HANSER

Leseprobe

Carsten Seifert

Spiele entwickeln mit Unity 5

2D- und 3D-Games mit Unity und C# für Desktop, Web & Mobile

ISBN (Buch): 978-3-446-44563-5

ISBN (E-Book): 978-3-446-44580-2

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-44563-5

sowie im Buchhandel.

Inhalt

Vorwort

Für viele von uns sind Computerspiele heutzutage allgegenwärtige Wegbegleiter. Egal wo, auf dem Smartphone, dem Tablet oder dem heimischen PC sind sie installiert und täglich in Benutzung. Manchmal dienen sie als Zeitvertreib, bis der nächste Bus kommt, manchmal sind sie aber auch Bestandteil eines intensiven Hobbys.

Aber nicht nur das Spielen kann Spaß machen, auch das Entwickeln dieser Games kann begeistern. Sowohl im Freizeitbereich als auch in der Arbeitswelt wird der Beruf des Spieleentwicklers immer beliebter. Es ist also kein Wunder, dass sich in den letzten Jahren bereits ganze Studiengänge dem Entwickeln von Computerspielen gewidmet haben.

In diesem Buch möchte ich Ihnen Unity, eine weit verbreitete Entwicklungsumgebung für Computerspiele, näherbringen und erläutern, wie Sie mit diesem Werkzeug Spiele selber entwickeln können. Dabei richtet sich das Buch sowohl an Einsteiger, Umsteiger und auch an Spieleentwickler, die mit Unity nun richtig durchstarten möchten.

An dieser Stelle möchte ich mich ganz besonders bei meiner Frau Cornelia bedanken, die mich während des Schreibens so geduldig unterstützt hat und mir jederzeit beim Formulieren und Korrigieren hilfsbereit zur Seite stand.

Auch danke ich ganz herzlich Alexej Bodemer, der für das Beispiel-Game dieses Buches alle 3D-Modelle, Texturen und Musikdateien entworfen und zur Verfügung gestellt hat.

Zudem gilt mein Dank Will Goldstone und Unity Technologies, die mir die bis dato aktuellsten Beta-Versionen zur Verfügung gestellt haben.

Weiter möchte ich Sieglinde Schärl, Kristin Rothe und dem gesamten Hanser-Verlag-Team danken, die mir nicht nur das Schreiben dieses Buches ermöglicht haben, sondern auch jederzeit mit Rat und Tat zur Seite standen.

Nicht zuletzt danke ich auch meiner gesamten Community, die mich während des Schreibens auf meinem Blog und meinen sozialen Kanälen so konstruktiv begleitet hat.

Süderbrarup, Juni 2015

Carsten Seifert

7 Licht und Schatten

Damit eine Kamera die Textur eines *Mesh* und damit auch dessen Form darstellen kann, benötigt Ihre Szene zunächst einmal Licht. Licht hat wiederum einen enormen Einfluss auf die Darstellung der Textur, man denke nur an die Helligkeit, Position und Ausrichtung der Lichtquelle.

Unity bietet hierfür verschiedene Arten von sogenannten *Light*-Objekten an, die z. B. Echtzeitschatten erzeugen und mit anderen Effekten ausgestattet werden können. Neben dieser Echtzeitbeleuchtung unterstützt Unity auch *Lightmapping*, ein Verfahren, mit dem Sie Texturen generieren können, die beleuchtete Flächen vortäuschen. Da Lichtberechnungen sehr rechenintensiv sind, kann durch das Nutzen von *Lightmapping* die Performance erheblich gesteigert werden.

7.1 Ambient Light

Unity besitzt eine globale Beleuchtung namens *Ambient Light*, die die komplette Szene mit einer Grundhelligkeit ausstattet. Die Einstellungen für diese Beleuchtung finden Sie im Lighting-Fenster, das Sie über das Menü WINDOWS/LIGHTING erreichen. Im Bereich „Environment Lighting“ (siehe Bild 7.1) finden Sie die Parameter des *Ambient Light*:

- **Ambient Source** legt die Quelle bzw. die Farbe(n) des Lichts fest. Sie haben die Wahl zwischen „Skybox“ (hier werden die Farben der *Skybox* als Grundlage genommen), „Gradient“ (erlaubt Ihnen, einen Farbübergang mit maximal drei Farben zu bestimmen) und „Color“ (bei dem eine einzige Farbe festgelegt wird).
- **Ambient Intensity** bestimmt die Lichtstärke des *Ambient Light*.
- **Ambient GI** legt fest, ob das *Ambient Light* zur Laufzeit berechnet wird oder beim *Baken* der *Lightmaps* erstellt wird. Da das Licht entweder über die *Global Illumination*-Verfahren *Precomputed Realtime GI* oder *Baked GI* gesteuert wird, müssen Sie hier nur eine Auswahl treffen, wenn beide im *Lighting*-Fenster aktiv sind. Ist nur eine aktiviert, wird diese automatisch genommen. Mehr zu diesen Verfahren lesen Sie im Abschnitt „Global Illumination“.

Möchten Sie ein Spiel entwickeln, das keine Grundbeleuchtung besitzen soll, weil es beispielsweise im Dunkeln spielt, können Sie die *Ambient Intensity* auf 0 stellen. Oder Sie stellen die *Ambient Source* auf „Color“ und setzen die Farbe auf Schwarz.

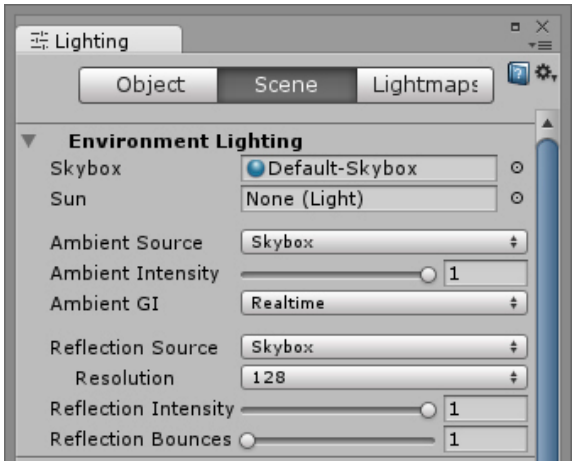

Bild 7.1
Lighting-Fenster

7.2 Lichtarten

Über das Menü GAMEOBJECT/LIGHT können Sie vier unterschiedliche Beleuchtungsobjekte Ihrer Szene zufügen: *Directional Light*, *Point Light*, *Spot Light* und das *Area Light*. Letzteres nimmt eine Sonderrolle ein, worauf ich noch eingehen werde. Alle Objekte besitzen eine *Light*-Komponente, die das Herzstück einer Lichtquelle darstellt. Die Komponente besitzt unterschiedliche Parameter, die je nach *Type* der *Light*-Komponente zur Verfügung stehen. Die folgenden Parameter stehen aber mit Ausnahme des *Area Lights* immer zur Auswahl:

- **Type** bestimmt die Art des Lichtes und damit auch die zur Verfügung stehenden Parameter.
- **Baking** bestimmt das Verhalten beim Erstellen und Nutzen von *Lightmaps*. *Realtime* schließt dieses Licht beim Erstellungsprozess der *Lightmaps* (auch *baken* genannt) aus. *Mixed* berücksichtigt das Licht beim *Lightmapping*, ist aber auch zur Laufzeit im normalen Spiel aktiv, um die nichtstatischen Objekte zu beleuchten. *Baked* bindet das Licht in das *Baken* ein, deaktiviert es aber während des normalen Spiels.
- **Range** setzt die Reichweite der Lichtquelle fest.
- **Intensity** definiert die Lichtstärke.
- **Bounce Intensity** bestimmt die Helligkeit des Lichts, das von angestrahlten Flächen zurückgeworfen wird, auch indirekte Beleuchtung genannt. Mehr zu diesem Parameter erfahren Sie im Abschnitt „Global Illumination“.
- **Color** legt die Lichtfarbe fest.

- **Cookie** ermöglicht, eine Lichtschablone vor eine Lichtquelle zu legen. Dies ist je nach *Type* entweder eine einzelne Schwarz-Weiß-Grafik oder eine *Cubemap*, bestehend aus sechs solcher Grafiken.
- **Shadow Type** legt die Art des Schattens fest. Allgemein stehen *No Shadows*, *Hard Shadows* und *Soft Shadows* zur Verfügung. Auf die Details werde ich gleich noch eingehen.
- **Draw Halo** bestimmt, ob ein *Light Halo* dargestellt werden soll.
- **Flare** legt ein *Flare*-Objekt zum Darstellen eines Lichtscheins für diese Lichtquelle fest (siehe „Flare“).
- **Render Mode** legt fest, ob dieses Licht beim *Forward Rendering* per Pixel oder per Vertex gerendert werden soll. *Auto*: Unity bestimmt, auf welche Art gerendert wird. *Important* bedeutet per Pixel, *Not Important* bedeutet per Vertex. Der Wert *Pixel Light Count* (zu finden in den *Quality Settings*) bestimmt bei *Auto*, wie viele Lichtquellen insgesamt per Vertex gerendert werden können.
- **Culling Mask** definiert, welche Objekte eines *Layers* nicht von dieser Lichtquelle beeinflusst/beleuchtet werden.

7.2.1 Directional Light

Ein *Directional Light* beleuchtet die komplette Szene aus einer Richtung. Die Position der Lichtquelle spielt dabei keine Rolle, nur die Rotation der Quelle ist hierbei wichtig. Ein *Directional Light* kann sowohl mit *Forward Rendering* als auch *Deferred Lighting* (siehe Abschnitt „Rendering Paths“) Echtzeitschatten erzeugen.

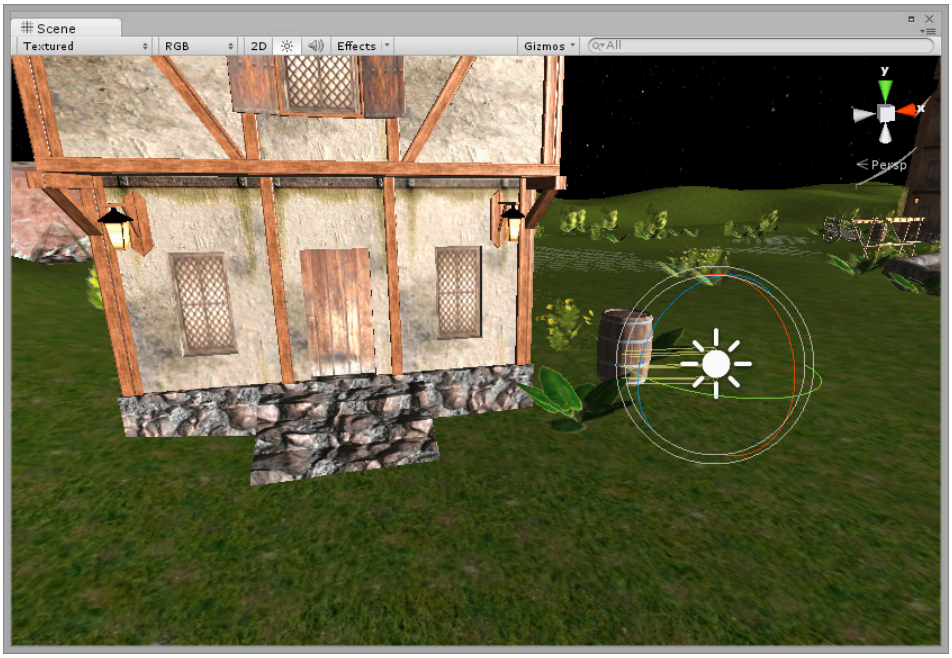

Bild 7.2 Directional Light

7.2.2 Point Light

Das *Point Light* ist eine Lichtquelle, die in alle Richtungen gleichmäßig abstrahlt, vergleichbar mit einer Glühlampe an der Decke. Im Gegensatz zum *Directional Light* ist hier die Position, aber nicht die Rotation wichtig. Über den Parameter *Range* legen Sie die Reichweite der Lichtquelle fest. Echtzeitschatten von *Point Lights* werden nur in Kombination mit dem *Rendering Path Deferred Lighting* unterstützt.

