2

2 Einführung in die 3D-Grafik

2.1 Was steht in diesem Kapitel?

Sie werden zunächst einen Einblick in die grundlegenden Techniken erhalten, die in der 3D-Grafik zum Einsatz kommen.

Danach beschäftigen wir uns intensiv mit dem mathematischen und geometrischen Teil, vor allem mit Vektoren und Matrizen. Das ist unbedingt notwendig, um im nächsten Kapitel die ersten Schritte in Richtung Direct3D wagen zu können. Sie müssen nicht unbedingt jedes Detail verstehen. Einige Zusammenhänge stehen nur aus Gründen der Vollständigkeit in diesem Kapitel und sind nicht unbedingt notwendig, um den Rest des Buches nachvollziehen zu können.

Schließlich behandeln wir die Beleuchtungsberechnung und die Rendering-Pipeline, die alle Schritte von einem Haufen Dreiecke bis hin zum fertigen Bild auf dem Bildschirm beschreibt.

Was in diesem Kapitel steht, ist größtenteils unabhängig von der verwendeten Grafik-API (zum Beispiel Direct3D oder OpenGL). Beachten Sie auch die Übungsaufgaben am Ende des Kapitels!

2.2 Repräsentation und Darstellung einer 3D-Szene

Wir werden uns nun einige Möglichkeiten ansehen, wie man eine 3D-Szene beschreiben kann. Betrachten Sie als einfaches Beispiel eine Szene, die eine Kugel, einen Würfel und eine Pyramide enthält, die auf einer Ebene stehen.

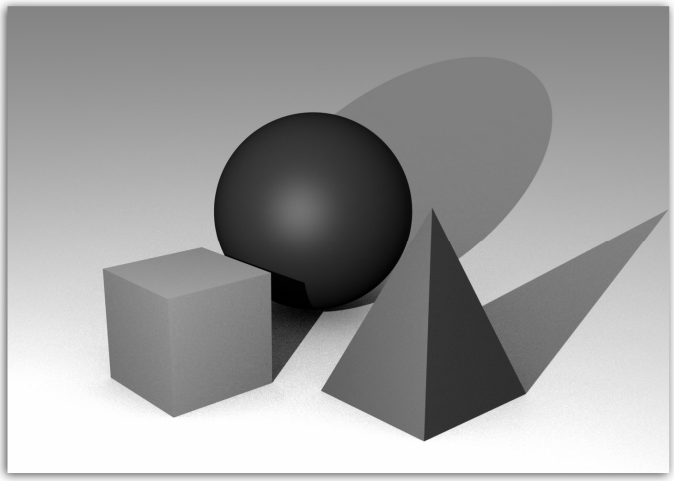


Abbildung 2.1 Beispielszene mit einer Ebene, einer Kugel, einem Würfel und einer Pyramide

Die verschiedenen Körper könnten beispielsweise mit Gleichungen beschrieben werden, die für alle Punkte, die in diesen Körpern liegen, erfüllt sein müssen. Für die primitiven Objekte der Beispielszene sind solche Gleichungen sehr einfach herzuleiten. Aber wie sieht es mit komplexeren Objekten aus, die man aus Computerspielen kennt, wie Flugzeuge, Zombies oder Kettensägen?

Triangulation

Die Beschreibung von ganzen Objekten nur durch mathematische Formeln ist für die Echtzeitgrafik eher weniger interessant, da es viel Aufwand erfordert, sie in ein Bild umzusetzen. Hier hat sich eine Methode durchgesetzt, bei der die Oberflächen der Objekte in Dreiecke unterteilt werden. Diese Technik nennt man *Triangulation*.

Sie werden sich jetzt vielleicht fragen, wie man beispielsweise eine Kugeloberfläche in Dreiecke aufteilen soll, und diese Frage ist völlig berechtigt, denn mit endlich vielen Dreiecken schafft man das nicht. Je mehr Dreiecke man hinzufügt, desto kugelähnlicher wird das Gebilde, aber eine perfekte Kugel kann daraus nie entstehen. Normalerweise benötigt man auch gar keine perfekten Formen, denn irgendwann ist ein Detailgrad erreicht, bei dem der Unterschied selbst aus kürzester Entfernung kaum noch auffällt.

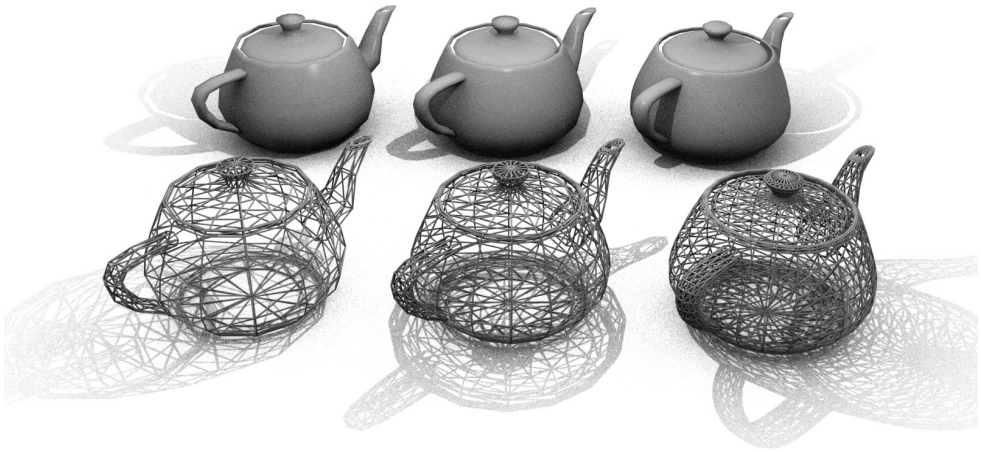


Abbildung 2.2 Die legendäre *Utah-Teekanne*, eines der ältesten und bekanntesten Objekte der Computergrafik, in verschieden detailliert triangulierten Versionen

Warum gerade Dreiecke? Aus Dreiecken können alle anderen Vielecke zusammengesetzt werden, und Dreiecke lassen sich besonders einfach mit dem Computer zeichnen. Das Problem, ein komplexes 3D-Objekt wie zum Beispiel ein Flugzeug zu zeichnen, kann also auf das Problem reduziert werden, viele Dreiecke zu zeichnen.

Rasterung und Raytracing

Und wie zeichnet man nun ein Dreieck? Diese Arbeit werden wir später der Grafikkarte überlassen, da sie ja speziell dafür gebaut wurde, viele Milliarden davon in jeder Sekunde abzuarbeiten. Trotzdem ist es hilfreich zu wissen, wie das ungefähr funktioniert. Wir gehen davon aus, dass wir schon wissen, an welcher Stelle auf dem Bildschirm sich die Eckpunkte des Dreiecks befinden. Wie diese berechnet werden, schauen wir uns später an.

Prinzipiell hat man das Problem, dass man die „unendlich genau“ beschreibbare Szene auf ein Bild mit endlich vielen Bildpunkten abbilden muss, zum Beispiel den Bildschirm, der nur begrenzt viele Pixel darstellen kann. Ein Ansatz ist die *Rasterung*. Dabei werden die darzustellenden Dreiecke gerastert, also auf diskrete Bildpunkte abgebildet. Ein Dreieck kann man dann zeichnen und ausfüllen, indem man es Zeile für Zeile von oben nach unten abläuft und jede Zeile füllt. Eine Zeile entspricht dabei einer Pixelzeile. Dieses Verfahren, das heutige Grafikkarten anwenden, nennt man *Scanline Rendering*.

Der Begriff „Rendern“ wird uns übrigens noch sehr oft begegnen. In der Computergrafik versteht man darunter im Allgemeinen die Erzeugung eines digitalen Bilds. Wenn davon gesprochen wird, „ein Objekt zu rendern“, dann ist damit eigentlich nur gemeint, das Objekt zu zeichnen.

Rasterung steht im Gegensatz zu anderen Verfahren wie zum Beispiel *Raytracing*. Beim Raytracing wird, vereinfacht gesagt, für jeden Pixel des Bilds ein virtueller Strahl („Sichtstrahl“) in die Szene geschossen, und es wird berechnet, ob er ein Objekt schneidet. Wenn

das der Fall ist, wird der Pixel entsprechend eingefärbt. Vom Schnittpunkt aus können weitere Strahlen abgeschossen werden, um Reflexionen, Brechungen oder Schatten einfach berechnen zu können. Raytracing-Verfahren liefern Bilder von sehr hoher Qualität, sind generell sehr flexibel und funktionieren sowohl mit der mathematischen Darstellungsweise von 3D-Szenen als auch mit triangulierten Objekten. Ein Raytracer kann zum Beispiel „perfekte“ Kugeln rendern.

Für Echtzeitanwendungen ist Raytracing allerdings noch nicht gut zu gebrauchen, da es auf heutigen Standardprozessoren noch zu langsam ist. Das Rendern statischer Szenen lässt sich durch Vorberechnungen zwar beschleunigen, aber das funktioniert nicht mehr so gut, sobald sich Objekte bewegen oder verformen können (was bei Spielen im Allgemeinen der Fall ist). Allerdings gibt es bereits spezielle Raytracing-Hardware[1], und es ist als sehr wahrscheinlich anzusehen, dass die Rasterung und das Scanline Rendering langfristig vom Raytracing abgelöst werden.

Materialien, Farben und Texturen

Eine 3D-Welt, die nur aus Eckpunkten von Dreiecken besteht, käme sehr trist daher, da wir damit allein nichts über die Farbgebung aussagen können, sondern nur über die Form. Darum weist man Objekten Materialien zu, die der Oberfläche bestimmte Eigenschaften wie Farbe, Transparenz oder Glanz verleihen oder sie mit einer *Textur* überziehen. Eine Textur ist ein Bild, das über ein Objekt gespannt werden kann, um Oberflächendetails zu simulieren.

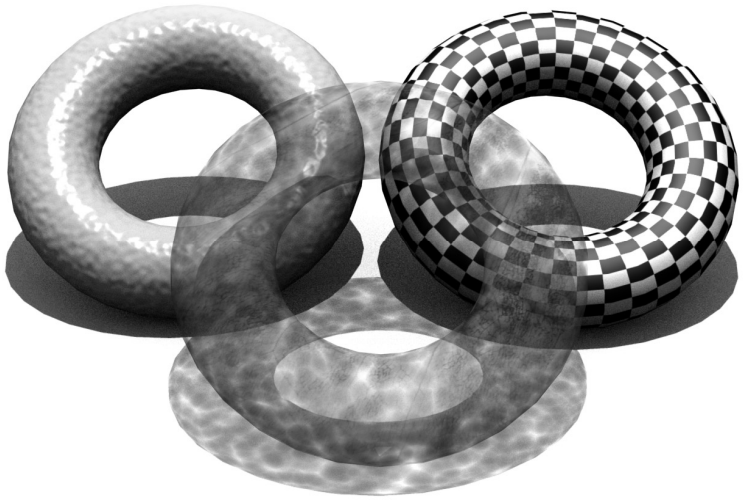


Abbildung 2.3 Die drei Donuts unterscheiden sich nur durch ihre Materialien und Texturen.

[1] Details finden Sie unter http://graphics.cs.uni-sb.de/SaarCOR/.

Beleuchtung

Die Beleuchtung einer Szene gibt dem Betrachter wichtige Hinweise über die räumliche Anordnung der Objekte und kann stark zur Atmosphäre eines Spiels beitragen. Dabei spielen vor allem Schatten eine große Rolle. Man denke beispielsweise an einen schwach beleuchteten Korridor, der um eine Ecke führt. Die auf dem Boden tanzenden Schatten verraten, dass dort ein Monster der übelsten Sorte auf den Spieler wartet …

Es gibt eine Vielzahl von Modellen, mit denen Licht und Schatten berechnet werden können. Man kann sie grob in *globale* und *lokale Beleuchtungsmodelle* unterteilen.

Globale Beleuchtungsmodelle versuchen, die Interaktion aller in einer Szene befindlichen Objekte durch Reflexion und Absorption von Licht zu simulieren. Solche Verfahren sind im Allgemeinen so aufwendig, dass sie in der heutigen Zeit in Echtzeit nur mit massiv parallel arbeitenden Systemen durchgeführt werden können. Hauptsächlich kommen diese Modelle bei computergenerierten Bildern und Filmen zum Einsatz, bei denen die benötigte Rechenzeit zweitrangig ist und es in erster Linie auf die Qualität des Ergebnisses ankommt. Ein einzelnes Bild zu berechnen kann dann schon ein paar Stunden dauern. Mit einigen Tricks, zum Beispiel der Berechnung der Beleuchtung statischer Objekte im Voraus, kann aber auch der Spieleentwickler der heutigen Zeit davon profitieren.

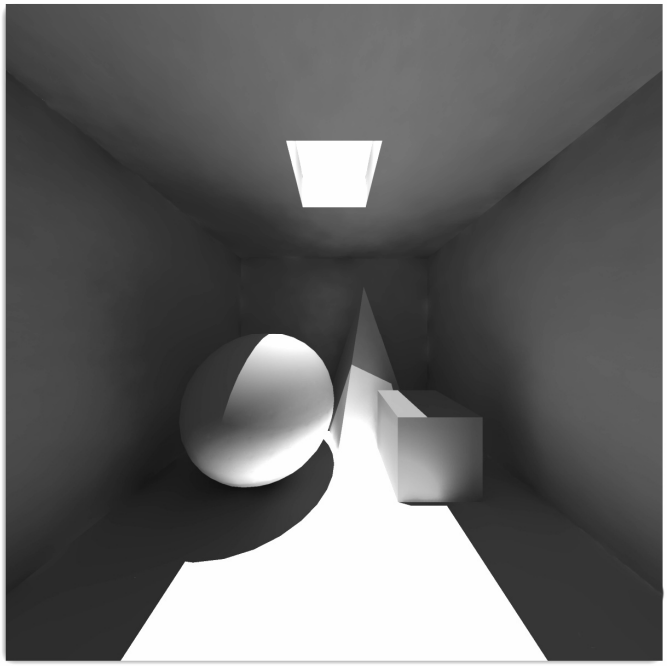


Abbildung 2.4 Globale Beleuchtung: Licht tritt nur durch die Öffnung an der Decke in den Kasten ein, aber indirekt wird auch der Teil der Szene beleuchtet, den die Lichtstrahlen von außen nicht unmittelbar erreichen.

Lokale Beleuchtungsmodelle berechnen die Beleuchtung für eine Oberfläche, ohne dabei die indirekte Beleuchtung durch andere Objekte in der Szene zu berücksichtigen. Dadurch sind sie wesentlich einfacher zu implementieren und arbeiten schneller.

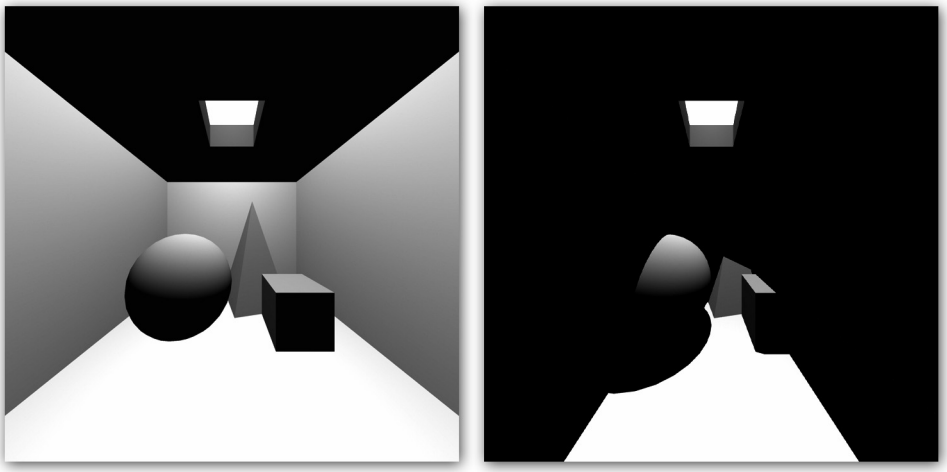


Abbildung 2.5 Lokale Beleuchtung ohne und mit Schatten

In der Praxis setzt man meistens eine Mischung aus globalen und lokalen Beleuchtungsmodellen ein und versucht durch möglichst viele Annäherungen das beste optische Ergebnis zum geringsten Preis (Rechenzeit, Speicherbedarf) zu erzielen.

2.3 Vektoren

Nun gehen wir weiter ins Detail und stellen uns die Frage, wie wir Punkte und Richtungen im zwei- und dreidimensionalen Raum darstellen können, und wie wir mit ihnen rechnen können. Dazu verwendet man *Vektoren*.

Es gibt eine sehr allgemeine mathematische Definition eines Vektors und eine sehr anschauliche geometrische Definition. Die mathematische Definition besagt, dass ein Vektor ein Element eines Vektorraums ist (so wie beispielsweise die Zahl $\sqrt{2}$ ein Element der reellen Zahlen $\mathbb{R}$ ist). Vektoren können miteinander und mit Skalaren (Zahlen) verknüpft werden, und das Ergebnis dieser Operation ist wieder ein Element des Vektorraums. Dies nennt man Abgeschlossenheit. Es gibt noch ein paar weitere Anforderungen, die an einen Vektorraum gestellt werden, aber darauf wollen wir hier nicht näher eingehen. Für uns ist besonders die geometrische Betrachtungsweise interessant.

2.3.1 Vektoren als Pfeile

Einen Vektor kann man sich als Pfeil mit einer bestimmten Länge und Richtung. Alle Pfeile, die in diesen zwei Eigenschaften übereinstimmen, stellen denselben Vektor dar.

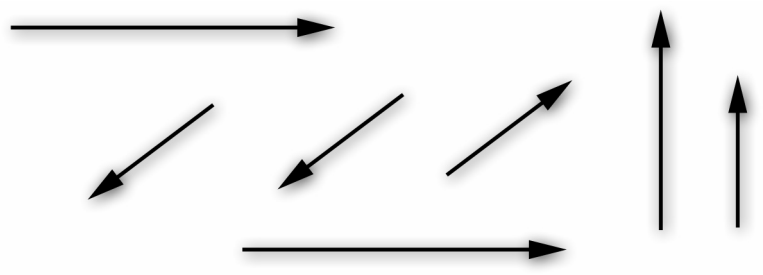


Abbildung 2.6 Sieben Pfeile, die fünf verschiedene Vektoren darstellen

Diese Pfeile kann man sich in jeder beliebigen Dimension vorstellen. Die Abbildung zeigt Pfeile im zweidimensionalen Raum, aber natürlich funktioniert das auch im Dreidimensionalen.

Ein spezieller Vektor ist der *Nullvektor*, der auch mit $\vec{0}$ bezeichnet wird. Vektorvariablen kennzeichnet man für gewöhnlich mit einem kleinen Pfeil. Der Nullvektor hat die Länge null und keine Richtung. Man würde sich also schwer tun, ihn als Pfeil zu zeichnen.

Vektoren mit derselben Länge, die in entgegengesetzte Richtungen zeigen, heißen *Gegenvektoren*. In der Abbildung sind die beiden mittleren Vektoren Gegenvektoren. Den Gegenvektor von $\vec{a}$ schreibt man auch $-\vec{a}$.

Hat ein Vektor die Länge 1, dann nennt man ihn auch *Einheitsvektor*. Im Gegensatz zum Nullvektor gibt es unendlich viele Einheitsvektoren.

Rechnen mit Pfeilen

Mit der Pfeildarstellung kann man bereits rechnen. Zwei Vektoren $\vec{a}$ und $\vec{b}$ werden addiert, indem man den Anfang von $\vec{b}$ an die Spitze von $\vec{a}$ legt. Der Pfeil, den man dann vom Anfang von $\vec{a}$ zur Spitze von $\vec{b}$ zeichnen kann, stellt $\vec{a}+\vec{b}$ dar. Da die Addition kommutativ ist ($\vec{a}+\vec{b}=\vec{b}+\vec{a}$), funktioniert das natürlich auch umgekehrt. Der Nullvektor ist das neutrale Element der Addition, das heißt die Addition des Nullvektors zu einem anderen Vektor verändert diesen nicht, so wie die Addition von Null oder die Multiplikation mit der Eins bei Zahlen ebenfalls wirkungslos ist.

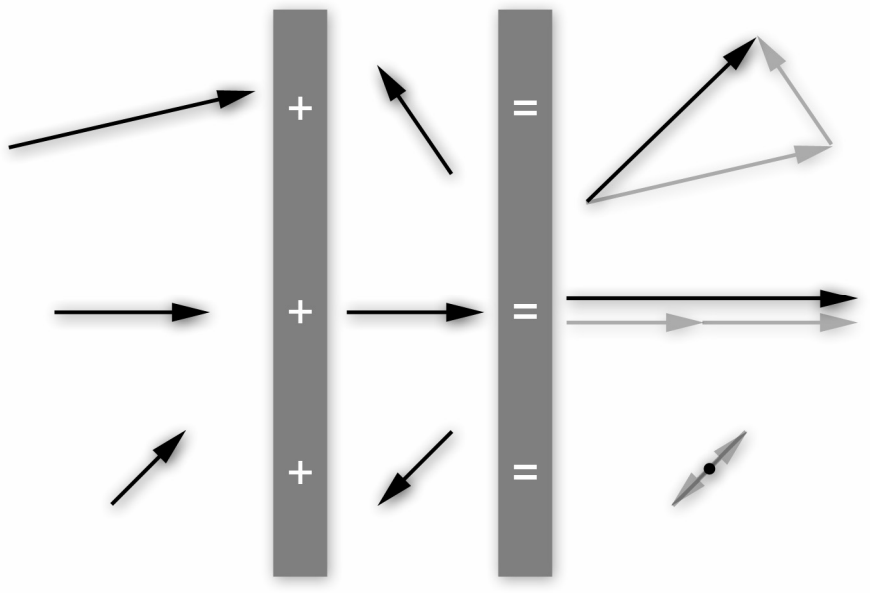


Abbildung 2.7 Addition von Vektoren durch Aneinanderlegen von Pfeilen: unten wird ein Vektor mit seinem Gegenvektor addiert. Es ergibt sich der Nullvektor.

Natürlich kann man Vektoren auch voneinander subtrahieren. Um $\vec{a} - \vec{b}$ zu berechnen, addiert man einfach den Gegenvektor von $\vec{b}$ zu $\vec{a}$, rechnet also $\vec{a} - \vec{b} = \vec{a} + \left(-\vec{b}\right)$.

Es gibt noch eine andere Rechenoperation, die für einen Vektor definiert sein muss. Dies ist die Multiplikation mit einem Skalar, also einer Zahl. Ein Vektor muss mit einer Zahl multipliziert werden können, und das Ergebnis muss wieder ein Vektor sein.

Auch die Multiplikation mit Skalaren lässt sich für Pfeile sinnvoll umsetzen. Multiplizieren wir einen Vektor beispielsweise mit der Zahl 2, dann verdoppeln wir seine Länge (die Richtung bleibt gleich). Bei der Multiplikation mit $\frac{1}{2}$ wird die Länge halbiert.

Was ist mit negativen Zahlen? Die Länge kann natürlich nicht negativ werden, aber die Richtung kann umgedreht werden. Wird ein Vektor zum Beispiel mit $-\frac{1}{4}$ multipliziert, dann wird die Länge geviertelt und die Richtung umgedreht.

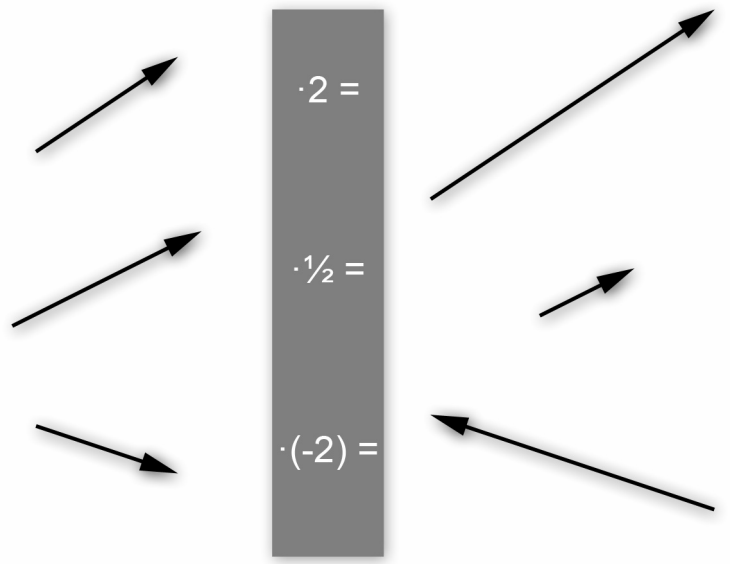


Abbildung 2.8 Vektoren in Pfeildarstellung werden mit Skalaren multipliziert, wobei sich ihre Länge ändern und die Richtung umgedreht werden kann.

2.3.2 Die Koordinatendarstellung

Pfeile sind zwar sehr anschaulich und eignen sich gut für Abbildungen in Büchern, aber es lässt sich mit ihnen im Computer schlecht rechnen, und wir werden später sehr viele Rechnungen mit Vektoren durchführen. Darum ist eine Darstellungsweise sinnvoll, die nur mit Zahlen auskommt. Dazu definieren wir uns ein *Koordinatensystem*.

Das Koordinatensystem hat einen *Koordinatenursprung* (dies ist ein Punkt) und *n Koordinatenachsen* (dies sind Richtungen). *n* steht für die Dimension des Raums, in dem wir arbeiten möchten (zum Beispiel $n = 3$ im Dreidimensionalen). Die Achsen schneiden sich im Ursprung. Wenn sie alle orthogonal (senkrecht) zueinander sind, dann spricht man von einem *kartesischen Koordinatensystem*.

Aus der Schule sollte jeder das zweidimensionale kartesische Koordinatensystem kennen. Dies ist das am häufigsten verwendete Koordinatensystem, da es sehr intuitiv ist. Seine Achsen werden üblicherweise als *x*- und *y*-Achse bezeichnet, wobei die *x*-Achse nach rechts und die *y*-Achse nach oben zeigt.

Das zweidimensionale kartesische Koordinatensystem lässt sich denkbar einfach auf drei Dimensionen erweitern, indem man eine dritte Achse, die *z*-Achse einführt, die senkrecht zu den beiden anderen Achsen ist und in die Tiefe zeigt. (Natürlich ist dann die grafische Darstellung auf einem Blatt Papier nicht mehr eindeutig)

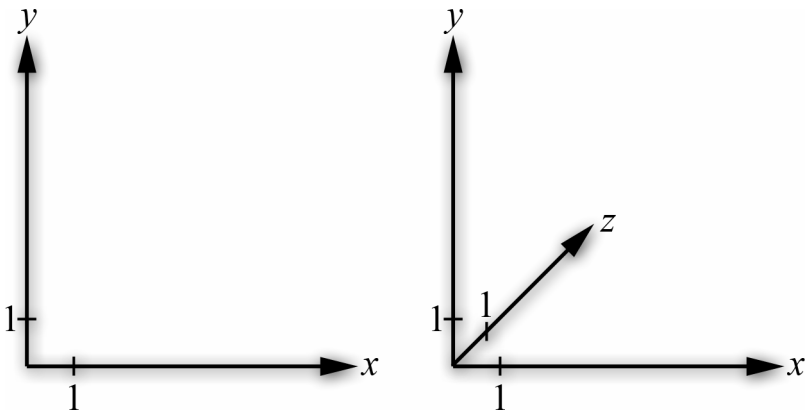


Abbildung 2.9 Zwei- und dreidimensionales kartesisches Koordinatensystem

Wir werden von nun an immer das hier gezeigte 3D-Koordinatensystem verwenden, bei dem die *x*-Achse nach rechts, die *y*-Achse nach oben und die *z*-Achse in die Tiefe zeigt. Es wird auch als *linkshändiges Koordinatensystem* bezeichnet. Wir wählen dieses Koordinatensystem, da es bei Direct3D der Standard ist.

Man kann die Anordnung der drei Achsen mit dem Daumen (*y*), dem Zeigefinger (*z*) und dem Mittelfinger (*x*) der linken Hand nachbilden. Alternativ kann man sich vorstellen, das Koordinatensystem wie eine Schraube zu drehen, und zwar von *x* nach *y*. Eine Schraube mit Rechtsgewinde wird sich in einem rechtshändigen Koordinatensystem in Richtung *z* bewegen.

Die Vektoren, die entlang den Achsen verlaufen, nennt man auch *Basisvektoren*.

Mit einem Koordinatensystem, auf das wir uns beziehen, können wir nun Vektoren nur mit Hilfe von Zahlen beschreiben. Diese Zahlen werden *Komponenten* genannt. Man kann sie auf verschiedene Arten darstellen, zum Beispiel so:

$$\vec{a} = (a_1,\ a_2,\ a_3) \text{ oder } \vec{a} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

Hier sind a_1, a_2 und a_3 die Komponenten des Vektors $\vec{a}$. Es ist also ein dreidimensionaler Vektor, da es drei Stück sind. Es ist üblich, die Komponenten mit demselben Buchstaben wie die Vektorvariable zu bezeichnen, aber ohne den Vektorpfeil und mit tiefgestelltem Index, der die Nummer der Komponente angibt. Da wir die Komponenten als Koordinaten betrachten, werde ich im Folgenden auch von den *Koordinaten eines Vektors* sprechen statt von den Komponenten.

Im Prinzip ist es egal, in welcher der beiden Formen man einen Vektor schreibt. Die Darstellung als Zeile (links) eignet sich besser zum Abdrucken innerhalb eines Texts, dafür sieht die Spaltendarstellung schöner aus.

Wie erhält man aus den Koordinaten nun wieder einen Vektor? Das ist relativ einfach. Wenn $\vec{x}$, $\vec{y}$ und $\vec{z}$ die Basisvektoren des Koordinatensystems sind, auf das sich die Koordinaten beziehen, dann erhält man den ursprünglichen Vektor durch $a_1 \cdot \vec{x} + a_2 \cdot \vec{y} + a_3 \cdot \vec{z}$. Das heißt: wir gehen a_1 Einheiten entlang der *x*-Achse, dann a_2 Einheiten entlang der *y*-Achse und zum Schluss a_3 Einheiten entlang der *z*-Achse. Natürlich kann die Reihenfolge beliebig vertauscht werden.

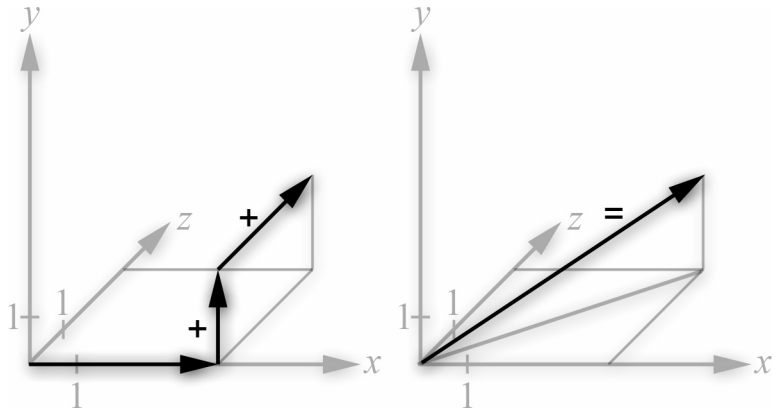


Abbildung 2.10 Der Vektor (4, 2, 3) im dreidimensionalen kartesischen Koordinatensystem

Ein paar Beispiele für die Koordinatenschreibweise:

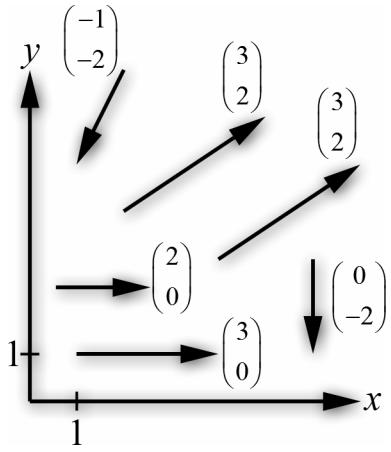


Abbildung 2.11 Vektorpfeile mit ihrer Koordinatendarstellung

Die Koordinatendarstellung der Basisvektoren $\vec{x}$, $\vec{y}$ und $\vec{z}$ ist:

$$\vec{x} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \quad \vec{y} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad \vec{z} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Die Koordinaten des Koordinatenursprungs sind:

$$0 \cdot \vec{x} + 0 \cdot \vec{y} + 0 \cdot \vec{z} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \vec{0}$$

2.3.3 Orts- und Richtungsvektoren

Mit einem Vektor kann man sowohl eine Position (einen Punkt) als auch eine Richtung beschreiben, aber nicht beides gleichzeitig. Als *Richtungsvektor* betrachtet würde der Vektor $(1 \quad 0)$ beispielsweise für die Richtung „rechts“ stehen und $(-1 \quad 1)$ für „links oben“. Die Länge des Vektors kann in physikalischen Anwendungen zum Beispiel als Geschwindigkeit interpretiert werden.

Um einen Vektor als Position zu interpretieren (*Ortsvektor*), wandert man einfach vom Koordinatenursprung aus entlang des vom Vektor beschriebenen Pfeils. Die Stelle, an der man dann ankommt, ist die gesuchte Position. Beispielsweise könnte in der Abbildung der Vektor $(4 \quad 2 \quad 3)$ als Richtung (das wäre die Richtung des Pfeils) oder als Position (das wäre die Position der Pfeilspitze) aufgefasst werden.

Wenn man die Ortsvektoren zweier Punkte A und B kennt und nun an der Richtung interessiert ist, mit der man von A nach B gelangen kann, dann muss man einfach nur die Ortsvektoren der Punkte $\vec{A}$ und $\vec{B}$ voneinander subtrahieren. Den Richtungsvektor von A nach B schreibt man auch $\overrightarrow{AB}$:

$$\overrightarrow{AB} = \vec{B} - \vec{A}$$
$$\vec{A} + \overrightarrow{AB} = \vec{A} + \vec{B} - \vec{A} = \vec{B}$$

Wichtig ist, dass ein Vektor immer nur eins von beidem darstellt, nicht Position und Richtung gleichzeitig!

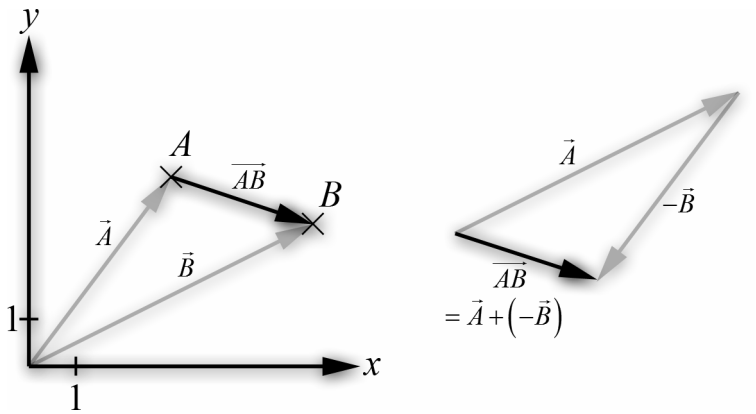


Abbildung 2.12 Die Punkte A und B mit den Ortsvektoren $\vec{A}$ und $\vec{B}$ und dem Richtungsvektor $\overrightarrow{AB}$, der von A nach B zeigt

Orts- und Richtungsvektoren unterscheiden sich äußerlich natürlich nicht voneinander. Darum muss man sich immer im Klaren darüber sein, mit welcher Art man es zu tun hat, um dann die richtigen Berechnungen durchzuführen.

Rechnen mit Koordinaten

Vorhin haben wir Vektoren durch das Aneinanderlegen von Pfeilen addiert und subtrahiert und die Multiplikation mit Skalaren durch eine Streckung der Pfeile realisiert. Selbstverständlich kann man diese Berechnungen auch mit der Koordinatendarstellung anstellen.

Wir wollen als Beispiel die Vektoren (1, 3) und (3, 7) addieren. Stellen wir uns dazu die Vektoren zunächst noch einmal als Pfeile vor. Der eine Pfeil zeigt eine Einheit nach rechts und drei nach oben, und der andere zeigt drei Einheiten nach rechts und sieben nach oben. Zum Addieren legen wir den einen Pfeil an die Spitze des anderen, wandern folglich insgesamt $1+3=4$ Einheiten nach rechts und $3+7=10$ Einheiten nach oben.

$$\begin{pmatrix}1\\3\end{pmatrix}+\begin{pmatrix}3\\7\end{pmatrix}=\begin{pmatrix}4\\10\end{pmatrix}$$

Um zwei Vektoren zu addieren, muss man also nur ihre Koordinaten addieren. Das geht natürlich auch mit dreidimensionalen Vektoren. Es gilt:

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \end{pmatrix} \text{ und } \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ x_3 + y_3 \end{pmatrix}$$

allgemein:

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{pmatrix}$$

Wir werden es in diesem Buch normalerweise nur mit zwei-, drei- oder vierdimensionalen Vektoren zu tun bekommen, aber man sollte trotzdem wissen, dass beliebig viele Dimensionen möglich sind. Zahlen kann man übrigens als *eindimensionale* Vektoren betrachten.

Die Subtraktion von Vektoren in Koordinatenform funktioniert analog zur Addition, nur dass das + durch ein – ersetzt wird.

Außerdem ist es natürlich auch möglich, zwei Vektoren komponentenweise miteinander zu multiplizieren. Diese Art der Multiplikation kommt jedoch eher selten vor. Wir wollen sie mit dem Multiplikationsstern ($*$) darstellen.

Betrachten wir noch die Multiplikation mit einem Skalar, die in der Pfeildarstellung ja zu einer Streckung, Stauchung oder Umkehr des Pfeils führt. Wir wollen das Produkt aus dem Vektor (1, 2, –8) und der Zahl –2 berechnen. Dazu müssen wir einfach nur alle Koordinaten mit –2 multiplizieren:

$$-2 \cdot \begin{pmatrix} 1 \\ 2 \\ -8 \end{pmatrix} = \begin{pmatrix} -2 \\ -4 \\ 16 \end{pmatrix}$$

Offensichtlich gilt:

$$r \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} r \cdot x_1 \\ r \cdot x_2 \end{pmatrix} \text{ und } r \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} r \cdot x_1 \\ r \cdot x_2 \\ r \cdot x_3 \end{pmatrix}$$

allgemein:

$$r \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} r \cdot x_1 \\ r \cdot x_2 \\ \vdots \\ r \cdot x_n \end{pmatrix}$$

Einen Vektor kann man auch durch eine Zahl dividieren. Dazu multipliziert man ihn einfach mit dem Kehrwert der Zahl.

Wie Sie sehen, ist das Rechnen mit Koordinaten ziemlich einfach und intuitiv. Sie werden als Nächstes noch ein paar weitere Eigenschaften und Operationen von Vektoren kennen lernen, die in der 3D-Grafik sehr wichtig sind.

2.3.4 Die Vektorlänge

Die Länge eines Vektors kennen wir bisher nur von der Pfeildarstellung. Dort entspricht sie der Länge des Pfeils. Statt der Länge kann man auch vom Betrag sprechen und schreibt daher die Länge des Vektors $\vec{x}$ als $|\vec{x}|$.

Berechnen der Vektorlänge mit dem Satz des Pythagoras

In der Koordinatenform kann die Länge eines Vektors mit Hilfe des *Satz des Pythagoras* berechnet werden. Diesen Satz sollte eigentlich jeder kennen. Er besagt, wie man in einem rechtwinkligen Dreieck die Länge einer Seite berechnen kann, wenn die Längen der beiden anderen Seiten bekannt sind.

Wenn *a*, *b* und *c* die Seiten eines rechtwinkligen Dreiecks sind, wobei die Seite *c* die Hypotenuse ist (sie liegt dem rechten Winkel gegenüber), dann gilt:

$$a^2 + b^2 = c^2$$

Infolgedessen lässt sich die Seite *c* durch $\sqrt{a^2 + b^2}$ berechnen.

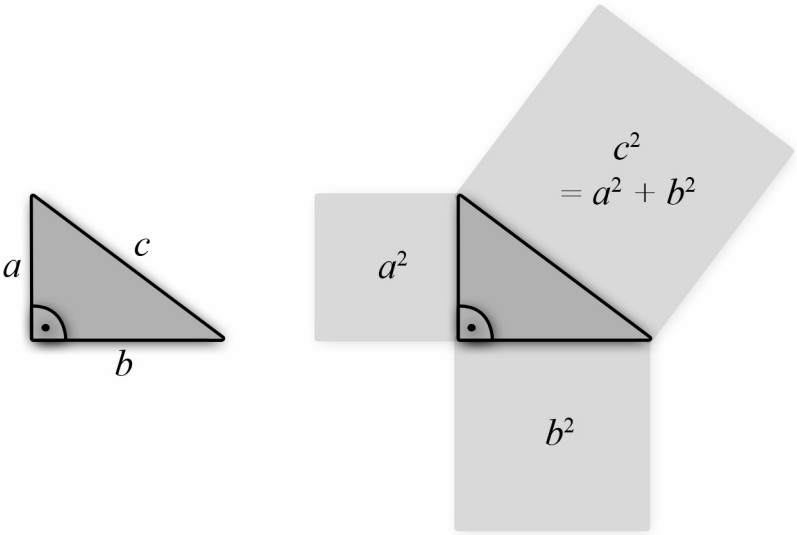


Abbildung 2.13 Mit dem Satz des Pythagoras lässt sich die Länge einer Seite eines rechtwinkligen Dreiecks ausrechnen, wenn die Längen der beiden anderen Seiten bekannt sind.

Nehmen wir zum Beispiel die Seitenlängen $a = 3$ und $b = 4$. Wie lang ist dann die Seite *c*?

$$c^2 = a^2 + b^2 = 3^2 + 4^2 = 25$$
$$c = \sqrt{25} = 5$$

c hat also die Länge 5.

Nun möchten wir die Länge des Vektors $\begin{pmatrix}7 & 5\end{pmatrix}$ berechnen. Der Satz des Pythagoras lässt sich hier sehr einfach anwenden, indem man aus dem Pfeil des Vektors, dessen Länge wir suchen, ein rechtwinkliges Dreieck erzeugt:

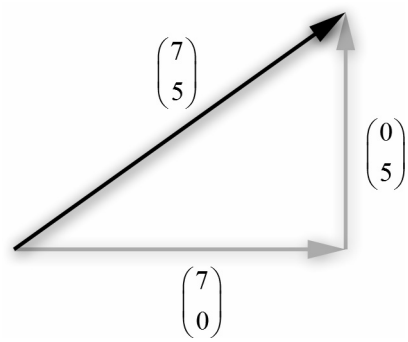


Abbildung 2.14 Der Pfeil, dessen Länge gesucht ist, als Hypotenuse eines rechtwinkligen Dreiecks

Die Längen der beiden anderen Seiten entsprechen offensichtlich den Koordinaten des Vektors, also 7 und 5. Darum gilt für die Länge des Vektors:

$$\left|\begin{pmatrix}7\\5\end{pmatrix}\right|^2 = 7^2 + 5^2 \text{ und somit } \left|\begin{pmatrix}7\\5\end{pmatrix}\right| = \sqrt{7^2 + 5^2} = \sqrt{74} \approx 8.6$$

Dieses Berechnungsschema, also das Aufsummieren der Quadrate der Koordinaten und das anschließende Ziehen der Wurzel, funktioniert für beliebige Dimensionen:

$$\left|\begin{pmatrix}x_1\\x_2\end{pmatrix}\right| = \sqrt{x_1^2 + x_2^2} \text{ und } \left|\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix}\right| = \sqrt{x_1^2 + x_2^2 + x_3^2}$$

allgemein:

$$\left|\begin{pmatrix}x_1\\x_2\\\vdots\\x_n\end{pmatrix}\right| = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2} = \sqrt{\sum_{i=1}^{n} x_i^2}$$

Das Quadrat der Länge eines Vektors $\vec{x}$, also $|\vec{x}|^2$, schreibt man auch einfach als $\vec{x}^2$. Man kann hier die Betragsstriche weglassen. Warum man das tun kann, werden wir später noch sehen.

