, Bäume auf deinem Terrain zu platziern. Wähle den gewünschten Baum aus und male Baumgruppen mit der linken Maustaste. Hältst du beim Malen **SHIFT** gedrückt, löscht du die Bäume wieder. Die Parameter

Tree Density (dt. „Baumdichte“), *Tree Height* (dt. „Baumhöhe“) und *Color Variation* (dt. „Farbvariation“) erlauben dir, die oberen und unteren Grenzen für das zufällige Aussehen der gemalten Bäume zu festzulegen. Du musst zunächst jedoch über die Schaltfläche **EDIT TREES...** mindestens ein *Prefab* angeben.

6. **Details malen:** Dieses Werkzeug erlaubt dir, kleine Details wie Gras und Pflanzen auf dein Terrain zu malen. Mit linker Maustaste malst du, hältst du dabei **SHIFT** gedrückt, löschst du. Die Größe und Farbe eines jeden Details werden dabei zufällig bestimmt, um sie möglichst natürlich aussehen zu lassen. Über die Parameter *Opacity* und *Target Strength* bestimmt du, wie dicht die Details hinzugefügt werden sollen. Grass-Textures (*Sprites*) oder *Detail Meshes* müssen zunächst über die **EDIT DETAILS...**-Schaltfläche hinzugefügt werden. Bild 6.66 zeigt beispielhaft eine Konfiguration für natürlich aussehendes Gras. Die Min- und Max-Werte sowie die *Healthy*- (dt. „gesund“) und *Dry*-Farbwerte (dt. „trocken“) bestimmen, welche Werte der Zufallsgenerator als Grenzwerte verwendet.
7. **Terrain Einstellungen:** Hier kannst du diverse Einstellungen für das Terrain vornehmen; zum Beispiel ab welcher Distanz für die Bäume und Details eine geringere Detailstufe verwendet werden soll oder wie stark das Gras (*Details*) vom Wind hin und her geweht werden soll.

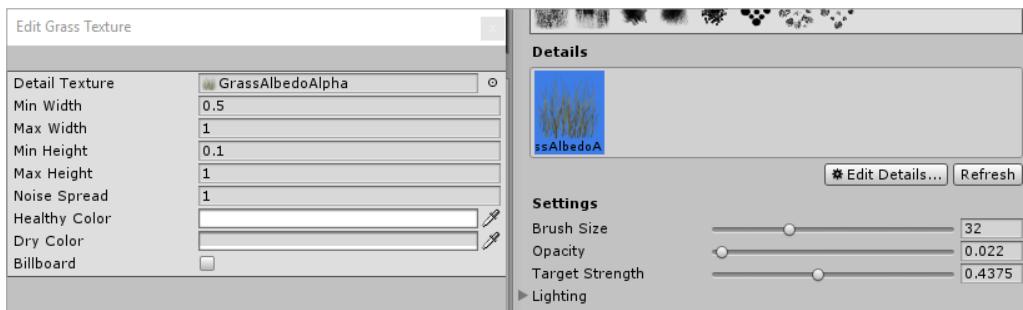


Bild 6.66 Beispielhafte Konfiguration für natürlich aussehendes Gras

💡

Beispiel-Textures für die Terrain-Erstellung

Wenn du ein Terrain erstellen möchtest, benötigst du zunächst Textures, die du dafür verwenden kannst. Unity stellt dir in dem Standard-Asset-Package „Environment“ ein paar Varianten zur Verfügung, die du für dein Spiel frei verwenden kannst.

Du importierst das Paket über die Toolbar: **ASSETS/IMPORT PACKAGE/ENVIRONMENT**

■ 6.8 Animation System

Egal ob Freund oder Feind, sobald du Lebewesen wie Menschen, Tiere oder auch abstoßende Monster in deine Spielwelt integrierst, benötigen sie gute Animationen. Denn vollkommen unabhängig von ihrer künstlichen Intelligenz erweckst du sie erst damit wirklich zum Leben.

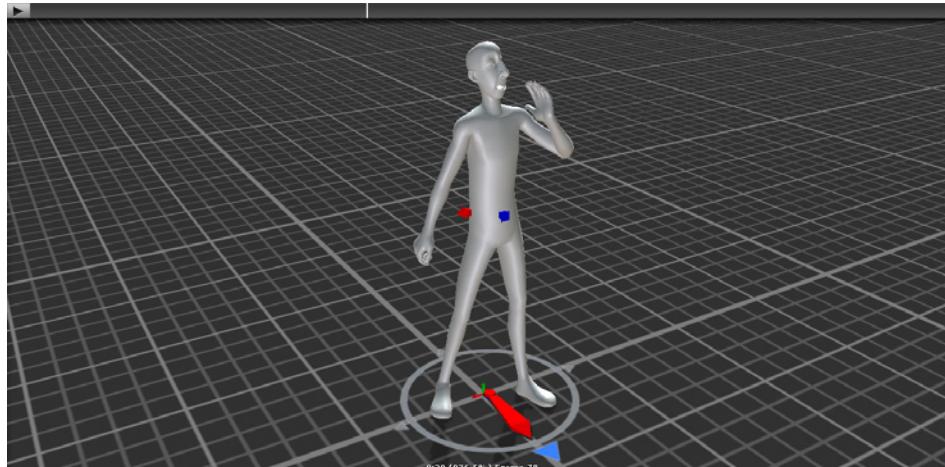


Bild 6.67 Eine Animation

Unity bietet dir mit *Mecanim* ein mächtiges System, um sowohl einfache als auch sehr komplexe Animationen nahtlos wiederzugeben.

Das *Animation System* besteht unter anderem aus diesen Teilen:

- **Animation Clip:** Dies ist Unitys Bezeichnung für eine einzelne Animation, z.B. eine „Lauf“-Animation oder eine „Warte“-Animation.
- **Animation Controller:** Dieser Bestandteil beschreibt einen Zustandsautomaten, welcher definiert, welche Animation-Clips zusammengehören und hintereinander abgespielt werden können. Außerdem legst du hier fest, welche Bedingungen erfüllt sein müssen, damit eine bestimmte Animation abgespielt werden darf. Über diesen Controller bestimmt das *Animation System*, welche *Animation Clips* abgespielt werden sollen.
- **Avatar:** Der Avatar beschreibt, wie dein 3D-Modell intern strukturiert ist. In der Regel legt Unity diesen automatisch an.
- **Animator Component:** Das *Animator*-Component ist der Teil, wo die einzelnen Teile zusammenlaufen. Der *Animator* verwendet einen *Animation Controller* und einen passenden *Avatar*, um das *GameObject*, zu dem es hinzugefügt wurde, zu animieren. Außerdem stellt dir dieses Component Methoden und Eigenschaften zur Verfügung, mit denen du die Animationen aus einem *Script* heraus steuern kannst.

Im Detail zu erklären, wie man ein 3D-Modell erstellt und animiert, würde den Rahmen dieses Buches sprengen, weshalb wir in diesem Kapitel existierende 3D-Modelle und Animationen verwenden, wie du sie z.B. aus dem *Unity Asset Store* herunterladen kannst.

Wichtig zu wissen ist jedoch: Nicht jedes Modell kann animiert werden. Es muss entweder aus mehreren einzelnen Objekten bestehen (z.B. Fahrradgestell, zwei Räder als einzelne Objekte) oder über ein sogenanntes *Skeleton-Rig* verfügen. Dieses „Skelett“ sagt dem *Animation System*, wo sich in dem 3D-Modell Gelenke befinden und wie lang die einzelnen Knochen sind. Wie du in Bild 6.68 sehen kannst, enthält das Skelett typischerweise nur die Gelenke, die auch tatsächlich für die Animation relevant sind. Wenn ein Modell über ein solches *Skeleton-Rig*, oder auch kurz *Rig*, verfügt, bezeichnet man es als *rigged*. Das *Rig* beschreibt, wo sich innerhalb des Modells Gelenke und Knochen befinden, also welche Stellen des Modells sich bewegen dürfen.

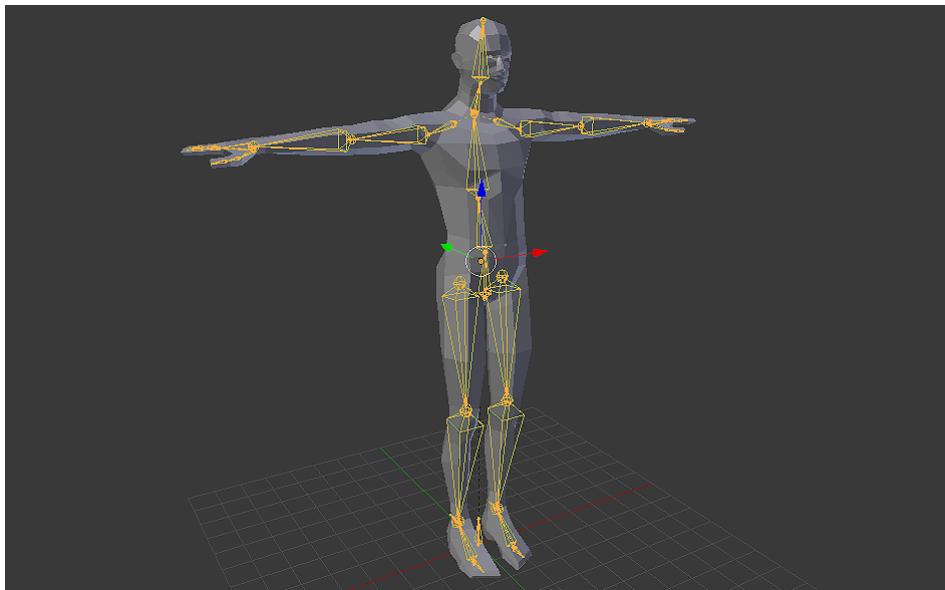


Bild 6.68 Die orangen Linien zeigen die Knochen und Gelenke des Rigs für dieses Modell.

Aktuelle Unity-Versionen sind auch noch mit dem alten Legacy-Animation-System ausgestattet, welches das *Animation-Component* nutzt. Auf die Details dieses veralteten Systems werde ich in diesem Kapitel nicht eingehen.

6.8.1 Animation Workflow

Das *Animation System* ist sehr komplex, weshalb ich dir zunächst eine Übersicht über den typischen Arbeitsablauf geben möchte. Wir beginnen beim Import der Animationen und Modelle und enden mit dem animierten *GameObject*.

Die Zahlen der nachfolgenden Auflistung entsprechen den Zahlen in Bild 6.69.

- 1. Modell importieren:** Wenn du ein animiertes 3D-Modell importierst, kannst du die einzelnen *Animation Clips* in Unitys *Project Browser* sehen. Dazu musst du über den kleinen Pfeil den Dateiinhalt des 3D-Modells ausklappen.

- 2. Animation Controller erstellen:** Anschließend erstellst du aus den *Animation Clips*, die du verwenden möchtest, einen *Animation Controller*. Am besten stellst du dir die einzelnen Animationen als Zustände vor und in dem *Animation Controller* bestimmst du, welche Zustände miteinander verbunden sein sollen. Für jeden Übergang kannst du Bedingungen bestimmen, wann der Zustand gewechselt werden soll.
- 3. Modell in die Scene ziehen:** Als Nächstes ziehst du das zu animierende 3D-Modell in deine Scene. Wenn das Modell über ein *Rig* verfügt, wird es automatisch mit einem *Animator*-Component ausgestattet, wo du den von dir erstellten *Animation Controller* angeben kannst. Der Avatar wird typischerweise automatisch generiert und von dem Component verwendet.
- 4. Animator via Code steuern:** Anschließend kannst du ein Script schreiben über das *Animator*-Component, das die Parameter deines *Animation Controllers* kontrolliert. Auf diese Weise kannst du dann die Animation deines *GameObjects* steuern.

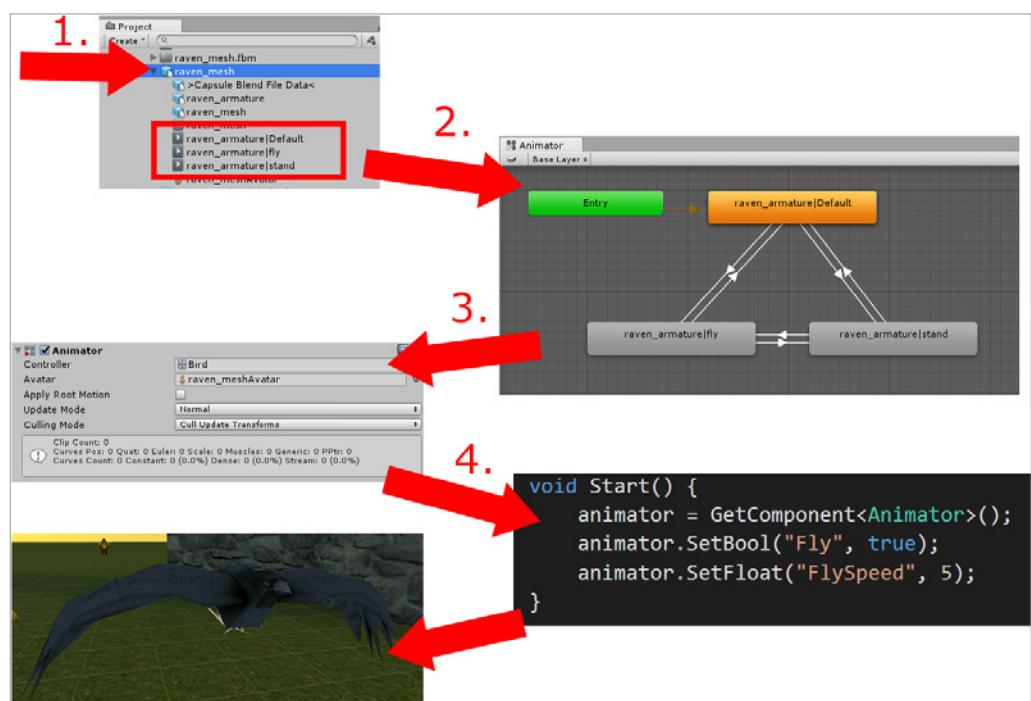


Bild 6.69 Der Workflow vom Import bis zum fertig animierten *GameObject*

Das waren auch schon alle Schritte, die du kennen musst, um ein animiertes 3D-Modell zu deinem Spiel hinzufügen zu können. In den folgenden Kapiteln werden wir uns die einzelnen Kapitel nochmals näher ansehen.

