Bundesrealgymnasium
Fadingerstraße 4, 4020 Linz

Mathematische Grundlagen der 3D-Grafik

Fachbereichsarbeit
aus
Mathematik

2008/2009

vorgelegt von
David Nadlinger

eingereicht bei
OStR. Mag. Herbert Lenz

Inhaltsverzeichnis

Vorwort

Heutzutage begegnen uns ständig computergenerierte Bilder, sei es in Form von Spezialeffekten oder künstlichen Charakteren in einem Kinofilm, sei es in Computerspielen oder bei der Visualisierung von Bauplänen, lange bevor die ersten Bagger rollen. Diese Möglichkeit, virtuelle Welten im Auge eines Betrachters Realität werden zu lassen, macht die 3D-Grafik in meinen Augen zu einem der interessantesten Anwendungsgebiete der Mathematik.

Als ich mich vor mittlerweile über fünf Jahren zum ersten Mal mit dieser Thematik beschäftigte, faszinierte mich vor allem die Entdeckung, dass hinter all den fantastischen Bildern letztendlich doch nichts anderes als »trockene« Mathematik steckt. Bei meinen ersten Versuchen, diese Mechanismen zu verstehen, war mir neben einigen guten Büchern vor allem mein Vater behilflich, der sich redlich bemühte, mir das nötige mathematische Handwerkszeug beizubringen, als selbst die trigonometrischen Funktionen noch Neuland für mich waren. An dieser Stelle ein herzliches Dankeschön dafür!

Während der letzten Jahre setzte ich mich immer wieder mit dem Thema auseinander, so dass ich heute mit Fug und Recht behaupten kann, die Vorgänge hinter den Kulissen von Computerspielen oder Modellierungsprogrammen zu begreifen. Ein Bereich blieb aber lange Zeit eine Art »blinder Fleck«: die genaue Bedeutung der zahlreichen mathematischen Hilfsmittel. Obwohl ich in meinen kleinen Programmen etwa zahlreiche Transformationen anwendete, hatte ich diese immer als gegeben betrachtet – die Herleitungen auch tatsächlich zu verstehen, schien mir ein hoffnungsloses Unterfangen zu sein. Mit der Zeit aber zeichneten sich für mich viele mathematische Zusammenhänge immer klarer ab, und bald entstand die Idee, mich endlich einmal intensiver mit den mathematischen Grundlagen der 3D-Grafik auseinanderzusetzen. Mangels konkreter Motivation legte ich diesen Plan aber erst einmal ad acta.

Als ich schließlich während der siebten Klasse von der interessanten Möglichkeit erfuhr, eine Fachbereichsarbeit zu verfassen, dauerte es nicht lange, bis ich mich auf der Suche nach einem geeigneten Thema wieder an dieses Vorhaben erinnerte. Hatte ich anfangs noch Zweifel, ob das Thema genug Stoff für eine ganze Fachbereichsarbeit bieten würde, musste ich nach dem Erstellen der ersten Konzepte die Themenstellung sogar explizit auf die Grundlagen eingrenzen, da der Umfang der Arbeit auszuufern drohte. Vor einigen Monaten begann ich schließlich mit der konkreten Bearbeitung des theoretischen und praktischen Teils dieses Projektes. Das Ergebnis dieser Bemühungen halten Sie, verehrter Leser, nun in Form dieses Dokuments und der beiliegenden CD-ROM in Händen.

Zum Schluss dieses Vorwortes möchte ich mich noch besonders bei meinem Mathematikprofessor Mag. Herbert Lenz bedanken, der sich in seinem voraussichtlich vorletzten Dienstjahr noch in das Abenteuer gestürzt hat, eine Fachbereichsarbeit zu betreuen. Obwohl die Korrekturarbeiten sicher mehr als nur ein paar Abende seiner wohlverdienten Freizeit beansprucht haben dürften, bekam ich stets gute Ratschläge und Verbesserungsvorschläge zu hören. Danke!

Linz, im Februar 2009

(David Nadlinger)

1. Einleitung

Die vorliegende Fachbereichsarbeit beschäftigt sich mit den mathematischen Grundlagen und Hintergründen der *3D-Grafik*. Unter diesem Begriff wird jenes Teilgebiet der *Computergrafik* verstanden, das die Generierung von zweidimensionalen Bildern aus der Beschreibung einer dreidimensionalen Umgebung im weitesten Sinne umfasst. Die Einsatzgebiete der 3D-Grafik sind während der letzten Jahre mit der steigenden Verfügbarkeit von leistungsfähigen Computersystem stetig gewachsen. Heute gehören dazu natürlich Computerspiele und Spezialeffekte in Filmen, aber auch die Visualisierung von Messdaten in der Medizin, die Darstellung von dreidimensionalen Plänen im CAD[1]-Bereich und vieles mehr.

In einem großen Teil der Verfahren und Algorithmen der 3D-Grafik kommen zwei mathematische Konzepte zur Anwendung, die in der Schule nicht behandelt werden: Matrizen und Quaternionen. Ich werde deshalb in Kapitel 2 auf die im weiteren Verlauf der Arbeit angewendeten Definitionen und Sätze eingehen. In diesem Zusammenhang möchte ich gleich darauf hinweisen, dass der Fokus dieser Arbeit ausdrücklich nicht auf den zahlreichen Eigenschaften dieser Strukturen liegt, sondern auf deren Anwendung in der 3D-Grafik. Diese können samt den Beweisen zu den verschiedenen Sätzen und Rechenregeln in jedem Standardwerk zur Linearen Algebra nachgeschlagen werden. Anstatt alle Beweise wiederzugeben, werde ich daher an einigen Stellen auf Fachliteratur zum Thema verweisen. Allein mit den Herleitungen aller in den verschiedenen Teilbereichen der Mathematik wichtigen Sätze zur Matrizenrechnung ließen sich sonst wohl ganze Buchbände füllen.

In Kapitel 3 werde ich versuchen, einen kurzen Überblick über die Umsetzung der 3D-Grafik am Computer zu geben. Dazu zählen natürlich in erster Linie die verschiedenen theoretischen Herangehensweisen, aber auch die praktische Realisierung in Form von Hardware wird kurz zur Sprache kommen.

Die beiden darauffolgenden Kapitel haben die verschiedenen Transformationen zum Thema, welche die Objekte auf dem Weg zum fertigen Bild durchlaufen. Kapitel 4 behandelt dabei die Skalierung, die Translation und die verschiedenen Möglichkeiten, um Rotationen darzustellen – kurz gesagt also die Operationen, um die Platzierung und Ausrichtung eines Objekts in der virtuellen Umgebung festzulegen. In Kapitel 5 werde ich genauer auf die Operationen eingehen, mit deren Hilfe schließlich eine Ansicht der virtuellen Welt auf den Bildschirm gebracht wird.

Im Rahmen dieser Fachbereichsarbeit habe ich mich nicht nur theoretisch mit dem Themenkomplex 3D-Grafik auseinandergesetzt. Begleitend zu der Arbeit am Text ist auch ein Projekt namens *d4* entstanden, das die praktische Umsetzung der mathematischen Grundlagen zeigt. In Anhang A werde ich Aufbau und Funktionsweise dieses Programms kurz beschreiben.

[1] von engl. *Computer Aided Design*, »Computerunterstützter Entwurf«. Erstellung von Modellen und Plänen mit Hilfe von Computer-Software, etwa im Maschinenbau.

2. Mathematische Grundlagen

In den folgenden Abschnitten möchte ich die mathematischen Grundprinzipien behandeln, auf denen nahezu alle Konzepte und Operationen der 3D-Grafik aufbauen. Wie schon in der Einleitung erwähnt, lassen sich die Definitionen und Sätze in jedem Standardwerk zur Linearen Algebra nachschlagen, beispielsweise in den im Anhang »Verwendete Literatur« aufgeführten Werken »Taschenbuch der Mathematik« (Bronstein und Semendjajew 1981), »Elemente der Mathematik« (Athen und Wigand 1967) und »Lineare Optimierung und Anwendungen« (Dorninger u. a. 1977).

2.1. Vektoren

Vektoren sind wohl die wichtigste Struktur in der 3D-Grafik. Sie werden nicht nur in ihrer dreidimensionalen Form eingesetzt, sondern auch in ihrer zweidimensionalen Ausprägung und um eine vierte Komponente erweitert als Vektoren mit homogenen Koordinaten (siehe Abschnitt 2.2.3).

Trotz der universellen Verwendung gehen die benötigten Operationen nicht über den Schulstoff hinaus. Insbesondere das Skalarprodukt und das Kreuzprodukt werden sehr häufig benötigt, zum Beispiel um über den Zusammenhang $\vec{a} \cdot \vec{b} = |\vec{a}|\,|\vec{b}| \cos \alpha$ den von zwei Vektoren eingeschlossenen Winkel zu berechnen oder über das Kreuzprodukt die Flächennormale eines Polygons zu bestimmen.

Ich möchte an dieser Stelle noch kurz auf eine Eigenheit in Bezug auf Vektoren hinweisen, die im Schulunterricht bestenfalls kurz erwähnt wird: Vektoren können auf zwei Arten notiert werden, in Spaltenform

$$\vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix},$$

oder in Zeilenform

$$\vec{v} = \begin{pmatrix} x & y & z \end{pmatrix}.$$

Scheint der Unterschied zunächst noch rein kosmetischer Natur zu sein, zeigt sich spätestens beim Arbeiten mit Matrizen, dass die Entscheidung für eine der Formen doch erhebliche Konsequenzen nach sich zieht (siehe Abschnitt 2.2.4). In der Grafikprogrammierung werden beide Varianten gleichermaßen verwendet; in diesem Dokument werde ich Spaltenvektoren verwenden, um den Bezug zur Schulmathematik zu wahren.

Eingebettet in Fließtext sind Zeilenvektoren zwar wesentlich platzsparender[1], man kann einen Spaltenvektor aber als transponierten Zeilenvektor $\vec{v} = \begin{pmatrix} x & y & z \end{pmatrix}^T$ anschreiben, um diesen Vorteil zu übernehmen (siehe Abschnitt 2.2.4).

[1] Zeilenvektoren sind nicht nur platzsparender, sondern auch leichter in Kommentare in Programmcode einzubetten. Angeblich ist das einer der Gründe, warum in den Anfangszeiten der Computergrafik oft Zeilenvektoren verwendet wurden.

2.2. Matrizen

Auf den ersten Blick gleichen Matrizen einfachen Tabellen von Zahlen, es gibt aber einen wesentlichen Unterschied: Für sie sind Rechenoperationen wie Addition und Multiplikation definiert.

Eine Matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \in \mathbb{R}^{m \times n} \tag{2.1}$$

ist ein Schema aus $m \cdot n$ Elementen a_{ij}, die in m Zeilen und n Spalten angeordnet sind. Es handelt sich um eine $m \times n$-Matrix (sprich: *m kreuz n*).

Die Elemente $a_{k1}, a_{k2}, \ldots, a_{km}$ (mit $1 \leq k \leq n$) bilden den k-ten Zeilenvektor der Matrix und analog dazu die Elemente $a_{1l}, a_{2l}, \ldots, a_{nl}$ den l-ten Spaltenvektor. (vgl. Dorninger u. a. 1977, 39)

Matrizen mit $m = n$ werden als *quadratische Matrizen* bezeichnet und nehmen bei vielen Operationen eine Sonderstellung ein.

Zwei Matrizen heißen gleich, wenn sie von gleicher Dimension sind und in allen an gleichen Stellen stehenden Elementen übereinstimmen. (vgl. Bronstein und Semendjajew 1981, 204)

Als *Hauptdiagonale* wird die gedachte Linie von der linken oberen Ecke bis zur rechten unteren Ecke einer Matrix bezeichnet – die Elemente der Hauptdiagonale sind also a_{11}, a_{22}, etc. Quadratische Matrizen, die außerhalb der Hauptdiagonale nur Elemente mit dem Wert 0 haben, werden als *Diagonalmatrizen* bezeichnet.

Grundsätzlich können die Werte der Matrix jeder algebraischen Struktur entstammen, für die Addition und Multiplikation definiert sind. Im Rahmen dieser Arbeit werde ich aber nur Matrizen über $\mathbb{R}$ betrachten, andere Zahlenräume sind in der 3D-Grafik nicht gebräuchlich.

Viele Eigenschaften und Operationen in Bezug auf Matrizen werden aber hauptsächlich im zweiten großen Einsatzgebiet der Matrizenrechnung gebraucht, dem Lösen von linearen Gleichungssystemen. Ich werde im Weiteren nur auf die in der 3D-Grafik benötigen Teile eingehen.

2.2.1. Addition

Zwei Matrizen der gleichen Dimensionen $m \times n$ werden addiert, indem man jeweils die Elemente der beiden Matrizen addiert. Die Elemente der Matrix $C = A + B$ sind also durch

$$c_{ij} = a_{ij} + b_{ij} \tag{2.2}$$

mit $1 \leq i \leq m$ und $1 \leq j \leq n$ definiert.

Folglich ist das Ergebnis der Addition wiederum eine $m \times n$-Matrix.

Die Matrizenaddition ist assoziativ und kommutativ. (vgl. Bronstein und Semendjajew 1981, 205)

Es ist leicht zu erkennen, dass die Matrizenaddition – analog zur Null bei der Addition von Skalaren – ein neutrales Element besitzt: eine Matrix deren Elemente alle 0 sind, kurz *Nullmatrix* genannt.

In der 3D-Grafik wird die Addition äußerst selten gebraucht, da sie im Gegensatz etwa zur Multiplikation oder zur Inversion keine geometrische Entsprechung hat.

2.2.2. Multiplikation

Das Produkt C zweier Matrizen $A \in \mathbb{R}^{m \times o}$ und $B \in \mathbb{R}^{o \times n}$ ist als

$$c_{ij} = \sum_{k=1}^{o} a_{ik} \cdot b_{kj} \tag{2.3}$$

mit $1 \leq i \leq m$ und $1 \leq j \leq n$ definiert (C ist also eine $m \times n$-Matrix).

Etwas anschaulicher formuliert erhält man das i-te Element der j-ten Spalte des Ergebnisses, indem man das Skalarprodukt der i-ten Zeile der linken Matrix mit der j-ten Spalte der rechten Matrix bildet – kurz »Zeile mal Spalte«.

Die Spaltenanzahl der linken Matrix und die Zeilenanzahl der rechten Matrix müssen also gleich sein (oben durch o ausgedrückt), damit eine Multiplikation möglich ist. Das entstehende Produkt hat dabei die Dimensionen $m \times n$, also die Zeilenanzahl der linken Matrix und die Spaltenanzahl der rechten Matrix.

Ein Beispiel zur Veranschaulichung:

$$\begin{aligned} \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{pmatrix} \cdot \begin{pmatrix} 6 & 7 \\ 8 & -9 \\ -10 & 11 \end{pmatrix} &= \begin{pmatrix} 0 \cdot 6 + 1 \cdot 8 + 2 \cdot (-10) & 0 \cdot 7 + 1 \cdot (-9) + 2 \cdot 11 \\ 3 \cdot 6 + 4 \cdot 8 + 5 \cdot (-10) & 3 \cdot 7 + 4 \cdot (-9) + 5 \cdot 11 \end{pmatrix} \\ &= \begin{pmatrix} 0 + 8 - 20 & 0 - 9 + 22 \\ 18 + 32 - 50 & 21 - 36 + 55 \end{pmatrix} = \begin{pmatrix} -12 & 13 \\ 0 & 40 \end{pmatrix} \end{aligned} \tag{2.4}$$

Für die Multiplikation von quadratischen Matrizen gibt es ein neutrales Element, für das

$$A \cdot E = E \cdot A = A \tag{2.5}$$

gilt. E wird als *Einheitsmatrix* bezeichnet und ist eine Diagonalmatrix, deren Elemente entlang der Hauptdiagonale alle den Wert 1 haben. Es gibt natürlich für alle verschiedenen Größen $n \times n$ jeweils eine eigene Einheitsmatrix E_n. Im Normalfall ist aber ohnehin ersichtlich, welche Dimensionen die verwendete Einheitsmatrix haben muss, deswegen wird in der Praxis meistens auf den Index verzichtet.

Sofern die Multiplikationen nicht durch die Dimensionen der Matrizen unmöglich sind, ist die Matrizenmultiplikation assoziativ. Das Kommutativgesetz gilt aber nicht! (vgl. Bronstein und Semendjajew 1981, 205)

2.2.3. Transformation von Vektoren

Da ein n-dimensionaler Spaltenvektor ja im Grunde nichts anderes ist als eine $n \times 1$-Matrix, kann man natürlich auch eine Matrix mit einem Vektor multiplizieren. Wenn es sich dabei um eine quadratische $n \times n$-Matrix handelt, ist das Ergebnis wiederum ein n-dimensionaler Vektor. Rechnet man eine solche Multiplikation in ihrer allgemeinen Form aus

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} m_{11} \cdot x + m_{12} \cdot y + m_{13} \cdot z \\ m_{21} \cdot x + m_{22} \cdot y + m_{23} \cdot z \\ m_{31} \cdot x + m_{32} \cdot y + m_{33} \cdot z \end{pmatrix}, \tag{2.6}$$

so bemerkt man, dass die Zeilen der Matrix jeweils die Koeffizienten einer Linearkombination der Komponenten des Vektors darstellen. Es kann daher *jede lineare Abbildung* zwischen zwei (endlichdimensionalen) Vektorräumen als Matrix dargestellt werden. Die Multiplikation einer Matrix mit einem Vektor wird aus diesem Grund auch als *Transformation* des Vektors bezeichnet. Wie man auch leicht durch Einsetzen überprüfen kann, ändert eine Multiplikation mit der Einheitsmatrix den Vektor naheliegenderweise nicht.

Allerdings reicht eine Linearkombination der Komponenten nicht aus, um alle in der 3D-Grafik benötigten Transformationen darzustellen – manchmal ist es nötig, alle Komponenten des Vektors durch eine von ihnen zu dividieren oder zu den Komponenten von ihrem Wert unabhängige Konstanten zu addieren. Um auch diesen größeren Raum der *affinen Transformationen* in Matrixform darstellen zu können, bedient man sich eines Tricks und erweitert den Vektor um eine zusätzliche Dimension, die meistens mit w bezeichnet wird. Man erhält die sogenannten *homogenen Koordinaten* – das Pendant mit homogenen Koordinaten zu einem dreidimensionalen Vektor ist also ein Vektor im $\mathbb{R}^4$.

Um einen Vektor mit »normalen« kartesischen Koordinaten in einen Vektor mit homogenen Koordinaten zu überführen, setzt man einfach $w = 1$:

$$\begin{pmatrix} x & y & z \end{pmatrix}^T \rightarrow \begin{pmatrix} x & y & z & 1 \end{pmatrix}^T \tag{2.7}$$

Für den umgekehrten Fall dividiert man zuerst alle Komponenten durch w:

$$\begin{pmatrix} x & y & z & w \end{pmatrix}^T \rightarrow \begin{pmatrix} \frac{x}{w} & \frac{y}{w} & \frac{z}{w} & 1 \end{pmatrix}^T \tag{2.8}$$

Danach kann man aus den ersten drei Komponenten des Vektors wieder einen Vektor mit kartesischen Koordinaten bilden, w kann also einfach weggelassen werden. (vgl. Bishop und Van Verth 2004, 219)

In der Grafikprogrammierung werden homogene Koordinaten meist nur dort verwendet, wo sie gebraucht werden, wenn also affine Transformationen durchgeführt werden sollen. Bei allen anderen Berechnungen wird mit kartesischen Koordinaten gearbeitet, zwischen den beiden Koordinatentypen wird nach Bedarf hin- und hergesprungen. Auch wenn diese Vorgehensweise aus mathematischer Sicht nicht konsequent ist, stellt sie in der Praxis einen sinnvollen Kompromiss zwischen Performance[2] und Flexibilität dar.