Entfernung zwischen zwei Punkten

Da wir nun die Länge eines Vektors berechnen können, ist es auch ganz leicht möglich, die Entfernung zwischen zwei Punkten zu bestimmen. Die Entfernung zwischen zwei Punkten A und B ist nämlich die Länge des Vektors $\overrightarrow{AB} = \vec{B} - \vec{A}$. Ein Beispiel:

$$\vec{A} = \begin{pmatrix} 4 \\ 7 \\ 11 \end{pmatrix} \text{ und } \vec{B} = \begin{pmatrix} 0 \\ 8 \\ 15 \end{pmatrix}$$

$$\overrightarrow{AB} = \vec{B} - \vec{A} = \begin{pmatrix} 0 \\ 8 \\ 15 \end{pmatrix} - \begin{pmatrix} 4 \\ 7 \\ 11 \end{pmatrix} = \begin{pmatrix} -4 \\ 1 \\ 4 \end{pmatrix}$$

$$\left|\overrightarrow{AB}\right|^2 = (-4)^2 + 1^2 + 4^2$$

$$\left|\overrightarrow{AB}\right| = \sqrt{(-4)^2 + 1^2 + 4^2} = \sqrt{33} \approx 5.74$$

Die Entfernung zwischen A und B beträgt folglich ungefähr 5.74 Einheiten.

Mittelpunkt einer Strecke

Stellen wir uns wieder zwei Punkte A und B vor. Nun wollen wir wissen, welcher Punkt M genau in der Mitte auf der Strecke zwischen den Punkten liegt. Dazu beginnen wir beim Punkt A und wandern von dort aus die Hälfte von $\overrightarrow{AB}$ weiter:

$$\begin{aligned} \vec{M} &= \vec{A} + \tfrac{1}{2} \cdot \overrightarrow{AB} \\ &= \vec{A} + \tfrac{1}{2} \cdot \left(\vec{B} - \vec{A}\right) \\ &= \tfrac{1}{2} \cdot \vec{A} + \tfrac{1}{2} \cdot \vec{B} \\ &= \frac{\vec{A} + \vec{B}}{2} \end{aligned}$$

Man kann also einfach den „Mittelwert“ der Vektoren $\vec{A}$ und $\vec{B}$ berechnen und erhält den Ortsvektor des Streckenmittelpunkts.

Normierte Vektoren

Ein Vektor $\vec{x}$ heißt *normiert* oder *Einheitsvektor*, wenn seine Länge 1 ist, also wenn $\left|\vec{x}\right| = 1$ gilt. Normierte Vektoren haben gewisse Eigenschaften, die wir später noch kennen lernen werden.

Man kann einen Vektor normieren, indem man ihn durch seine Länge teilt. Für den Nullvektor $\vec{0}$ funktioniert das natürlich nicht, da dies eine Division durch Null bedeuten würde. Ist also ein Vektor $\vec{x} \neq \vec{0}$ gegeben, dann kann der dazugehörige Einheitsvektor $\vec{n}$ wie folgt berechnet werden:

$$\vec{n} = \frac{\vec{x}}{|\vec{x}|}$$

Der Vektor $\vec{n}$ zeigt in dieselbe Richtung wie $\vec{x}$, hat aber eine Länge von 1. Stattdessen kann man auch $\vec{x}_0$ schreiben. Die tiefgestellte Null zeigt, dass der Vektor normiert ist.

Normierte Vektoren werden meistens als Richtungsvektoren betrachtet. Ein normierter Richtungsvektor enthält lediglich eine Richtungsinformation und zum Beispiel keine Geschwindigkeitsangabe. Jemand könnte sagen: „Ich fahre gerade in Richtung Westen“. Hier ist nur eine Richtungsangabe gemacht worden, und unter der Annahme, dass Westen links ist, könnte man diese Richtung als (–1, 0) darstellen.

Wenn nun aber gesagt wird: „Ich fahre gerade mit 240 km/h in Richtung Westen“, dann haben wir neben der Richtungsangabe auch eine Aussage über die Geschwindigkeit und könnten dies als Vektor (–240, 0) schreiben.

2.3.5 Das Skalarprodukt

Wir haben schon zwei Operationen kennen gelernt, die sich mit Vektoren durchführen lassen: die Addition zweier Vektoren und die Multiplikation eines Vektors mit einem Skalar. Dies sind aber nicht die einzigen Operationen. Eine weitere ist das *Skalarprodukt*, auch *inneres Produkt* und im Englischen *dot product* genannt.

Das Skalarprodukt verknüpft zwei Vektoren zu einem Skalar. Mit seiner Hilfe lassen sich beispielsweise Winkel zwischen zwei Richtungsvektoren berechnen. Das Skalarprodukt wird mit einem Punkt notiert, zum Beispiel $\vec{a} \bullet \vec{b}$. Ich habe mich dafür entschieden, den Punkt für das Skalarprodukt dicker darzustellen als den für die normale Multiplikation von Zahlen. Dieser Punkt kann aber, so wie der Multiplikationspunkt bei Zahlen, weggelassen werden. In der linearen Algebra ist das Skalarprodukt wie folgt definiert:

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \bullet \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n = \sum_{i=1}^{n} x_i y_i$$

Das bedeutet: die Koordinaten der Vektoren werden miteinander multipliziert, und die Produkte werden aufsummiert. Ein paar Beispiele:

$$\begin{pmatrix}1\\2\end{pmatrix}\bullet\begin{pmatrix}4\\3\end{pmatrix}=1\cdot 4+2\cdot 3=10$$

$$\begin{pmatrix}4\\-1\\0\end{pmatrix}\bullet\begin{pmatrix}1\\1\\2\end{pmatrix}=4\cdot 1+(-1)\cdot 1+0\cdot 2=3$$

$$\begin{pmatrix}1\\0\\0\end{pmatrix}\bullet\begin{pmatrix}0\\0\\1\end{pmatrix}=1\cdot 0+0\cdot 0+0\cdot 1=0$$

Das „Quadrat“ eines Vektors

Vor ein paar Seiten wurde gesagt, dass man das Quadrat der Länge eines Vektors, also $|\vec{x}|^2$, auch als $\vec{x}^2$ schreiben kann. $\vec{x}^2$ bedeutet $\vec{x}\bullet\vec{x}$, also das Skalarprodukt des Vektors $\vec{x}$ mit sich selbst. Rechnen wir einmal nach, was dabei herauskommt:

$$\begin{pmatrix}x_1\\x_2\\\vdots\\x_n\end{pmatrix}^2=\begin{pmatrix}x_1\\x_2\\\vdots\\x_n\end{pmatrix}\bullet\begin{pmatrix}x_1\\x_2\\\vdots\\x_n\end{pmatrix}=x_1x_1+x_2x_2+\ldots+x_nx_n={x_1}^2+{x_2}^2+\ldots+{x_n}^2=\sum_{i=1}^{n}{x_i}^2=\left|\begin{pmatrix}x_1\\x_2\\\vdots\\x_n\end{pmatrix}\right|^2$$

Wie man sieht, gilt tatsächlich $\vec{x}^2=|\vec{x}|^2$. Man kann das Quadrat der Vektorlänge (und damit natürlich auch die Vektorlänge selbst) folglich mit Hilfe des Skalarprodukts berechnen.

Eigenschaften des Skalarprodukts

Das Skalarprodukt ist, genau wie die Vektoraddition und die Multiplikation mit Skalaren, kommutativ. Es gilt also $\vec{x}\bullet\vec{y}=\vec{y}\bullet\vec{x}$. Das lässt sich leicht zeigen:

$$\vec{x}\bullet\vec{y}=\begin{pmatrix}x_1\\x_2\\\vdots\\x_n\end{pmatrix}\bullet\begin{pmatrix}y_1\\y_2\\\vdots\\y_n\end{pmatrix}=\sum_{i=1}^{n}x_iy_i=\sum_{i=1}^{n}y_ix_i=\begin{pmatrix}y_1\\y_2\\\vdots\\y_n\end{pmatrix}\bullet\begin{pmatrix}x_1\\x_2\\\vdots\\x_n\end{pmatrix}=\vec{y}\bullet\vec{x}$$

Eine weitere Eigenschaft ist die Distributivität bezüglich der Vektoraddition (und der Vektorsubtraktion). Das heißt:

$$\vec{a}\bullet\left(\vec{b}+\vec{c}\right)=\vec{a}\bullet\vec{b}+\vec{a}\bullet\vec{c}$$

$$\left(\vec{a}+\vec{b}\right)\bullet\vec{c}=\vec{a}\bullet\vec{c}+\vec{a}\bullet\vec{b}$$

Außerdem gilt das Assoziativgesetz der Skalarmultiplikation:

$$(r \cdot \vec{a}) \bullet \vec{b} = r \cdot (\vec{a} \bullet \vec{b}) = \vec{a} \bullet (r \cdot \vec{b})$$

Mit dem Skalarprodukt lässt sich also fast genauso rechnen wie man es von der normalen Multiplikation kennt.

Winkel berechnen

Bislang dürfte es schwer fallen, sich unter dem Skalarprodukt etwas vorzustellen. Das wird sich nun ändern, denn es gibt noch eine äquivalente Definition für das Skalarprodukt, die im euklidischen Raum gilt (in dem wir schon die ganze Zeit rechnen). Diese Definition lautet wie folgt:

$$\vec{x} \bullet \vec{y} = |\vec{x}| \cdot |\vec{y}| \cdot \cos \measuredangle(\vec{x}, \vec{y})$$

$\measuredangle(\vec{x}, \vec{y})$ steht hierbei für den Winkel zwischen den (Richtungs-)Vektoren $\vec{x}$ und $\vec{y}$. Welche Aussage macht diese Gleichung? Das Skalarprodukt aus zwei Vektoren steht offenbar in einem Verhältnis zum Kosinus des Winkels, den diese beiden Vektoren einschließen. Als Faktoren treten dabei die Längen der Vektoren auf.

Wenn wir die Gleichung durch Umformen und mit Hilfe der Umkehrfunktion des Kosinus nach $\measuredangle(\vec{x}, \vec{y})$ auflösen, können wir den Winkel zwischen zwei Vektoren sofort berechnen:

$$\measuredangle(\vec{x}, \vec{y}) = \arccos \frac{\vec{x} \bullet \vec{y}}{|\vec{x}| \cdot |\vec{y}|}$$

Wenn $\vec{x}$ und $\vec{y}$ normiert sind, dann gilt $|\vec{x}| \cdot |\vec{y}| = 1$ und daher $\measuredangle(\vec{x}, \vec{y}) = \arccos(\vec{x} \bullet \vec{y})$.

Wir wollen diesen Zusammenhang zwischen dem Skalarprodukt, den Vektorlängen und dem Winkel anhand eines Beispiels überprüfen, indem wir den Winkel zwischen zwei Vektoren ausrechnen.

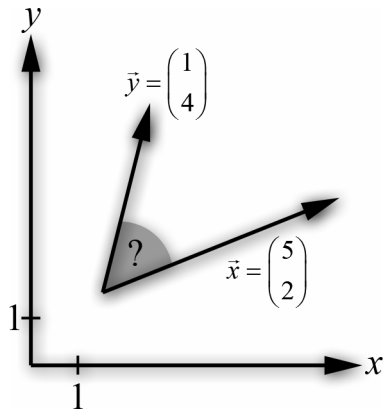


Abbildung 2.15 Gesucht ist der Winkel zwischen den beiden Vektoren $\vec{x}$ und $\vec{y}$.

Man könnte den Winkel jetzt messen, aber wir wollen ihn mit Hilfe des Skalarprodukts berechnen (denn ein Computer hat normalerweise kein Geodreieck zur Hand). Dafür müssen wir nur die beiden Vektoren in die zuvor genannte Gleichung einsetzen:

$$\measuredangle(\vec{x},\,\vec{y}) = \arccos\frac{\vec{x}\bullet\vec{y}}{|\vec{x}|\cdot|\vec{y}|} = \arccos\frac{\begin{pmatrix}5\\2\end{pmatrix}\bullet\begin{pmatrix}1\\4\end{pmatrix}}{\left|\begin{pmatrix}5\\2\end{pmatrix}\right|\cdot\left|\begin{pmatrix}1\\4\end{pmatrix}\right|}$$

$$= \arccos\frac{5\cdot 1+2\cdot 4}{\sqrt{5^2+2^2}\cdot\sqrt{1^2+4^2}} = \arccos\frac{13}{\sqrt{493}} \approx 0.945$$

Der gesuchte Winkel ist demnach ungefähr 0.945. Aber Vorsicht, denn dies ist kein Winkel in Grad, sondern im *Bogenmaß*. Um vom Bogenmaß nach Grad umzurechnen, muss man mit dem Umrechnungsfaktor $\frac{180°}{\pi} \approx 57.296°$ multiplizieren, umgekehrt durch ihn dividieren. Unser Ergebnis von 0.945 im Bogenmaß entspricht also ungefähr 54.14°. Wer es nicht glaubt, darf den Winkel in der Abbildung gerne nachmessen.

Diese Eigenschaft des Skalarprodukts kommt in der 3D-Grafik zum Beispiel beim Berechnen der Beleuchtung zur Anwendung, denn dort ist man meistens am Winkel interessiert, in dem die virtuellen Lichtstrahlen auf die Oberfläche eines Objekts treffen.

Mit dem Skalarprodukt kann man auch sehr leicht testen, ob zwei Vektoren orthogonal (senkrecht) zueinander sind. Wenn das der Fall ist, dann beträgt der Winkel zwischen ihnen genau 90°. Da der Kosinus von 90° Null ist und er als Faktor im Skalarprodukt enthalten ist, wird dadurch das ganze Ergebnis Null.

$$\measuredangle(\vec{x},\,\vec{y}) = 90° \Leftrightarrow \vec{x}\bullet\vec{y} = 0$$

Ein paar Beispiele für orthogonale Vektoren:

$$\begin{pmatrix}1\\0\end{pmatrix}\bullet\begin{pmatrix}0\\1\end{pmatrix}=0 \qquad \begin{pmatrix}4\\7\end{pmatrix}\bullet\begin{pmatrix}-7\\4\end{pmatrix}=0 \qquad \begin{pmatrix}8\\2\\1\end{pmatrix}\bullet\begin{pmatrix}-1\\3\\2\end{pmatrix}=0 \qquad \begin{pmatrix}9\\9\end{pmatrix}\bullet\begin{pmatrix}0\\0\end{pmatrix}=0$$

Wie man sich leicht überlegen kann, ist der Nullvektor zu jedem Vektor orthogonal. Wenn $\vec{a}$ und $\vec{b}$ orthogonal zueinander sind, kann man das auch so schreiben: $\vec{a} \perp \vec{b}$.

Für einen zweidimensionalen Vektor lässt sich sehr einfach ein Vektor finden, der senkrecht auf ihm steht und dieselbe Länge hat. Dafür vertauscht man die Koordinaten und ändert bei einer davon das Vorzeichen:

$$\begin{pmatrix}x_1\\x_2\end{pmatrix}\bullet\begin{pmatrix}-x_2\\x_1\end{pmatrix} = x_1\cdot(-x_2)+x_2\cdot x_1 = -x_1x_2+x_2x_1 = 0$$

Analog lässt sich neben der Orthogonalität auch die Parallelität zweier Vektoren mit Hilfe des Skalarprodukts bestimmen. Wenn zwei Vektoren parallel sind, dann sind sie Vielfache

voneinander, unterscheiden sich also nur durch einen Faktor, und sie schließen einen Winkel von 0° ein. Der Kosinus von 0° ist 1. Folglich entspricht das Skalarprodukt der Vektoren in diesem Fall genau dem Produkt ihrer Längen:

$$\measuredangle(\vec{x}, \vec{y}) = 0° \Leftrightarrow \vec{x} \bullet \vec{y} = |\vec{x}| \cdot |\vec{y}|$$

Ein paar Beispiele für parallele Vektoren:

$$\begin{pmatrix} 10 \\ 4 \end{pmatrix} \bullet \begin{pmatrix} 5 \\ 2 \end{pmatrix} = 58 = \sqrt{10^2 + 4^2} \cdot \sqrt{5^2 + 2^2} = \left|\begin{pmatrix} 10 \\ 4 \end{pmatrix}\right| \cdot \left|\begin{pmatrix} 5 \\ 2 \end{pmatrix}\right|$$

$$\begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix} \bullet \begin{pmatrix} 2 \\ 2 \\ 4 \end{pmatrix} = 12 = \sqrt{1^2 + 1^2 + 2^2} \cdot \sqrt{2^2 + 2^2 + 4^2} = \left|\begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix}\right| \cdot \left|\begin{pmatrix} 2 \\ 2 \\ 4 \end{pmatrix}\right|$$

Sind $\vec{a}$ und $\vec{b}$ parallel, dann schreibt man auch $\vec{a} \parallel \vec{b}$. Diese Schreibweise schließt aber auch den Fall ein, dass die Vektoren entgegengesetzte Richtungen haben. Manchmal wird das auch als Antiparallelität bezeichnet.

Projektion

Wir wollen noch eine vorerst letzte Anwendung des Skalarprodukts betrachten. In der Abbildung wird ein Vektor $\vec{v}$ auf eine Gerade mit der Richtung $\vec{u}$ projiziert, und es ist nach der Länge des projizierten Vektors gefragt.

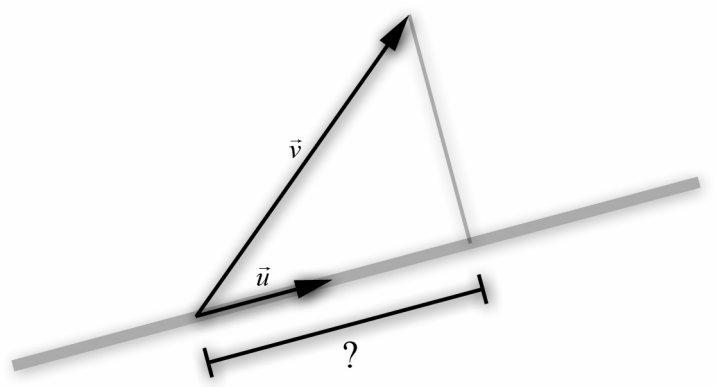


Abbildung 2.16 Wie lang ist die Projektion von $\vec{v}$ auf der von $\vec{u}$ festgelegten Gerade?

Um diese Frage beantworten zu können, teilen wir unseren Vektor $\vec{v}$ in zwei Komponenten $\vec{v}_P$ und $\vec{v}_S$ auf, wobei die Summe wieder $\vec{v}$ ergeben soll. $\vec{v}_P$ soll dabei parallel zu $\vec{u}$ sein und $\vec{v}_S$ senkrecht zu $\vec{u}$.

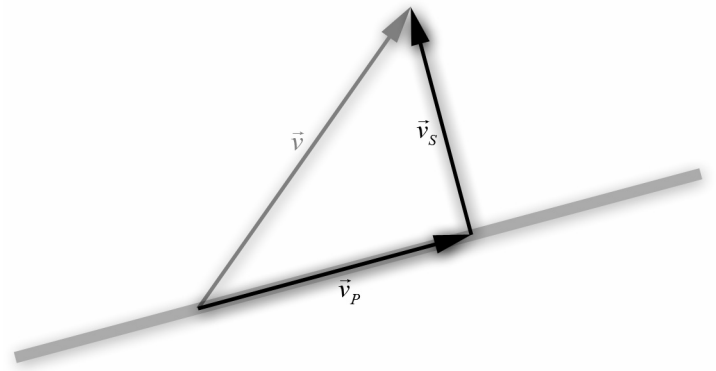


Abbildung 2.17 $\vec{v}$ wird in $\vec{v}_P$ und $\vec{v}_S$ aufgeteilt.

Wie man sieht, entspricht $|\vec{v}_P|$ genau der gesuchten Länge. Um den Vektor $\vec{v}_P$ zu berechnen, stellen wir ihn als Vielfaches von $\vec{u}$ dar. Das können wir tun, weil $\vec{v}_P$ per Definition ja parallel zu $\vec{u}$ ist:

$$\vec{v}_P = r \cdot \vec{u} \qquad (1)$$

Weiterhin haben wir definiert:

$$\vec{v}_P + \vec{v}_S = \vec{v} \qquad (2)$$

Und da $\vec{v}_S$ senkrecht zu $\vec{u}$ sein soll:

$$\vec{v}_S \bullet \vec{u} = 0 \qquad (3)$$

Unter Verwendung dieser drei Gleichungen ergibt sich:

$$\begin{aligned}
&\vec{v}_P + \vec{v}_S = \vec{v} && (2)\\
&\vec{v}_S = \vec{v} - \vec{v}_P \\
&\vec{v}_S = \vec{v} - r \cdot \vec{u} && (1)\\
&\left(\vec{v} - r \cdot \vec{u}\right) \bullet \vec{u} = 0 && (3)\\
&\vec{v} \bullet \vec{u} - r \bullet \vec{u}^2 = 0 \\
&r = \frac{\vec{v} \bullet \vec{u}}{\vec{u}^2}
\end{aligned}$$

Wegen (1) ist die Länge von $\vec{v}_P$, und damit unsere gesuchte Länge, *r*-mal so groß wie die Länge von $\vec{u}$. Wenn $\vec{u}$ normiert ist, gilt $\vec{u}^2 = 1$ und damit $r = |\vec{v}_P| = \vec{v} \bullet \vec{u}$. Das Skalarprodukt gibt in diesem Fall also sofort die projizierte Länge an.

Ich entschuldige mich für all diese Formeln, aber dieses Beispiel ist wirklich hilfreich, um den Umgang mit dem Skalarprodukt zu üben, und es ist auch in der Spieleprogrammierung nicht ganz unnütz. Beispielsweise kann man auf diese Weise das „entlang-Sliden“ an einer

Wand umsetzen oder die Richtung eines Strahls berechnen, der an einer Oberfläche reflektiert wird.

2.3.6 Das Kreuzprodukt

Neben dem Skalarprodukt gibt es noch eine weitere Operation. Sie verknüpft Vektoren zu einem neuen Vektor und wird *Kreuzprodukt* (auch *Vektorprodukt*) genannt. Wir betrachten das Kreuzprodukt nur für dreidimensionale Vektoren. Dort ist es eine zweistellige Operation, das heißt es verknüpft *zwei* Vektoren.

Das Kreuzprodukt wird mit einem Multiplikationskreuz geschrieben: $\vec{x} \times \vec{y}$ („ $\vec{x}$ kreuz $\vec{y}$ “). $\vec{x} \times \vec{y}$ liefert einen Vektor, der sowohl zu $\vec{x}$ als auch zu $\vec{y}$ senkrecht ist, also:

$$(\vec{x} \times \vec{y}) \bullet \vec{x} = 0 \text{ und } (\vec{x} \times \vec{y}) \bullet \vec{y} = 0$$

Es lässt sich wie folgt berechnen:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \times \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} x_2 y_3 - x_3 y_2 \\ x_3 y_1 - x_1 y_3 \\ x_1 y_2 - x_2 y_1 \end{pmatrix}$$

Nun gibt es im allgemeinen Fall nicht *den einen* Vektor, der zu $\vec{x}$ und $\vec{y}$ senkrecht ist, sondern unendlich viele (wenn das für einen Vektor gilt, gilt es auch für alle seine Vielfachen). Welchen davon liefert das Kreuzprodukt? Hier hilft uns die folgende Definition:

$$\vec{x} \times \vec{y} = |\vec{x}| \cdot |\vec{y}| \cdot \sin \measuredangle(\vec{x}, \vec{y}) \cdot \vec{n}_0$$

$\measuredangle(\vec{x}, \vec{y})$ ist hier wieder der Winkel zwischen $\vec{x}$ und $\vec{y}$, und $\vec{n}_0$ ist der zu $\vec{x}$ und $\vec{y}$ senkrechte *Einheitsvektor*. $\vec{n}_0$ hat also die Länge 1. Aus dieser Definition folgt für die Länge des Kreuzprodukts: $|\vec{x} \times \vec{y}| = |\vec{x}| \cdot |\vec{y}| \cdot \sin \measuredangle(\vec{x}, \vec{y})$.

Jetzt kennen wir die Länge des Ergebnisvektors, den das Kreuzprodukt liefert, aber in welche Richtung zeigt er? Das ist immer noch nicht eindeutig, denn es gibt im Allgemeinen zwei Einheitsvektoren, die senkrecht zu zwei anderen Vektoren sind. Diese beiden Kandidaten für $\vec{n}$ haben entgegengesetzte Richtungen.

Da wir mit einem linkshändigen Koordinatensystem arbeiten, können wir die *Linke-Hand-Regel* verwenden, um die Richtung von $\vec{x} \times \vec{y}$ zu bestimmen. Dazu nimmt man die linke Hand und streckt Daumen, Zeige- und Mittelfinger senkrecht zueinander aus. Der Mittelfinger steht für den Vektor $\vec{x}$, der Daumen für $\vec{y}$ und der Zeigefinger für $\vec{x} \times \vec{y}$.

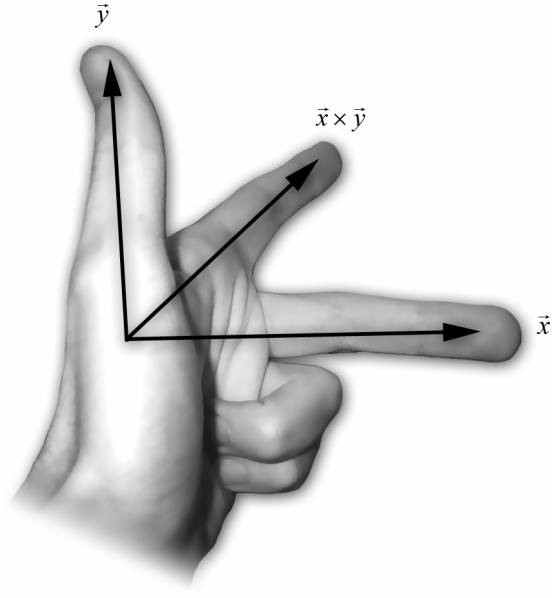


Abbildung 2.18 Die Linke-Hand-Regel erlaubt es, die Richtung des Kreuzprodukts zu bestimmen.

In unserem dreidimensionalen kartesischen Koordinatensystem ist beispielsweise die *z*-Achse das Kreuzprodukt aus der *x*- und der *y*-Achse.

Eigenschaften des Kreuzprodukts

Das Kreuzprodukt ist im Gegensatz zum Skalarprodukt *nicht* kommutativ! Wenn man die Operanden vertauscht, ändert sich das Vorzeichen des Ergebnisvektors:

$$\vec{x} \times \vec{y} = -\left(\vec{y} \times \vec{x}\right)$$

Das kann man daran erkennen, dass die Berechnung jeder Koordinate mit Hilfe der Subtraktion funktioniert, die ebenfalls diese Eigenschaft hat.

Die Distributivgesetze gelten jedoch für das Kreuzprodukt:

$$\vec{a} \times \left(\vec{b} + \vec{c}\right) = \vec{a} \times \vec{b} + \vec{a} \times \vec{c}$$
$$\left(\vec{a} + \vec{b}\right) \times \vec{c} = \vec{a} \times \vec{c} + \vec{a} \times \vec{b}$$

Das Assoziativgesetz der Skalarmultiplikation gilt ebenfalls:

$$\left(r \cdot \vec{a}\right) \times \vec{b} = r \cdot \left(\vec{a} \times \vec{b}\right) = \vec{a} \times \left(r \cdot \vec{b}\right)$$

Wenn das Kreuzprodukt von zwei parallelen Vektoren gebildet wird, dann ergibt sich der Nullvektor:

$$\vec{a} \times (r \cdot \vec{a}) = \vec{0}$$

Insbesondere:

für $r = 1$: $\vec{a} \times \vec{a} = \vec{0}$

für $r = 0$: $\vec{a} \times \vec{0} = \vec{0}$

Flächenberechnung mit dem Kreuzprodukt

Eine Anwendung des Kreuzprodukts ist die Berechnung der Flächen von Parallelogrammen und Dreiecken. Die Länge von $\vec{x} \times \vec{y}$ entspricht nämlich der Fläche des Parallelogramms, das durch die Vektoren $\vec{x}$ und $\vec{y}$ aufgespannt wird.

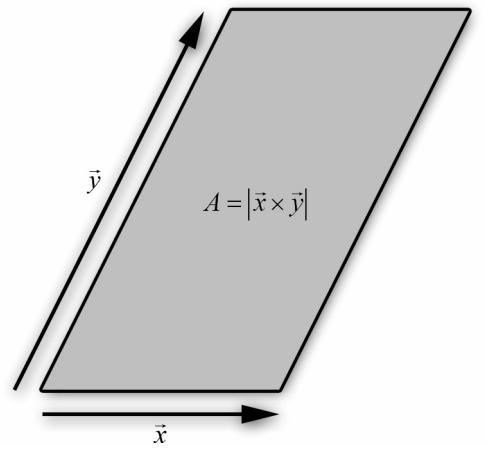


Abbildung 2.19 Der Betrag des Kreuzprodukts entspricht der Fläche des Parallelogramms.

2.4 Matrizen

2.4.1 Welcome to the Matrix

Matrizen sind in der 3D-Grafik genauso wichtig wie Vektoren. Mit ihnen kann man Objekte auf einfache Weise im Raum verschieben, drehen, strecken, stauchen, spiegeln oder projizieren – allesamt wichtige Dinge bei Spielen. Aber was ist eine Matrix überhaupt? Jedenfalls haben Matrizen nichts mit dem gleichnamigen Film zu tun.

Die Matrix als Tabelle

Eine Matrix ist sozusagen die nächste Stufe eines Vektors. Einen Vektor haben wir entweder als Zeile oder als Spalte von Komponenten dargestellt, also als eindimensionale Anordnung. Eine Matrix hingegen hat Zeilen *und* Spalten, wie eine Tabelle. Die Einträge nennt man wie bei Vektoren auch *Komponenten*.

Eine Matrix mit *m* Zeilen und *n* Spalten wird auch als $m \times n$ *-Matrix* bezeichnet (gesprochen: „*m*-mal-*n*-Matrix“ oder „*m*-kreuz-*n*-Matrix“). Wenn die Matrix gleich viele Zeilen wie Spalten hat, nennt man sie *quadratisch*.

Variablen, die für Matrizen stehen, schreibt man meistens mit Großbuchstaben. Die Komponenten einer Matrix schreibt man, wie bei den Vektoren, als Kleinbuchstaben mit tiefgestelltem Index. Der Index enthält zuerst die Zeilennummer und dann die Spaltennummer der Komponente.

Ein Beispiel für eine 3x4-Matrix:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}$$

Achtung: a_{23} ist nicht als „*a dreiundzwanzig*“ lesen, sondern als „*a zwei, drei*“!

Die Anzahl der Zeilen *m* nennt man auch *Zeilendimension*, die Anzahl der Spalten *n* auch *Spaltendimension* einer Matrix. Die Zeilen und Spalten einer Matrix kann man als Vektoren auffassen. Man nennt sie daher auch *Zeilenvektoren* und *Spaltenvektoren*. Der zweite Zeilenvektor der Matrix von vorhin wäre $(a_{21},\ a_{22},\ a_{23},\ a_{24})$.

2.4.2 Rechnen mit Matrizen

Auch für Matrizen ist eine Reihe von Rechenoperationen definiert. Für manche davon kann man die Definition von den entsprechenden Vektoroperationen übernehmen.

Addition und Subtraktion

Man addiert zwei Matrizen, indem man ihre Komponenten addiert. Die Matrizen müssen dieselbe Zeilen- und Spaltendimension haben, ansonsten lassen sie sich nicht addieren.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} \end{pmatrix}$$

allgemein:

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} + \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mn} \end{pmatrix} = \begin{pmatrix} a_{11}+b_{11} & \cdots & a_{1n}+b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & \cdots & a_{mn}+b_{mn} \end{pmatrix}$$

Bei der Subtraktion werden die Komponenten subtrahiert, ansonsten bleibt alles gleich. Verkürzt könnte man auch schreiben:

$$C = A + B \qquad C = A - B$$
$$c_{ij} = a_{ij} + b_{ij} \qquad c_{ij} = a_{ij} - b_{ij}$$

Ein Beispiel für die Addition von Matrizen:

$$\begin{pmatrix} 1 & 5 \\ -5 & 2 \end{pmatrix} + \begin{pmatrix} 2 & 0 \\ -1 & 3 \end{pmatrix} = \begin{pmatrix} 3 & 5 \\ -6 & 5 \end{pmatrix}$$

Multiplikation mit Skalaren

Eine Matrix wird mit einem Skalar multipliziert, indem man ihre Komponenten mit ihm multipliziert:

$$r \cdot \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} = \begin{pmatrix} r \cdot a_{11} & \cdots & r \cdot a_{1n} \\ \vdots & \ddots & \vdots \\ r \cdot a_{m1} & \cdots & r \cdot a_{mn} \end{pmatrix}$$

oder kürzer: $C = r \cdot A$

$$c_{ij} = r \cdot A_{ij}$$

Zum Beispiel:

$$4 \cdot \begin{pmatrix} 7 & 0 & -1 \\ 2 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 28 & 0 & -4 \\ 8 & 8 & 12 \end{pmatrix}$$

Matrixmultiplikation

Kommen wir nun zur Multiplikation von Matrizen. Man kann sie mit dem Skalarprodukt bei Vektoren vergleichen. Darum werde ich für die Matrixmultiplikation ebenfalls den dicken Punkt als Operatorzeichen verwenden.

Sei A eine $l \times m$-Matrix und B eine $m \times n$-Matrix (A hat also so viele Spalten wie B Zeilen hat). Dann können A und B miteinander multipliziert werden, und das Produkt ist eine $l \times n$-Matrix C. Diese Matrix C hat demnach so viele Zeilen wie A und so viele Spalten wie B.

Zur Berechnung des Eintrags c_{ij} bildet man das Skalarprodukt aus dem i-ten Zeilenvektor von A und dem j-ten Spaltenvektor von B:

$$\underbrace{C}_{l\times n} = \underbrace{A}_{l\times m} \bullet \underbrace{B}_{m\times n}$$

$$c_{ij} = \sum_{k=1}^{m} a_{ik} \cdot b_{kj}$$

Zum Beispiel:

$$\begin{pmatrix} 1 & 3 & 2 \\ 0 & -1 & 4 \end{pmatrix} \bullet \begin{pmatrix} 1 & 2 & 1 & 0 \\ -1 & 3 & 4 & 2 \\ -2 & 0 & 5 & 1 \end{pmatrix}$$

$$= \left(\begin{array}{c|c|c|c} 1\cdot 1+3\cdot(-1)+2\cdot(-2) & 1\cdot 2+3\cdot 3+2\cdot 0 & 1\cdot 1+3\cdot 4+2\cdot 5 & 1\cdot 0+3\cdot 2+2\cdot 1 \\ \hline 0\cdot 1+(-1)\cdot(-1)+4\cdot(-2) & 0\cdot 2+(-1)\cdot 3+4\cdot 0 & 0\cdot 1+(-1)\cdot 4+4\cdot 5 & 0\cdot 0+(-1)\cdot 2+4\cdot 1 \end{array}\right)$$

$$= \begin{pmatrix} -6 & 11 & 23 & 8 \\ -7 & -3 & 16 & 2 \end{pmatrix}$$

Die Trennlinien in der zweiten Zeile haben keine Bedeutung, sie sollen nur die Lesbarkeit verbessern. Schauen Sie sich zur Verdeutlichung die folgende Abbildung an.

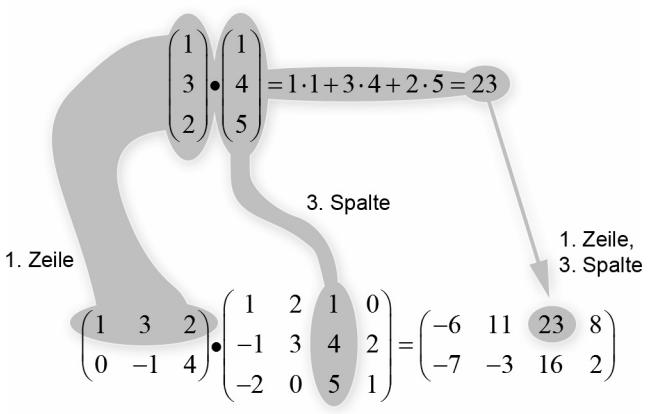


Abbildung 2.20 Die 23 in der Ergebnismatrix befindet sich in der 1. Zeile der 3. Spalte. Sie entsteht durch das Skalarprodukt aus dem 1. Zeilenvektor der ersten Matrix und dem 3. Spaltenvektor der zweiten Matrix.

Die Matrixmultiplikation ist *nicht* kommutativ! Im Allgemeinen gilt also für Matrizen $A \bullet B \neq B \bullet A$. Die beiden Matrizen aus unserem Beispiel könnte man auch gar nicht andersherum multiplizieren, da dann die Spaltendimension der ersten Matrix nicht mit der Zeilendimension der zweiten Matrix übereinstimmen würde.

Das Assoziativgesetz gilt aber:

$$(A \bullet B) \bullet C = A \bullet (B \bullet C)$$

Die Einheitsmatrix

Wenn man zwei quadratische Matrizen multipliziert, dann ist das Ergebnis wieder eine quadratische Matrix identischer Größe. So wie die Multiplikation einer Zahl mit 1 diese Zahl nicht verändert, gibt es auch bei den Matrizen ein neutrales Element der Multiplikation. Dies ist die *Einheitsmatrix*. Man nennt sie manchmal auch *Identitätsmatrix*. Sie wird mit dem Buchstaben *E* oder *I* (*Identity*) bezeichnet.

Die Einheitsmatrix enthält in ihrer Hauptdiagonalen (sie verläuft von links oben nach rechts unten) nur Einsen, und die restlichen Einträge sind mit null:

$$E_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad E_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad E_n = \underbrace{\begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 \\ \vdots & 0 & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}}_{n \text{ Spalten}}$$

Für alle Matrizen *A* gilt:

$$A \bullet E = E \bullet A = A$$

Eine Matrix mit der Einheitsmatrix zu multiplizieren hat also keine Wirkung, wie man leicht nachvollziehen kann. Dabei ist es egal, ob die Einheitsmatrix von links oder von rechts multipliziert wird.

2.4.3 Weitere Operationen

Invertierbarkeit einer Matrix

Wir wollen später auch Matrizen „dividieren“ können. Dazu ist es notwendig, den „Kehrwert“ einer Matrix zu berechnen (sie zu *invertieren* oder *umzukehren*). Bei den Zahlen ist zum Beispiel $\frac{1}{42}$ der Kehrwert von 42, denn $\frac{1}{42} \cdot 42 = 1$. Die 1 ist bei den Zahlen das neutrale Element der Multiplikation, wie vorhin schon erwähnt wurde. Man bezeichnet den Kehrwert daher auch als *multiplikatives Inverses*. Etwas Ähnliches kann man, unter bestimmten Voraussetzungen, auch mit Matrizen machen. Man kann also zu gewissen Matrizen eine inverse Matrix mit derselben Zeilen- und Spaltendimension finden, so dass das Produkt der

beiden die Einheitsmatrix ergibt (denn sie ist bei den Matrizen das neutrale Element der Multiplikation). Die Inverse einer Matrix A schreibt man allerdings nicht $\frac{1}{A}$, sondern A^{-1}.

Wenn es zu einer Matrix A eine Inverse A^{-1} gibt, so dass $A \bullet A^{-1} = A^{-1} \bullet A = E$ gilt, dann nennt man die Matrix *invertierbar* oder *regulär*. Es sind jedoch nicht alle Matrizen invertierbar. Welche Bedingungen müssen dazu erfüllt sein? Erst einmal ist es offensichtlich, dass nur quadratische Matrizen invertierbar sein können, da auch die Einheitsmatrix quadratisch ist. Schauen wir uns einmal eine Matrix an:

$$A = \begin{pmatrix} 2 & 4 \\ 8 & 6 \end{pmatrix}$$

Nun wollen wir A^{-1} berechnen. Da wir zunächst überhaupt nicht wissen, welche Werte in A^{-1} stehen müssen, damit $A \bullet A^{-1} = E$ gilt, schreiben wir vier Variablen hinein.

$$A = \begin{pmatrix} 2 & 4 \\ 8 & 6 \end{pmatrix}, \quad A^{-1} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

$$A \bullet A^{-1} = E$$

$$\begin{pmatrix} 2 & 4 \\ 8 & 6 \end{pmatrix} \bullet \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 2a+4c & 2b+4d \\ 8a+6c & 8b+6d \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Durch die letzte Zeile erhalten wir ein *lineares Gleichungssystem* (*LGS*), bestehend aus vier Gleichungen mit vier Variablen:

$$\begin{pmatrix} 2a+4c & 2b+4d \\ 8a+6c & 8b+6d \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\begin{aligned} \Rightarrow\ & 2a+4c = 1 \ \wedge && \text{(I)} \\ & 2b+4d = 0 \ \wedge && \text{(II)} \\ & 8a+6c = 0 \ \wedge && \text{(III)} \\ & 8b+6d = 1 && \text{(IV)} \end{aligned}$$

Wenn es uns gelingt, dieses LGS eindeutig zu lösen, können wir die Werte für a, b, c und d ausrechnen und erhalten somit die gesuchte inverse Matrix A^{-1}. Für die Lösbarkeit eines LGS gibt es drei Möglichkeiten:

- Das LGS hat genau eine Lösung (es ist eindeutig).
- Das LGS hat unendlich viele Lösungen.
- Das LGS hat keine Lösung.

Nur im ersten Fall, also wenn es eine eindeutige Lösung gibt, können wir die inverse Matrix berechnen. In unserem Beispiel ist das der Fall. Wir können das LGS lösen:

$$3\cdot(\text{I})-2\cdot(\text{III}):\quad -10a=3 \;\Rightarrow\; a=-0.3 \overset{(\text{I})}{\Rightarrow} c=\tfrac{1-2\cdot(-0.3)}{4}=0.4$$

$$3\cdot(\text{II})-2\cdot(\text{IV}):\quad -10b=-2 \;\Rightarrow\; b=0.2 \overset{(\text{II})}{\Rightarrow} d=\tfrac{-2\cdot 0.2}{4}=-0.1$$

Wir erhalten somit:

$$A^{-1}=\begin{pmatrix}-0.3 & 0.4\\ 0.2 & -0.1\end{pmatrix}$$

$$A\cdot A^{-1}=\begin{pmatrix}2 & 4\\ 8 & 6\end{pmatrix}\bullet\begin{pmatrix}-0.3 & 0.4\\ 0.2 & -0.1\end{pmatrix}=\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}=E$$

In diesem Fall hat das Invertieren funktioniert, weil das LGS eindeutig lösbar war. Aber woran erkennt man das? Nun, zum Lösen eines LGS darf man eine Gleichung mit einer Zahl ungleich Null multiplizieren oder eine Gleichung durch die Summe dieser Gleichung mit einer anderen Gleichung ersetzen. Damit ist es auch erlaubt, Gleichungen voneinander abzuziehen, nachdem man sie mit Zahlen ungleich Null multipliziert hat. Das ist genau das, was oben bei der Lösung des LGS getan wurde. Um den Wert für die Variable *a* zu erhalten, wurde zum Beispiel das Zweifache der dritten Gleichung (III) vom Dreifachen der ersten Gleichung (I) abgezogen. Warum? Ganz einfach: weil dann die Variable *c* wegfällt und nur noch $-10a=3$ übrig bleibt, woraus sich sofort $a=-0.3$ ergibt.