6.8.2 Humanoide Animationen und Avatare

Das *Mecanim*-Animation-System besitzt ein sehr wichtiges Feature namens *Animation Retargeting*. Um zu verstehen, warum es so wichtig ist, muss man zunächst wissen: Im Normalfall kann eine Animation nur für das Modell verwendet werden, für das sie erstellt wurde; höchstens noch für Modelle mit einem exakt identischen *Skeleton Rig*. Weil es aber keinen standardisierten Aufbau für das *Rig* gibt, unterscheiden sie sich häufig von Modell zu Modell. Der Aufbau ist zwar in der Regel grob ähnlich, im Detail gibt es jedoch je nach Hersteller Unterschiede: Manche haben mehr Gelenke, manche weniger, manche sind anders verschachtelt als andere. Es ist also im Normalfall nicht möglich, eine beliebige Animation für jedes andere Modell zu verwenden.

Hier kommt *Animation Retargeting* ins Spiel. Dieses Feature erlaubt dir, beliebige *humanoide* Animationen für beliebige *humanoide* Modelle zu verwenden, und zwar selbst dann, wenn der innere Aufbau des Modells nicht identisch ist. Realisiert wird das über den sogenannten *Avatar*. Dieser sagt der Engine, welcher Teil des 3D-Modells welchem humanoiden Körperteil entspricht.

Humanoid bedeutet, dass die Modelle und Animationen ähnlich wie ein Mensch aufgebaut sein müssen. Sie benötigen, vereinfacht gesagt, einen Kopf, einen Körper, zwei Arme und zwei Beine, wie Bild 6.70 zeigt. Ob das Modell nun tatsächlich einen Menschen darstellt oder zum Beispiel einen Ork, ist nicht relevant. Du kannst mit diesem System aber nicht eine für Menschen vorgesehene Animation für eine achtbeinige Spinne verwenden und ein gutes Ergebnis erhalten. Tiere wie Spinnen oder Vögel benötigen weiterhin spezielle Animationen, die gezielt für das jeweilige Modell erstellt wurden.

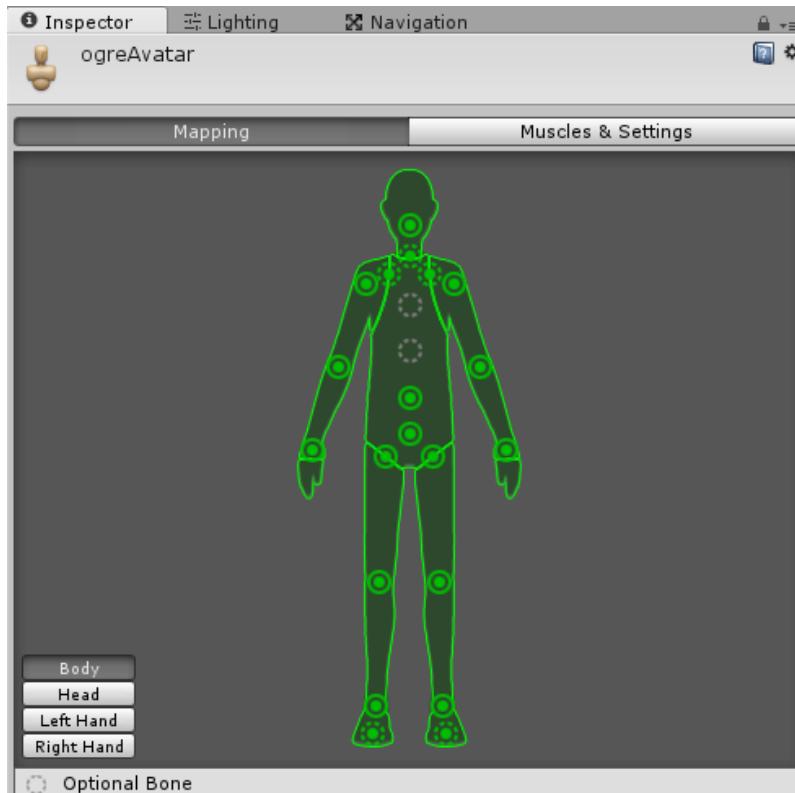


Bild 6.70 Ein humanoider Avatar: Die gestrichelten Gelenke sind optional, alle anderen müssen in dem *Rig* des humanoiden Modells vorhanden sein.

In der Praxis erlaubt dir diese Funktion, dass du einfach irgendein Character-Modell, das über ein *Rig* verfügt, aus dem Asset Store herunterladen und es mit jeder beliebigen *humanoiden Animation* verwenden kannst. Das Modell und die Animation müssen nicht aufwendig aufeinander abgestimmt werden, wie es ohne diese Funktion nötig wäre.

In den allermeisten Fällen musst du nicht einmal selbst den Aufbau des Modells angeben: Unity erkennt in der Regel automatisch, welcher Teil des Modells welchem humanoiden Körperteil entspricht. Alles, was du also tun musst, ist, sowohl die Animation als auch das Modell als *Humanoid* zu markieren. Wie du das machst, erkläre ich dir in Kapitel 6.8.3.

6.8.3 Modelle & Animationen importieren

Beim Importieren von Animationen stehen dir viele Einstellungsmöglichkeiten zur Verfügung, weshalb ich dir in diesem Abschnitt einmal die wichtigsten kurz vorstellen möchte.

Wenn du ein 3D-Modell, welches Animationen beinhaltet, in dein Projekt importierst, erkennt Unity diese automatisch und zeigt sie dir im *Project Browser* an. In Bild 6.71 hat das Modell „humanoid_small_steps.fbx“ zum Beispiel zwei Animationen, welche du an dem grauen Icon mit dem Play-Symbol erkennst.

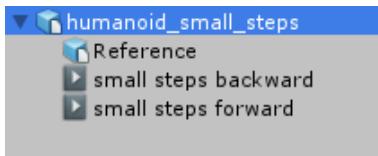


Bild 6.71 Dieses Modell enthält zwei Animationen.

Grundsätzlich kannst du die Animationen so schon verwenden, allerdings möchtest du in den meisten Fällen noch ein paar weitere Einstellungen vornehmen, die wir nun besprechen.

6.8.3.1 Animation oder Modell Humanoid markieren

Wenn dein Modell oder Animation eine humanoide Figur darstellt, musst du Unity noch mitteilen, dass es dieses Asset als eine humanoide Animation interpretieren soll. Das machst du, indem du die jeweilige Datei (nicht die Animation darin), wie in Bild 6.72 zu sehen, im *Project Browser* anklickst und dann im *Inspector* zum Reiter **Rig** wechselst. Dort findest du die Option *Animation Type*, wo du den Typ *Humanoid* auswählen kannst. Diese Option musst du sowohl für das Modell, das du verwenden möchtest, als auch für die Animationen entsprechend anpassen und schon kannst du sie miteinander verwenden.

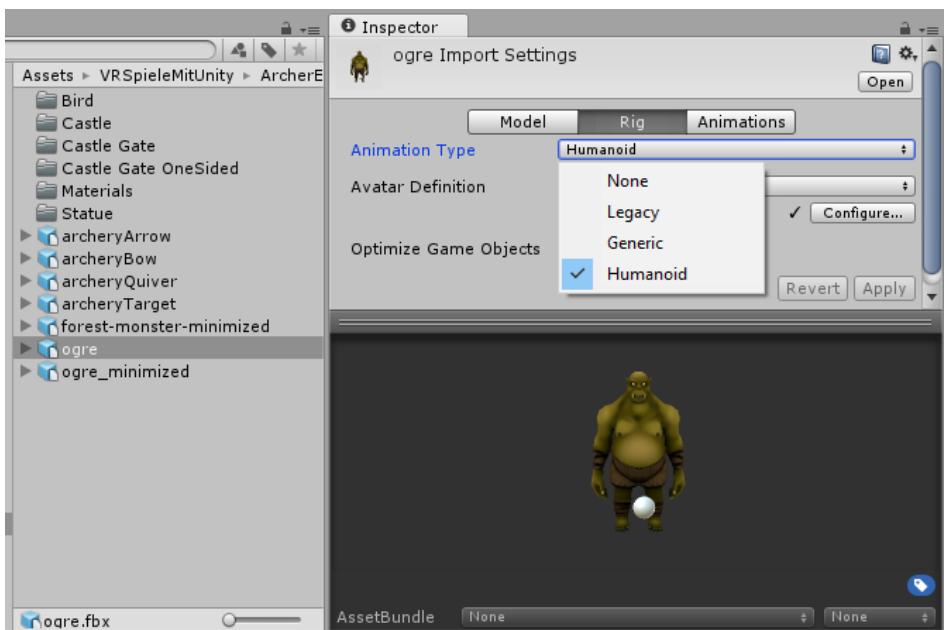


Bild 6.72 Modelle, die als humanoid markiert sind, sind kompatibel mit allen Animationen, die ebenfalls als humanoid markiert wurden.

Wenn du diese Einstellung vorgenommen hast, sind deine Modelle und Animationen bereits mit anderen Humanoiden-Assets kompatibel.

 **Animation-Type-Fehler erkennen:**

Sollte eine Animation mal nicht abgespielt werden, sondern dein Charakter hockt nur in einer sprungähnlichen Animation in der Luft, versuchst du wahrscheinlich, ein humanoides Modell mit einer nicht als „humanoid“ markierten Animation zu verwenden – oder anders herum.

6.8.3.2 Animation Rotation und Loop

Als Nächstes schauen wir uns ein paar Einstellungen an, die du für Animationen vornehmen kannst. Dazu zählt zum Beispiel, dass die Animation korrekt gedreht ist oder dass sie automatisch in einer Endlosschleife (*loop*) abgespielt wird.

Klicke zunächst die Datei, welche die Animation beinhaltet, im *Project Browser* an. Im *Inspector* siehst du dann die *Import Settings* für diese Datei. Wechsel hier zu dem Reiter **ANIMATIONS**.

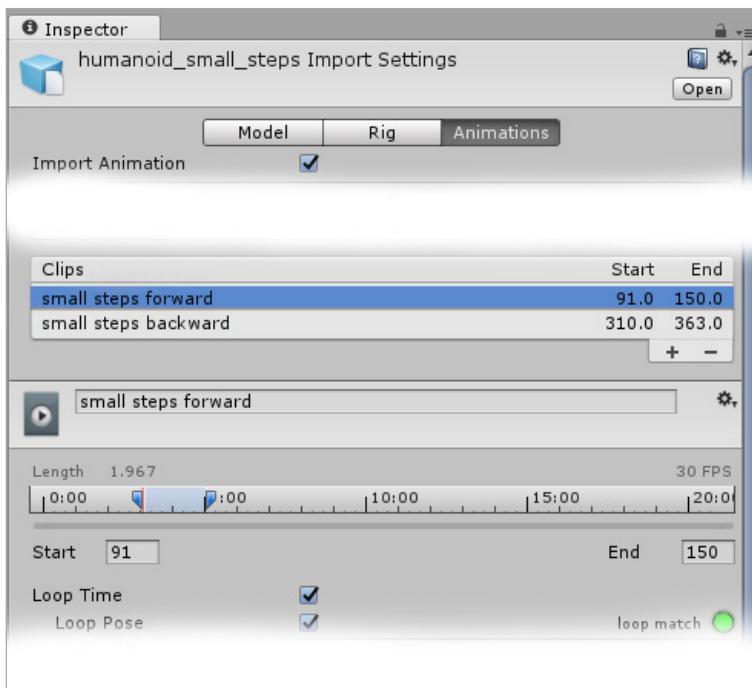


Bild 6.73 Die relevanten Einstellungen findest du in der Clip-Auswahl abwärts.

In dem Animations-Reiter, unter dem Punkt *Clips*, findest du alle *Animation Clips*, die Unity innerhalb der ausgewählten Datei gefunden hat. Hier kannst du den *Clip* auswählen, den du bearbeiten möchtest. Sollte deine Datei, wie in Bild 6.73, mehrere *Animation Clips* beinhalten, musst du die folgenden Einstellungen für jeden der *Clips* vornehmen. Meine Empfehlungen sollen dir als Hilfestellung für den Anfang dienen. Bei besonderen Animationen oder

je gewünschtem Effekt kann es durchaus sinnvoll sein, eine andere Einstellung zu wählen, im Normalfall ist das aber nicht notwendig.

- **Length, Start, End:** Die Zeitleiste bestimmt den Start- und Endpunkt für den *Animation Clip*. In der Regel musst du hier keine Anpassungen vornehmen.
- **Loop match:** Dieser Indikator zeigt dir an, wie gut sich diese Animation nahtlos wiederholen lässt. Im Idealfall sollten für unsere Einstellungen alle Indikatoren, bis auf der für den Punkt „Root Transform (XZ)“, auf *Grün* stehen.
- **Loop Time:** Gibt an, ob diese Animation automatisch in einer Endlosschleife abgespielt werden soll oder nur einmalig. Für Bewegungsanimationen wie zum Beispiel Gehen oder Rennen solltest du diese Option aktivieren. Ausnahmen stellen Animationen dar, die in der Regel nicht wiederholt werden, z.B. Sprung-Animationen (hier würde vor und nach dem Sprung die Laufanimation abgespielt werden).
- **Loop Pose:** Korrigiert die Animation automatisch so, dass eine nahtlose Wiederholung möglich ist. Dadurch kann man den Zeitpunkt des Neuanfangs nicht mehr erkennen. Diese Option ist vor allem dann hilfreich, wenn der „Loop-Match“-Indikator nicht grün ist. Die Korrektur verwendet jedoch nur eine einfache Interpolation, wodurch die Ergebnisse bei zu großen Unterschieden nicht optimal sein können.