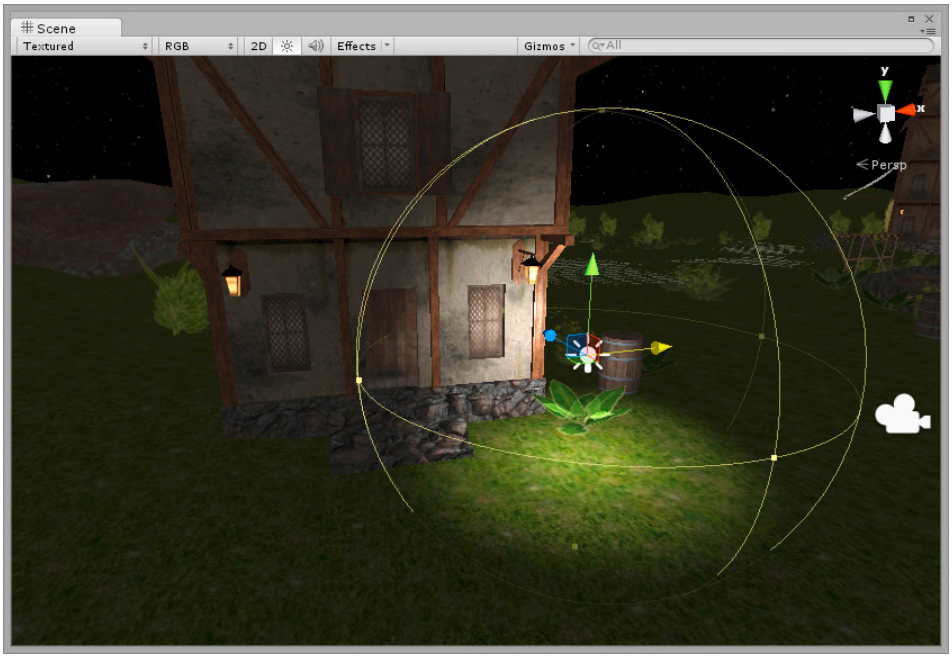

Bild 7.3 Point Light

7.2.3 Spot Light

Ein *Spot Light* scheint trichterförmig von der Lichtquelle in eine bestimmte Richtung, ähnlich wie eine Taschenlampe. Über den Parameter *Range* legen Sie fest, wie weit sie scheint. *Spot Angle* legt den äußeren Abstrahlwinkel fest. Echtzeitschatten von *Spot Lights* werden wie beim *Point Light* ebenfalls nur in Kombination mit dem *Rendering Path Deferred Lighting* unterstützt.

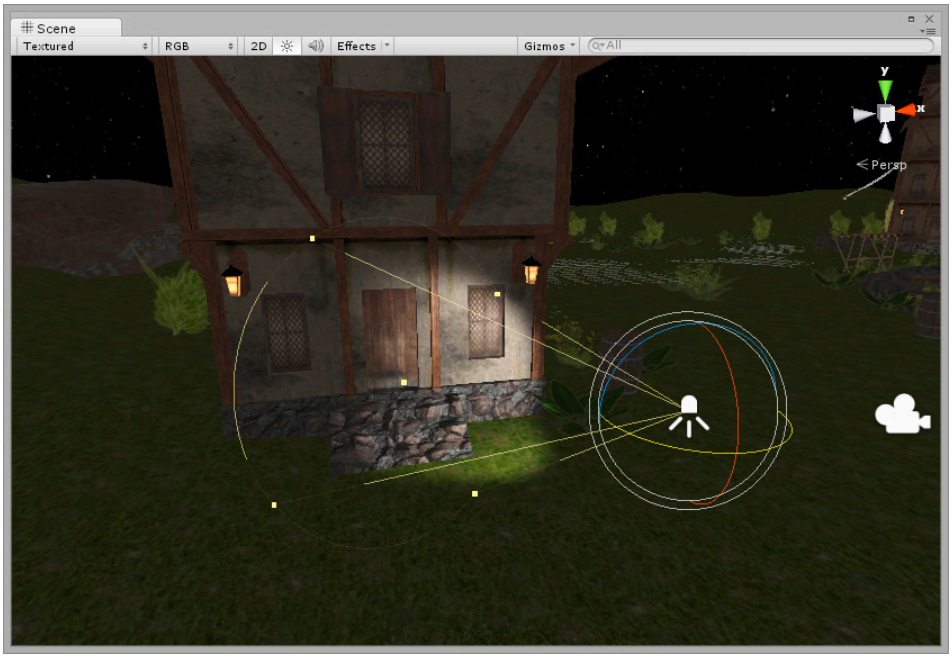

Bild 7.4 Spot Light

7.2.4 Area Light

Ein *Area Light* nimmt eine Sonderstellung bei den *Light Types* ein, da es ausschließlich beim Erstellen von *Lightmaps* berücksichtigt wird, nicht aber zur Echtzeit virtuelles Licht emittiert. Diese Lichtquellen arbeiten also ausschließlich so, als wenn der *Baking*-Parameter auf „Baken“ gestellt wäre.

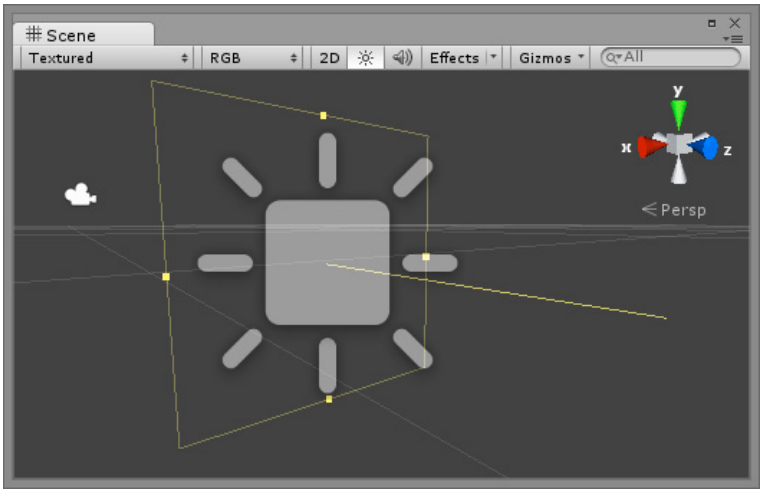

Bild 7.5 Area Light

Ein *Area Light* erscheint nach dem Erstellen der Lightmap wie eine Rechteckfläche, die in alle Richtungen einer Seite scheint, vergleichbar mit dem Bildschirm eines Fernsehers. Die Richtung wird mit einer Linie dargestellt (siehe Bild 7.5). Die Fläche wird durch die Parameter *Width* und *Height* festgelegt. Und auch hier wird die Reichweite der Beleuchtung über die *Intensity* definiert.

7.3 Schatten

Unity bietet zwei unterschiedliche Echtzeitschattenarten an: *Hard* und *Soft Shadows*. *Hard Shadows* sind eine performance-schonende Variante, *Soft Shadows* sind dafür detaillierter. Zudem können Sie einer Lichtquelle auch den Parameter *No Shadows* mitgeben. In dem Fall werfen angestrahlte Objekte überhaupt keine Schatten.

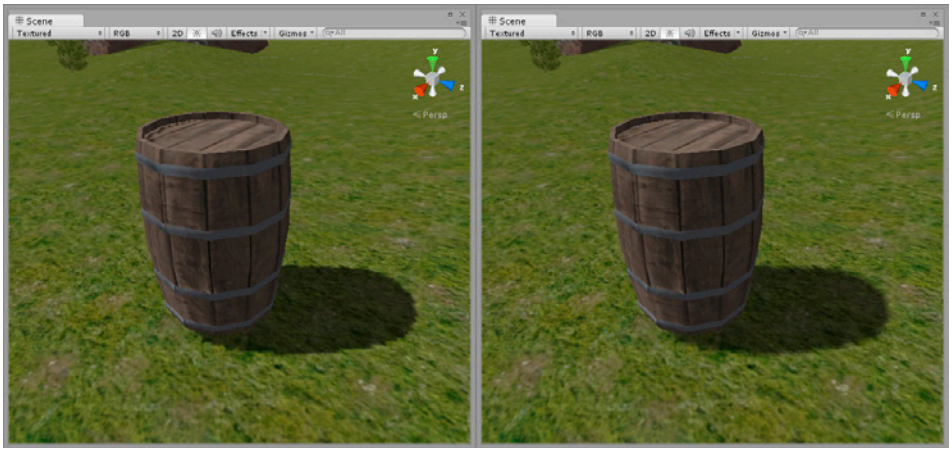

Bild 7.6 Vergleich: Hard Shadows vs. Soft Shadows

Neben der Schattenart bieten die Lichtquellen noch weitere Parameter an.

- **Strength** legt die Dunkelheit des Schattens fest.
- **Resolution** definiert die Qualität bzw. die Auflösung des Schattens. Standardmäßig wird hier für die Einstellung auf die *Quality Settings* (EDIT/PROJECT SETTINGS/QUALITY) verwiesen. Sie kann aber auch überschrieben werden.
- **Bias** hat einen Einfluss auf den Abstand des Objektes zum Schatten. Beginnt der Schatten zu nah am Objekt, kann dies zu optischen Fehlern führen. Deshalb ist standardmäßig ein Wert von 0,05 vorgegeben.

7.3.1 Einfluss des MeshRenderers auf Schatten

Über die *MeshRenderer*-Komponente eines jeden sichtbaren Objektes haben Sie die Möglichkeit zu steuern, ob ein Objekt überhaupt Schatten erzeugen soll oder nicht. Dies können Sie über die Eigenschaft **Cast Shadows** steuern. Genauso können Sie über die Eigenschaft **Receive Shadows** definieren, ob auf dem Objekt selber Schatten anderer Objekte dargestellt werden sollen. Wenn Sie beispielsweise einem Spieler einen Unsichtbarkeitszauber zuführen, darf dieser selber keine Schatten werfen, aber auch keine Schatten darstellen, die andere Objekte auf ihn werfen. In diesem Fall wird der Schatten einfach zum nächsten Objekt „durchgeleitet“, sodass dort der Schatten dargestellt wird. Das Bild 7.7 zeigt zwei Fässer. Während beim linken Fass beide Parameter aktiv sind und dieses einen Schatten wirft, wurden beim rechten Fass beide Parameter deaktiviert. Der Schatten des linken Objektes wird deshalb nicht auf dem rechten Fass dargestellt. Zudem wirft das rechte Fass auch selber keinen Schatten, sodass nur der Schattenwurf des linken gezeigt wird.

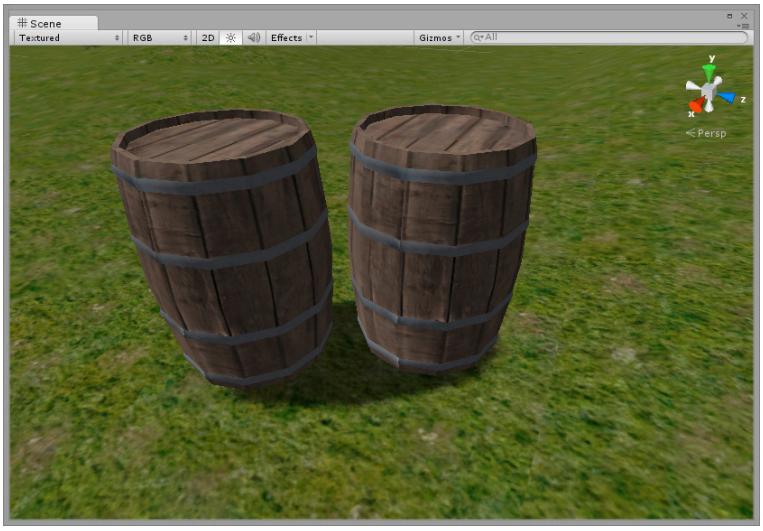

Bild 7.7 Einfluss des „Cast Shadows“- und „Receive Shadows“-Parameter

Der Parameter *Cast Shadows* bietet noch weitere Einstellmöglichkeiten an. Neben *On* und *Off* können Sie auch *Two Sided* auswählen. In diesem Fall wirft das *Mesh* in beide Richtungen einen Schatten. Haben Sie beispielsweise eine Plane, um die Sie eine Lichtquelle rotieren lassen, würde der Schatten sich genauso verhalten, wie Sie es erwarten würden, auch wenn das *Mesh* selber vielleicht nur aus einer Richtung zu sehen ist (siehe „Normalenvektor“ im Kapitel 5 „Objekte in der zweiten und dritten Dimension“). Als dritte Auswahlmöglichkeit können Sie *Shadows Only* auswählen. In diesem Fall wird nur der Schatten dargestellt, nicht aber das *Mesh*.