Das klappt aber nicht bei jedem LGS. Was wäre zum Beispiel, wenn unsere Gleichungen (I) und (III) so aussähen:

$$\begin{aligned} 2a+4c &= 1 \;\wedge && \text{(I)}\\ 8a+16c &= 0 && \text{(III)}\end{aligned}$$

Wenn man genau hinsieht, erkennt man, dass die linke Seite von (III) das Vierfache der linken Seite von (I) ist. Das bedeutet, dass wir keine Möglichkeit haben, die Gleichungen so miteinander zu verknüpfen, dass eine der beiden Variablen wegfällt. Wenn wir jetzt die Gleichung (I) mit 4 multiplizieren, erhalten wir:

$$\begin{aligned} 8a+16c &= 4 \;\wedge && \text{(I')}\\ 8a+16c &= 0 && \text{(III)}\end{aligned}$$

Dieses LGS hat offensichtlich keine Lösung, denn $8a+16c$ kann nicht sowohl 4 als auch 0 sein. Wenn man (III) von (I') abzieht, erhält man $0=4$.

Betrachten wir noch eine Variante:

$$\begin{aligned} 2a+4c &= 1 \;\wedge && \text{(I)}\\ 0a+0c &= 0 && \text{(III)}\end{aligned}$$

Die Gleichung (III) bedeutet einfach nur $0=0$ und ist damit überflüssig. Es bleibt also nur die Gleichung (I) übrig – eine Gleichung mit zwei Variablen. Sie kann keine eindeutige Lösung haben, sondern hat unendlich viele, zum Beispiel $a=\frac{1}{2}\wedge c=0$ oder $a=-\frac{3}{2}\wedge c=1$.

Daraus folgt, dass eine Matrix nur dann invertierbar ist, wenn sie keine Zeile und keine Spalte besitzt, die nur aus Nullen besteht, und wenn keine Zeile oder Spalte als *Linearkombination* anderer Zeilen beziehungsweise Spalten darstellbar ist. Das bedeutet: man darf eine Zeile oder Spalte nicht als Summe von Vielfachen anderer Zeilen oder Spalten schreiben dürfen. Bei unserer Beispielmatrix trafen diese Bedingungen zu, darum konnten wir sie invertieren.

Die Determinante

Die *Determinante* ordnet einer quadratischen Matrix eine Zahl zu (ähnlich wie die Vektorlänge einem Vektor eine Zahl zuordnet). Die Determinante einer Matrix A wird als $\det(A)$ oder $|A|$ geschrieben.

Mit der Determinante einer Matrix kann man herausfinden, ob sie invertierbar ist, also ob die oben aufgezählten Bedingungen zutreffen. Das ist sie nämlich genau dann der Fall, wenn die Determinante ungleich null ist.

Für 2×2 - und 3×3 -Matrizen kann man die Determinanten wie folgt berechnen (bei größeren Matrizen ist die direkte Berechnung eher unhandlich):

$$\det\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = a_{11}a_{22} - a_{21}a_{12}$$

$$\det\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{matrix} a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} \\ - a_{31}a_{22}a_{13} - a_{32}a_{23}a_{11} - a_{33}a_{21}a_{12} \end{matrix}$$

Für größere Matrizen lässt sich die Determinante mit dem *Laplaceschen Entwicklungssatz* rekursiv berechnen. Ich möchte hier aber nicht so tief ins Detail gehen.

Ein Beispiel für die Determinante einer 3×3 -Matrix:

$$\det\begin{pmatrix} 2 & -2 & 1 \\ 1 & 4 & 5 \\ 3 & 3 & 0 \end{pmatrix} = \begin{matrix} 2\cdot 4\cdot 0 + (-2)\cdot 5\cdot 3 + 1\cdot 1\cdot 3 \\ -3\cdot 4\cdot 1 - 3\cdot 5\cdot 2 - 0\cdot 1\cdot(-2) \end{matrix} = -69$$

Invertieren von beliebigen Matrizen

Wir haben anhand eines Beispiels gesehen, wie man eine Matrix invertieren kann. Für beliebige invertierbare 2×2 - und 3×3 -Matrizen kann man die inverse Matrix wie folgt berechnen:

für 2×2-Matrizen:

$$A^{-1}=\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}^{-1}=\frac{1}{\det(A)}\cdot\begin{pmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{pmatrix}$$

für 3×3-Matrizen:

$$A^{-1}=\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}^{-1}=\frac{1}{\det(A)}\cdot\begin{pmatrix} a_{22}a_{33}-a_{23}a_{32} & a_{13}a_{32}-a_{12}a_{33} & a_{12}a_{23}-a_{13}a_{22} \\ a_{23}a_{31}-a_{21}a_{33} & a_{11}a_{33}-a_{13}a_{31} & a_{13}a_{21}-a_{11}a_{23} \\ a_{21}a_{32}-a_{22}a_{31} & a_{12}a_{31}-a_{11}a_{32} & a_{11}a_{22}-a_{12}a_{21} \end{pmatrix}$$

In den Formeln spiegelt sich wider, warum sich nur Matrizen invertierten lassen, deren Determinante ungleich null ist, denn sonst stände eine Null im Nenner.

Es gelten folgende Rechenregeln:

$$\begin{aligned} (r\cdot A)^{-1} &= \tfrac{1}{r}\cdot A^{-1} \\ \left(A^{-1}\right)^{-1} &= A \\ (A\bullet B)^{-1} &= B^{-1}\bullet A^{-1} \end{aligned}$$

Betrachten wir noch einmal unsere 2×2 -Matrix von vorhin, um die Formel zu überprüfen:

$$A=\begin{pmatrix} 2 & 4 \\ 8 & 6 \end{pmatrix}\Rightarrow \det(A)=2\cdot 6-8\cdot 4=-20$$

$$A^{-1}=\frac{1}{\det(A)}\cdot\begin{pmatrix} 6 & -4 \\ -8 & 2 \end{pmatrix}=-\frac{1}{20}\cdot\begin{pmatrix} 6 & -4 \\ -8 & 2 \end{pmatrix}=\begin{pmatrix} -0.3 & 0.2 \\ 0.4 & -0.1 \end{pmatrix}$$

$$A\bullet A^{-1}=\begin{pmatrix} 2 & 4 \\ 8 & 6 \end{pmatrix}\bullet\begin{pmatrix} -0.3 & 0.2 \\ 0.4 & -0.1 \end{pmatrix}=\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}=E$$

Transponieren einer Matrix

Eine Matrix zu *transponieren* bedeutet ihre Zeilen und Spalten zu vertauschen. Der Eintrag a_{ij} wird dadurch zum Eintrag a_{ji} . Die Transponierte der Matrix A schreibt man A^T .

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}^T=\begin{pmatrix} a_{11} & \cdots & a_{m1} \\ \vdots & \ddots & \vdots \\ a_{1n} & \cdots & a_{nm} \end{pmatrix}$$

Zum Beispiel:

$$\begin{pmatrix} 1 & 3 & 3 \\ 7 & 4 & 2 \end{pmatrix}^T=\begin{pmatrix} 1 & 7 \\ 3 & 4 \\ 3 & 2 \end{pmatrix}$$

Wenn eine Matrix A mit ihrer Transponierten identisch ist ($A^T = A$), was natürlich nur bei quadratischen Matrizen der Fall sein kann, dann nennt man die Matrix *symmetrisch*. Ein Beispiel für eine symmetrische Matrix ist die Einheitsmatrix E.

Für das Rechnen mit transponierten Matrizen gilt:

$$\begin{aligned}(A+B)^T &= A^T + B^T \\ r \cdot A^T &= (r \cdot A)^T \\ (A^T)^T &= A \\ (A \bullet B)^T &= B^T \bullet A^T\end{aligned}$$

Außerdem gilt folgender Zusammenhang zwischen dem Invertieren und dem Transponieren: es macht keinen Unterschied, ob eine Matrix zuerst transponiert und dann invertiert wird oder umgekehrt. Das heißt:

$$(A^T)^{-1} = (A^{-1})^T$$

2.5 Transformation

Bisher haben wir zwar schon ganz fleißig mit Matrizen herumgerechnet, aber ich habe Ihnen noch nicht viel darüber gesagt, wofür man sie eigentlich genau einsetzt. Dafür müssen wir ein bisschen weiter ausholen.

3D-Objekte werden meistens als Menge von Dreiecken gespeichert. Genauer gesagt speichert man (unter Anderem) die Koordinaten der Eckpunkte dieser Dreiecke. Dabei handelt es sich normalerweise um 3D-Vektoren. Nun möchte man das Objekt irgendwo in der Szene platzieren, es in eine bestimmte Richtung schauen lassen oder seine Größe anpassen. Dazu müssen die Eckpunkte der Dreiecke *transformiert* werden. Eine Transformation kann zum Beispiel eine Verschiebung (*Translation*), eine Drehung (*Rotation*), eine Vergrößerung/Verkleinerung (*Skalierung*), eine Scherung oder eine beliebige Kombination davon sein. Die gespeicherten Daten dienen sozusagen nur als Schablone für das Objekt, und durch eine geeignete Transformation kann es in der Szene platziert und gerendert werden.

Solche Transformationen lassen sich mit Hilfe von Matrizen durchführen. Eine Matrix steht dabei für eine bestimmte Transformation.

2.5.1 Transformation eines Punkts

Wir transformieren einen Punkt P, dessen Ortsvektor $\vec{P}$ wir kennen, mit Hilfe einer Matrix A, indem wir $\vec{P}$ mit A multiplizieren. Das Ergebnis ist der Ortsvektor des transformierten Punkts Q. Der Punkt P wird also mit der Matrix A auf den Punkt Q abgebildet.

Aber wie kann man einen Vektor mit einer Matrix multiplizieren? Dazu fassen wir den Vektor einfach als Matrix mit nur einer einzigen Zeile auf und führen dann die normale Matrixmultiplikation durch:

$$\vec{P} \bullet A = (p_1,\ p_2,\ p_3) \bullet \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = (q_1,\ q_2,\ q_3) = \vec{Q}$$

wobei:

$$q_1 = p_1 \cdot a_{11} + p_2 \cdot a_{21} + p_3 \cdot a_{31}$$
$$q_2 = p_1 \cdot a_{12} + p_2 \cdot a_{22} + p_3 \cdot a_{32}$$
$$q_3 = p_1 \cdot a_{13} + p_2 \cdot a_{23} + p_3 \cdot a_{33}$$

Die Einträge a_{ij} der Matrix A bestimmen, wie der Punkt P auf den Punkt Q abgebildet wird. Eine solche Transformation nennt man übrigens eine *lineare Transformation*. Aber wie funktioniert das genau? Wie können wir damit Drehungen oder andere Transformationen hinbekommen?

2.5.2 Matrizen als Koordinatensysteme

Schauen Sie sich noch einmal genau an, wie die Koordinaten q_1, q_2, q_3 des transformierten Punkts Q berechnet werden. Man kann die Rechnung auch in Vektorform hinschreiben:

$$\vec{Q} = p_1 \cdot \begin{pmatrix} a_{11} \\ a_{12} \\ a_{13} \end{pmatrix} + p_2 \cdot \begin{pmatrix} a_{21} \\ a_{22} \\ a_{23} \end{pmatrix} + p_3 \cdot \begin{pmatrix} a_{31} \\ a_{32} \\ a_{33} \end{pmatrix}$$

p_1, p_2, p_3 sind die Koordinaten des untransformierten Punkts P, und die drei Zeilenvektoren der Matrix A bilden nun die Achsen (Basisvektoren) eines Koordinatensystems. Die x-Achse ist die erste, die y-Achse die zweite und die z-Achse die dritte Zeile. Die Matrix bestimmt also, in welche Richtungen diese Achsen zeigen und wie lang sie sind.

Indem wir die Achsen in andere Richtungen zeigen lassen, können wir Objekte drehen oder kippen, und indem wir die Länge der Achsen ändern, können wir Streckungen, Stauchungen, Verkleinerungen und Vergrößerungen erzielen. Durch Umdrehen einer Achse erreicht man eine Spiegelung. Wie man die entsprechenden Matrizen genau erzeugt, werden wir später sehen.

Die Einheitsmatrix E stellt sozusagen den „Normalfall“ eines Koordinatensystems dar. Auf Grund ihrer Eigenschaft als neutrales Element bildet sie jeden Punkt auf sich selbst ab, führt also keine Transformation durch:

$$\vec{P} \bullet E = (p_1,\ p_2,\ p_3) \bullet \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = (p_1,\ p_2,\ p_3) = \vec{P}$$

Die Achsen sind die gewöhnlichen Achsen aus unserem Koordinatensystem, also:

$$\vec{x} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \quad \vec{y} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad \vec{z} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

2.5.3 Skalierung

Fangen wir mit der einfachsten aller Transformationen an: der Skalierung. Skalieren bedeutet vergrößern oder verkleinern. Dazu gehen wir von der Einheitsmatrix aus und verändern die Länge der Basisvektoren. Wir multiplizieren sie mit einem Skalierungsfaktor *s* und ändern dadurch ihre Längen und drehen eventuell ihre Richtungen um. $|s| > 1$ bedeutet eine Vergrößerung, $0 < |s| < 1$ bedeutet eine Verkleinerung. Wird *s* negativ, dann bedeutet das zusätzlich eine Spiegelung am Koordinatenursprung. Unsere Skalierungsmatrix für den Skalierungsfaktor *s* sieht also so aus:

$$M_{\text{Skalierung}} = \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & s \end{pmatrix}$$

Wenn alle drei Achsen gleichmäßig skaliert werden, nennt man das eine *uniforme Skalierung*. Natürlich kann man auch verschiedene Skalierungsfaktoren für die Achsen verwenden, wodurch das Objekt gestaucht oder gestreckt wird.

$$M_{\text{Skalierung}} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix}$$

Transformieren wir nun einen Punkt mit dieser Matrix, dann bewirkt das, dass seine Koordinaten mit den entsprechenden Skalierungsfaktoren multipliziert werden. Da die übrigen Einträge der Matrix null sind, geschieht sonst nichts:

$$(x,\ y,\ z) \bullet \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix} = (x \cdot s_x,\ y \cdot s_y,\ z \cdot s_z)$$

An der folgenden Abbildung können Sie sehen, welche Auswirkungen verschiedene Skalierungsmatrizen auf ein Objekt haben:

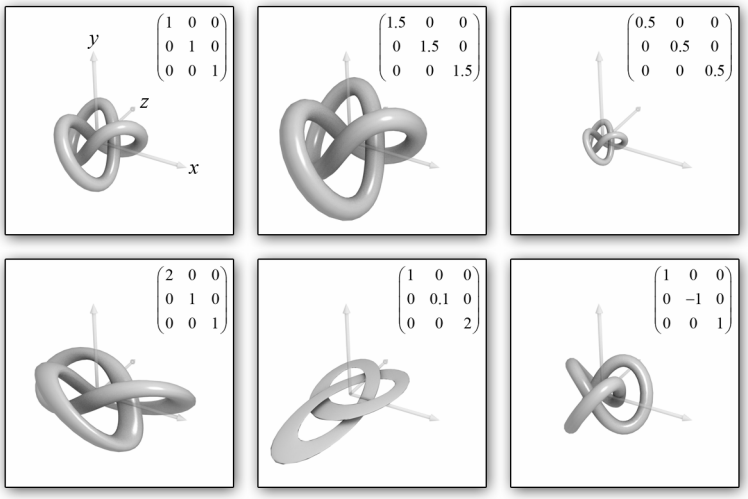


Abbildung 2.21 Die Auswirkungen verschiedener Skalierungsmatrizen auf ein 3D-Objekt

Die Skalierung geschieht mit solchen Matrizen immer um den Koordinatenursprung herum. Das heißt der Punkt (0, 0, 0) bleibt immer der an derselben Stelle, egal wie die Skalierungsmatrix auch aussehen mag. Darum entwirft man 3D-Objekte meistens so, dass ihr Mittelpunkt bei (0, 0, 0) liegt, denn sonst würde sich das Objekt beispielsweise bei einer Vergrößerung vom Ursprung wegbewegen.

2.5.4 Rotation

Die nächste Transformation, die wir uns anschauen werden, ist die Rotation. Mit einer Rotationsmatrix dreht man ein Objekt um eine bestimmte Achse. Eine Vergrößerung oder Verkleinerung findet dabei nicht statt.

Um ein Objekt zu drehen, müssen wir die Achsen in der Matrix drehen. Am einfachsten ist es, wenn man sich erst einmal auf die Rotation im Zweidimensionalen beschränkt. Wir wollen die neuen Achsenvektoren $\vec{x}'$ und $\vec{y}'$ berechnen, die um einen Winkel α gegen den Uhrzeigersinn gedreht sind. Das geht mit Hilfe des Sinus und des Kosinus im Einheitskreis:

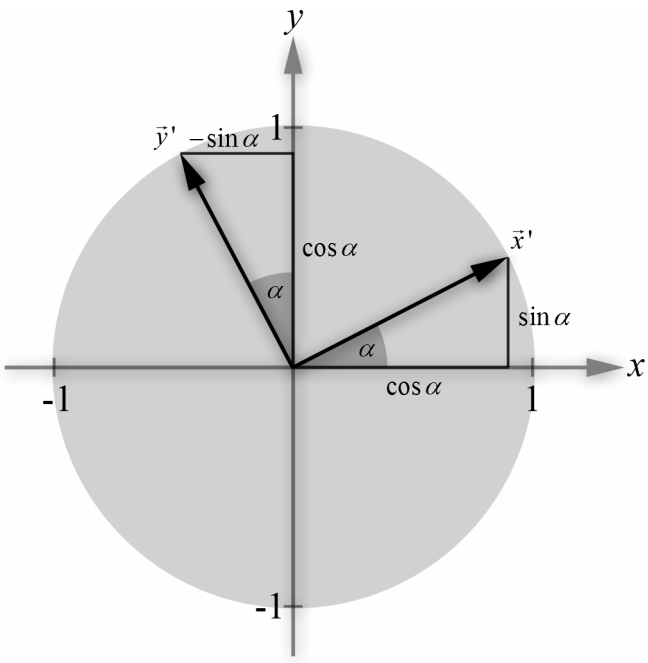


Abbildung 2.22 Die gedrehten Achsenvektoren lassen sich mit Sinus und Kosinus bestimmen.

Nun kann man ablesen:

$$\vec{x}' = \begin{pmatrix} \cos\alpha \\ \sin\alpha \end{pmatrix} \text{ und } \vec{y}' = \begin{pmatrix} -\sin\alpha \\ \cos\alpha \end{pmatrix}$$

Wenn wir diese Achsen jetzt in die Zeilen einer Matrix einsetzen, erhalten wir folgende 2D-Rotationsmatrix:

$$M_{\text{Rotation2D}} = \begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix}$$

Die neuen Achsen haben immer noch die Länge 1 (denn es gilt: $\sin^2\alpha + \cos^2\alpha = 1$), und sie sind auch senkrecht zueinander. Beides sind Merkmale einer Rotationsmatrix.

Wir haben jetzt eine Matrix für die Rotation im Zweidimensionalen. Zufälligerweise entspricht diese Rotation der Rotation um die *z*-Achse im Dreidimensionalen, denn dort werden ebenfalls nur die *x*- und *y*-Achsen gedreht, und die *z*-Achse bleibt gleich. Die Matrix für die 3D-Rotation um die *z*-Achse lautet also:

$$M_{\text{RotationZ}} = \begin{pmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin a & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Für die Rotation um die *x*- und *y*-Achse kann man analog vorgehen und erhält:

$$M_{\text{RotationX}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{pmatrix} \quad M_{\text{RotationY}} = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{pmatrix}$$

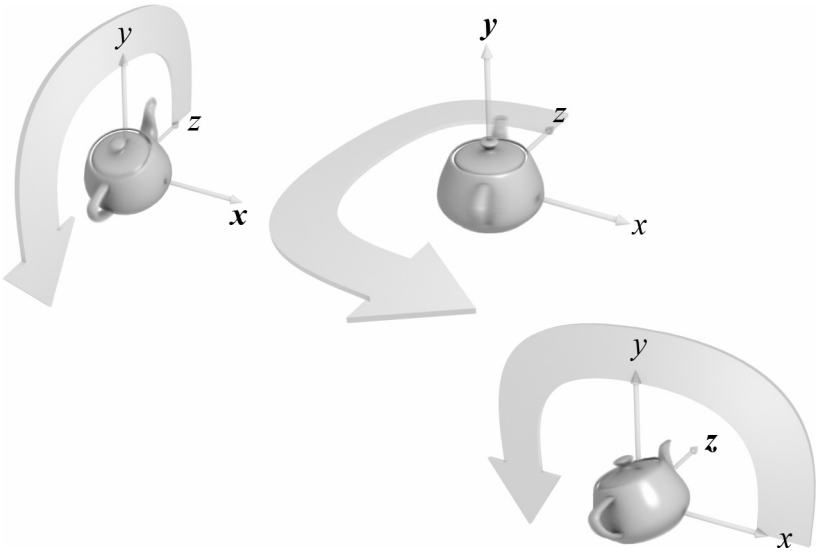


Abbildung 2.23 Rotation um die *x*-, *y*- und *z*-Achse

Möchte man um eine beliebige Achse drehen (also nicht um die *x*-, *y*- oder *z*-Achse), dann kann man die unten angegebene Rotationsmatrix anwenden. Hierbei stellt der Einheitsvektor $\vec{a} = (a_1,\ a_2,\ a_3)$ die Achse dar, um die gedreht werden soll:

$$M_{\text{RotationAchse}} = \begin{pmatrix} \cos\alpha + a_1^2(1-\cos\alpha) & a_1a_2(1-\cos\alpha) - a_3\sin\alpha & a_1a_3(1-\cos\alpha) + a_2\sin\alpha \\ a_2a_1(1-\cos\alpha) + a_3\sin\alpha & \cos\alpha + a_2^2(1-\cos\alpha) & a_2a_3(1-\cos\alpha) + a_1\sin\alpha \\ a_3a_1(1-\cos\alpha) - a_2\sin\alpha & a_3a_2(1-\cos\alpha) + a_1\sin\alpha & \cos\alpha + a_3^2(1-\cos\alpha) \end{pmatrix}$$

Diese Matrix sollten Sie auf jeden Fall auswendig lernen!

Das war natürlich nur ein Scherz. Die Matrix steht hier nur der Vollständigkeit halber. Wenn Sie Lust dazu haben, können Sie für $\vec{a}$ einmal die *x*-Achse $\vec{x} = (1,\ 0,\ 0)$ einsetzen und überprüfen, ob dann die oben angegebene Matrix $M_{\text{RotationX}}$ herauskommt.

2.5.5 Translation

Unter einer *Translation* versteht man eine Verschiebung. Translationen werden benötigt, um die Positionen von Objekten in einer Szene festlegen zu können. Dies erreicht man,

indem man einfach zu den Koordinaten jedes Punkts einen *Translationsvektor* hinzu addiert. Betrachten Sie die folgende Abbildung, auf der ein 2D-Objekt durch eine Translation an eine andere Stelle verschoben wird.

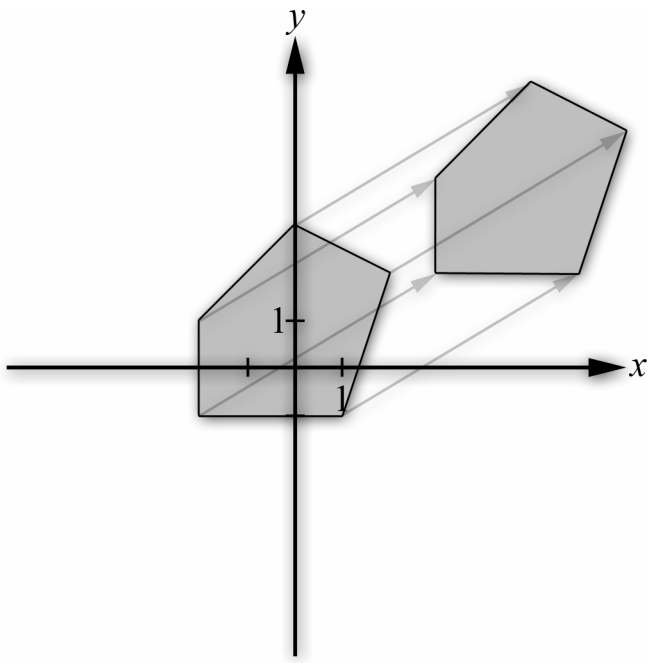


Abbildung 2.24 Das 2D-Objekt wird um den Translationsvektor $\vec{t} = (3,\ 2)$ verschoben, indem zu allen Punktkoordinaten dieser Vektor addiert wird.

Wie können wir eine Translation mit Hilfe einer Matrix durchführen? Schauen wir uns noch einmal an, wie ein Punkt *P* mit Hilfe einer Matrix *A* auf einen Punkt *Q* abgebildet wird:

$$\vec{Q} = p_1 \cdot \begin{pmatrix} a_{11} \\ a_{12} \\ a_{13} \end{pmatrix} + p_2 \cdot \begin{pmatrix} a_{21} \\ a_{22} \\ a_{23} \end{pmatrix} + p_3 \cdot \begin{pmatrix} a_{31} \\ a_{32} \\ a_{33} \end{pmatrix}$$

Es fehlt uns die Möglichkeit, einen konstanten Vektor hinzu zu addieren. Was wir bräuchten, wäre etwas wie:

$$\vec{Q} = p_1 \cdot \begin{pmatrix} a_{11} \\ a_{12} \\ a_{13} \end{pmatrix} + p_2 \cdot \begin{pmatrix} a_{21} \\ a_{22} \\ a_{23} \end{pmatrix} + p_3 \cdot \begin{pmatrix} a_{31} \\ a_{32} \\ a_{33} \end{pmatrix} + \vec{t}$$

Diese Art der Transformation, bei der ein Translationsvektor hinzuaddiert wird, nennt man *affine Transformation*.

Homogene Koordinaten

Um auch Translationen mit Hilfe von Matrizen ausdrücken zu können, fügt man eine weitere Dimension hinzu. Aus einem 3D-Vektor wird ein 4D-Vektor, und zwar durch Anfügen einer *homogenen Koordinate*, die wir *w* nennen. Wir legen fest, dass sie den Wert 1 haben soll. Aus dem 3D-Vektor (*x*, *y*, *z*) machen wir den 4D-Vektor (*x*, *y*, *z*, 1).

Um einen 4D-Vektor zu transformieren, brauchen wir nun eine 4×4 -Matrix. Die können wir leicht aus einer 3×3 -Matrix erzeugen, indem wir die 3×3 -Matrix links oben in die 4×4 -Einheitsmatrix einsetzen. Für einen 4D-Vektor kann man die Transformation jetzt wie folgt schreiben:

$$\vec{Q} = p_1 \cdot \begin{pmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{14} \end{pmatrix} + p_2 \cdot \begin{pmatrix} a_{21} \\ a_{22} \\ a_{23} \\ a_{24} \end{pmatrix} + p_3 \cdot \begin{pmatrix} a_{31} \\ a_{32} \\ a_{33} \\ a_{34} \end{pmatrix} + \underbrace{p_4}_{1} \cdot \begin{pmatrix} a_{41} \\ a_{42} \\ a_{43} \\ a_{44} \end{pmatrix}$$

p_4 ist die *w*-Koordinate, bei der wir festgelegt haben, dass sie 1 sein soll. Das heißt, dass der letzte Vektor (die vierte Zeile der Matrix) bei der Transformation einfach hinzu addiert wird, und das ist doch genau das, was wir erreichen wollten. Wir können also die vierte Zeile der Transformationsmatrix als Translationsvektor verwenden. Eine Translationsmatrix sieht demnach wie folgt aus, wobei $\vec{t} = (t_x,\ t_y,\ t_z)$ der Translationsvektor ist:

$$M_{\text{Translation}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{pmatrix}$$

$$(x,\ y,\ z,\ 1) \bullet M_{\text{Translation}} = (x + t_x,\ y + t_y,\ z + t_z,\ 1)$$

Die Transformationsmatrizen, die wir schon kennen gelernt haben (Skalierung, Rotation), können wie oben gezeigt ganz leicht zu 4×4 -Matrizen erweitert werden. Eine Skalierungsmatrix sieht dann zum Beispiel so aus:

$$M_{\text{Skalierung}} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Transformation eines Richtungsvektors

Ein Richtungsvektor soll bei der Transformation nicht verschoben werden. Für einen Richtungsvektor machen nur Rotationen oder Skalierungen einen Sinn. Darum setzen wir in diesem Fall die *w*-Koordinate auf null. Dadurch fällt die Translation weg.

Ortsvektor: aus $(x,\ y,\ z)$ wird $(x,\ y,\ z,\ 1)$.

Richtungsvektor: aus $(x,\ y,\ z)$ wird $(x,\ y,\ z,\ 0)$.

Projektion

Wenn wir den 3D-Ortsvektor $\vec{P}$ eines Punkts P zu einem 4D-Vektor mit $w = 1$ erweitern ($\vec{P}'$) und diesen dann mit einer 4×4-Matrix transformieren, dann erhalten wir wieder einen 4D-Vektor ($\vec{Q}'$). Eigentlich wollen wir aber einen 3D-Vektor ($\vec{Q}$) und müssen daher die vierte Koordinate wieder loswerden. Wir haben gesagt, dass diese Koordinate den Wert 1 haben soll. Wenn das nach der Transformation der Fall ist, können wir die vierte Koordinate einfach wegfallen lassen und sind fertig. Wenn sie aber einen anderen Wert als 1 hat, führen wir zuerst eine *Projektion* durch. Das bedeutet: wir dividieren den ganzen Vektor durch seine *w*-Koordinate, die nach der Division durch sich selbst natürlich wieder 1 ist.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

Aus $\vec{P} = (p_1,\ p_2,\ p_3)$

wird $\vec{P}' = (p_1,\ p_2,\ p_3,\ 1)$.

Transformation: $\vec{Q}' = \vec{P}' \bullet A = (q_1,\ q_2,\ q_3,\ q_4)$

wobei:
$$q_1 = p_1 \cdot a_{11} + p_2 \cdot a_{21} + p_3 \cdot a_{31} + a_{41}$$
$$q_2 = p_1 \cdot a_{12} + p_2 \cdot a_{22} + p_3 \cdot a_{32} + a_{42}$$
$$q_3 = p_1 \cdot a_{13} + p_2 \cdot a_{23} + p_3 \cdot a_{33} + a_{43}$$
$$q_4 = p_1 \cdot a_{14} + p_2 \cdot a_{24} + p_3 \cdot a_{34} + a_{44}$$

Projektion: $$\frac{\vec{Q}'}{q_4} = \left(\frac{q_1}{q_4},\ \frac{q_2}{q_4},\ \frac{q_3}{q_4},\ 1 \right) \Rightarrow \vec{Q} = \left(\frac{q_1}{q_4},\ \frac{q_2}{q_4},\ \frac{q_3}{q_4} \right)$$

Schauen wir uns dazu einmal ein Beispiel an. Wir möchten den Punkt *P* mit dem Ortsvektor $\vec{P} = (2, -1, 4)$ mit einer Matrix *A* transformieren.

$$A = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Aus $\vec{P} = (2,\ -1,\ 4)$

wird $\vec{P}' = (2,\ -1,\ 4,\ 1).$

Transformation: $\vec{Q}' = \vec{P}' \bullet A = \left(4,\ -2,\ 7,\ \underbrace{4}_{w}\right)$

Projektion: $\frac{\vec{Q}'}{4} = (1,\ -0.5,\ 1.75,\ 1) \Rightarrow \vec{Q} = (1,\ -0.5,\ 1.75)$

Da die *w*-Koordinate nach der Transformation 4 ist, muss $\vec{Q}'$ durch 4 dividiert werden. Der transformierte 3D-Punkt hat nun also die Koordinaten (1, –0.5, 1.75).

2.5.6 Aneinanderreihen von Transformationen

Matrizen zur Transformation zu verwenden hat einen großen Vorteil. Man kann nämlich beliebige Transformationen „verketten“ und erhält dann eine einzige neue Transformation, die genau dieselbe Wirkung hat, als ob die einzelnen Transformationen hintereinander ausgeführt worden wären.

Das Aneinanderreihen von Transformationen funktioniert mit der Matrixmultiplikation. Seien $M_1,\ M_2,\ \ldots,\ M_n$ die Matrizen für diese Transformationen, dann können wir den Vektor $\vec{P}$ auf den Vektor $\vec{Q}$ abbilden, indem wir ihn nacheinander mit allen Matrizen transformieren:

$$\vec{Q} = \left(\left(\vec{P} \bullet M_1\right) \bullet M_2\right) \bullet \ldots) \bullet M_n$$

Man kann aber genauso gut, auf Grund der Assoziativregeln, die Matrizen miteinander multiplizieren und dann den Vektor $\vec{P}$ mit dem Produkt dieser Matrizen transformieren:

$$M_{\text{Gesamt}} = M_1 \bullet M_2 \bullet \ldots \bullet M_n$$
$$\vec{Q} = \vec{P} \bullet M_{\text{Gesamt}}$$

Was hat man dadurch gewonnen? Einerseits braucht man mehrere Transformationen, die auf einen Punkt angewendet werden sollen, nicht als Spezialfall zu betrachten. Andererseits kann man beliebig viele Transformationen hintereinander ausführen, ohne bei der Transformation selbst mehr Rechenschritte ausführen zu müssen. Bei der Darstellung komplexer 3D-Objekte, die aus zehntausenden Punkten bestehen, macht dies einen großen Unterschied. Es muss lediglich das Produkt der Transformationsmatrizen berechnet werden, und das auch nur ein einziges Mal.

Die (richtige) Reihenfolge

Da die Matrixmultiplikation nicht kommutativ ist, ist die Reihenfolge relevant, in der die Matrizen miteinander multipliziert werden. Es macht zum Beispiel einen Unterschied, ob man zuerst dreht und dann verschiebt oder umgekehrt.

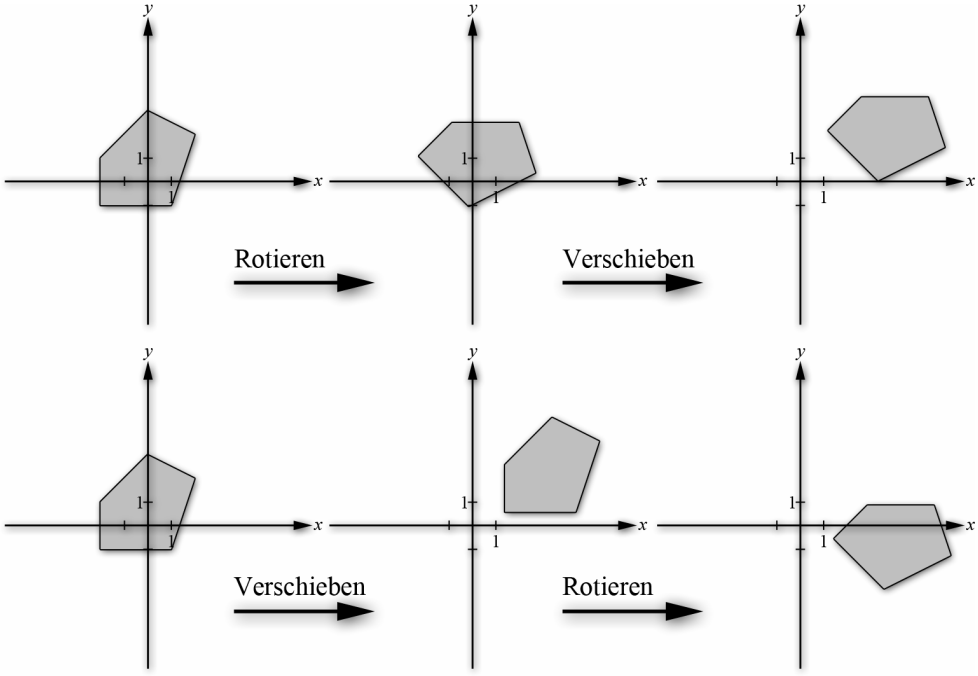


Abbildung 2.25 Oben wird zuerst rotiert und dann verschoben, unten umgekehrt. Die Ergebnisse unterscheiden sich voneinander.

Für gewöhnlich verwendet man folgende Reihenfolge: zuerst skalieren, dann rotieren und zum Schluss verschieben. Wenn man eine uniforme Skalierung verwendet, kann man die Skalierung und die Rotation auch vertauschen.

$$M = M_{Skalierung} \bullet M_{Rotation} \bullet M_{Translation}$$

2.5.7 Die Rolle der inversen und der transponierten Matrix

Welche Auswirkung hat es, wenn wir eine Matrix M invertieren und mit M^{-1} Punkte transformieren? Wie man sich gut vorstellen kann, wird dadurch die Transformation von M „rückgängig gemacht“. Wenn M zum Beispiel eine Verschiebung um 10 Einheiten nach rechts bewirkt, dann verschiebt M^{-1} um 10 Einheiten nach links. Wenn M um das Zweifache vergrößert, dann verkleinert M^{-1} um das Zweifache, und so weiter.

Die Multiplikation mit der normalen Matrix bedeutet eine Transformation aus dem „lokalen Raum“ in den „absoluten Raum“. Die Multiplikation mit der invertierten Matrix bedeutet hingegen das Umgekehrte, nämlich eine Transformation aus dem absoluten Raum in den lokalen Raum.

Was heißt das jetzt genau? Stellen wir uns ein 3D-Objekt vor. Es besteht aus Dreiecken. Irgendwo sind die Koordinaten der Eckpunkte dieser Dreiecke gespeichert. Diese Koordinaten beziehen sich auf den lokalen Raum des Objekts. Der Punkt (0, 0, 0) ist der Mittelpunkt des Objekts – egal, wo das Objekt letztendlich in der Szene platziert wird. Die Platzierung in der Szene geschieht durch eine Transformationsmatrix, die die Position, die Größe und die Orientierung des Objekts speichert. Wenn das Objekt gerendert wird, werden alle Eckpunkte mit dieser Matrix multipliziert. Dadurch werden sie vom lokalen Raum des Objekts in den absoluten Raum der Szene transformiert. Mit der Transformationsmatrix kann man also die Frage beantworten: „Welche absoluten Koordinaten hat der zum Objekt relative Punkt *P*?“. Umgekehrt kann man mit der inversen Transformationsmatrix für einen Punkt mit absoluten Koordinaten dessen zum Objekt relative Koordinaten ermitteln.

Absolute Koordinaten bezeichnet man auch als *Weltkoordinaten* und zu einem Objekt relative Koordinaten als *Objektkoordinaten*, *lokale Koordinaten* oder *Modellkoordinaten*.

Wahrscheinlich klingt das alles noch sehr verwirrend. Manchmal sagt eine Abbildung mehr als tausend Worte.

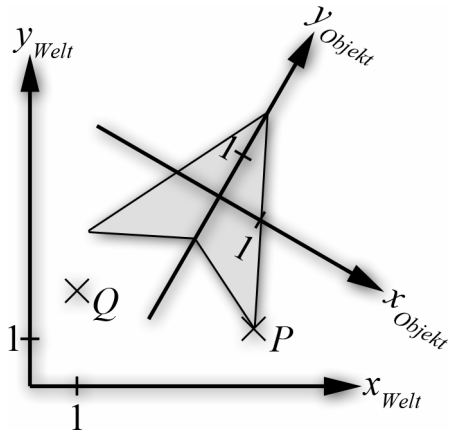


Abbildung 2.26 Weltkoordinatensystem und Objektkoordinatensystem

In der Abbildung ist der Punkt *P* markiert. Er ist ein Eckpunkt des dargestellten 2D-Objekts. Seine Objektkoordinaten sind (2, –2). Würde man das 2D-Objekt in einer Datei abspeichern, dann würde man diese Koordinaten hineinschreiben. Durch eine Transformationsmatrix wird das Objekt nun in der Szene platziert. Diese Matrix führt eine Drehung um 30° im Uhrzeigersinn und eine Verschiebung um (4, 4) durch. Die Objektkoordinaten (0, 0) entsprechen also den Weltkoordinaten (4, 4).

Die Weltkoordinaten des Punkts *P* finden wir heraus, indem wir seine Objektkoordinaten (2, –2) mit der Transformationsmatrix des Objekts multiplizieren. Darum nennt man diese Matrix auch *Weltmatrix*. Probieren wir das einmal aus:

$$M = \begin{pmatrix} \cos(-30°) & \sin(-30°) & 0 \\ -\sin(-30°) & \cos(-30°) & 0 \\ 0 & 0 & 1 \end{pmatrix} \bullet \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 4 & 4 & 1 \end{pmatrix} \approx \begin{pmatrix} 0.866 & -0.5 & 0 \\ 0.5 & 0.866 & 0 \\ 4 & 4 & 1 \end{pmatrix}$$

$$(2,\ -2,\ 1) \bullet M = (4.732,\ 1.268,\ 1)$$

Die Weltkoordinaten des Punkts *P* sind also (4.732, 1.268). Wenn man sich die Abbildung anschaut, sieht man, dass diese Koordinaten richtig sind. Bei dieser Rechnung für 2D-Vektoren wurde übrigens dasselbe getan wie schon zuvor für 3D-Vektoren: um auch Translationen, also Verschiebungen, mit Hilfe von Matrizen durchführen zu können, erhält der Vektor eine zusätzliche homogene Koordinate.

Der zweite markierte Punkt ist *Q*. Seine Weltkoordinaten lauten (1, 2). Wir wollen nun seine Objektkoordinaten $\vec{x}$ herausfinden. Hier kommt die inverse Weltmatrix ins Spiel. Wenn wir den Punkt mit dieser Matrix transformieren, erhalten wir die gesuchten Objektkoordinaten.

Gesucht sind die Objektkoordinaten $\vec{x}$. Für sie muss gelten:

$$\vec{x} \bullet M = (1,\ 2,\ 1)$$

Wir multiplizieren die Gleichung mit M^{-1} (von rechts):

$$\vec{x} \bullet \underbrace{M \bullet M^{-1}}_{=E} = (1,\ 2,\ 1) \bullet M^{-1}$$

$$\vec{x} = (1,\ 2,\ 1) \bullet M^{-1}$$

$$\vec{x} \approx (1,\ 2,\ 1) \bullet \begin{pmatrix} 0.866 & 0.5 & 0 \\ -0.5 & 0.866 & 0 \\ -1.464 & -5.454 & 1 \end{pmatrix} \approx (-1.598,\ -3.232,\ 1)$$

Die Objektkoordinaten des Punkts *Q* sind also (–1.598, –3.232). Auch diese Werte stimmen mit der Abbildung überein.

Invertieren leicht gemacht

Ist Ihnen aufgefallen, dass im Beispiel die Matrix *M* und ihre Inverse M^{-1} ziemlich ähnlich aussehen (mal abgesehen von der letzten Zeile)? Wenn wir nur die linke obere 2×2-Teilmatrix betrachten, dann sind dort einfach nur die Zeilen mit den Spalten vertauscht worden. Dieses Vertauschen der Zeilen mit den Spalten kennen wir ja schon als Transponieren. Ist das Zufall, oder gibt es da einen Zusammenhang?

Ja, einen solchen Zusammenhang gibt es. Für eine *orthogonale Matrix M* gilt:

$$M^{-1} = M^T$$

Eine quadratische Matrix heißt orthogonal, wenn ihre Spaltenvektoren normiert und paarweise senkrecht zueinander sind. Das Produkt zweier orthogonaler Matrizen ist wiederum eine orthogonale Matrix. Durch die Einschränkungen können mit orthogonalen Matrizen nur bestimmte Arten von Transformationen durchgeführt werden, nämlich so genannte *Kongruenzabbildungen*. Eine Kongruenzabbildung lässt Winkel und Abstände zwischen Punkten unverändert. Dazu gehören Drehungen und Spiegelungen, nicht jedoch Skalierungen, da hier Abstände beeinflusst werden.