Empfehlung: Aktiviert, wenn Loop Time ebenfalls aktiviert ist.

- **Root Transform Rotation:** Diese Optionen beziehen sich auf die Rotation des Modells. Du solltest darauf achten, dass der Charakter in der Vorschauansicht in dieselbe Richtung schaut wie der rote Pfeil am Boden. Der rote Pfeil sollte wiederum möglichst auf den hellblauen Pfeil zeigen, der die aktuelle Laufrichtung angibt.
- **Bake Into Pose:** Bei vielen Animationen ist die Rotation nicht perfekt. Wenn die Animation dann in einer Endlosschleife gespielt wird, kommt es nach einigen Wiederholungen zu Abweichungen; also z.B., dass der Charakter um eine leichte Kurve läuft, obwohl er nur geradeaus laufen sollte. Über diese Option kannst du die Rotation in der Animation sperren.

Empfehlung: Aktiviert

- **Based Upon:** Soll das *Animation System* die Rotation der Animation basierend auf der Basis des Rigs (*Original*) oder der Rotation des Modells (*Body Orientation*) erkennen?

Empfehlung: Original

- **Root Transform Position (Y):** Diese Optionen beziehen sich auf Änderungen des Charakters auf der vertikalen Achse (zum Beispiel für Sprünge).

- **Bake Into Pose:** Verhindert Bewegungen des Charakters auf der y-Achse während der Animation.

Empfehlung: Aktiviert

- **Based Upon:** Soll das *Animation System* die y-Position des Modells basierend auf der Basis des Rigs (*Original*), den Füßen der Figur (*Feet*) oder dem Schwerpunkt des Körpers (*Center of Mass*) erkennen?

Empfehlung: Original

- **Offset:** Stimmt in der Vorschau der rote Pfeil nicht mit dem hellblauen Pfeil überein, kannst du hier einen Korrekturwert in Grad angeben.

- **Root Transform Position (XZ):** Diese Optionen beziehen sich auf horizontale Änderungen des Charakters auf der x- und z-Achse (zum Beispiel Laufbewegungen).
 - **Bake Into Pose:** Verhindert Bewegungen des Charakters auf der x-z-Ebene.
Empfehlung: Deaktiviert
 - **Based Upon:** Soll das *Animation System* die horizontale Position des Charakters basierend auf der Basis des Rigs (*Original*) oder dem Schwerpunkt des Körpers (*Center of Mass*) erkennen?
Empfehlung: Center of Mass
- **Mirror:** Spiegelt links und rechts für diese Animation
Empfehlung: Deaktiviert

Wenn du die Einstellungen wie hier beschrieben vorgenommen hast, sollten die Rotation und Position des Charakters in dem Vorschaufenster aussehen wie in Bild 6.74. Wenn du die Animation abspielst, kann der Charakter laufen, er sollte dabei allerdings nicht die Richtung ändern (es sei denn, das ist von dir gewollt).



Bild 6.74 Eine fertig konfigurierte Laufanimation

6.8.4 Animation Controller

Animation Controller erlauben es dir, sehr bequem eine Vielzahl von *Animation Clips* zu verwalten und zu steuern, ohne lange und komplizierte *Scripte* schreiben zu müssen. Der *Animation Controller* übernimmt vollkommen automatisch das Überblenden zwischen ein-

zernen Animationen und er legt auch fest, in welcher Reihenfolge Animationen abgespielt werden dürfen.

Bild 6.75 zeigt beispielhaft einen einfachen *Animation Controller* mit mehreren *States* (Zuständen). Zu jedem *State* gehört ein *Animation Clip*, der abgespielt wird, wenn der State aktiv ist. Der orange State ist der sogenannte *Default State* (dt. „Standardzustand“). Dieser State ist standardmäßig aktiv, wenn das *GameObject* aktiviert wird. In vielen Fällen verwendet man dafür eine *Idle*-Animation. Also eine „Nichtstun“-Animation, bei der die jeweilige Figur nur herumsteht.

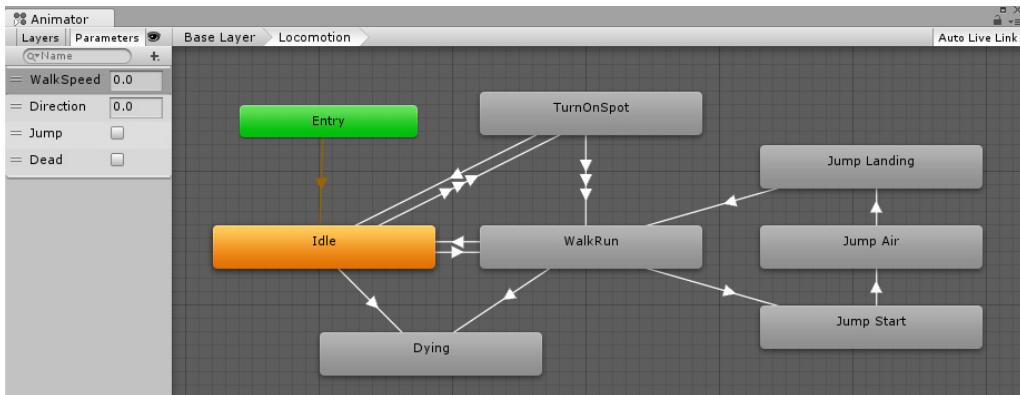


Bild 6.75 Ein einfacher Animation Controller mit mehreren Zuständen

Von diesem State aus können verschiedene andere States erreicht werden. Dabei ist an jeden der Übergänge (die weiße Pfeile) eine bestimmte *Bedingung* geknüpft, die in den meisten Fällen abhängig von den *Parametern*, links im Bild, ist. Ist der Parameter *WalkSpeed* zum Beispiel größer 0, wird von dem „Idle“-State in den „WalkRun“-State gewechselt. Von diesem aus können wiederum weitere States, wie zum Beispiel „Jump Start“, erreicht werden. Damit dieser Übergang stattfindet, müssen lediglich die entsprechenden Übergangsbedingungen erfüllt sein. Die Bedingungen kannst du im *Inspector* ansehen und verändern, indem du einfach auf den gewünschten Übergangspfeil klickst.

Bild 6.76 zeigt beispielsweise den Übergang von dem „Idle“-State in den „WalkRun“-State. Die Option **Has Exit Time** bestimmt, ob die unter *Conditions* angegebenen Bedingungen kontinuierlich oder nur am Ende der Animation überprüft werden soll. Ist *Has Exit Time* wie im Bild deaktiviert, wechselt der *Animation Controller* sofort zum nächsten State, sobald die Bedingung erfüllt ist. Ist die Option aktiv, wird der State frühestens bei der im Zeitstrahl darunter angegebenen Zeit verlassen. Diese Funktion ist in den meisten Fällen für spezielle Animationen sinnvoll, die nicht mittendrin abgebrochen werden können (z. B. ein Salto oder eine Geste). In dem Zeitstrahl kannst du auch die Länge der Überblendung zwischen den beiden Animation Clips bestimmen. Die Dauer der Überblendung wird auch verwendet, wenn *Has Exit Time* deaktiviert ist und der Clip jederzeit verlassen werden kann.

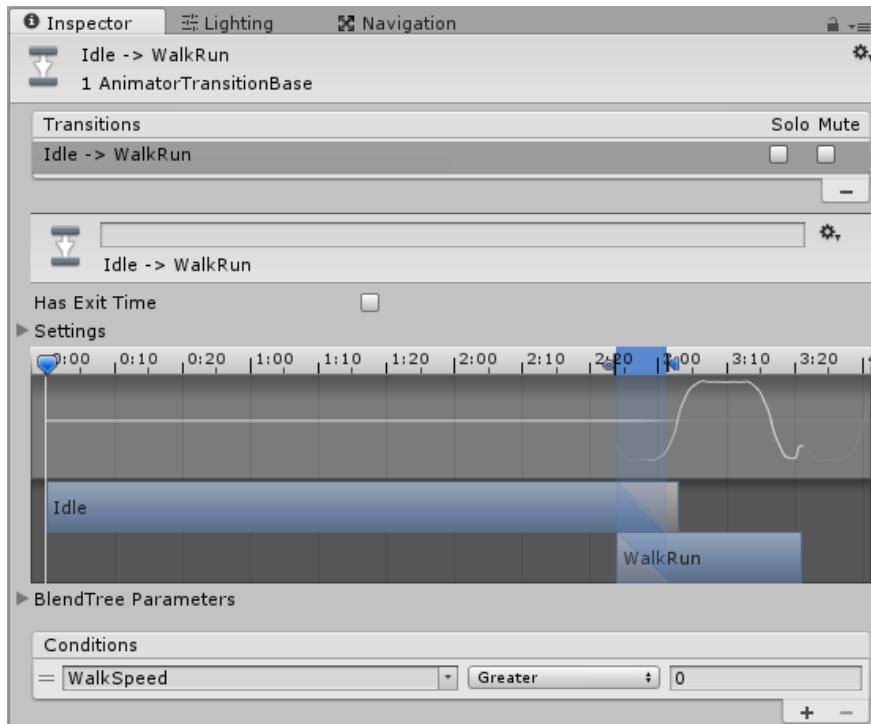


Bild 6.76 Die Übergangsbedingung vom „Idle“-State in den „WalkRun“-State

Ist keine Bedingung angegeben, wird dieser Übergang automatisch gewählt. Dies kann dann zum Beispiel zusammen mit *Has Exit Time* verwendet werden, um nach dem Abspielen eines Animation Clips automatisch in den nächsten State zu wechseln. In unserem Beispiel könnte man dies bei dem Übergang zwischen „Jump Start“ und „Jump Air“ verwenden, da nach dem Absprung automatisch die „In der Luft“-Animation abgespielt werden soll. Es ist also keine Bedingung für den Übergang notwendig.

6.8.4.1 Animation Controller erstellen

Einen neuen *Animation Controller* legst du über das Kontextmenü im *Project Browser* an. Klicke mit der rechten Maustaste in den gewünschten Ordner und wähle **CREATE/ANIMATION CONTROLLER**. Nachdem du ihn benannt hast, kannst du den neuen *Animation Controller* mit einem Doppelklick automatisch im *Animator*-Fenster öffnen.

States anlegen

Im *Animator*-Fenster kannst du mittels *Drag & Drop* die *Animation Clips*, die du verwenden möchtest, aus dem *Project Browser* in das *Animator*-Fenster ziehen. Für jeden Clip wird automatisch ein neuer State angelegt. Wenn du einen State anklickst, kannst du im *Inspector* weitere Einstellungen wie zum Beispiel die Abspielgeschwindigkeit bestimmen.

Parameter anlegen

Parameter legst du über die Menüleiste links im Animator-Fenster an. Zunächst musst du ggf. einmal im oberen Teil der Leiste von dem Reiter *Layers* zu **PARAMETERS** wechseln. Dort kannst du dann über das kleine + neue Parameter hinzufügen. Parameter können entweder Zahlentypen (*Float* für Kommazahlen, *Int* für ganzzahlige Werte) oder *boolesche* Typen sein. Bei Letzteren unterscheidet man zwischen *Bools*, die dauerhaft ein- oder ausgeschaltet werden, und *Triggern*. *Trigger* kennen auch nur die beiden Zustände *true* und *false*, das Besondere ist jedoch, dass sie automatisch auf *false* gesetzt werden, sobald sie für eine Übergangsbedingung ausgewertet wurden. Sie können also dafür genutzt werden, bestimmte Animationen nur einmalig und nicht dauerhaft auszulösen.

Übergänge anlegen

Übergänge erstellst du über das Kontextmenü: Klicke mit der rechten Maustaste auf den State, von dem der Übergang ausgehen soll, und wähle **MAKE TRANSITION**. Anschließend musst du auf den State klicken, der das Ziel des Übergangs sein soll. Den neuen Übergang kannst du jetzt anklicken und im *Inspector*, wie im vorherigen Kapitel besprochen, konfigurieren.

Fehler finden

Wenn du dein Spiel im Editor testest, während das Animator-Fenster offen ist, kannst du in der *Hierarchy* ein *GameObject* auswählen, das ein *Animator*-Component besitzt. Im Animator-Fenster wird dann der dazugehörige *Animation Controller* mit einer Live-Ansicht angezeigt. Das bedeutet, du siehst den aktuellen Zustand und auch die aktuelle Parameter-Konfiguration des jeweiligen *GameObjects*.

6.8.4.2 Animator via Script steuern

Wenn du ein Modell, das über ein *Rig* verfügt, in deine Scene ziehst, wird zu dem erzeugten *GameObject* automatisch ein *Animator*-Component hinzugefügt. Diesem Component musst du dann zunächst den *Animation Controller* zuweisen, den es verwenden soll. Danach ist das Modell bereit, von dir über ein Script gesteuert zu werden.

Dazu benötigst du zunächst einen Verweis auf das *Animator*-Component, das du steuern möchtest. Danach kannst du über die Methoden *SetFloat*, *SetInteger*, *SetBool* und *SetTrigger* die Parameter des *Animation Controllers* verändern und so Einfluss auf den aktuellen Zustand des *Animators* nehmen. Listing 6.13 enthält für jede Methode einen Beispiel-Aufruf.

Listing 6.13 Beispiel zur Steuerung der Parameter eines Animators

```
private Animator animator;
void Start(){
    animator = GetComponent<Animator>();
    animator.SetFloat("FloatParamName", 5.2f);
    animator.SetInteger("IntParamName", 2);
    animator.SetBool("BoolParamName", true);
    animator.SetTrigger("TriggerName"); // Aktiviert Trigger
    //animator.ResetTrigger("TriggerName") // Setzt Trigger zurück, wenn aktiv
}
```

■ 6.9 Wegfindung



Bild 6.77 Die Wegfindung dient dazu, den idealen Weg zu einem bestimmten Ziel zu finden.