Die Darstellung aller Transformationen als Matrizen hat den großen Vorteil, dass beliebig viele Transformationen mittels Multiplikation in eine Matrix kombiniert werden können, die dann wiederum auf viele Vektoren angewendet werden kann. Dies ist möglich, weil die Matrizenmultiplikation assoziativ ist:

$$(M_3 \cdot (M_2 \cdot (M_1 \cdot \vec{v}))) = (M_3 \cdot M_2 \cdot M_1) \cdot \vec{v} \tag{2.9}$$

Durch diese Universalität von Matrizen ist es möglich, bestimmte zeitintensive Berechnungen in der Computergrafik sehr effizient zu gestalten (siehe Kapitel 3.2.1).

2.2.4. Transposition

Bei der Transposition werden die Zeilen und Spalten einer Matrix vertauscht, die Matrix wird quasi entlang ihrer Hauptdiagonale gespiegelt:

$$(A^T)_{ij} = a_{ji} \tag{2.10}$$

[2] Performance: Leistung bzw. Geschwindigkeit eines Computerprogramms

Ein konkretes Beispiel:

$$\begin{pmatrix} 1 & -2 & 3 \\ 4 & 5 & -6 \end{pmatrix}^T = \begin{pmatrix} 1 & 4 \\ -2 & 5 \\ 3 & -6 \end{pmatrix} \tag{2.11}$$

Folglich wird eine $m \times n$-Matrix zu einer $n \times m$-Matrix. Wird eine $n \times 1$-Matrix, also ein Spaltenvektor, transponiert, ergibt sich daher eine $1 \times n$-Matrix, der entsprechende Zeilenvektor.

Wie sich leicht überprüfen lässt, hebt sich die Transposition auf, wenn sie zwei Mal auf eine Matrix angewendet wird. Es gilt also

$$(A^T)^T = A. \tag{2.12}$$

Bezüglich der Addition ist die Transposition distributiv, bezüglich der Multiplikation distributiv unter Umkehrung der Reihenfolge. (vgl. Bronstein und Semendjajew 1981, 206)

Es gelten also die Gleichungen

$$(A + B)^T = A^T + B^T \tag{2.13}$$

und

$$(A \cdot B)^T = B^T \cdot A^T. \tag{2.14}$$

Der Zusammenhang aus Gleichung 2.14 zieht einen wesentlichen Unterschied in der Verwendung von Spalten- und Zeilenvektoren nach sich. Dazu ein kleines Beispiel: Gegeben sei ein Spaltenvektor $\vec{s}$ und eine Matrix A, durch Multiplikation aus mehreren Teilmatrizen $A_1, A_2, \ldots, A_n$ kombiniert, die jeweils einer Transformation entsprechen. Der Vektor wird nun von *rechts* an die Matrix multipliziert:

$$\vec{s'} = A \cdot \vec{s} = A_1 \cdot A_2 \cdot \ldots \cdot A_n \cdot \vec{s} = A_1 \cdot A_2 \cdot \ldots \cdot (A_n \cdot \vec{s}). \tag{2.15}$$

Wie durch die Klammernsetzung angedeutet, verhält sich der Ergebnisvektor $\vec{s'}$, als hätte man die Teiltransformationen der Reihe nach beginnend mit A_n, also der zuletzt zu A multiplizierten Matrix, auf den Vektor angewendet.

Wird nun statt einem Spaltenvektor ein Zeilenvektor $\vec{z} = \vec{s}^T$ verwendet, ergibt sich aus Gleichung 2.14 (wie auch aus den für die Multiplikation nötigen Dimensionen der Argumente), dass ein Zeilenvektor von *links* an die *transponierte* Matrix multipliziert werden muss. Wenn diese wieder aus n Teiltransformationen zusammengesetzt wird, reicht es nicht, die einzelnen Matrizen zu transformieren, es muss zusätzlich noch die Multiplikationsreihenfolge der Teilmatrizen umgekehrt werden:

$$\vec{s'}^T = \vec{s}^T \cdot A^T = \vec{s}^T \cdot A_n^T \cdot A_{n-1}^T \cdot \ldots \cdot A_1^T. \tag{2.16}$$

Dies ist insofern erwähnenswert, als in der 3D-Grafik zwei große Standards existieren, von denen einer Spaltenvektoren, der andere Zeilenvektoren vorsieht (mehr dazu in Kapitel 3.3) und manche Algorithmen daher nicht 1:1 von einem Standard auf den anderen übertragbar sind.

2.2.5. Determinante

Die Determinante ist eine spezielle Funktion, die einer quadratischen Matrix A eine Zahl $\det A$ zuordnet. Das Ergebnis kann auf verschiedene Weisen interpretiert werden, die bekannteste Verwendung ist sicherlich zur Volumenberechnung in der Vektorrechnung (der Betrag der Determinante dreier Vektoren aus dem $\mathbb{R}^3$ entspricht dem Spatprodukt).

Wenn die Matrix als Koeffizientenmatrix eines linearen Gleichungssystems aufgefasst wird, besitzt dieses genau dann eine eindeutige Lösung, wenn ihre Determinante ungleich 0 ist (solche Matrizen werden als *reguläre Matrizen* bezeichnet). (vgl. Dorninger u. a. 1977, 56)

In der Grafikprogrammierung ist die Determinante selbst praktisch bedeutungslos, allerdings wird sie für ein Verfahren zur Inversion von Matrizen benötigt (siehe Abschnitt 2.2.6).

Die Determinante einer 2×2-Matrix

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

lautet

$$\det A = ad - cb. \tag{2.17}$$

Mit Hilfe des Laplace'schen Entwicklungssatzes kann die Berechnung von Determinanten aller höheren Dimensionen auf 2×2-Determinanten zurückgeführt werden (vgl. Bronstein und Semendjajew 1981, 202): Dazu wird zunächst das Konzept der *Minoren* eingeführt. Der Minor M_{ij} einer Matrix $A \in \mathbb{R}^{n \times n}$ ist die Determinante der $(n-1) \times (n-1)$-Untermatrix, die durch Streichen der i-ten Zeile und der j-ten Spalte von A entsteht. Das Produkt

$$\tilde{a}_{ij} = (-1)^{i+j} M_{ij} \tag{2.18}$$

wird als *Kofaktor* von A bezeichnet. Der Laplace'sche Erweiterungssatz besagt nun, dass die Determinante einer Matrix als Summe der Produkte ihrer Elemente mit den zugehörigen Kofaktoren »nach« einer beliebigen Zeile oder Spalte entwickelt werden kann:

$$\det A = \sum_{i=1}^{n} a_{ik} \tilde{a}_{ik} = \sum_{j=1}^{n} a_{kj} \tilde{a}_{kj} \tag{2.19}$$

(mit $1 \leq k \leq n$).

Hieraus lässt sich schon eine Eigenschaft in Bezug auf die Transposition erkennen: Die Determinante einer Matrix verändert sich nicht, wenn man die Matrix transponiert. (vgl. Bronstein und Semendjajew 1981, 201) Es gilt also:

$$\det A = \det A^T \tag{2.20}$$

Nach diesem Verfahren lässt sich eine komplette Formel für die Determinante ableiten, was für die Geschwindigkeit der Berechnung am Computer von Vorteil ist. Für eine 3×3-Matrix B erhält man beispielsweise nach dem Vereinfachen

$$\det B = b_{11}b_{22}b_{33} + b_{12}b_{23}b_{31} + b_{13}b_{21}b_{32} - b_{13}b_{22}b_{31} - b_{12}b_{21}b_{33} - b_{11}b_{23}b_{32}. \tag{2.21}$$

Wie man leicht erkennen kann, wächst die Komplexität dieses Verfahrens mit steigender Dimension ziemlich schnell. Zur Berechnung höherdimensionaler Determinanten können daher andere algorithmische Verfahren sinnvoller sein. In der 3D-Programmierung sind diese aber mehr oder weniger bedeutungslos, da es hier auf schnelle Berechenbarkeit bei kleinen Dimensionen ankommt.

2.2.6. Inverse Matrix

Eine Matrix A kann eine inverse Matrix A^{-1} besitzen, für die

$$A \cdot A^{-1} = A^{-1} \cdot A = E \tag{2.22}$$

gilt. A^{-1} wird auch kurz als Inverse bezeichnet. Es existiert nur zu regulären Matrizen eine inverse Matrix, also zu quadratischen Matrizen, deren Determinante ungleich 0 ist. Wie man sich leicht vor Augen führen kann, können nicht-quadratische Matrizen prinzipiell keine inverse Matrix besitzen. (vgl. Bronstein und Semendjajew 1981, 206)

Im Bezug auf die Rolle der Matrix als Darstellung einer geometrischen Transformation ist es wichtig zu erkennen, dass die Multiplikation eines Vektors mit der inversen Matrix dem *Rückgängig machen* der Transformation entspricht. Hat man also beispielsweise einen Vektor $\vec{v}$ mit der Matrix A transformiert, erhält man nach einer Multiplikation mit der Inversen A^{-1} wieder den ursprünglichen Vektor. Dies lässt sich auch direkt aus der Definition ableiten: $A^{-1} \cdot (A \cdot \vec{v}) = (A^{-1} \cdot A) \cdot \vec{v} = E \cdot \vec{v} = \vec{v}$.

Eine singuläre, also nicht invertierbare Matrix ist oft auf den ersten Blick erkennbar, wenn man die Auswirkungen der Matrix auf einen Vektor betrachtet. Beispielsweise fällt es leicht zu verstehen, dass zu der Matrix

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.23}$$

keine inverse Matrix existieren kann, denn wenn man einen Vektor mit dieser Matrix transformiert, geht die Information aus der y-Koordinate »verloren«. Somit ist es nachträglich nicht mehr möglich, den ursprünglichen Vektor wiederherzustellen, die Multiplikation also umzukehren.

In Bezug auf Transposition gilt für die Inversion (vgl. Wikipedia 2009b):

$$(A^{-1})^T = (A^T)^{-1}. \tag{2.24}$$

Im Wesentlichen gibt es zwei Verfahren, um die Inverse zu einer Matrix zu berechnen. Für die folgenden Betrachtungen soll

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 2 & 3 & 0 \\ 3 & 4 & 1 \end{pmatrix}$$

als Ausgangsmatrix dienen.

Die erste Variante beruht auf dem Gauß-Eliminationsverfahren. Dazu erweitert man zunächst die Ausgangsmatrix mit der Einheitsmatrix der gleichen Dimension zu einer Matrix der Form $(A \,|\, E)$:

$$\left(\begin{array}{ccc|ccc} 1 & 2 & 0 & 1 & 0 & 0 \\ 2 & 3 & 0 & 0 & 1 & 0 \\ 3 & 4 & 1 & 0 & 0 & 1 \end{array}\right) \tag{2.25}$$

Danach wendet man den Gauß-Jordan-Algorithmus auf diese Matrix an[3], bis auf der linken Seite die Einheitsmatrix steht, was in diesem Fall zu folgendem Ergebnis führt:

$$\left(\begin{array}{ccc|ccc} 1 & 0 & 0 & -3 & 2 & 0 \\ 0 & 1 & 0 & 2 & -1 & 0 \\ 0 & 0 & 1 & 1 & -2 & 1 \end{array}\right) \tag{2.26}$$

[3] Vereinfacht gesagt addiert und subtrahiert man dabei Vielfache von Zeilen der Matrix zu bzw. von anderen Zeilen. Der Algorithmus wird auch oft zum Lösen von linearen Gleichungssystemen in Matrixform benutzt.

Auf der rechten Seite kann die inverse Matrix A^{-1} nun direkt abgelesen werden.

Mit diesem Verfahren ist die inverse Matrix recht einfach händisch zu berechnen. Es hat allerdings den Nachteil, dass sich daraus keine »fertige« Formel ableiten lässt und es daher für den Einsatz in der Grafikprogrammierung nicht ausreichend schnell zu berechnen ist[4].

Daher wird in der 3D-Programmierung meist auf die zweite Variante zurückgegriffen, nach der die Inverse zur Matrix A folgendermaßen definiert ist:

$$A^{-1} = \frac{1}{\det A} \cdot \operatorname{adj} A \tag{2.27}$$

adj A bezeichnet hier die *Adjunkte* der Matrix A, die auch *komplementäre Matrix* genannt wird. Sie besteht aus der transponierten Matrix der Kofaktoren (siehe Gleichung 2.18):

$$\operatorname{adj} A = \begin{pmatrix} \tilde{a}_{11} & \tilde{a}_{21} & \cdots & \tilde{a}_{n1} \\ \tilde{a}_{12} & \tilde{a}_{22} & \cdots & \tilde{a}_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{a}_{1n} & \tilde{a}_{2n} & \cdots & \tilde{a}_{nn} \end{pmatrix}. \tag{2.28}$$

Mit Hilfe dieses Zusammenhangs lassen sich Formeln für die Inversion von Matrizen beliebiger Dimension herleiten. Dabei wird allerdings bald deutlich, dass die Komplexität bzw. die Anzahl der nötigen Rechenoperationen mit der Erhöhung der Dimensionen sehr schnell ansteigt:

Ergibt sich für die Inverse einer 2×2-Matrix

$$B = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \tag{2.29}$$

noch der relativ übersichtliche Ausdruck

$$B^{-1} = \frac{1}{ad - bc} \cdot \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}, \tag{2.30}$$

führt das Verfahren für eine 3×3-Matrix

$$C = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \tag{2.31}$$

schon zu der wesentlich komplizierteren Formel

$$C^{-1} = \frac{1}{\det C} \cdot \begin{pmatrix} ei - hf & ch - bi & bf - ce \\ fg - di & ai - cg & cd - af \\ dh - eg & bg - ah & ae - bd \end{pmatrix}. \tag{2.32}$$

In Bezug auf die 3D-Grafik am Computer bleibt noch festzustellen, dass die Inversion im Vergleich zu anderen Operationen (etwa der Matrizenmultiplikation) recht rechenaufwändig ist und daher aus Performancegründen nicht allzu oft ausgeführt werden sollte (siehe Abschnitt 3.3).

[4] Dies gilt wiederum nur für relativ kleine Matrizen, wie sie in der 3D-Grafik ausschließlich zur Anwendung kommen. Bei höherdimensionalen Matrizen führen andere algorithmische Ansätze in der Regel schneller zum Ziel.

2.3. Quaternionen

Der Körper der komplexen Zahlen $\mathbb{C}$ als Erweiterung der reellen Zahlen ist ja bereits aus dem Schulunterricht bekannt. Neben einigen anderen wichtigen Eigenschaften wie der algebraischen Abgeschlossenheit[5], haben die komplexen Zahlen die angenehme Eigenschaft, dass sie gleichzeitig ein zweidimensionaler reeller Vektorraum sind und sich mit ihrer Hilfe viele Brücken zwischen Geometrie und Algebra schlagen lassen.

Es verwundert deshalb nicht, dass viele Mathematiker versuchten, ein dreidimensionales Äquivalent zu den komplexen Zahlen zu finden. Einer von ihnen war auch der irische Mathematiker Sir William Rowan Hamilton (1805–1865), der wie alle anderen zunächst an dem Problem scheiterte, dass für »dreidimensionale komplexe Zahlen« keine vernünftigen Rechenregeln zu finden waren. Schließlich kam ihm die entscheidende Idee (angeblich, während er mit seiner Frau über die Broom Bridge in Dublin spazierte), das Konzept um eine zusätzliche vierte Komponente zu erweitern. (zu diesem Abschnitt vgl. Koch 2008, 29-37)

Eine *Quaternion*[6] ist also eine Zahl der Form

$$q = w + xi + yj + zk = \left[w;\ \begin{pmatrix} x & y & z \end{pmatrix}^T\right] \in \mathbb{H} \tag{2.33}$$

mit $w, x, y, z \in \mathbb{R}$, wobei j und k analog zu dem von den komplexen Zahlen bekannten i zwei weitere voneinander verschiedene imaginäre Einheiten sind. Der Raum aller Quaternionen wird in Anlehnung an den Namen ihres geistigen Vaters als $\mathbb{H}$ bezeichnet.

Für i, j und k gilt der Zusammenhang

$$i^2 = j^2 = k^2 = i \cdot j \cdot k = -1, \tag{2.34}$$

der auch als Hamilton'sche Regel bekannt ist.

Zwischen den drei imaginären Einheiten besteht demnach eine zyklische Abhängigkeit:

$$i \cdot j = -j \cdot i = k \tag{2.35}$$
$$j \cdot k = -k \cdot j = i \tag{2.36}$$
$$k \cdot i = -i \cdot k = j \tag{2.37}$$

Die Multiplikation der imaginären Einheiten und damit auch der Quaternionen generell ist also nicht kommutativ!

Der (reelle) Betrag einer Quaternion ist analog zu den komplexen Zahlen und den Vektoren als

$$|q| = \sqrt{w^2 + x^2 + y^2 + z^2} \tag{2.38}$$

definiert. Eine Quaternion mit $|q| = 1$ wird als *Einheitsquaternion* bezeichnet.

Wie schon durch die Zeichen x, y und z angedeutet, kann man eine Quaternion auch als Struktur mit einem *skalaren Anteil* w und einem *vektoriellen Anteil* $\begin{pmatrix} x & y & z \end{pmatrix}^T$ gesehen werden. Eine alternative Schreibweise für die Quaternion $q = w + v_x i + v_y j + v_z k$ lautet daher $[w;\ \vec{v}]$. Quaternionen mit dem

[5] Die algebraische Abgeschlossenheit der komplexen Zahlen ist der Inhalt des bekannten Fundamentalsatzes der Algebra, der Ende des 18. Jahrhunderts von Carl Friedrich Gauß bewiesen wurde.

[6] In der Literatur wird der Begriff sowohl im Maskulinum, als auch im Femininum gebraucht. Ich habe mich in diesem Dokument dazu entschlossen, die weibliche Form zu verwenden, so wie es der »Duden« verschlägt.

Realteil 0 heißen auch *reine Quaternionen*, eine reine Quaternion $q = 0 + v_x i + v_y j + v_z k$ kann jederzeit als Vektor $\vec{v} = \begin{pmatrix} x & y & z \end{pmatrix}^T \in \mathbb{R}^3$ ausgedrückt werden und umgekehrt. Die einem Vektor $\vec{v}$ zugeordnete reine Quaternion wird als $\hat{v}$ abgekürzt (vgl. Formella und Fellner 2005, 92; Eberly 2002, 4):

$$\hat{v} = [0;\ \vec{v}] \tag{2.39}$$

Die konjugierte Quaternion zu einer Quaternion $q = w + v_x i + v_y j + v_z k$ ist als

$$\overline{q} = w - v_x i - v_y j - v_z k = [w;\ -\vec{v}] \tag{2.40}$$

definiert. Die Konjugation ändert also das Vorzeichen des Vektorteils der Quaternion, lässt den Skalarteil aber unverändert.

Zwei Quaternionen q_1 und q_2 werden *addiert* und *subtrahiert*, indem man so wie bei Vektoren und komplexen Zahlen die einzelnen Koeffizienten addiert bzw. subtrahiert:

$$\begin{aligned} q_1 \pm q_2 &= w_1 + x_1 i + y_1 j + z_1 k \pm (w_2 + x_2 i + y_2 j + z_2 k) \\ &= w_1 \pm w_2 + (x_1 \pm x_2) \cdot i + (y_1 \pm y_2) \cdot j + (z_1 \pm z_2) \cdot z \\ &= [w_1 \pm w_2;\ \vec{v_1} \pm \vec{v_2}] \end{aligned} \tag{2.41}$$

Eine Quaternion q kann nach den ganz gewöhnlichen Rechenregeln mit einem *Skalar* λ multipliziert werden:

$$\lambda \cdot q = \lambda \cdot (w + xi + yj + zk) = \lambda w + \lambda x i + \lambda y j + \lambda z k = [\lambda w;\ \lambda \cdot \vec{v}] \tag{2.42}$$

Zwei Quaternionen q_1 und q_2 können ebenfalls ganz gewöhnlich miteinander *multipliziert* werden. Ausmultiplizieren (unter Beachtung der Hamilton-Regeln) und Anwenden des Distributivgesetzes ergibt:

$$\begin{aligned} q_1 \cdot q_2 &= (w_1 + x_1 i + y_1 j + z_1 k) \cdot (w_2 + x_2 i + y_2 j + z_2 k) \\ &= w_1 w_2 - (x_1 x_2 + y_1 y_2 + z_1 z_2) \\ &\quad + (w_1 x_2 + w_2 x_1 + y_1 z_2 - y_2 z_1) \cdot i \\ &\quad + (w_1 y_2 + w_2 y_1 + x_2 z_1 - x_1 z_2) \cdot j \\ &\quad + (w_1 z_2 + w_2 z_1 + x_1 y_2 - x_2 y_1) \cdot k \end{aligned} \tag{2.43}$$

Fasst man den Imaginärteil der Quaternion als Vektor auf, kann man die Multiplikation auch wie folgt auflösen:

$$q_1 \cdot q_2 = [w_1 w_2 - \vec{v_1} \cdot \vec{v_2};\ \vec{v_1} \times \vec{v_2} + w_1 \cdot \vec{v_2} + w_2 \cdot \vec{v_1}] \tag{2.44}$$

Das Produkt zweier Quaternionen ist nicht kommutativ, aber assoziativ und distributiv.