Wenn wir also wissen, dass eine Matrix orthogonal ist, dann können wir sie invertieren, indem wir sie einfach transponieren. Das geht wesentlich schneller, da man nur ein paar Werte vertauschen muss und keine Determinantenberechnungen und Divisionen notwendig sind. Bei orthogonalen Matrizen ist die Determinante übrigens immer 1 oder –1.

Unsere Beispielmatrix M ist scheinbar nicht orthogonal, da die letzte Zeile nicht durch einfaches Transponieren zu ermitteln ist. Das liegt daran, dass diese Matrix zusätzlich zur Rotation noch eine Translation durchführt. Hier noch einmal die beiden Matrizen:

$$M = \begin{pmatrix} 0.866 & -0.5 & 0 \\ 0.5 & 0.866 & 0 \\ 4 & 4 & 1 \end{pmatrix} \quad M^{-1} \approx \begin{pmatrix} 0.866 & 0.5 & 0 \\ -0.5 & 0.866 & 0 \\ -1.464 & -5.454 & 1 \end{pmatrix}$$

Wir können M in eine Rotations- und eine Translationskomponente aufteilen:

$$M = \underbrace{\begin{pmatrix} 0.866 & -0.5 & 0 \\ 0.5 & 0.866 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{\text{Rotation}} \bullet \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 4 & 4 & 1 \end{pmatrix}}_{\text{Translation}}$$

Nun invertieren wir:

$$M^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 4 & 4 & 1 \end{pmatrix}^{-1} \bullet \begin{pmatrix} 0.866 & -0.5 & 0 \\ 0.5 & 0.866 & 0 \\ 0 & 0 & 1 \end{pmatrix}^{-1}$$

Denn für Matrizen gilt:

$$\left(A \bullet B\right)^{-1} = B^{-1} \bullet A^{-1}$$

Die Rotationsmatrix ist orthogonal und kann daher ganz leicht durch Transponieren invertiert werden. Die Translationsmatrix ist zwar nicht orthogonal, lässt sich aber trotzdem sehr einfach invertieren, da sie „fast“ die Einheitsmatrix ist. Wir müssen nur den Translationsvektor umkehren, also um (–4, –4) statt um (4, 4) verschieben. Damit ergibt sich dann:

$$M^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 4 & 4 & 1 \end{pmatrix}^{-1} \bullet \begin{pmatrix} 0.866 & -0.5 & 0 \\ 0.5 & 0.866 & 0 \\ 0 & 0 & 1 \end{pmatrix}^{T}$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -4 & -4 & 1 \end{pmatrix} \bullet \begin{pmatrix} 0.866 & 0.5 & 0 \\ -0.5 & 0.866 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0.866 & 0.5 & 0 \\ -0.5 & 0.866 & 0 \\ -1.464 & -5.454 & 1 \end{pmatrix}$$

Somit haben wir ein Verfahren, mit dem wir sehr schnell Matrizen invertieren können, die Kombinationen aus Rotationen (oder Spiegelungen) und Translationen enthalten. Das ist bei den meisten „üblichen“ Transformationsmatrizen für Objekte der Fall, sofern man auf Skalierungen verzichtet.

2.5.8 Hierarchische Transformationen

Oft hat man es mit Objekten zu tun, die aus mehreren Teilen bestehen, die alle einzeln bewegt werden können. Dabei können diese Teile in einer Art hierarchischen Beziehung zueinander stehen. Nehmen wir als Beispiel einen Panzer. Ein Teilobjekt ist das Fahrgestell (die „Wanne“). Auf dem Fahrgestell befindet sich der drehbare Turm. Der Turm enthält wiederum die Kanone, die sich anheben und absenken lässt. Weitere Teilobjekte sind die Laufräder, die die Ketten antreiben. Das Objekt Panzer kann man als eine hierarchische Anordnung der Teilobjekte ansehen. Dies entspricht einem Baum.

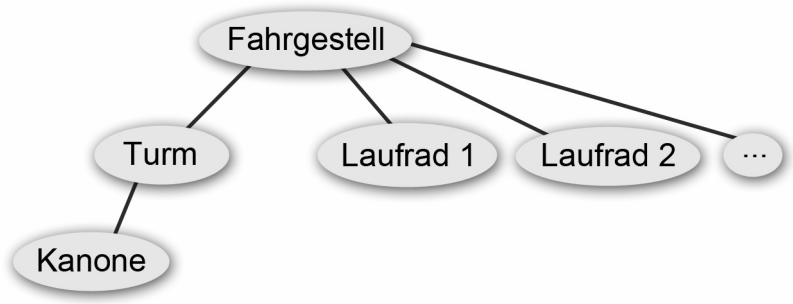


Abbildung 2.27 Hierarchie von Teilobjekten als Baum

Das Fahrgestell ist das *Wurzelobjekt* (*root object*). Die Kanone ist ein *Kindobjekt* (*child object*) des Turms. Das Fahrgestell ist das *Vaterobjekt* (*parent object*) des Turms. Turm und Laufräder sind *Geschwisterobjekte* (*sibling objects*) da sie dasselbe Vaterobjekt haben.

Die hierarchischen Beziehungen können wir dadurch zum Ausdruck bringen, dass eine Transformation eines Objekts sich auch auf alle Kinder dieses Objekts auswirkt. Wenn wir beispielsweise das Fahrgestell drehen, dann dreht sich auch der Turm mit (wir wollen einmal vernachlässigen, dass moderne Panzer in der Lage sind, den Turm und die Kanone unabhängig vom Fahrgestell stets in dieselbe Richtung zeigen zu lassen). Wenn jedoch der

Turm gedreht wird, wirkt sich diese Drehung nicht auf das Fahrgestell aus, wohl aber auf die Kanone, da sie ein Kind des Turms ist.

Bei solch einer hierarchischen Transformation hat jedes Teilobjekt eine *lokale Transformationsmatrix*. Sie enthält die Transformation (die Position, die Rotation und die Skalierung) dieses Teilobjekts *relativ zum Vaterobjekt*. Nehmen wir an, die Kanone besäße die folgende lokale Transformationsmatrix:

$$M_{\text{Kanone lokal}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & -0.5 & 0 \\ 0 & 0.5 & 0.866 & 0 \\ 0 & 0.25 & 0.9 & 1 \end{pmatrix}$$

Das würde bedeuten, dass die Kanone aus der Sicht ihres Vaterobjekts, dem Turm, leicht nach oben zeigt und sie sich 0.25 Einheiten „über“ dessen Mittelpunkt befindet und 0.9 Einheiten „vor“ ihm. Die Begriffe „über“ und „vor“ sind hier relativ und beziehen sich auf das Koordinatensystem des Turms.

Um nun die absolute Transformationsmatrix für die Kanone herauszufinden, also die Weltmatrix, mit der man die Kanone an der richtigen Stelle rendern könnte, wandert man den Baum hinauf bis zur Wurzel und multipliziert alle lokalen Matrizen:

$$M_{\text{Kanone absolut}} = M_{\text{Kanone lokal}} \bullet M_{\text{Turm lokal}} \bullet M_{\text{Fahrgestell lokal}}$$

Das Fahrgestell ist das Wurzelobjekt, darum ist die Multiplikationskette dort am Ende. Beim Wurzelobjekt stimmen lokale und absolute Matrix überein, da es hier kein übergeordnetes Objekt mehr gibt, auf das sich die Transformation beziehen könnte.

2.5.9 Kameratransformation

Eine sehr wichtige Transformation fehlt uns noch. Es ist die so genannte *Kameratransformation* oder *Sichttransformation*. Mit dieser Transformation ist es möglich, eine virtuelle Kamera in der Szene zu platzieren und die Objekte aus ihrer Sicht zu zeichnen. Die Kameratransformation transformiert Punkte aus dem Weltkoordinatensystem in das *Kamerakoordinatensystem*. Die dazugehörige Matrix nennt man dementsprechend *Kameramatrix* oder *Sichtmatrix*. Im Kamerakoordinatensystem sind die Koordinaten relativ zur Kamera. (–2, 4, 8) beschreibt beispielsweise einen Punkt, der sich zwei Einheiten links von der Kamera, vier Einheiten über ihr und acht Einheiten vor ihr befindet.

Eine Kameramatrix lässt sich sehr leicht generieren. Das nötige Handwerkszeug dafür haben Sie schon kennen gelernt. Wir betrachten die Kamera erst einmal als normales Objekt und berechnen eine Transformationsmatrix für sie, die ihre Position und ihre Ausrichtung beinhaltet. Dann muss diese Matrix nur noch invertiert werden. Die Transformationsmatrix kann man wieder in zwei Komponenten aufteilen: eine Rotationsmatrix und eine Translationsmatrix. Die Translationsmatrix hängt von der Kameraposition ab, und die

Rotationsmatrix von der Ausrichtung der Kamera. Die folgende Matrix steht für eine Kamera, die sich an der Position (7, 2, 42) befindet und nach links schaut:

$$M = \left(\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \bullet \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 7 & 2 & 42 & 1 \end{pmatrix} \right)^{-1} = \begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ -42 & -2 & 7 & 1 \end{pmatrix}$$

Die Rotationsmatrix (die linke Matrix in der großen Klammer) enthält in den Zeilen die Achsen der Kamera. Die erste Zeile enthält die *x*-Achse. Die Kamera schaut nach links, also zeigt ihre *x*-Achse in die Tiefe, darum ist sie (0, 0, 1). Die *y*-Achse der Kamera (zweite Zeile) zeigt nach oben, also (0, 1, 0), und ihre *z*-Achse zeigt nach links, also in Richtung (–1, 0, 0). Wenn man zwei dieser Achsen kennt, kann man die Dritte auch mit Hilfe des Kreuzprodukts ausrechnen.

Im Allgemeinen erhalten wir für eine Kamera mit den Achsen $\vec{x}$, $\vec{y}$, $\vec{z}$ (normiert und paarweise senkrecht) und der Position $\vec{p}$ die folgende Kameramatrix:

$$M_{\text{Kamera}} = \left(\begin{pmatrix} x_1 & x_2 & x_3 & 0 \\ y_1 & y_2 & y_3 & 0 \\ z_1 & z_2 & z_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \bullet \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ p_1 & p_2 & p_3 & 1 \end{pmatrix} \right)^{-1}$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -p_1 & -p_2 & -p_3 & 1 \end{pmatrix} \bullet \begin{pmatrix} x_1 & y_1 & z_1 & 0 \\ x_2 & y_2 & z_2 & 0 \\ x_3 & y_3 & z_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} x_1 & y_1 & z_1 & 0 \\ x_2 & y_2 & z_2 & 0 \\ x_3 & y_3 & z_3 & 0 \\ -\vec{x} \cdot \vec{p} & -\vec{y} \cdot \vec{p} & -\vec{z} \cdot \vec{p} & 1 \end{pmatrix}$$

Die Kameratransformation erfolgt normalerweise nach der Welttransformation. Die Eckpunkte von 3D-Objekten werden also zuerst mit der Weltmatrix des Objekts in das Weltkoordinatensystem transformiert und von dort mit Hilfe der Kameramatrix in das Kamerakoordinatensystem.

2.5.10 Perspektivische Transformation und Projektion

Zum Abschluss betrachten wir die perspektivische Transformation, die nach der Kameratransformation angewandt wird. Sie ist eine der wichtigsten Transformationen überhaupt, denn sie ermöglicht es, eine dreidimensionale Szene auf eine zweidimensionale Bildebene abzubilden. Sie ist aber auch die Transformation, die am schwierigsten zu verstehen ist.

Solange es noch keine (erschwinglichen) 3D-Bildschirme oder Holodecks gibt, kommt man ohne sie aber kaum aus.

2.5.10.1 Wir bauen eine Projektionsmatrix

Ganz einfach gesagt sorgt die perspektivische Transformation mit einer *Projektionsmatrix* dafür, dass weiter entfernte Objekte kleiner erscheinen als nahe Objekte. Eine solche Matrix werden wir nun versuchen aufzubauen.

Wie kann man es schaffen, dass ein Objekt kleiner wird? Das geht, indem man die Koordinaten der Punkte, aus denen es besteht, durch irgendeinen Wert größer als 1 dividiert. Dividieren wir die Koordinaten zum Beispiel durch 2, dann ist das Objekt nur noch halb so groß. Nun wollen wir erreichen, dass ein Objekt umso kleiner erscheint, desto tiefer es sich in der Szene befindet (aus der Sicht der Kamera). Diese Tiefe wird durch die *z*-Koordinate beschrieben. Wir können also die *z*-Koordinate eines Punkts verwenden, um zu bestimmen, durch welchen Wert die Koordinaten dividiert werden sollen. Je größer *z*, desto größer soll dieser Wert sein.

Erinnern Sie sich noch an die homogene Koordinate, mit der wir aus einem dreidimensionalen Vektor einen vierdimensionalen Vektor gemacht haben? Wir haben sie verwendet, um mit Hilfe von Matrizen auch Translationen durchführen zu können, und wir haben festgelegt, dass die homogene Koordinate (*w*-Koordinate) den Wert 1 haben soll. Wenn das nicht der Fall war, dann haben wir eine Projektion durchgeführt, haben also den ganzen Vektor durch *w* geteilt, damit *w* wieder 1 wird.

Wir haben durch *w* dividiert! Wir können die *w*-Koordinate also verwenden, um unsere Objekte in der Tiefe kleiner erscheinen zu lassen. Dazu brauchen wir eine Matrix, die die *w*-Koordinate aus der Tiefe, also der *z*-Koordinate, berechnet. Eine ganz einfache Möglichkeit wäre die folgende Projektionsmatrix, die nur $w = z$ setzt und sonst nichts tut:

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Sehen Sie, was an dieser Matrix anders ist als beispielsweise an einer Rotationsmatrix? Bislang haben wir die rechte Spalte nie genutzt, sie war immer $(0,\ 0,\ 0,\ 1)^T$. Die rechte Spalte ist für die Berechnung der *w*-Koordinate des transformierten Vektors verantwortlich. Die oben gezeigte Matrix *M* lässt *x*, *y* und *z* unverändert und berechnet *w* aus $0 \cdot x + 0 \cdot y + 1 \cdot z + 0 \cdot w$, also *z*. Sie kopiert folglich die *z*-Koordinate in die *w*-Koordinate. Wenn wir statt der 1 einen größeren Wert einsetzen, dann werden die Objekte mit steigender Tiefe schneller kleiner.

Wir wollen nun ein paar Punkte mit dieser Matrix transformieren. Die Punkte haben alle dieselben *x*- und *y*-Koordinaten, nämlich (4, 3), und unterscheiden sich nur durch ihre Tiefe *z*. Wenn die Matrix das tut, was sie tun soll, dann müsste der tiefste Punkt nach der

Transformation und der Projektion am weitesten zum Koordinatenursprung und damit zum Bildmittelpunkt gerückt sein.

$$(4,\ 3,\ 1,\ 1) \bullet M = (4,\ 3,\ 1,\ 1) \qquad \Rightarrow \text{nach Projektion: } (4,\ 3,\ 1)$$
$$(4,\ 3,\ 2,\ 1) \bullet M = (4,\ 3,\ 2,\ 2) \qquad \Rightarrow \text{nach Projektion: } \left(\tfrac{4}{2},\ \tfrac{3}{2},\ \tfrac{2}{2}\right) = (2,\ 1.5,\ 1)$$
$$(4,\ 3,\ 3,\ 1) \bullet M = (4,\ 3,\ 3,\ 3) \qquad \Rightarrow \text{nach Projektion: } \left(\tfrac{4}{3},\ \tfrac{3}{3},\ \tfrac{2}{3}\right) \approx (1.333,\ 1,\ 1)$$
$$(4,\ 3,\ 10,\ 1) \bullet M = (4,\ 3,\ 10,\ 10) \qquad \Rightarrow \text{nach Projektion: } \left(\tfrac{4}{10},\ \tfrac{3}{10},\ \tfrac{2}{10}\right) = (0.4,\ 0.3,\ 1)$$

Wenn wir die transformierten Punkte auf den Bildschirm zeichnen würden, wozu wir nur die *x*- und *y*-Koordinaten verwenden (denn der Bildschirm kann nur ein zweidimensionales Bild anzeigen), dann würden die weiter entfernten Punkte näher am Bildmittelpunkt liegen als diejenigen, die näher an der Kamera sind. Die Matrix funktioniert also.

Es gibt aber noch ein paar Probleme. Was geschieht zum Beispiel mit Punkten, deren Tiefe null ist? Sie würden zu einer Division durch null führen. Oder was passiert, wenn ein Punkt eine negative *z*-Koordinate hat? Dann befindet er sich hinter der Kamera, und Teile der Szene, die sich hinter der Kamera befinden, sollten eigentlich nicht gezeichnet werden.

Noch einmal von vorne

Überlegen wir uns noch einmal, was wir eigentlich erreichen wollen. Eine dreidimensionale Szene soll perspektivisch auf eine zweidimensionale rechteckige Fläche abgebildet werden. Wir nennen diese Fläche die *Bildebene* (auch wenn sie eigentlich keine Ebene ist, da sie ja begrenzt ist).

Die Bildebene stellt man sich am besten wie ein Fenster vor, durch das man nach draußen in die große weite 3D-Welt blickt. Das Fenster, also die Bildebene, hat eine Breite, eine Höhe und es befindet sich in einer gewissen Entfernung zu uns, dem Betrachter. Je näher wir an das Fenster heran gehen, desto größer wird der Ausschnitt der Welt, den wir dadurch sehen können. Wir wollen die Breite und die Höhe der Bildebene mit *W* (width) und *H* (height) bezeichnen und die Distanz zum Betrachter mit *D*.

Für die Bildebene definieren wir uns folgendes Koordinatensystem: Der Mittelpunkt liegt bei (0, 0), die linke obere Ecke bei (–1, 1) und die rechte untere Ecke bei (1, –1). Die *x*-Achse zeigt also wie gewohnt nach rechts und die *y*-Achse nach oben. Punkte, die außerhalb unserer Bildebene liegen, sind nicht sichtbar (im Fenstermodell würden sie von der Wand verdeckt, die das Fenster umgibt).

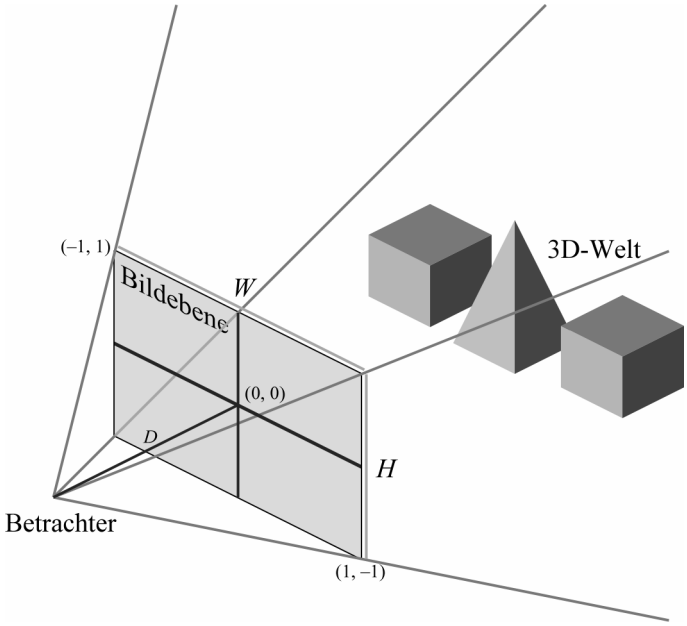


Abbildung 2.28 Die Bildebene mit der Breite *W* und der Höhe *H* kann man sich als Fenster vorstellen, durch das der Betrachter die 3D-Welt wahrnimmt. Seine Distanz zur Bildebene bezeichnen wir mit *D*.

Nun ist es unsere Aufgabe, für einen beliebigen Punkt in der 3D-Welt einen entsprechenden Punkt auf der zweidimensionalen Bildebene zu finden, ihn also perspektivisch auf die Bildebene zu projizieren.

Dazu ziehen wir eine Gerade zwischen der Position des Betrachters und dem Punkt. Dort, wo die Gerade die Bildebene schneidet, liegt der gesuchte projizierte Punkt. Die entsprechenden Berechnungen erfolgen mit dem *Strahlensatz*. In der nächsten Abbildung schauen wir uns der Einfachheit halber die Situation von oben an und beachten nur die *x*- und die *z*-Achse. Gegeben ist ein Punkt mit den Koordinaten (*x*, *z*), und wir suchen nun die projizierte *x*-Koordinate *x'* auf der Bildebene.

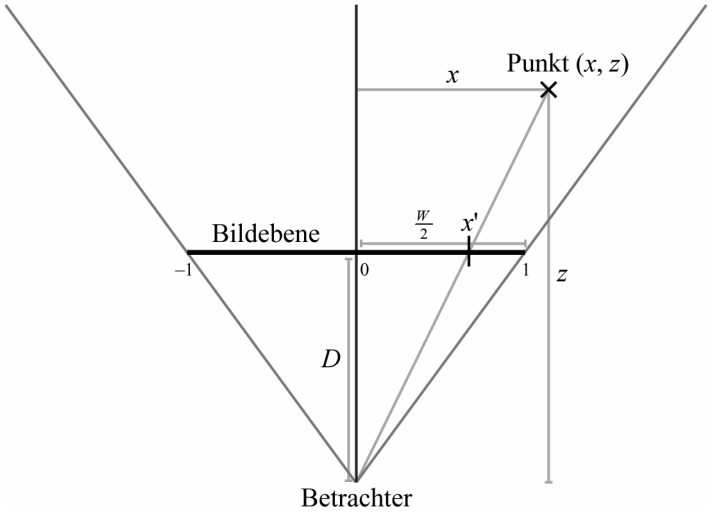


Abbildung 2.29 Projektion von oben betrachtet: Mit Hilfe des Strahlensatzes können wir die projizierte *x*-Koordinate *x*' des Punkts (*x*, *z*) berechnen. Alle Punkte, die auf der Geraden durch (*x*', *D*) und (*x*, *z*) liegen, werden auf denselben Punkt projiziert.

Laut Strahlensatz ist das Verhältnis zwischen *x*' und *D* gleich dem Verhältnis zwischen *x* und *z*. Daraus können wir *x*' berechnen:

$$\frac{x'}{D} = \frac{x}{z} \Rightarrow x' = \frac{x \cdot D}{z}$$

Wenn wir *x*' nun im Koordinatensystem der Bildebene haben wollen (–1 am linken Rand, 1 am rechten Rand), dann müssen wir noch durch die halbe Breite der Bildebene teilen und erhalten:

$$x_{\text{Bild}} = \frac{x \cdot \frac{2 \cdot D}{W}}{z}$$

Analoges gilt natürlich auch für die *y*-Koordinate:

$$y_{\text{Bild}} = \frac{y \cdot \frac{2 \cdot D}{H}}{z}$$

Später möchten wir die Projektion mit Hilfe einer Matrix durchführen. Wenn Sie sich die Berechnungen für x_{Bild} und y_{Bild} ansehen, dann stellen Sie fest, dass dort durch *z* dividiert wird. Eine Division bei einer Transformation eines Punkts durch eine Matrix kann man nur mit Hilfe der vierten Koordinate, der *w*-Koordinate, erreichen. Es werden dabei aber alle Koordinaten durch denselben Wert dividiert, und das ist in unserem Fall *z*. Es steht also jetzt schon fest, dass die spätere Projektionsmatrix auf jeden Fall die *w*-Koordinate des resultierenden 4D-Vektors auf *z* setzen muss. Die Brüche, die als Faktoren in den Zählern

stehen, sind dabei kein Problem, denn es handelt sich um Konstanten, die vorberechnet werden können.

Was passiert mit *z*?

Wir können jetzt die *x*- und *y*-Bildkoordinaten eines 3D-Punkts mit Hilfe der Projektion bestimmen. Aber was soll mit der Tiefe, der *z*-Koordinate des Punkts geschehen? Man könnte annehmen, dass man sie nicht mehr braucht, da man ja nur auf einer zweidimensionalen Fläche zeichnet. Tatsächlich wird die Tiefe aber noch für die Lösung des so genannten *Sichtbarkeitsproblems* gebraucht, indem die Tiefe jedes Pixels in einem *Z-Buffer* gespeichert wird. So kann unabhängig von der Zeichenreihenfolge sichergestellt werden, dass ein nahes Objekt ein fernes Objekt überdeckt.

Damit der Z-Buffer richtig funktionieren kann, sollten sich die *z*-Werte zwischen 0 und 1 befinden, wobei 0 für die kleinstmögliche Tiefe steht und 1 für die größtmögliche Tiefe. Eine kleinstmögliche Tiefe haben wir schon, nämlich die Entfernung *D* vom Betrachter zur Bildebene (Fenster). Der Betrachter schaut nur nach draußen, ihn interessieren nur die Objekte hinter dem Fenster.

Aber wie sieht es mit einer größtmöglichen Tiefe aus? Diese haben wir noch nicht definiert und holen das jetzt nach. Wir nennen sie Z_{fern} , und um einheitlich zu bleiben, nennen wir *D* von nun an Z_{nah} . Alles, was sich jenseits von Z_{fern} befindet, wird später nicht mehr sichtbar sein. Normalerweise verwendet man in Computerspielen atmosphärische Effekte wie Nebel oder Dunst, um zu verschleiern, dass die Szene irgendwo in der Ferne aufhört. Auch in der Realität kann man schließlich nicht beliebig weit sehen.

Wir müssen jetzt also noch dafür sorgen, dass die *z*-Koordinaten der zu projizierenden Punkte auf den Bereich zwischen 0 und 1 umgerechnet werden (sofern sie zwischen Z_{nah} und Z_{fern} liegen). Als erstes ziehen wir Z_{nah} von *z* ab, denn bei Z_{nah} soll die resultierende *z*-Koordinate null sein. Dann muss noch sichergestellt werden, dass bei Z_{fern} der Wert 1 herauskommt. Wenn wir es nur bei der Subtraktion belassen, erhalten wir bei Z_{fern} nach der „erzwungenen“ Division den Wert $\frac{Z_{\text{fern}}-Z_{\text{nah}}}{Z_{\text{fern}}}$. Wir wollen aber, dass 1 herauskommt, also müssen wir noch durch diesen Wert dividieren. Schließlich erhalten wir:[2]

$$z_{\text{Bild}} = \frac{z - Z_{\text{nah}}}{z} \cdot \frac{Z_{\text{fern}}}{Z_{\text{fern}} - Z_{\text{nah}}} = \frac{z \cdot \frac{Z_{\text{fern}}}{Z_{\text{fern}} - Z_{\text{nah}}} - \frac{Z_{\text{fern}} \cdot Z_{\text{nah}}}{Z_{\text{fern}} - Z_{\text{nah}}}}{z}$$

Auf Grund der bereits erwähnten unvermeidlichen Division durch *z* ist die Verteilung von z_{Bild} nicht linear. Das heißt: Wenn *z* verdoppelt wird, gilt das nicht auch für z_{Bild} . Das kann bei unglücklich gewählten Werten für Z_{nah} und Z_{fern} zu Genauigkeitsproblemen im Z-Buffer führen. Generell sollte Z_{nah} so groß wie möglich und Z_{fern} so klein wie möglich ge-

[2] Die folgende Gleichung lässt sich natürlich vereinfachen, aber in dieser Form lässt sie sich später sehr leicht für die Matrixschreibweise verwenden.

wählt werden. Lassen Sie sich davon aber erst einmal nicht verwirren, wir werden dieses Thema wieder aufgreifen, wenn es so weit ist.

Die perspektivische Projektionsmatrix

Nun haben wir alle Formeln beisammen, um eine Projektionsmatrix für die perspektivische Projektion aufzustellen. Hier sind sie noch einmal:

$$x_{\text{Bild}} = \frac{x \cdot \frac{2 \cdot D}{W}}{z}$$

$$y_{\text{Bild}} = \frac{y \cdot \frac{2 \cdot D}{H}}{z}$$

$$z_{\text{Bild}} = \frac{z \cdot \frac{Z_{\text{fern}}}{Z_{\text{fern}} - Z_{\text{nah}}} - \frac{Z_{\text{fern}} \cdot Z_{\text{nah}}}{Z_{\text{fern}} - Z_{\text{nah}}}}{z}$$

In allen drei Gleichungen kommen nur lineare Terme vor, und es wird überall durch z dividiert. Das sieht also sehr viel versprechend aus. Das Ganze lässt sich nun relativ einfach in Matrixschreibweise darstellen:

$$M_{\text{Persp. Projektion}} = \begin{pmatrix} \frac{2 \cdot D}{W} & 0 & 0 & 0 \\ 0 & \frac{2 \cdot D}{H} & 0 & 0 \\ 0 & 0 & \frac{Z_{\text{fern}}}{Z_{\text{fern}} - Z_{\text{nah}}} & 1 \\ 0 & 0 & -\frac{Z_{\text{fern}} \cdot Z_{\text{nah}}}{Z_{\text{fern}} - Z_{\text{nah}}} & 0 \end{pmatrix}$$

$$(x,\ y,\ z,\ 1) \bullet M_{\text{Persp. Projektion}} = (x',\ y',\ z',\ w') = (x',\ y',\ z',\ z)$$

Projizierte Bildkoordinaten:

$$(x_{\text{Bild}},\ y_{\text{Bild}},\ z_{\text{Bild}}) = \left(\frac{x'}{w'},\ \frac{y'}{w'},\ \frac{z'}{w'}\right) = \left(\frac{x'}{z},\ \frac{y'}{z},\ \frac{z'}{z}\right)$$

Clipping

Nach der Projektion sind nur die Punkte sichtbar, für die gilt:

$$-1 \le x_{\text{Bild}} \le 1 \wedge$$
$$-1 \le y_{\text{Bild}} \le 1 \wedge$$
$$0 \le z_{\text{Bild}} \le 1$$

Wenn x_{Bild} beispielsweise kleiner als –1 ist, dann liegt der Punkt zu weit links, als dass er gesehen werden könnte, oder wenn y_{Bild} größer als 1 ist, dann liegt er zu weit unten. Ist z_{Bild} kleiner als null, dann liegt der Punkt zu nah am Betrachter („vor dem Fenster"), also näher als Z_{nah} . Ist z_{Bild} hingegen größer als 1, dann ist der Punkt weiter als Z_{fern} vom Betrachter weg und damit ebenfalls nicht mehr sichtbar.

Man kann diese Bedingungen auch schon vor der Division durch $w' = z$ überprüfen. Dann lauten sie so:

$$-w' \le x' \le w' \wedge$$
$$-w' \le y' \le w' \wedge$$
$$0 \le z' \le w'$$

Tatsächlich wird genau das getan. Wenn sich nun herausstellt, dass ein zu renderndes Dreieck teilweise unsichtbar ist, dann wird es dort „abgeschnitten". Dabei können zusätzliche, kleinere Dreiecke entstehen. Das nennt man *Clipping*.[3]

Der Bildwinkel

Nun ist es nicht sehr praktisch, beim Erzeugen einer Projektionsmatrix immer mit der Breite *W* und der Höhe *H* der Bildebene herumzuhantieren. Üblicherweise verwendet man bei der Definition einer Projektionsmatrix stattdessen den horizontalen und vertikalen *Bildwinkel*. Wir nennen diese Winkel α_x (horizontal) und α_y (vertikal).

Eine Panoramaaufnahme hat zum Beispiel einen horizontalen Bildwinkel von 360°. Durch Veränderung der Bildwinkel lässt sich der Zoom bestimmen. Teleobjektive haben zum Beispiel sehr kleine Bildwinkel, während Weitwinkelobjektive (wie der Name schon sagt) große Bildwinkel aufweisen. Mit unseren Augen erreichen wir übrigens einen horizontalen Bildwinkel von fast 180°, wobei wir aber nur in einem kleinen Bereich scharf sehen können. Statt von den Bildwinkeln spricht man auch häufig vom *Sichtfeld* oder, auf Englisch, vom *FOV* (*Field of View*).

Die meisten Computerspiele verwenden einen horizontalen Bildwinkel von 90° oder weniger. Wenn sich der Spieler aber beispielsweise in einem Rennspiel mit sehr hoher Geschwindigkeit bewegt, wird oft das Sichtfeld entsprechend vergrößert, um den Eindruck zu verstärken. Blickt der Spieler durch ein (Ziel-)Fernrohr, dann wird das Sichtfeld verkleinert, um eine Vergrößerung zu erreichen.

---

[3] Oft reden versierte Computerspieler von „Clipping-Fehlern", wenn Spielobjekte in feste Objekte wie eine Wand hineinragen, zum Beispiel die Kanone auf dem drehbaren Turm eines Panzers. Technisch gesehen ist das falsch, da dieser „Fehler" durch eine unzureichende Kollisionserkennung entsteht und nicht durch fehlerhaftes Clipping.

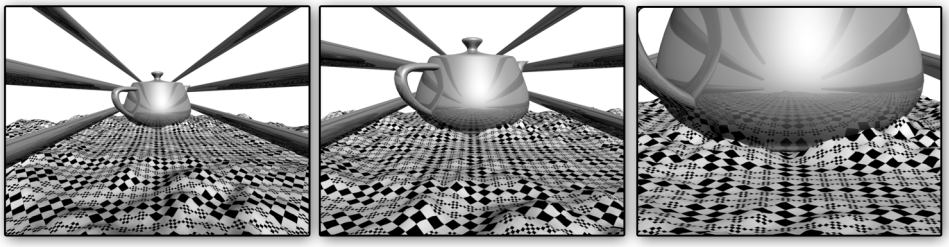


Abbildung 2.30 Bilder mit vertikalen Bildwinkeln von 70°, 45° und 20°

Welcher Zusammenhang besteht nun zwischen den Sichtwinkeln und unserer bisherigen Darstellung mit Hilfe der Breite, Höhe und Distanz der Bildebene? Diesen Zusammenhang zeigt die folgende Abbildung:

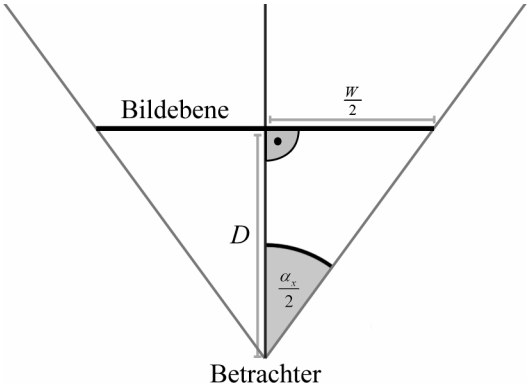


Abbildung 2.31 Dieses rechtwinkligen Dreieck birgt die Beziehung zwischen *D*, *W* und α_x.

Mit etwas Trigonometrie ergibt sich:

$$\cot\frac{\alpha_x}{2} = \frac{2 \cdot D}{W} \quad \text{und} \quad \cot\frac{\alpha_y}{2} = \frac{2 \cdot D}{H}$$

Auf den rechten Seiten der beiden Gleichungen stehen genau die Ausdrücke, die wir vorhin in die Matrix eingetragen haben. Wir können sie dort also durch die linken Seiten der Gleichungen ersetzen.

Normalerweise möchte man aber nur einen der beiden Bildwinkel angeben (üblicherweise den Vertikalen) und den anderen Winkel mit Hilfe des Bildseitenverhältnisses und des gegebenen Winkels berechnen. Ansonsten würden nicht aufeinander abgestimmte Bildwinkel das Bild verzerren. Ein Quadrat würde dann nicht als Quadrat, sondern als Rechteck dargestellt werden.

Eine Projektionsmatrix, die die Angabe des vertikalen Bildwinkels α_y und des Bildseitenverhältnisses *R* (Breite geteilt durch Höhe) gestattet, sieht so aus:

$$M_{\text{Persp. Projektion}} = \begin{pmatrix} R \cdot \cot\frac{\alpha_y}{2} & 0 & 0 & 0 \\ 0 & \cot\frac{\alpha_y}{2} & 0 & 0 \\ 0 & 0 & \frac{Z_{\text{fern}}}{Z_{\text{fern}} - Z_{\text{nah}}} & 1 \\ 0 & 0 & -\frac{Z_{\text{fern}} \cdot Z_{\text{nah}}}{Z_{\text{fern}} - Z_{\text{nah}}} & 0 \end{pmatrix}$$

2.5.10.2 Der View-Frustum

Stellen wir uns noch einmal die Situation mit dem Betrachter und dem Fenster vor, durch das er nach draußen schaut. Da wir seine Sichtweite in alle sechs Richtungen begrenzt haben, lässt sich der für ihn sichtbare Bereich durch einen Pyramidenstumpf beschreiben, also eine Pyramide, die dort abgeflacht ist, wo normalerweise die Spitze wäre. Diesen Pyramidenstumpf nennt man auch den *View-Frustum*. Für diesen englischen Begriff ist mir leider keine gute deutsche Übersetzung bekannt, darum werde ich ihn so übernehmen.

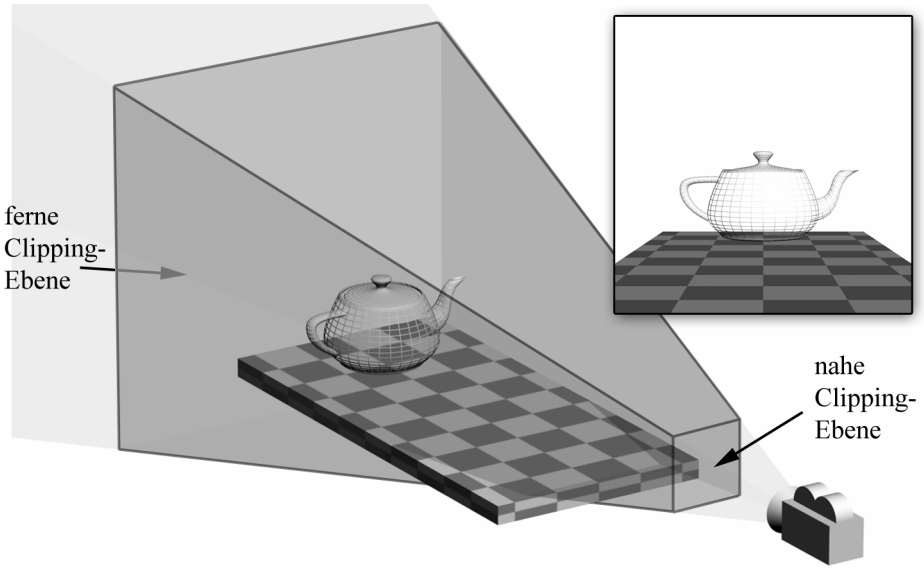


Abbildung 2.32 Der View-Frustum ist ein Pyramidenstumpf. Er bestimmt den Sichtbereich.

Der View-Frustum wird durch 6 Ebenen begrenzt. Weil an diesen Ebenen der Szeneninhalt, wie vorhin schon erwähnt, „abgeschnitten“ wird, da er sowieso nicht sichtbar wäre und somit auch nicht gerendert werden muss, nennt man diese Ebenen auch *Clipping-Ebenen* (*clipping planes*). Im Einzelnen spricht man von der *nahen Clipping-Ebene* (*near clipping plane*), der *fernen Clipping-Ebene* (*far clipping plane*) und der *linken*, *rechten*,

oberen und *unteren Clipping-Ebene*. Die nahe Clipping-Ebene befindet sich Z_{nah} Einheiten vom Betrachter entfernt, und die ferne Clipping-Ebene liegt bei Z_{fern} .

Durch die perspektivische Transformation und die anschließende Projektion wird der View-Frustum zu einem Würfel verzerrt. Genau genommen ist es nur ein halber Würfel, da z_{Bild} nur Werte zwischen 0 und 1 annehmen darf, jedoch x_{Bild} und y_{Bild} zwischen –1 und 1 liegen müssen. Da die vordere Seite des View-Frustums kleiner ist als die hintere Seite, werden weit entfernte Objekte so ganz automatisch kleiner, da sie stärker zusammengedrückt werden.

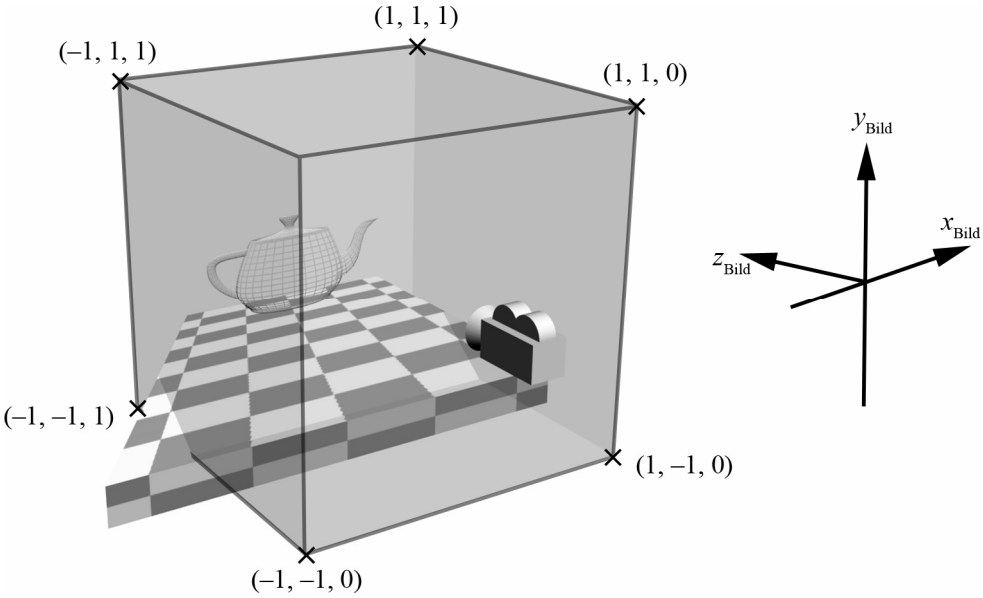


Abbildung 2.33 Die perspektivische Transformation mit der anschließenden Projektion verzerrt den View-Frustum zu einem halben Würfel.

Die Koordinaten innerhalb dieses Würfels wollen wir *3D-Bildkoordinaten* nennen. 3D deshalb, weil sie noch eine Tiefe besitzen, die wir später noch brauchen werden.

Um letztlich die 2D-Pixelkoordinaten eines Punkts zu ermitteln, müssen x_{Bild} und y_{Bild} nur noch an den Zeichenbereich (*Viewport*) angepasst werden. Wenn wir in ein Fenster einer Breite von 800 und einer Höhe von 600 Pixel zeichnen, dann entspricht beispielsweise (0, 0) den Pixelkoordinaten (400, 300), also der Bildmitte, und (–1, 0.5) entspricht den Pixelkoordinaten (0, 150). Somit sind alle Transformationen bis zum Schluss unabhängig von der tatsächlich verwendeten Auflösung.

2.5.10.3 Eine einfachere Projektion: die Parallelprojektion

Manchmal möchte man gar nicht, dass Objekte kleiner werden, wenn sie sich vom Betrachter entfernen. Der View-Frustum ist in diesem Fall kein richtiger Pyramidenstumpf, sondern einfach nur ein in die Tiefe lang gezogener Quader. Wenn das der Fall ist, spricht

man von der *Parallelprojektion* oder der *orthographischen Projektion*. Solche Projektionen kann man verwenden, um isometrische Ansichten zu erzeugen, wie sie für Rollenspiele oder Strategiespielen typisch sind. Hier belässt man die *w*-Koordinate auf 1, führt also später keine Division durch. Bei der Parallelprojektion sind parallele Geraden auch auf dem Bildschirm parallel, was bei der perspektivischen Projektion nicht der Fall ist.