7.4 Light Cookies

Bei einem *Light Cookie* handelt es sich um eine Lichtschablone, die das abgegebene Licht in bestimmte Formen bringt. Wenn Sie ein Comic-Fan sind, dann kennen Sie sicher das Batman-Zeichen, dass die Polizei von Gotham-City an den Himmel wirft, um Batman zu rufen. Genau das ist ein *Light Cookie*, genauer gesagt ein *Spot Light* mit einem *Light Cookie*.

Das Bild 7.8 zeigt Ihnen einen ähnlichen Effekt. Dort wurde im rechten Motiv ein *Spot Light* mit einem *Light Cookie* versehen, das einen taschenlampenähnlichen Effekt erzeugt. Diesen wie auch weitere *Light Cookies* liefert Unity in seinen *Standard Assets* „Effects“ mit.

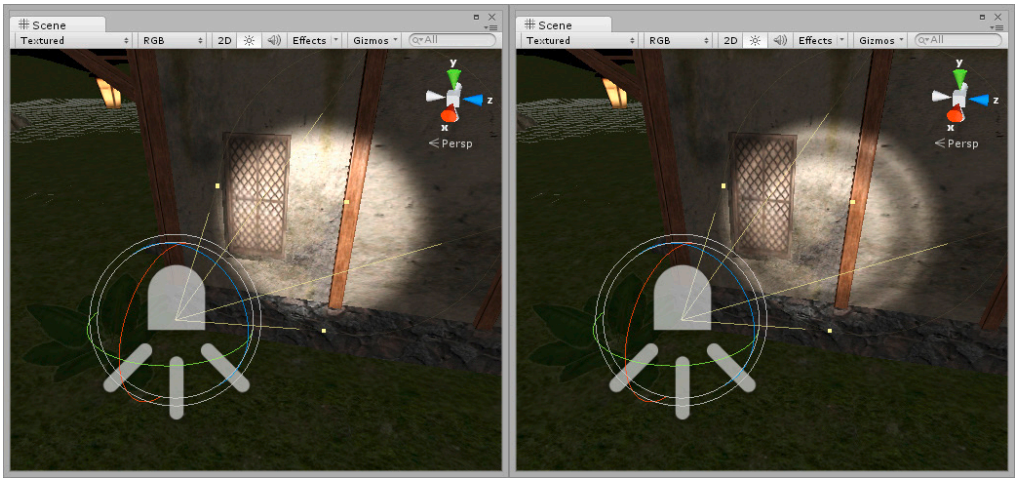

Bild 7.8 Spot Light mit einem taschenlampenähnlichen Light Cookie

7.4.1 Import Settings eines Light Cookies

Der Kern eines *Light Cookies* ist eine Schwarz-Weiß-Grafik, auch *Grayscale Texture* genannt. Schwarz bedeutet hierbei keine Lichtdurchlässigkeit, Weiß lässt das komplette Licht durch. Eine wichtige Rolle bei einer Cookie-Textur spielt der *Wrap Mode*.

- Bei *Spot Lights* wird häufig der *Wrap Mode* auf *Clamp* gestellt, um auf diese Weise nur eine Abbildung des *Cookies* zu erhalten.
- Bei *Directional Lights* wird häufig der *Wrap Mode* auf *Repeat* gestellt, um so das *Cookie*-Motiv zu wiederholen. Ein *Directional Light* bietet hierfür noch den zusätzlichen Parameter *Cookie Size* an, der die Größe eines einzelnen Motives festlegt. Auf diese Weise können zum Beispiel in einer gesamten Landschaft Lichtunregelmäßigkeiten erzeugt werden, die etwa durch Wolken erzeugt werden.

7.4.2 Light Cookies und Point Lights

Da *Point Lights* in alle Richtungen strahlen, reicht es nicht aus, eine einfache Textur als *Cookie* zu nutzen. Hierfür werden *Cubemaps* genutzt. Cubemaps sind Texturen, die eine Rundumsicht darstellen. Wie der Name schon verrät, können Sie sich das vorstellen wie einen aufgeklappten Würfel, Unity nennt diese Darstellung „6 Frames Layout“. Sie können aber auch andere Formate für eine *Cubemap* nutzen. In den Import-Settings der Textur können Sie dies im *Mapping*-Parameter hinterlegen, wenn Sie den *Texture Type Cubemap* gewählt haben.

Bei einem Point Light können Sie sich das nun so vorstellen, dass diese Textur wieder zu einem Würfel zusammengeklappt wird und die Lichtquelle in der Mitte ist. Dadurch wirkt sich der Light Cookie nun in alle Richtungen aus.

Zusätzlich unterstützt Unity noch sogenannte Legacy Cubemaps. Dies ist eine Asset-Art, der Sie sechs getrennte Texturen zuweisen. So eine Legacy *Cubemap* erzeugen Sie über **ASSET/CREATE/LEGACY/CUBEMAP**. Auch wenn diese Art von *Cubemaps* vielleicht einfacher zu erstellen ist, so unterstützen diese leider nicht die gleichen grafischen Funktionen wie die oberen (die aber bei *Light Cookies* nicht so relevant sind).

Den *Texture*-Slots werden nun je nach Effektwunsch verschiedene oder gleiche *Cookie*-Texturen zugewiesen. Anschließend wird die *Cubemap* dann dem *Cookie*-Slot der *Light*-Komponente vom *Point Light* zugewiesen, und schon haben wir auch dort den *Light Cookie*-Effekt, allerdings dieses Mal in alle Richtungen.

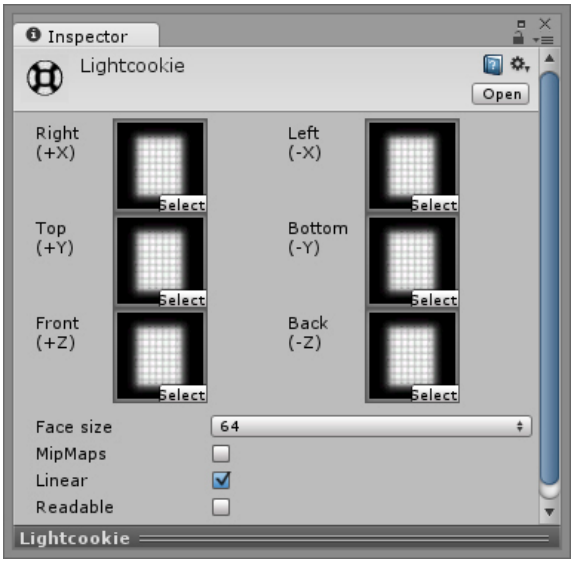

Bild 7.9
Beispiel eines Cubemaps Light Cookie

Das Bild 7.10 zeigt den Vergleich eines *Point Lights* ohne und mit einem *Light Cookie*.

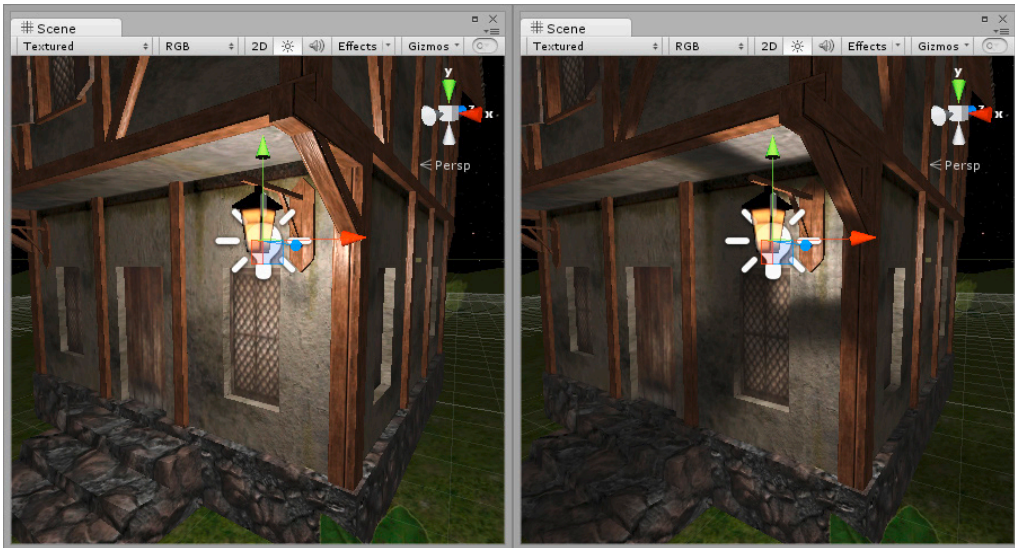

Bild 7.10 Einsatz eines Light Cookies bei einem Point Light

7.5 Light Halos

Ein *Light Halo* ist ein Lichtschleier, der in der realen Welt durch Staubpartikel in der Luft entsteht. Da in Unity natürlich kein Staub vorhanden ist, wird dieser eben durch ein solches *Halo* simuliert. Über die *Light*-Komponenten-Eigenschaft *Draw Halo* können Sie einen Standard-*Halo* aktivieren, der sich an der Lichtstärke *(Intensity)*, der Lichtfarbe *(Color)* sowie an der Reichweite *(Range)* der Light-Komponente orientiert.

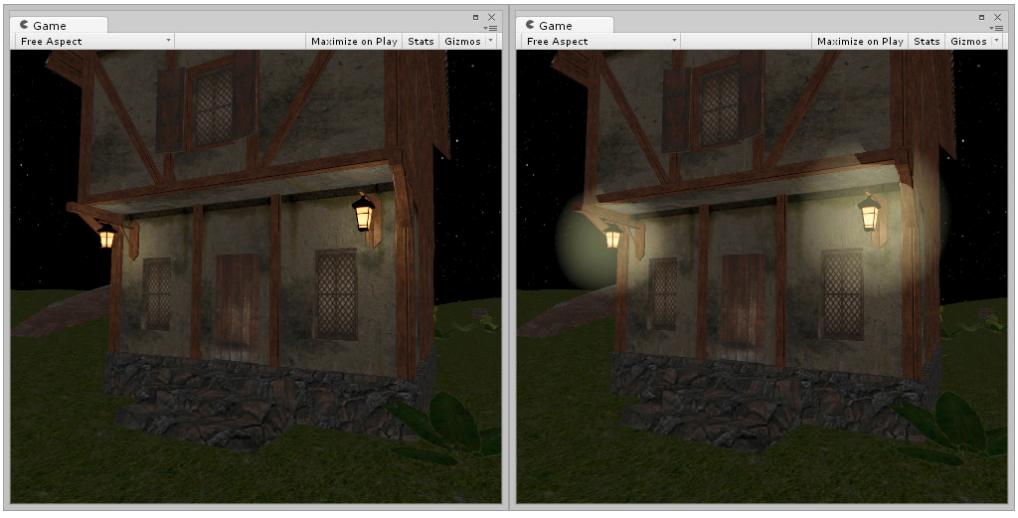

Bild 7.11 Light Halos

Das Bild 7.11 zeigt den Vergleich zweier *Point Lights* mit deaktiviertem und aktiviertem *Draw Halo*-Parameter.

7.5.1 Unabhängige Halos

Sie können neben den Standard-*Halos*, die Sie an den *Light*-Komponenten aktivieren können, auch unabhängige *Halos* erzeugen. Hierfür fügen Sie einem beliebigen *GameObject* über COMPONENT/EFFECTS/HALO (oder über ADD COMPONENT im *Inspector*) eine *Halo*-Komponente zu. Dies kann auch das gleiche Objekt sein, das bereits eine *Light*-Komponente besitzt. Der Unterschied ist nur, dass sich dieses *Halo* nicht an den Eigenschaften der *Light*-Komponente orientiert, sondern frei konfigurierbar ist.