Ähnlich wie die Matrizenmultiplikation bezüglich der Transposition (siehe Gleichung 2.14) verhält sich auch die Quaternionenmultiplikation bezüglich der Konjugation distributiv unter Vertauschung der Reihenfolge:

$$\overline{(q_1 \cdot q_2)} = \overline{q_2} \cdot \overline{q_1} \tag{2.45}$$

Wenn man eine Quaternion mit ihrer konjugierten Quaternion multipliziert

$$\begin{aligned} q \cdot \overline{q} &= [w;\ \vec{v}] \cdot [w;\ -\vec{v}] \\ &= [w \cdot w - \vec{v} \cdot (-\vec{v});\ \vec{v} \times (-\vec{v}) + w \cdot (-\vec{v}) + w \cdot \vec{v}], \end{aligned} \tag{2.46}$$

fällt auf, dass der gesamte Imaginärteil des Ergebnisses Null wird. Übrig bleibt, wie auch bei den komplexen Zahlen, eine reelle Zahl, die dem Quadrat des Betrags der Quaternion entspricht:

$$\begin{aligned} q \cdot \overline{q} &= w \cdot w + \vec{v} \cdot \vec{v} \\ &= w^2 + x^2 + y^2 + z^2 \\ &= |q|^2 \end{aligned} \tag{2.47}$$

Die *Division* ist für Quaternionen nicht definiert, aber analog zu den Matrizen gibt es zu einer Quaternion q ein *inverses Element* q^{-1}, für das gilt:

$$q \cdot q^{-1} = q^{-1} \cdot q = 1 \tag{2.48}$$

Diese *Inverse* einer Quaternion q mit $|q| \neq 0$ ist als

$$q^{-1} = \frac{1}{|q|} \cdot \overline{q} \tag{2.49}$$

definiert. Für den speziellen Fall, dass eine Quaternion den Betrag 1 hat, es sich also um eine Einheitsquaternionen handelt, gilt daher:

$$q^{-1} = \overline{q} \tag{2.50}$$

3. Grundlagen der 3D-Grafik

Ging es im vorigen Kapitel um die grundlegenden mathematischen Konzepte, werde ich versuchen, in den folgenden Abschnitten einen Überblick über die Umsetzung der 3D-Grafik am Computer zu geben, bevor ich in Kapitel 4 und 5 genauer auf die dafür nötigen mathematischen Operationen und Algorithmen eingehen werde.

Rufen wir uns zunächst einmal das Ziel der 3D-Grafik in Erinnerung: das Erzeugen eines zweidimensionalen Bildes aus einer Beschreibung einer dreidimensionalen Umgebung (Szene). Daraus ergibt sich mehr oder weniger von selbst eine Unterteilung in zwei Bereiche, nämlich die digitale Repräsentation einer Szene, der sogenannten *Modellierung*, und die eigentlichen Berechnung des Bildes, der *Bildsynthese* (viel gebräuchlicher ist der englische Begriff *Rendering*).

Die Modellierung erfolgt in der Regel manuell (wenn man von Experimenten etwa mit fraktaler Geometrie absieht) und kann auf vielfältige Weisen geschehen, zum Beispiel durch Digitalisierung von realen Gegenständen durch einen 3D-Scanner oder durch händische Erstellung des Modelles mittels Modellierungssoftware am Computer. Diese vorbereiteten Daten sind die Ausgangsbasis für das Rendering, das dann automatisch abläuft.

Grundsätzlich ist zwischen zwei großen Einsatzbereichen der 3D-Grafik zu unterscheiden. Der eine ist die unmittelbar durch den Benutzer gesteuerte Erzeugung von Ansichten einer 3D-Szene in Echtzeit, zum Beispiel in einem Computerspiel oder einem CAD-Programm. Der andere ist die Berechnung mehr oder weniger realistischer Bilder, beispielsweise in einem Animationsfilm oder zur Architekturvisualisierung.

Den beiden Einsatzbereichen gemein ist die stete Forderung nach Effizienz. Bei der interaktiven Anwendung muss eine gewisse Anzahl an Bildern pro Zeitintervall (typischerweise 20–30 Bilder pro Sekunde) mit einer begrenzten Rechenleistung erzeugt werden, damit die Darstellung für das Auge flüssig wirkt und die Eingaben des Benutzers ohne merkbare Verzögerung umgesetzt werden. Bei der Berechnung eines realistischen Bildes wird zwar im Normalfall viel mehr Rechenaufwand pro Bild akzeptiert, aber auch hier wird versucht, die benötigte Zeit möglichst gering zu halten, da so Korrekturen leichter möglich sind, aber auch schlicht um den Aufwand und somit die Kosten zu senken.

Deswegen wird immer nach Verfahren gesucht, die mit möglichst geringem Rechenaufwand und Speicherplatzbedarf die Ergebnisse aufwändiger Berechnungen möglichst gut annähern. Diese Verfahren müssen mit den realen Phänomen nicht mehr unbedingt viel zu tun haben, so lange das Ergebnis echt genug erscheint, um das Auge zu täuschen.

Die in diesem Kapitel geschilderten Grundlagen sind in einschlägiger Fachliteratur nachzulesen, im Besonderen seien die im Anhang »Verwendete Literatur« gelisteten Werke »3D-Spiele mit C++ und DirectX« (Rudolph 2003) und »3D-Spiele-Programmierung« (Zerbst u. a. 2004) erwähnt.

3.1. Modellierung

Wie schon erwähnt, ist das Ziel der Modellierungsphase, eine digitale Repräsentation von (dreidimensionalen) Objekten zu erstellen, anhand welcher später ein Bild berechnet werden kann.

Hier stößt man schnell auf das Problem, dass die realen phyikalischen Phänomene, die das Erscheinungsbild eines Objektes verursachen, viel zu komplex sind, um exakt nachgebildet werden zu können, vor allem im Hinblick auf die begrenzte Rechenleistung, mit der anschließend die Abbildung erzeugt wird.

Das derzeit am weitesten verbreitete Verfahren, um dieser Komplexität Herr zu werden, beruht darauf, die Speicherung der Form des Objektes (also der Oberfläche) von der des Materials (also der Eigenschaften dieser Oberfläche) zu trennen. Dieses Verfahren, das als *Oberflächendarstellung* (engl. *Boundary Representation*) bezeichnet wird, geht allerdings davon aus, dass die Szene aus Objekten besteht, die eine definierte Oberfläche haben. Manche Dinge, insbesondere atmosphärische Erscheinungen wie Wolken, Nebel, etc. sind daher schwer darzustellen.

Außerdem liefern manche Verfahren ihre Ergebnisse prinzipbedingt in der Form eines *Voxelgitters*[1], also eines dreidimensionales Gitters, bei dem jedem Element ein Wert zugeordnet ist. Dazu zählen unter anderem viele bildgebende Verfahren wie Computertomographie, die in der Forschung und der Medizin verwendet werden. Diese Daten lassen sich naturgemäß leichter mit den Mitteln der *Voxelgrafik* darstellen[2].

3.1.1. Geometrie

Um die Frage zu klären, wie die Oberflächen am effizientesten beschrieben werden können, muss man wissen, wie diese Daten nachher weiterverarbeitet werden, im Falle der 3D-Grafik also gerendert werden – grundsätzlich sind nämlich viele Varianten denkbar. Die folgenden Aussagen bezüglich Geschwindigkeit gehen von der derzeit in der Echtzeit-3D-Grafik eingesetzen Technik aus, die im Weiteren besprochen wird. Für andere spezielle Rendering-Techniken können sich andere Repräsentationen als effizienter erweisen.

Ein Teil der Verfahren beruht direkt auf einem mathematischen Zusammenhang, zum Beispiel könnte man die Oberfläche einer Kugel (mit dem Mittelpunkt im Koordinatenursprung und dem Radius r) mit Hilfe der Gleichung $x^2 + y^2 + z^2 = r^2$ beschreiben. Sich nur auf mathematische Grundkörper zu beschränken, wäre für die Modellierung realistischer Objekte natürlich viel zu unflexibel, aber es beruhen auch viele andere Verfahren auf Beschreibungen in Gleichungsform. Ein im CAD-Bereich oft eingesetztes Verfahren ist beispielsweise die Darstellung als Rotationskörper einer Polynom- oder einer ähnlichen Funktion[3].

[1] Voxel: Kofferwort aus »Volumen« und »Pixel«, bezeichnet analog zum zweidimensionalen Pixel ein Element eines regelmäßigen dreidimensionalen Gitters.

[2] Voxelgrafik bietet noch einige andere Vorteile, wenn es um die Berechnung von fotorealistischen Grafiken geht. So muss man die Zahl der Dreiecke sehr stark erhöhen, wenn man mit den Mitteln der Dreiecksgrafik sehr feine geometrische Strukturen darstellen will. Neben den damit verbundenen Leistungseinbußen treten einige störende Phänomene wie Flimmern auf, wenn ein großer Teil der Dreiecke am Bildschirm kleiner als ein Pixel ist. Da Voxelgrafik diese Probleme umgeht und für detailreiche Szenen nützliche Optimierungsansätze ermöglicht, wird seit kurzem wieder verstärkt damit experimentiert. (vgl. Bertuch 2009)

[3] Polynomfunktionen haben den Nachteil, dass sie bei höherer Ordnung schnell störend schwingen (Runges Phänomen). Abhilfe schaffen andere Interpolationsverfahren wie *Splines*. (vgl. Quarteroni und Saleri 2006, 73)

Diese Darstellungen erlauben zwar eine mathematisch exakte Nachbildung vieler verschiedener Oberflächen – auch gekrümmte Oberflächen können später aus diesen Daten in beliebiger Genaugkeit bereichnet werden –, aber sie haben gemein, dass die Berechnungen, um aus ihnen Bilder zu erzeugen, für Echtzeitanwendungen (noch) zu aufwändig sind.

Stattdessen wird auf ein wesentlich einfacheres Verfahren zurückgegriffen, nämlich die Darstellung als *Polygonnetz*. Die Oberfläche eines Objektes wird also durch Eckpunkte definiert, die jeweils zu einem Polygon gehören. Diese Punkte werden auch als als *Vertex* (pl. *Vertices*) bezeichnet. Jeder Vertex umfasst zumindest einen Ortsvektor, der seine Position im Raum angibt. Wie in den nächsten Abschnitten erläutert, können einem Vertex aber durchaus noch mehr Daten zugeordnet sein.

Meist werden nur Polygone mit drei Eckpunkten, also *Dreiecke* verwendet, da sich für diesen Sonderfall viele Algorithmen zur Bildberechnung stark vereinfachen lassen (beispielsweise ist ein Dreieck nie konkav, wodurch sich das Füllen der Bereiche am Bildschirm stark vereinfacht).

Das Polygonnetz kann sowohl direkt in einem entsprechenden Modellierungsprogramm erzeugt werden, als auch aus anderen Darstellungen (zum Beispiel den oben angesprochenen) errechnet werden.

Das Ergebnis kann bereits als so gennantes *Drahtgittermodell* angezeigt werden, auf Englisch wird diese Darstellungsart *Wireframe* genannt. Dabei werden die Vertices einfach durch Linien miteinander verbunden.

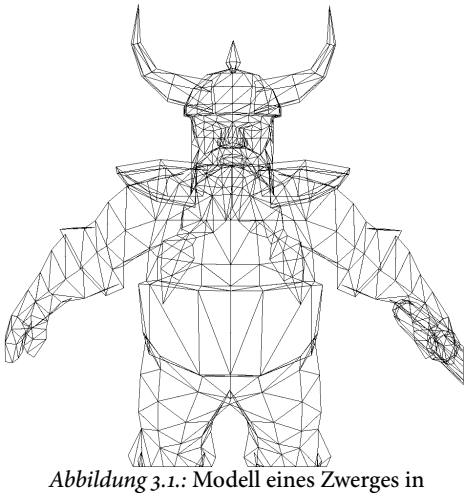

Abbildung 3.1.: Modell eines Zwerges in Wireframe-Darstellung (nur Vorderseiten).

3.1.2. Oberflächeneigenschaften

Strenggenommen könnten die Flächen des Modells bereits jetzt gefüllt dargestellt werden. Es fehlen allerdings noch jegliche Informationen über das Material des Objekts. Für die Berechnung eines Bildes sind natürlich in erster Linie die *Reflexionseigenschaften* des Objektes interessant, ist es doch das reflektierte Licht, welches das Bild entstehen lässt.

In der Materialdefinition der Objekte werden daher Informationen über das für das Material typische Reflexionsverhalten gespeichert. Diese Parameter beeinflussen zum Beispiel das Aussehen der Glanzlichter – auf einer polierten Metallkugel ist die direkte (spekulare) Reflektion der Lichtquelle deutlich abgegrenzt zu erkennen, während eine rauhe Plastikkugel hauptsächlich diffuse Reflexionen zeigen wird. Daneben sind noch einige andere Parameter für Spezialeffekte denkbar, etwa für sogenannte selbstleuchtende Materialien, die auch in der Dunkelheit sichtbar sind.

Zusätzlich werden jedem Vertex zwei Eigenschaften zugeordnet: die *Farbe* des Eckpunkts und sein *Normalvektor*. Aus diesen Werten, den genannten Materialeigenschaften und den Informationen über die in der Szene vorhandenen Lichtquellen wird der Farbwert des Dreiecks an bestimmten Stellen (und damit die Farbe der Pixel, die von dem Dreieck abgedeckt werden) berechnet (genaueres in Abschnitt 3.2.2).

Mittlerweile sollte sich das mathematische Unterbewusstsein schon zu Wort gemeldet haben – auf einen Vertex, also einen einzelnen Ortsvektor, lässt sich natürlich keine Normale bilden. Der Grund für diese seltsame Konstruktion liegt in der Verwendung der Normalvektoren für die Lichtberechnung. Zunächst würde man annehmen, dass die Normalvektoren der Eckpunkte eines Polygons ohnehin in die gleiche Richtung zeigen würden. Dies ist grundsätzlich richtig, und mit einer Normale pro Dreieck kann man die Lichtberechnungen auch schon ausführen. Diese Berechnungsart wird als *Flat Shading* bezeichnet.

Abbildung 3.2.: Flat Shading.

Der Nachteil dieser Methode wird deutlich, wenn man bedenkt, dass ja aus Effizienzgründen auch alle runden Körper mit Dreiecksnetz angenähert werden. Bei Verwendung von Flat Shading ergeben sich dann hässliche Kanten zwischen den Flächen, wie in Abbildung 3.2 zu sehen.

Um dieses Problem zu umgehen, wird nun jedem Vertex ein Normalvektor zugeordnet. Für harte Kanten entspricht dieser weiterhin der Flächennormalen des Dreiecks. Für weiche Kanten hingegen wird dieser Vektor als Mittelwert der Normalvektoren aller angenzenden Flächen berechnet[4]. Bei der Helligkeitsberechnung der Pixel, die zu diesem Dreieck gehören, wird dann linear zwischen den Helligkeitswerten der Eckpunkte (*Gouraud Shading*, Abbildung 3.3) oder gleich deren Normalvektoren (*Phong Shading*) interpoliert.

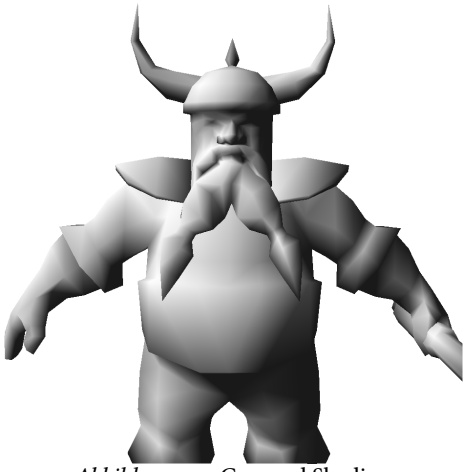

Abbildung 3.3.: Gouraud Shading.

Eine weitere wichtige Materialeigenschaft ist die Transparenz. Dazu ist noch anzumerken, dass eine physikalisch korrekte Simulation der Lichtbrechung und der dadurch hervorgerufenen Phänomene für Echtzeitanwendungen viel zu rechenaufwändig wäre. Statdessen wird im Normalfall das Objekt einfach durchsichtig gezeichnet, die Farbe des dahinter befindlichen Objekts also nicht vollständig überdeckt.

3.1.3. Texturen

Bis jetzt sind alle Informationen, also Farbe, etc. in den Vertices gespeichert. Dadurch sind die Daten zwar praktisch handzuhaben, aber es gibt ein Problem: Für die fotorealistische Darstellung eines Objektes würden viel zu viele Eckpunkte benötigt, um alle Details abzubilden. Beispielweise lässt sich die Form einer typischen Holztür recht gut mit einem einfachen Quader darstellen. Die feinen Strukturen des Holzes mit Geometrie darzustellen, wäre aber bereits um mehrere Größenordnungen zu aufwändig. Genauso verhält es sich mit vielen anderen Oberflächen, von polierten Steinen bis zu bedruckten Metallschildern, von Plakaten bis hin zu den Gebrauchs- und Schmutzspuren, die Abbildungen von Gegenständen erst richtig realistisch aussehen lassen.

[4] Die Interpolation der Normalvektoren aus denen der angrenzenden Flächen ist die einfachste Möglichkeit. Natürlich können die Normalvektoren der Vertices auch anders generiert werden, beispielsweise von Hand durch den Künstler.

Die Antwort der 3D-Grafik auf diese Probleme sind die sogenannten *Texturen*. Dabei handelt es sich um Bilder, die gleichsam wie eine Tapete auf die Geometrie der Objekte »geklebt« werden. Dazu werden in den Vertices die (zweidimensionalen) *Texturkoordinaten* gespeichert, die dem Vertex eine bestimmte Position auf der Textur, ein sogenanntes *Texel*[5], zuordnen. Beim Zeichnen eines Dreiecks werden dann für jedes Pixel die Texturkoordinaten aus denen der Eckpunkte interpoliert und der Wert der Textur an dieser Stelle ausgelesen.

Im einfachsten und häufigsten Fall handelt es sich dabei um die Farbe des Objektes (siehe Abbildung 3.4). Texturen können aber auch zur Modifikation vieler anderer Parameter eingesetzt werden, beispielsweise als *Alpha Map*, die die Transparenz des Objektes steuert, oder als *Normal Map* für die Flächennormalen, um eine rauhe Oberfläche zu simulieren.

Abbildung 3.4.: Diffuse Textur kombiniert mit Gouraud Shading.

Bei allen Arten von Texturen (mit Ausnahme einiger moderner Verfahren wie *Parallax Occlusion Mapping*, vgl. Brawley und Tatarchuk 2004, 135 ff.) muss aber beachtet werden, dass sie die eigentliche Geometrie eines Objektes nicht verändern und damit Vorgänge wie Schatten- und Verdeckungsberechnung nicht beeinflussen. Auch wenn etwa durch eine Normal Map eine strukturierte Oberfläche simuliert wird, erscheint der Rand des Objektes also als gerade Linie, was schnell unrealistisch wirken kann.