Wenn sich Figuren, die durch ein Script gesteuert werden, möglichst intelligent durch das Level bewegen sollen, wird ein Wegfindungssystem benötigt. Das Script sagt dem Wegfindungssystem, wo die Figur sich derzeit befindet und wo sie hin möchte. Das Wegfindungssystem berechnet mit diesen Informationen automatisch den kürzesten, schnellsten oder optimalen Weg.

Bei einem umfangreichen Wegfindungssystem, wie es in Unity vorhanden ist, gibt es nämlich auch die Möglichkeit, bestimmte Wege oder Bereiche als ineffizienter zu markieren als andere. Einer gepflasterten Straße würde man zum Beispiel eine deutlich bessere Effizienz zuordnen als einem bewachsenen Feld. Die Wegfindung würde dann stets die Länge eines möglichen Umweges mit den jeweiligen Effizienzwerten verrechnen und so den bestmöglichen Weg finden. Ziel dieser Variante ist es, dass die Wege möglichst natürlich auf den Spieler wirken. Läuft die Figur einen riesigen Umweg, nur weil sie sonst ein kurzes Stück durch ein Feld laufen müsste, würde es nicht plausibel auf den Spieler wirken. Genauso würde es merkwürdig für den Spieler sein, wenn die Figur die ganze Zeit quer durch das Feld läuft, obwohl direkt daneben ein Weg wäre. Über die Effizienzwerte hilfst du der Wegfindung also, möglichst logische Wege zu finden.

Wir fangen aber zunächst mit den Grundlagen der Wegfindung in Unity an.

6.9.1 Wegfindungssystem einrichten

Wir schauen uns an dieser Stelle Unitys neues *component-basiertes Wegfindungssystem* an. Zum Zeitpunkt der Veröffentlichung dieses Buches ist das System noch nicht fest in Unity integriert, sondern wird von den Unity-Entwicklern als externer Download angeboten. In

Zukunft wird sich das wahrscheinlich ändern, weshalb ich dir dieses neue System anstelle des alten vorstellen werde. Wenn du diesem Buch also mit einer aktuelleren Unity-Version als 2017.1.0f3 folgst, solltest du zunächst einmal prüfen, ob das neue System mittlerweile in Unity integriert ist:

Dazu musst du einfach prüfen, ob in deiner Unity-Version das Component *NavMesh Surface* bereits vorhanden ist. Am einfachsten geht das, indem du bei einem beliebigen GameObject im *Inspector* auf **Add COMPONENT** klickst. Dort müsste es, wie in Bild 6.78 rechts, in der Kategorie *Navigation* angezeigt werden. Solltest du das *Component*, wie auf der linken Seite des Bildes, nicht sehen, musst du den Anweisungen im Kasten unten folgen. Ist das Component bereits vorhanden, kannst du den Kasten überspringen.

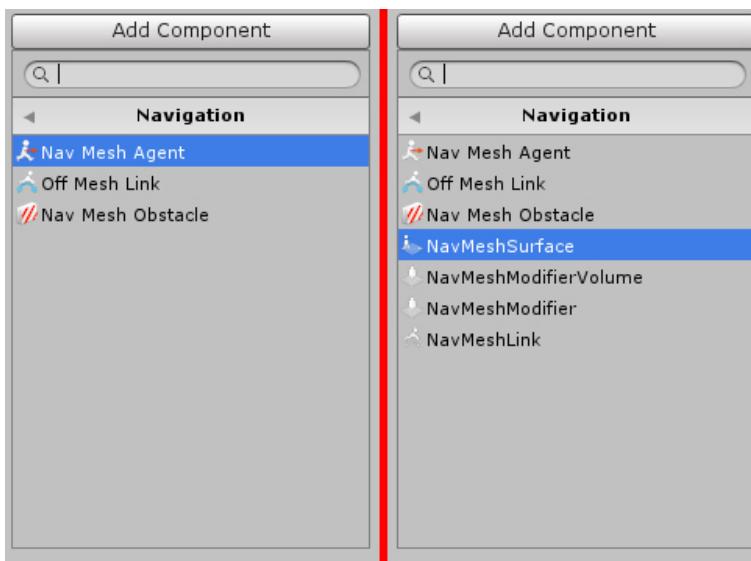


Bild 6.78 Links: Das neue Navigationssystem fehlt.
Rechts: Das neue Navigationssystem ist bereits vorhanden.

 **Unitys neues Navigationssystem importieren**

<http://www.vrspieleentwickeln.de/zusatz/>

Auf der begleitenden Webseite findest du unter **Sonstiges** ein vorbereitetes Unity Package für die *NavMeshComponents*.

Diese Version kannst du einfach herunterladen und ohne Änderungen in dein existierendes Projekt importieren.

Von GitHub herunterladen:

Alternativ kannst du dir auch von Unitys GitHub-Seite die aktuellste Version herunterladen. Dies ist jedoch ein komplettes Beispielprojekt (!). Aus diesem Projekt musst du dir die Ordner *Assets/NavMeshComponents*, *Assets/Gizmos* und *Assets/Examples* herauskopieren und in dein Projekt importieren. Importierst du das gesamte Projekt, werden zum Beispiel deine *Project Settings* überschrieben, was du wahrscheinlich nicht möchtest.

<https://github.com/Unity-Technologies/NavMeshComponents>

6.9.2 Grundlagen

Unitys Wegfindingssystem verwendet ein sogenanntes *NavMesh* (kurz für „Navigation Mesh“), um zu markieren, welche Bereiche der Scene laufbar sind und welche nicht. Eine Spielfigur, die sich auf einem *NavMesh* bewegt und dieses zur Wegfindung nutzt, nennt man *NavMeshAgent*. In Bild 6.79 kannst du das blaue *NavMesh* sehen, das für den gelben *NavMeshAgent* erzeugt wurde. Auf der gesamten blauen Fläche könnte dieser laufen, ohne dass er mit einer Wand kollidiert.

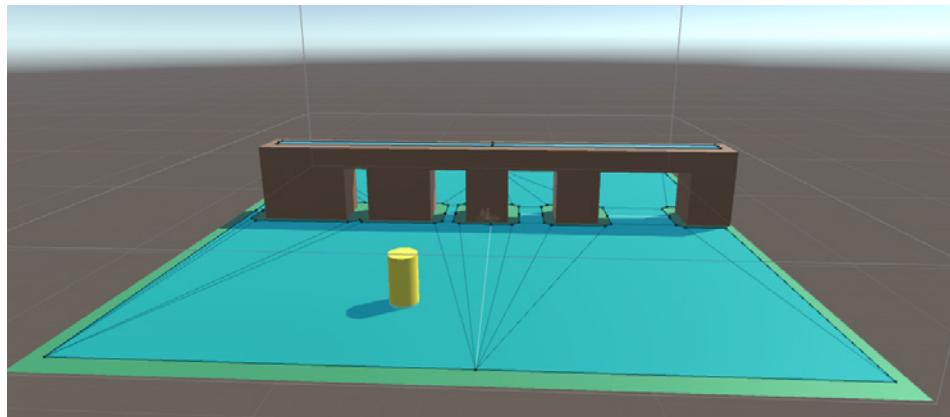


Bild 6.79 Der gelbe Zylinder ist ein *NavMeshAgent*, die blaue Fläche ist das *NavMesh*.

Alle *NavMeshAgent* wird ein sogenannter *Agent Type* zugewiesen, wie du ihn in Bild 6.80 sehen kannst. Der *Agent Type* legt fest, welche Eigenschaften die jeweilige Figur besitzt. Dazu gehört unter anderem, wie breit und hoch sie ist oder wie hoch Stufen und wie steil Steigungen maximal sein dürfen, damit dieser *Agent* sie noch erklimmen kann. Nicht jede Spielfigur bekommt dabei ihren eigenen individuellen *Agent Type*: In der Regel unterteilt man die Figuren in grobe Gruppen, deren Eigenschaften ungefähr ähnlich sind.

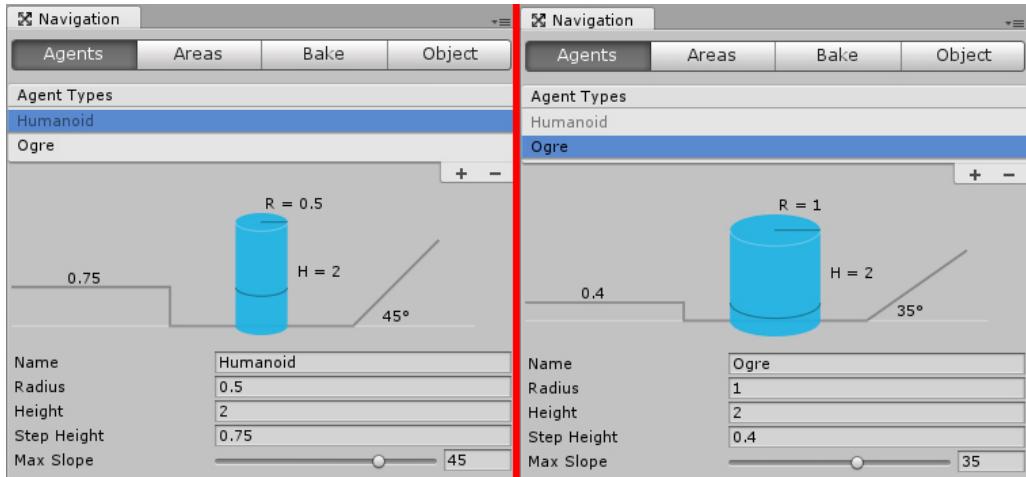


Bild 6.80 Die Konfiguration eines Agents für menschenähnliche Figuren links und für breitere Orks rechts

Für jeden *Agent Type* muss ein eigenes *NavMesh* berechnet werden, welches jeweils nur für Spielfiguren mit dem entsprechenden Typ gültig ist. Für die beiden *Agent Types* aus Bild 6.80 würden die beiden *NavMeshs* zum Beispiel aussehen wie in Bild 6.81.

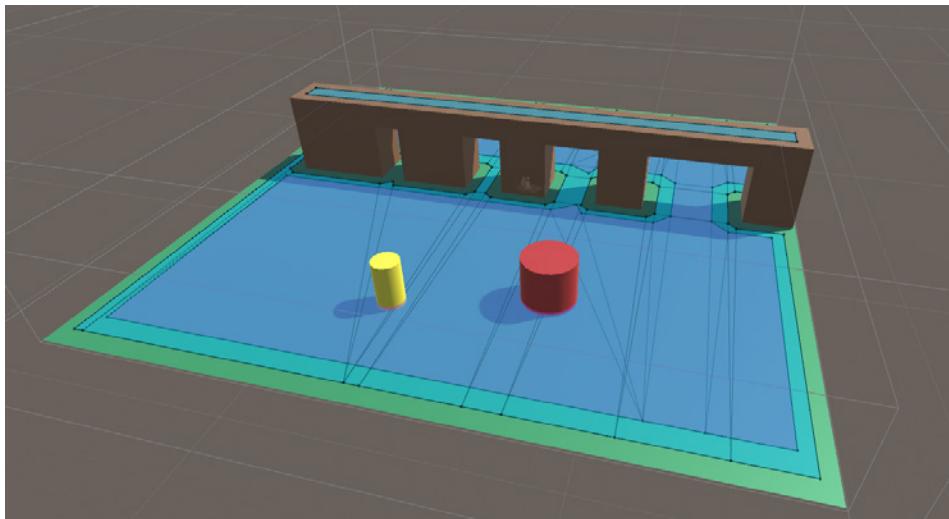


Bild 6.81 Hellblau ist das *NavMesh* für den „Humanoid“-Agent-Type und dunkelblau das *NavMesh* für den breiteren „Ogre“-Type.

Wenn ein *NavMesh* berechnet wurde, verbindet es zunächst ausschließlich Bereiche deiner Scene, die direkt miteinander verbunden sind. In Bild 6.81 könnten *Agents* zwar theoretisch oben auf dem Torbogen laufen, es gibt jedoch keine Möglichkeit, dort hoch zu gelangen, wenn sie sich auf der Bodenplatte befinden. Das *Navigation System* bietet jedoch die Möglichkeit, zwei nicht zusammenhängende *NavMesh*-Teile miteinander zu verknüpfen und die

Spielfigur von einem Teil zum anderen springen zu lassen. Dafür verwendet man sogenannte *NavMeshLinks*. Bild 6.82 zeigt als Beispiel, wie vier getrennte Bereiche des *NavMeshs* mit *NavMeshLinks* verbunden wurden.

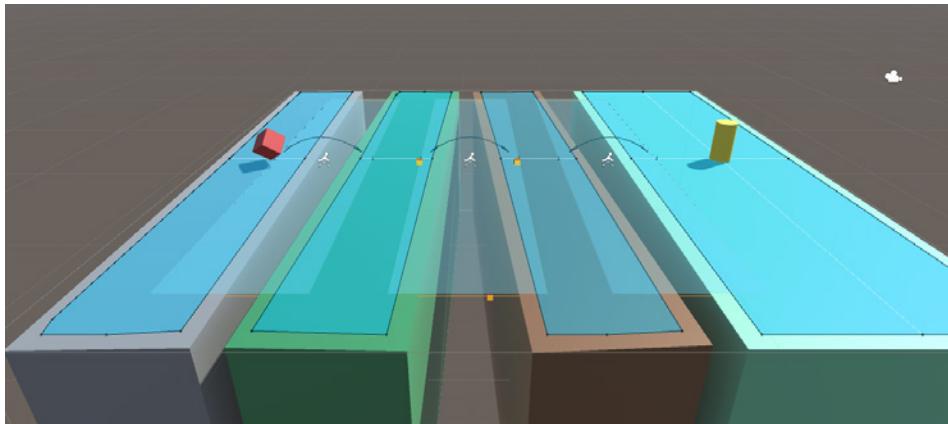


Bild 6.82 Die einzelnen NavMesh-Teile sind über NavMesh-Links miteinander verbunden, sodass ein Agent über die Lücken hinwegspringen kann.