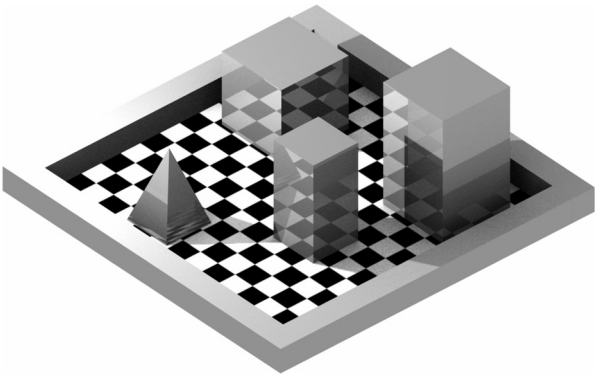


Abbildung 2.34 Mit Parallelprojektion gerendertes Bild

Eine Projektionsmatrix für die Parallelprojektion sieht so aus:

$$M_{\text{Parallelprojektion}} = \begin{pmatrix} \frac{2}{W} & 0 & 0 & 0 \\ 0 & \frac{2}{H} & 0 & 0 \\ 0 & 0 & -\frac{1}{Z_{\text{nah}} - Z_{\text{fern}}} & 0 \\ 0 & 0 & \frac{Z_{\text{nah}}}{Z_{\text{nah}} - Z_{\text{fern}}} & 1 \end{pmatrix}$$

Damit wäre das Thema der Projektion vorerst abgeschlossen. Wenn Sie nicht alles davon verstanden haben, ist das kein Grund, sich Sorgen zu machen. Ich muss gestehen, dass ich auch sehr lange gebraucht habe, um all die Details der Projektion richtig nachvollziehen zu können.

2.6 Einfache geometrische Objekte

In diesem Abschnitt werden wir uns mit einigen einfachen geometrischen Objekten wie Strahlen, Ebenen und Kugeln beschäftigen. Diese Objekte gehören sozusagen zum Handwerkszeug der 3D-Grafik.

2.6.1 Geraden, Strahlen und Strecken

Geraden

Die Gerade ist nach dem Punkt wohl das zweit grundlegendste geometrische Objekt. Eine Gerade beschreibt eine unendlich lange und unendlich dünne gerade Linie von Punkten.

Wir können eine Gerade durch den Ortsvektor eines Punkts *O*, der auf dieser Geraden liegt, und die Richtung der Geraden definieren. Den Ortsvektor wollen wir $\vec{O}$ nennen und den Richtungsvektor $\vec{u}$. Dann können wir die Gerade als Funktion schreiben:

$$\vec{L}(r) = \vec{O} + r \cdot \vec{u} \qquad \text{mit } r \in \mathbb{R}$$

Der Buchstabe *L* steht hierbei für *line*. Indem wir den Parameter *r* variieren, können wir jeden Punkt auf der Geraden erreichen. Dazu fangen wir beim Punkt *O* an und gehen dann das *r*-fache des Richtungsvektors weiter. Welchen Punkt *O* auf der Geraden wir wählen, spielt prinzipiell keine Rolle. Außerdem ist es auch egal, wie lang $\vec{u}$ ist. Durch die Funktion wird trotzdem dieselbe Gerade beschrieben. Da eine Gerade ein eindimensionales Objekt ist, brauchen wir auch nur einen Parameter, um jeden Punkt auf ihr beschreiben zu können. Diese Art der Beschreibung nennt man auch *explizite Darstellung* oder *Parameterdarstellung*.

Betrachten wir als Beispiel einmal eine Gerade mit $\vec{O} = (4,\ -1,\ 0)$ und $\vec{u} = (5,\ -5,\ 1)$. Die Gerade zeigt nach rechts unten und leicht in die Tiefe. Wenn wir nun verschiedene Werte für *r* einsetzen, erhalten wir Punkte, die auf der Geraden liegen:

$$\begin{aligned}
\vec{L}(0) &= \vec{O} + 0 \cdot \vec{u} = \vec{O} &&= (4,\ -1,\ 0) \\
\vec{L}(1) &= \vec{O} + 1 \cdot \vec{u} = \vec{O} + \vec{u} &&= (9,\ -6,\ 1) \\
\vec{L}(10) &= \vec{O} + 10 \cdot \vec{u} &&= (54,\ -51,\ 10) \\
\vec{L}(-2.5) &= \vec{O} - 2.5 \cdot \vec{u} &&= (-8.5,\ 1.5,\ -2.5)
\end{aligned}$$

Eine Gerade kann man natürlich auch durch zwei Punkte *A* und *B* konstruieren. Dazu wählen wir zum Beispiel $\vec{O} = \vec{A}$ und $\vec{u} = \vec{B} - \vec{A}$.

Wie können wir überprüfen, ob ein Punkt *X* auf einer Geraden liegt? Dazu müssten wir ein *r* finden, so dass $\vec{L}(r) = \vec{X}$ gilt. Wenn es ein solches *r* gibt, dann liegt der Punkt auf der Geraden, ansonsten nicht. Hier hilft uns die Parameterdarstellung nicht weiter. Dazu gibt es die *implizite Darstellung*. Sie gibt eine Bedingung an, die für genau die Punkte auf der Geraden gilt. Eine solche Bedingung könnte lauten: Der Punkt *X* liegt genau dann auf der Geraden mit dem Punkt *O*, wenn die Gerade durch *O* und *X* parallel zur Geraden ist. Im Dreidimensionalen können wir mit Hilfe des Kreuzprodukts zwei Vektoren auf Parallelität testen. Das Kreuzprodukt ergibt dann nämlich den Nullvektor. Ein Punkt *X* liegt also genau dann auf der Geraden mit dem Punkt *O* und dem Richtungsvektor $\vec{u}$, wenn gilt:

$$\left(\vec{X} - \vec{O}\right) \times \vec{u} = \vec{0}$$

Als Beispiel nehmen wir wieder die Gerade von vorhin und setzen für X den Punkt $\vec{L}(2) = (14,\ -11,\ 2)$ ein, der ja offensichtlich auf der Geraden liegt.

$$\begin{aligned}\left(\vec{X} - \vec{O}\right) \times \vec{u} &= \left((14,\ -11,\ 2) - (4,\ -1,\ 0)\right) \times (5,\ -5,\ 1) \\ &= (10,\ -10,\ 2) \times (4,\ -1,\ 0) = (0,\ 0,\ 0) = \vec{0}\end{aligned}$$

Setzen wir nun noch einen Punkt ein, der nicht auf der Geraden liegt. Dann dürfte als Ergebnis auch nicht der Nullvektor herauskommen. Wir wählen $\vec{Y} = (5,\ -3,\ 4)$:

$$\begin{aligned}\left(\vec{Y} - \vec{O}\right) \times \vec{u} &= \left((5,\ -3,\ 4) - (4,\ -1,\ 0)\right) \times (5,\ -5,\ 1) \\ &= (1,\ -2,\ 4) \times (5,\ -5,\ 1) = (18,\ 19,\ 5) \neq \vec{0}\end{aligned}$$

Wie Sie sehen, funktioniert dieser Test. Die implizite Darstellung erlaubt es uns zu überprüfen, ob ein Punkt auf der Geraden liegt, wohingegen die Parameterdarstellung uns alle Punkte liefert, die auf der Geraden liegen. Manchmal ist die eine Form praktischer, manchmal die Andere.

Strahlen

Ein Strahl (englisch: *ray*), auch als *Halbgerade* bezeichnet, unterscheidet sich von der Geraden dadurch, dass er auf einer Seite begrenzt ist (er ist aber trotzdem unendlich lang). Wir beschreiben auch den Strahl durch einen Punkt O mit dem Ortsvektor $\vec{O}$ und einen Richtungsvektor $\vec{u}$. Hier ist O allerdings nicht irgendein Punkt auf dem Strahl, sondern der Anfangspunkt des Strahls. Die Parameterdarstellung ist jener der Geraden sehr ähnlich, mit der Einschränkung, dass r nur positiv (oder null) sein darf, da negative Werte „nach hinten aus dem Strahl heraus“ führen würden:

$$\vec{R}(r) = \vec{O} + r \cdot \vec{u} \qquad \text{mit } r \in \mathbb{R}^+$$

Die implizite Darstellung muss ebenfalls etwas abgeändert werden. Sie muss nun auch die Bedingung beinhalten, dass der Punkt X auf der richtigen Seite von O liegen muss, und zwar auf der Seite, in deren Richtung $\vec{u}$ zeigt. Das können wir mit dem Skalarprodukt testen. Wenn zwei Richtungsvektoren $\vec{a}$ und $\vec{b}$ in entgegengesetzte Richtungen zeigen, dann ist ihr Skalarprodukt negativ. Zeigen sie aber in dieselbe Richtung, dann ist es positiv. Damit können wir die implizite Darstellung eines Strahls angeben. Ein Punkt X liegt genau dann auf dem Strahl mit dem Anfangspunkt O und dem Richtungsvektor $\vec{u}$, wenn gilt:

$$\left(\vec{X} - \vec{O}\right) \times \vec{u} = \vec{0} \ \wedge \ \left(\vec{X} - \vec{O}\right) \bullet \vec{u} \geq 0$$

Strecken

Eine Strecke (englisch: *line segment*) ist eine Gerade, die auf beiden Seiten begrenzt ist. Sie hat also eine endliche Länge. Am einfachsten lässt sich eine Strecke durch ihren Anfangspunkt A und ihren Endpunkt B beschreiben. Die Parameterdarstellung lautet dann:

$$\vec{S}(r) = \vec{A} + r \cdot \left(\vec{B} - \vec{A}\right) \quad \text{mit } 0 \le r \le 1$$

Für $r = 0$ bekommen wir den Anfangspunkt A, für $r = 1$ den Endpunkt B und für $r = 0.5$ den Mittelpunkt der Strecke.

Wie können wir für einen Punkt X, von dem wir wissen, dass er auf der Geraden durch A und B liegt, den Wert r bestimmen, so dass $\vec{S}(r) = \vec{X}$ gilt? Dazu müssen wir die Gleichung ein wenig umformen:

$$\begin{aligned}
&\vec{A} + r \cdot \left(\vec{B} - \vec{A}\right) = \vec{X} && \left| -\vec{A} \right. \\
&r \cdot \left(\vec{B} - \vec{A}\right) = \vec{X} - \vec{A} && \left| \bullet \left(\vec{B} - \vec{A}\right) \right. \\
&r \cdot \left(\vec{B} - \vec{A}\right)^2 = \left(\vec{X} - \vec{A}\right) \bullet \left(\vec{B} - \vec{A}\right) && \left| \div \left(\vec{B} - \vec{A}\right)^2 \right. \\
&r = \frac{\left(\vec{X} - \vec{A}\right) \bullet \left(\vec{B} - \vec{A}\right)}{\left(\vec{B} - \vec{A}\right)^2}
\end{aligned}$$

Nun können wir auch die implizite Darstellung einer Strecke angeben. Ein Punkt X liegt demnach genau dann auf der Strecke von A nach B, wenn gilt:

$$\left(\vec{X} - \vec{A}\right) \times \left(\vec{B} - \vec{A}\right) = \vec{0} \ \wedge \ 0 \le \frac{\left(\vec{X} - \vec{A}\right) \bullet \left(\vec{B} - \vec{A}\right)}{\left(\vec{B} - \vec{A}\right)^2} \le 1$$

Ein paar Beispiele:

$\vec{A} = (1,\ 2,\ 3)$

$\vec{B} = (10,\ 9,\ 8)$

$\vec{B} - \vec{A} = (9,\ 7,\ 5)$

$\vec{X}$	$\vec{X} - \vec{A}$	$\left(\vec{X} - \vec{A}\right) \times \left(\vec{B} - \vec{A}\right)$	$r = \frac{\left(\vec{X} - \vec{A}\right) \bullet \left(\vec{B} - \vec{A}\right)}{\left(\vec{B} - \vec{A}\right)^2}$	Auf der Strecke?
(5, 5, 5)	(4, 3, 2)	$(1, -2, 1) \neq \vec{0}$	/	Nein
(10, 9, 8)	(9, 7, 5)	$(0, 0, 0) = \vec{0}$	1	Ja
(−17, −12, −7)	(−18, −14, −10)	$(0, 0, 0) = \vec{0}$	−2	Nein
(3.25, 3.75, 4.25)	(2.25, 1.75, 1.25)	$(0, 0, 0) = \vec{0}$	0.25	Ja

Beim ersten Punkt muss *r* gar nicht berechnet werden, da der Test mit dem Kreuzprodukt schon ergibt, dass der Punkt nicht einmal auf der Geraden durch *A* und *B* liegt. Dann kann er natürlich auch nicht auf der Strecke liegen.

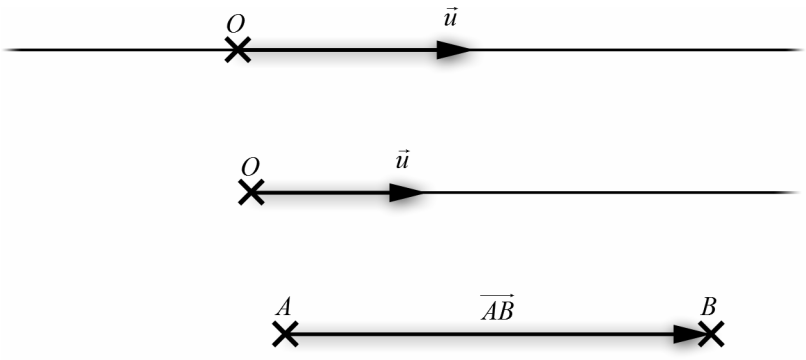


Abbildung 2.35 Gerade, Strahl und Strecke

2.6.2 Ebenen

Während Geraden, Strahlen und Strecken eindimensionale Objekte sind, hat eine Ebene (englisch: *plane*) zwei Dimensionen. Sie erstreckt sich unendlich weit in zwei Richtungen.

Darstellungen

Die Parameterdarstellung der Ebene sieht fast so aus wie die der Geraden, nur dass wir hier neben einem Punkt zwei Parameter und zwei Richtungen haben:

$$\vec{P}(r,\, s) = \vec{O} + r \cdot \vec{u} + s \cdot \vec{v} \qquad \text{mit } r, s \in \mathbb{R} \text{ und } \vec{u} \not\parallel \vec{v}$$

Den (beliebigen) Punkt auf der Ebene nennen wir wieder *O*. Die beiden Richtungsvektoren $\vec{u}$ und $\vec{v}$ dürfen nicht parallel sein, denn sonst beschreibt die Gleichung keine Ebene, sondern eine Gerade.

Der Richtungsvektor, der senkrecht auf der Oberfläche der Ebene steht, heißt *Normalenvektor*. Wir nennen ihn $\vec{n}$. Man berechnet ihn mit dem Kreuzprodukt von $\vec{u}$ und $\vec{v}$. Bei einer Ebene, die dem „Boden“ entspricht (unendliche Ausdehnung auf der *x*- und der *z*-Achse) zeigt dieser Vektor beispielsweise nach oben.

Wie entscheidet man nun, ob ein Punkt *X* auf einer Ebene liegt? Dazu berechnet man die Richtung von einem beliebigen Punkt der Ebene (zum Beispiel *O*) nach *X*. Dieser Richtungsvektor $\overrightarrow{OX}$ muss nun „in der Ebene“ liegen. Das heißt: Er muss sich als Linearkombination der beiden Richtungsvektoren $\vec{u}$ und $\vec{v}$ darstellen lassen (es müssen zwei Zahlen *r* und *s* existieren, so dass $\overrightarrow{OX} = r \cdot \vec{u} + s \cdot \vec{v}$). Das ist genau dann der Fall, wenn der Vektor senkrecht zum Normalenvektor der Ebene ist, also wenn $\overrightarrow{OX} \bullet \vec{n} = 0$ gilt.

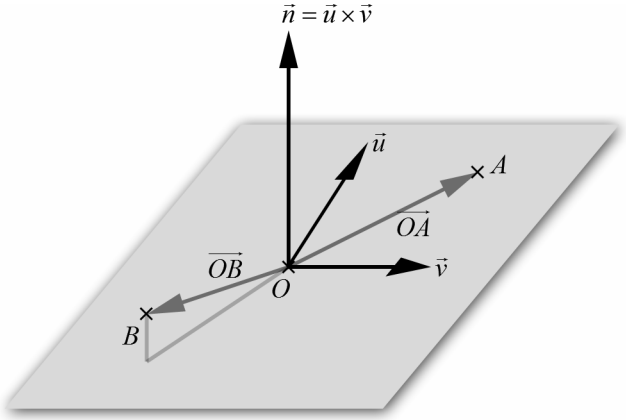


Abbildung 2.36 Eine Ebene mit dem Punkt *O*, den Richtungsvektoren $\vec{u}$ und $\vec{v}$ sowie dem Normalenvektor $\vec{n}$. Der Punkt *A* liegt auf der Ebene, denn $\overrightarrow{OA}$ ist senkrecht zu $\vec{n}$. *B* liegt nicht auf der Ebene, denn $\overrightarrow{OB}$ ist nicht senkrecht zu $\vec{n}$. *B* liegt über der Ebene, wie in der Abbildung angedeutet ist. In Wirklichkeit ist die Ebene natürlich unendlich groß.

Die implizite Darstellung einer Ebene sieht nun so aus:

$$\left(\vec{X} - \vec{O}\right) \bullet \vec{n} = 0$$

Diese Form nennt man auch *Normalenform*. Sie beschreibt alle Punkte *X*, die auf der Ebene mit dem Punkt *O* und dem Normalenvektor $\vec{n}$ liegen. Wenn man $\vec{n}$ in die Klammer einmultipliziert, erhält man:

$$\vec{X} \bullet \vec{n} - \vec{O} \bullet \vec{n} = 0$$

Nun legen wir fest, dass wir als Normalenvektor einen Einheitsvektor verwenden, also $\vec{n}_0$ (gleiche Richtung wie $\vec{n}$, aber mit einer Länge von 1), und definieren die Variable *d* als $\vec{O} \bullet \vec{n}_0$. Was wir dann erhalten, nennt man die *Hessesche Normalform*:

$$\vec{X} \bullet \vec{n}_0 - d = 0$$

Der Parameter *d*, also eigentlich $\vec{O} \bullet \vec{n}_0$, ist der Abstand der Ebene vom Koordinatenursprung. Mit *d* können wir die Ebene also entlang ihres Normalenvektors $\vec{n}_0$ verschieben (in eine andere Richtung würde eine Verschiebung auch wenig Sinn machen, da die Ebene dort schließlich unendlich weit ausgebreitet ist). Wir können eine Ebene jetzt allein durch $\vec{n}_0$ und *d* eindeutig darstellen.

Die Hessesche Normalform hat eine praktische Besonderheit: Wenn man die Koordinaten eines Punkts in sie einsetzt, ist das Ergebnis der Abstand des Punkts von der Ebene. Das Vorzeichen gibt dabei an, auf welcher Seite der Ebene sich der Punkt befindet.

$x = \vec{X} \bullet \vec{n}_0 - d$

$x = 0$: X befindet sich auf der Ebene.

$x > 0$: X befindet sich x Einheiten vor der Ebene.

$x < 0$: X befindet sich $-x$ Einheiten hinter der Ebene.

Die „vordere Seite“ der Ebene ist die, in deren Richtung der Normalenvektor zeigt.

Konstruktion

Eine Ebene kann durch drei Punkte A, B und C, die nicht auf einer Geraden liegen, definiert werden. Dazu setzt man zum Beispiel $\vec{O} = \vec{A}$, $\vec{u} = \overrightarrow{AB}$ und $\vec{v} = \overrightarrow{AC}$. Daraus ergibt sich $\vec{n} = \overrightarrow{AB} \times \overrightarrow{AC}$. $\vec{n}$ wird im nächsten Schritt zu $\vec{n}_0$ normiert. Dann fehlt noch d. Wir setzen $d = \vec{A} \bullet \vec{n}_0$ (oder auch $\vec{B} \bullet \vec{n}_0$ oder $\vec{C} \bullet \vec{n}_0$, das macht keinen Unterschied, denn es muss nur irgendein Punkt sein, der auf der Ebene liegt).

Es soll zum Beispiel eine Ebene konstruiert werden, die die folgenden Punkte enthält:

$$\vec{A} = (4,\ 0,\ 0)$$
$$\vec{B} = (2,\ 1,\ 2)$$
$$\vec{C} = (0,\ 2,\ 6)$$

Wir berechnen nun den Normalenvektor und den Abstand der Ebene:

$$\vec{O} = \vec{A} = (4,\ 0,\ 0)$$
$$\vec{u} = \overrightarrow{AB} = \vec{B} - \vec{A} = (-2,\ 1,\ 2)$$
$$\vec{v} = \overrightarrow{AC} = \vec{C} - \vec{A} = (-4,\ 2,\ 6)$$
$$\vec{n} = \vec{u} \times \vec{v} = (2,\ 4,\ 0)$$
$$\vec{n}_0 = \frac{\vec{n}}{|\vec{n}|} = \frac{(2,\ 4,\ 0)}{\sqrt{20}} \approx (0.447,\ 0.984,\ 0)$$
$$d = \vec{A} \bullet \vec{n}_0 = \vec{B} \bullet \vec{n}_0 = \vec{C} \bullet \vec{n}_0 \approx 1.789$$

Die Hessesche Normalform der gesuchten Ebene lautet also:

$$\vec{X} \bullet (0.447,\ 0.984,\ 0) - 1.789 = 0$$

Wozu braucht man Ebenen in der 3D-Grafik? Vielleicht erinnern Sie sich noch an die Projektionstransformation und den View-Frustum. Der View-Frustum ist der Bereich in Form einer abgestumpften Pyramide, der alle sichtbaren Punkte in der Szene enthält. Er wird durch sechs Ebenen begrenzt.[4] Will man herausfinden, ob ein Punkt sichtbar ist, dann

[4] Auch ein Quader kann durch sechs Ebenen beschrieben werden. Solche konvexen Körper, die sich durch (beliebig viele, aber nicht unendlich viele) Ebenen beschreiben lassen, nennt man *Polyeder*. Für solche Objekte lassen sich gewisse Berechnungen besonders effizient durchführen, wie zum Beispiel Schnitttests mit Strahlen.

kann man dazu den Punkt in die Gleichungen dieser Ebenen einsetzen. Sobald man sieht, dass er sich auf der „falschen“ Seite einer der sechs Ebenen befindet, weiß man, dass er damit auch außerhalb des View-Frustums und damit unsichtbar ist. Aber Ebenen spielen auch noch in anderen Bereichen eine Rolle.

2.6.3 Kugeln

Das vorerst letzte geometrische Objekt, das wir betrachten wollen, ist die Kugel (englisch: *sphere*). Eine Kugel hat einen Mittelpunkt *C* (*center*) und einen Radius *r*. Die Punkte, die auf der Kugeloberfläche liegen, werden dadurch charakterisiert, dass ihr Abstand zum Mittelpunkt genau dem Radius entspricht. Das besagt auch die implizite Darstellung einer Kugel:

$$\left|\vec{X}-\vec{C}\right| = r \quad \text{oder} \quad \left(\vec{X}-\vec{C}\right)^2 = r^2$$

Die Parameterdarstellung einer Kugel funktioniert mit zwei Winkeln (Kugelkoordinaten). Sie ist für unsere Zwecke nicht wichtig, aber der Vollständigkeit halber möchte ich sie trotzdem angeben:

$$\vec{S}(\alpha,\ \beta) = \vec{C} + r \cdot \begin{pmatrix} \sin\alpha \cdot \cos\beta \\ \sin\alpha \cdot \sin\beta \\ \cos\beta \end{pmatrix} \quad \text{mit } 0 \le \alpha \le \pi \text{ und } 0 \le \beta \le 2\pi$$

In der 3D-Grafik werden Kugeln häufig verwendet, um bestimmte Algorithmen zu beschleunigen. Dazu wird beispielsweise eine Kugel um ein komplexes 3D-Modell gelegt (eine so genannte *Bounding-Sphere*). Möchte man nun herausfinden, ob das Modell sichtbar ist, testet man zunächst, ob seine Bounding-Sphere sichtbar ist. Das geht sehr schnell, weil die Kugel eine so gleichmäßige Form hat und sich leicht beschreiben lässt. Wenn sich dann ergibt, dass die Bounding-Sphere nicht sichtbar ist, dann ist das darin liegende Modell erst recht nicht sichtbar und braucht nicht gerendert zu werden.

Das gleiche kann man auch mit einem Quader (*Box*) tun und erhält dann eine *Bounding-Box*. Für manche Objekte eignet sich eine Kugel besser, für manche ein Quader. Das Ganze funktioniert nicht nur für Sichtbarkeitstests, sondern beispielsweise auch für Kollisionstests. Wenn die Bounding-Spheres oder Bounding-Boxes zweier Objekte nicht kollidieren (was sehr effizient festgestellt werden kann), dann können auch die Objekte selbst nicht kollidieren.

2.7 Farben

Farben spielen in der Computergrafik und der Spieleprogrammierung naturgemäß eine große Rolle, denn wer würde (in der heutigen Zeit) schon gerne ein Spiel spielen, das nur Schwarzweißgrafik zu bieten hat? Wir werden uns zunächst damit beschäftigen, wie die

Farbwahrnehmung funktioniert, und dann auf das RGB-Farbmodell eingehen, das sich in der Computergrafik etabliert hat.

2.7.1 Licht und Farbe

Farbe hat etwas mit dem Sehen zu tun, und sehen können wir nur mit Hilfe von Licht, das in unsere Augen fällt. Als (sichtbares) Licht bezeichnet man einen bestimmten Bereich aus dem elektromagnetischen Spektrum, der von unseren Augen wahrgenommen wird. Als sichtbares Licht bezeichnet man elektromagnetische Wellen mit einer Wellenlänge zwischen ungefähr 380 und 780 nm (1 nm = 1 Nanometer = 1 Milliardstel Meter $=10^{-9}\,\mathrm{m}$). 380 nm, die kürzeste sichtbare Wellenlänge, entspricht Violett, und 780 nm entspricht Rot. Dazwischen liegen Blau, Grün, Gelb und Orange. Den Bereich unmittelbar „unterhalb" von Rot bezeichnet man als Infrarot (IR-Strahlung), und den Bereich unmittelbar „oberhalb" von Violett als Ultraviolett (UV-Strahlung).

Elektromagnetische Strahlung breitet sich mit Lichtgeschwindigkeit (im Vakuum sind das $299\,792\,458\,\frac{m}{s}$) aus und kann entweder als Welle oder als Teilchen (Photonen) betrachtet werden. Die Energie der Strahlung hängt von der Frequenz ab, und damit von der Wellenlänge. Eine größere Wellenlänge bedeutet eine niedrigere Frequenz und eine geringere Energie und umgekehrt. Die Intensität des Lichts hängt nicht von der Frequenz ab, sondern von der Amplitude der Welle (im Wellenmodell) oder der Anzahl der Photonen pro Zeiteinheit (im Teilchenmodell).

Das sichtbare Licht macht nur einen sehr kleinen Teil des Spektrums aus. Am unteren Ende befindet sich die langwellige Radiostrahlung, mit der wir unsere Radio- und Fernsehprogramme übertragen, und die auch für die Astronomie sehr interessant ist (Radioteleskope). Danach folgen die Mikrowellen, das Infrarote, das sichtbare Licht und das bereits gefährliche UV-Licht. Am oberen Ende des Spektrums liegen Röntgen- und Gammastrahlung.

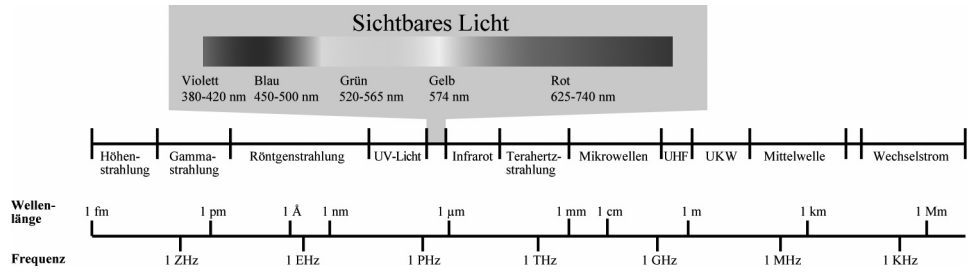


Abbildung 2.37 Das elektromagnetische Spektrum und das sichtbare Licht

Wenn man einen Regenbogen anschaut, dann sieht man darin das gesamte Spektrum des sichtbaren Lichts – von Rot bis Violett. Bei diesen Farben handelt es sich um monochromatisches Licht, das heißt, dass es nur aus einer einzigen Wellenlänge besteht (Spektralfarben). Bei dem Licht, das wir in unserer Umgebung wahrnehmen, handelt es sich aber

meistens um polychromatisches Licht, das aus vielen Wellenlängen zusammengesetzt ist. Viele Farben, die wir kennen, findet man nicht im Regenbogen wieder, da sie nur durch eine solche Farbmischung entstehen können.

Das Licht, das die Sonne ausstrahlt, enthält – bis auf einige Lücken (*Fraunhoferlinien*) – das gesamte sichtbare Spektrum und noch mehr. Solches Licht, in dem das ganze sichtbare Spektrum ungefähr gleich stark vertreten ist, nehmen wir als Weiß wahr. Da der blaue Anteil in der Erdatmosphäre gestreut wird, empfinden wir das Sonnenlicht jedoch eher als gelblich.

2.7.2 Farbe und ihre Wahrnehmung

Wenn Licht in unser Auge und auf die Netzhaut fällt, wird es in Nervenimpulse umgewandelt, die über den Sehnerv in unser Gehirn geleitet werden. Zu diesem Zweck befinden sich auf der Netzhaut lichtempfindliche Sinneszellen, die *Fotorezeptoren*. Davon gibt es zwei Arten: *Stäbchen* und *Zapfen*.

Die Stäbchen, von denen ein durchschnittlicher Mensch in jedem Auge ungefähr 120 Millionen besitzt, tragen nicht zur Farbwahrnehmung bei, sondern helfen uns beim Sehen in dunklen Umgebungen. Im äußeren Bereich der Netzhaut sind sie stärker vertreten als im Zentrum, weshalb man kleine Helligkeitsunterschiede in der Dunkelheit am besten aus dem Augenwinkel sehen kann. Das ist wohl durch die Evolution bedingt, denn wenn man von der Seite plötzlich angegriffen ist, ist es relativ egal, welche Farbe der Angreifer hat, solange man rechtzeitig erkennt, dass er da ist.

Von den Zapfen haben wir wiederum drei Typen: Blau-, Grün- und Rotrezeptoren. Je nach Typ sind die Zapfen für einen bestimmten Teil des Spektrums besonders empfindlich. Die Rotrezeptoren nehmen hauptsächlich den roten Anteil des Lichts wahr, die Grünrezeptoren den Grünen und die Blaurezeptoren den Blauen. Sie ermöglichen es uns, zwischen verschiedenen Farben zu unterscheiden.[5] Wenn alle drei Rezeptoren ungefähr gleich stark (oder gar nicht) stimuliert werden, entsteht der Eindruck von Grautönen (inklusive Weiß und Schwarz). Wenn die Sinneszellen durch zu intensives Licht überreizt werden, werden wir geblendet. Die drei Rezeptoren sind unterschiedlich empfindlich. Am empfindlichsten ist der Grünrezeptor, darum erscheint uns die Farbe Grün am hellsten. Am unempfindlichsten sind die Blaurezeptoren, während Rot in der Mitte liegt.

In einem durchschnittlichen Auge befinden sich ungefähr 6 Millionen Zapfen. Da die Zapfen nicht so lichtempfindlich sind wie die Stäbchen, können wir in der Dämmerung nur sehr schwer Farben erkennen, da dann nur die Stäbchen „funktionieren“.

[5] In den Augen von Vögeln befinden sich sogar vier Typen von Zapfen. Dadurch können Vögel auch UV-Licht wahrnehmen. Andere Tierarten wie Hunde oder Katzen haben hingegen nur zwei Typen von Zapfen.

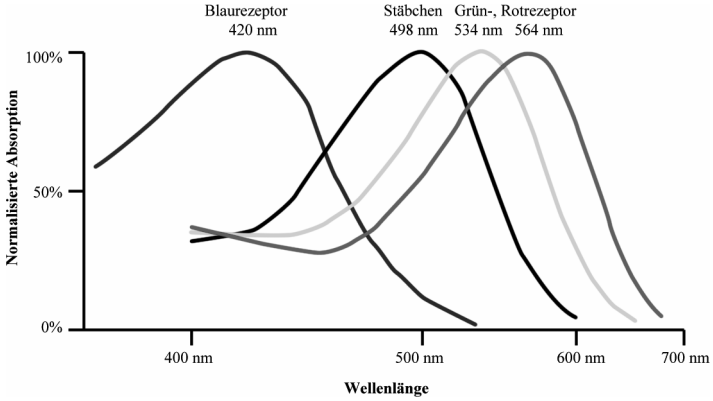


Abbildung 2.38 Absorptionskurven der drei Zapfentypen und der Stäbchen

Was ist Farbe nun also? Der Begriff lässt sich physikalisch nicht sehr gut beschreiben, vielmehr hängt Farbe von unserer Wahrnehmung und auch vom Spektrum der Lichtquelle ab, von der die Umgebung beleuchtet wird. Wenn beispielsweise sagen, dass ein Apfel grün ist, dann heißt das, dass von seiner Oberfläche hauptsächlich grünes Licht reflektiert wird. Licht anderer Wellenlängen wird absorbiert. Betrachten wir den Apfel in weißem Licht, dann erscheint er tatsächlich grün, aber was passiert, wenn wir ihn nur mit rotem Licht bestrahlen? Ein „perfekt grüner" Apfel würde uns dann als schwarz erscheinen, da seine Oberfläche das rote Licht absorbiert und somit kein Licht reflektiert wird, das in unser Auge fallen könnte.

2.7.3 Das RGB-Farbmodell

Da das menschliche Auge, wie bereits beschrieben, Sinneszellen für rotes, grünes und blaues Licht besitzt, ist es nahe liegend, diese drei Farben zu verwenden, um daraus alle für uns wahrnehmbaren Farben zu konstruieren (*Farbsynthese*). Beim *RGB-Farbmodell* (Rot, Grün, Blau oder Red, Green, Blue) wird genau dies getan.[6]

Praktisch alle Farbbildschirme bestehen aus einer Vielzahl roter, grüner und blauer Leuchtzellen (winzig kleine „Lämpchen"), die einzeln angesteuert und in ihrer Leuchtkraft verändert werden können. Ein Bildpunkt besteht dann aus einer roten, einer grünen und einer blauen Zelle. Das gilt sowohl für Röhrenmonitore als auch für LCD-/TFT-Displays, nur dass die Leuchtzellen unterschiedlich funktionieren und angesteuert werden. Da die Leuchtzellen sehr nah beieinander liegen, sind sie für unser Auge aus normaler Distanz nicht einzeln auflösbar und verschmelzen zu einem einzigen Punkt in der Farbe, die sich aus der Überlagerung der drei Grundfarben ergibt.

[6] Tatsächlich kann man mit dem RGB-Farbmodell jedoch nicht wirklich restlos alle für uns wahrnehmbaren Farben erzeugen.

Da sich Rot, Grün und Blau additiv überlagern, spricht man hier auch von einem *additiven Farbmodell*. Wann immer man den Anteil einer der drei Grundfarben erhöht, erhöht man dadurch die Helligkeit der resultierenden Mischfarbe.

Im RGB-Farbmodell wird eine Farbe als dreidimensionaler Vektor dargestellt, dessen Komponenten für die Intensitäten der Grundfarben Rot, Grün und Blau stehen. Deren Werte können dabei zwischen 0 (kein Licht) und 1 (volle Intensität) liegen. In einem Koordinatensystem mit den Achsen Rot, Grün und Blau liegen die darstellbaren Farben dann in einem Würfel von RGB(0, 0, 0) bis RGB(1, 1, 1).

Tabelle 2.1 RGB-Darstellungen einiger Farben

Farbe	RGB-Darstellung
Rot	RGB(1, 0, 0)
Grün	RGB(0, 1, 0)
Blau	RGB(0, 0, 1)
Gelb	RGB(1, 1, 0)
Magenta	RGB(1, 0, 1)
Cyan	RGB(0, 1, 1)
Orange	RGB(1, 0.5, 0)
Schwarz	RGB(0, 0, 0)
Weiß	RGB(1, 1, 1)
50% Grau	RGB(0.5, 0.5, 0.5)

Darstellung im Computer

Im Computer speichert man RGB-Farben meistens in Form eines 24- oder 32-Bit-Werts (Double Word) ab. Für jeden Farbkanal stehen dabei 8 Bits, also 1 Byte zur Verfügung. Dementsprechend sind die Rot-, Grün- und Blauanteile jeweils in 256 Abstufungen darstellbar (0 bis 255), was zu einer Gesamtanzahl von $256^3 = 16\,777\,216$ verschiedenen Farben führt (auch als *True Color* bezeichnet). Bei 32 Bits, was für (32-Bit-)Prozessoren praktischer und schneller ist als mit 24 Bits zu rechnen, bleibt das vierte Byte entweder ungenutzt oder wird für den *Alphakanal* verwendet, mit dem die Transparenz einer Farbe festgelegt werden kann.

Die Zahl von ungefähr 16.7 Millionen Farben entspricht auch der Anzahl der Farben, die die meisten heutigen Grafikkarten und Displays darstellen können. Für die Anzeige auf dem Monitor ist es also nicht nötig, mit noch höherer Genauigkeit zu arbeiten. Bei der Bildverarbeitung, also beim Rechnen mit Bilddaten, kann es sich hingegen schon lohnen, beispielsweise mit Fließkommazahlen statt mit Bytes zu rechnen, da sich sonst Rundungsfehler bei vielen hintereinander angewendeten Operationen schnell bemerkbar machen. Natürlich benötigt ein Bild dann auch wesentlich mehr Speicherplatz. Auch beim *HDRR* (*High Dynamic Range Rendering*), das man als Spezialeffekt heute in fast allen 3D-

Spielen findet, wird bis kurz vor der Darstellung auf dem Monitor mit höherer Genauigkeit gerechnet – aber mehr dazu später.

Wer sich schon einmal mit der Gestaltung von Webseiten mit HTML und CSS beschäftigt hat, der wird die Darstellung von 24-Bit-RGB-Farben als Hexadezimalwert kennen. Jeweils zwei Hexadezimalziffern stehen dabei für einen Farbkanal, zuerst Rot, dann Grün und zuletzt Blau. Die Farbe Rot entspricht dann beispielsweise `#FF0000`, Grün ist `#00FF00`, Blau ist `#0000FF`, und `#FF8000` stellt Orange dar.

Rechnen mit Farben

Da wir eine Farbe als RGB-Vektor darstellen können, können wir mit diesen Vektoren natürlich auch wie gewohnt rechnen. Doch was kommt dabei heraus?

Wenn wir zwei Farben addieren, dann entspricht das einer *additiven Farbmischung*. Rot plus Grün ergibt zum Beispiel Gelb, denn (1, 0, 0) + (0, 1, 0) = (1, 1, 0). Multiplizieren wir einen Farbvektor mit einem Skalar, dann verändern wir damit die Intensität der Farbe. Wenn man zwei Farbvektoren komponentenweise multipliziert, dann kann man sich die eine Farbe als die Farbe einer Lampe vorstellen und die andere Farbe als die Farbe eines Filters, der vor die Lampe gehalten wird. Das Ergebnis ist dann die Farbe, die man durch den Filter sieht. Schicken wir zum Beispiel orangefarbenes Licht durch einen grünen Filter, dann gelangt nur der weniger intensive grüne Anteil von Orange durch den Filter hindurch: $\left(1,\ \frac{1}{2},\ 0\right) * \left(0,\ 1,\ 0\right) = \left(0,\ \frac{1}{2},\ 0\right)$. Bei der Beleuchtung wird die komponentenweise Multiplikation später noch sehr häufig verwendet werden.

Zum Abschluss wollen wir uns noch ansehen, wie man eine Farbe in eine Graustufe umrechnet, also wie man die Helligkeit oder *Luminanz* einer Farbe (auch mit *Y* bezeichnet) berechnen kann. Man könnte auf die Idee kommen, einfach den Mittelwert des Rot-, Grün- und Blauanteils zu nehmen, aber da wir Rot, Grün und Blau unterschiedlich hell wahrnehmen, müssen die Farben gewichtet werden: Rot mit 0.299, Grün mit 0.587 und Blau mit 0.114. Grün nehmen wir somit, wie bereits gesagt, mit Abstand am hellsten wahr, gefolgt von Rot und schließlich Blau. Die Graustufe kann also mit einem Skalarprodukt berechnet werden:

$$Y = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \bullet \begin{pmatrix} 0.299 \\ 0.587 \\ 0.114 \end{pmatrix}$$

2.7.4 Andere Farbmodelle

CMY(K)

Das RGB-Farbmodell ist einfach zu verstehen und anzuwenden, aber es gibt auch noch andere, zum Beispiel das *CMY-Farbmodell*. Im Prinzip handelt es sich dabei um die subtraktive Variante des RGB-Farbmodells. Bei den RGB-Farben geht man von Schwarz aus

und macht die Farbe durch Hinzufügen der Grundfarben Rot, Grün und Blau heller. Bei CMY (Cyan, Magenta, Yellow – also Cyan, Magenta und Gelb) geht man von Weiß aus und zieht die entsprechenden Grundfarben davon ab. Darum spricht man von einem *subtraktiven Farbmodell*. Die Umrechnung zwischen RGB und CMY funktioniert sehr einfach:

RGB nach CMY:	CMY nach RGB:
$C = 1 - R$	$R = 1 - C$
$M = 1 - G$	$G = 1 - M$
$Y = 1 - B$	$B = 1 - Y$

Das CMY-Farbmodell wird für den Farbdruck verwendet. Das Blatt Papier ist weiß, und jeder Farbtropfen, der auf das Papier gebracht wird, absorbiert Teile des einfallenden Lichts. Farben, die heller als das Papier sind, lassen sich darum nicht drucken (dazu müsste man selbst leuchtende Farben verwenden).

CMY(0, 0, 0) ergibt also Weiß, und CMY(1, 1, 1) ergibt Schwarz. Das funktioniert zwar in der Theorie, aber in der Praxis leider nicht perfekt. In der Patrone eines Farbdruckers befinden sich Cyan, Magenta und Gelb. Die Mischung dieser drei Farben ergibt aber kein richtiges Schwarz, da es sehr schwierig ist, die benötigten Farben exakt herzustellen. Darum verwendet man als vierte Farbe in der Patrone Schwarz – einerseits, um damit Grautöne und Schwarz selbst zu drucken, und andererseits zum effizienteren Abdunkeln anderer Farben. Außerdem spart man somit Tinte und sorgt für schärfere Bilder. Das um die Farbe Schwarz erweiterte CMY-Modell bezeichnet man als *CMYK-Farbmodell* (K für Blac**k** oder Key). Die Umrechnung von CMY nach CMYK und zurück funktioniert so:

CMY nach C'M'Y'K:

$$K = \min\{\mathrm{C}, \mathrm{M}, \mathrm{Y}\}$$

$$\begin{pmatrix} C' \\ M' \\ K' \end{pmatrix} = \begin{cases} \vec{0}, \text{ wenn } K = 1 \\ \begin{pmatrix} \frac{C-K}{1-K} \\ \frac{M-K}{1-K} \\ \frac{Y-K}{1-K} \end{pmatrix}, \text{ sonst} \end{cases}$$

C'M'Y'K nach CMY:

$$C = C' \cdot (1 - K) + K$$

$$M = M' \cdot (1 - K) + K$$

$$Y = Y' \cdot (1 - K) + K$$

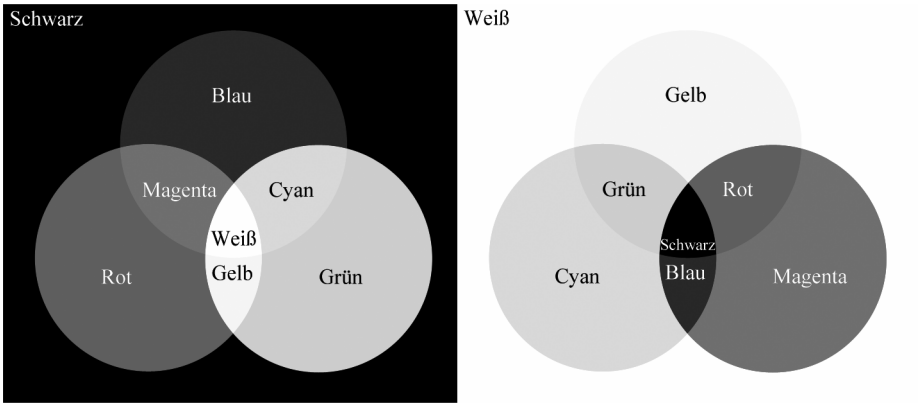


Abbildung 2.39 Additive und subtraktive Farbsynthese

YUV

Ein anderes Farbmodell ist das *YUV-Farbmodell*. Hierbei wird eine Farbe in ihre Luminanz Y und ihre *Chrominanz* (U und V) aufgeteilt. Die Chrominanz bestimmt den Farbton und die Sättigung. Im YUV-Farbraum lassen sich auf Grund dieser Darstellung einige Berechnungen leichter durchführen als im RGB-Farbraum. Für Y liegt der Wertebereich zwischen 0 und 1. U und V können Werte von –0.5 bis 0.5 annehmen.