3.2. Rendering

Beim Rendering wird aus der Beschreibung einer Szene eine zweidimensionale Abbildung, ein »gerendertes« Bild erzeugt. Zu dieser Beschreibung gehören natürlich die einzelnen Objekte, aber auch Daten über die in der Szene vorhandenen Lichtquellen und die Position des *virtuellen Betrachters*, im Fachjargon auch einfach *Kamera* genannt, »durch dessen Auge« die Szene betrachtet wird.

Wie oben schon erwähnt, sind grundsätzlich viele Verfahren denkbar, um das Bild zu berechnen. In Anlehnung an die physikalischen Vorgänge in der Realität könnte man zum Beispiel die Photonen, die von den Lichtquellen ausgesendet werden, verfolgen, bis sie absorbiert werden oder auf die virtuelle Kamera treffen. Mit diesem Verfahren wurde tatsächlich unter dem Namen *Photon Tracing* (wörtlich »Photonenverfolgung«) experimentiert (vgl. etwa Wikipedia 2009a). Es erlaubt eine hochrealistische Darstellung der Szene, ist aber für den praktischen Einsatz zu rechenaufwändig, weshalb zu diesem Thema auch kaum wissenschaftliche Publikationen zu finden sind.

Die Idee der Nachverfolung von Strahlen wird von einem anderen Berechnungsverfahren aufgegriffen, das als *Ray Tracing* (wörtlich »Strahlverfolgung«) bezeichnet wird. Hier wird aber die Richtung umgekehrt – statt von den Lichtquelle, wird für jedes Pixel des Bildes ein Strahl von der Kamera emittiert. Wenn dieser Strahl auf ein Objekt trifft, dann werden vom Schnittpunkt neue Strahlen ausgesendet, um Refraktion und Reflexion zu berechnen und zu bestimmen, ob der Punkt von einer Lichtquelle beleuchtet wird oder im Schatten liegt. Da dieses Verfahren je nach Rechenaufwand sehr realistische Ergebnisse liefern kann

[5] An sich gibt es keinen Unterschied zwischen Pixel und Texel, der Begriff dient nur zur Verdeutlichung, dass auf eine Textur Bezug genommen wird und nicht etwa auf ein Pixel im Ergebnisbild.

(beispielweise können Phänomene wie Lichtbrechung optisch korrekt simuliert werden), wird es gerne in Situationen verwendet, in denen mehr Rechenleistung zu Verfügung steht und die Bildqualität wichtig ist, beispielweise für Filme.

Für den Einsatz in Echtzeitsituationen ist aber auch Ray Tracing (derzeit) nicht schnell genug. Deswegen wird auf ein anderes Verfahren zurückgegriffen, das als *Rasterisierung* oder *Scanline Rendering* bezeichnet wird. Dahinter verbirgt sich folgende Idee: Die Abbildung der Szene aus Sicht der virtuellen Kamera kann als Koordinatentransformation aller Objekte in das Koordinatensystem des Bildschirms betrachtet werden. Wenn diese Transformation auf alle Vertices in der Szene angewendet wird, kennt man die Position alle Dreiecke am Bildschirm, man braucht diese also nur noch zu füllen. Dieser Prozess wird als *Rendering-* oder *Grafikpipeline* bezeichnet und wird in der Computergrafik von speziellen Bausteinen übernommen, den Grafikkarten.

Die Grafikpipeline lässt sich in zwei Abschnitte aufteilen: Zuerst erfolgen die Koordinatentransformationen und einige andere Berechnungen, die auf der Ebene der Eckpunkte ablaufen, sich also der Vektoralgebra bedienen. Danach wird die Farbe der einzelnen Pixel des Ergebnisbildes (das im Normalfall am Bildschirm präsentiert wird) bestimmt, die Berechnungen laufen also auf Pixel-Ebene ab.

In den folgenden beiden Abschnitten werde ich einen Überblick über die Operationen geben, die in der Grafikpipeline stattfinden. Nach einer kurzen Betrachtung der computertechnischen Umsetzung (Abschnitt 3.3) folgt in Kapitel 4 und 5 eine detaillierte Besprechung dieser Operationen aus mathematischer Sicht.

3.2.1. Koordinatentransformationen

Wie schon in Abschnitt 3.1.1 besprochen, wird jeder Körper aus Dreiecken zusammengesetzt, deren Eckpunkte durch Ortsvektoren angegeben werden. Die Koordinaten dieser Vektoren beziehen sich auf das *Modellkoordinatensystem*, das nur für ein Objekt Gültigkeit hat. Ursprung und Einheiten dieses Systems können von dem Künstler, der die Modelle erstellt, im Prinzip beliebig gewählt werden[6].

Der erste Schritt der Verarbeitung besteht nun darin, die Koordinaten in das *Weltkoordinatensystem* zu übertragen, das eine Bezugsbasis für die gesamte Szene herstellt. Dafür sind normalerweise drei Operationen notwendig, nämlich Translation (um das Objekt an eine bestimmte Stelle zu bewegen), Skalierung (um das Objekt in die richtige Größe zu bringen), und Rotation (um die Ausrichtung des Objekts zu korrigieren). Die Matrix, die diese Transformation beinhaltet, wird *World Matrix* (Weltmatrix) genannt. Im Weltkoordinatensystem sind auch alle anderen Objekte der Szene definiert, etwa die Positionen der Lichtquellen oder der Standpunkt der virtuellen Kamera.

Im nächsten Schritt wird die Szene so transformiert, dass sich die Kamera im Koordinatenursprung befindet und in Richtung der z-Achse blickt. Die Matrix, die die Weltkoordinaten in dieses *Kamerakoordinatensystem* transformiert, wird als *View Matrix* (Sichtmatrix) bezeichnet. Sinn dieser Transformation ist es, die nachfolgenden Operationen zu vereinfachen. An dieser Stelle werden auch die Berechnungen ausgeführt, auf die später beim Füllen der Dreiecke zurückgegriffen wird, zum Beispiel Lichtberechnungen auf Vertex-Ebene (mehr dazu in Abschnitt 3.2.2).

[6] In der Praxis wird man natürlich versuchen, sinnvolle Werte zu wählen, also beispielsweise den Ursprung in die Mitte des Objektes legen und eine Einheit als einen Zentimeter interpretieren.

Als nächstes folgt das »Herzstück« der 3D-Grafik, die *Projektion* durch die *Projection Matrix*. Diese Operation bildet den gesamten sichtbaren Raum in das *kanonische Sichtvolumen* ab. Dabei handelt es sich um einen mathematisch einfachen Körper, in DirectX (siehe Abschnitt 3.3) beispielsweise in einen Quader zwischen den Punkten $\begin{pmatrix}-1 & -1 & 0\end{pmatrix}^T$ und $\begin{pmatrix}1 & 1 & 1\end{pmatrix}^T$. Diese Abbildung muss zwei Anforderungen erfüllen: Zum einen muss die x- und die y-Koordinate der Vertices im kanonischen Sichtvolumen schon ihrer endgültigen projizierten Position entsprechen, die projizierte Position muss sich also durch eine einfache orthogonale Parallelprojektion, durch ein »Weglassen« der z-Koordinate ergeben. Zum anderen muss die z-Reihenfolge der Vertices erhalten bleiben. Wenn die z-Koordinate eines Vektors steigt, muss also auch die z-Koordinate des transformierten Vektors (streng monoton) steigen. Dies ist die Voraussetzung für die Sichtbarkeits- oder Verdeckungsberechnung, die später stattfindet.

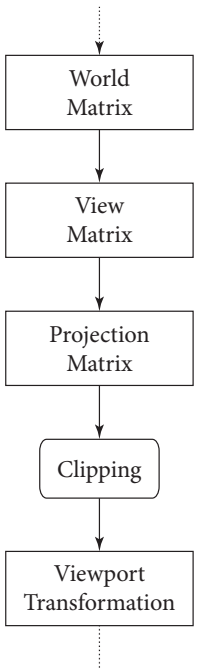

Abbildung 3.5.: Koordinatentransformationen in der Grafikpipeline.

Die Verwendung eines kanonischen Sichtvolumens hat den Vorteil, dass die Art der Projektion keinen Einfluss auf die nachfolgenden Verarbeitungsschritte hat. Unabhängig von der Art der Projektion lassen sich für diese also die gleichen Algorithmen einsetzen, wodurch sich die Umsetzung in Hardware viel effizienter gestalten lässt.

Nach der Projektion werden alle Bereiche verworfen, die sich außerhalb des Sichtvolumens befinden. Dieser Vorgang wird *Clipping* genannt, die Koordinaten nach der Multiplikation mit der Projection Matrix werden daher auch als *Clipping-Koordinaten* bezeichnet.

Um die endgültige Position der Dreiecke zu erhalten, muss das Ergebnis der Projektion nur noch auf die Auflösung des Ausgabemediums »gestreckt« werden. Dieser Schritt wird als *Viewport Transformation* bezeichnet, die Eckpunkte der Dreiecke liegen danach in *Viewport-Koordinaten* vor. In den meisten Fällen hat dieses Koordinatensystem seinen Ursprung in der linken oberen Ecke des Bildschirms. Die x-Koordinaten wachsen dabei nach rechts, die y-Koordinaten nach unten.

Unter Anwendung der in Kapitel 2.2.3 gezeigten Rechenregel können World, View und Projection Matrix durch Multiplikation leicht in eine Matrix kombiniert werden. Dadurch ist nur mehr eine Matrizenmultiplikation nötig, um einen großen Teil der Transformationen zu erledigen, wodurch dieser Schritt sehr effizient erledigt werden kann.

3.2.2. Rasterung

Nach der Viewport Transformation sind die Koordinatentransformationen abgeschlossen. Nun gilt es, die Dreiecke am Bildschirm zu füllen. Der Speicherbereich, in den die Farbinformationen geschrieben werden, wird als *Framebuffer* bezeichnet.

Die Berechnung der Farbe an einer bestimmten Stelle eines Dreieckes ist aus mathematischer Sicht einfach, es muss nur linear zwischen den Farben der Eckpunkte interpoliert werden. Ähnlich verhält es sich auch mit den Texturen, die auf das Dreieck angewendet wurden – ausgehend von den Texturkoordinaten der Vertices wird die Stelle interpoliert, an der das Texturbild ausgelesen werden muss. Alle Farbwerte werden miteinander verrechnet und dann mit dem Helligkeitswert multipliziert, den das verwendete

Beleuchtungsmodell liefert. Das Ergebnis stellt den Farbwert eines Pixels dar, der dann in den Framebuffer geschrieben wird.

Dabei muss aber darauf geachtet werden, dass Dreiecke, die weiter vom Betrachter entfernt sind (deren z-Wert nach der Projektion also größer ist) von den Dreiecken überdeckt werden, die näher beim Betrachter liegen. Dieses sogenannte *Sichtbarkeitsproblem* ist keineswegs so trivial, wie es auf den ersten Blick vielleicht wirken mag[7]. Zur Lösung dieses Problems hat sich der *Z-Buffer-Algorithmus* durchgesetzt. Dabei wird ein sogenannter Z-Buffer angelegt, ein Speicherbereich mit der gleichen Auflösung wie der Framebuffer – zu jedem Pixel des Ergebnisbildes existiert also ein Wert im Z-Buffer. Zu Beginn der Berechnung werden alle Tiefenwerte auf den größtmöglichen Wert gesetzt, danach werden alle Dreiecke in (prinzipiell) willkürlicher Reihenfolge gerastert.

Für jedes Pixel wird nun überprüft, ob sein z-Wert kleiner ist als der Wert im Buffer, es also näher beim Betrachter ist, als alle vorherigen Pixel. Wenn dies der Fall ist, werden der neue Farbwert und die z-Koordinate in den Framebuffer beziehungsweise in den Z-Buffer geschrieben. Andernfalls wird das Pixel verworfen.

Nachdem diese Berechnungen für alle Pixel aller sichtbaren Dreiecke durchgeführt worden sind, befindet sich im Framebuffer eine perspektivisch korrekte Abbildung der Szene. Diese muss jetzt nur noch auf dem Bildschirm angezeigt oder in eine Datei geschrieben werden.

3.3. Umsetzung in Hardware

So gut wie alle der gerade beschriebenen Berechnungen der Grafik-Pipeline werden in einem modernen Computer von dem Prozessor auf der Grafikkarte erledigt. Der Aufbau dieses Grafikchips ist speziell auf die Pipeline abgestimmt.

Beispielsweise sind eigene optimierte Einheiten für die Transformationen der Vertices zuständig, die auf modernen Grafikkarten durch sogenannte *Vertex Shader* programmiert werden können. Durch die Verwendung von Matrizen lassen sich die meisten Berechnungen auf eine einzige Operation, nämlich die Multiplikation einer Matrix mit einem Vektor, zurückführen. Das ermöglicht es, viele Recheneinheiten in der Grafikkarte speziell für diese Operation zu optimieren, wodurch diese im Vergleich etwa zu einem typischen Hauptprozessor sehr schnell arbeiten.

Auch für die Berechnung der Pixelfarben gibt es solche spezielle Einheiten, die durch sogenannte *Fragment-* oder *Pixel-Shader* programmiert werden können. Generell müssen in der Grafikkarte die gleichen Operationen auf eine große Menge Daten angewendet werden. Die Gesamtleistung der Grafikkarte kann also einfach dadurch erhöht werden, dass man viele Einheiten vorsieht, die die Daten parallel abarbeiten[8].

Für den Zugriff auf die Grafikkarte haben sich in der Computerprogrammierung zwei große Industriestandards herausgebildet, das von Microsoft entwickelte *DirectX* und das von einem Industriekonsortium rund um den Initiator Silicon Graphics verwaltete *OpenGL*. Die meisten Unterschiede zwischen den beiden Standards sind rein technischer Natur und sind vor dem mathematischen Hintergrund dieser Arbeit nicht weiter relevant.

[7] Man bedenke, dass sich beispielsweise die »z-Bereiche« mehrerer Dreiecke überlappen können, weswegen eine einfache Sortierung nach aufsteigender z-Koordinate nicht mehr möglich ist.

[8] Die Ende 2008 auf den Markt gekommenen High-End-Modelle der großen Grafikkartenhersteller haben mehrere hundert solcher Einheiten, die parallel arbeiten. (vgl. Advanced Micro Devices, Inc. 2008; nVidia Corporation 2009)

Zwei der Eigenheiten wirken sich aber auch auf die mathematischen Betrachtungen aus. Erstens verwendet DirectX ein linkshändiges, OpenGL aber ein rechtshändiges Koordinatensystem. Bei DirectX führt die z-Achse bei wachsender Koordinate also in den Bildschirm hinein, während sie bei OpenGL aus dem Bildschirm herauszeigt. Zweitens, und dieser Punkt hat weitaus größere Auswirkungen, verwendet OpenGL die aus der Schulmathematik bekannten Spaltenvektoren, DirectX aber Zeilenvektoren. Wie schon in Abschnitt 2.2.4 behandelt, ergibt sich daraus, dass bei einer Kombination mehrerer Transformationen in Matrizenform die Multiplikationsreihenfolge umgekehrt werden muss. Um die Koordinatentransformationen der Grafikpipeline in eine Matrix zusammenfassen, würde man also in OpenGL $M_{ges} = M_{proj} \cdot M_{view} \cdot M_{world}$ rechnen, in DirectX aber $M_{ges} = M_{world} \cdot M_{view} \cdot M_{proj}$. Ich habe mich in dieser Arbeit für die OpenGL-Variante, also rechtshändige Koordinatensysteme und Spaltenvektoren, entschieden, da sie den Konventionen der Schulmathematik entspricht.

In einem weiteren wichtigen Punkt gleichen sich DirectX und OpenGL wieder sehr: den Techniken, um die Grafikpipeline im Hinblick auf größtmögliche Geschwindigkeit zu optimieren. Die Optimierung kann grundsätzlich auf zwei Ebenen erfolgen: Einerseits versucht man natürlich, die Berechnungen selbst möglichst schnell zu machen. Andererseits versucht man aber auch, unnötige Berechnungen schon von vornherein zu vermeiden.

Unter den ersten Punkt fallen beispielsweise alle Techniken, die den Eigenheiten der Grafikhardware Rechnung tragen. Zum Beispiel werden auf einem Computer gewisse Operationen prinzipbedingt sehr schnell ausgeführt, etwa Addition, Subtraktion, und Multiplikation. Andere Operationen sind wesentlich »teurer« und werden deshalb nach Möglichkeit vermieden. Dazu zählen insbesondere die Division und Operationen wie das Radizieren und die Berechnung transzendenter Funktionen, die der Prozessor annähern muss. Es wird auch versucht, die eingesetzten Algorithmen so zu modifizieren, dass kompliziert zu berechnende Werte nach Möglichkeit zwischengespeichert und mehrfach verwendet werden können.

Der zweite Punkt umfasst alle Techniken, die versuchen, aus der gesamten Menge der Dreiecke schon möglichst bald die Teile auszusortieren, die im fertigen Bild ohnehin nicht sichtbar sind. Beispielsweise könnten in einem Computerspiel, das in einem Bürokomplex spielt, schon von vornherein große Bereiche der Szene ausgeschlossen werden, weil sie sowieso von den Wänden des Raumes, in dem sich die Kamera gerade befindet, verdeckt werden. Aber auch ohne solche speziellen Annahmen kann typischerweise schon ein großer Teil der Geometrie verworfen werden, bevor er das Ende der Grafikpipeline erreicht. Beispielsweise sind Gegenstände, die sich hinter der Kamera befinden, nie sichtbar und müssen gar nicht erst transformiert werden. Genauso können auch alle von der Kamera abgewandten Polygone eines Objektes verworfen werden, solange das Objekt nicht transparent ist. Die letzgenannte Technik ist unter dem Namen *Backface Culling* bekannt.

Nach diesem kurzen Exkurs auf die computertechnische Seite der 3D-Grafik wenden wir uns im nächsten Kapitel nun wieder dem mathematischen Teil zu.

4. Objekt-Transformationen

Das Thema dieses Kapitels sind die Transformationen, welche auf die Objekte im dreidimensionalen Raum angewendet werden können und dabei direkte Entsprechungen in der realen Welt haben. In Bezug auf die im vorigen Kapitel beschriebene Grafikpipeline werden diese normalerweise beim Übergang vom Modellkoordinatensystem in das Weltkoordinatensystem, also in der World Matrix, angewendet (unter leicht anderen Vorzeichen auch in der View Matrix, mehr dazu in Kapitel 5.1). Im Speziellen soll es natürlich um die Darstellung dieser Transformationen als Matrix gehen.

Neben den im Weiteren behandelten Transformationen, nämlich Skalierung, Translation und Rotation, sind in der Geometrie noch einige weitere Transformationen bekannt, diese sind aber für die 3D-Grafik kaum relevant – nicht zuletzt deshalb, weil sie in der realen Welt auch nicht so häufig vorkommen.

In der 3D-Grafik ist es oft erforderlich, bei der Transformation eines Objektes auch die Normalvektoren der Vertices zu transformieren. Dafür reicht es aber nicht aus, einfach die Transformationsmatrix auch auf die Normalvektoren anzuwenden. Es würden dann unter anderem alle Transformationen Probleme bereiten, die nicht winkeltreu sind. Stattdessen müssen die Normalvektoren gesondert behandelt werden:

Es sei $\vec{v}$ ein Vektor zwischen zwei Punkten auf einer Ebene mit dem Normalvektor $\vec{n}$. Offensichtlich gilt der Zusammenhang

$$\vec{n} \cdot \vec{v} = 0, \tag{4.1}$$

den man auch so anschreiben kann

$$\vec{n}^T \cdot \vec{v} = 0, \tag{4.2}$$

wenn man die Vektoren als Matrizen auffasst. (vgl. Vornberger und Fox 2006, 155) Mit einer invertierbaren Matrix M gilt natürlich auch

$$\vec{n}^T \cdot M^{-1} \cdot M \cdot \vec{v} = 0. \tag{4.3}$$

M entspricht dabei einer beliebigen auf den Vektor $\vec{v}$ angewendeten Transformation, aus welcher der Vektor $\vec{v'}$ hervorgeht:

$$\left(\vec{n}^T \cdot M^{-1}\right) \cdot \vec{v'} = 0. \tag{4.4}$$

Der Normalvektor der transformierten Ebene, $\vec{n'}$, muss natürlich normal zu $\vec{v'}$ sein:

$$\vec{n'}^T \cdot \vec{v'} = 0. \tag{4.5}$$

Aus Gleichung 4.4 erhält man durch zweimaliges Transponieren (mit Hilfe des Zusammenhangs aus Gleichung 2.14)

$$\left(\left(M^{-1}\right)^T \cdot \vec{n}\right)^T \cdot \vec{v'} = 0. \tag{4.6}$$

Kombiniert man die beiden letztgenannten Zusammenhänge, erhält man die Gleichung

$$\left(\left(M^{-1}\right)^T \cdot \vec{n}\right)^T \cdot \vec{v'} = \vec{n'}^T \cdot \vec{v'}. \tag{4.7}$$

die sich leicht zu

$$\left(M^{-1}\right)^T \cdot \vec{n} = \vec{n'} \tag{4.8}$$

vereinfachen lässt.