7.6 Lens Flares

Lens Flares simulieren Linsenreflexionen, also Effekte, die bei Gegenlicht in Kameralinsen entstehen. Das Bild 7.12 zeigt zwei unterschiedliche *Lens Flare*-Effekte, die Unity bereits in den *Standard Assets* „Effects“ bereitstellt. Das linke Motiv zeigt einen kleinen Effekt, der sich lediglich an der Lichtquelle selber zeigt, das rechte Teilbild zeigt einen sehr ausgeprägten Effekt, der auch verschobene Linseneffekte erzeugt.

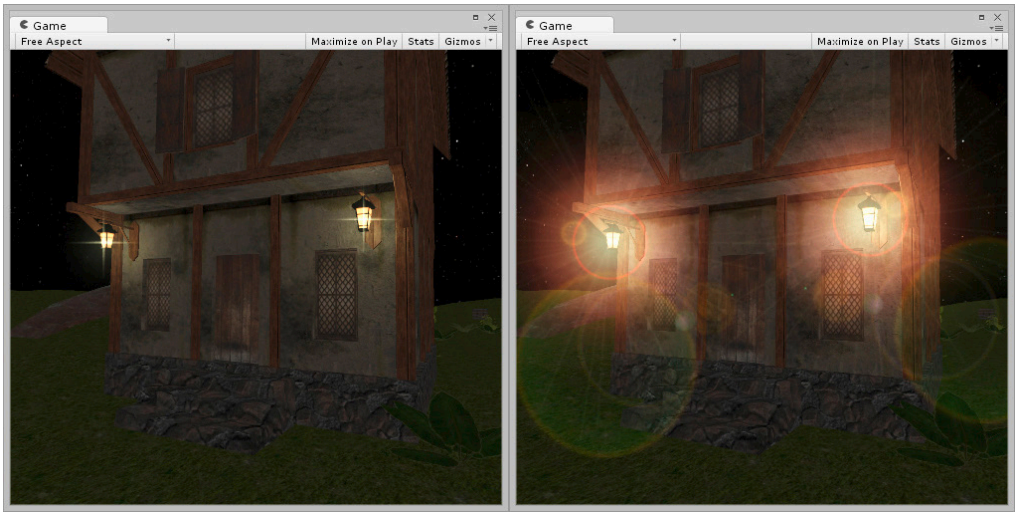

Bild 7.12 Lens Flares „Small Flare“ und „50 mm Zoom“

Zum Nutzen von *Lens Flares* sind mehrere Dinge wichtig. Zunächst benötigen Sie ein *Flare*-Objekt, das den eigentlichen Effekt und dessen Verhalten beschreibt (siehe *Standard Assets*). Da *Flare*-Objekte aber nicht alleine existieren können, müssen Sie diese nun einem *GameObject* zuweisen. Dies können Sie entweder über die *Flare*-Variable einer *Light*-Komponente

machen oder aber über eine separate *Lens Flare*-Komponente, die Sie einem beliebigen *GameObject* zuweisen können. Diese finden Sie über **COMPONENT/EFFECTS/LENS FLARE**.

Als Letztes muss die Kamera noch eine *Flare Layer*-Komponente besitzen. Standardmäßig ist dies aber der Fall (siehe Kapitel 6 „Kameras, die Augen des Spielers“).

7.6.1 Eigene Lens Flares

Sie können natürlich nicht nur die mitgelieferten *Lens Flares* nutzen, sondern auch eigene erzeugen. Dies machen Sie über das Menü **ASSETS/CREATE** bzw. über die rechte Maustaste im *Project Browser*.

Dem *Flare-Asset* können Sie einen *Texture-Atlas* zuweisen und anschließend das Verhalten an sich festlegen. Ein *Texture-Atlas* ist eine große Textur, die aus vielen kleinen Bildern besteht. Damit Unity weiß, an welcher Stelle sich ein Unterbild befindet, müssen diese *Flare*-Texturen einen bestimmten Aufbau besitzen. Unity bietet hierfür sechs unterschiedliche Layouts an. Über die *Texture Layout*-Eigenschaft teilen Sie schließlich dem *Flare*-Objekt mit, welchen Aufbau Sie auf der Textur nutzen. Details erfahren Sie über den Hilfe-Button oben rechts im *Inspector* des *Flare*-Objekts.

7.7 Projector

Eine weitere Möglichkeit, Licht und Schatten zumindest optisch darzustellen, sind sogenannte Projektoren bzw. *Projectors*. Wie der Name schon vermuten lässt, arbeitet dieser wie ein Beamer (oder ein Tageslichtprojektor oder DIA-Projektor), der ein Bild bzw. Material abstrahlt. Alle Objekte, die diesen Projektierungskegel schneiden, werden dann mit diesem Material überlagert. Hierdurch können sehr interessante Effekte erzielt werden.

7.7.1 Standard Projectors

In den *Standard Assets* gibt es unter anderem einen *Blob Light Projector*, der eine weiße, kreisförmige Textur auf die angestrahlten Objekte projiziert. Dies wirkt wie ein Lichtkegel, nur dass es von der Berechnung her eben kein Licht, sondern nur eine halbtransparente Textur ist. Objekte können also auch keine Schatten werfen.

Als Gegenstück gibt es ebenso auch noch den *Blob Shadow Projector*. Dieser wirft keine helle Textur auf die Objekte, sondern eine schwarze. Platzieren Sie diesen *Projector* über einem Objekt und strahlen Sie auf diesen hinab, können Sie damit einen Schatten simulieren.

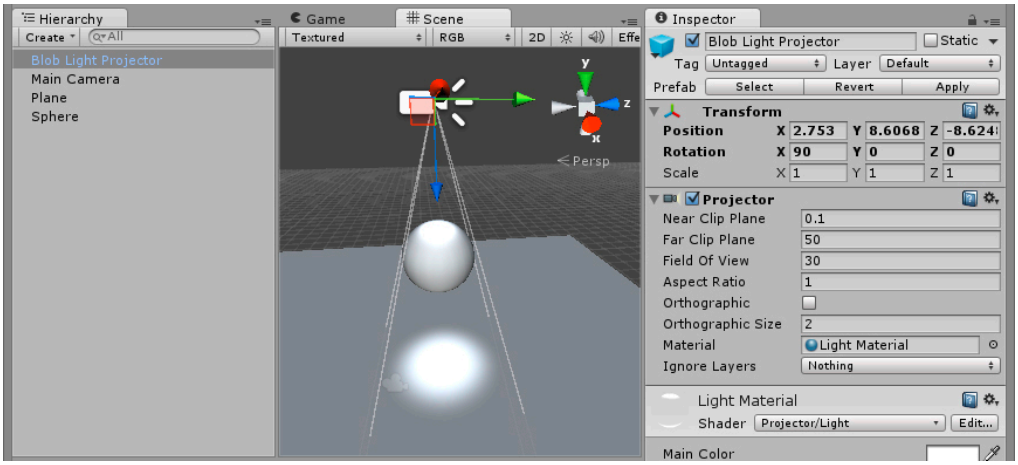

Bild 7.13 Blob Light Projector

Hierfür müssen Sie dem schattenwerfenden Objekt einen Layer zuweisen, den Sie in der *Ignore Layers*-Eigenschaft des *Projectors* hinterlegen. Hierdurch wird die schwarze Textur nicht mehr auf diesem Objekt, sehr wohl aber auf dem Untergrund angezeigt (siehe Bild 7.14). Am besten ist es natürlich, wenn Sie dem *Material*, das dem *Projector* zugewiesen wird, eine Textur zuweisen, die der Form des Objektes entspricht.

Damit der Schatten sich nun auch mit dem Objekt mitbewegt, empfiehlt es sich, den *Projector* als Kind-Objekt dem schattenwerfenden Objekt zuzuweisen – in Bild 7.14 wäre das die Kugel.

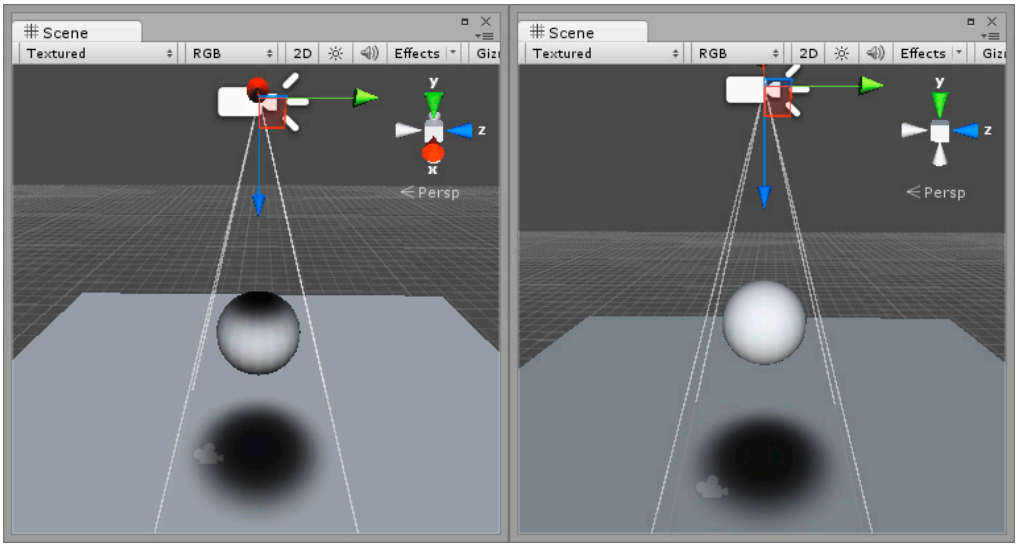

Bild 7.14 Blob Shadow Projector

7.8 Lightmapping

Licht- und Schattenberechnungen sind sehr rechenintensiv. Und umso detaillierter diese dargestellt werden sollen, desto mehr muss gerechnet werden. Um trotzdem in einem Spiel sehr detaillierte Schatten und Lichtszenarien zu erhalten, gibt es das sogenannte *Lightmapping*. Dieses Verfahren berechnet bereits zur Entwicklungszeit die Licht- und Schatteneffekte und erstellt Texturen, die dann über die Modelle gelegt werden. Danach können die Lichtquellen deaktiviert werden, und trotzdem wirken die Objekte, als würden diese angestrahlt werden.

Ein wichtiger Punkt beim herkömmlichen *Lightmapping* ist der, dass nur Objekte berücksichtigt werden, die sich nicht bewegen. Das betrifft sowohl die Lichtobjekte als auch die beleuchteten Objekte. Der Grund hierfür ist ganz einfach: Stellen Sie sich vor, Sie berechnen den Schatten eines Autos und „brennen" dessen Schatten in die Textur der Straße ein. Nun fährt das Auto weg und die Straßentextur mit dem Schatten bleibt an der gleichen Stelle. Dies ist natürlich nicht gerade das, was man als realistisch bezeichnen würde. Deshalb berücksichtigt Unity beim normalen *Lightmapping* nur Objekte, die statisch sind, also Objekte, die sich nicht bewegen, und Lichtquellen, bei denen der Baking-Modus auf *Mixed* oder *Baken* gestellt ist. Allerdings bietet Unity auch eine Möglichkeit, bewegliche Objekte vom Lightmapping profitieren zu lassen. Hierbei werden sogenannte *Light Probes* eingesetzt, die ich aber noch im folgenden Abschnitt „Light Probes" behandeln werde.

Zum Markieren statischer Objekte besitzen alle *GameObjects* in einer Szene eine kleine Checkbox oben rechts im *Inspector* mit dem Namen *Static*. Setzen Sie diesen Haken bei allen Objekten, die sich nicht bewegen und beim *Lightmapping* berücksichtigt werden sollen.