Bei der JPEG-Kompression wird das Bild zunächst in einen leicht abgewandelten YUV-Farbraum transformiert. Anschließend wird die Tatsache ausgenutzt, dass das menschliche Auge Farbinformationen örtlich nicht so exakt auflösen kann wie Helligkeitsinformationen (weil die Zapfen weit weniger dicht verteilt sind als die Stäbchen), indem der Chrominanzanteil des Bildes auf die halbe Auflösung heruntergerechnet wird. So ergeben sich bereits erste Einsparungen in der Datenmenge, die anschließend durch weitere Kompressionsverfahren noch stärker reduziert wird.

Die Umrechnung zwischen RGB und YUV funktioniert wie folgt:

RGB nach YUV:

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

$$U = (B - Y) \cdot 0.493$$

$$V = (R - Y) \cdot 0.877$$

YUV nach RGB:

$$B = Y + \frac{U}{0.493}$$

$$R = Y + \frac{V}{0.877}$$

$$G = 1.7 \cdot Y - 0.509 \cdot B - 0.245 \cdot R$$

2.8 Beleuchtungsberechnung

Die Beleuchtung spielt für die Wirkung einer gerenderten 3D-Szene eine bedeutende Rolle. Wenn etwas mit Licht und Schatten nicht stimmt, merken wir das sehr schnell, da wir

aus unserem Alltag sehr genaue Vorstellungen haben, wie eine Umgebung unter bestimmten Lichtbedingungen auszusehen hat.

Die Beleuchtung der realen Welt kann im Computer jedoch nahezu unmöglich perfekt nachgestellt werden, da man theoretisch jedes einzelne Photon modellieren und seinen Weg durch die Szene verfolgen müsste. Also muss man die Berechnungen so weit wie möglich vereinfachen, ohne das Ergebnis allzu unrealistisch aussehen zu lassen.

2.8.1 Was ist ein Vertex?

Bevor wir zur eigentlichen Beleuchtung kommen, muss noch ein überaus wichtiger Begriff erklärt werden, der uns von nun an ständig begleiten wird.

Wir haben uns schon ausgiebig mit den Eckpunkten von Dreiecken befasst, beispielsweise bei der Transformation durch Matrizen. Dabei war ein solcher Eckpunkt einfach nur ein Vektor, der die Position des Punkts beschrieb. Aber ein Eckpunkt kann noch mehr Informationen beinhalten als nur eine Positionsangabe, zum Beispiel eine Farbe. Beim Zeichnen eines Dreiecks kann diese Farbe dann *interpoliert* werden. Nehmen wir zum Beispiel ein Dreieck mit einem roten, einem grünen und einem blauen Eckpunkt. Die Pixel, die exakt an diesen Eckpunkten liegen, sollen später genau die entsprechende Farbe erhalten, aber auf der Dreiecksfläche hätten wir gern einen kontinuierlichen Farbverlauf. In der Mitte würden beispielsweise alle drei Eckpunktfarben gleichermaßen zum Ergebnis beitragen, so dass die Farbe dort grau wäre.

Einen Eckpunkt, dem man weitere Informationen hinzufügen kann, also beispielsweise eine Farbe, nennt man *Vertex*. Die Mehrzahl davon lautet *Vertizes*, in Anlehnung an das Wort Index und dessen Plural Indizes. Ein Vertex kann noch viel mehr beinhalten als nur eine Position und eine Farbe, zum Beispiel einen Normalenvektor, der senkrecht zur Oberfläche steht, die durch die Dreiecke angenähert werden soll. Dieser Normalenvektor spielt bei der Beleuchtung eine große Rolle.

Ein weiterer Bestandteil eines Vertex können Texturkoordinaten für die Texturierung sein. Darauf werden wir später noch genauer eingehen.

2.8.2 Das Phong-Modell

Ich möchte Ihnen nun das *Phong-Modell* vorstellen, das in der Praxis gängigste Beleuchtungsmodell für Echtzeitanwendungen. Es handelt sich um ein lokales Beleuchtungssystem, das mit einer sehr groben Annäherung auch die globale Komponente versucht zu berücksichtigen. Noch einmal zur Wiederholung: ein Beleuchtungsmodell nennt man lokal, wenn es nur die örtliche Beleuchtung der Objekte berechnet und dabei weder Schatten berücksichtigt[7] noch die Tatsache, dass Licht von einer Oberfläche auf eine andere reflektiert werden kann und sich die Objekte auf diese Weise gegenseitig beleuchten.

[7] Schattenwürfe können dennoch relativ leicht in ein solches Beleuchtungsmodell eingefügt werden.

Das Phong-Modell wurde bereits 1975 vorgestellt, als man von Echtzeit-3D-Grafik, wie sie heute möglich ist, nur träumen konnte. Das Phong-Modell ist physikalisch gesehen nicht korrekt (Lichtquellen werden als punktförmig betrachtet, und der Energieerhaltungssatz wird verletzt), es liefert aber dennoch gute Ergebnisse. In diesem Modell setzt sich die Beleuchtung aus den folgenden Komponenten zusammen, die wir uns gleich näher ansehen werden:

- *Ambientes Licht* (*Ambient Light*)
- *Diffuses Licht* (*Diffuse Light*)
- *Spekulares Licht* (*Specular Light*)
- *Selbstbeleuchtung* (*Self Illumination*)

2.8.2.1 Ambientes Licht

Das ambiente Licht erhellt die gesamte Szene und beleuchtet somit jede Oberfläche, und zwar unabhängig davon, wo sie sich befindet und wohin sie zeigt. Wenn Sie in einem völlig abgedunkelten Raum eine Taschenlampe einschalten und sie auf die Wand richten, dann wird durch die Reflexion auch der Rest des Raums zumindest ein wenig Licht abbekommen. Durch das ambiente Licht versucht man dem Rechnung zu tragen und erhellt einfach die gesamte Szene um einen bestimmten Wert. Das ist natürlich keine exakte Lösung, sondern nur eine simple Annäherung an eine echte globale Beleuchtung, aber sie erfüllt meistens ihren Zweck.

Die ambiente Beleuchtung A berechnet sich aus dem komponentenweisen Produkt der ambienten Lichtfarbe, die wir A_L nennen wollen, und der ambienten Materialfarbe A_M. Also:

$$A = A_L * A_M$$

Die Unterscheidung zwischen Licht- und Materialfarbe wird uns noch öfter begegnen. Später werden wir jedem Dreieck, das wir zeichnen wollen, ein Material zuordnen und somit definieren, was mit dem einfallenden Licht geschieht.

Normalerweise setzt man die ambiente Lichtfarbe nur auf relativ kleine Werte, zum Beispiel RGB(0.1, 0.1, 0.1). Wenn wir in diesem Fall dann eine ambiente Materialfarbe von Rot hätten, also RGB(1, 0, 0), dann wäre A = RGB(0.1, 0, 0), was einem sehr dunklen Rot entspricht. Die Grün- und Blauanteile des ambienten Lichts würden dann von der Oberfläche nicht reflektiert.

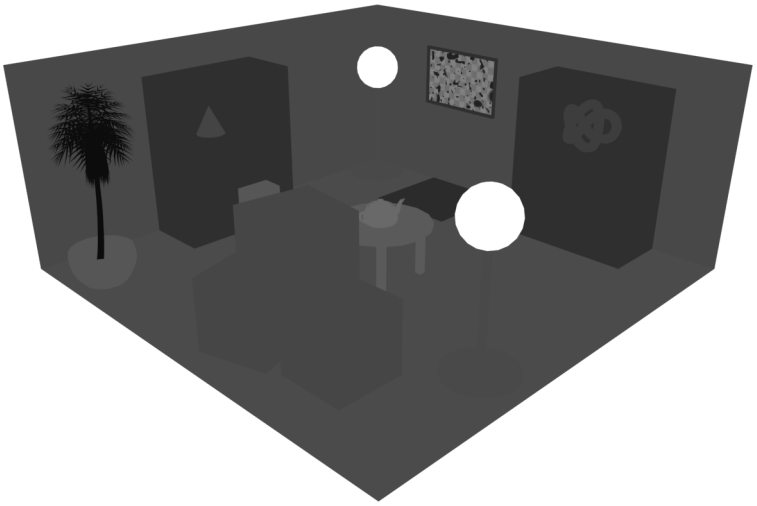


Abbildung 2.40 Szene, die ausschließlich mit ambientem Licht beleuchtet wurde. Es sind kaum Kontraste sichtbar, aber die Szene hat eine gewisse Grundhelligkeit. Die beiden Kugeln deuten die Position der punktförmigen Lichtquellen an.

2.8.2.2 Diffuses Licht

Die diffuse Beleuchtung entsteht durch die direkte Lichteinstrahlung von der Lichtquelle auf die Oberfläche. Von dort wird das Licht gestreut und fällt schließlich in das Auge des Betrachters. Dabei hängt die Stärke der Beleuchtung von dem Winkel ab, unter dem die Lichtstrahlen auftreffen. Je flacher der Winkel ist, desto weniger Licht erreicht die Oberfläche. Betrachten wir als Beispiel einen Sonnenuntergang: wenn die Sonne tief am Horizont steht, beleuchtet sie die Oberfläche nur unter einem sehr flachen Winkel, wodurch dort nur wenig Licht pro Quadratmeter auftrifft. Steht die Sonne jedoch im Zenit, dann treffen die Lichtstrahlen senkrecht auf die Erde, und die Beleuchtung ist maximal.

Genauer gesagt nimmt die Beleuchtung im Phong-Modell nicht linear mit kleiner werdendem Winkel ab, sondern sie ist proportional zum Kosinus des Winkels zwischen der Richtung von der Oberfläche zur Lichtquelle (Lichtvektor) und dem Normalenvektor der Oberfläche. Scheint das Licht senkrecht auf die Oberfläche, dann beträgt der Winkel zwischen Normalenvektor und Lichtvektor 0°. Da der Kosinus von 0° gleich 1 ist, ist die Beleuchtung dann maximal. Beim Beispiel des Sonnenuntergangs wäre der Winkel nah bei 90° und der Kosinus fast null, was wie erwünscht zu einer sehr schwachen Beleuchtung führen würde. Was passiert, wenn das Licht von hinten auf die Oberfläche fällt? Dann ist der Winkel größer als 90° und der Kosinus negativ. Damit wir hier keine Beleuchtung „abziehen“, machen wir einfach alle negativen Werte zu null.

Den Lichtvektor wollen wir $\vec{L}$ nennen und den Normalenvektor $\vec{N}$. Wenn beide Vektoren normiert sind ($\vec{L}_0$ und $\vec{N}_0$), dann entspricht ihr Skalarprodukt genau dem gesuchten Kosinus des Winkels zwischen ihnen. Die diffuse Beleuchtung D kann nun wie folgt berechnet werden:

$$D = D_L * D_M \cdot \max\left\{\cos\sphericalangle\left(\vec{L},\ \vec{N}\right),\ 0\right\}$$
$$= D_L * D_M \cdot \max\left\{\vec{L}_0 \bullet \vec{N}_0,\ 0\right\}$$

D_L und D_M geben hierbei die diffuse Lichtfarbe und die diffuse Materialfarbe an. Der diffuse Anteil ist meistens der Größte von allen.

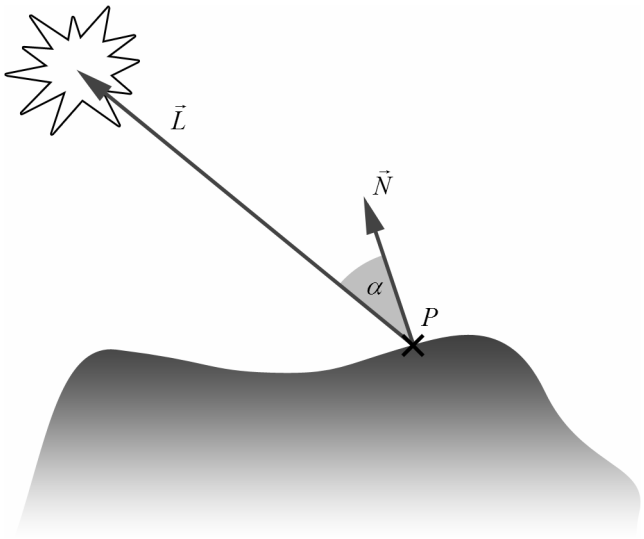


Abbildung 2.41 Diffuse Beleuchtung: der Winkel *α* zwischen dem Normalenvektor und der Richtung zur Lichtquelle bestimmt den Grad der Beleuchtung. Je kleiner er ist, desto stärker wird die Oberfläche beleuchtet.

Anders als das ambiente Licht bringt die diffuse Beleuchtung Kontrast in die Szene und erlaubt es dem Betrachter, ihren räumlichen Aufbau besser zu erkennen.

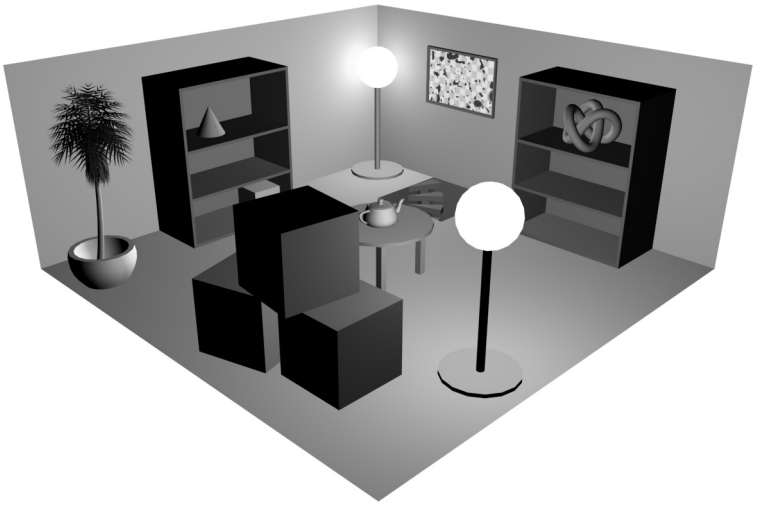


Abbildung 2.42 Diffuser Anteil der Beleuchtung: nun lässt sich deutlich besser erkennen, wie die Objekte in der Szene angeordnet sind. Oberflächen, die den Lichtquellen abgewandt sind (beispielsweise die Seitenwände der Regale) sind dunkel.

2.8.2.3 Spekulares Licht

Oft möchte man glänzende Oberflächen wie zum Beispiel Porzellan oder poliertes Metall darstellen. Den typischen Glanzeffekt (*Specular Highlights*) kann man jedoch nicht nur mit Hilfe von ambientem und diffusem Licht erzielen. Genau zu diesem Zweck gibt es das spekulare Licht. Das spekulare Licht modelliert die Spiegelung der Lichtquelle in der Objektoberfläche. Bei der Berechnung spielt im Gegensatz zur diffusen Beleuchtung auch die Position des Betrachters eine Rolle, denn ob die von der Lichtquelle ausgesandten und von der Oberfläche reflektierten Strahlen in sein virtuelles Auge treffen, hängt natürlich von seinem Aufenthaltsort ab.

Die spekulare Beleuchtung funktioniert wie folgt: man berechnet zunächst, in welche Richtung das auf die Oberfläche auftreffende Licht im Falle einer perfekten Reflexion reflektiert würde. „Perfekte Reflexion“ bedeutet ganz einfach, dass der Einfallswinkel gleich dem Ausfallswinkel ist (gemessen am Normalenvektor der Oberfläche). Bei sehr glatten Oberflächen ist das der Fall, bei raueren Oberflächen hingegen werden die Strahlen in mehr oder weniger zufällige Richtungen reflektiert (dies wird durch die diffuse Beleuchtung modelliert). Wenn wir nun den perfekten Reflexionsvektor $\vec{R}$ kennen, dann müssen wir noch herausfinden, wie genau sich der Betrachter in der „Schussbahn“ der perfekt reflektierten Strahlen befindet. Dazu bestimmen wir mit dem Skalarprodukt den Kosinus des Winkels zwischen $\vec{R}$ und der Richtung vom zu beleuchtenden Punkt zum Beobachter. Diesen Vektor wollen wir $\vec{V}$ nennen. Das Skalarprodukt wird nun mit p, dem *spekularen Exponenten* potenziert. Dieser Exponent, der normalerweise zwischen 1 und ∞ liegt, bestimmt die Stärke und Abgegrenztheit des Glanzeffekts. Je höher man den Wert wählt, desto schärfer und kleiner erscheinen die Glanzpunkte auf der Oberfläche, da durch das

Potenzieren kleine Werte noch kleiner werden, Werte nahe bei 1, wo der Betrachter also fast oder exakt in der Richtung der perfekten Reflexion steht, hingegen weniger betroffen sind.

Die Vektoren $\vec{R}$ und $\vec{V}$ berechnen wir wie folgt:

$$\vec{R} = -\vec{L} + 2 \cdot \vec{N} \cdot \left(\vec{N}_0 \bullet \vec{L} \right)$$
$$\vec{V} = \overrightarrow{PC} = \vec{C} - \vec{P}$$

Anschließend lässt sich die spekulare Beleuchtung errechnen:

$$S = S_{\mathrm{L}} * S_{\mathrm{M}} \cdot \max \left\{ \cos^{p} \measuredangle \left(\vec{R}, \vec{V} \right),\ 0 \right\}$$
$$= S_{\mathrm{L}} * S_{\mathrm{M}} \cdot \max \left\{ \left(\vec{R}_0 \bullet \vec{L}_0 \right)^{p},\ 0 \right\}$$

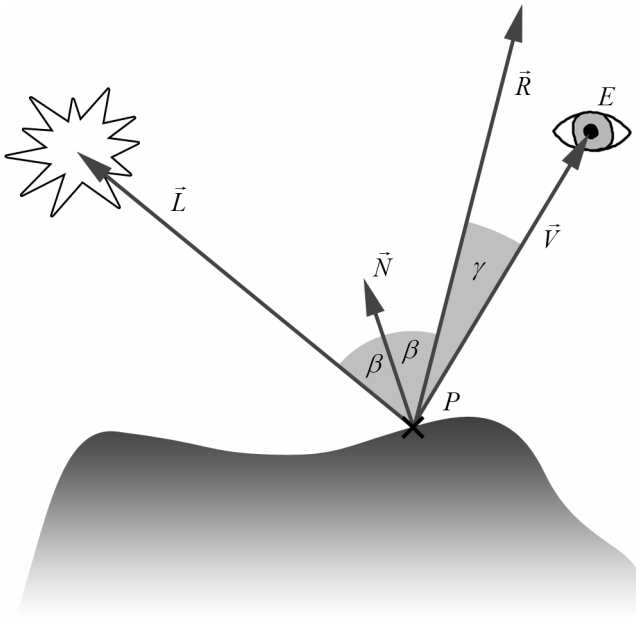


Abbildung 2.43 Spekulare Beleuchtung: Licht wird von der Oberfläche perfekt reflektiert. Je kleiner der Winkel *γ*, desto mehr davon fällt direkt in das Auge des Betrachters beim Punkt *E*.

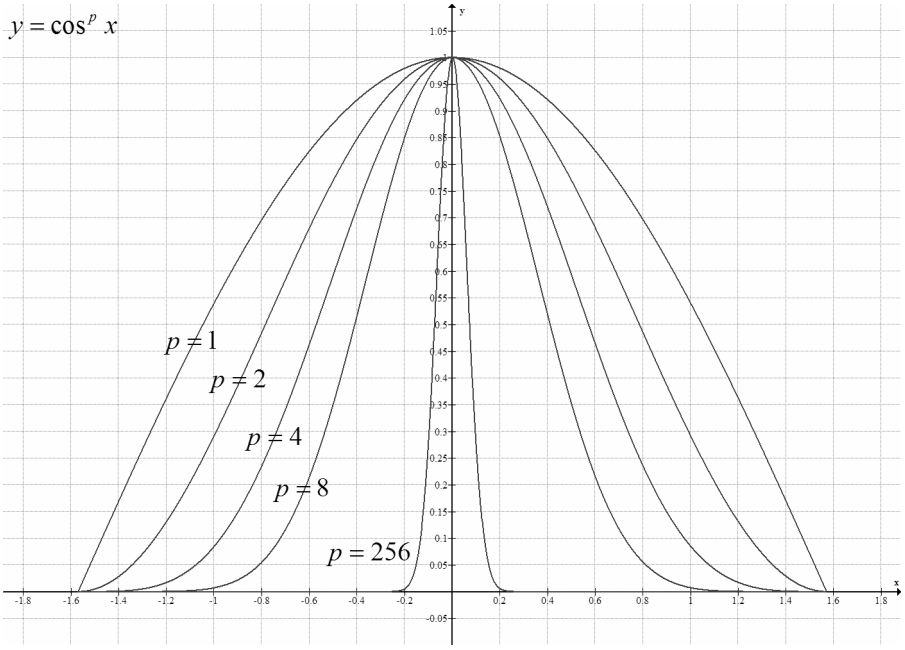


Abbildung 2.44 Kosinuskurven mit verschiedenen Exponenten: je höher der Exponent, desto abgegrenzter ist der Glanzeffekt bei der spekularen Beleuchtung. Die Kurve mit dem Exponenten 256 (ganz innen) erreicht nur für sehr kleine Winkel hohe Werte. Der Übergang ist im Gegensatz zu dem der anderen Kurven sehr scharf.

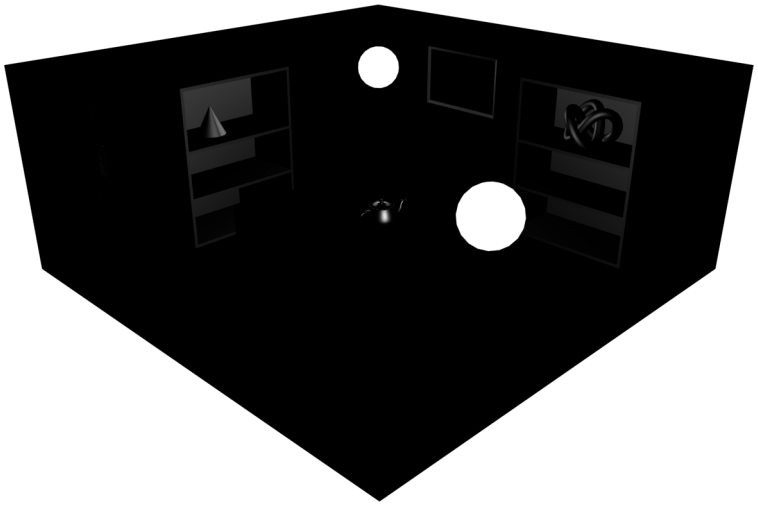


Abbildung 2.45 Spekularer Anteil der Beleuchtung: hier sind nur die glänzenden Oberflächen zu erkennen, wie die Teekanne auf dem Tisch oder die Objekte in den Regalen. Die Lampen sind in dieser Szene eine Ausnahme, da sie unabhängig von der Beleuchtung immer weiß sind.

2.8.2.4 Selbstbeleuchtung

Manchmal möchte man, dass ein Objekt eine gewisse Helligkeit hat, auch ohne von einer Lichtquelle angestrahlt zu werden. Ein gutes Beispiel dafür sind Kontrolllämpchen oder Bildschirme. Vor allem bei einem kleinen Lämpchen kann man es sich meistens nicht leisten, dafür eine Lichtquelle zu verwenden. Darum gibt es die Selbstbeleuchtung, die einfach zu den drei anderen Beleuchtungsanteilen hinzuaddiert wird. Objekte, die man auf diese Weise leuchten lässt, sind natürlich nicht in der Lage, andere Objekte zu beleuchten, da sie keine echten Lichtquellen sind.

2.8.2.5 Und nun alles zusammen

Beim Phong-Modell werden der ambiente, diffuse und spekulare Anteil der Beleuchtung sowie die Selbstbeleuchtung addiert, um die Gesamtbeleuchtung zu erhalten. Die vier Anteile können jeweils noch mit einem Faktor gewichtet werden, wobei nach dem Phong-Modell die Summe der Faktoren 1 ergeben soll.

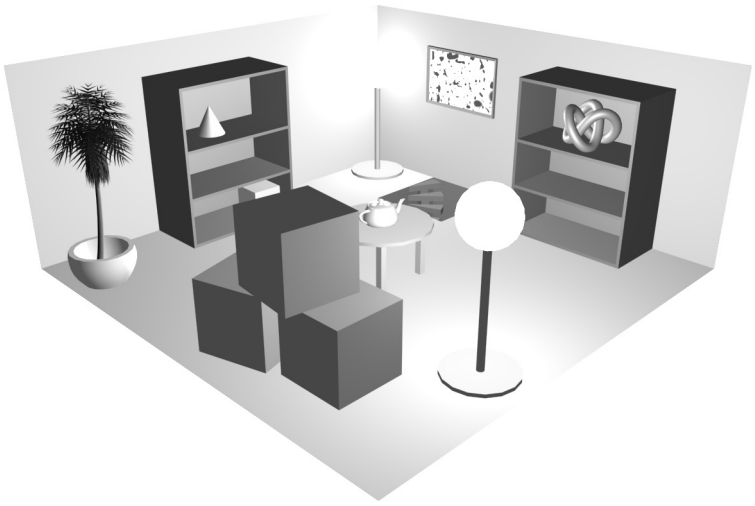


Abbildung 2.46 Ambiente, diffuse und spekulare Beleuchtung zusammengerechnet

Mehrere Lichtquellen

Bislang haben wir bei unseren Berechnungen immer angenommen, dass es nur eine einzige Lichtquelle gibt. In der Praxis möchte man natürlich nicht mit dieser Einschränkung arbeiten. Zum Glück ist es ziemlich einfach, mehrere Lichtquellen zu berücksichtigen. Dazu muss man lediglich die Beleuchtung durch die einzelnen Lichtquellen addieren.

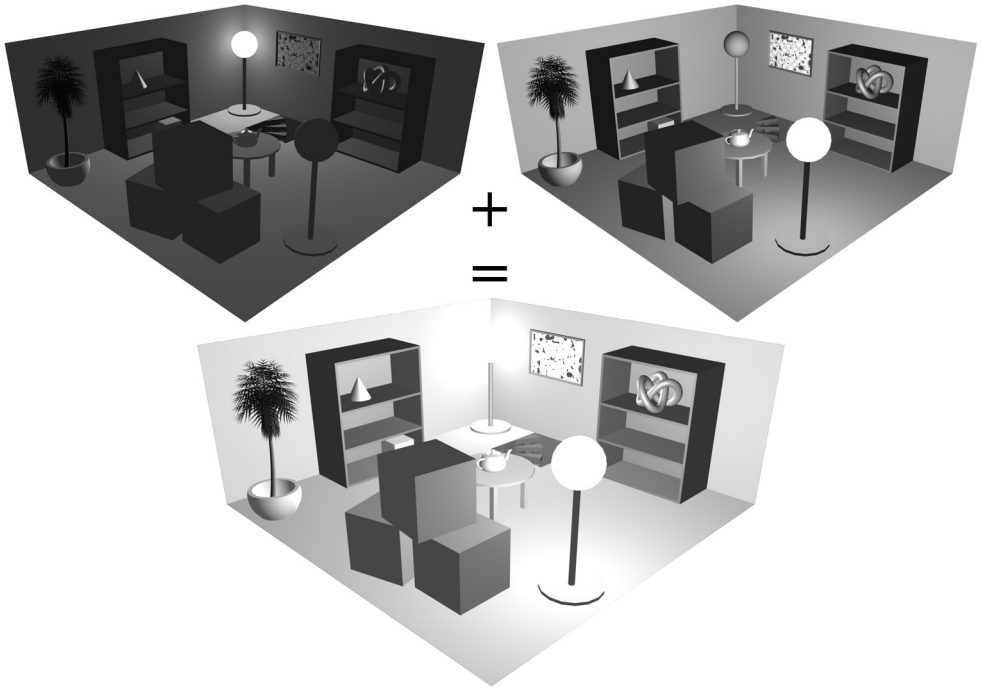


Abbildung 2.47 Bei mehreren Lichtquellen addiert man die Beleuchtung durch die einzelnen Lichtquellen und erhält eine korrekte Gesamtbeleuchtung.

2.8.2.6 Verschiedene Arten von Lichtquellen

Üblicherweise unterscheidet man zwischen drei Arten von Lichtquellen: *Punktlichtquellen*, *Richtungslichtquellen* und *Spotlichtquellen*.

Punktlichtquellen

Eine Punktlichtquelle ist eine punktförmige Lichtquelle mit einer wohldefinierten Position im Raum. Sie strahlt ihr Licht gleichmäßig in alle Richtungen ab. Am ehesten könnte man sie mit einer sehr kleinen Glühbirne vergleichen.

In der Realität nimmt die Beleuchtung mit dem Quadrat der Entfernung zur Lichtquelle ab. Ein Punkt B, der viermal so weit von einer Lichtquelle entfernt ist wie ein Punkt A, bekommt also im Vergleich zu A nur ein Sechzehntel der Beleuchtung ab. Diese Tatsache lässt sich auch in der Computergrafik sehr leicht umsetzen, indem man die Beleuchtung jedes Punkts durch das Quadrat seiner Distanz zur Lichtquelle dividiert.

Punktlichtquellen werden in Computerspielen meistens für Lampen, Feuer, Explosionen oder Ähnliches verwendet.

Richtungslichtquellen

Richtungslichtquellen sind Punktlichtquellen, die unendlich weit entfernt sind. Daher kommen ihre Lichtstrahlen perfekt parallel in der Szene an. Sie haben keine Position, sondern werden durch die Richtung ihrer ankommenden Lichtstrahlen beschrieben. In der Realität gibt es natürlich keine unendlich weit entfernten Lichtquellen, doch wenn man beispielsweise die Sonne betrachtet, dann können die Lichtstrahlen, die hier auf der Erde ankommen, schon fast als parallel betrachtet werden.

Bei Punktlichtquellen müssen wir für jeden Punkt den Lichtvektor, also den Vektor vom Punkt zur Lichtquelle hin bestimmen und ihn normieren (was eine teures Wurzelziehen mit sich bringt), bei Richtungslichtquellen ist dieser Lichtvektor jedoch immer gleich. Richtungslichtquellen sind also gern gesehen, da sie uns einige Rechenoperationen sparen und daher schneller verarbeitet werden können als Punktlichtquellen.

In der Praxis setzt man Richtungslichtquellen hauptsächlich für Sonnenlicht oder Hintergrundbeleuchtung ein.

Spotlichtquellen

Eine Spotlichtquelle ist eine Punktlichtquelle, die zusätzlich noch eine Orientierung hat und ihr Licht nur innerhalb eines bestimmten Lichtkegels abstrahlt, so wie beispielsweise ein Scheinwerfer. Zu den Parametern einer Spotlichtquelle zählen meist zwei Winkel und ein Abschwächungswert, mit denen man den Lichtkegel und die Verteilung der Lichtintensität innerhalb des Kegels festlegt.

Spotlichtquellen werden für Scheinwerfer, Taschenlampen und andere Lichtquellen verwendet, die ihr Licht hauptsächlich in eine bestimmte Richtung abstrahlen.

2.8.3 Shading

Jetzt haben Sie gelernt, wie man die Beleuchtung eines Punkts auf einer Oberfläche mit Hilfe des Phong-Modells berechnen kann. Der nächste Schritt besteht darin, diese Beleuchtung tatsächlich durchzuführen. Wie genau man das tut, bestimmt das *Shading*.

2.8.3.1 Flat-Shading

Beim *Flat-Shading* führt man die Beleuchtungsberechnung immer nur für einen einzigen Punkt pro Primitive aus, bei einem Dreieck beispielsweise für den Mittelpunkt oder einen der drei Eckpunkte. Die sich dadurch ergebende Farbe verwendet man dann für die gesamte Primitive. Flat-Shading ist das einfachste und schnellste Shading-Verfahren. Es wird heute eigentlich kaum noch eingesetzt, höchstens für bestimmte Spezialeffekte.

Beim Flat-Shading kann man jede einzelne Primitive erkennen. Oberflächen erscheinen nicht glatt, sondern hart und kantig.

2.8.3.2 Gouraud-Shading

Die Weiterentwicklung des Flat-Shadings ist das *Gouraud-Shading*. Hierbei bestimmt man die Beleuchtung für alle Eckpunkte und interpoliert die Ergebnisse dann über die Primitiven.

Gouraud-Shading produziert weiche Farbverläufe, aber die Qualität der Beleuchtung hängt davon ab, wie fein die Oberflächen trianguliert sind. Stellen Sie sich hierzu eine Mauer vor, die von einer Lichtquelle direkt vor ihr beleuchtet wird. Wenn die Mauer sehr grob trianguliert ist, dann kann es passieren, dass sie von der Lichtquelle nur sehr schwach beleuchtet wird – nämlich dann, wenn sich in der Nähe der Lichtquelle kein Vertex befindet. Wenn sich die Vertizes nur an den Kanten der Mauer befinden, dann wird die Beleuchtung auch nur dort berechnet und über die Maueroberfläche interpoliert. Da die Beleuchtung am Rand eher schwach ist, ist die Gesamtbeleuchtung ebenfalls schwach.

Auch dieses Shading-Verfahren wird heute immer seltener eingesetzt.

2.8.3.3 Phong-Shading

Kommen wir nun zum Spitzenreiter der Shading-Verfahren: dem *Phong-Shading* (nicht zu verwechseln mit dem Phong-Beleuchtungsmodell). Beim Phong-Shading wird die Beleuchtung nicht mehr auf Basis der Vertizes oder der Primitiven berechnet, sondern für jeden einzelnen Pixel. Die Beleuchtung erreicht immer die höchst mögliche Auflösung und ist völlig unabhängig vom Grad der Triangulierung. Natürlich erhöht sich dadurch im Allgemeinen der Rechenaufwand.

Phong-Shading wird mit *Pixel-Shadern* realisiert und kann mit anderen Verfahren wie *Normal-Mapping* (manchmal auch aus *Bump-Mapping* bezeichnet) kombiniert werden, um Oberflächen noch detaillierter erscheinen zu lassen. Heute, wo die Leistung der Grafikchips dafür ausreicht, setzen die meisten Spiele diese Per-Pixel-Beleuchtung ein.

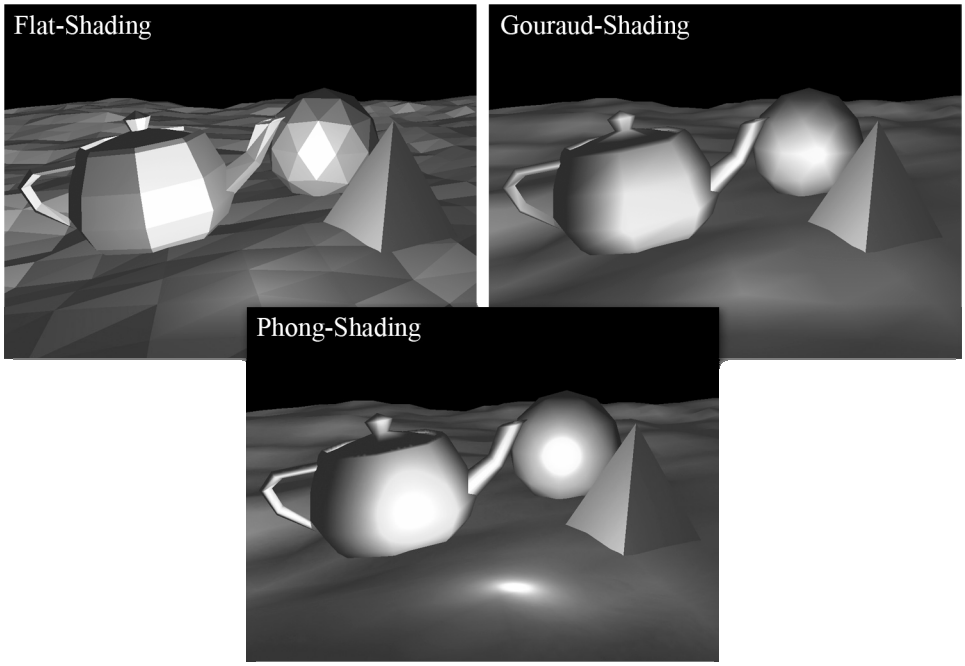


Abbildung 2.48 Vergleich der Shading-Verfahren: beachten Sie, dass beim Phong-Shading auch auf dem sehr grob triangulierten Boden noch ein Glanzpunkt erscheint, was beim Gouraud-Shading nicht der Fall ist.

2.8.4 Was ist mit Schatten?

Wahrscheinlich ist Ihnen beim Betrachten der letzten Abbildungen aufgefallen, dass es keine echten Schatten gibt. Das liegt daran, dass Schatten vom Phong-Modell ganz einfach ignoriert werden. Es wurde gesagt, dass die meisten Computerspiele mit dem Phong-Modell arbeiten, aber Schatten fehlen heute in keinem Spiel mehr. Wie machen diese Spiele das also?

Das Grundproblem

Um Schattenwürfe korrekt darzustellen, sind Informationen über die gesamte Szene erforderlich. Darum lassen sich Schatten mit einem rein lokalen Beleuchtungssystem nicht berechnen.

Wenn ein Punkt im Schatten liegt, heißt das, dass irgendein Teil der Szene den Weg von der Lichtquelle zu diesem Punkt hin blockiert. Oder anders gesagt: wenn die Lichtquelle den Punkt „sehen“ kann, dann liegt er nicht im Schatten.

Es gibt viele Ansätze, mit denen man dieses Problem mehr oder weniger gut lösen kann. Alle davon haben ihre Vor- und Nachteile, so dass es nicht *die* Lösung gibt. Es hängt immer von der Situation und den Anforderungen ab, welche davon sich als am besten geeig-

net herausstellt. Ich werde hier die Wichtigsten kurz vorstellen. Später, wenn es an die Implementierung geht, wird kein Detail ausgelassen.

2.8.4.1 Shadow-Mapping

Ein sehr intuitiver, bildbasierter Ansatz zur Berechnung von Schatten ist das *Shadow-Mapping*. Intuitiv ist es deshalb, weil es genau nach dem oben genannten Prinzip funktioniert: alles, was die Lichtquelle nicht sehen kann, liegt im Schatten.

Beim Shadow-Mapping rendert man vor dem Rendern der eigentlichen Szene alle relevanten Objekte in die so genannte *Shadow-Map*, und zwar aus der Sicht der Lichtquelle. Bei der Shadow-Map handelt es sich um eine Textur, die Tiefen- statt Farbinformationen speichert. Für die Shadow-Map ist also nicht die Farbe der Objekte von Interesse, sondern nur ihre Tiefe oder ihr Abstand zur Lichtquelle.

Nachdem die Shadow-Map gefüllt ist, kann man mit ihr auf sehr effiziente Weise feststellen, ob ein beliebiger Punkt im Schatten liegt oder nicht (auch *Schattentest* genannt). Dazu berechnet man durch eine Reihe von Transformationen, auf welche Stelle in der Shadow-Map der Punkt abgebildet wird und in welcher Distanz er zur Lichtquelle liegt. Nun schlägt man in der Shadow-Map die berechneten Koordinaten nach und liest die dort gespeicherte Tiefe. Dieser Tiefenwert wird nun mit dem berechneten Wert verglichen. Ist der Wert in der Shadow-Map kleiner (mit einer gewissen Toleranz), dann „sieht“ die Lichtquelle an dieser Stelle einen anderen, näher gelegenen Teil der Szene – der Punkt liegt also im Schatten. Sind die Werte jedoch annähernd gleich, dann erreichen ihn die Lichtstrahlen ungehindert.

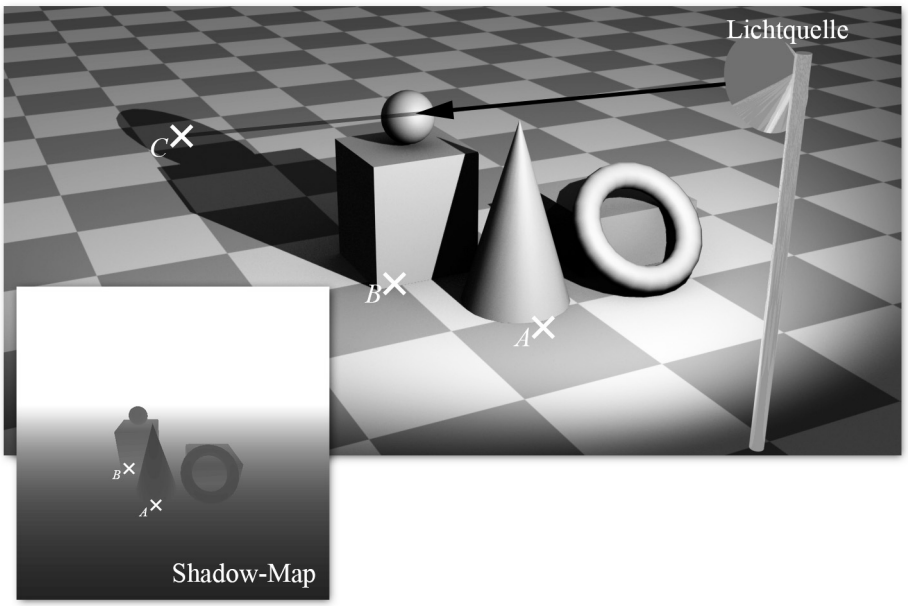


Abbildung 2.49 Shadow-Mapping: die Tiefeninformationen werden aus der Sicht der Lichtquelle in die Shadow-Map gerendert, um feststellen zu können, ob ein Punkt im Schatten liegt oder nicht. Die Punkte *A* und *B* sind in der Shadow-Map sichtbar, werden also von der Lichtquelle angestrahlt. Der Punkt *C* jedoch kann von der Lichtquelle nicht gesehen werden und liegt daher im Schatten.