Der Normalvektor einer Fläche wird also transformiert, indem man ihn mit der *transponierten Inversen der Transformationsmatrix* multipliziert. Die Länge des Vektors ist in den obigen Bedingungen allerdings nicht enthalten, wird für die weiteren Berechnungen ein Einheitsvektor gebraucht, muss das Ergebnis also noch normalisiert werden.

4.1. Skalierung

Die erste und einfachste Transformation, die ich hier vorstellen möchte, ist die Skalierung.

Um ein Objekt zu skalieren, multipliziert man die Koordinaten seiner Eckpunkte jeweils mit einem Skalierungsfaktor λ_i. Gesucht ist also eine Matrix S, für die gilt:

$$S \cdot \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x \cdot \lambda_x \\ y \cdot \lambda_y \\ z \cdot \lambda_z \\ w \end{pmatrix} \tag{4.9}$$

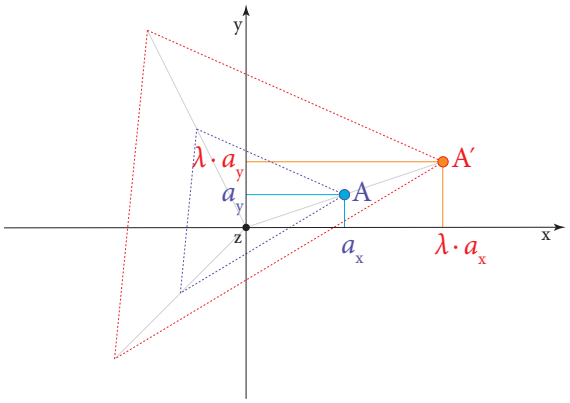

Abbildung 4.1.: Isotrope Skalierung eines Dreiecks um den Faktor λ.

Wenn man die Teilgleichung der Multiplikation für eine Koordinate aufstellt (in diesem Fall x), erkennt man recht schnell, wie die dazugehörige Zeile der Matrix aussehen muss:

$$\begin{aligned} x \cdot m_{11} + y \cdot m_{12} + z \cdot m_{13} + w \cdot m_{14} &= x \cdot \lambda_x \\ m_{11} = \lambda_x; m_{12} = 0; m_{13} = 0; m_{14} &= 0 \end{aligned} \tag{4.10}$$

Löst man auf die gleiche Weise auch die Teilgleichungen für die anderen Koordinaten, erhält man schließlich folgende Matrix für die *Skalierung um den Koordinatenursprung*:

$$S(\lambda_x, \lambda_y, \lambda_z) = \begin{pmatrix} \lambda_x & 0 & 0 & 0 \\ 0 & \lambda_y & 0 & 0 \\ 0 & 0 & \lambda_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.11}$$

Es handelt sich dabei also um eine Diagonalmatrix mit den Skalierungsfaktoren der einzelnen Koordinaten in der Hauptdiagonale (man könnte sie auch als Skalierungsfaktoren der Einheiten der Koordinatenachsen betrachten).

Wenn die Skalierungsfaktoren aller drei Achsen gleich sind, spricht man von einer gleichförmigen oder *isotropen* Skalierung, andernfalls von einer ungleichförmigen oder *antisotropen* Skalierung. Da sich

bei einer isotropen Skalierung die Achsen relativ zueinander nicht ändern, sondern auf dreidimensionale Geometrie bezogen nur die Größe des Objektes, wird die isotrope Skalierung zuweilen auch als *Maßstabsänderung* bezeichnet.

Haben alle drei Koeffizienten den Wert 1, hat die Skalierung keine Auswirkungen – wie sich leicht überprüfen lässt, ergibt sich aus Gleichung 4.11 die *Einheitsmatrix*.

Wie man leicht vermuten kann, ist die Skalierung umkehrbar (solange keiner der Faktoren 0 ist). Die inverse Skalierungsmatrix S^{-1} ist dabei die Skalierungsmatrix mit dem Kehrwert der Faktoren:

$$S^{-1}(\lambda_x, \lambda_y, \lambda_z) = \begin{pmatrix} \frac{1}{\lambda_x} & 0 & 0 & 0 \\ 0 & \frac{1}{\lambda_y} & 0 & 0 \\ 0 & 0 & \frac{1}{\lambda_z} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = S(\lambda_x^{-1}, \lambda_y^{-1}, \lambda_z^{-1}) \tag{4.12}$$

In manchen Situationen möchte man ein Objekt nicht um den Koordinatenursprung, sondern um einen anderen Punkt im Raum skalieren. In diesem Fall ist die einfachste Lösung, das Objekt zuerst mit einer Translationsmatrix in den Ursprung zu verschieben, die Skalierung durchzuführen und dann das Objekt wieder zurück zu verschieben. Die drei Operationen können in einer Matrix kombiniert werden.

Die *Spiegelung* kann als Sonderfall der Skalierung betrachtet werden. Dabei ist mindestens ein Koeffizient negativ. Eine Spiegelung an der xy-Ebene lässt sich beispielsweise als Skalierung mit $\lambda_z = -1$ darstellen.

4.2. Translation

Eine weitere elementare Transformation ist die Translation, also die Verschiebung von Objekten im Raum.

Um alle Eckpunkte eines Objekts um den Vektor $\vec{v}$ zu verschieben, addiert man die beiden Vektoren einfach nach den Gesetzen der Vektorrechnung komponentenweise. Die gesuchte Translationsmatrix T muss also die folgende Bedingung erfüllen:

$$T \cdot \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x + v_x \\ y + v_y \\ z + v_z \\ w \end{pmatrix} \tag{4.13}$$

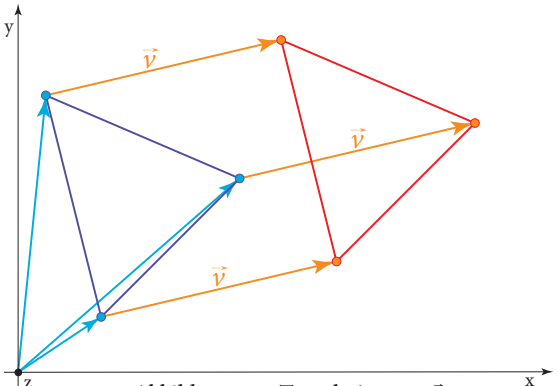

Abbildung 4.2.: Translation um $\vec{v}$.

Beim Aufstellen der Teilgleichungen (hier exemplarisch für die y-Koordinate) wird aber bald ein

Problem deutlich:

$$\begin{aligned} x \cdot m_{21} + y \cdot m_{22} + z \cdot m_{23} + w \cdot m_{24} &= y + v_y \\ m_{21} = 0 \quad m_{23} = 0 \quad m_{24} &= 0 \\ y \cdot m_{22} &= y + v_y \\ m_{22} &= \frac{y + v_y}{y} \\ &= 1 + \frac{v_y}{y} \end{aligned} \tag{4.14}$$

Anscheinend gibt es keinen Weg, um die für die Translation nötigen Koeffizienten zu berechnen, ohne die Koordinaten des Vektors zu kennen, der transformiert werden soll. Genau das ist aber Voraussetzung, um die Matrix für mehrere Vektoren verallgemeinern zu können – es handelt sich bei der Translation offensichtlich nicht um eine lineare Transformation[1]!

Der Ausweg aus diesem Dilemma liegt in der Verwendung von homogenen Koordinaten. w hat ja keinen beliebigen Wert, sondern (zumindest vor der Transformation) den Wert 1. Die dementsprechend geänderten Bedingungen für die Translationsmatrix lauten:

$$T \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + v_x \\ y + v_y \\ z + v_z \\ 1 \end{pmatrix} \tag{4.15}$$

Stellt man nun die Teilgleichung für die y-Koordinate auf, erhält man:

$$x \cdot m_{21} + y \cdot m_{22} + z \cdot m_{23} + 1 \cdot m_{24} = y + v_y \tag{4.16}$$

Diese Bedingung ist einfach erfüllt, indem man

$$m_{21} = 0; m_{22} = 1; m_{23} = 0; m_{24} = v_y \tag{4.17}$$

setzt.

Auf die gleiche Art kann man auch die anderen Teilgleichungen auflösen. Man erhält als Ergebnis die *Translationsmatrix*

$$T(\vec{v}) = \begin{pmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.18}$$

So wie die Skalierung ist auch die Translation einfach umkehrbar. Wenn man die inverse Matrix zu der Translationsmatrix $T(\vec{v})$ berechnet, erhält man als Ergebnis

$$T^{-1}(\vec{v}) = \begin{pmatrix} 1 & 0 & 0 & -v_x \\ 0 & 1 & 0 & -v_y \\ 0 & 0 & 1 & -v_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = T(-\vec{v}), \tag{4.19}$$

[1] Wie schon in Kapitel 2.2.3 erwähnt, kann jede lineare Transformation bzw. Abbildung zwischen zwei n-dimensionalen Vektorräumen als $n \times n$-Matrix dargestellt werden. Nähere Behandlungen dieses Zusammenhangs finden sich in einschlägiger Fachliteratur.

also auf die Verschiebung um einen Vektor gleicher Länge in die entgegengesetzte Richtung.

Wie sich leicht zeigen lässt, kann man, um mehrere Translationen nacheinander auf einen Vektor anzuwenden, einfach die einzelnen Verschiebungsvektoren addieren und so eine gemeinsame Matrix erzeugen:

$$T(\vec{a}) \cdot T(\vec{b}) \cdot \vec{p} = \begin{pmatrix} 1 & 0 & 0 & a_x + b_x \\ 0 & 1 & 0 & a_y + b_y \\ 0 & 0 & 1 & a_z + b_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \vec{p} = T(\vec{a} + \vec{b}) \cdot \vec{p} \tag{4.20}$$

Dieser Zusammenhang mag trivial erscheinen, aber aus Gleichung 4.20 ergibt sich noch eine weitere interessante Eigenheit von Translationsmatrizen: Nachdem die Vektoraddition kommutativ ist, mit der man die Translationen zusammenfassen kann, ist auch die Multiplikation von Translationsmatrizen *kommutativ*.

4.3. Rotation

Die Herleitung der dritten und letzten der hier behandelten Transformationen, der Rotation, ist etwas schwieriger als die vorangegangenen. Am einfachsten ist es, sich zunächst auf die Rotation um eine der Koordinatenachsen zu beschränken, im Folgenden wollen wir zunächst die *Rotation um die z-Achse*, also in der *xy-Ebene* betrachten.

4.3.1. Rotation um die Koordinatenachsen

Abbildung 4.3 zeigt einen Vektor $\vec{p} = (x \quad y \quad z)^T$. Nach der Rotation um den Winkel γ um die z-Achse ergibt sich der Vektor $\vec{p}' = (x' \quad y' \quad z')^T$.

Da die Rotation in der xy-Ebene stattfindet, ändert sich die z-Koordinate des Vektors nicht. Es gilt also

$$z' = z. \tag{4.21}$$

Für die x- und y-Koordinaten kann man über die Definition von Sinus und Cosinus ebenfalls direkt aus der Skizze die folgenden Zusammenhänge ableiten:

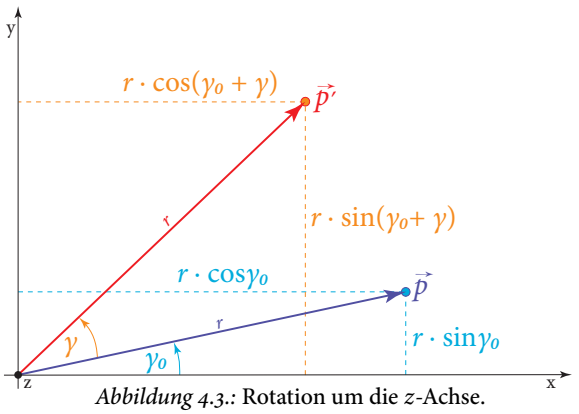

Abbildung 4.3.: Rotation um die z-Achse.

$$x = r \cdot \cos\gamma_0 \qquad y = r \cdot \sin\gamma_0 \tag{4.22}$$

$$x' = r \cdot \cos(\gamma_0 + \gamma) \qquad y' = r \cdot \sin(\gamma_0 + \gamma) \tag{4.23}$$

Unter Zuhilfenahme der trigonometrischen Additionstheoreme erhält man für x' und y' aus 4.23 nach Ausmultiplizieren:

$$x' = r \cdot \cos\gamma_0 \cos\gamma - r \cdot \sin\gamma_0 \sin\gamma \qquad y' = r \cdot \sin\gamma_0 \cos\gamma + r \cdot \cos\gamma_0 \sin\gamma \tag{4.24}$$

Die Gleichungen 4.22 ergeben umgeformt:

$$\cos\gamma_0 = \frac{x}{r} \qquad\qquad \sin\gamma_0 = \frac{y}{r} \tag{4.25}$$

Setzt man 4.25 in 4.24 ein, ergibt sich schließlich:

$$x' = x \cdot \cos\gamma - y\sin\gamma \qquad\qquad y' = y\cos\gamma + x\sin\gamma \tag{4.26}$$

Die gesuchte Rotationsmatrix R_z muss also die folgende Bedingung erfüllen:

$$R_z \cdot \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x\cos\gamma - y\sin\gamma \\ x\sin\gamma + y\cos\gamma \\ z \\ w \end{pmatrix} \tag{4.27}$$

Nun ist es ein leichtes, wie für die anderen Transformationen auch die passende Rotationsmatrix für die *Rotation um die z-Achse* herzuleiten:

$$R_z(\gamma) = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.28}$$

Analog lassen sich auch die Matrizen für die Rotation um die anderen beiden Achsen aufstellen. Man erhält als Ergebnis für die *Rotation um die x-Achse*

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.29}$$

und für die *Rotation um die y-Achse*

$$R_y(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.30}$$

Jede Rotation lässt sich selbstverständlich durch nochmalige Rotation um den gleichen Betrag in die andere Richtung umkehren. Beim Aufstellen der inversen Matrix

$$R_z^{-1}(\gamma) = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = R_z(-\gamma) \tag{4.31}$$

fällt auf, dass die Inverse der Rotationsmatrix gleich ihrer Transponierten ist. Dies ist auch tatsächlich eine Eigenheit aller Rotationsmatrizen, beziehungsweise aller Transformationen, bei denen eine beliebige Länge erhalten bleibt. (vgl. Weisstein 2008)

4.3.2. Eulersche Winkel

Die gerade behandelten Rotationsmatrizen sind auf Drehungen um die Koordinatenachsen beschränkt, aber natürlich lassen sich mehrere der Matrizen kombinieren, um jede beliebige Rotation ausdrücken zu können.

Ein beliebtes Verfahren, um Ausrichtungen im Raum anzugeben, sind die sogenannten Eulerschen Winkel. Dabei handelt es sich um drei Rotationswinkel um jeweils eine Achse, durch die jede beliebige Rotation ausgedrückt werden kann. Es gibt mehrere Konventionen, in welcher Reihenfolge die Rotationen auf welche Achsen angewendet werden. Eine der am häufigsten benutzten sind die aus der Luftfahrt bekannten *Roll-Nick-Gier-Winkel* bezeichnet (engl. *roll, pitch, yaw*). Sie geben die Rotation als Kombination einer Rotation um die z-Achse (γ), gefolgt von einer Rotation um die y-Achse (β) und schließlich einer Rotation um die x-Achse (α) an.

Mit einem Wertebereich von $-\pi$ bis π für α und γ und von $-\frac{\pi}{2}$ bis $\frac{\pi}{2}$ für β lassen sich alle Ausrichtungen im Raum angeben. Bis auf zwei Ausnahmen (dazu gleich mehr) ist auch jedem Rotationszustand genau ein Satz an Winkeln zugeordnet, die Abbildung ist also bis auf diese Ausnahmen bijektiv.

Um eine Rotationsdarstellung in Eulerschen Winkeln in Matrixform zu überführen, genügt es, einfach die Rotationsmatrizen für die einzelnen Achsen miteinander zu multiplizieren:

$$
\begin{aligned}
R_{zyx}(\gamma,\beta,\alpha) &= R_x(\alpha)\cdot R_y(\beta)\cdot R_z(\gamma) \\
&= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} \cos\beta\cos\gamma & -\cos\beta\sin\gamma & \sin\beta & 0 \\ \cos\alpha\sin\gamma + \sin\alpha\sin\beta\cos\gamma & \cos\alpha\cos\gamma - \sin\alpha\sin\beta\sin\gamma & -\sin\alpha\cos\beta & 0 \\ \sin\alpha\sin\gamma - \cos\alpha\sin\beta\cos\gamma & \sin\alpha\cos\gamma + \cos\alpha\sin\beta\sin\gamma & \cos\alpha\cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
\end{aligned}
\tag{4.32}
$$

Eulersche Winkel sind ein vergleichbar einfaches und intuitives Mittel, um beliebige Rotationen anzugeben, und werden in der 3D-Grafik gerne für statische Angaben von Rotationen verwendet. Kommen jedoch Animationen, also Veränderungen des Rotationszustandes hinzu, haben Eulersche Winkel aber einige Nachteile:

Zum einen ist es schwer, zwischen zwei in Eulerschen Winkeln gegebenen Rotationszuständen so zu interpolieren, dass die Winkelgeschwindigkeit der resultierenden Rotationsbewegung konstant bleibt. Dies ist für manche Anwendungen wünschenswert, etwa Kamerafahrten, damit die Animation flüssig erscheint.

Zum anderen gibt es wie schon erwähnt für die Zuordnung von Eulerschen Winkeln zu Rotationen zwei kritische Punkte, im Falle der zyx-Konvention bei $\beta = \pm\frac{\pi}{2}$, also bei einer Rotation von 90° um die

y-Achse. Setzt man $-\frac{\pi}{2}$ in die Matrix ein, erhält man

$$\begin{aligned} R_{zyx}\left(\alpha, -\frac{\pi}{2}, \gamma\right) &= \begin{pmatrix} 0 & 0 & -1 & 0 \\ \cos\alpha\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\cos\gamma + \sin\alpha\sin\gamma & 0 & 0 \\ \sin\alpha\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\cos\gamma - \cos\alpha\sin\gamma & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 0 & -1 & 0 \\ -\sin(\alpha-\gamma) & \cos(\alpha-\gamma) & 0 & 0 \\ \cos(\alpha-\gamma) & \sin(\alpha-\gamma) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned} \tag{4.33}$$

Die Rotation hängt nun ausschließlich von der Differenz $\alpha - \gamma$ ab – das System hat einen Freiheitsgrad eingebüßt! Es gibt jetzt unendlich viele Kombinationen von α und γ, die zu der gleichen Ausrichtung im Raum führen; die Abbildung der Eulerschen Winkel auf die Rotationsmatrizen ist also nicht injektiv.

Dieses Problem tritt auch bei kardanischen Aufhängungen in der Mechanik auf, beispielsweise bei Trägheitsnavigationssystemen[2]. Es wird daher als *Gimbal Lock* bezeichnet (engl. *gimbal* bezeichnet einen Kardanring, zu Deutsch also etwa »Kardanring-Blockade«). In der Mechanik hat man oft die Möglichkeit, Situationen, die zu einem Gimbal Lock führen können, rechtzeitig zu erkennen und den Gimbal Lock zu vermeiden, etwa durch Änderung des Flugmanövers. Bei vom Benutzer ausgelösten oder gesteuerten Rotationen in Computerprogrammen ist dies aber kaum möglich, deswegen greift man normalerweise auf andere Verfahren zurück, um Rotationen zu speichern und zu verarbeiten.