Da sich mittlerweile mehrere Funktionen dieser *Static*-Eigenschaft bedienen, können Sie über den zusätzlichen Pfeil an der rechten Seite der *Static*-Checkbox ein weiteres Menü aufklappen. Hier können Sie definieren, für welche Funktionen diese *Static* Eigenschaft gilt. Achten Sie darauf, dass hier *Lightmap Static* aktiviert ist.

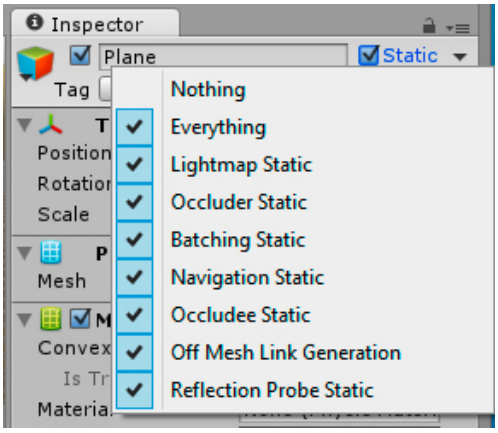

Bild 7.15
„Lightmap Static"-Option im Static-Menü

Beachten Sie, dass in Unity per Default *Continuous Baking* aktiviert ist. Das bedeutet, dass Unity automatisch die *Lightmaps* neu berechnet, sobald in einer Szene eine Aktion gemacht wurde, die berechnete *Lightmaps* beeinflussen könnten.

Um dies zu ändern, öffnen Sie das *Lighting-Fenster* (Window/Lighting) und entfernen Sie den Haken bei dieser Eigenschaft. Diese finden Sie ganz unten im Reiter „Scene“. Haben Sie *Continuous Baking* (ab Unity 5.1 wird voraussichtlich die Bezeichnung „Auto“ angezeigt) deaktiviert, wird das *Baken* nur noch manuell über den daneben befindlichen Knopf **BUILD** gestartet. Diese Einstellung sollten Sie auf jeden Fall dem Automatismus vorziehen, da das ständige Erstellen von *Lightmaps* den Arbeitsfluss doch erheblich ausbremst. Und umso größer das Projekt wird, desto länger dauert das Erstellen der *Lightmaps*.

Achten Sie darauf, dass beim *Baken* nur *Area Lights*, Lichtquellen mit dem *Baking*-Modus *Mixed* oder *Baked*, sowie *Materials* mit *Emission*-Werten bzw. Texturen, deren *Global Illumination*-Eigenschaften auf „Baked“ stehen (siehe „Standard-Shader“ im Kapitel 5 „Objekte in der zweiten und dritten Dimension“), berücksichtigt werden.

Emissionswerte und Texturen

Durch eine Emission-Textur (oder einen einfachen Wert) verleiht der Standard-Shader einem Material das Aussehen, als würde es leuchten. Für gewöhnlich werden diese *Shader* deshalb auch für Lichtquellen-Meshes wie Lampen, Laternen etc. genutzt. Steht nun zusätzlich der *Global Illumination*-Parameter des Materials auf „Baked“, werden beim Lightmapping-Verfahren die Objekte mit diesen Materialien ebenfalls als Lichtquellen berücksichtigt und in die Lightmaps „gebrannt“.

Da das Erstellen der *Lightmaps* je nach Einstellung und *Szenenaufbau* durchaus einige Zeit dauern kann, wird der Fortschritt des Vorgangs unten rechts als Balken angezeigt. Nach dem *Baken* werden die fertigen *Lightmaps* schließlich im *Lightmaps*-Bereich des *Lighting*-Fensters angezeigt und in der Szene über die statischen Objekte gelegt.

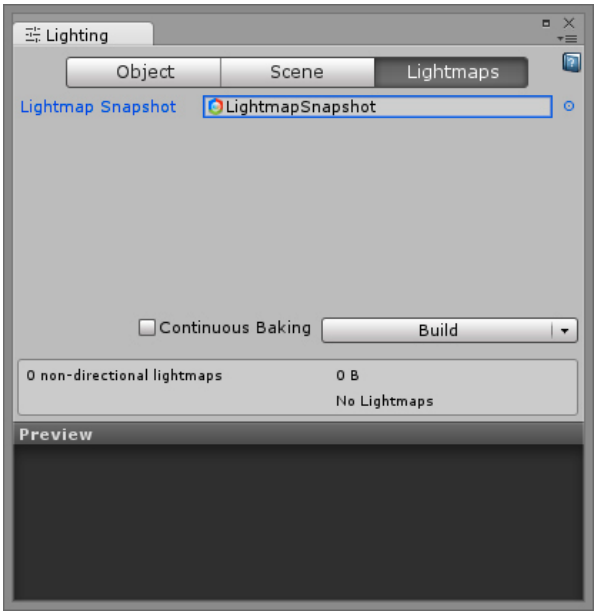

Bild 7.16
Lighting-Fenster mit Lightmaps-Bereich

Mithilfe der *Lightmap Snapshot*-Eigenschaft (siehe Bild 7.16) können Sie auch zwischen verschiedenen *Lightmaps* wechseln oder auch gar kein *Lightmapping* zuweisen („None“). Auf diese Weise können Sie zu Testzwecken schnell zwischen verschiedenen Lightmapping-Szenarien oder auch die Szene ohne *Lightmapping* betrachten – vorausgesetzt natürlich, Sie haben *Continuous Baking* deaktiviert.

Beachten Sie zudem auf dem *Scene*-Reiter des *Lighting*-Fensters die Funktion *Baked GI*. Sie berechnet die *Global Illumination* (siehe Abschnitt „Global Illumination“), also die indirekte Beleuchtung beim *Lightmapping*. Ist diese deaktiviert, können Sie auch keine *Lightmaps* erstellen.

Lightmapping-Beispiel

Im Online-Bereich dieses Buches finden Sie eine Video-Demonstration des *Lightmapping*-Verfahrens, in der die wichtigsten Parameter vorgestellt werden. Weiter werden auch *Emissive Materials* und das *Area Light* gezeigt.

7.8.1 Light Probes

Lightmapping bietet Ihnen in Sachen Performance, aber auch in Sachen der Detailauflösung große Vorteile. Ein großer Nachteil des herkömmlichen *Lightmappings* ist hierbei, dass es nur statische Objekte berücksichtigen kann.

Mithilfe von *Light Probes* können nun auch bewegliche Objekte vom *Lightmapping* profitieren. Beachten Sie dabei, dass bei jedem beweglichen Objekt, welches von den *Light Probes* beeinflusst werden soll, die *Use Light Probes*-Eigenschaft vom *MeshRenderer* aktiviert sein muss.

Hinter den *Light Probes* steckt der Gedanke, dass die Beleuchtung im Vorwege an strategischen Stellen gespeichert wird. Befindet sich ein Objekt nun zwischen verschiedenen Messstellen, dann wird die Beleuchtung näherungsweise berechnet und dem Objekt zugewiesen. *Light Probes* sind nun genau diese Messstellen und werden, sobald sie in der *Hierarchy* selektiert werden, in der *Scene View* gelb dargestellt (siehe Bild 7.17).

Auch wenn Sie mit *Light Probes* vom *Lightmapping* profitieren, sollten Sie trotzdem statische Objekte auch immer als solche definieren. Denn Lichtberechnungen des normalen *Lightmappings* sehen immer besser aus als über *Light Probes* interpolierte Lichtwerte. Zudem sollten Sie bedenken, dass durch *Light Probes* selber keine Schatten entstehen, durch *Lightmapping* aber schon.

Das Bild 7.17 zeigt eine kleine Szene mit einem statischen Cube, der ein Material mit einer grünen *Emission*-Farbe besitzt (siehe „Der Standard-Shader“ im Kapitel 5 „Objekte in der zweiten und dritten Dimension“). Hinzu kommen eine statische Plane (*Lightmap Static* ist hier aktiv) sowie ein bewegliches Capsule-Objekt, bei dem der *Use Light Probes*-Parameter des *MeshRenderers* aktiviert ist. Dank der *Light Probes* (gelb dargestellt) wird nun auch die Kapsel beleuchtet.

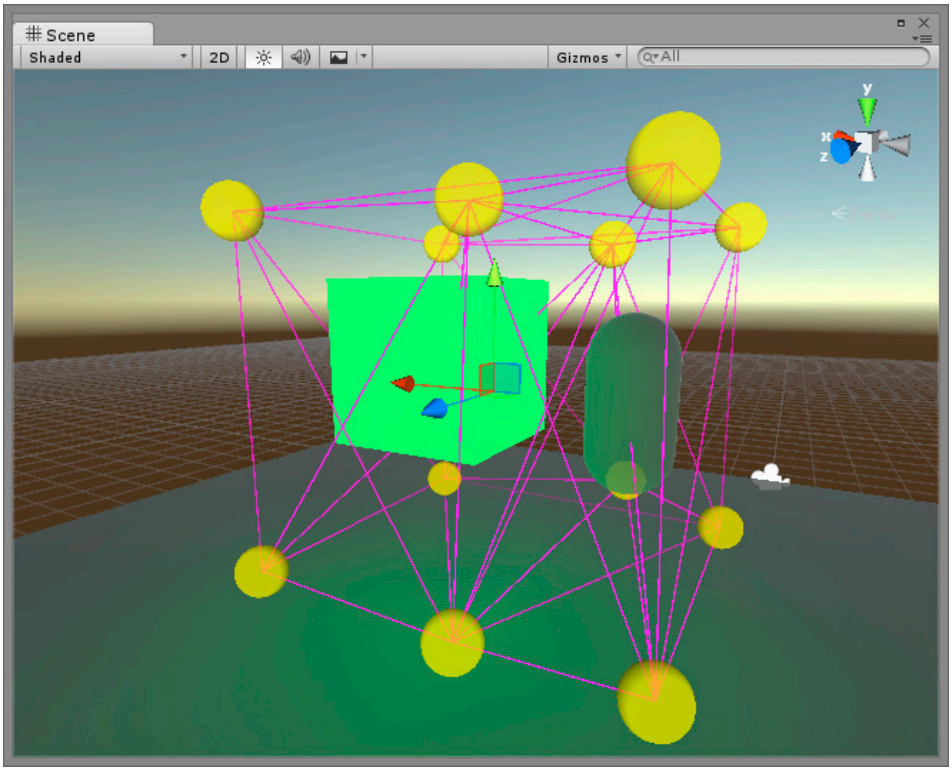

Bild 7.17 Im 3D-Raster angeordnete Light Probes

Light Probes fügen Sie Ihrer Szene über GAMEOBJECT/LIGHT/LIGHT PROBE GROUP zu. Wenn Sie dann die *Light Probe Group* in der *Hierarchy* selektieren, können Sie jedes einzelne *Light Probe* separat noch in Ihrer Szene verschieben. Zudem können Sie über ein kleines Menü im *Inspector* zusätzliche *Light Probes* zu der Gruppe hinzufügen als auch existierende löschen.

Am Anfang ist es sinnvoll, *Light Probes* zu positionieren, dass sie wie in Bild 7.17 in einem gleichmäßigem 3D-Gatter angeordnet werden. Später sollten Sie aber darauf achten, die Anzahl soweit es geht zu reduzieren und die Positionen entsprechend zu optimieren, da jedes einzelne *Light Probe* einiges an Speicher kostet. So ist es z. B. empfehlenswert, größere Abstände zwischen den *Light Probes* zu halten, bei denen es keine großen Unterschiede in der Beleuchtung gibt. Sind die Unterschiede sehr stark, sollten sie wiederum näher positioniert werden.

Zum Testen der *Light Probe*-Positionen können Sie ein beliebiges nichtstatisches Objekt in Ihrer Szene selektieren und dieses verschieben. Ihnen wird dabei nicht nur die spätere Beleuchtung angezeigt, es wird Ihnen auch grafisch dargestellt, welcher Bereich Ihrer *Light Probe Group* aktuell zum Berechnen der Ausleuchtung herangezogen wird (siehe Bild 7.18). Dabei können Sie zum einen an den *Light Probes* erkennen, welche Lichtinformationen diese besitzen (also von wo welches Licht auf diese trifft), zum anderen zeigt der gelb markierte Bereich die *Light Probes*, die die Berechnung beeinflussen. Beachten Sie hierbei, dass die *Light Probes* selber nur dann gelb dargestellt werden, wenn Sie die *Light Probe Group* auch in der *Hierarchy* selektieren.