Vor- und Nachteile

Ein Vorteil des Shadow-Mappings besteht darin, dass das Verfahren vollständig von der Grafikkarte durchgeführt werden kann, ohne dass der Hauptprozessor eingreifen muss. Mit Shadow-Mapping ist es außerdem möglich, auch solche Objekte Schatten werfen zu lassen, die mit Hilfe einer größtenteils transparenten Textur gerendert werden. Das ist häufig für Gras, Büsche oder die Blätter von Bäumen der Fall. Hierfür verwendet man in der Regel nur ein einfaches Viereck, das man dann mit einer Textur belegt, die das gewünschte Motiv enthält, zum Beispiel ein Grasbüschel. Der Rest der Textur ist transparent. Die vom Shadow-Mapping produzierten Schatten sind nicht zwangsläufig scharfkantig, sondern können Übergänge von Licht nach Schatten aufweisen – normalerweise ein erwünschter Effekt. Ein weiterer wichtiger Vorteil ist, dass man mit Shadow-Mapping mehrere Lichtquellen gleichzeitig in einem Render-Durchgang verarbeiten kann.

Doch das Shadow-Mapping kommt nicht ohne Nachteile daher. So eignet sich diese Technik in ihrer ursprünglichen Form beispielsweise nicht für Punktlichtquellen, da hierzu eine 360°-Rundumsicht gerendert werden müsste, was nicht ohne Weiteres möglich ist (man kann die Szene jedoch mehrfach rendern, jeweils mit einer anderen Blickrichtung). Shadow-Mapping funktioniert nur mit Richtungslichtquellen (hier wird eine orthogonale Projektion verwendet) und Spotlichtquellen (perspektivische Projektion). Weiterhin ist die Qualität der Schatten stark von der Auflösung und der Bittiefe der Shadow-Map abhängig.

Erhöht man die Auflösung oder die Bittiefe, erhöht sich damit jedoch auch der Speicherverbrauch und im Allgemeinen damit auch die zum Rendern benötigte Zeit. Zusätzlich kommt es bei ungünstigen Konstellationen von Lichtquelle und Kamera zu unschönen Artefakten, die sich nur mit ausgefeilten Tricks unter Kontrolle bekommen lassen.

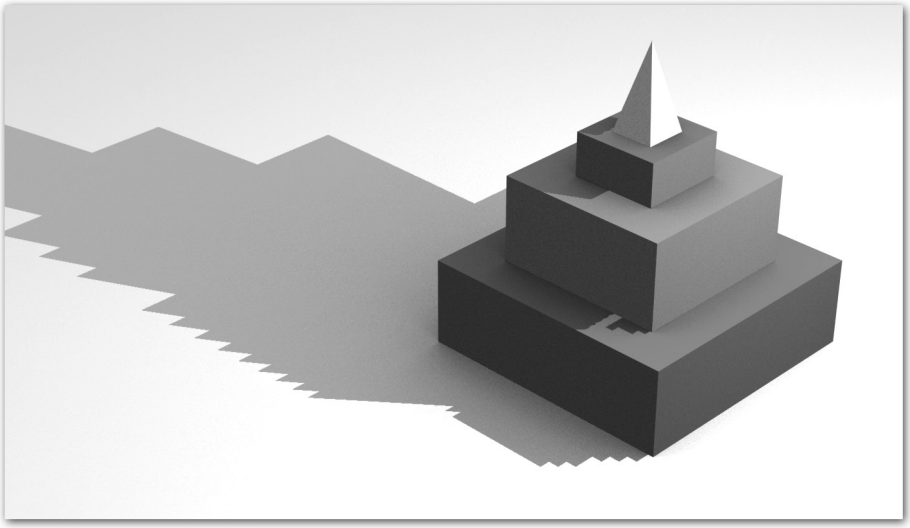


Abbildung 2.50 Wird die Shadow-Map zu klein gewählt oder der Platz schlecht ausgenutzt, entstehen Artefakte wie diese. Hier sind die einzelnen Pixel in der Shadow-Map klar zu erkennen. Mit einem Filter kann das Problem kaschiert werden, oder man vergrößert die Shadow-Map.

2.8.4.2 Andere Verfahren

Stencil-Shadowing

Neben dem Shadow-Mapping existiert noch ein anderes sehr populäres Verfahren, das *Stencil-Shadowing*. Hierbei handelt es sich um ein geometrisches Verfahren, bei dem die Silhouette eines Objekts entlang der Richtung zur Lichtquelle sozusagen in die Unendlichkeit lang gezogen wird. Je nach dem, wie sich die Vorder- und Rückseiten dann auf dem Bildschirm überlappen, ergibt sich eine Licht- oder eine Schattenregion. Dieses Verfahren erzeugt perfekte scharfkantige Schatten („binäre Schatten“), ist im Allgemeinen langsamer als Shadow-Mapping und benötigt für mehrere Lichtquellen ebenso viele Render-Durchgänge. Der geometrische Aufbau der Objekte muss außerdem bestimmten Bedingungen genügen, damit die Schatten korrekt erscheinen, und da das Stencil-Shadowing ein rein geometrisches Verfahren ist, berücksichtigt es keine Löcher, die durch teilweise transparente Texturen erzeugt werden.

Als Vorteil ist die im Vergleich zum Shadow-Mapping relativ einfache und problemlose Implementierung anzuführen. Ein weiterer Grund, der einen dazu bewegen könnte, auf

diese Technik zu setzen, ist dass man mit dem Stencil-Shadowing auch Punktlichtquellen verarbeiten kann.

Vorberechnete Beleuchtung

Bei statischen, also nicht beweglichen Umgebungen und Lichtquellen bietet es sich an, die Beleuchtung im Vorhinein zu berechnen und in Texturen zu speichern (so genannte *Light-Maps*). Das hat den Vorteil, dass bei der Vorberechnung globale Beleuchtungsalgorithmen angewendet werden können, die sehr hochqualitative Ergebnisse liefern (weiche Schatten, Lichtreflexionen, realistisches Tageslicht, …). In Echtzeit ist dies heute noch nicht mit angemessener Geschwindigkeit machbar.

Vorberechnete Beleuchtung kommt in fast allen heutigen 3D-Spielen zum Einsatz, vor allem bei der statischen Levelgeometrie (Landschaften, Gebäude, Korridore) – natürlich in Kombination mit einem dynamischen Verfahren wie Shadow-Mapping oder Stencil-Shadowing.

2.9 Die Rendering-Pipeline

In den Abschnitten dieses Kapitels, die nun hinter uns liegen, haben wir uns vor allem mit den mathematischen und geometrischen Grundlagen der 3D-Grafik beschäftigt. Aber es fehlen noch die Abläufe im Großen – was muss alles passieren, damit aus einem Haufen Punkte, Matrizen, Texturen und Lichtquellen ein fertiges Bild auf den Bildschirm gezeichnet werden kann?

Die Rendering-Pipeline beschreibt genau diese Schritte. Sie sind praktisch immer gleich, egal ob beispielsweise Direct3D, OpenGL oder eine andere Grafikschnittstelle verwendet wird. Manche Teile der Pipeline sind fest in der Grafikhardware verdrahtet und können nur begrenzt verändert werden, andere hingegen sind mittlerweile durch Vertex-, Pixel- und Geometry-Shader komplett programmierbar. Im Allgemeinen wird die Hardware immer flexibler, aber die grundlegenden Schritte, die hier beschrieben werden, werden sich vermutlich nicht bedeutend ändern.

Ich werde in diesem Abschnitt nicht alles bis in letzte Detail erläutern. Sie sollen hier nur einen Überblick erhalten. Später, wenn wir uns mit Direct3D beschäftigen, werden die hier angesprochenen Themen vertieft.

2.9.1 Eingabe

Am Anfang der Rendering-Pipeline existiert die Szene nur in Form einer mathematischen Beschreibung mit Hilfe von einzelnen Vertizes und Informationen darüber, ob sie zu Dreiecken oder Linien verbunden oder einfach nur als Punktwolke dargestellt werden sollen. Außerdem werden Materialeigenschaften (Farben, Texturen), Transformationsmatrizen und Lichtquellen festgelegt.

Dreiecke, Linien und Punkte nennt man übrigens *Primitive*. Die Grafikkarte kann sie direkt zeichnen. Alle komplexeren Formen müssen aus Primitiven zusammengesetzt werden, zum Beispiel ein Viereck aus zwei Dreiecken oder ein Auto aus einigen tausenden Dreiecken.

2.9.2 Welttransformation

Der erste Schritt in der Rendering-Pipeline besteht darin, die Vertizes aus dem lokalen Koordinatensystem in das Weltkoordinatensystem zu transformieren. Sie erinnern sich: wenn ein 3D-Objekt modelliert wird, dann befinden sich seine Vertizes im lokalen Koordinatensystem des Objekts. Mit Hilfe der *Welttransformationsmatrix* (kurz: *Weltmatrix*) wird dieses Objekt dann in der Szene platziert. Die Weltmatrix ist normalerweise eine Kombination aus einer Rotations- und Translationsmatrix, um die Position und Ausrichtung des Objekts festzulegen, und eventuell auch aus einer Skalierungsmatrix, um die Größe zu bestimmen.

2.9.3 Kameratransformation

Nach der Transformation ins Weltkoordinatensystem werden die Vertizes mit Hilfe einer Kameramatrix weiter ins Kamerakoordinatensystem transformiert. Erst dadurch wird es möglich, den Beobachter durch die Szene zu bewegen. Die Kamera liegt in diesem Koordinatensystem immer im Koordinatenursprung, also bei (0, 0, 0), und sie schaut in Richtung (0, 0, 1). Alle Koordinaten sind nun also relativ zur Kamera.

2.9.4 Per-Vertex-Beleuchtung und -Nebel

Wenn sich alle Vertizes in einem gemeinsamen Koordinatensystem, nämlich dem Kamerakoordinatensystem, befinden, kann die Beleuchtung auf Vertexbasis durchgeführt werden, zum Beispiel nach dem bereits vorgestellten Phong-Modell.

Sie erinnern sich: wenn die Beleuchtung nur auf Vertexbasis durchgeführt wird (Gouraud-Shading), ist das Ergebnis davon abhängig, wie fein die Oberflächen trianguliert sind, denn die Beleuchtung wird über die Dreiecke interpoliert.

Vielleicht wundern Sie sich, warum die Beleuchtung im Kamerakoordinatensystem geschieht und nicht im Weltkoordinatensystem. Das liegt daran, dass die spekulare Beleuchtung von der Kameraposition abhängig ist und sich daher im Kamerakoordinatensystem leichter berechnen lässt. Prinzipiell ist es aber egal, in welchem Koordinatensystem die Beleuchtung stattfindet, solange sich die Objekte und die Lichtquellen im selben System befinden (sonst würden die Berechnungen keinen Sinn machen).

Neben der Beleuchtung kann man an dieser Stelle in der Pipeline auch Nebelberechnungen durchführen. Dabei berechnet man für jeden Vertex anhand dessen Tiefe relativ zur Kamera, wie stark er vom Nebel beeinflusst werden soll. Je höher die *z*-Koordinate, desto ausge-

prägter soll der Nebeleffekt sein. Durch Nebel lässt sich verschleiern, dass die Sichtweite auf Grund der fernen Clipping-Ebene nicht unbegrenzt ist.

2.9.5 Projektionstransformation

Die dritte Transformation, die angewandt wird, ist die Projektion. Sie fügt den 3D-Vektoren eine vierte Koordinate hinzu und sorgt letztendlich (im Fall der perspektivischen Transformation) dafür, dass weiter entfernte Objekte später auf dem Bildschirm kleiner erscheinen. Die Division durch die vierte Koordinate findet aber noch nicht statt, sondern erst später.

2.9.6 Back-Face-Culling und Clipping

Als Nächstes folgen die ersten Optimierungsschritte. Wenn das *Back-Face-Culling* aktiviert ist, verwirft die Rendering-Pipeline alle Dreiecke, die dem Beobachter abgewandt sind, deren Normalenvektor also vom Beobachter weg zeigt. Sie werden sich vielleicht fragen, welchen Sinn das hat. Normalerweise schwebt ein Dreieck nicht einfach so ganz allein im Raum herum, sondern es ist Teil eines größeren Objekts aus hunderten oder tausenden von Dreiecken. Meistens bilden diese Dreiecke eine geschlossene Oberfläche. Die Dreiecke, die vom Back-Face-Culling verworfen werden, liegen dann auf den Rückseiten dieser Oberflächen und sind von vorne nicht sichtbar, da sie von Dreiecken auf der Vorderseite verdeckt werden. Sie können sich das leicht anhand eines Würfels klar machen. Dort sind immer nur höchstens drei der sechs Seiten sichtbar. Durch das Back-Face-Culling fallen ungefähr 50% aller Dreiecke weg und müssen nicht weiter bearbeitet werden. Wenn ein Objekt allerdings keine geschlossene Oberfläche aufweist oder transparent ist, dann führt das Back-Face-Culling zu unerwünschten Ergebnissen.

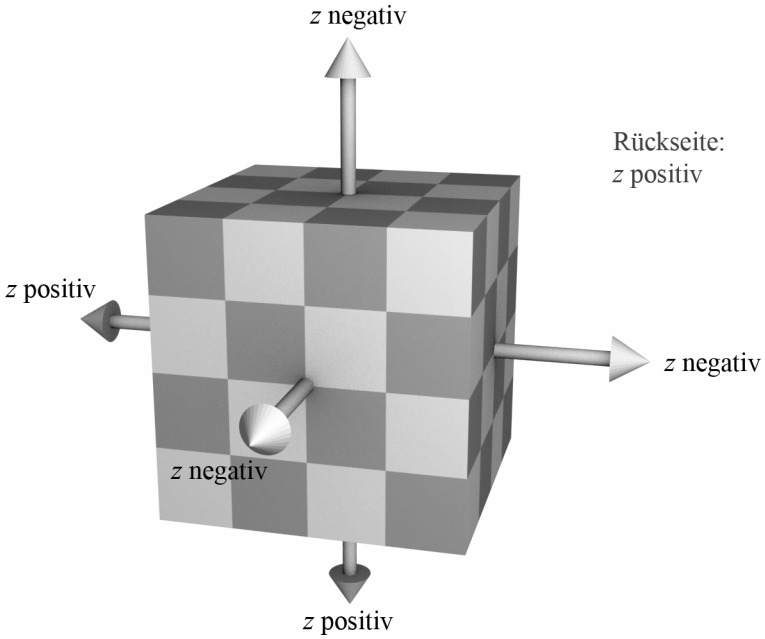


Abbildung 2.51 Ein Würfel mit den Normalenvektoren seiner sechs Seiten. Bei geschlossenen Objekten wie diesem sind die Rückseiten nie sichtbar. Für die Normalenvektoren der Rückseiten gilt, dass sie vom Betrachter weg zeigen (die *z*-Komponente ist positiv).

Neben dem Back-Face-Culling findet an dieser Stelle der Pipeline auch das Clipping statt. Alle Primitive werden an den sechs Ebenen des View-Frustums abgeschnitten. Wenn ein Dreieck komplett außerhalb liegt, wird es komplett verworfen. Durch das Clipping fallen weitere Primitive weg, aber es können durch das Abschneiden auch neue entstehen. Da die Vertizes zu diesem Zeitpunkt durch die Projektionsmatrix transformiert wurden und vierdimensionale Koordinaten haben, ist das Clipping relativ einfach zu berechnen. Ein Punkt liegt nämlich genau dann innerhalb des View-Frustums, wenn er folgende Bedingung erfüllt:

$$-w \leq x \leq w \wedge$$
$$-w \leq y \leq w \wedge$$
$$0 \leq z \leq w$$

2.9.7 Die Division durch *w*

Erst jetzt, nach dem Clipping, erfolgt die Division durch die *w*-Koordinate. Anschließend wird die *w*-Koordinate, die ja nun 1 ist, weggelassen. So werden aus den vierdimensionalen Vektoren wieder dreidimensionale Vektoren. Nur durch diese Division kann die Projektionsmatrix dafür sorgen, dass weiter entfernte Objekte kleiner erscheinen. Die Koordinaten befinden sich nach der Division im 3D-Bildkoordinatensystem, das heißt sie bezie-

hen sich auf den Würfel, zu dem der View-Frustum verzerrt wurde. Die Koordinaten sind zu diesem Zeitpunkt noch unabhängig von der verwendeten Auflösung in Pixel.

2.9.8 Viewport-Transformation

Nun ist es an der Zeit, die 3D-Bildkoordinaten in 2D-Pixelkoordinaten umzurechnen. Dies geschieht bei der *Viewport-Transformation*. Den Viewport kann man sich als Rechteck vorstellen, das den Zeichenbereich auf der Zieloberfläche festlegt. Die 3D-Bildkoordinaten werden auf dieses Rechteck abgebildet. Stellen Sie sich beispielsweise ein Rennspiel vor, bei dem vier Spieler gegeneinander antreten können. Jeder Spieler bekommt ein Viertel des Bildschirms zugewiesen (Split-Screen). Das kann man erreichen, indem man für jede der vier Ansichten einen anderen Viewport verwendet. Bei einer Bildschirmauflösung von 1280×1024 Pixel wäre der Viewport für die rechte obere Ansicht beispielsweise durch das Rechteck (640, 0, 640, 512) definiert. (640, 0) ist dabei die Koordinate der linken oberen Ecke des Rechtecks, und (640, 512) ist seine Größe.

Eine Besonderheit bei dieser Transformation ist, dass die *y*-Achse umgekehrt wird. Im 3D-Bildkoordinatensystem zeigt die *y*-Achse nach oben, aber im 2D-Pixelkoordinatensystem zeigt sie nach unten. Die 2D-Pixelkoordinaten (0, 0) liegen in der linken oberen Ecke des Bilds. 2D-Pixelkoordinaten müssen übrigens nicht ganzzahlig sein.

Die Viewport-Transformation funktioniert wie folgt:

$$x_{\text{Pixel}} = X + W \cdot \frac{x_{\text{Bild}} + 1}{2}$$

$$y_{\text{Pixel}} = Y + H \cdot \frac{y_{\text{Bild}} + 1}{2}$$

Hierbei geben *W* und *H* die Breite beziehungsweise die Höhe sowie (*X*, *Y*) die Koordinaten der linken oberen Ecke des Viewport-Rechtecks an, jeweils in Pixel.

2.9.9 Rasterung

Nach der Viewport-Transformation liegen die Vertizes der zu rendernden Primitiven in Pixelkoordinaten vor. Der eigentliche Zeichenvorgang kann nun also endlich stattfinden. Es gibt aber noch ein Problem: die Pixelkoordinaten sind „unendlich genau“[8], aber wir haben nur endlich viele Pixel zur Verfügung, um diese darauf abzubilden. Diese Abbildung bezeichnet man auch als Rasterung: Man bestimmt für jede Primitive, durch welche Pixel sie läuft und füllt diese entsprechend mit Farbe. Man kann dies zum Beispiel tun, indem man den Mittelpunkt der Pixel betrachtet. Wenn eine Primitive den Mittelpunkt eines Pixels überdeckt, dann gehört dieser Pixel zu ihr.

[8] natürlich nur so genau, wie die Fließkommadarstellung im Computer es zulässt

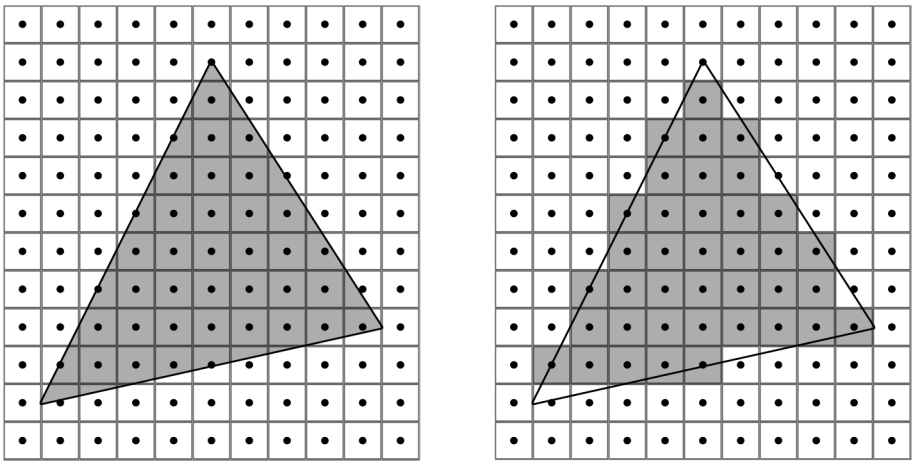


Abbildung 2.52 Rasterung eines Dreiecks auf die einfache Art: Jeder Pixel, dessen Mittelpunkt im Dreieck liegt, wird ausgefüllt.

Diese Methode produziert keine sehr ansprechenden Resultate, aber sie ist die schnellste und die einfachste. Schiefe Kanten sind sehr „pixelig“. Mit Anti-Aliasing können bessere Ergebnisse erzielt werden. Doch mehr dazu später.

2.9.9.1 Sichtbarkeitstest / Z-Buffering

Das so genannte *Sichtbarkeitsproblem* besteht darin herauszufinden, welche Teile einer Szene nach der Projektion tatsächlich sichtbar sind, also nicht von anderen Teilen verdeckt werden.

Zeichnet man eine Reihe von Primitiven, dann wird die zuletzt gezeichnete immer die vorhergehenden verdecken, so wie ein Maler auf einer Leinwand den alten Bildinhalt überdeckt, wenn er etwas Neues darüber malt. Das ist jedoch nicht das, was man sich wünschen würde, denn es sollte nicht von der Reihenfolge des Zeichnens abhängen, welche Primitive vorne steht, sondern von der Tiefe. Weiter vorne stehende Objekte sollen die weiter hinten liegenden Objekte verdecken.

Natürlich kann man die Primitiven vor dem Rendern nach ihrer Tiefe sortieren und sich von der am weitesten hinten liegenden bis zur vordersten Primitive durcharbeiten. Dies bezeichnet man als den *Maleralgorithmus*, denn ein Maler malt normalerweise zuerst den Hintergrund, dann den Mittelgrund und schließlich den Vordergrund. Aber es gibt Situationen, in denen man keine Sortierung finden kann, die ein korrektes Ergebnis liefert.

Abbildung 2.53 Egal, in welcher Reihenfolge man die vier Balken zeichnet: dieses Muster wird man mit dem Maleralgorithmus nicht darstellen können, ohne die Balken zu zerteilen.

In diesen Fällen ist das Zerteilen der Dreiecke zwar eine Lösung, aber dies für eine Szene mit Hunderttausenden oder Millionen von Dreiecken zu tun, wäre viel zu aufwändig.

Der Z-Buffer als Lösung des Sichtbarkeitsproblems

Eine bessere Lösung des Sichtbarkeitsproblems ist der Einsatz eines *Tiefenpuffers* (*Z-Buffer*). Der Z-Buffer hat dieselben Ausmaße wie das Bild (zum Beispiel 1280×1024 Pixel) und speichert für jeden Pixel einen Tiefenwert. Zu Beginn wird der Z-Buffer mit der maximalen Tiefe initialisiert. Wird nun eine Primitive gerendert, dann vergleicht man für jeden zu zeichnenden Pixel im so genannten *Z-Test* dessen Tiefe mit der im Z-Buffer gespeicherten Tiefe an dieser Stelle. Nur wenn die Tiefe des zu zeichnenden Pixels kleiner (oder gleich) der Tiefe im Z-Buffer ist, wird der Pixel gezeichnet. Die Tiefe des neuen Pixels wird dann in den Z-Buffer geschrieben und ersetzt den alten Wert. Wenn die Tiefe des Pixels jedoch größer als die im Z-Buffer gespeicherte Tiefe ist, dann bedeutet das, dass bereits ein anderer Pixel an dieser Stelle gezeichnet wurde, das sich näher am Betrachter befindet. Der Pixel wird folglich verworfen, da er von ihm verdeckt wird.

Der Z-Buffer-Algorithmus ist die Standardlösung für das Sichtbarkeitsproblem und wird von der Grafikkarte hardwareseitig ausgeführt. Doch auch der Z-Buffer arbeitet nicht perfekt, da die gespeicherten Tiefenwerte natürlich nur eine begrenzte Genauigkeit haben können. Früher waren 16-Bit-Z-Buffer üblich. Das bedeutet, dass pro Pixel 16 Bits Tiefeninformation gespeichert werden. Es gibt dann nur $2^{16} = 65536$ verschiedene Tiefenwerte. Es kann also leicht vorkommen, dass zwei verschiedene Tiefenwerte, zum Beispiel 1000 und 1000.01 auf denselben Wert im Z-Buffer abgebildet werden. Wenn das geschieht, hängt es doch wieder von der Reihenfolge des Zeichnens ab, welcher Pixel schließlich sichtbar ist und welcher verdeckt wird. Heute verwendet man meist 24- oder 32-Bit-Z-Buffer. Dadurch wird dieses Problem reduziert oder sogar ganz beseitigt.

Vielleicht fragen Sie sich, was eigentlich genau die Tiefenwerte sind, die da berechnet, verglichen und im Z-Buffer gespeichert werden. Dabei handelt es sich um die *z*-Komponenten der 3D-Bildkoordinaten (also nach der Projektion und der Division durch *w*), woraus sich auch die Bezeichnung „Z-Buffer“ ergibt. Die Tiefe kann einen Wert zwischen 0 und 1 annehmen, wobei 0 für die nahe und 1 für die ferne Clipping-Ebene steht. Auf Grund der Division durch *w* sind die Tiefenwerte jedoch nicht linear verteilt. Im Bereich um die nahe Clipping-Ebene ist die Genauigkeit wesentlich höher als weiter hinten bei der fernen Clipping-Ebene. Besonders dann, wenn man Z-Buffer mit niedriger Bittiefe einsetzt, kann das schnell wieder zu den oben genannten Problemen führen. Als Faustregel gilt deshalb, dass man die nahe Clipping-Ebene so weit wie möglich nach hinten und die ferne Clipping-Ebene so weit wie möglich nach vorne schieben sollte, um die verfügbare Genauigkeit möglichst gut zu verteilen.

2.9.9.2 Texturierung

Wenn ein Pixel den Z-Test bestanden hat und damit zumindest vorerst sichtbar ist (er könnte ja immer noch von anschließend gerenderten Primitiven verdeckt werden), kommen Texturen ins Spiel. Texturen sind normalerweise zweidimensionale Bilder, die über die Primitiven „gespannt“ werden. Man verwendet sie, um Oberflächen detaillierter erscheinen zu lassen als sie es tatsächlich sind. Möchte man zum Beispiel einen Baumstamm modellieren, dann könnte man theoretisch die feinen Strukturen der Baumrinde mit sehr vielen kleinen Dreiecken nachbilden. Alternativ modelliert man den Baumstamm nur sehr grob, nämlich als angenäherten Zylinder, und weist ihm anschließend eine Textur zu, die das Muster einer Baumrinde enthält. Das ist die übliche Vorgehensweise. Texturen kann man allerdings noch für sehr viel mehr Dinge gebrauchen, wie wir später noch sehen werden.

Die Bildpunkte einer Textur bezeichnet man übrigens nicht als Pixel, sondern als *Texel* (*Texture Element*). Typische Texturgrößen sind 256×256, 512×512 oder 1024x1024. Hier sind Breite und Höhe jeweils Zweierpotenzen. Insbesondere ältere Grafikkarten haben Probleme mit Texturen, bei denen das nicht der Fall ist, und noch ältere Karten lehnen sogar alle nichtquadratischen Texturen ab. Heute ist das jedoch kein Problem mehr, aber trotzdem werden solche Texturgrößen immer noch gern genutzt.

Texturkoordinaten

Nun muss man irgendwie festlegen können, auf welche Weise eine Textur beispielsweise über ein Dreieck gelegt werden soll. Hierbei gibt es unendlich viele Möglichkeiten, die Textur zu vergrößern, zu verkleinern, zu drehen, zu verschieben, zu spiegeln oder auf sonstige Weise zu verzerren. Also führt man so genannte *Texturkoordinaten* ein. Texturkoordinaten sind so wie Farbe oder Normalenvektor ein Bestandteil eines Vertex. Sie ordnen jedem Vertex eine Position auf der Textur zu. Beim Rendern werden die Texturkoordinaten ähnlich wie die Farbe zwischen den Vertizes über die Dreiecksoberfläche interpo-

liert. Für jeden Pixel des Dreiecks ist damit bekannt, welchem Punkt auf der Textur er entspricht.

Normalerweise liegen die Texturkoordinaten im Bereich zwischen (0, 0) und (1, 1). In Direct3D entspricht (0, 0) der linken oberen Ecke der Textur und (1, 1) der rechten unteren Ecke. Dadurch, dass man nur mit diesem festen Wertebereich arbeitet, muss man die tatsächliche Größe einer Textur, gemessen in Texeln, bei der Generierung der Texturkoordinaten gar nicht kennen, und Texturen können sehr leicht gegen größere oder kleinere ausgetauscht werden.

Wenn jedoch die tatsächlichen Koordinaten in Texeln gemeint sind, dann wird von *Texelkoordinaten* die Rede sein. So entsprechen beispielsweise die Texturkoordinaten (0.5, 0.25) bei einer Textur der Größe 1024×1024 den Texelkoordinaten (512, 256).

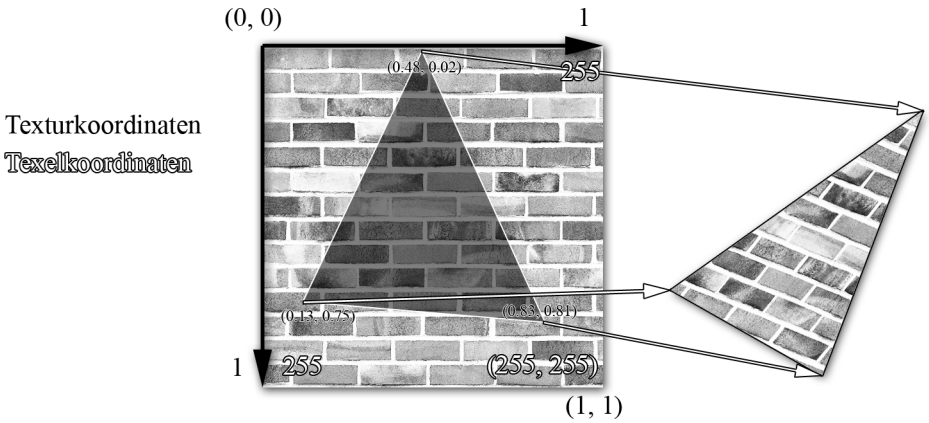


Abbildung 2.54 Eine Textur der Größe 256×256 (*links*) wird auf ein Dreieck abgebildet (*rechts*). Dazu erhält jeder Eckpunkt Texturkoordinaten, die festlegen, welcher Punkt auf der Textur auf ihn abgebildet werden soll.

Texturfilterung

Bei der Texturierung ergibt sich folgendes Problem: Die zwischen den Vertizes interpolierten Texelkoordinaten fallen im Allgemeinen in den wenigsten Fällen exakt auf einen bestimmten Texel. Was sollte die Grafikkarte beispielsweise tun, wenn die Texelkoordinaten zwischen vier Texeln liegt? Soll sie nun einen davon auswählen (welchen?) oder irgendetwas Anderes tun?

Den Wert in einer Textur an einer bestimmten Stelle herauszufinden bezeichnet man auch als *Sampling*, und das, was man herausbekommt, als *Sample*.[9] Eingedeutscht kann man

[9] Der Begriff *Sampling* stammt eigentlich aus der Signalverarbeitung. Eine Textur kann als diskretes (da aus einzelnen diskreten Werten bestehend, nämlich den Texeln) zweidimensionales Signal aufgefasst werden.

auch davon sprechen, „eine Textur zu sampeln“, was irgendwie besser klingt als die eigentlich korrekte Übersetzung „eine Textur abzutasten“.

Wie sampelt die Grafikkarte nun also eine Textur in solch einem Fall, wo die Texturkoordinaten nicht exakt auf einem bestimmten Texel liegen? Das kann der Programmierer entscheiden, indem er einen *Texturfilter* wählt.

Der einfachste Filter ist der *Nearest-Filter*, der – wie der Name schon vermuten lässt – einfach den am nahsten gelegenen Texel heraussucht und dessen Farbe zurückliefert. Dieser Filter führt jedoch zu unschönen „pixeligen“ Texturen, wie man sie vor allem aus der Anfangszeit der 3D-Spiele kennt, wo es noch keine Hardwarebeschleunigung durch Grafikkarten gab.

Anstatt einfach den nächsten Texel zu nehmen, kann man sich auch ein wenig mehr Arbeit machen und zwischen den vier benachbarten Texeln, die in Frage kommen, linear interpolieren. Als Ergebnis erhält man weiche Farbverläufe zwischen den Texeln. Dies entspricht einem *bilinearen Filter*. Dieser Filter wird am häufigsten eingesetzt, und auf heutigen Grafikkarten bekommt man ihn quasi geschenkt, da er nicht langsamer als der Nearest-Filter ist.

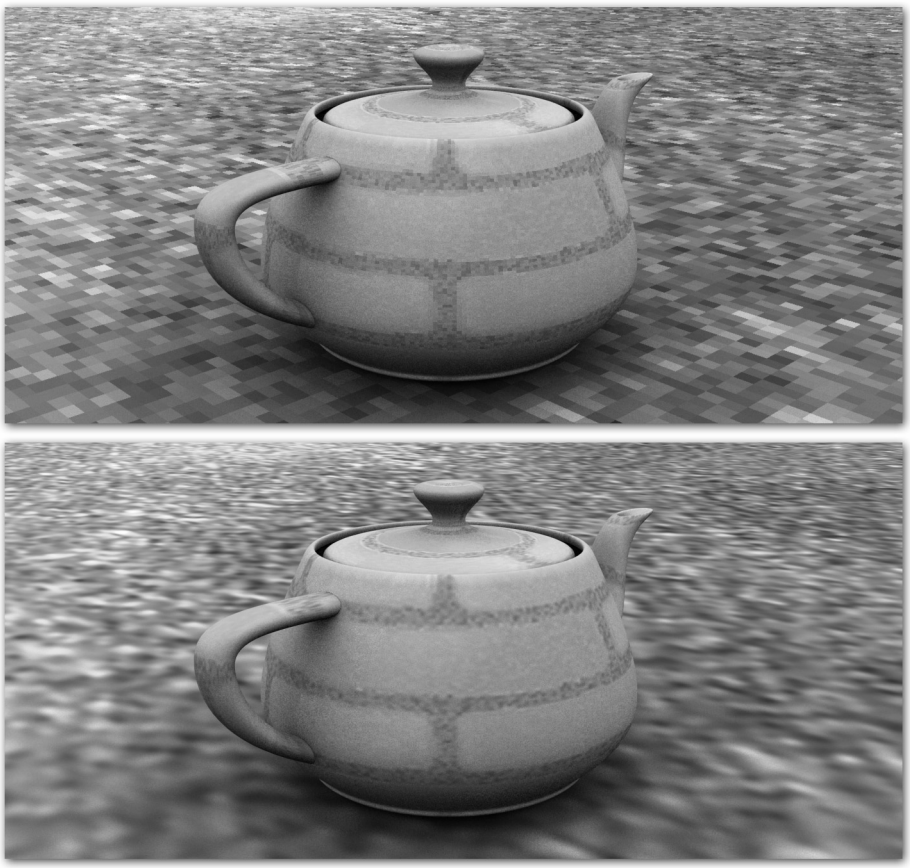


Abbildung 2.55 *Oben*: Beim Nearest-Filter wird immer der am nächsten liegende Texel verwendet. Dadurch entsteht ein „Klötzcheneffekt", der besonders beim Boden gut erkennbar ist. *Unten*: Mit einem interpolierenden Filter fließen auch die benachbarten Texel in die Pixelfarbe ein, wodurch die Farben weich ineinander übergehen.

Da die lineare Interpolation sehr wichtig ist und auch in anderen Bereichen als der Texturierung häufig benötigt wird, möchte ich noch kurz darauf eingehen, wie der bilineare Filter funktioniert.

Sei $T_{u,v}$ die Farbe des Texels mit den Texelkoordinaten (u, v). In unserem Beispiel wollen wir die Farbe an den Texelkoordinaten (37.4, 20.8) herausfinden. Dieser Punkt liegt offensichtlich zwischen den vier Texeln $T_{37,\,20}$ (links oben), $T_{38,\,20}$ (rechts oben), $T_{37,\,21}$ (links unten) und $T_{38,\,21}$ (rechts unten). Also müssen wir zwischen diesen vier Farben interpolieren.

Die bilineare Interpolation funktioniert nun wie folgt: zuerst interpoliert man jeweils getrennt zwischen $T_{37,\,20}$ und $T_{38,\,20}$ (oben) sowie zwischen $T_{37,\,21}$ und $T_{38,\,21}$ (unten). Damit erhalten wir die Zwischenwerte $T_{37.4,\,20}$ (oben) und $T_{37.4,\,21}$ (unten). Nun interpolieren wir noch vertikal zwischen diesen beiden Werten und erhalten schließlich unsere gesuchte Farbe $T_{37.4,\,20.8}$. Wir interpolieren also zunächst zweimal horizontal und dann noch einmal

vertikal. Bei der horizontalen Interpolation ist der Interpolationsfaktor $\frac{37.4-37}{38-37} = 0.4$. Entsprechend ist der vertikale Interpolationsfaktor 0.8. Die Zwischenwerte und das Endergebnis lassen sich dann also wie folgt berechnen:

$$T_{37.4,\,20} = T_{37,\,20} + 0.4 \cdot \left(T_{38,\,20} - T_{37,\,20}\right)$$
$$T_{37.4,\,21} = T_{37,\,21} + 0.4 \cdot \left(T_{38,\,21} - T_{37,\,21}\right)$$
$$T_{37.4,\,20.8} = T_{37.4,\,20} + 0.8 \cdot \left(T_{37.4,\,21} - T_{37.4,\,20}\right)$$

Man kann übrigens auch zuerst vertikal interpolieren und dann horizontal, das macht keinen Unterschied.

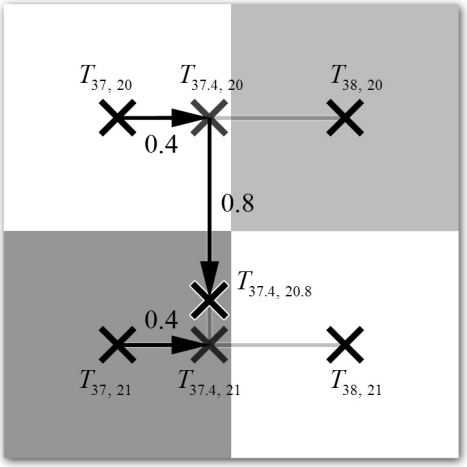


Abbildung 2.56 Bilineare Interpolation zwischen vier benachbarten Texeln: oben und unten wird jeweils horizontal interpoliert, und zwischen den interpolierten Werten wird vertikal erneut interpoliert.

MIP-Mapping

Wenn eine Textur auf dem Bildschirm wesentlich kleiner gerendert wird als ihre Originalgröße, zum Beispiel weil das Objekt sehr weit weg ist, dann kommt es je nach Inhalt der Textur zu einem hässlichen Effekt namens *Aliasing*. Besonders dann, wenn die Textur starke hochfrequente Anteile enthält (sich schnell ändernde Farbintensitäten und Kontraste), wird das zu einem Problem. Das klingt noch alles sehr theoretisch, darum betrachten wir ein konkretes Beispiel.

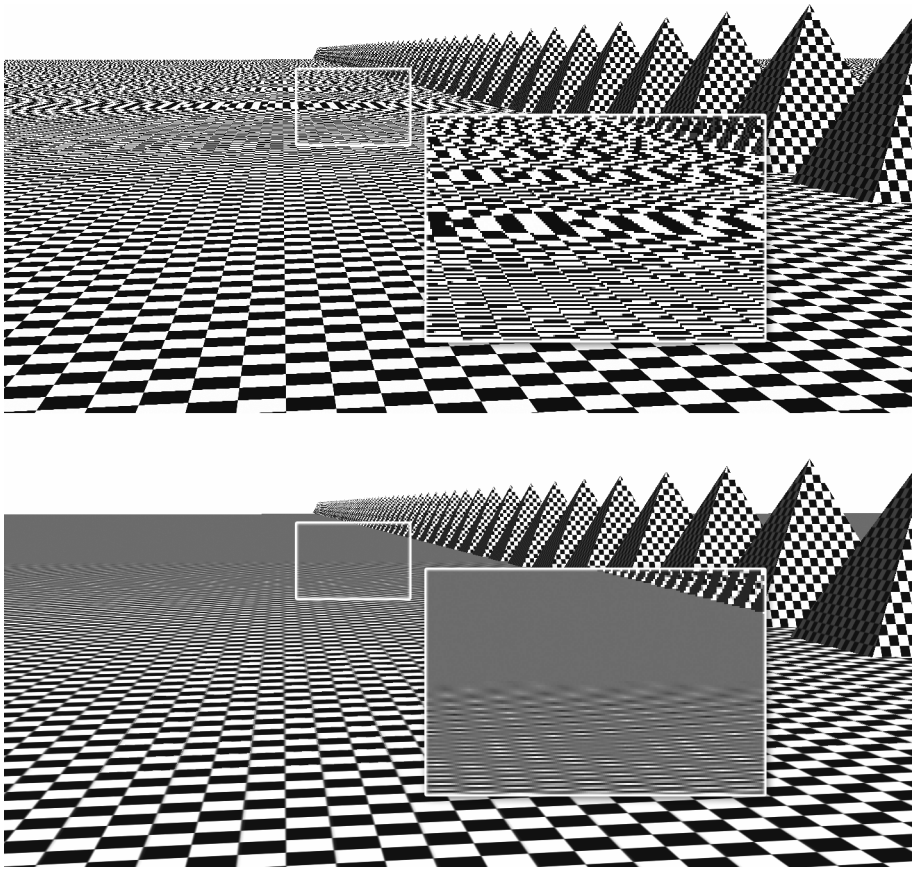


Abbildung 2.57 *Oben*: Ohne MIP-Mapping treten ab einer gewissen Entfernung Aliasing-Effekte auf. *Unten*: Mit MIP-Mapping wird das Aliasing unterdrückt, was jedoch zu Lasten der Texturschärfe geht.

Wie Sie sehen, wird die Schachbretttextur in der oberen Bildhälfte nur bis zu einer gewissen Entfernung korrekt dargestellt. Danach verwandelt sich der Boden in einen undefinierbaren Brei aus schwarzen und weißen Pixeln mit seltsamen Mustern, die nicht mehr dem des Schachbretts entsprechen. Würde man die Kamera auch nur ein kleines Stückchen bewegen, würde sich das Muster komplett ändern. Genau das ist Aliasing. Wie kann das passieren? Im nahen Bereich, wo noch alles in Ordnung ist, deckt ein Pixel auf dem Bildschirm höchstens einen Texel auf der Textur ab. Aber weiter hinten liegen mehrere Texel im Bereich eines Pixels, aber die Textur wird nur an einer einzigen Stelle gesampelt. Wollte man korrekt vorgehen, müsste man den Mittelwert aus allen Texeln bestimmen, die in dem entsprechenden Bereich liegen, aber das ist sehr aufwändig.

Das *MIP-Mapping* (in der unteren Bildhälfte angewandt) bietet hier Abhilfe. Man erzeugt dazu von einer Textur mehrere verkleinerte Varianten (*MIP-Levels* genannt). Jeder MIP-Level ist in Höhe und Breite halb so groß wie der Vorherige. Entweder hört man bei einer bestimmten Größe oder Anzahl von MIP-Levels auf, oder man rechnet die Textur bis auf

die Größe 1×1 Texel herunter. Beim Herunterrechnen kann man Filter von höherer Qualität verwenden als das in Echtzeit beim Rendern möglich wäre. Eine simple Methode ist beispielsweise, immer Blöcke von 2×2 Texeln zu betrachten und aus dem Mittelwert den Texel für den nächst kleineren MIP-Level zu erzeugen. MIP steht übrigens für *multum in parvo*, was *viel auf kleinem Platz* bedeutet.