4.3.3. Transformationsmatrix aus transformierten Basisvektoren

In den vorigen beiden Abschnitten haben wir die Konstruktion einer Rotationsmatrix ausgehend von einem oder mehreren Rotationswinkeln besprochen. In manchen Fällen kennt man aber bereits die *Basisvektoren* des neuen Koordinatensystems (in den Koordinaten des alten Systems) und will eine Matrix aufstellen, um andere Vektoren zu transformieren.

Hat man drei linear unabhängige Vektoren im alten und im neuen (dreidimensionalen) Koordinatensystem gegeben, lässt sich die Transformationsmatrix T für jede beliebige lineare Transformation bestimmen, indem man die Vektoren in die Gleichung $\vec{v'} = T \cdot \vec{v}$ einsetzt und das Gleichungssystem löst. Daraus ergeben sich allerdings neun Gleichungen in neun Variablen – händisch eine mühsame und für den Computer eine relativ rechenaufwändige Angelegenheit. Kennt man aber die transformierten Äquivalente $\vec{e'_1}$, $\vec{e'_2}$ und $\vec{e'_3}$ der *kanonischen Basisvektoren* $\vec{e}_1 = \begin{pmatrix}1 & 0 & 0\end{pmatrix}^T$, $\vec{e}_2 = \begin{pmatrix}0 & 1 & 0\end{pmatrix}^T$ und $\vec{e}_3 = \begin{pmatrix}0 & 0 & 1\end{pmatrix}^T$, vereinfachen sich die Gleichungen wesentlich:

$$T \cdot \begin{pmatrix}1\\0\\0\end{pmatrix} = \begin{pmatrix}e'_{1x}\\e'_{1y}\\e'_{1z}\end{pmatrix} \qquad \begin{aligned} 1\cdot t_{11} + 0\cdot t_{12} + 0\cdot t_{13} = e'_{1x} &\quad\Rightarrow\quad t_{11} = e'_{1x} \\ 1\cdot t_{21} + 0\cdot t_{22} + 0\cdot t_{23} = e'_{1y} &\quad\Rightarrow\quad t_{21} = e'_{1y} \\ 1\cdot t_{31} + 0\cdot t_{32} + 0\cdot t_{33} = e'_{1z} &\quad\Rightarrow\quad t_{31} = e'_{1z} \end{aligned} \tag{4.34}$$

$$T \cdot \begin{pmatrix}0\\1\\0\end{pmatrix} = \begin{pmatrix}e'_{2x}\\e'_{2y}\\e'_{2z}\end{pmatrix} \qquad \begin{aligned} 0\cdot t_{11} + 1\cdot t_{12} + 0\cdot t_{13} = e'_{2x} &\quad\Rightarrow\quad t_{12} = e'_{2x} \\ 0\cdot t_{21} + 1\cdot t_{22} + 0\cdot t_{23} = e'_{2y} &\quad\Rightarrow\quad t_{22} = e'_{2y} \\ 0\cdot t_{31} + 1\cdot t_{32} + 0\cdot t_{33} = e'_{2z} &\quad\Rightarrow\quad t_{32} = e'_{2z} \end{aligned} \tag{4.35}$$

[2] Ein echtes Problem ist der Gimbal Lock in der Raumfahrt, wo die kardanische Aufhängung tatsächlich Drehungen um alle drei Freiheitsgrade kompensieren muss. (vgl. Hoag 1963)

$$T \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} e'_{3x} \\ e'_{3y} \\ e'_{3z} \end{pmatrix} \qquad \begin{aligned} 0 \cdot t_{11} + 0 \cdot t_{12} + 1 \cdot t_{13} = e'_{3x} \quad &\Rightarrow \quad t_{13} = e'_{3x} \\ 0 \cdot t_{21} + 0 \cdot t_{22} + 1 \cdot t_{23} = e'_{3y} \quad &\Rightarrow \quad t_{23} = e'_{3y} \\ 0 \cdot t_{31} + 0 \cdot t_{32} + 1 \cdot t_{33} = e'_{3z} \quad &\Rightarrow \quad t_{33} = e'_{3z} \end{aligned} \tag{4.36}$$

Für die drei neuen Basisvektoren $\vec{e'_1}$, $\vec{e'_2}$ und $\vec{e'_3}$ entsteht also die Matrix

$$T\left(\vec{e'_1}, \vec{e'_2}, \vec{e'_3}\right) = \begin{pmatrix} e'_{1x} & e'_{2x} & e'_{3x} \\ e'_{1y} & e'_{2y} & e'_{3y} \\ e'_{1z} & e'_{2z} & e'_{3z} \end{pmatrix}, \tag{4.37}$$

beziehungsweise für die Verwendung mit homogenen Koordinaten

$$T\left(\vec{e'_1}, \vec{e'_2}, \vec{e'_3}\right) = \begin{pmatrix} e'_{1x} & e'_{2x} & e'_{3x} & 0 \\ e'_{1y} & e'_{2y} & e'_{3y} & 0 \\ e'_{1z} & e'_{2z} & e'_{3z} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.38}$$

In Worten ausgedrückt, enthält eine Transformationsmatrix also in ihren Spalten die *Bilder der kanonischen Basisvektoren*.

4.3.4. Darstellung als Quaternion

Wie schon bei der Einführung der Quaternionen in Kapitel 2.3 erwähnt, können Quaternionen als Verbindung eines Skalars und eines dreidimensionalen Vektors aufgefasst werden, wobei der Realteil w den Skalar und der Imaginärteil $xi + yj + zk$ den Vektor $\begin{pmatrix} x & y & z \end{pmatrix}^T$ darstellt. Sie eigenen sich deshalb gut für die Darstellung einer Rotation um einen bestimmten Winkel um eine beliebige Achse – der Rotationswinkel entspricht dem Realteil, die Rotationsachse dem Imaginärteil.

Um mit der Hilfe von Quaternionen die Rotation eines Vektors $\vec{x}$ um den Winkel 2φ mit der Achse $\vec{n}$ durchzuführen, bildet man zunächst die Quaternion $q = [\cos\varphi;\ \vec{n} \cdot \sin\varphi]$. Das Ergebnis der Rotation lautet dann $q \cdot \hat{x} \cdot \overline{q}$. Der Vektor $\vec{x}$ wird hier, wie in Kapitel 2.3 besprochen, als reine Quaternion behandelt. Das Ergebnis der Berechnung ist ebenfalls wieder eine reine Quaternion, kann also wieder als dreidimensionaler Vektor aufgefasst werden. So lautet die Rotationsformel (mit $|\vec{n}| = 1$) schließlich:

$$\hat{x}' = [\cos\varphi;\ \vec{n} \cdot \sin\varphi] \cdot \hat{x} \cdot [\cos\varphi;\ -\vec{n} \cdot \sin\varphi] \tag{4.39}$$

Um zu beweisen, dass die oben abgebildete Formel wirklich eine Rotation des Vektors darstellt, betrachten wir zunächst einmal drei Einheitsvektoren $\vec{v_1}$, $\vec{v_2}$ und $\vec{v_3}$ aus dem $\mathbb{R}^3$ (vgl. Shankel 2000). Sie sind komplanar und bilden jeweils den Winkel φ. Es gilt daher:

$$|\vec{v_1}| = |\vec{v_2}| = |\vec{v_3}| = 1 \tag{4.40}$$

$$\vec{v_1} \cdot \vec{v_2} = \vec{v_2} \cdot \vec{v_3} = \cos\varphi \tag{4.41}$$

$$\vec{v_1} \times \vec{v_2} = \vec{v_2} \times \vec{v_3} \tag{4.42}$$

$\vec{v_3}$ ensteht also durch eine Rotation von $\vec{v_1}$ um den Winkel 2φ mit der Achse $\vec{v_1} \times \vec{v_2}$ (siehe Skizze 4.4).

Wir definieren nun die Quaternion q über $\vec{v}_1$ und $\vec{v}_2$:

$$q = [\vec{v}_1 \cdot \vec{v}_2;\ \vec{v}_1 \times \vec{v}_2] \tag{4.43}$$

Wir wollen zunächst beweisen:

$$q \cdot \hat{v}_1 \cdot \overline{q} = \hat{v}_3 \tag{4.44}$$

Für q gilt

$$q = \hat{v}_2 \cdot \overline{\hat{v}_1}, \tag{4.45}$$

weil nach der Multiplikationsvorschrift für Quaternionen (siehe Gleichung 2.44) gilt:

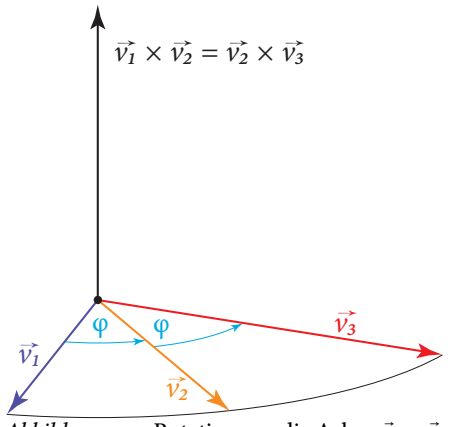

Abbildung 4.4.: Rotation um die Achse $\vec{v}_1 \times \vec{v}_2$.

$$\begin{aligned} \hat{v}_2 \cdot \overline{\hat{v}_1} &= [0 + \vec{v}_2 \cdot \vec{v}_1;\ -\vec{v}_2 \times \vec{v}_1 - 0 \cdot \vec{v}_1 + 0 \cdot \vec{v}_2] \\ &= [\vec{v}_2 \cdot \vec{v}_1;\ -\vec{v}_2 \times \vec{v}_1] \\ &= [\vec{v}_1 \cdot \vec{v}_2;\ \vec{v}_1 \times \vec{v}_2] \end{aligned} \tag{4.46}$$

Dann gilt natürlich auch

$$q = \hat{v}_3 \cdot \overline{\hat{v}_2}, \tag{4.47}$$

weil ja nach Gleichung 4.41 und 4.42 die Skalar- und Kreuzprodukte der beiden Vektorenpaare identisch sind.

Wir setzen nun den Zusammenhang aus 4.45 in Gleichung 4.44 ein und erhalten:

$$\begin{aligned} \hat{v}_2 \cdot \overline{\hat{v}_1} \cdot \hat{v}_1 \cdot \overline{q} &= \hat{v}_3 \\ \hat{v}_2 \cdot |\hat{v}_1| \cdot \overline{q} &= \hat{v}_3 \end{aligned} \tag{4.48}$$

Nachdem $\vec{v}_1$ ein Einheitsvektor ist, hat auch $\hat{v}_1$ den Betrag 1, der Ausdruck vereinfacht sich also zu:

$$\hat{v}_2 \cdot \overline{q} = \hat{v}_3 \tag{4.49}$$

Nun setzen wir den Zusammenhang aus 4.47 für $\overline{q}$ ein und vereinfachen nach Gleichung 2.45:

$$\begin{aligned} \hat{v}_2 \cdot \overline{\left(\hat{v}_3 \cdot \overline{\hat{v}_2}\right)} &= \hat{v}_3 \\ \hat{v}_2 \cdot \hat{v}_2 \cdot \overline{\hat{v}_3} &= \hat{v}_3 \end{aligned} \tag{4.50}$$

Nachdem $\hat{v}_2$ eine reine Quaternion und $\vec{v}_2$ ein Einheitsvektor ist, vereinfacht sich der Ausdruck $\hat{v}_2 \cdot \hat{v}_2$:

$$\begin{aligned} \hat{v}_2 \cdot \hat{v}_2 &= [0 - \vec{v}_2 \cdot \vec{v}_2;\ \vec{v}_2 \times \vec{v}_2 + 0 \cdot \vec{v}_2 + 0 \cdot \vec{v}_2] \\ &= [-|\vec{v}_2|^2;\ \vec{0}] \\ &= -1 \end{aligned} \tag{4.51}$$

Wir setzen in 4.50 ein und erhalten:

$$\begin{aligned} -\overline{\hat{v}_3} &= \hat{v}_3 \\ -[0;\ -\vec{v}_3] &= [0;\ \vec{v}_3] \\ \vec{v}_3 &= \vec{v}_3 \end{aligned} \tag{4.52}$$

Damit wäre der Zusammenhang aus Gleichung 4.44 bewiesen

$$q \cdot \hat{v_1} \cdot \overline{q} = \hat{v_3},$$

beziehungsweise nach dem Einsetzen für q

$$\left[\vec{v_1} \cdot \vec{v_2};\ \vec{v_1} \times \vec{v_2}\right] \cdot \hat{v_1} \cdot \left[\vec{v_1} \cdot \vec{v_2};\ -(\vec{v_1} \times \vec{v_2})\right] = \hat{v_3}. \tag{4.53}$$

Unter Beachtung der Zusammenhänge $\vec{a} \cdot \vec{b} = |a| \cdot |b| \cdot \cos\varphi$ und $\left|\vec{a} \times \vec{b}\right| = |a| \cdot |b| \cdot \sin\varphi$ und der Tatsache, dass $\vec{v_1}$ und $\vec{v_2}$ Einheitsvektoren sind, können wir stattdessen auch schreiben:

$$\left[\cos\varphi;\ \vec{n} \cdot \sin\varphi\right] \cdot \hat{v_1} \cdot \left[\cos\varphi;\ -\vec{n} \cdot \sin\varphi\right] = \hat{v_3} \tag{4.54}$$

Nun brauchen wir nur noch beide Seiten mit einem beliebigen Skalar a zu multiplizieren ($\hat{v_1}$ und $\hat{v_3}$ sind ja auf Einheitsquaternionen beschränkt), und es ergibt sich die oben aufgestellte Formel:

$$\begin{aligned} \left[\cos\varphi;\ \vec{n} \cdot \sin\varphi\right] \cdot a \cdot \hat{v_1} \cdot \left[\cos\varphi;\ -\vec{n} \cdot \sin\varphi\right] &= a \cdot \hat{v_3} \\ \left[\cos\varphi;\ \vec{n} \cdot \sin\varphi\right] \cdot \hat{x} \cdot \left[\cos\varphi;\ -\vec{n} \cdot \sin\varphi\right] &= \hat{x}' \end{aligned} \tag{4.55}$$

Wenn die Quaternionen $q_1, q_2, \ldots, q_n$ eine Rotation darstellen, dann lassen sich die Rotationen in eine Quaternion q kombinieren, indem man die Quaterionen – ähnlich wie bei Transformationsmatrizen – einfach miteinander multipliziert: $q = q_n \cdot q_{n-1} \cdot \ldots \cdot q_1$. Der zugehörige Beweis ist nahezu trivial: Rotationen zu verketten heißt ja, das Ergebnis der einen Rotation als Ausgangswert für die nächste Transformation zu verwenden. Im Falle von drei Rotationen lautet die Formel also

$$\hat{x}' = q_3 \cdot (q_2 \cdot (q_1 \cdot \hat{x} \cdot \overline{q_1}) \cdot \overline{q_2}) \cdot \overline{q_3}, \tag{4.56}$$

was wir wegen der Assoziativität der Quaternionenmultiplikation als

$$\hat{x}' = (q_3 \cdot q_2 \cdot q_1) \cdot \hat{x} \cdot (\overline{q_1} \cdot \overline{q_2} \cdot \overline{q_3}) \tag{4.57}$$

schreiben können, was wir gleich zu

$$\begin{aligned} \hat{x}' &= (q_3 \cdot q_2 \cdot q_1) \cdot \hat{x} \cdot \overline{(q_3 \cdot q_2 \cdot q_1)} \\ &= q \cdot \hat{x} \cdot \overline{q} \end{aligned} \tag{4.58}$$

vereinfachen können, da sich die Quaternionenmultiplikation ja bezüglich der Konjugation distributiv unter Vertauschung der Reihenfolge verhält (siehe Gleichung 2.45).

Quaternionen stellen also wegen ihrer Beziehung zu Rotationswinkel und -achse ein praktisches Mittel dar, um Rotationen anzugeben und mehrere Rotationen können einfach miteinander kombiniert werden (auch in Hinblick auf die am Computer dafür benötigten Rechenoperationen). Zusätzlich kann es für manche Computeranwendungen nützlich sein, dass nur vier Werte gespeichert werden müssen, um jede beliebige Rotation darzustellen.

Die Repräsentation als Quaternion hat aber auch Nachteile. Zum einen ist die direkte Transformation von Vektoren durch Quaternionen eine relativ rechenaufwändige Angelegenheit, was für den Einsatz in der 3D-Grafik nicht gerade optimal ist. Zum anderen unterstützt die derzeitige Grafikhardware wie in Kapitel 3.2.1 beschrieben lediglich Matrizen, um die Vertices zu transformieren. Daher liegt es nahe,

einen Weg zu suchen, eine Rotationsquaternion in eine Rotationsmatrix zu konvertieren. (vgl. Eberly 2002, 8-9)

Nachdem ja $\hat{x}' = q \cdot \hat{x} \cdot \overline{q}$ gilt, liegt es nahe, dass man auch nach einer Matrix suchen kann, für die $\vec{x}' = M \cdot \vec{x}$ gilt, die also bei entsprechender Interpretation von reinen Quaternionen als Vektoren die Bedingung $q \cdot \hat{x} \cdot \overline{q} = M \cdot \vec{x}$ erfüllt. Es gibt zahlreiche Wege, die Matrix direkt aus diesem Zusammenhang herzuleiten (vgl. Koch 2008, 56-58; Formella und Fellner 2005, 93-96), ich möchte aber stattdessen eine andere Möglichkeit zur Herleitung anführen, die leichter nachzuvollziehen ist.

Wie in Abschnitt 4.3.3 besprochen, kann man eine Transformationsmatrix als Aneinanderreihung der transformierten kanonischen Basisvektoren aufstellen. Um eine Matrix aus einem Rotationsquaternion zu erzeugen, muss man also nur die drei Basisvektoren direkt transformieren und die Ergebnisse in die Matrix schreiben.