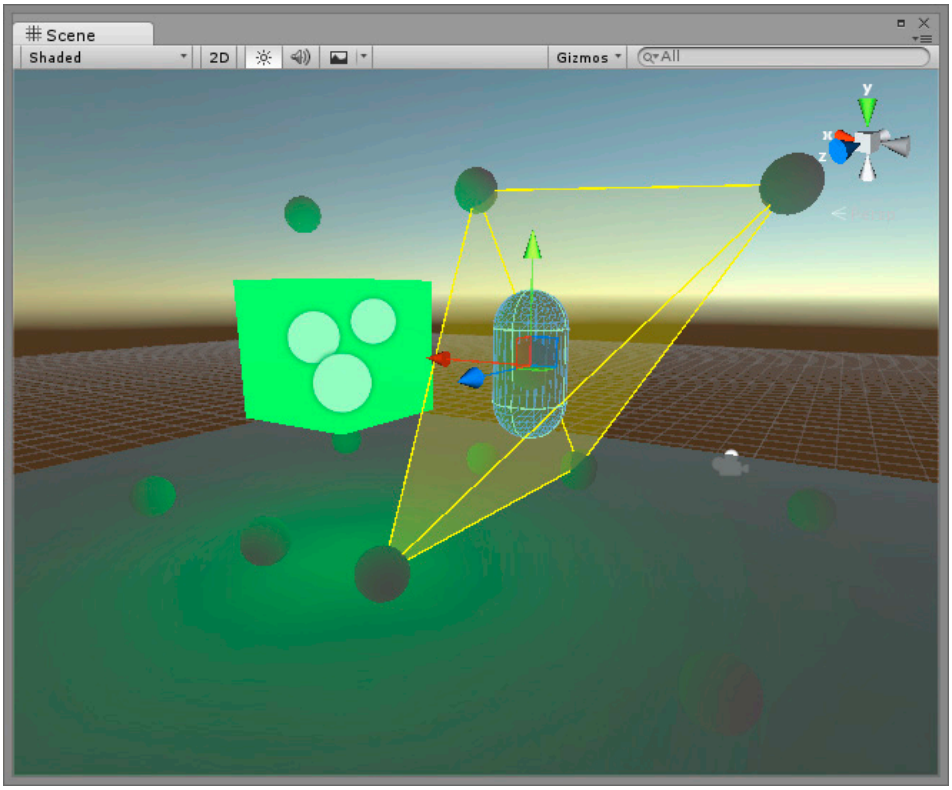

Bild 7.18 Mit Light Probes ausgeleuchtetes Objekt

Umso mehr sich die bewegliche Kapsel in Bild 7.18 einem der Light Probes annähert, desto stärker ist dessen Einfluss auf die Ausleuchtung des Objektes. Das bedeutet, dass ein im Zentrum des gelb markierten Bereiches befindliches Objekt von allen vier Light Probes gleichermaßen beeinflusst wird.

7.9 Rendering Paths

Um Objekte mit Licht und Schatten ansehnlich darzustellen, ist die Wahl des richtigen Rendering-Verfahrens ein wichtiger Punkt. Das Rendern berechnet hierbei das Bild, das am Ende auf dem Bildschirm des Spielers dargestellt wird.

Unity unterstützt hier gleich drei unterschiedliche *Rendering*-Techniken, die ich im Folgenden noch weiter vorstellen werde. Sie können für jede Plattform individuell eine Default-Rendering-Technik definieren. Dies machen Sie in den *Player Settings* (**PROJECT SETTINGS/PLAYER**) im Bereich *Other Settings*. Zusätzlich können Sie noch einmal bei jeder Kamera ein zu nutzendes *Rendering*-Verfahren hinterlegen. Hierfür nutzen Sie die Eigenschaft *Rende-*

ring Path. Wird hier die Default-Einstellung „Use Player Settings“ belassen, wird die Einstellung aus den *Player Settings* übernommen.

Ein Hauptunterschied dieser Rendering-Verfahren sind die eingesetzten Methoden zum Berechnen der Beleuchtung.

- Beim **Vertex Lighting** wird die Auswirkung einer Lichtquelle lediglich anhand der *Vertices* der beleuchteten 3D-Modelle bzw. der *Meshes* berechnet und auf die gesamten Flächen hochgerechnet.
- Beim **Pixel Lighting** wird die Beleuchtung für jeden einzelnen Pixel separat berechnet. Diese Vorgehensweise ist natürlich ressourcenhungriger, ermöglicht aber z. B. Normal-Mapping oder auch Echtzeitschatten. Allerdings sollten Sie beachten, dass einige ältere Grafikkarten *Pixel Lighting* nicht unterstützen.

7.9.1 Forward Rendering

Forward Rendering ist ein Misch-Rendering-Verfahren, das verschiedene Berechnungsarten der Lichtquellen nutzt. In den *Quality Settings* (EDIT/PROJECT SETTINGS/QUALITY) können Sie für jede Plattform über den Parameter *Pixel Light Count* bestimmen, wie viele Lichtquellen im *Pixel Rendering*-Verfahren berechnet werden dürfen. Die anderen Lichtquellen werden mit dem *Vertex Lit*-Verfahren oder als *Spherical Harmonics* berechnet. Letzteres ist ein Verfahren, das ebenfalls auf Basis von *Vertices* arbeitet und aufgrund von Näherungen sehr schnell berechnet werden kann.

Unity wählt bei diesem Verfahren abhängig vom *Pixel Light Count*-Wert selbstständig die wichtigsten Lichtquellen aus und rendert diese dann entsprechend im *Pixel-Lighting-Modus*. Dabei zählt das hellste *Directional Light* automatisch zu den wichtigen.

Sie können aber auch die Auswahlmöglichkeiten selber bestimmen. Stellen Sie den *Render Mode* einer Lichtquelle auf *Important*, so wird diese per Pixel berechnet. Stellen Sie diese auf *Not Important*, wird sie auf jeden Fall per Vertex oder als *Spherical Harmonics* berechnet. Nur bei *Auto* wählt Unity selbstständig.

Beim *Forward Rendering* werden neben der hinterlegten Anzahl an Per-Pixel-Lichtern noch bis zu vier Lichter nach dem *Vertex Lighting*-Verfahren berechnet. Der Rest wird schließlich als *Spherical Harmonics* kalkuliert.

In Bild 7.19 sehen Sie eine kleine Szene, die mit dem *Forward Rendering*-Verfahren dargestellt wird. Sie besitzt zwei Lichtquellen, wobei in den *Quality Settings* als *Pixel Light Count* ein Wert von 1 hinterlegt wurde. Hierdurch wird nur bei einer Lichtquelle das *Pixel Rendering*-Verfahren eingesetzt, weshalb auch nur ein Schatten zu sehen ist.

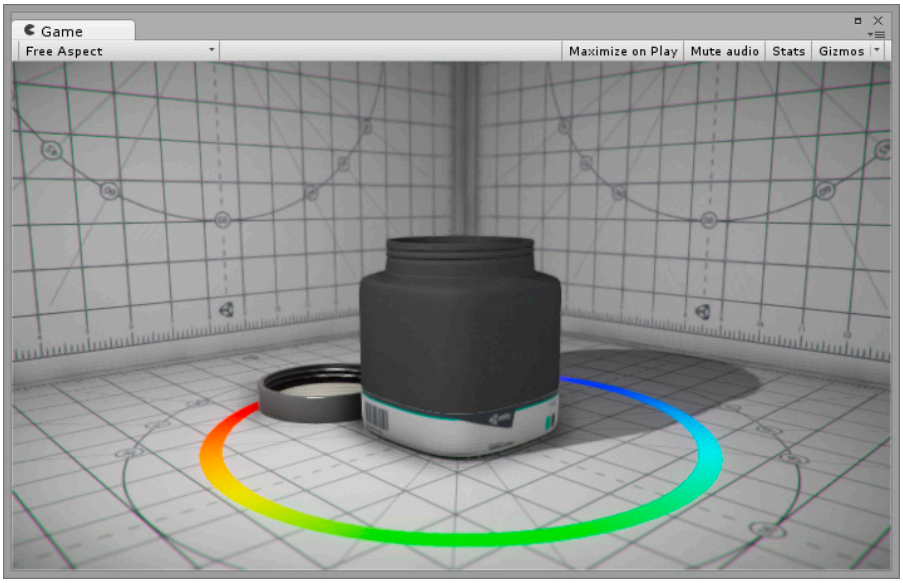

Bild 7.19 Mit Forward Rendering dargestellte Szene

7.9.2 Vertex Lit

Vertex Lit nutzt ausschließlich das *Vertex Lighting*. Es unterstützt keine Echtzeitschatten und auch keine *Shader*-basierten Effekte wie *Normalmaps* oder Echtzeitschatten. Dafür ist es aber mit Abstand am performantesten und bietet die umfassendste Hardwareunterstützung. Allerdings wird es nicht von Konsolen unterstützt.

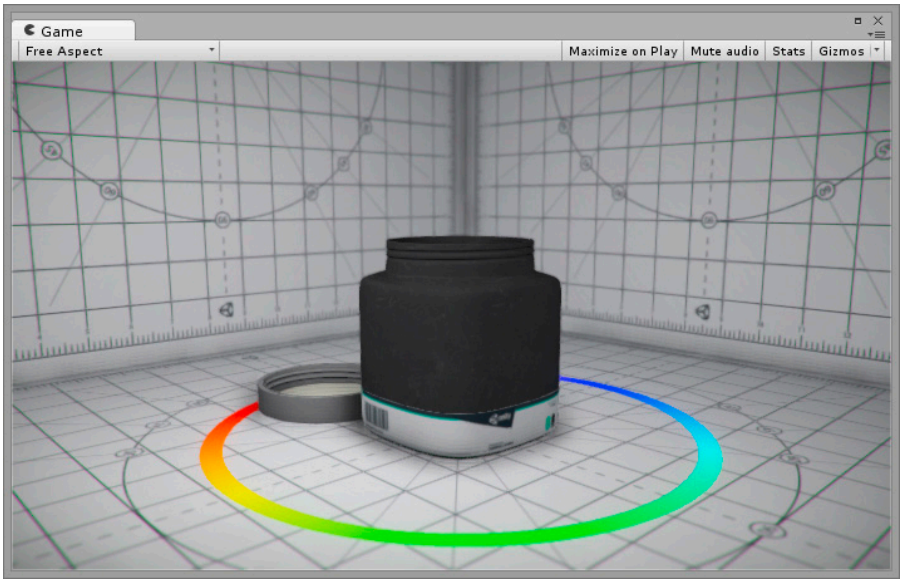

Bild 7.20 Gerenderte Szene mit Vertex Lit

Das Bild 7.20 zeigt eine Szene mit zwei Lichtquellen und ein Material, das zur Darstellung einer rauen Oberfläche eine *Normalmap* nutzt. Aufgrund des *Vertix Lit*-Verfahrens werden sowohl die Schatten der Lichtquellen als auch der Effekt der *Normalmap* nicht dargestellt.

7.9.3 Deferred Lighting

Deferred Lighting ist die anspruchsvollste Rendering-Art mit den detailreichsten Darstellungen von Schatten und Licht, da alle Lichtquellen nach dem Pixel Lighting-Verfahren berechnet werden. Im Gegensatz zum *Forward Rendering* unterstützt es deshalb auch beliebig viele Lichtquellen mit Echtzeitschatten, was sich aber natürlich auch im Performance-Bedarf bemerkbar macht. Außerdem wird dieses Verfahren nicht von jedem Gerät unterstützt, weshalb gerade im Mobile-Bereich die Verwendung vorher genauer geprüft werden sollte.