Abbildung 2.58 Eine (coole) Textur mit allen MIP-Levels bis 1×1: die Farbe des einzelnen Pixels, aus dem der kleinste MIP-Level besteht, entspricht dem Mittelwert aller Pixel des Originalbilds.

Beim Rendern wird nun für jeden Pixel der passende MIP-Level ermittelt, so dass nicht mehrere Texel auf denselben Pixel fallen und kein Aliasing auftritt. Dadurch werden die Texturen zwar etwas unscharf, aber das ist immer noch besser als das quasi-zufällige Rauschen, das ohne MIP-Mapping entstehen würde. Außerdem lässt sich die Wahl des MIP-Levels meist noch durch einen Parameter leicht beeinflussen, so dass die Umschaltung auf den nächst kleineren MIP-Level ein wenig später erfolgt als normalerweise. Diese Umschaltung zwischen zwei MIP-Levels ist meistens deutlich erkennbar. Abhilfe verschafft die so genannte *trilineare Filterung*. Hier wird vor der normalen bilinearen Interpolation noch zwischen zwei benachbarten MIP-Levels interpoliert. Dadurch gehen die MIP-Levels weich ineinander über statt abrupt.

Wie man sich leicht vorstellen kann, benötigt eine Textur mit einer vollständigen Reihe von MIP-Levels knapp 50% mehr Speicherplatz als ohne MIP-Levels. Die Performance hingegen kann von MIP-Mapping profitieren, weil der Cache der Grafikkarte wirkungsvoller arbeitet, wenn die Texel aus der Textur nur in kleinen Abständen gelesen werden. Ohne MIP-Mapping sind die Speicherzugriffe mehr oder weniger chaotisch.

Anisotropische Filterung

In manchen Fällen produziert selbst das MIP-Mapping noch eher unansehnliche Ergebnisse, nämlich dann, wenn der Beobachter in einem sehr flachen Winkel auf eine texturierte Fläche schaut, zum Beispiel auf den Boden. Hier wird die Textur in x-Richtung nur wenig gestaucht, aber in y-Richtung in großen Entfernungen sehr stark. Ginge es nur nach der x-Richtung, so würde ein relativ großer MIP-Level gewählt, aber die extreme Stauchung in

y-Richtung führt dazu, dass ein sehr kleiner MIP-Level verwendet wird. Die Textur erscheint daher sehr verwaschen und unscharf.

Der so genannte *anisotropische Filter* versucht dies zu kompensieren, indem der in diesem Fall in *y*-Richtung mehrere Samples nimmt und sie mittelt. Wie viele Samples das sein können, lässt sich einstellen. Der Filter wirkt sich natürlich negativ auf die Performance aus, weshalb man sich seinen Einsatz gut überlegen sollte.

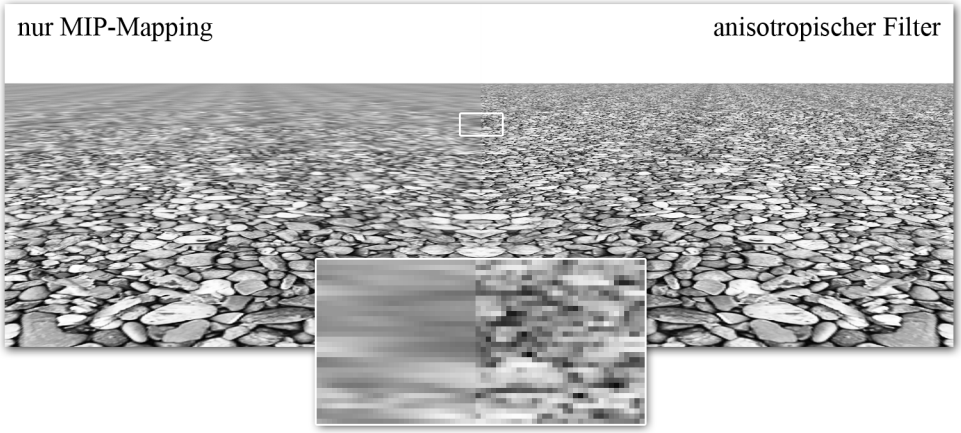


Abbildung 2.59 Mit purem MIP-Mapping (*links*) wird die Bodentextur sehr schnell unscharf. Der anisotropische Filter (*rechts*) verhindert das bis zu einem gewissen Grad. In der Vergrößerung ist der Unterschied deutlich erkennbar.

2.9.9.3 Per-Pixel-Beleuchtung und -Nebel

Die meisten heutigen Spiele berechnen die Beleuchtung und eventuelle Nebeleffekte auf Pixelbasis (Phong-Shading). Das funktioniert mit Pixel-Shadern. Pixel-Shader sind kleine Programme, die auf der Grafikkarte ausgeführt werden, und zwar für jeden gerenderten Pixel. Der Shader erhält eine Reihe von Eingaben wie Position, Normalenvektor, Farbe und Texturkoordinaten, und berechnet daraus die Farbe des Pixels, beispielsweise durch Anwendung des Phong-Beleuchtungsmodells. Mit Shadern lassen sich nahezu beliebig komplexe Effekte realisieren. Die Per-Pixel-Beleuchtung ist nur einer davon.

2.9.9.4 Anti-Aliasing

Unter *Anti-Aliasing* versteht man die Bekämpfung von Aliasing-Artefakten, vor allem an den Kanten der gerenderten Primitiven. Zeichnet man zum Beispiel ein Dreieck mit einer fast vertikalen Kante, dann macht diese Kante irgendwo einen abrupten Sprung von der einen Pixelspalte in die Nächste. Ein weiterer Effekt, der sich in Spielen oft beobachten lässt, wenn das Anti-Aliasing deaktiviert ist, betrifft sehr dünne Objekte wie Stromkabel oder Antennen. Unter bestimmten Umständen sind diese plötzlich gar nicht mehr sichtbar,

wenn ihre Dicke auf dem Bildschirm kleiner als ein Pixel wird. Bewegt man dann die Kamera, erscheinen sie zeitweise wieder und flimmern unansehnlich.

Supersampling

Die wohl einfachste Methode zur Verringerung von Aliasing-Artefakten ist das *Supersampling* (*Überabtastung*). Das Bild wird hierbei beispielsweise in doppelter Auflösung gerendert (1600×1200 statt 800×600) und danach wieder mit einem Filter verkleinert, zum Beispiel durch Mittelwertbildung über jeweils 2×2 Pixel. Dadurch werden vormals scharfe Kanten weichgezeichnet. Das Bild wirkt nun sehr viel ruhiger und ist angenehmer zu betrachten.

Leider ist Supersampling sehr teuer, wie man sich leicht denken kann. Es lastet vor allem die Speicherbandbreite aus, da nun ein Vielfaches der Pixel geschrieben und aus den Texturen gelesen werden müssen.

Multisampling

Ein intelligenteres Verfahren ist das so genannte *Multisampling*. Dieses Verfahren wird heute von allen Grafikkarten unterstützt. Es filtert die Primitiven ausschließlich an den Kanten, da die Aliasing-Artefakte dort meistens am ehesten auffallen. Für jeden Pixel wird grundsätzlich nur ein einziger Farbwert berechnet, im Gegensatz zum Supersampling. Außerdem wird für mehrere verschiedene Stellen innerhalb des Pixels getestet, ob sie noch innerhalb der Primitive liegen (es werden Samples genommen). Den Grad des Multisamplings gibt man durch die Anzahl der genommenen Samples an, beim 4×-Multisampling sind es beispielsweise 4 Stück. Liegen 3 von 4 Samples innerhalb der Primitive, dann wird der Pixel zu 75% in das Endergebnis eingehen.

Die Samples innerhalb eines Pixels können in verschiedenen Mustern angeordnet sein. Als sehr gut hat sich hierbei das *Rotated Grid*-Muster herausgestellt. Die Samples decken hierbei das Bild gitterförmig ab, wobei das Gitter aber nicht an den Bildschirmachsen ausgerichtet ist, sondern leicht gedreht.

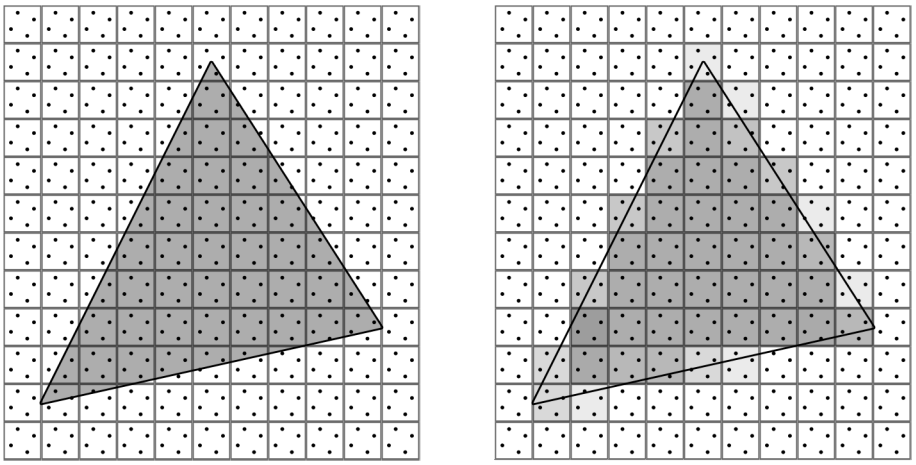


Abbildung 2.60 **4×**-Multisampling mit einem Rotated Grid-Muster und vier Samples pro Pixel: die Kanten sind jetzt geglättet. Anhand der Anzahl der Samples, die innerhalb der Primitive liegen, wird die Intensität jedes Pixels berechnet.

Je mehr Samples genommen werden, desto höher ist die Qualität des Anti-Aliasings. Allerdings steigt auch der Speicherverbrauch und die Speicherauslastung, denn für jede Sample-Position muss ein eigener Bild- und Tiefenpuffer angelegt werden. Dadurch, dass pro Pixel immer nur ein Farbwert berechnet wird, hält sich der zusätzliche Rechenaufwand jedoch in Grenzen. Es hängt also von der Situation ab, wie stark das Multisampling die Performance beeinträchtigt.

Anti-Aliasing ist ein sehr komplexes Thema. Einen sehr guten Überblick über die eingesetzten Verfahren gibt es bei *3DCenter*[10].

[10] http://www.3dcenter.de/artikel/multisampling_anti-aliasing/

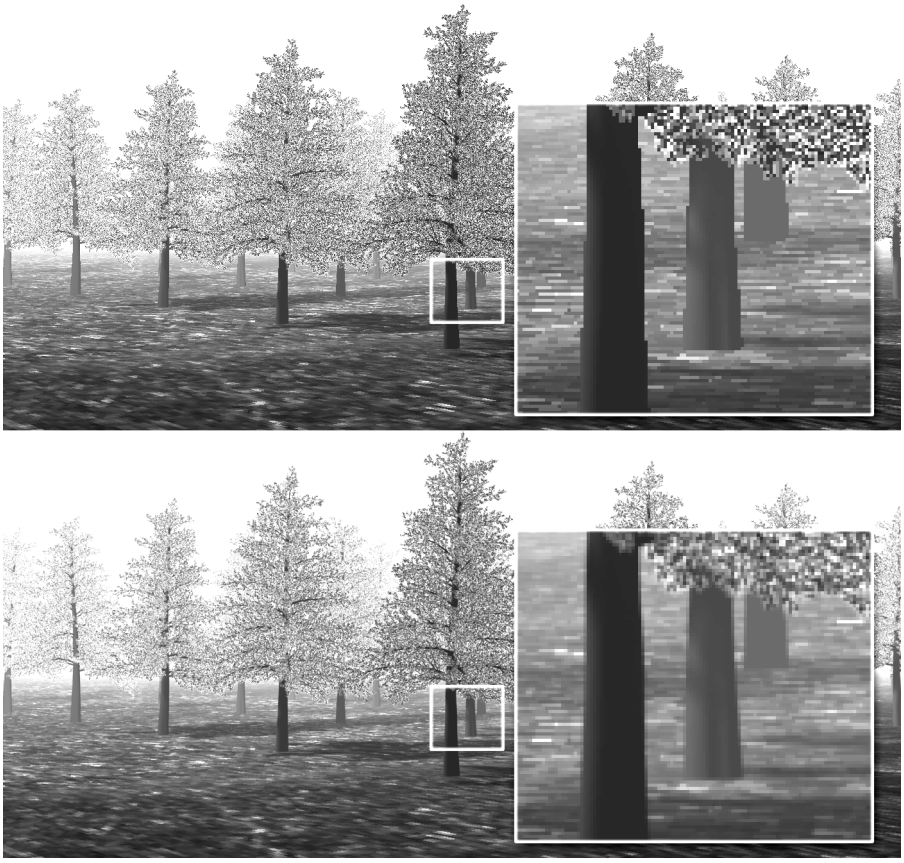


Abbildung 2.61 *Oben*: Ohne Anti-Aliasing ist an schrägen Kanten wie bei den Baumstämmen deutlich die Treppenbildung zu sehen. *Unten*: Mit Anti-Aliasing sind die Kanten nicht mehr „binär“, sondern erscheinen wesentlich weicher.

2.9.9.5 Blending

Bislang sind wir immer davon ausgegangen, dass beim Rendern ein neuer Pixel die bereits vorhandene Pixelfarbe überschreibt. Hätten wir beispielsweise zuerst ein rotes Quadrat gezeichnet und danach einen grünen Kreis, dann hätten die grünen Pixel die roten Pixel einfach überschrieben. Das muss aber nicht so sein.

Beim *Blending* werden die neue und die alte Pixelfarbe mit Hilfe einer ausgewählten Funktion kombiniert, und der alte Farbwert wird mit dem Ergebnis dieser Operation überschrieben. So können unter Anderem transparente Flächen wie Glas, aber auch Wasser und Spezialeffekte wie Feuer oder Rauch dargestellt werden.

Genauer gesagt werden der alte und der neue Farbwert jeweils komponentenweise mit Blendfaktoren (vierdimensionale Vektoren) multipliziert und anschließend miteinander

verknüpft. Die Anwendung wählt die Blendfaktoren und die Verknüpfungsfunktion, den Rest erledigt die Grafikkarte.

$$C_{\text{Ergebnis}} = F\left(C_{\text{Neu}} * B_{\text{Neu}},\ C_{\text{Alt}} * B_{\text{Alt}}\right)$$

Hier ist C_{Ergebnis} die Farbe des Pixels, der letztendlich gezeichnet wird. C_{Neu} und C_{Alt} sind die neue beziehungsweise alte Pixelfarbe. F ist die wählbare Verknüpfungsfunktion. B_{Neu} und B_{Alt} sind die ebenfalls wählbaren Blendfaktoren.

Als Verknüpfungsfunktion sind Addition und Subtraktion sowie Minimums- und Maximumsbildung üblich. Für die Blendfaktoren gibt es meist ebenfalls eine Reihe vordefinierter Möglichkeiten.

Alpha-Blending

Die wohl am häufigsten angewendete Art des Blendings ist das *Alpha-Blending*. Hierbei gibt die Alphakomponente der Farbe des neuen Pixels dessen *Opazität* an. Sie erinnern sich: Der Alphawert kann als vierte Komponente für RGB-Farben benutzt werden, um die Transparenz festzulegen. Opazität bedeutet übrigens das Gegenteil von Transparenz beziehungsweise Durchsichtigkeit: Eine Opazität von 100% bedeutet 0% Transparenz, eine Opazität von 0% bedeutet völlige Durchsichtigkeit.

Beim Alpha-Blending wählt man $B_{\text{Neu}} = (\alpha, \alpha, \alpha, \alpha), B_{\text{Alt}} = (1-\alpha, 1-\alpha, 1-\alpha, 1-\alpha)$, wobei α die Alphakomponente des neuen Pixels ist, und als Verknüpfungsfunktion die Addition. Der neue Pixel fließt also mit dem Gewicht α in das Ergebnis ein und der alte Pixel mit dem Gewicht $1 - \alpha$.

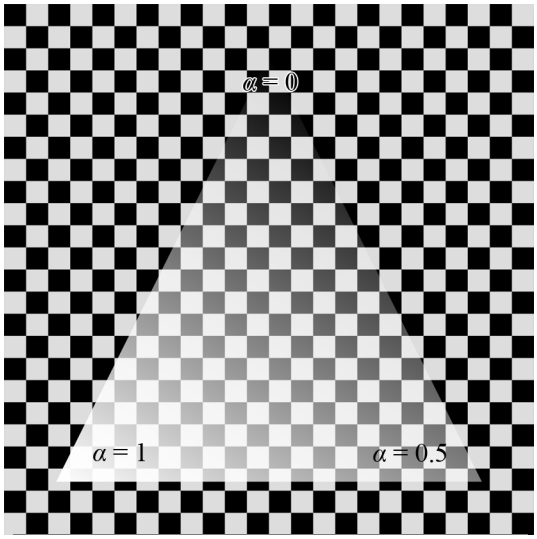


Abbildung 2.62 Alpha-Blending: Die Vertizes dieses Dreiecks haben verschiedene Alphawerte, die über das Dreieck interpoliert werden. Die obere Ecke ist komplett transparent, die linke Ecke ist komplett undurchsichtig, und die rechte Ecke ist halbtransparent.

Der Alphawert kann beispielsweise auch aus einer Textur stammen. Auf diese Weise kann man zum Beispiel einen Zaun sehr leicht rendern. Man benötigt nur eine Textur mit Alphakanal. In den Löchern des Zauns setzt man den Alphawert auf null, so dass die Textur an diesen Stellen transparent dargestellt wird.

Achtung, Z-Buffer!

Transparente Primitiven sollte man erst ganz am Ende rendern, also nach den undurchsichtigen Objekten. Weiterhin sollte das Schreiben in den Z-Buffer ausgeschaltet werden, denn transparente Objekte können sich nicht verdecken. Damit die transparenten Objekte auch untereinander in der richtigen Reihenfolge erscheinen, empfiehlt es sich, sie anhand ihrer Tiefe zu sortieren und sie von hinten nach vorne zu zeichnen.

2.9.10 Ausgabe / Darstellung

Nun sind wir bei der letzten Station der Rendering-Pipeline angelangt. Zum Schluss werden die gerenderten Pixel auf dem Bildschirm angezeigt oder auf eine andere Art weiterverarbeitet.

2.9.10.1 Darstellung auf dem Bildschirm und Double-Buffering

Die Grafikkarte verwendet einen Teil ihres Speichers für die Darstellung des Bilds auf dem Monitor. Diesen Speicherbereich nennt man auch *Front-Buffer*. Sein Inhalt wird entsprechend der eingestellten Bildwiederholfrequenz (zum Beispiel 100 Hz) über den Aus-

gang der Grafikkarte (VGA, DVI, S-Video, …) an den Monitor gesendet, und zwar zeilenweise von oben nach unten, denn genau so baut ein klassischer Kathodenstrahlmonitor (CRT) sein Bild auf.

Nach jeder Bildzeile gibt es eine kleine Pause, damit der Monitor seine Elektronenstrahlen zurückfahren und in die nächste Zeile lenken kann (*horizontaler Strahlrücklauf*). Nach der letzten Bildzeile ist eine größere Pause notwendig, da die Strahlen wieder in die linke obere Bildecke wandern müssen (*vertikaler Strahlrücklauf*). Natürlich gibt es bei LCDs keinen Elektronenstrahl, der das Bild aufbaut, aber aus Kompatibilitätsgründen ist das Timing hier dasselbe.

Fazit: Möchte man nun also ein Bild auf dem Monitor sichtbar machen, dann muss man es in den Front-Buffer schreiben.

„Unfertige Frames“

Rendert man ein *Frame* (also ein Bild) direkt in den Front-Buffer, dann kommt es vor, dass die Grafikkarte zu dem Zeitpunkt, wo das Bild an den Monitor gesendet wird, die Szene noch nicht komplett gezeichnet hat. Auf dem Bildschirm erscheint dann ein unfertiges Frame, was man natürlich vermeiden möchte.

Double-Buffering

Das *Double-Buffering* lost das gerade angesprochene Problem, indem ein zusätzlicher „unsichtbarer“ Bildpuffer eingeführt wird, in den das Frame zunächst gerendert und anschließend komplett in den Front-Buffer kopiert wird.[11] Diesen Puffer nennt man *Back-Buffer*. Mit Double-Buffering können keine unfertigen Frames mehr auf den Monitor gelangen.

Allerdings kann es zu einem ungewollten Effekt namens *Tearing* kommen. Auf der oberen Hälfte des Bildschirms ist dann noch das letzte Frame zu sehen, während er auf der unteren Hälfte das aktuelle Frame anzeigt. Das passiert, wenn das Kopieren der Bilddaten vom Back-Buffer in den Front-Buffer zur falschen Zeit passiert, nämlich während die Grafikkarte das Bild an den Monitor schickt.

Vertikale Strahlsynchronisation (V-Sync)

Um auch das Problem des Tearings zu lösen, bedarf es eines genauen Timings. Der Back-Buffer darf nur während des vertikalen Strahlrücklaufs in den Front-Buffer kopiert werden, also „zwischen zwei Bildern“. Dies nennt man *vertikale Strahlsynchronisation* oder kurz *V-Sync*. Durch diese Option wird die *Framerate*, also die Anzahl der gerenderten Frames pro Sekunde, auf die Bildwiederholrate des Monitors beschränkt. Darum wird V-Sync bei Performance-Benchmarks oder auch von „Hardcore-Zockern“ ausgeschaltet, um die größte mögliche Framerate zu erzielen.

[11] Alternativ können Front- und Back-Buffer auch einfach vertauscht werden (*Flipping*).

2.9.10.2 Rendern in eine Textur

Anstatt ein Bild in den Back-Buffer zu rendern, um es auf dem Bildschirm anzeigen zu lassen, kann man die Szene auch in eine Textur rendern und diese dann weiter verwenden. Das ist zum Beispiel beim bereits erwähnten Shadow-Mapping notwendig, wo die Szene aus der Sicht der Lichtquelle gerendert wird und die Tiefeninformationen in die Shadow-Map geschrieben werden.

Das Rendern in Texturen ist für viele Spezialeffekte unverzichtbar, und darum werden wir später auch von diesem Feature Gebrauch machen.

2.10 Ausblick

Wenn Sie bis hier hin durchgehalten haben, dann haben Sie den trockenen, theoretischen Teil nun überstanden. Da die 3D-Computergrafik ein sehr komplexes Thema ist, ist es meiner Meinung nach unbedingt notwendig, ein gewisses Maß an Vorkenntnissen mitzubringen, bevor man sich an eine API wie Direct3D heranwagt. Diese Kenntnisse sollte Ihnen dieses Kapitel vermittelt haben, so dass wir im nächsten Kapitel tatsächlich eigene 3D-Objekte auf den Bildschirm zaubern werden.

Zum Schluss möchte ich Ihnen noch einmal nahe legen, die Übungsaufgaben für dieses Kapitel zu bearbeiten, um sich selbst zu testen. Sie decken hauptsächlich die mathematischen Grundlagen ab. Die Musterlösungen finden Sie ganz hinten im Buch. Viel Erfolg!

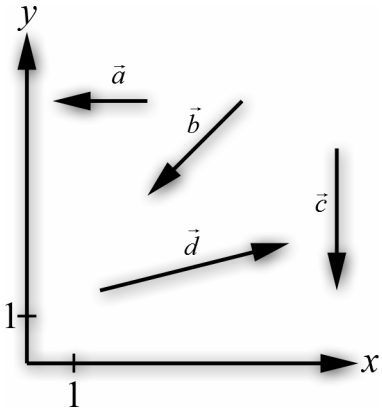
2.11 Übungsaufgaben

2.11.1 Vektoren

Aufgabe 1)

Ordnen Sie den Vektoren in der Abbildung die richtigen vier Koordinatendarstellungen zu:

$$\begin{pmatrix} \frac{\pi}{12} \\ -41 \end{pmatrix} \quad \begin{pmatrix} 4 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 4 \\ -1 \end{pmatrix} \quad \begin{pmatrix} -2 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 8 \\ 0 \\ -12 \end{pmatrix} \quad \begin{pmatrix} -2 \\ 2 \end{pmatrix} \quad \begin{pmatrix} 0 \\ -3 \end{pmatrix} \quad \begin{pmatrix} -2 \\ -2 \end{pmatrix} \quad \begin{pmatrix} \sqrt{\pi} \\ \pi \\ \pi^2 \end{pmatrix}$$



Aufgabe 2)

Berechnen Sie:

$$\left(2\cdot\begin{pmatrix}5\\2\\-4\end{pmatrix}+\begin{pmatrix}0\\1\\-4\end{pmatrix}-\frac{1}{2}\cdot\begin{pmatrix}8\\4\\2\end{pmatrix}\right)*\begin{pmatrix}2\\3\\1\end{pmatrix}$$

Aufgabe 3)

Berechnen Sie die Längen der folgenden Vektoren:

$$\vec{a}=\begin{pmatrix}1\\1\end{pmatrix}\quad \vec{b}=\begin{pmatrix}-1\\-1\end{pmatrix}\quad \vec{c}=\begin{pmatrix}5\\0\end{pmatrix}\quad \vec{d}=\begin{pmatrix}1\\4\\2\end{pmatrix}\quad \vec{e}=\begin{pmatrix}0\\0\\0\end{pmatrix}\quad \vec{f}=\begin{pmatrix}1\\1\\1\\1\end{pmatrix}$$

Aufgabe 4)

Berechnen Sie die Entfernungen zwischen den Punkten, deren Ortsvektoren unten angegeben sind. Welche zwei Punkte liegen am dichtesten zusammen, welche am weitesten auseinander?

$$\vec{A}=\begin{pmatrix}2\\4\end{pmatrix}\quad \vec{B}=\begin{pmatrix}0\\9\end{pmatrix}\quad \vec{C}=\begin{pmatrix}-4\\5\end{pmatrix}\quad \vec{D}=\begin{pmatrix}1\\8\end{pmatrix}$$

Aufgabe 5)

Gesucht ist eine Funktion $\vec{P}\left(\vec{A},\ \vec{B},\ s\right)$, die den Ortsvektor des Punkts berechnet, der sich an der Stelle s auf der Strecke $\overline{AB}$ befindet. Damit ist Folgendes gemeint: Für $s = 0$ soll der

Startpunkt der Strecke, also $\vec{A}$, herauskommen, für $s = 1$ der Endpunkt $\vec{B}$. Für $s = \frac{1}{2}$ soll der Mittelpunkt der Strecke herauskommen, und so weiter. Geben Sie diese Funktion an!

Berechnen Sie anschließend:

$$\vec{P}\left(\begin{pmatrix}5\\-4\\3\end{pmatrix}, \begin{pmatrix}10\\4\\7\end{pmatrix}, \frac{7}{10}\right)$$

Aufgabe 6)

Was ergibt:

$$\begin{pmatrix}4\\3\\-2\end{pmatrix} \bullet \begin{pmatrix}1\\8\\1\end{pmatrix} + \begin{pmatrix}4\\0\end{pmatrix} \bullet \begin{pmatrix}0\\4\end{pmatrix} - \begin{pmatrix}3\\-1\\4\end{pmatrix}^2 \cdot \begin{pmatrix}7\\3\end{pmatrix}^2 + 145$$

Aufgabe 7)

Welche Winkel schließen die folgenden Vektoren jeweils miteinander ein?

$$\vec{a} = \begin{pmatrix}1\\0\end{pmatrix} \text{ und } \vec{b} = \begin{pmatrix}0\\1\end{pmatrix}$$

$$\vec{c} = \begin{pmatrix}2\\0\end{pmatrix} \text{ und } \vec{d} = \begin{pmatrix}1\\4\end{pmatrix}$$

$$\vec{e} = \begin{pmatrix}2\\1\\-3\end{pmatrix} \text{ und } \vec{f} = \begin{pmatrix}-1\\0\\3\end{pmatrix}$$

Aufgabe 8)

Wie könnte man mit Hilfe des Skalarprodukts erkennen, ob zwei Vektoren entgegengesetzte Richtungen haben?

Aufgabe 9)

Gelten die binomischen Formeln auch für Vektorvariablen und das Skalarprodukt, gelten also die folgenden Gleichungen?

$$\left(\vec{a}+\vec{b}\right)^2 = \vec{a}^2 + 2\vec{a}\vec{b} + \vec{b}^2$$
$$\left(\vec{a}-\vec{b}\right)^2 = \vec{a}^2 - 2\vec{a}\vec{b} + \vec{b}^2$$
$$\left(\vec{a}+\vec{b}\right)\left(\vec{a}-\vec{b}\right) = \vec{a}^2 - \vec{b}^2$$

Aufgabe 10)

Berechnen Sie die folgenden Kreuzprodukte:

a) $\begin{pmatrix}1\\4\\8\end{pmatrix}\times\begin{pmatrix}2\\4\\0\end{pmatrix}$ b) $\begin{pmatrix}1\\0\\0\end{pmatrix}\times\begin{pmatrix}0\\0\\1\end{pmatrix}$ c) $\begin{pmatrix}0\\0\\1\end{pmatrix}\times\begin{pmatrix}1\\0\\0\end{pmatrix}$ d) $\begin{pmatrix}2\\-3\\4\end{pmatrix}\times\begin{pmatrix}-4\\6\\-8\end{pmatrix}$

Aufgabe 11)

Welche Fläche A_P hat das durch die Vektoren $\vec{a} = (2,\ 4,\ 1)$ und $\vec{b} = (3,\ -1,\ 2)$ aufgespannte Parallelogramm? Zwei Vektoren können auch ein Dreieck aufspannen. Berechnen Sie diese Fläche A_D ebenfalls!

Aufgabe 12)

Gelten die binomischen Formeln für Vektorvariablen und das Kreuzprodukt?

2.11.2 Matrizen

Aufgabe 1)

Berechnen Sie:

a) $\begin{pmatrix}2 & -2\\-1 & 1\end{pmatrix} + 5\cdot\begin{pmatrix}7 & 7\\4 & 2\end{pmatrix} - \begin{pmatrix}1 & 4\\4 & 1\end{pmatrix}$

b) $\begin{pmatrix}4 & 2 & 1\\-1 & 0 & 2\end{pmatrix}\bullet\begin{pmatrix}2 & 3\\4 & 2\\1 & 6\end{pmatrix}$

c) $\begin{pmatrix}1 & 2\\4 & 1\\3 & 2\end{pmatrix}\bullet\begin{pmatrix}1 & 6 & 3 & 8\\5 & 2 & 7 & 4\end{pmatrix}$

Aufgabe 2)

Berechnen Sie die Inversen der folgenden Matrizen, falls sie invertierbar sind! Wie können Sie Ihre Ergebnisse überprüfen?

$$A = \begin{pmatrix} -1 & -4 \\ 2 & 8 \end{pmatrix} \qquad B = \begin{pmatrix} 2 & 3 & 6 \\ -5 & 12 & 25 \end{pmatrix} \qquad C = \begin{pmatrix} 2 & 1 \\ 3 & 2 \end{pmatrix}$$

$$D = \begin{pmatrix} 2 & 1 \\ 3 & 2 \end{pmatrix}^T \qquad E = \begin{pmatrix} 2 & 6 & -1 \\ -1 & -3 & \frac{1}{2} \\ \frac{1}{2} & \frac{3}{2} & -\frac{1}{4} \end{pmatrix} \qquad F = \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix}$$

Aufgabe 3)

Geben Sie eine Skalierungsmatrix an, die ein Objekt auf der *x*-Achse um den Faktor 2 vergrößert, es auf der *y*-Achse spiegelt (ohne Vergrößerung oder Verkleinerung) und es auf der *z*-Achse in seiner Größe halbiert. Transformieren Sie damit anschließend den Punkt mit den Koordinaten (4, 2, –5).

Aufgabe 4)

Wie sieht eine Matrix aus, die eine Rotation um die *y*-Achse mit einem Winkel von 30° durchführt? Transformieren Sie mit dieser Matrix den Punkt (1, –1, 2).

Aufgabe 5)

Geben Sie eine Translationsmatrix für den Translationsvektor (1, 3, 7) an, und transformieren Sie damit den Punkt (47, 11, –15).

Aufgabe 6)

Die Rotationsmatrizen, die wir kennen gelernt haben, drehen immer um den Koordinatenursprung. Wie könnte man eine Rotationsmatrix konstruieren, die um einen beliebigen Punkt dreht?

Aufgabe 7)

Überlegen Sie sich, wie Sie überprüfen können, ob eine Matrix orthogonal ist. Probieren Sie Ihr Verfahren dann mit den folgenden Matrizen aus:

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 5 & 0 \\ 0 & 5 \end{pmatrix} \quad C = \begin{pmatrix} 4 & 2 & 1 \\ 6 & 3 & 1.5 \\ 8 & 4 & 2 \end{pmatrix} \quad D = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix}$$

Aufgabe 8)

Zerlegen Sie die folgende Transformationsmatrix in eine Rotations- und eine Translationskomponente, und berechnen Sie dann auf einfache Weise die Inverse der Matrix.

$$M = \begin{pmatrix} 0.94 & 0 & 0.342 & 0 \\ 0 & 1 & 0 & 0 \\ -0.342 & 0 & 0.94 & 0 \\ 45 & 22 & -17 & 1 \end{pmatrix}$$

Aufgabe 9)

Ein 2D-Objekt hat die unten angegebene Transformationsmatrix *M*. Der Punkt *P* des Objekts hat die Objektkoordinaten (4, –3). Berechnen Sie die Weltkoordinaten dieses Punkts!

Welche Objektkoordinaten hat der Punkt *Q* mit den Weltkoordinaten (0, 1)?

Die Transformationsmatrix *M* lautet:

$$M = \begin{pmatrix} 0.5 & 0.866 & 0 \\ -0.866 & 0.5 & 0 \\ 5 & 3 & 1 \end{pmatrix}$$

2.11.3 Einfache geometrische Objekte

Aufgabe 1)

Geben Sie die Parameterdarstellung und die implizite Darstellung der Geraden an, die durch die Punkte *A* und *B* mit den Koordinaten $\vec{A} = (10,\ 5,\ 4)$ und $\vec{B} = (12,\ -6,\ 3)$ geht.

Aufgabe 2)

Gegeben sei eine Strecke von *P* nach *Q* mit $\vec{P} = (5,\ -5,\ 3)$ und $\vec{Q} = (10,\ -1,\ 7)$. Bestimmen Sie für die folgenden Punkte, ob sie auf dieser Strecke liegen:

$$\vec{A} = (6,\ -3,\ 4)$$
$$\vec{B} = (7.5,\ -3,\ 5)$$
$$\vec{C} = (15,\ 3,\ 11)$$

Aufgabe 3)

Bringen Sie die folgenden Ebenengleichungen in die Hessesche Normalform:

a) $\vec{P}(r, s) = \begin{pmatrix} 1 \\ 4 \\ 2 \end{pmatrix} + r \cdot \begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix} + s \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$

b) $\vec{X} \bullet \begin{pmatrix} 4 \\ 4 \\ -4 \end{pmatrix} = 48$

c) $\vec{P}(r, s) = \begin{pmatrix} 2 + r - s \\ 1 + 2r - 4s \\ 5s \end{pmatrix}$

Aufgabe 4)

Ermitteln Sie eine geeignete Gleichung für die Ebene, die durch die drei Punkte *A*, *B* und *C* verläuft, und bestimmen Sie dann, ob auch der Punkt *P* auf dieser Ebene liegt:

$$\vec{A} = \begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix},\ \vec{B} = \begin{pmatrix} 0 \\ 4 \\ 2 \end{pmatrix},\ \vec{C} = \begin{pmatrix} 0 \\ 8 \\ -4 \end{pmatrix},\ \vec{P} = \begin{pmatrix} 12 \\ 32 \\ -8 \end{pmatrix}$$

Aufgabe 5)

Überlegen Sie sich, wie man überprüfen kann, ob eine Gerade eine Ebene schneidet, und wie man gegebenenfalls die Koordinaten des Schnittpunkts herausfindet. Beachten Sie auch eventuelle Sonderfälle!

Was müsste geändert werden, um auch Strahlen und Strecken verwenden zu können?

Aufgabe 6)

Gegeben sind eine Kugel (Mittelpunkt und Radius) sowie eine Ebene in Hessescher Normalform. Wie kann zwischen den folgenden Fällen unterscheiden?

- Die Kugel befindet sich komplett vor der Ebene.
- Die Kugel befindet sich komplett hinter der Ebene.
- Die Kugel schneidet die Ebene.
- Die Kugel berührt die Ebene.

2.11.4 Farben

Aufgabe 1)

Ordnen Sie die folgenden als RGB-Vektor angegebenen Farben nach ihrer Luminanz:

$$\begin{pmatrix} 0.6 \\ 0.25 \\ 0.2 \end{pmatrix} \begin{pmatrix} 0.1 \\ 0 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \begin{pmatrix} 0.5 \\ 0.9 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0.5 \\ 0.7 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Aufgabe 2)

Haben die im Hexadezimal-RGB-Format angegebenen Farben `#1F070C` und `#D93154` dieselbe Chrominanz?

2.11.5 Vermischtes – Multiple Choice

Bei den folgenden Fragen ist jeweils genau eine der vorgegebenen Antworten korrekt.

Nr.	Frage und Antwortmöglichkeiten	Antw.
1	Was ist ein Vertex? A) ein Vektor, der senkrecht auf einer Oberfläche steht B) ein Eckpunkt einer Primitive, optional erweitert um beispielsweise Normalenvektor, Farbe oder Texturkoordinaten C) eine Shader-Art, also ein kleines Programm, das die Grafikkarte für jeden zu verarbeitenden Eckpunkt einmal ausführt	
2	In welcher Reihenfolge werden die Vertizes sinnvollerweise transformiert? A) Projektionsmatrix, Kameramatrix, Weltmatrix B) Weltmatrix, Kameramatrix, Projektionsmatrix C) Weltmatrix, Mondmatrix, Sonnenmatrix	
3	Was versteht man in der Computergrafik unter einer Primitiven? A) ein grundlegendes geometrisches Objekt, das direkt gerendert werden kann (Dreieck, Linie oder Punkt) B) einen ineffizienten (primitiven) Algorithmus C) eine grundlegende Transformation wie Translation, Rotation oder Skalierung	
4	Was ist korrekt? Das Phong-Modell ist ein … A) Verfahren zur Umrechnung zwischen RGB- und YUV-Farbraum. B) lokales Beleuchtungsmodell mit einer angenäherten globalen Komponente. C) rein globales Beleuchtungsmodell.	
5	Welche Beleuchtungskomponente ist für Glanzlichter (Highlights) zuständig? A) die ambiente Beleuchtungskomponente B) die diffuse Beleuchtungskomponente C) die spekulare Beleuchtungskomponente	

Nr.	Frage und Antwortmöglichkeiten	Antw.
6	Welche Aussage zum Shadow-Mapping ist *falsch*? A) Beim Shadow-Mapping werden nur die Farbinformationen der Szene aus der Sicht der Lichtquelle in die Shadow-Map gerendert. B) Punktlichtquellen lassen sich mit Shadow-Mapping nur schlecht realisieren. C) Die Qualität der Schatten hängt unter Anderem von der Größe und Bittiefe der Shadow-Map ab.	
7	Warum verbessert Back-Face-Culling bei geschlossenen Objekten die Performance? A) weil ungefähr 50% weniger Dreiecke gerendert werden müssen, die sowieso nicht sichtbar wären B) weil ein schnelleres Beleuchtungsverfahren verwendet wird C) weil damit der Speicherverbrauch im Z-Buffer geringer ist	
8	Wie wird ein Dreieck im Allgemeinen texturiert? A) Das macht die Grafikkarte von ganz allein. B) Man gibt den Vertizes Texturkoordinaten, die ihnen jeweils eine Position auf der Textur zuweisen. C) mit dem so genannten Z-Test	
9	Wofür verwendet man einen Z-Buffer? A) Im Z-Buffer wird das nächste anzuzeigende Frame gespeichert, das zur Darstellung in den Front-Buffer kopiert wird. B) um Objekte in der Szene schneller transformieren zu können C) um dafür zu sorgen, dass sich die Objekte auch ohne vorherige Sortierung nach ihrer Tiefe korrekt verdecken (Lösung des Sichtbarkeitsproblems)	
10	Was gibt es bei transparenten Objekten bezüglich des Z-Buffers zu beachten? A) Bei transparenten Objekten sollte das Schreiben in den Z-Buffer vermieden werden, aber der Z-Test sollte eingeschaltet sein. B) Sowohl das Schreiben in den Z-Buffer als auch der Z-Test ist unbedenklich. C) Man sollte nur den Z-Test deaktivieren.	
11	Welche Aussage zum Anti-Aliasing ist *richtig*? A) Beim Supersampling rendert man die Szene in halber Auflösung, um das Bild anschließend auf die gewünschte Auflösung hochzurechnen. B) Beim Multisampling wird an vier gitterförmig angeordneten Sample-Positionen innerhalb jedes Pixels getestet, ob die zu rendernde Primitive durch diesen Punkt verläuft. Daraus wird berechnet, wie stark der Pixel gefärbt werden darf. C) Mit eingeschaltetem Multisampling werden nur die Kanten der Primitiven geglättet, und insgesamt werden nicht mehr Pixel berechnet (Texturierung, Beleuchtung, …) als ohne Multisampling. Darum ist Multisampling effizienter als Supersampling.	

Nr.	Frage und Antwortmöglichkeiten	Antw.
12	Was ist unter MIP-Mapping zu verstehen? A) Von einer Textur wird eine Reihe von vergrößerten Kopien erzeugt, und ausgehend vom Abstand der Primitiven zur Kamera wird die passende Größe gewählt. B) Von einer Textur wird eine Reihe von verkleinerten, gefilterten Kopien erzeugt, und ausgehend von der Änderungsrate der Texelkoordinaten pro Pixel wird die passende Größe gewählt, um zu vermeiden, dass mehrere Texel auf einen Pixel fallen, was zu Aliasing führen würde. C) Beim MIP-Mapping wird zunächst horizontal zwischen zwei benachbarten Texeln linear interpoliert und anschließend diagonal, um Aliasing zu vermeiden.	
13	Was ist korrekt? Beim Alpha-Blending … A) ersetzt der neue Pixel den alten Pixel nicht, sondern der neue Pixel wird mit seinem Alphawert α gewichtet und der alte Pixel mit $1 - \alpha$. B) wird der alte Pixel zum neuen Pixel unverändert hinzuaddiert. C) entscheidet die Grafikkarte mit Hilfe des so genannten Alpha-Verfahrens, welcher der beiden Pixel heller ist, und verwirft dann den Dunkleren.	
14	Was sind Front- und Back-Buffer? A) besonders schnelle Speicherbereiche, die speziell für Kriegsspiele und Bäckereisimulationen genutzt werden B) Beides sind Bereiche innerhalb des Grafikspeichers. Der Front-Buffer enthält das Bild, das von der Grafikkarte kontinuierlich an den Monitor gesendet wird. Der Back-Buffer dient als Zwischenpuffer zum Aufbau des jeweils nächsten Frames. C) Am Ende jeder Zeile, wenn der Monitor seinen Elektronenstrahl zurücklenkt, werden Front- und Back-Buffer vertauscht, um Aliasing-Effekte zu verringern. Diese Puffer werden dabei im Speicher des Monitors aufbewahrt.	
15	Wie kann ein anisotropischer Filter dabei helfen, die Texturschärfe bei der Verwendung von MIP-Mapping zu verbessern? A) indem grundsätzlich eine größere MIP-Ebene verwendet wird B) indem abhängig von der Vergrößerung/Verkleinerung der Textur in x- und y-Richtung mehrere Samples pro Pixel genommen werden C) durch einen Scharfzeichnungsfilter, wie er auch von Bildbearbeitungsprogrammen angeboten wird	