Wenn der gelbe Agent jetzt zu dem roten Würfel laufen möchte, kann er die Verknüpfungen nutzen, um die Lücken zu überspringen. Dabei führt er allerdings nicht automatisch einen parabelförmigen Sprung oder Ähnliches aus, sondern bewegt sich über den direkten Weg von einem *NavMesh*-Teil zum nächsten. Optional kannst du aber auch ein eigenes Verhalten programmieren und zum Beispiel eine Sprunganimation abspielen.

6.9.3 NavMeshs berechnen

Wenn du ein *NavMesh* berechnen möchtest, machst du das über das *NavMeshSurface*-Component. Erstelle am besten ein neues *GameObject* und füge das Component über das **ADD COMPONENT**-Menü im *Inspector* hinzu. In dem Component stehen dir unter anderem folgende Optionen zur Verfügung:

- **Agent Type:** Bestimmt, für welchen *Agent Type* dieses *NavMesh* berechnet werden soll. Du benötigst für jeden *Agent Type*, den du verwenden möchtest, ein *eigenes Nav Mesh Surface*-Component.
- **Collect Objects:** Hier kannst du bestimmen, ob das *NavMesh*, das durch dieses Component erzeugt wird, alle *GameObjects* in der Scene berücksichtigen soll oder nur *GameObjects* in einem bestimmten Bereich. Alternativ kannst du auch definieren, dass nur Kinder des jeweiligen *GameObjects* berücksichtigt werden.
- **Include Layers:** Hierüber kannst du *GameObjects*, die bestimmten *Layers* zugeordnet sind, von dem *NavMesh* ausschließen.
- **Use Geometry:** Soll das *NavMesh* Kollisionen mit den *Collidern* oder mit den dargestellten 3D-Modellen vermeiden?

- **Advanced/Default Area:** Hier kannst du eine Area angeben, die für dieses *NavMesh* verwendet werden soll. Wenn du später sehr komplexe Scenes hast, kannst du hierüber Bereiche markieren, die zum Beispiel weniger effizient sind.

Du kannst zu einem GameObject problemlos mehrere *NavMeshSurface*-Components hinzufügen, beispielsweise eines für jeden *Agent Type*.

6.9.4 NavMeshs verbinden

Wie schon erwähnt, kannst du mit sogenannten *NavMeshLinks* mehrere Bereiche deines *NavMeshs* über „Sprünge“ miteinander verknüpfen. Erzeuge dazu einfach ein neues leeres GameObject und füge über das **ADD COMPONENT**-Menü aus der Kategorie „Navigation“ das *Nav Mesh Link*-Component hinzu. Anschließend kannst du das GameObject und damit auch die Verknüpfung so positionieren, dass es sich zwischen den beiden Teilen, die du verbinden möchtest, befindet. In der *Scene View* muss du anschließend die beiden orangen Würfel so verschieben, dass sie sich jeweils in einem Teil deines *NavMeshs* befinden. Dies sind der Start- und Endpunkt deiner Verknüpfung. Zusätzlich ist es noch möglich, die Breite der Verknüpfung anzupassen (*Width*).

Bild 6.83 zeigt dir beispielhaft, wie eine konfigurierte Verknüpfung aussehen kann.

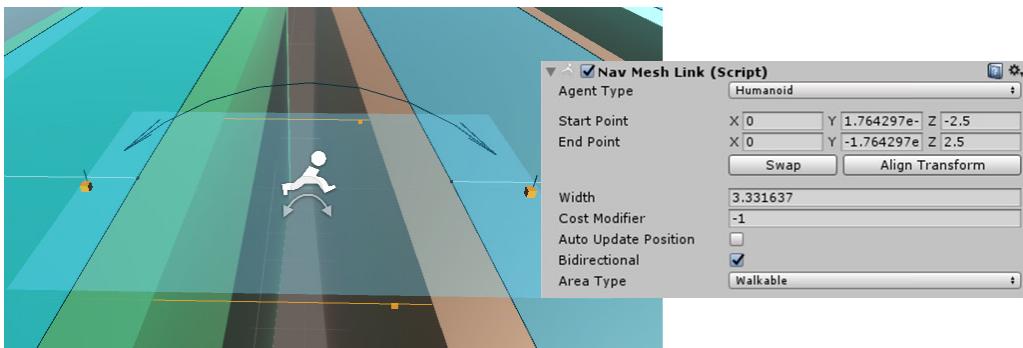


Bild 6.83 Beispiel für einen Nav Mesh Link über einen Abgrund

6.9.5 Dynamische Hindernisse

Im Normalfall aktualisiert sich ein *NavMesh* nicht automatisch, sondern ist nach dem Erstellen zunächst statisch. Über das *NavMeshObstacle*-Component kannst du bewegliche Hindernisse in das *NavMesh* einfügen, welche von allen *Agents* als unüberwindbares Hindernis interpretiert werden.

Neben der Form des Hindernisses kannst du über die *Carve*-Option einstellen, dass die Objekte tatsächlich ein Loch in das *NavMesh* schneiden. Das sorgt dafür, dass *Agents* die Hindernisse zuverlässig umgehen und nicht nur vermeiden. Diese Option kostet jedoch viel Rechenpower, weshalb es sinnvoll ist, die Einstellung „Carve Only Stationary“ zu aktivieren. Diese sorgt nämlich dafür, dass nur ein Loch in das *NavMesh* geschnitten wird, wenn sich

das *GameObject* gerade nicht bewegt. Du solltest die Anzahl der dynamischen Hindernisse trotzdem so gering wie möglich halten.

6.9.6 NavMeshAgent erstellen

Ein *NavMeshAgent* ist, wie in der Einleitung schon erwähnt, ein *GameObject* das sich auf dem *NavMesh* bewegen soll. Dafür musst du einfach zu dem jeweiligen *GameObject* ein *Nav Mesh Agent*-Component hinzufügen. Zu dem *GameObject* wird dann automatisch ein zylinderförmiger Collider hinzugefügt, der allerdings ausschließlich für die Navigation auf dem *NavMesh* verwendet wird. In dem Component stehen dir folgende Optionen zur Verfügung, um das Laufverhalten deines *GameObjects* anzupassen.

- **Agent Type:** Bestimmt nicht nur Höhe und Breite dieses Agents, sondern auch, welches *NavMesh* er verwenden soll.
- **Base Offset:** Hierüber kannst du den *NavMesh*-Collider vertikal verschieben. Ob er richtig positioniert ist, kannst du in der *Scene View* erkennen.
- **Steering:** Hier kannst du die maximale Lauf- (*Speed*) und Drehgeschwindigkeit (*Angular Speed*) sowie Beschleunigung (*Acceleration*) bestimmen. Außerdem kannst du mittels der *Stopping Distance* angeben, wie nah dieser Agent mindestens an den Zielort heranlaufen soll. Mit der Option *Auto Breaking* legst du fest, dass das *GameObject* frühzeitig aufhört zu beschleunigen, sodass es nicht über das Ziel hinausschießt.
- **Obstacle Avoidance:** Hier kannst du festlegen, wie sich dieser Agent gegenüber *NavMesh Obstacles* verhalten soll. Mittels *Radius* und *Height* legst du eine weitere *Collider*-Größe fest, die nur für das Vermeiden dieser dynamischen Hindernisse verwendet wird. Die Qualität bestimmt, wie viel Rechenleistung für das Vermeiden dieser Hindernisse aufgewendet werden soll. Dynamische Hindernisse können sehr schnell viel Performance kosten.
- **Path Finding:** Hier kannst du mittels *Auto Traverse Off Mesh Link* bestimmen, ob dieser Agent automatisch *Nav Mesh Links* folgen soll. Wenn du z.B. eine Sprunganimation abspielen möchtest, solltest du diese Option deaktivieren. Ist *Auto Repath* aktiviert, prüft dieser Agent, während er den Pfad abläuft, ob der Pfad noch gültig ist. Mittels der Option *Area Mask* kannst du festlegen, auf welchen *Areas* dieser Agent laufen darf. Diese Option ist nur relevant, wenn du auch bei dem Erzeugen deiner *NavMeshes* unterschiedliche *Default Areas* angegeben hast.

6.9.7 NavMeshAgent über Script steuern

Das *NavMeshAgent*-Component kannst du über ein Script steuern. Du brauchst dafür lediglich einen Verweis auf das jeweilige *NavMeshAgent*-Component und kannst über die *SetDestination*-Methode einen neuen Zielort festlegen. Die Methode gibt einen *Boolean* zurück, ob ein gültiger Pfad gefunden wurde oder nicht. Die Berechnung eines Pfades kann einige Frames dauern. Um doppelte Anfragen an das *Navigation System* zu verhindern, kannst du über die Variable *pathPending* herausfinden, ob derzeit bereits ein Pfad für die-

sen Agent berechnet wird. Die Variable `destination` enthält das aktuelle Ziel. Alternativ zu `SetDestination` kannst du auch den Wert dieser Variablen ändern, um eine neue Berechnung des Pfades auszulösen. Dennoch erhältst du in diesem Fall kein Feedback darüber, ob ein passender Pfad gefunden wurde oder nicht.

Listing 6.14 zeigt dir ein Beispiel-Script, das dem dazugehörigen `NavMeshAgent` sagt, er soll immer zu der Position des angegebenen Transforms laufen.

Listing 6.14 Beispiel-Script, um einen `NavMeshAgent` laufen zu lassen

```
using UnityEngine;
using UnityEngine.AI;
[RequireComponent(typeof(NavMeshAgent))]
public class NavAgentFollowTransform : MonoBehaviour
{
    private NavMeshAgent agent;
    private Transform target;

    void Start()
    {
        agent = GetComponent<NavMeshAgent>();
    }
    void Update()
    {
        //Wenn nicht gerade ein Pfad ausgerechnet wird und das aktuelle Ziel nicht
        //der Position des zu folgenden Transforms entspricht, setze Position
        if(!agent.pathPending && agent.destination != target.position)
        {
            agent.SetDestination(target.position);
        }
    }
}
```

Wenn du einen `NavMeshAgent` zusammen mit Animationen verwendest, solltest du entweder im `Animator` die Option *Apply Root Motion* deaktivieren, damit das GameObject alleinig durch den `NavMeshAgent` bewegt wird. Alternativ kannst du auch die Speed-Variablen im `NavMeshAgent` auf 0 setzen und deinen Animator basierend auf den vom `NavMeshAgent` berechneten Pfad steuern. Mischformen sind natürlich auch denkbar.

6.9.8 Erweiterte Beispiele

Wir haben uns in diesem Abschnitt die einfachen, am meisten verwendeten Funktionen des Navigationssystems und seiner Components angeschaut. Es ist aber noch viel möglich: zum Beispiel das Berechnen von `NavMeshs` zur Laufzeit und das Kombinieren mehrerer `NavMeshs`, um komplexe Situationen darzustellen. Wenn du dich noch näher mit den `NavMeshs` beschäftigen möchtest, schau dir am besten als Nächstes die Beispiele im Ordner `NavMeshComponents/Examples` des NavMesh-Unity Packages an. Wenn du das NavMesh-Projekt von GitHub verwendetest, findest du die Beispiele in dem Ordner „Examples“.

■ 6.10 Savegames – Daten speichern und laden

Speicherstände sind ein elementarer Bestandteil von Videospielen. Egal ist dabei, ob es ein einfacher Highscore oder komplexer Spielstand mit Spielfortschritt und alle getroffenen Entscheidungen sind, in Unity sind dafür die sogenannten *PlayerPrefs* (kurz für *Player Preferences*) vorgesehen. Anders als man es erwarten könnte, existiert keine universale Speicherfunktion, die den aktuellen Stand des Spiels automatisch abspeichert. Eine solche Funktion musst du für dein Spiel selber programmieren. Dabei musst du selber bestimmen, was genau gespeichert und geladen werden soll.

In den *PlayerPrefs* kannst du keine komplexen Objekte speichern, sondern ausschließlich einzelne *Strings*, *Bools*, *Integers* und *FLOATs*. Deinen Spielstand musst du also aus diesen einzelnen Datentypen zusammensetzen.

Das Speichern von Informationen funktioniert über Schlüsselwert-Paare, die mit folgenden Funktionen gesetzt werden können:

- **SetString (name : string, value : string) : void**
- **SetInt (name : string, value : int) : void**
- **SetFloat (name : string, value : float) : void**

Alle Änderungen sind allerdings zunächst nur temporär, das bedeutet, nach einem Neustart des Spiels sind die Werte alle zurückgesetzt. Erst mit einem Aufruf auf der Speicher-Methode werden alle gesetzten Werte auch auf die Festplatte geschrieben.

- **Save () : void**

Listing 6.15 zeigt dir Beispiele für die Verwendung der Methoden.

Listing 6.15 Speichern von Informationen

```
public void SaveGame(string playerName, int currentLevel, float highscore){  
    PlayerPrefs.SetString("Name", playerName);  
    PlayerPrefs.SetInt("Level", currentLevel);  
    PlayerPrefs.SetFloat("Highscore", highscore);  
    PlayerPrefs.Save(); // Speichert die Änderungen auf die Festplatte  
}
```

Die auf der Festplatte gespeicherten *PlayerPrefs* werden bei Spielstart automatisch in den Speicher geladen, sodass du die gespeicherten Informationen zum Beispiel in der Start-Methode auslesen kannst. Das Auslesen erfolgt über die entsprechenden Get-Methoden, welche ich dir nachfolgend vorstellen werde. Alle Get-Methoden haben einen optionalen zweiten Parameter, mit dem du einen Standardwert angeben kannst. Existiert noch kein gespeicherter Wert mit dem von dir angegebenen Namen, wird dann dieser Wert zurückgegeben. Ohne diesen Standardwert geben die Methoden in diesen Fällen einen leeren String bzw. die Zahl 0 zurück.