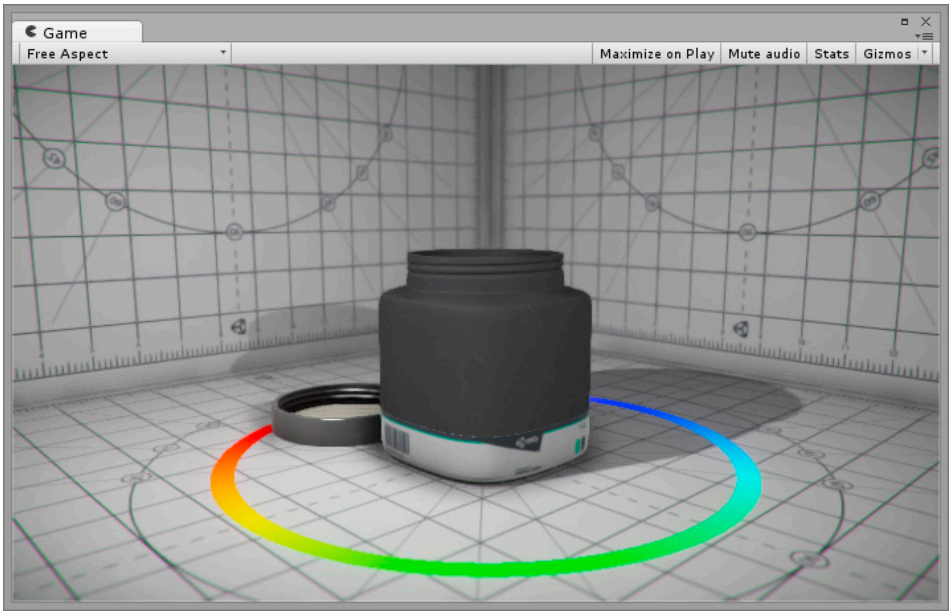

Bild 7.21 Gerenderte Szene mit Deferred Lighting

In Bild 7.21 sehen Sie eine Szene mit zwei Lichtquellen. Aufgrund des gewählten *Deferred Lighting*-Verfahrens wird bei beiden Lichtquellen das *Pixel Rendering*-Verfahren genutzt, sodass auch zwei Schatten zu sehen sind. *Normalmaps* und andere *Shader*-Effekte werden ebenfalls dargestellt.

Neben den höheren Performancekosten hat das *Deferred Lighting* noch einen weiteren Nachteil, den Sie aber ausgleichen können. Im Gegensatz zum *Forward Rendering* wird bei diesem Verfahren kein hardwareseitiges *Antialiasing*,, also keine Kantenglättung, unterstützt. Stattdessen müssen Sie hier der Kamera den „Antialiasing“-*Image Effect* zufügen, den Sie in den Standard Assets „Effects“ finden.

7.10 Global Illumination

In der realen Welt erreichen Lichtstrahlen nicht nur die Stellen, wo sie auf direktem Wege hinkommen. Sie werden auch von den Flächen reflektiert, auf die sie stoßen, und gelangen so über Umwege auch an versteckte Stellen. Ansonsten würde es z. B. an einem Sommertag im Schatten komplett schwarz sein, was es aber nun mal nicht ist.

In Unity nennt sich dieses Verfahren zur indirekten Beleuchtung *Global Illumination*. Wie intensiv die indirekte Beleuchtung dabei ist, können Sie bei jeder Lichtquelle separat einstellen. Dort regeln Sie über den *Bounce Intensity*-Parameter, wie stark das Licht von einer Fläche abgestrahlt wird und die Umgebung erhellt. Das Bild 7.22 zeigt Ihnen, wie durch die reflektierende (indirekte) Beleuchtung eines Spotlights auch der Dachkasten und die Seitenbalken eines Holzhauses angeleuchtet werden.

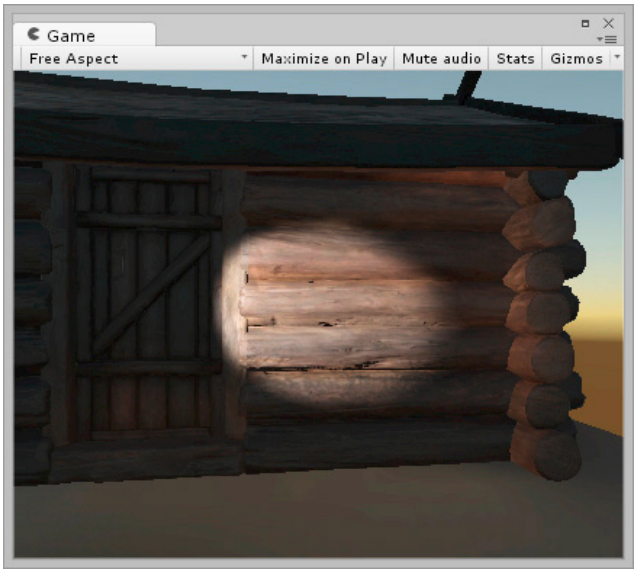

Bild 7.22
Indirekte Beleuchtung durch ein Spotlight

Da die Berechnung indirekter Beleuchtung sehr rechenintensiv sein kann, gibt es in Unity zwei unterschiedliche Verfahren, die *Global Illumination* in Spielen ermöglichen: *Baked GI* und *Precomputed Realtime GI*.

7.10.1 Baked GI

Die erste *Global Illumination*-Variante wird beim *Lightmapping* genutzt und wird auch *Baked GI* genannt. Hier wird die Ausbreitung der indirekten Beleuchtung von entsprechend markierten Lichtquellen zur Entwicklungszeit berechnet und über die Objekte gelegt. Die *Baking*-Eigenschaft der Lichtquellen muss deshalb auf *Mixed* oder *Baken* gestellt sein. Allerdings werden bei diesem Verfahren nur statische Objekte berücksichtigt, die mit *Static* oder *Lightmap Static* gekennzeichnet wurden.

Im *Lighting-Fenster* (WINDOW/LIGHTING) gibt es für *Baked GI* einen extra Bereich, in dem Sie für dieses Verfahren einige Grundeinstellungen vornehmen und auch die komplette Funktion deaktivieren können. Dort finden Sie auch die *Ambient Occlusion*-Eigenschaft, mit der Sie den Lichteinfluss in verdeckten Stellen wie z. B. Innenecken einschränken und kleine Schatten erzeugen können. Beachten Sie, dass Sie kein *Lightmapping* machen können, wenn Sie *Baked GI* deaktiviert haben.

7.10.2 Precomputed Realtime GI

Das zweite *Global Illumination*-Verfahren ist das sogenannte *Precomputed Realtime GI*, also vorberechnetes Echtzeit-GI. Auch hier werden nur statische Objekte berücksichtigt. Allerdings wird hier dieses Mal alles zur Laufzeit berechnet, sodass Sie jede Lichtquelle mit den *Baking*-Modus *Realtime* auch während des Spiels verschieben oder anderweitig ändern können, und trotzdem funktioniert die indirekte Beleuchtung. Allerdings ist diese Variante nicht ganz so detailliert wie *Baked GI* und natürlich auch nicht ganz so performance-freundlich. Beachten Sie deshalb, dass bei sehr großen Beleuchtungsänderungen die Berechnungen auch mal über mehrere Frames dauern können.

Diese Funktion ist zu Anfang eines Projektes genauso aktiviert wie *Baked GI*. Dies macht auch Sinn, weil auf diese Weise sowohl Realtime- als auch in *Lightmaps* integrierte Lichtquellen beim GI berücksichtigt werden. Möchten Sie *Precomputed Realtime GI* allerdings deaktivieren, können Sie dies im *Lighting*-Fenster machen, wo Sie auch noch einige weitere Einstellmöglichkeiten zu diesem Verfahren finden.

7.11 Reflexionen (Spiegelungen)

Möchten Sie in Unity Reflexionen bzw. Spiegelungen darstellen, gibt es hierfür verschiedene Verfahren.

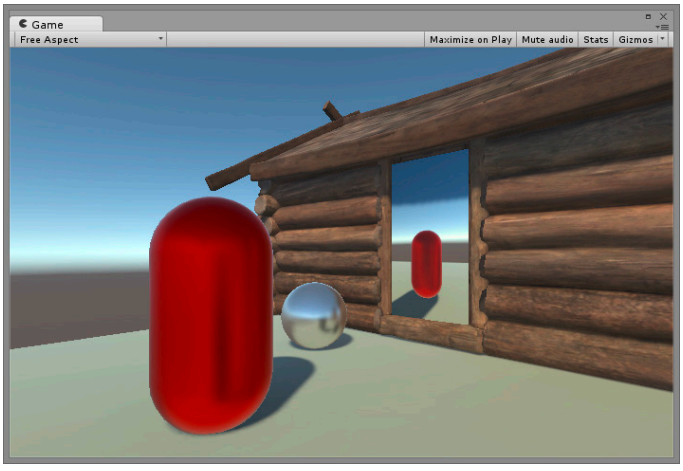

Bild 7.23
Szene mit einem Spiegel und anderen spiegelnden Objekten

Wenn ein korrekte Spiegelungsverhalten wichtig ist, wie z. B. bei einem Spiegel oder einer Wasseroberfläche, werden hierzu *Render Textures* eingesetzt. In dem Fall wird eine Kamera vor dem spiegelnden Objekt in die entgegengesetzte Richtung platziert und stellt dann auf der *Render Texture* das Kamerabild dar. Tipps zum Nutzen von *Render Textures* für Spiegel erhalten Sie im Abschnitt „Render Texture“ von Kapitel 6 „Kameras, die Augen des Spielers“.

Bei Objekten, wo kein 100 % korrektes Bild notwendig ist, wie z. B. bei einem Metallbecher oder den Radkappen eines Autos, ist dieses Verfahren natürlich zu aufwendig und auch zu rechenintensiv. Deshalb werden bei solchen Anwendungszwecken einfach dreidimensionale Bilder genutzt, die statt einer echten Reflexion auf diesen Objekten als Spiegelung dargestellt werden.

Per Default wird dabei die prozedural erzeugte Skybox der aktuellen Szene auf den Objekten dargestellt. Sie können aber auch im *Lighting*-Fenster ein individuelles Bild in Form einer *Cubemap* festlegen, das stattdessen standardmäßig genutzt wird.

Dies können Sie über den *Reflection Source*- Parameter im „Environment Lighting“-Bereich des *Lighting*-Fensters einstellen. Dort können Sie auch weitere Grundeinstellungen für Reflexionen vornehmen.

Damit Unity die dort hinterlegte *Reflection Source* automatisch bei reflektierenden Materialien nutzt, um die Spiegelungen/Reflexionen darzustellen, brauchen Sie nichts weiter zu machen. Das Einzige, was Sie tun müssen, ist, beim *Shader* eines Materials zu definieren, wie das Reflexionsverhalten sein soll, also wie glatt eine Oberfläche ist und wie metallisch sie wirken soll (siehe „Der Standard-Shader“ im Kapitel 5 „Objekte in der zweiten und dritten Dimension“).

Neben dieser ganz grundsätzlichen Reflexionsvorlage können Sie aber auch noch individuelle Reflexions-*Cubemaps* auf den Objekten darstellen. Hierbei kommen sogenannte *Reflections Probes* zum Einsatz, die ich im Folgenden etwas genauer erläutere.

7.11.1 Reflection Probes

Mit *Reflection Probes* können Sie raumabhängige Reflexions-*Cubemaps* erzeugen, die dann automatisch den Materialien zugewiesen werden, die sich in deren Umgebung befinden. Die auf diese Weise generierten *Cubemaps* können sowohl statisch sein als auch in Echtzeit ihre Inhalte aktualisieren, sodass sich die Reflexionen auf den Materialien sogar während des Spiels ändern können.

Reflection Probes sind kugelförmige Gebilde (siehe Bild 7.24), die im Grunde eine 360°-Kamera darstellen, die ihre komplette Umgebung in Bildform festhält. Dabei können Sie über den *Type*-Parameter definieren, zu welchem Zeitpunkt und wie oft dieses Bild generiert wird.