Dafür multiplizieren wir zunächst einmal die Rotationsformel aus und vereinfachen mit Hilfe der Grassmann-Identität[3] (vgl. Koch 2008, 52):

$$\begin{aligned}
q \cdot \hat{x} \cdot \overline{q} &= [w;\ \vec{n}] \cdot [0;\ \vec{x}] \cdot [w;\ -\vec{n}] \\
&= [-\vec{n} \cdot \vec{x};\ \vec{n} \times \vec{x} + w \cdot \vec{x}] \cdot [w;\ -\vec{n}] \\
&= \begin{bmatrix} -w \cdot (\vec{n} \cdot \vec{x}) + (\vec{n} \times \vec{x} + w \cdot \vec{x}) \cdot \vec{n}; \\ -(\vec{n} \times \vec{x} + w \cdot \vec{x}) \times \vec{n} + w \cdot (\vec{n} \times \vec{x} + w \cdot \vec{x}) - \vec{n} \cdot \vec{x} \cdot (-\vec{n}) \end{bmatrix} \\
&= \begin{bmatrix} -w \cdot (\vec{n} \cdot \vec{x}) + (\vec{n} \times \vec{x}) \cdot \vec{n} + w \cdot (\vec{n} \cdot \vec{x}); \\ -((\vec{n} \times \vec{x}) \times \vec{n}) - w \cdot (\vec{x} \times \vec{n}) + w \cdot (\vec{n} \times \vec{x}) + w^2 \cdot \vec{x} + (\vec{n} \cdot \vec{x}) \cdot \vec{n} \end{bmatrix} \\
&= \left[0;\ \vec{n} \times (\vec{n} \times \vec{x}) + 2 \cdot w \cdot (\vec{n} \times \vec{x}) + w^2 \cdot \vec{x} + (\vec{n} \cdot \vec{x}) \cdot \vec{n}\right] \\
&= \left[0;\ 2 \cdot (\vec{n} \cdot \vec{x}) \cdot \vec{n} + 2 \cdot w \cdot (\vec{n} \times \vec{x}) + (w^2 - \vec{n} \cdot \vec{n}) \cdot \vec{x}\right]
\end{aligned} \tag{4.59}$$

Mithilfe dieser Formel transformieren wir nun die drei Basisvektoren $\vec{e_1} = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}^T$, $\vec{e_2} = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}^T$ und $\vec{e_3} = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^T$. Für die Komponenten von $\vec{e_1'}$ erhalten wir (durch Einsetzen von $\vec{n} = \begin{pmatrix} x & y & z \end{pmatrix}^T$ und $\vec{x} = \vec{e_1} = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}^T$ in Gleichung 4.59) die Gleichungen[4]:

$$\begin{aligned}
e_{1x}' &= 2 \cdot (x + 0 + 0) \cdot x + 2 \cdot w \cdot (y \cdot 0 - z \cdot 0) + (w^2 - x^2 - y^2 - z^2) \cdot 1 \\
&= x^2 + w^2 - y^2 - z^2 \\
&= (1 - y^2 - z^2 - w^2) + w^2 - y^2 - z^2 \\
&= 1 - 2 \cdot (y^2 + z^2)
\end{aligned} \tag{4.60}$$

$$\begin{aligned}
e_{1y}' &= 2 \cdot (x + 0 + 0) \cdot y + 2 \cdot w \cdot (1 \cdot z - 0 \cdot x) + (w^2 - x^2 - y^2 - z^2) \cdot 0 \\
&= 2xy + 2wz \\
&= 2 \cdot (xy + wz)
\end{aligned} \tag{4.61}$$

$$\begin{aligned}
e_{1z}' &= 2 \cdot (x + 0 + 0) \cdot z + 2 \cdot w \cdot (x \cdot 0 - y \cdot 1) + (w^2 - x^2 - y^2 - z^2) \cdot 0 \\
&= 2xz - 2wy \\
&= 2 \cdot (xz - wy)
\end{aligned} \tag{4.62}$$

[3] $\vec{a} \times (\vec{b} \times \vec{c}) = \vec{b} \cdot (\vec{a} \cdot \vec{c}) - \vec{c} \cdot (\vec{a} \cdot \vec{b})$

[4] q ist eine Einheitsquaternion, also gilt $w^2 + x^2 + y^2 + z^2 = 1$!

Analog dazu werden auch die anderen beiden Vektoren transformiert:

$$\vec{e_2'} = \begin{pmatrix} 2 \cdot (xy - wz) \\ 1 - 2 \cdot (x^2 + z^2) \\ 2 \cdot (yz + wx) \end{pmatrix} \tag{4.63}$$

$$\vec{e_3'} = \begin{pmatrix} 2 \cdot (xz + wy) \\ 2 \cdot (yz - wx) \\ 1 - 2 \cdot (x^2 + y^2) \end{pmatrix} \tag{4.64}$$

Durch Nebeneineinanderstellen von $\vec{e_1'}$, $\vec{e_2'}$ und $\vec{e_3'}$ ergibt sich schließlich die *Rotationsmatrix zu einer Rotationsquaternion* $q = w + xi + yj + zk$

$$R(q) = \begin{pmatrix} 1 - 2 \cdot (y^2 + z^2) & 2 \cdot (xy - wz) & 2 \cdot (xz + wy) \\ 2 \cdot (xy + wz) & 1 - 2 \cdot (x^2 + z^2) & 2 \cdot (yz - wx) \\ 2 \cdot (xz - wy) & 2 \cdot (yz + wx) & 1 - 2 \cdot (x^2 + y^2) \end{pmatrix}, \tag{4.65}$$

beziehungsweise zur Verwendung mit homogenen Koordinaten

$$R(q) = \begin{pmatrix} 1 - 2 \cdot (y^2 + z^2) & 2 \cdot (xy - wz) & 2 \cdot (xz + wy) & 0 \\ 2 \cdot (xy + wz) & 1 - 2 \cdot (x^2 + z^2) & 2 \cdot (yz - wx) & 0 \\ 2 \cdot (xz - wy) & 2 \cdot (yz + wx) & 1 - 2 \cdot (x^2 + y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.66}$$

5. Betrachtungstransformationen

Im vorigen Kapitel haben wir die verschiedenen Formen von Skalierung, Translation und Rotation behandelt, die für die Platzierung und Ausrichtung der einzelnen Modelle in der Szene, also für das Berechnen der World Matrix, nötig sind. Das Thema dieses Kapitels sind die Transformationen, die für den Übergang vom Weltkoordinatensystem in das Bildschirmkoordinatensystem verwendet werden.

5.1. View Matrix

In der 3D-Programmierung haben sich zwei praktische Arten eingebürgert, um die Ansicht auf die Szene festzulegen: Die Position des Betrachters kombiniert mit seiner Blickrichtung oder einem Punkt, den er anvisiert. Die View Matrix, die aus diesen Parametern berechnet wird, überführt das Weltkoordinatensystem in das Kamerakoordinatensystem, das seinen Ursprung im Auge des virtuellen Betrachters hat. Die x-Achse des neuen Systems zeigt dabei nach rechts, die y-Achse nach oben, und die z-Achse (in einem rechtshändigen System) gegen die Blickrichtung.

Im ersten Fall sind drei Vektoren gegeben: der Positionsvektor $\vec{p}$, der Richtungsvektor $\vec{d}$ und der Hochvektor $\vec{u}$. Die Bedeutung von $\vec{p}$ und $\vec{u}$ sollte klar sein, der Hochvektor ist nötig, um festzulegen, welche Richtung für den Betrachter »oben« ist. Alle Vektoren haben Einheitslänge, $\vec{u}$ steht senkrecht auf $\vec{d}$. Der dritte Basisvektor des neuen Koordinatensystems, der Rechtsvektor $\vec{r}$, lässt sich leicht über das Kreuzprodukt berechnen:

$$\vec{r} = \vec{d} \times \vec{u} \tag{5.1}$$

Im zweiten Fall ist neben $\vec{p}$ nur die Position des Ziels, $\vec{t}$, gegeben. Der Hochvektor muss aus dem Hochvektor $\vec{u}_w$ des Weltkoordinatensystems (normalerweise $\begin{pmatrix}0 & 1 & 0\end{pmatrix}^T$) berechnet werden. In der Regel wird diese Darstellung in die erste Variante konvertiert, um dann daraus die Matrix zu berechnen. Dazu wird zunächst der normierte Richtungsvektor $\vec{d}$ berechnet, der von der Position der Kamera zum Ziel zeigt:

$$\vec{d} = \frac{\vec{t} - \vec{p}}{\left|\vec{t} - \vec{p}\right|} \tag{5.2}$$

Der Rechtsvektor muss senkrecht auf der zwischen Welt-Hochvektor und Richtungsvektor aufgespannten Ebene stehen. Er lässt sich daher über das Kreuzprodukt gewinnen:

$$\vec{r} = \frac{\vec{d} \times \vec{u}_w}{\left|\vec{d} \times \vec{u}_w\right|} \tag{5.3}$$

Der Hochvektor berechnet sich nun als Kreuzprodukt aus Rechts- und Richtungsvektor

$$\vec{u} = \vec{r} \times \vec{d} = \frac{\left(\vec{d} \times \vec{u}_w\right) \times \vec{d}}{\left|\left(\vec{d} \times \vec{u}_w\right) \times \vec{d}\right|}, \tag{5.4}$$

was sich unter Zuhilfenahme der Grassmann-Identität zu dem schneller berechenbaren Ausdruck

$$\vec{u} = \frac{\vec{u_w} - \left(\vec{d} \cdot \vec{u_w}\right) \cdot \vec{d}}{\left|\vec{u_w} - \left(\vec{d} \cdot \vec{u_w}\right) \cdot \vec{d}\right|} \tag{5.5}$$

vereinfachen lässt.

In beiden Fällen geht es nun darum, aus $\vec{p}$, $\vec{d}$, $\vec{u}$ und $\vec{r}$ die Transformationsmatrix zu bestimmen. Die Transformation in das Kamerakoordinatensystem kann in zwei Schritte aufgeteilt werden: Zuerst wird der Ursprung des Koordinatensystems in den Standpunkt der Kamera verschoben, dann werden die Achsen korrekt ausgerichtet.

Um den Ursprung in $\vec{p}$ zu verschieben, müssen alle Objekte um $-\vec{p}$ verschoben werden:

$$M_{trans} = \begin{pmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.6}$$

Die Vektoren $-\vec{d}$, $\vec{u}$ und $\vec{r}$ sind ja die Basisvektoren des Kamerakoordinatensystems. Deshalb kann mit ihrer Hilfe nun (wie in Kapitel 4.3.3 beschrieben) die Matrix für den Rotationsteil aufgestellt werden. Dabei ist zu beachten, dass die Rotation der Kamera ja rückgängig gemacht werden soll, die entstehende Matrix also invertiert werden muss:

$$M_{rot} = \begin{pmatrix} r_x & u_x & -d_x & 0 \\ r_y & u_y & -d_y & 0 \\ r_z & u_z & -d_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ -d_x & -d_y & -d_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.7}$$

Um schließlich die View Matrix zu erhalten, müssen die beiden Matrizen nur noch multipliziert werden:

$$\begin{aligned} M_{view} &= M_{rot} \cdot M_{trans} \\ &= \begin{pmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ -d_x & -d_y & -d_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ M_{view} &= \begin{pmatrix} r_x & r_y & r_z & -\vec{r} \cdot \vec{p} \\ u_x & u_y & u_z & -\vec{u} \cdot \vec{p} \\ -d_x & -d_y & -d_z & \vec{d} \cdot \vec{p} \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned} \tag{5.8}$$

5.2. Projection Matrix

Die Positionen aller Vertices liegen nach der Transformation durch die View Matrix also im lokalen Koordinatensystem der Kamera vor. Um aus den dreidimensionalen Koordinaten im Raum die zweidimensionalen Bildschirmkoodinaten zu erhalten, müssen sie in irgendeiner Form projiziert werden. Für diese Projektion gibt es je nach Anwendungsgebiet viele Möglichkeiten, darunter auch eher exotische wie

die Projektion auf einen Zylindermantel oder krummlinige Projektionen, um Linseneffekte zu simulieren. Aber selbst wenn man sich auf den einfachsten Fall, die Projektion auf eine Ebene, beschränkt, gibt es noch immer zahlreiche Unterarten. (vgl. Formella und Fellner 2005, 115-117)

Ich werde mich hier auf die bei Weitem am häufigsten verwendete Art beschränken, die *Zentralprojektion*. Hier wird ein Punkt projiziert, indem man die Linie durch den Punkt und ein gemeinsames Projektionszentrum mit der Bildebene schneidet. Geraden bleiben dabei auch in der Abbildung Geraden, im dreidimensionalen Raum parallele Kanten schneiden sich in einem gemeinsamen Fluchtpunkt. Die Zentralprojektion ist der Abbildung durch das menschliche Auge sehr ähnlich und ermöglicht somit natürlich wirkende Bilder. (vgl. Wikipedia 2009c)

Das *Sichtvolumen* der Zentralprojektion wird also grundsätzlich von den Strahlen durch den Augpunkt und die Ecken des Sichtfensters aufgespannt, ist also eine unendliche rechteckige Pyramide mit der Spitze im Augpunkt $\begin{pmatrix}0 & 0 & 0\end{pmatrix}^T$. Die Projektion wird normalerweise durch zwei Parameter charakterisiert: den vertikalen Öffnungswinkel des Sichtvolumens, φ, und das Seitenverhältnis des Sichtfensters (also im Normalfall des Bildschirmes), $\frac{W}{H} = r$. Wir werden hier nur die normalerweise verwendete *symmetrische Zentralprojektion* behandeln, bei der der Augpunkt in das Zentrum des Bildes projiziert wird.

Wie bereits in Kapitel 3.2.1 erwähnt, besteht die Aufgabe der Projection Matrix darin, dieses Sichtvolumen in das *kanonische Sichtvolumen* zu überführen, wonach die Geometrie wieder für alle Arten von Projektion gleich behandelt wird. Auch für das kanonische Sichtvolumen gibt es wieder unterschiedliche Konventionen, wir werden hier die DirectX-Variante behandeln, einen Quader zwischen den Punkten $\begin{pmatrix}-1 & -1 & 0\end{pmatrix}^T$ und $\begin{pmatrix}1 & 1 & 1\end{pmatrix}^T$. (vgl. Microsoft Corporation 2008)

Für die Transformation muss das Sichtvolumen aber zuerst begrenzt werden, da sonst verschiedene grundsätzliche und numerische Probleme entstehen würden[1]. Das geschieht mit zwei zur z-Achse senkrechten Ebenen, der *Near Clipping Plane* und der *Far Clipping Plane* (von engl. *to clip*, »abschneiden«). Das entstehende Volumen ist ein Pyramidenstumpf, in der 3D-Grafik meist mit dem englischen Begriff *Frustum* bezeichnet.

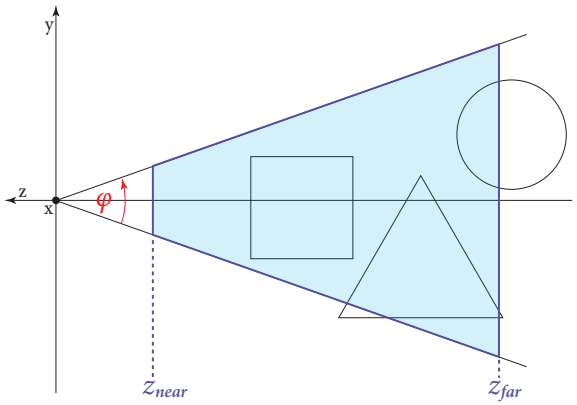

Abbildung 5.1.: Near- und Far Clipping Plane.

Besonders wichtig ist die Begrenzung des Sichtvolumens für die auf die Projektion folgende Sichtbarkeitsbestimmung, da hier wie auch sonst in Computerprogrammen mit einer beschränkten Genauigkeit gerechnet wird. Ist der Abstand von Near und Far Clipping Plane zu groß, macht sich die begrenzte Genauigkeit bemerkbar und es kommt zu einem als *Z-Fighting* bekannten Phänomen, bei dem auf nicht-deterministische Weise abwechselnd verschiedene Pixel von Polygonen in einer ähnlichen Tiefe im Vordergrund gezeichnet werden.

Zur Herleitung der Projection Matrix für die Zentralprojektion transformieren wir das Sichtvolumen zunächst einmal so, dass der horizontale und der vertikale Öffnungswinkel 90° betragen ($\varphi = 90°$ und $r = 1$). Dazu ist eine Skalierung entlang der x- und der y-Achse nötig: Wie sich aus Abbildung 5.1 und 5.2 leicht erkennen lässt, wächst die y-Koordinate der oberen und unteren Begrenzung des Frustums

[1] Beispielsweise würden Punkte, die exakt im Augpunkt liegen, Probleme verursachen und sehr weit entfernte Objekte würden im Hinblick auf die begrenzte Rechengenauigkeit problematisch sein.

bei steigender z-Koordinate mit dem Faktor $\pm \tan \frac{\varphi}{2}$, im transformierten Frustum aber mit dem Faktor ± 1. Die x-Koordinate der linken und der rechten Grenze verhalten sich bis auf den Faktor für das Seitenverhältnis genauso. Das Sichtvolumen muss also um den Faktor $1 : (\tan \frac{\varphi}{2})$ in y-Richtung und $1 : (\tan \frac{\varphi}{2} \cdot r)$ in x-Richtung skaliert werden.

Daneben spiegeln wir das Koordinatensystem an der xy-Ebene, um ein intuitiveres Arbeiten zu ermöglichen – mit steigendem Abstand von der Kamera wächst nun auch die z-Koordinate. Für diesen ersten Schritt ergibt sich also die Matrix

$$P_1 = \begin{pmatrix} \frac{1}{r \cdot \tan \frac{\varphi}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan \frac{\varphi}{2}} & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{5.9}$$

Danach werden die Punkte auf die Ebene $z = 1$ projiziert. Wie aus Abbildung 5.2 ersichtlich (ähnliche Dreiecke), gilt der Zusammenhang

$$\frac{x'}{1} = \frac{x}{z} \tag{5.10}$$

und analog für die y-Koordinate

$$\frac{y'}{1} = \frac{y}{z}. \tag{5.11}$$

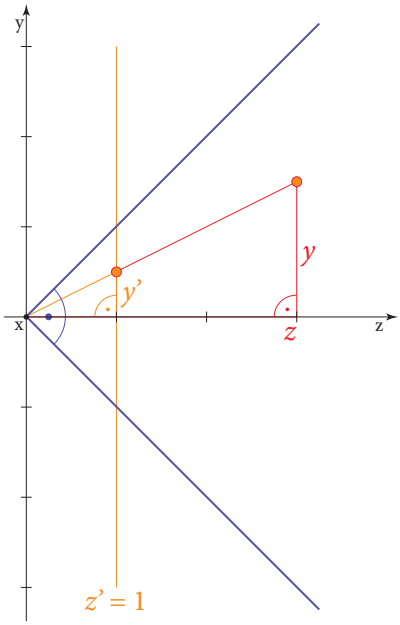

Abbildung 5.2.: Rechtwinkliges Sichtvolumen mit Projektion auf die Ebene $z = 1$.

Um die Projektion durchzuführen, müssen die x- und die y-Koordinate also durch die z-Koordinate dividiert werden. Um dies in eine Matrix zu verpacken, machen wir uns die homogenen Koordinaten zu Nutze, genauer die Konvention, dass der Vektor nachher durch Division durch w wieder auf $w = 1$ gebracht wird. Die Multiplikation mit der Matrix muss also einfach nur $w' = z$ setzen:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{5.12}$$

Nach der Transformation mit dieser Matrix hat ein Vektor die Koordinaten $(x \quad y \quad z \quad z)^T$, nach der homogenen Division also $(\frac{x}{z} \quad \frac{y}{z} \quad 1)^T$.

Für die Projektion reicht diese Transformation an sich, und wie man leicht nachprüfen kann, befinden sich die x- und y-Koordinaten alle Punkte im Sichtvolumen jetzt zwischen -1 und 1, wie es für das kanonische Sichtvolumen erforderlich ist. Allerdings ist durch die Projektion auch z' für alle Punkte 1, ist also unabhängig von z. Um später beim Zeichnen der Dreiecke die Sichtbarkeit korrekt bestimmen zu können, muss z' aber mit wachsender Tiefe monoton steigen, die z-Reihenfolge der Punkte muss also erhalten bleiben. Um diese Bedingung zu erfüllen, werden zwei zusätzliche Parameter, a und b, eingeführt:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{5.13}$$

Nach der Transformation und der homogenen Division ergibt sich mit dieser Matrix

$$z' = \frac{a \cdot z + b}{z}. \tag{5.14}$$

Nun müssen nur noch passende Werte für a und b gefunden werden. Aus der Definition des kanonischen Sichtvolumens folgen ja die Bedingungen, dass ein Punkt auf der Near Clipping Plane auf $z' = 0$, und ein Punkt auf der Far Clipping Plane auf $z' = 1$ abgebildet wird. Aus diesen Bedingungen ergibt sich also ein lineares Gleichungssystem mit zwei Variablen

$$\begin{aligned} 0 &= \frac{a \cdot z_{\text{near}} + b}{z_{\text{near}}} \\ 1 &= \frac{a \cdot z_{\text{far}} + b}{z_{\text{far}}}, \end{aligned} \tag{5.15}$$

welches sich einfach zu

$$\begin{aligned} a &= \frac{z_{\text{far}}}{z_{\text{far}} - z_{\text{near}}} \\ b = -z_{\text{near}} \cdot a &= -\frac{z_{\text{far}} \cdot z_{\text{near}}}{z_{\text{far}} - z_{\text{near}}}, \end{aligned} \tag{5.16}$$

auflösen lässt. (vgl. Bishop und Van Verth 2004, 220 f.; Bekaert 1999)

Für die Projektion des rechtwinkligen Sichtvolumens ergibt sich also die Matrix

$$P_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{z_{\text{far}}}{z_{\text{far}} - z_{\text{near}}} & -\frac{z_{\text{far}} \cdot z_{\text{near}}}{z_{\text{far}} - z_{\text{near}}} \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{5.17}$$

Kombiniert man die beiden Matrizen durch Multiplikation, erhält man die *Projection Matrix für die Zentralprojektion*:

$$\begin{aligned} M_{proj} = P_2 \cdot P_1 &= \begin{pmatrix} \frac{1}{r \cdot \tan\frac{\varphi}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\frac{\varphi}{2}} & 0 & 0 \\ 0 & 0 & -\frac{z_{\text{far}}}{z_{\text{far}} - z_{\text{near}}} & -\frac{z_{\text{far}} \cdot z_{\text{near}}}{z_{\text{far}} - z_{\text{near}}} \\ 0 & 0 & -1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} \frac{1}{r \cdot \tan\frac{\varphi}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\frac{\varphi}{2}} & 0 & 0 \\ 0 & 0 & \frac{z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} & \frac{z_{\text{near}} \cdot z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} \\ 0 & 0 & -1 & 0 \end{pmatrix} \end{aligned} \tag{5.18}$$

Nach der Transformation mit dieser Matrix befinden sich alle theoretisch sichtbaren Vertices im kanonischen Sichtvolumen, der Schritt der Projektion ist also abgeschlossen. Wie in Kapitel 3.2 beschrieben, folgt nun das *Clipping*. In diesem Schritt wird die darzustellende Geometrie so abgeschnitten, dass sie nicht über das kanonische Sichtvolumen hinausragt. Hier gibt es bei der Behandlung eines Dreiecks grundsätzlich drei Möglichkeiten: Erstens kann es vollständig innerhalb des Sichtvolumens liegen, in diesem Fall bleibt es natürlich unverändert. Zweitens kann es vollständig außerhalb des Sichtvolumens

liegen, es wird dann verworfen. Drittens kann es aber auch von einer oder mehreren der Begrenzungsflächen geschnitten werden. In diesem Fall wird das Dreieck verworfen und es werden neue Dreiecke generiert, welche die Fläche des Dreiecks, die innerhalb des Sichtvolumens liegt, ausfüllen.