- **GetString(name : string [, defaultValue : string]) : string**
- **GetInt(name : string [, defaultValue : int]) : int**
- **GetFloat(name : string [, defaultValue : float]) : float**

Listing 6.16 zeigt dir beispielhaft, wie du die Methode zum Laden eines einfachen Spielstandes in der Start-Methode verwenden kannst.

Listing 6.16 Laden von Informationen

```
string playerName;
int currentLevel;
float highscore;
void Start(){
    playerName= PlayerPrefs.GetString("Name");
    if(playerName == "") {
        Debug.LogWarning("Noch kein Name gespeichert!");
    }
    int currentLevel = PlayerPrefs.GetInt("Level", 0);
    float highscore = PlayerPrefs.GetFloat("Highscore ", 123546);

    //Lade die Scene etc.
}
```

■ 6.11 Performance optimieren

Mit Virtual Reality bewegen wir uns im High-End-Bereich der Hardware. Sowohl Computer als auch Smartphones benötigen einiges an Leistung, um die virtuellen Welten ohne Verzögerung darstellen zu können. Aus diesem Grund ist es wichtig, dass du die Performance deines Spiels im Auge behältst. Im Endeffekt soll dein Spiel ja nicht nur auf deiner Hardware laufen, sondern auf möglichst vielen Geräten.

In diesem Abschnitt lernst du, wie du die *Performance* deiner Anwendung beobachten kannst und was du diesbezüglich beim Entwickeln stetig im Blick haben solltest.

6.11.1 Batching

Wir beginnen mit ein wenig Hintergrundwissen zum *Rendering*, also der Berechnung eines einzelnen Frames:

Ein einzelner *Frame* wird nicht in einem Rutsch berechnet, sondern die Berechnung erfolgt in mehreren Schritten, welche sich *Zeichnungsvorgänge* oder auch *SetPass Calls* nennen. In einem Zeichnungsvorgang wird ohne Optimierung nur ein Material für ein Mesh und eine anstrahlende Echtzeit-Lichtquelle gezeichnet.

Die Ressourcen werden anhand dieser drei Eigenschaften gruppiert und in einem sogenannten *Batch* zusammengefasst, bevor sie von der CPU an den Grafikprozessor (GPU) übergeben werden. Je mehr unterschiedliche Meshes, Materials und Echtzeit-Lichtquellen in dem Frame vorhanden sind, desto mehr Batches müssen erzeugt werden. Pro Batch wird immer mindestens ein Zeichnungsvorgang benötigt.

Je nach verwendetem *Shader* kann es aber auch sein, dass pro *Batch* wiederum mehrere Zeichnungsvorgänge benötigt werden. Grob lässt sich sagen, pro *Texture*, die im *Material* angegeben werden kann, wird ein weiterer Zeichnungsvorgang benötigt. Auch die Color-Eigenschaft mancher *Shader* benötigt einen eigenen Zeichnungsvorgang.

Du merkst schon, die *Zeichnungsvorgänge* erhöhen sich schnell und deswegen ist es wichtig, sie im Auge zu behalten.

Unity bietet dir automatisierte Verfahren an, um die *Zeichnungsvorgänge* zu reduzieren. Dazu gehören das *Dynamic Batching* und das *Static Batching*. Bei diesen Methoden fasst Unity automatisch Objekte zusammen, die ähnliche Eigenschaften haben, und reduziert so die Anzahl der *Zeichnungsvorgänge*.

- **Static Batching** ist in der Lage, mehrere GameObjects, die dasselbe *Material* verwenden, in einen *Batch* zusammenzufassen. Wie der Name es schon vermuten lässt, müssen die GameObjects dafür jedoch im *Inspector* als *Static* markiert sein. Dieses Verfahren wird automatisch für geeignete GameObjects angewendet, deswegen ist alles, worauf du aktiv achten musst, dass du auch alle unbeweglichen Objekte als *Static* markierst. Außerdem ist es sinnvoll, darauf zu achten, dass möglichst viele unterschiedliche Modelle dasselbe *Material* verwenden.⁵
- **Dynamic Batching** kann auch mehrere nichtstatische GameObjects in einen Batch zusammenfassen. Dafür müssen die Modelle jedoch noch strengere Regeln einhalten: Die GameObjects müssen nicht nur dasselbe *Material* verwenden, sondern müssen unter anderem auch die exakt gleiche Skalierung haben. Dazu kommen noch weitere Kriterien, sodass die Ersparnis durch *Dynamic Batching* in den meisten Fällen eher gering ausfällt. Du solltest dich deshalb nicht darauf verlassen, sondern dich bemühen, die Kriterien für *Static Batching* so häufig wie möglich zu erfüllen.

6.11.2 Performance im Auge behalten

Wenn die Anwendung nicht gerade so langsam wie eine Diashow läuft, fällt eine schlechte Performance mit dem bloßen Auge nicht direkt auf. Genauso ist es nicht einfach, kleine Performance-Unterschiede sofort zu bemerken, wenn du zum Beispiel versuchst, deine Performance zu verbessern. Denn häufig gibt es nicht nur ein Problem, das du beheben kannst, und plötzlich läuft alles flüssig, in der Regel sind es viele Kleinigkeiten, die immer nur kleine Verbesserungen geben. Ohne Hilfsmittel wirst du also nur schwer erkennen können, ob deine Optimierungsversuche die Situation nun verbessert oder verschlechtert haben. Aus diesem Grund stelle ich dir in diesem Abschnitt zwei wichtige Hilfsmittel vor: die *Stats-Anzeige* und den *Unity Profiler*.

⁵ Dazu können zum Beispiel sogenannte *Atlas Maps* erstellt werden, die mehrere Textures in einer einzigen Texture zusammenfassen.

6.11.2.1 Die Stats-Anzeige



Bild 6.84 Die Stats-Anzeige gibt dir eine Übersicht über die aktuelle Performance.

Die *Stats-Anzeige* gibt dir eine allgemeine Übersicht über die aktuelle Performance deiner Anwendung, wenn du sie im Editor testest. Um dieses Fenster zu öffnen, musst du oben rechts in der *Game View* auf die Schaltfläche **STATS** klicken.

Die Stats-Anzeige, welche du auch in Bild 6.84 sehen kannst, gibt folgende Auskünfte:

- **FPS & Frame-Time:** Rechts neben der Überschrift *Graphics* findest du die *FPS*-Anzeige. Sie zeigt dir an, mit wie vielen „Frames per Second“ (dt. „Bildern pro Sekunde“) die *Game View* gerade berechnet wird. Die Zeitangabe dahinter gibt an, wie viele Millisekunden für die Berechnung eines Frames derzeit benötigt werden. Die beiden Werte hängen direkt zusammen: Dauert die Berechnung eines einzelnen Frames länger, können nur wenige Bilder pro Sekunde berechnet werden.
- **Batches:** Gibt an, wie viele *Batches* an den Grafikprozessor übergeben wurden. Mehr Batches bedeuten schlechtere Performance.
- **Saved by Batching:** Hier siehst du, wie viele *Batches* durch das *Batching* eingespart wurden. Der Wert fasst Ersparnisse durch *Static Batching* und *Dynamic Batching* zusammen.
- **Tris/Verts:** Hier siehst du, wie viele *Triangles* und *Vertices* für den aktuellen Frame gerendert wurden. Komplexere Modelle besitzen mehr *Triangles* und kosten deshalb mehr Performance.
- **SetPass calls:** Dieser Wert gibt an, wie viele Zeichenvorgänge für den aktuellen Frame benötigt wurden. Komplizierte *Shader* benötigen mehr Durchläufe als einfache und brauchen dadurch länger, um gezeichnet zu werden.



Der FPS-Wert der Stats-Anzeige ist für Android-Spiele nur indirekt aussagekräftig

Da dein Computer sehr wahrscheinlich deutlich mehr Leistung als dein Smartphone hat, wirst du beim Testen im Unity Editor wahrscheinlich deutlich mehr FPS erreichen als auf dem Smartphone. Nur weil deine Scene im Editor mit 739 Bildern pro Sekunde läuft, wird sie also nicht die gleiche Performance auf dem Smartphone erreichen. Siehst du im Unity Editor durch eine Optimierung aber eine verbesserte FPS-Zahl, wird sich die FPS auch auf dem Smartphone verbessert haben, nur entsprechend weniger stark.

6.11.2.2 Der Unity Profiler

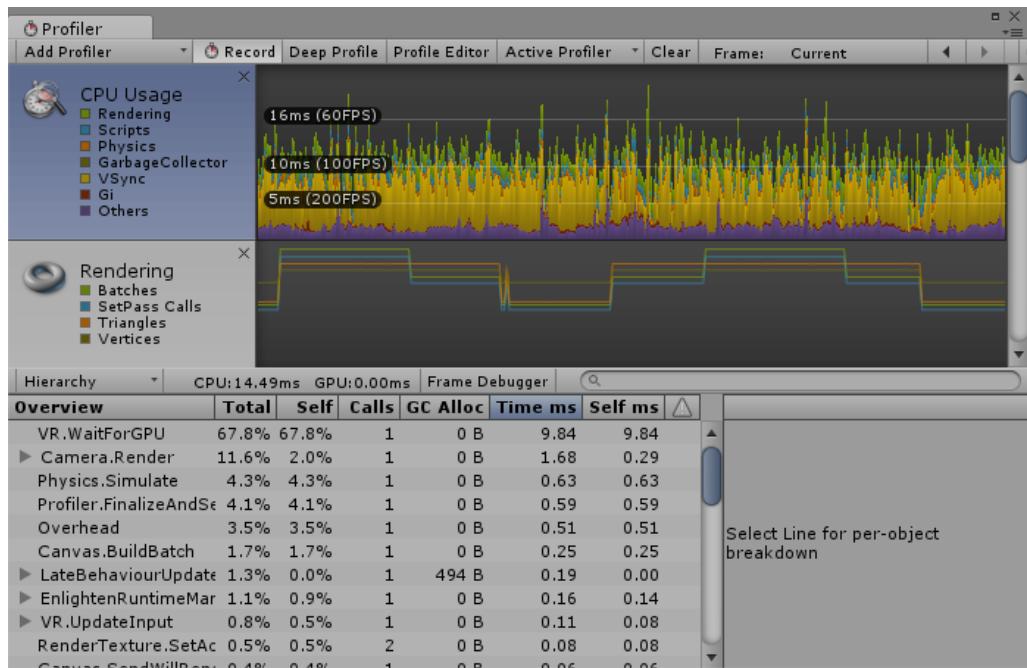


Bild 6.85 Der Profiler kann dir sowohl die Performance deines Spiels im Editor als auch die Performance auf deinem Smartphone anzeigen.

Der Unity *Profiler* bietet dir einen noch genaueren Einblick in die *Performance* deines Spiels, wie du in Bild 6.85 sehen kannst. Neben der Performance, die das *Rendering*⁶ kostet, zeigt er dir auch an, wie performant die *Scripts* deines Spiels laufen. Für eine bessere Übersicht über die Performance zeigt der Profiler nicht nur den aktuellen Wert an, sondern bietet eine Zeitleiste, welche die Performance als Graphen darstellt. Klickst du einen bestimmten Punkt des Graphen an, erhältst du weitere Informationen zu dem ausgewählten Frame. Ein

⁶ Das sind die Berechnungen des Grafikprozessors zur Darstellung des Bildes.

großer Vorteil des Profilers gegenüber der *Stats-Anzeige*: Er kann sich mit deinem Smartphone verbinden und so die *Performance des Spiels auf deinem Smartphone anzeigen*. Das ist besonders für GearVR- und GoogleVR-Spiele interessant.

Du findest das *Profiler*-Fenster in der Toolbar unter **WINDOW/PROFILER**.

Der *Profiler* bietet dir diverse unterschiedliche Statistiken. Das **CPU**-Diagramm zeigt dir, wie lange der Prozessor benötigt, um die letzten Frames zu berechnen, und demnach mit wie viel FPS das Spiel läuft. Weil der Prozessor auf den Grafikprozessor („GPU“) warten muss, beinhaltet die hier angegebene Zeit sowohl die Zeit, die während des Ausführens aller *Scripts* verstrichen ist, als auch die die für das *Rendering* benötigt wurde. Anhand der Farben und der Legende, die du links im Profiler findest, kannst du erkennen, welche Berechnungen wie viel Zeit benötigt haben. Der *Profiler* liefert dir so wichtige Informationen darüber, welche Berechnungen die Performance deines Spiels wie stark beeinflussen und demnach was du optimieren solltest.

Du kannst die einzelnen Diagramme des *Profilers* anklicken, um dann mehr Details zu dem jeweiligen Bereich zu erhalten. Wenn du zum Beispiel **CPU USAGE** auswählst, siehst du in der *Overview* (dt. „Übersicht“) darunter eine Auflistung aller aktiven Components und wie viel Berechnungszeit sie benötigen.

Um mehr Details zum *Rendering* zu erhalten, kannst du das **RENDERING**-Diagramm im Profiler anklicken. Hier siehst du dann in der Übersicht alle Informationen, die du auch in der *Stats-Anzeige* findest, und noch ein paar andere Details. Da das *Stats*-Fenster nur beim Testen im Editor funktioniert, kannst du diese Ansicht verwenden, wenn du gerade auf deinem *Smartphone* testest.

Bei komplexeren Projekten wirst du auch anfangen müssen, den Speicherverbrauch deines Spiels im Auge zu behalten. Diesen kannst du in dem **MEMORY**-Diagramm und der dazugehörigen Übersicht beobachten.