- **Baken** legt fest, dass das Bild zur Entwicklungszeit während des *Lightmapping-Bakens* erstellt wird. Hierbei werden nur *statische* Objekte im Bild erfasst, bei denen Static oder *Reflection Probe Static* aktiviert ist.
- **Realtime** bedeutet, dass das Erstellen des Bildes zur Laufzeit des Spiels stattfindet.
- **Custom** ermöglicht, eine eigene *Cubemap* diesem *Reflection Probe* zuzuweisen, die dann in dem Einflussbereich dieses Reflection Probes als Reflexion dargestellt wird. Alternativ

können Sie über einen **BAKE**-Button eine *Cubemap* erstellen. Über die zusätzliche *Dynamic Objects*-Option können Sie zudem auch die nichtstatischen Objekte mit in die *Cubemap* einfließen lassen.

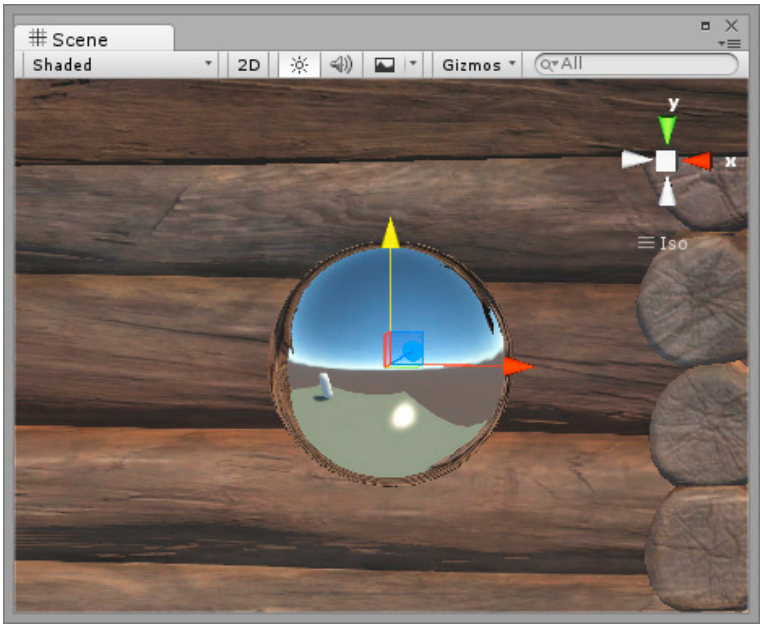

Bild 7.24 Light Probe

Wählen Sie die *Type*-Option *Realtime*, stehen Ihnen folgende Einstellmöglichkeiten zur Wahl:

- **On Awake** erstellt das Bild einmal, wenn der *Reflection Probe* erstmalig in einer Szene aktiv wird.
- **Every Frame** bewirkt, dass der Probe wie bei einer Kamera in jedem Frame das Bild erneut erstellt.
- **Via Scripting** bedeutet, dass das Bild nur dann erstellt wird, wenn per Code die Methode `RenderProbe` eines *Reflection Probes* ausgelöst wird.

Das Listing 7.1 zeigt Ihnen ein Beispiel, wie Sie bei der *Type*-Option *Via Scripting* das Erstellen eines neuen *Reflection Probe*-Bildes auslösen.

Listing 7.1 Per Code Reflection Probe-Bild erstellen

```
using UnityEngine;
using System.Collections;
public class RefreshReflectionProbe : MonoBehaviour {
  public ReflectionProbe probe;

  void OnTriggerEnter()
  {
    probe.RenderProbe();
  }
}
```

Um das Skript aus Listing 7.1 zu nutzen, weisen Sie dieses einem *Trigger-Collider* zu. Nun brauchen Sie nur noch den *Reflection Probe* auf die Variable `probe` zu ziehen und schon wird das Bild der *Reflection Probes* aktualisiert, sobald sich ein Objekt mit einem Collider (und einem Rigidbody) in diesen hineinbewegt. Das Bild 7.25 stellt es noch einmal bildlich dar.

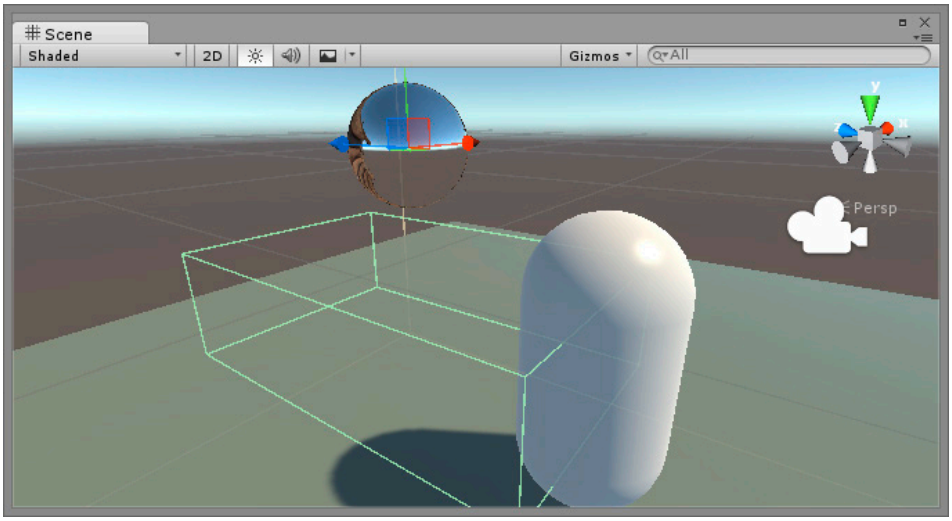

Bild 7.25 Per Trigger-Collider einen Reflection Probe aktualisieren

Wie eingangs erwähnt, fangen *Reflection Probes* die eigene Umgebung ein, um diese dann als Reflexionen auf anderen Materialien anzuwenden. Welche Umgebung hierbei eingefangen wird, wird durch eine gelbe Box gekennzeichnet.

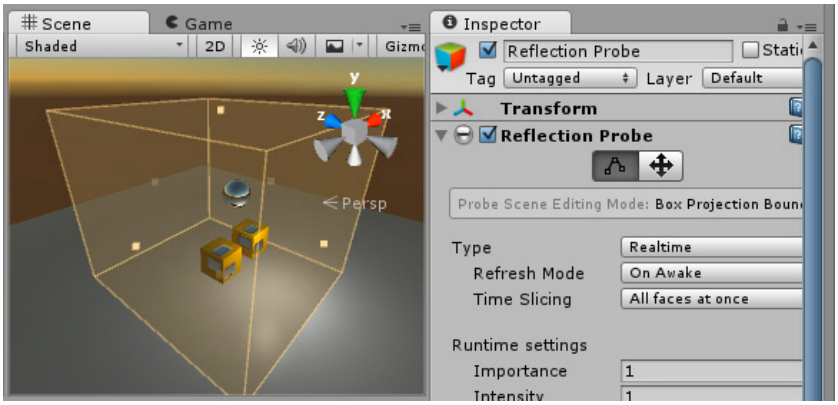

Bild 7.26 Reflection Probe-Box mit aktiviertem Size-Werkzeug

Mit dem *Size*-Tool können Sie nun diese Box modifizieren. Nutzen Sie hierfür die *Handle*-Punkte an den Flächen der Box (siehe Bild 7.26). Zum Aktivieren des *Size*-Tools drücken Sie den linken Funktions-Button im *Inspector* der *Reflection Probe*-Komponente. Neben dem *Size*-Tool finden Sie rechts daneben das *Origin*-Tool (gekennzeichnet durch das Pfeile-Kreuz). Mit diesem können Sie den *Reflection Probe* innerhalb der Box verschieben.

7.12 Qualitätseinstellungen

Wie Sie bereits sicher gemerkt haben, bietet Unity Ihnen viele unterschiedliche Einstellungen bezüglich der Hochwertigkeit der grafischen Darstellung.

Die Grundeinstellungen hierfür können Sie in den *Quality Settings* vornehmen, die Sie über EDIT/PROJECT SETTINGS/QUALITY erreichen.

7.12.1 Quality Settings

In den *Quality Settings* haben Sie die Möglichkeit, unterschiedliche Qualitätsstufen zu definieren, die dann je nach Stufe performance-lastiger und hochwertiger bzw. sparsamer und rudimentärer aussehen.

Selektieren Sie hierfür in der dort zu findenden Matrix ein *Quality Level* und nehmen im unteren Bereich dann die Grundeinstellungen für diese Stufe in den Bereichen *Rendering*, *Shadow* und *Other* vor. Über den Button ADD QUALITY LEVEL können Sie zudem der Matrix beliebig viele weitere Levels zufügen und dann definieren.

Außerdem können Sie in der Matrix die *Quality Level* den verschiedenen Plattformen zuordnen bzw. festlegen, welche Qualitätsstufen dort zur Verfügung stehen sollen.

7.12.2 Qualitätsstufen per Code festlegen

Zum Nutzen dieser Qualitätsstufen in Spielen bietet Unity Ihnen die Klasse `QualitySettings` an, die verschiedene statische Methoden bereithält, um z. B. die *Quality Levels* abzufragen oder zu verändern.

Das Listing 7.2 zeigt Ihnen ein einfaches Skript, mit dem die Qualitätsstufen gesenkt oder angehoben werden können. Der Name der aktuellen Stufe wird dabei in einem Text-Objekt der GUI angezeigt. Neben diesem Text-Objekt brauchen Sie lediglich noch zwei Buttons Ihrer Szene zuzufügen, die dann die beiden Funktionen `IncreaseQuality` und `DecreaseQuality` aufrufen. Mehr zu diesem Thema erfahren Sie im Kapitel 14 „GUI“.

Listing 7.2 Einfaches Skript zum Ändern der Qualitätsstufen

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;

public class SetQualityLevel : MonoBehaviour {
  //Text-Objekt zur Anzeige des aktuellen Quality-Level-Namens
  public Text qualityText;
  //Array zum Speichern alle Quality-Level-Namen
  public string[] names;
 void Start() {
    //QualityLevel-Namen abfragen und Array zuweisen
    names = QualitySettings.names;
    //GUI aktualisieren
```

```
    UpdateView();
  }

  public void IncreaseQuality()
  {
    //Qualitaetsstufe anheben
    QualitySettings.IncreaseLevel(true);
    //GUI aktualisieren
    UpdateView();
  }

  public void DecreaseQuality()
  {
    //Qualitaetsstufe senken
    QualitySettings.DecreaseLevel(true);
    //GUI aktualisieren
    UpdateView();
  }
  //Name des aktuellen Quality-Levels in der GUI aktualisieren
  void UpdateView()
  {
    //Quality-Level-Index abfragen, Name aus Array holen und GUI zuweisen
    qualityText.text = names[QualitySettings.GetQualityLevel()];
  }
}
```

Mit einer einfach gehaltenen Oberfläche könnte das Skript in Verbindung mit drei UI-Elementen so aussehen wie in Bild 7.27.

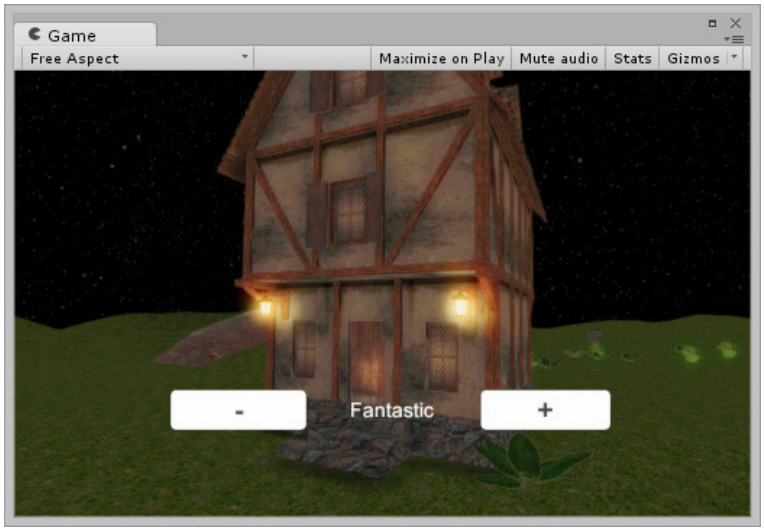

Bild 7.27 Einfache GUI zum Ändern der aktuellen Qualitätsstufe

Index

Symbole

A

B

C

D

E

F

G

H

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z