Steht im Profiler „WaitForGPU“ weit oben, ist die Performance deines Spiels derzeit in erster Linie durch das *Rendering* (SetPass calls, Batches, Vertices etc.) begrenzt und nicht durch *CPU-intensive Aufgaben* (Scripte, Physik etc.).

6.11.2.2.1 Unity Profiler mit Smartphone verbinden (GearVR, GoogleVR)

Der *Profiler* hat zwei Möglichkeiten, sich mit deinem Smartphone zu verbinden: zum einen über das *USB-Kabel* und zum anderen über *WLAN*. Da du dein Smartphone in das VR-Headset einlegen musst, empfiehlt es sich, die Variante über WLAN zu verwenden.

Stell als Erstes sicher, dass sich dein *Computer* und dein *Smartphone* im gleichen Netzwerk befinden, also zum Beispiel mit gleichen WLAN verbunden sind. Als Nächstes musst du das *Profiler*-Fenster und anschließend die *Build Settings* öffnen (**FILE/BUILD SETTINGS...**). Hier musst du jetzt darauf achten, dass die Option *Development Build AKTIVIERT* und die Option *Autoconnect Profiler DEAKTIVIERT* ist.

Jetzt kannst du die App mittels **BUILD AND RUN** auf deinem Smartphone installieren und starten.

Lässt du die Option *Autoconnect Profiler* aktiv, würde der *Profiler* sich automatisch via USB mit deinem Gerät verbinden, was bei der Verbindung über WLAN Probleme machen kann.

Sobald die App gestartet ist, kannst du im Profiler über die Schaltfläche **CONNECTED PLAYER** das Gerät auswählen, das du im Profiler analysieren möchtest. In vielen Fällen wird hier dein Smartphone bereits wie in Bild 6.86 aufgeführt und du musst es nur noch aktivieren. Der Eintrag *Android Player ADB* steht übrigens für die USB-Verbindung.

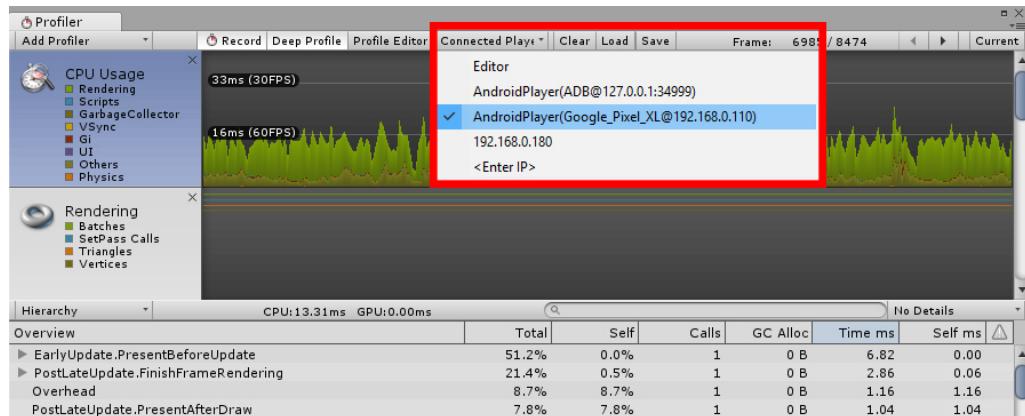


Bild 6.86 Bei einer Verbindung über das Netzwerk bist du unabhängig von der USB-Verbindung und kannst dich frei bewegen.

Sollte dein Gerät nicht, wie im Bild, in dem *Connected Player*-Menü auftauchen, obwohl die App derzeit läuft, kannst du über <ENTER IP> die lokale IP-Adresse deines Gerätes eingeben und dich so manuell verbinden. Sollte auch das nicht klappen, überprüfe deine Netzwerk- und Firewall-Einstellungen am PC bzw. auf dem Smartphone.

6.11.3 VR-Optimierungs-Checkliste und Grenzen

Wenn du eine Virtual-Reality-Anwendung programmierst, steht dir in den meisten Fällen nur halb so viel Performance zur Verfügung wie bei einer Nicht-VR-Anwendung. Das liegt daran, dass jedes Objekt für die stereoskopische Darstellung zweimal gezeichnet werden muss und dadurch jeder *Batch*, *SetPass Call* und jedes *Triangle* doppelt zählen. Deswegen ist das Thema „Performance-Optimierung“ in Virtual Reality auch mindestens doppelt so wichtig wie sonst.

Im Editor kannst du dies sehr gut beobachten, indem du die stereoskopische Darstellung über die Option **VIRTUAL REALITY SUPPORTED** in den *Player Settings* für einen Moment deaktivierst.

Bild 6.87 zeigt dir als Beispiel dafür die Stats-Anzeige des Editors mit und ohne Virtual-Reality-Darstellung. Weil die verwendete Scene sehr einfach ist, ist der Unterschied bei den Bildern pro Sekunde („FPS“) eher gering. Interessanter sind die Werte unter dem Punkt *Graphics*. Hier lässt sich nämlich ablesen, dass mit aktiviertem VR alle Werte mindestens doppelt so groß sind und dementsprechend auch doppelt so viel Performance kosten.

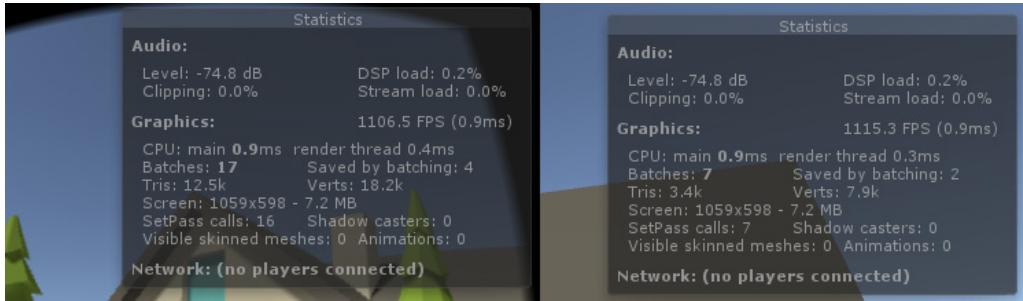


Bild 6.87 Dieselbe Scene und Kameraeinstellung, links mit aktivem „VR Mode“ und rechts ohne VR

Unity bietet dir, um dieses Problem einzuschränken, die *Single Pass*-Option an, die du in den Player Settings unter der *Virtual Reality Supported*-Einstellung aktivieren kannst. Ist Single Pass aktiviert, werden, wenn es möglich ist, die Objekte für beide Ansichten in einem einzigen Zeichnungsvorgang gerendert, sodass sich nicht alle Werte verdoppeln. Die verwendete Technik kann jedoch nicht immer verwendet werden, weshalb sich die Performance durch Single-Pass-Stereo zwar meist verbessert, aber nicht identisch zu einem Nicht-VR-Spiel wird. In manchen Fällen ist auch überhaupt kein Performance-Unterschied bemerkbar.

Wie viele *Set-Pass Calls* und *Vertices* du maximal benutzen darfst, um deine Ziel-FPS zu erreichen, kann so einfach leider nicht festgelegt werden, da dies sowohl von der Hardware als auch von deinem gesamten Spiel abhängt.

Für Smartphone-Entwicklung wird ganz allgemein empfohlen, stets unter *200 SetPass Calls* und *400.000 Vertices* pro Frame zu bleiben. Da Daydream und GearVR moderne Smartphones voraussetzen, können die Grenzen in der Regel auch problemlos überschritten werden. Aufgrund der noch größeren Vielfalt von Hardware ist es schwierig, eine vergleichbare Zahl für die PC-Entwicklung zu empfehlen. Grundsätzlich solltest du aber immer versuchen, die *SetPass Calls* und *Vertices* so gering wie möglich zu halten.

Die folgende Liste gibt dir eine Übersicht, auf was du zugunsten der *Rendering-Performance* achten solltest. Diese Liste dient nur als Richtlinie, du musst dich natürlich nicht an jeden der folgenden Punkte halten, solange dein Spiel flüssig läuft.

Rendering-Performance-Checkliste

- Erreiche 90 FPS (*Oculus Rift, HTC Vive*)
- Erreiche 60 FPS (*GearVR, Daydream, Smartphone*)
- Verwende so wenig unterschiedliche *Materials* wie möglich. Entwickle deine Modelle dafür so, dass mehrere verschiedene Modelle das gleiche *Material* verwenden können (Für *Static- & Dynamic Batching*).
- Markiere alle GameObjects, die sich nicht bewegen, als *Static*.

- Verwende möglichst nur ein einziges *Directional Light* als Lichtquelle.
- Verwende *Baked*-Lichter und -Schatten und verzichte auf *Mixed*- oder *Realtime*-Lichter und -Schatten.
- Nutze die *Skybox*, um weit entfernte Geometrie darzustellen (z.B. Berge am Horizont).
- Verwende ggf. *Draw Call Minimizer*-Add-ons, um die Performance zu optimieren.⁷
- **Besonders für Android-Apps, aber auch am PC hilfreich:**
Verwende Shader aus der „Mobile“-Kategorie,
 - am besten den *Mobile/Unlit (Supports Lightmap)* -Shader.
 - Wenn du Modelle aus dem Asset Store verwendest, ändere auch hier den Shader entsprechend.

■ 6.12 User Interface – Quickstart

Ein wichtiges Element bei einem Videospiel ist natürlich auch das User Interface. Egal ob es das Hauptmenü oder ein Inventar ist, der Spieler muss immer wieder mit verschiedenen Anzeigen und Menüs interagieren. Über gutes *User Interface*-Design könnte man durchaus ein vollkommen eigenes Buch schreiben, weswegen ich dir hier nur eine schnelle Einführung in das UI-System von Unity geben werde. Einige Tipps und Tricks zum *User Interface*-Design in VR hast du ja bereits in dem *Do's and Don'ts*-Kapitel erhalten.

6.12.1 User Interface im World Space erstellen

In diesem Abschnitt beschreibe ich dir die einzelnen Schritte, die nötig sind, um ein VR-taugliches User Interface im *World Space* zu erstellen. Eine solche *World-Space-UI* stellt die Grundlage für alle *User Interfaces* dar, die du in deine 3D-Welt integrieren kannst.

1. In Unity werden User Interfaces in einer sogenannten *Canvas* (dt. „Leinwand“) angezeigt. Erzeuge eine *Canvas* über das Create-Menü in der *Hierarchy* (**CREATE/UI/CANVAS**).
2. Zusätzlich zu dem angelegten *Canvas*-GameObject solltest du jetzt auch ein *Event System*-GameObject in der Hierarchy sehen. Dieses dient später dazu, mit dem Menü zu interagieren.
3. Wähle das *Canvas*-GameObject in der Hierarchy aus. In der *Scene View* solltest du jetzt ein riesiges weißes Rechteck sehen. Ändere im *Inspector* den *Render Mode* des *Canvas*-Components von *Screen Space – Overlay* zu *World Space*.

⁷ Zum Beispiel: <https://www.assetstore.unity3d.com/en/#!/content/2859> oder <https://www.assetstore.unity3d.com/en/#!/content/16888>. Es gibt jedoch auch andere Tools mit ähnlicher Funktion im Asset Store, suche nach „Draw Call Minimizer“.

4. Damit hast du die Canvas von der Bildschirmebene in die 3D-Welt verschoben, jetzt ist sie aber immer noch viel zu groß, um sie in die Scene integrieren zu können. Ändere die Skalierung (*Scale*) von dem *Canvas-GameObject* von (1, 1, 1) auf (0.0025, 0.0025, 0.0025)
5. Von der Größe her passt die Canvas nun schon deutlich besser in die 3D-Welt, allerdings liegt das Canvas-Objekt jetzt irgendwo in der Scene. Setze die Position über den Inspector auf (0, 0, 0), um die Canvas an einen Ort zu verschieben, von dem aus du sie gut positionieren kannst: zum Ursprung der Scene.

Bild 6.88 zeigt die Änderungen der Schritte 3, 4 und 5.



Bild 6.88 Nach dem Erstellen der Canvas solltest du sie zuerst wie hier beschrieben konfigurieren, damit du sie leichter in deiner Welt platzieren kannst.

In ihrer neuen Größe ist die *Canvas* auf einen Blick sichtbar und außerdem auch deutlich einfacher zu positionieren. Denn jetzt kannst du das *GameObject* bequem mit den Transform-Tools in deiner Scene verschieben und drehen. Wenn nötig, kannst du die Größe mit dem *Rect-Tool* im Detail anpassen. Bild 6.89 zeigt, wie die *Canvas* in eine Scene integriert werden könnte. Anschließend kannst du sie mit den UI-Elementen füllen.

Um ein UI-Element hinzuzufügen, kannst du es einfach über das Create-Menü in der Hierarchy erzeugen, es wird dann automatisch zu der *Canvas* in der Scene hinzugefügt (**CREATE / UI...**). Die UI-Elemente, die für den Anfang am wichtigsten sind, schauen wir uns jetzt noch im Detail an.

Canvas-Skalierung

Je nachdem, wie und wo du dein User Interface platzieren möchtest, kannst du auch eine größere oder kleinere Skalierung wählen. Erfahrungsgemäß sollte der Wert zwischen 0.01 und 0.0015 liegen.

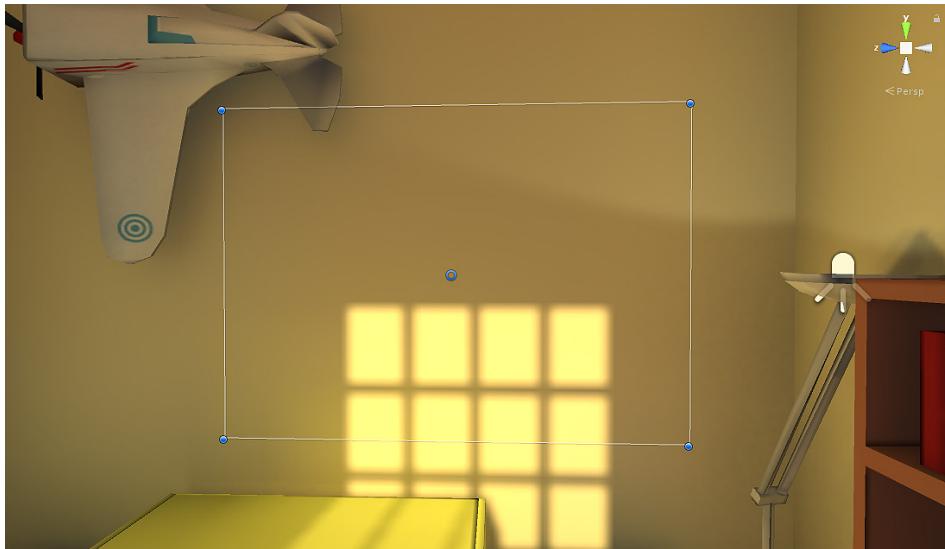


Bild 6.89 So könnte die Canvas mit ihrer neuen Größe in die Scene integriert werden.



Rect-Tool verwenden, um UI-Elemente zu positionieren

Das Rect-Tool, welches du ganz rechts bei den Transform-Tools findest, ist ein Werkzeug, das gezielt für das Verschieben und Anpassen von UI-Elementen in der Canvas erstellt wurde.

6.12.1.1 UI: Panel

Panels verwendest du in erster Linie, um deine Canvas mit einer Hintergrundfarbe zu füllen. Wenn du ein Panel über das Create-Menü erzeugst, wird ein neues GameObject mit einem Image-Component angelegt, welches automatisch so konfiguriert ist, dass es die gesamte Größe seines Elternobjektes (z.B. die Canvas) mit einem Hintergrundbild füllt. Das Panel nutzt dafür ein Image-Component, welches dir unter anderem diese Optionen im Inspector anbietet:

- **Source Image:** Diese Grafik zeigt das Panel an. Standardmäßig wird der „Background“-Sprite verwendet, der in Unity integriert ist.
- **Color:** Hier kannst du das *Source Image* einfärben und auch seine Transparenz über den Alpha-Kanal anpassen.
- **Image Type:** Hier kannst du bestimmen, wie die Grafik angezeigt werden soll. Wenn du eine eigene Grafik angibst, ändert sich dieser Wert automatisch auf *Simple*, wenn deine Grafik nicht als eine „slice“-bare Grafik erstellt wurde. Die Option *Simple* bietet dir eine Option, die den Aspekt Ratio der Grafik beibehält, sie also nicht verzerrt.

6.12.1.2 UI: Text

Erzeugst du über das Create-Menü einen Text, wird ein neues GameObject mit einem *Text*-Component angelegt. Das Component bietet dir einen reduzierten Text-Editor, mit dem du unter anderem den Text, die Farbe, die Schriftgröße und die Positionierung anpassen kannst.

Standardmäßig wird der Text relativ klein erzeugt. Ziehe den Kasten des GameObjects mit dem *Rect-Tool* größer und erhöhe dann die Schriftgröße beispielsweise auf „60“. Diese Variante, den Text zu vergrößern, erzeugt einen deutlich schärferen Text, als wenn du das GameObject größer skalierst. Ist der Text nach der Änderung der Schriftgröße nicht mehr sichtbar, ziehe den Kasten des Textes mit dem Rect-Tool noch größer.

6.12.1.3 UI: Buttons

Buttons sind ein einfaches, aber mächtiges und universal einsatzbares Element in jedem User Interface. Diese Schaltflächen funktionieren so, wie du es aus jeder anderen Anwendung kennst: Klickt du sie an, führen sie eine Methode aus, die zuvor festgelegt wurde.

Wenn du einen Button über das *Create*-Menü anlegst, wird standardmäßig ein GameObject mit einem *Button*-Component und einem *Image*-Component angelegt. Zusätzlich hat das GameObject noch ein Kind mit einem *Text*-Component, welcher den Anzeigetext des Buttons bestimmt. Das *Image*-Component stellt die Hintergrundgrafik des Buttons dar.

Buttons werden, wie Texte, in unserem Aufbau relativ klein angelegt. Ziehe das jeweilige Button-GameObject mit dem *Rect-Tool* größer und ändere anschließend die Schriftgröße des *Text*-Components, sodass der Text gut lesbar ist.

Über die verschiedenen *Color*-Eigenschaften legst du fest, wie der Button in seinen verschiedenen Zuständen aussehen soll. Diese Farbwerte werden verwendet, um das Bild des *Image*-Components je nach Zustand einzufärben. Alternativ zum Einfärben stehen dir auch noch andere Übergänge (*Transitions*) zur Verfügung. Du kannst zum Beispiel je nach Zustand auch die Grafik vollständig wechseln lassen.

Die wichtigste Option des *Button*-Components ist das *OnClick*-Event. Hier legst du fest, was passieren soll, wenn die Schaltfläche aktiviert wird. Du kannst hier ein beliebiges GameObject aus deiner Scene auswählen und anschließend eine beliebige Methode von einem der Components an dem GameObject zu dem Event hinzufügen. Bild 6.90 zeigt beispielhaft, wie dieses Feld konfiguriert aussehen könnte.

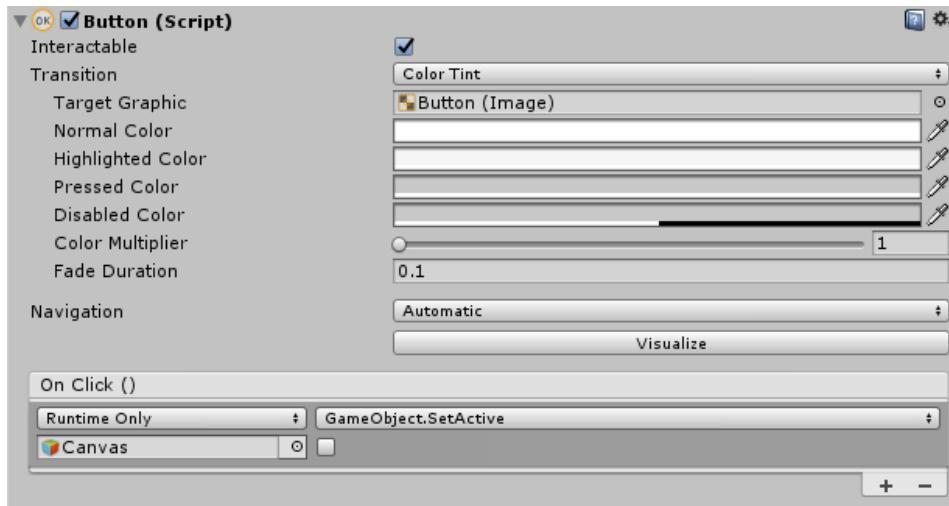


Bild 6.90 So sieht das Button-Component im Inspector aus.

Über die +-Schaltfläche kannst du einen neuen Eintrag zu der Liste hinzufügen. Bei dem linken *Object*-Feld kannst du dann ein GameObject aus deiner Scene auswählen. Anschließend musst du rechts auf die Schaltfläche **No FUNCTION** klicken. Hier kannst du jetzt als Erstes ein Component des jeweiligen GameObjects und danach eine Methode oder Variable des Components auswählen, die du aufrufen möchtest, wenn der Button geklickt wird. In Bild 6.90 wurden das *Canvas*-GameObject und die Methode *SetActive* ausgewählt, was dafür sorgen würde, dass die Canvas und somit das gesamte Menü deaktiviert wird.

6.12.1.4 Eigene Grafiken verwenden

Wenn du eine eigene Grafik in dein Unity-Projekt importierst, erkennt Unity sie zunächst als eine *Texture*, welche du in einem *Material* verwenden kannst, um ein 3D-Modell damit darzustellen. Damit du die Grafik in deinem User Interface verwenden kannst, musst du Unity sagen, dass es die Grafik als einen *Sprite* interpretieren soll.

Dazu wählst du die jeweilige Grafik im *Project Browser* aus, um die *Import Settings* der Grafik im *Inspector* sehen zu können. In diesem Fenster musst du jetzt, wie in Bild 6.91 zu sehen, den **TEXTURE TYPE** von *Default* auf *Sprite (2d and UI)* ändern. Klicke anschließend auf **APPLY**. Jetzt kannst du die Grafik zum Beispiel im *Image*-Component als Source Image angeben.

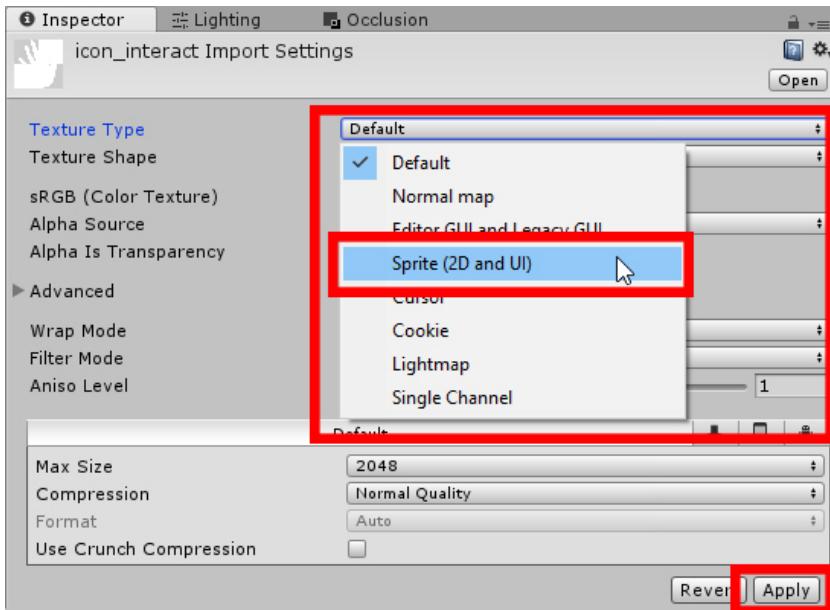


Bild 6.91 Diese Einstellung musst du ändern, um deine Grafik in einem User Interface verwenden zu können.

6.12.2 User Interface in VR bedienen

Standardmäßig kannst du mit einem VR-Headset nicht auf eine natürliche Art und Weise mit der *Unity UI* interagieren. Deshalb habe ich dir in Listing 6.17 ein Script bereitgestellt, welches dir zumindest schon erlaubt, einen Button auszuwählen und ihn per Tastendruck auszulösen. Diese Mechanik sollte für die meisten Menüs bereits vollständig ausreichen. Das Script schießt kontinuierlich Raycasts und prüft, ob das getroffene GameObject ein *Button*-Component besitzt. Wenn ja, hast du die Möglichkeit, diesen per Tastendruck zu aktivieren.

Falls du etwas Komplexeres bauen möchtest, findest du im *AssetStore* Lösungen, die dieses Problem ebenfalls angehen. Bei einigen kannst du sogar mit fast allen Unity-UI-Elementen vollständig interagieren.

Listing 6.17 VRUiInteractor-Script, welches dir erlaubt, mit UI-Buttons zu interagieren

```
using UnityEngine;
using UnityEngine.UI;
public class VRUiInteractor : MonoBehaviour {
    private Button selectedButton = null;
    // Rufe diese Methode auf, wenn die "Bestätigen"-Taste
    // auf dem jeweiligen Gamepad gedrückt wird.
    public void PressButton()
    {
        if(selectedButton != null) {
            selectedButton.onClick.Invoke();
```

```
        }
    }

    void Update () {
        RaycastHit hitInfo;
        if (Physics.Raycast(transform.position, transform.forward, out hitInfo, 5))
        {
            // Wenn das getroffene Objekt einen Button besitzt, wähle ihn aus!
            Button button = hitInfo.collider.GetComponent<Button>();
            if (button != null) {
                button.Select();
                selectedButton = button;
            }
        }
        // Beispiel-Aufruf von PressButton, klappt für PC und GearVR
        if(Input.GetButtonDown("Submit") || Input.GetKeyDown(KeyCode.Mouse0)) {
            PressButton();
        }
    }
}
```

Damit das Script funktioniert, musst du folgende Schritte beachten:

1. Erstelle ein *C#-Script* mit dem Namen *VRUiInteractor* und füge den Code aus Listing 6.17 dort ein (ersetze den gesamten bereits existierenden Inhalt).
2. Wähle das *Main Camera*-GameObject in der Hierarchy aus und füge das *VRUiInteractor*-Component zu dem GameObject hinzu.
3. Füge zu jedem *Button*-GameObject einen *Box Collider* hinzu. Dieser wird benötigt, damit der *Raycast* des Scripts dieses GameObject treffen kann.
4. Du muss noch die Größe des *Box Colliders* so anpassen, dass sie der Größe des Buttons entspricht. Für die Eigenschaften „Size X“ und „Size Y“ kannst du die Werte der Breite („Width“) und Höhe („Height“) aus dem (*Rect*)-*Transform* verwenden.

Wenn du jetzt mit deinem Headset auf einen Button schaust, wird er automatisch selektiert. Der zuletzt angesehene Button bleibt selektiert, bis du auf eine andere Schaltfläche schaust. Über **LEERTASTE** (PC) oder **TIPPEN AUF DAS TOUCHPAD** (GearVR) kannst du den aktuellen Button aktivieren. Um den Button mit einem der Controller zu aktivieren, die zu den VR-Headsets gehören (z.B. GoogleVR, SteamVR Controller oder Oculus Touch), musst du zunächst in einem eigenen Script die Eingaben von dem jeweiligen Controller einlesen und kannst dann die *PressButton()*-Methode aufrufen, um den derzeit aktiven Button zu aktivieren. Wie du Eingaben von diesen Controllern erkennst, erfährst du in dem Buchabschnitt über die einzelnen *SDKs*.