Die x- und y-Koordinaten aller Vertices befinden sich nach dem Clipping also im Bereich $[-1;1]$. Sie müssen dann nur noch in den Wertebereich des Ausgabemediums gebracht werden. Wenn ein typischer Bildschirm im 4 : 3-Format ganz ausgefüllt werden soll, könnte dies beispielsweise der Bereich $[0;1279]$ horizontal und $[0;959]$ vertikal sein[2]. Dieser Schritt wird als *Viewport Transformation* bezeichnet und besteht im Normalfall aus einer Translation und einer Skalierung. Dabei ist zu berücksichtigen, dass die Werte in y-Richtung in allen vorherigen Koordinatensystemen nach oben wachsen, die y-Achse am Bildschirm aber nach unten zeigt. Es ist also eine Spiegelung an der xz-Ebene erforderlich.

[2] Dies entspricht einer Bildschirmauflösung von 1280×960 – in der Informatik wird fast immer 0 als Index für das erste Element verwendet.

6. Ausblick

In den letzten beiden Kapiteln wurden alle Transformationen behandelt, die nötig sind, um die World, View, Projection und Viewport Matrix zu erzeugen. Nachdem die Vertices diese Transformationen durchlaufen haben, liegen ihre Positionen in Bildschirmkoordinaten vor, die Dreiecke müssen also »nur noch« gezeichnet werden.

Die Anführungszeichen sind mit Bedacht gesetzt, denn die Rasterisierung erscheint höchstens aus einem mathematischen Blickwinkel einfach. Aus programmiertechnischer Sicht ist das Füllen, wie in Kapitel 3 angedeutet, alles andere als trivial. Ein guter Teil der Komplexität entsteht dadurch, dass die Operationen alle äußerst schnell ausgeführt werden müssen, damit das Programm insgesamt eine akzeptable Leistung erbringt – man bedenke, dass die meisten aktuellen Bildschirme aus über 10^6 Bildpunkten bestehen, deren Farbwerte mindestens 20–30 Mal pro Sekunde berechnet werden müssen.

Unter anderem müssen also Algorithmen gefunden werden, mit denen die Größen, die für die Farbberechnung erforderlich sind, für alle Pixel eines Dreiecks möglichst effizient aus denen der Vertices interpoliert werden können. Daneben versucht man auch, die Anzahl der »unnötigerweise« berechneten Pixel zu reduzieren, indem man Pixel, die später sowieso von einem näher bei der Kamera liegenden Dreieck überdeckt werden, möglichst früh aussortiert, etc.

Dabei handelt es sich aber um Fragestellungen, deren Beantwortung den Rahmen dieser Arbeit in zweifacher Hinsicht sprengen würde: Zum einen, weil es sich dabei keineswegs um allgemeine Grundlagen handelt – die diesbezüglichen Ansätze der Grafikkartenhersteller unterschieden sich teilweise erheblich (und tun dies zum Teil auch heute noch). Zum anderen, weil die Probleme ohne Rücksichtnahme auf die internen Arbeitsweisen und Abläufe eines Computers kaum sinnvoll zu behandeln sind, was wiederum nicht zu der mathematischen Ausrichtung dieser Arbeit passen würde.

Weiters muss man sich bewusst sein, dass die hier behandelten Themen, wie auch schon im Titel der Arbeit ausgedrückt, lediglich die Grundlagen der 3D-Grafik sind. Es ist zwar möglich, rein mit diesen Hilfsmitteln ein Programm zu erstellen, das eine rudimentäre Darstellung einer Szene erzeugt (wie etwa das Beispielprogramm zu dieser Arbeit). Es fehlen jedoch noch viele der Fähigkeiten, welche die 3D-Grafik in den letzten Jahren erst so richtig interessant werden ließen und beispielsweise ihren Einsatz in Computerspielen überhaupt erst ermöglichen.

Dazu zählen einerseits die Techniken, welche die gerenderten Bilder erst halbwegs realistisch erscheinen lassen, also zum Bespiel die in Kapitel 3.1.3 kurz erwähnten Texturierungstechniken zur Simulation von Details oder Techniken zur Darstellung von Reflexionen und Tiefenunschärfe.

Andererseits sind dies die Techniken, die nötig sind, um größere Mengen an Objekten sinnvoll verwalten zu können. Je nach Einsatzgebiet werden die Daten hier in speziellen Strukturen gespeichert, die den Raum nach einem bestimmten Prinzip unterteilen (etwa in sogenannten Octrees oder BSP-Trees). Zusammen mit anderen Techniken wie zum Beispiel dem Portal-Rendering können so große Teile der eine bestimmte Kameraposition irrelevante Objekte gleich im Vornhinein ausgeschlossen werden. Diese Techniken ermöglichen erst, größere Umgebungen ohne »künstliche«, für den Spieler sichtbare Unterteilungen in Computerspiele zu integrieren.

Außerdem gehört zu einer Simulation einer dreidimensionalen Welt ja nicht nur die Grafikausgabe, sondern es müssen auch zahlreiche andere Berechnungen im dreidimensionalen Raum ablaufen. Für Computerspiele ist hier besonders die Kollisionserkennung wichtig, bei der möglichst effizient festgestellt werden soll, ob ein Objekt irgendein anderes Objekt der Szene berührt, was wegen der oft hohen Anzahl an Objekten in einer Szene keineswegs trivial ist.

Schließlich darf man bei der Auseinandersetzung mit der 3D-Grafik nicht vergessen, dass auf diesem Gebiet zur Zeit ein sehr hohes Innovationstempo herrscht. So könnte ich mir zum Beispiel gut vorstellen, dass in zehn Jahren alle Grafikkarten und Spielkonsolen nach einem völlig anderen Verfahren als dem hier behandelten arbeiten werden. Im Moment werden im Rennen um die Nachfolgeschaft vor allem den in Kapitel 3 kurz erwähnten Voxel-basierenden Methoden gute Chancen eingeräumt.

A. Beispielprogramm

Wie schon in der Einleitung erwähnt, habe ich mich während der letzten Monate nicht nur mit den theoretischen Grundlagen der 3D-Grafik auseinandergesetzt, sondern auch ein praktisches Projekt umgesetzt: einen kleinen Software-Renderer mit dem Namen »d4«, geschrieben in der Programmiersprache D.

Bei einem Software Renderer handelt es sich um ein Programm, dass die 3D-Pipeline in Software implementiert, die Berechnungen zum Erzeugen eines Bildes also ausschließlich auf dem Hauptprozessor des Computers ausführt. Diese Vorgehensweise ist eigentlich seit Mitte der 90er-Jahre überholt, da mit der in praktisch allen PCs vorhandenen Grafikkarte ein hoch spezialisiertes und viel effizienteres Mittel zu Verfügung steht, um diese Berechnungen durchzuführen. Allerdings eignet sich das Programmieren eines Software Renderers vorzüglich dazu, sich mit vielen interessanten Details auseinanderzusetzen, über die man sich normalerweise dank leistungsfähiger Hardware und abstrahierender Programmierschnittstellen nicht weiter den Kopf zerbrechen muss.

Das Projekt ist auf der beiliegenden CD im Verzeichnis `d4` zu finden. Im Unterverzeichnis `src` befindet sich der rund 6000-zeilige Quellcode des Programms[1]. Die Dokumentation des Quellcodes sollte ausreichend genau sein, um Ihnen zu ermöglichen, darin zu stöbern. Ebenfalls nur für Programmierer interessant dürfte das Unterverzeichnis `build` sein. Es enhält die Konfigurationsdatei für das Programm *dsss*[2], mit dessen Hilfe das Projekt einfach aus den Quellen kompiliert werden kann.

Das Unterverzeichnis `bin` enthält die ausführbaren Dateien für Windows (`d4.exe`) und Linux (`d4`). Zum Ausführen des Programms sind einige externe Bibliotheken[3] erforderlich, nämlich die Multimedia-Bibliothek *SDL*[4], die Bibliothek *Assimp*[5] zum Einlesen der Modelldateien und die Bibliothek *DevIL*[6] zum Laden der Texturen. Für die 32 bit-Versionen von Microsoft Windows befinden sich die Bibliotheksdateien (`*.dll`) bereits im `bin`-Verzeichnis, sodass in den meisten Fällen keine zusätzlichen Schritte nötig sein sollten. Unter Linux müssen die entsprechenden Pakete der Distribution installiert sein. Für den Fall, dass dies nicht ausreichen sollte, sind aktuelle Pakete auf den Websites der Bibliotheken abrufbar.

Das Programm erwartet als ersten Parameter den Dateinamen der Szene, die geladen werden soll. Dank der beiden exzellenten Bibliotheken werden jeweils knapp 30 mehr oder weniger verbreitete Modell- und Texturformate unterstützt, eine genaue Auflistung findet sich auf den Websites der Bibliotheken und in der beiliegenden `README`-Datei. Um diesen Parameter anzugeben, kann das Programm natürlich von der Kommandozeile gestartet werden. Unter Windows dürfte es jedoch leichter sein, eine Verknüpfung zu der `.exe`-Datei an geigneter Stelle (etwa auf dem Desktop) anzulegen, und dann einfach eine Modelldatei per *Drag & Drop* auf die Verknüpfung »fallen zu lassen«. Im Verzeichnis `bin/models` befinden sich

[1] Die Dateien im Paket `assimp` sind lediglich eine 1:1-Übertragung der C-Schnittstelle der Assimp-Bibliothek in die Programmiersprache D, im Gegensatz zu den restlichen Programmteilen also kaum als meine eigene Arbeit zu betrachten.

[2] `http://www.dsource.org/projects/dsss`

[3] Bei einer Programmbibliothek handelt es sich – vereinfacht gesagt – um eine Sammlung von Funktionen, die kein eigenständiges Programm bilden, sondern von anderen Programmen als Hilfsmodul eingebunden werden.

[4] `http://www.libsdl.org/`

[5] `http://assimp.sourceforge.net/`

[6] `http://openil.sourceforge.net/`

neben einigen einfachen geometrischen Grundkörpern auch zwei Modelle eines Fantasy-Zwerges von einem Künstler mit dem Pseudonym *Psionic* (Unterverzeichnis `dwarf`), die sich einer Textur bedienen.

Das Programm akzeptiert noch zwei optionale Parameter, wenn es über die Kommandozeile aufgerufen wird: Mit dem Argument `fakeColors` wird jeder Eckpunkt in einer eigenen Farbe gezeichnet, wodurch die dem Modell zugrundeliegende Struktur sichtbar gemacht wird. `smoothNormals` weist das Programm an, beim Importieren von Modellen ohne Vertexnormalen Normalvektoren zu generieren, die eine »glatte« Oberfläche erzeugen (siehe Kapitel 3.1.2).

Noch ein paar Worte zur Bedienung: Die Kamera kann mittels der Tasten `W`, `S`, `A` und `D` bewegt und mit den Pfeiltasten gedreht werden. Die auch als `Shift` bezeichnete `Umschalt`-Taste beschleunigt dabei die Bewegungen. Die Tasten in der unteren Reihe der Tastatur steuern verschiedene Funktionen des Programms: `Y` wechselt zwischen Flat Shading, Gouraud Shading und Gouraud Shading mit Texturen; `X` schaltet den Wireframe-Modus ein und aus; `C` steuert das Backface-Culling; `V` lässt die Welt um den Koordinatenursprung drehen; `B` erzeugt einen animierten Farbverlauf im Hintergrund. Ein Druck auf die `Esc`-Taste beendet das Programm.

Abbildungsverzeichnis

Verwendete Literatur

Advanced Micro Devices, Inc. 2008 Advanced Micro Devices, Inc.: *ATI Radeon™ HD 4850 & ATI Radeon™ HD 4870 – GPU Specifications*. 2008. – Online abrufbar unter `http://ati.amd.com/products/Radeonhd4800/specs.html`. – Zugriffsdatum: 25. Februar 2009

Athen und Wigand 1967 Athen, Hermann ; Wigand, Klaus: *Elemente der Mathematik*. Zweite Auflage. Hannover: Hermann Schroedel Verlag KG, 1967

Bekaert 1999 Bekaert, Philippe: *Perspective projection matrix*. Mai 1999. – Online abrufbar unter `http://www.cs.kuleuven.ac.be/cwis/research/graphics/INFOTEC/viewing-in-3d/node8.html`. – Zugriffsdatum: 24. Januar 2009

Bertuch 2009 Bertuch, Manfred: Klötzchenwelten: Fotorealistische Computergrafik mit Voxel-Raycasting. In: *c't – magazin für computertechnik* Nr. 4/2009, S. 182–187

Bishop und Van Verth 2004 Bishop, Lars M. ; Van Verth, James M.: *Essential Mathematics for Games and Interactive Applications: A Programmer's Guide*. San Fransisco: Morgan Kaufmann, 2004

Brawley und Tatarchuk 2004 Brawley, Zoe ; Tatarchuk, Natalya: Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Map Tracing. In: Engel, Wolfgang F. (Hrsg.): *ShaderX3: Advanced Rendering with DirectX and OpenGL*. Hingham: Charles River Media, 2004, S. 135–154

Bronstein und Semendjajew 1981 Bronstein, I. N. ; Semendjajew, K. A. ; Grosche, G. (Hrsg.) ; Ziegler, V. (Hrsg.): *Taschenbuch der Mathematik*. 20. Auflage. Moskau: Verlag Nauka, 1981

Dorninger u. a. 1977 Dorninger, D. ; Nöbauer, W. ; Timischl, W.: *Lineare Optimierung und Anwendungen*. Wien: Österreichischer Bundesverlag für Unterricht, Wissenschaft und Kunst, 1977

Eberly 2002 Eberly, David: *Rotation Representations and Performance Issues*. Januar 2002. – Online abrufbar unter `http://www.geometrictools.com/Documentation/RotationIssues.pdf`. – Zugriffsdatum: 25. Februar 2009

Farrell 2006 Farrell, Joe: *Deriving Projection Matrices*. Oktober 2006. – Online abrufbar unter `http://www.codeguru.com/cpp/misc/misc/math/article.php/c10123__3/`. – Zugriffsdatum: 25. Februar 2009

Formella und Fellner 2005 Formella, Arno ; Fellner, W. D.: *Grundlagen der Computergrafik. Vorlesung WS 2004/2005*. März 2005. – Online abrufbar unter `http://trevinca.ei.uvigo.es/~formella/doc/ig04/cg.pdf`. – Zugriffsdatum: 25. Februar 2009

Hoag 1963 Hoag, David: *Apollo Guidance and Navigation. Considerations of Apollo IMU Gimbal Lock*. April 1963. – Online abrufbar unter `http://history.nasa.gov/alsj/e-1344.htm`. – Zugriffsdatum: 25. Februar 2009

Koch 2008 Koch, Thomas: *Rotation mit Quaternionen in der Computergrafik*, Fachhochschule Gelsenkirchen, Diplomarbeit, Januar 2008. – Online abrufbar unter `http://host12.informatik.fh-gelsenkirchen.de/pub/cgr/tkoch/Thomas%20Koch%20-%20Rotationen%20mit%20Quaternionen%20in%20der%20Computergrafik.pdf`. – Zugriffsdatum: 25. Februar 2009

Microsoft Corporation 2008 Microsoft Corporation: *The Direct3D Transformation Pipeline*. April 2008. – Online abrufbar unter `http://msdn.microsoft.com/en-us/library/bb206260(VS.85).aspx`. – Zugriffsdatum: 25. Februar 2009

nVidia Corporation 2009 nVidia Corporation: *GeForce GTX 285 – Specifications*. 2009. – Online abrufbar unter `http://www.nvidia.com/object/product_geforce_gtx_285_us.html`. – Zugriffsdatum: 25. Februar 2009

Quarteroni und Saleri 2006 Quarteroni, Alfio ; Saleri, Fausto: *Wissenschaftliches Rechnen mit MATLAB*. Berlin: Springer, 2006

Rudolph 2003 Rudolph, Alexander: *3D-Spiele mit C++ und DirectX. Schritt für Schritt zum Profi. (... in 21 Tagen)*. München: Markt+Technik Verlag, 2003

Shankel 2000 Shankel, Jason: *An Explanation of Quaternion Rotation in Euclidean Space*. Oktober 2000. – Online abrufbar unter `http://shankel.best.vwh.net/QuatRot.html`. – Zugriffsdatum: 25. Februar 2009

Vornberger und Fox 2006 Vornberger, Oliver ; Fox, Patrick: *Computergrafik. Vorlesung gehalten im SS 2006*. Juni 2006. – Online abrufbar unter `http://www-lehre.inf.uos.de/~cg/2006/PDF/skript.pdf`. – Zugriffsdatum: 25. Februar 2009

Weisstein 2008 Weisstein, Eric W.: *Rotation Matrix*. 2008. – Online abrufbar unter `http://mathworld.wolfram.com/RotationMatrix.html`. – Zugriffsdatum: 25. Februar 2009

Wikipedia 2009a Wikipedia: *Photon tracing – Wikipedia, The Free Encyclopedia*. 2009. – Online abrufbar unter `http://en.wikipedia.org/w/index.php?title=Photon_tracing&oldid=265001558`. – Zugriffsdatum: 25. Februar 2009

Wikipedia 2009b Wikipedia: *Reguläre Matrix – Wikipedia, Die freie Enzyklopädie*. 2009. – Online abrufbar unter `http://de.wikipedia.org/w/index.php?title=Regul%C3%A4re_Matrix&oldid=57089487`. – Zugriffsdatum: 25. Februar 2009

Wikipedia 2009c Wikipedia: *Zentralprojektion – Wikipedia, Die freie Enzyklopädie*. 2009. – Online abrufbar unter `http://de.wikipedia.org/w/index.php?title=Zentralprojektion&oldid=56564479`. – Zugriffsdatum: 25. Februar 2009

Zerbst u. a. 2004 Zerbst, Stefan ; Düvel, Oliver ; Anderson, Eike: *3D-Spiele-Programmierung. Professionelle Entwicklung von 3D-Engines und -Spielen. Kompendium*. München: Markt+Technik Verlag, 2004

Diese Arbeit entstand unter Verwendung des Open-Source-Satzsystems LaTeX mit dem KOMA-Script-Paket und Minion Pro als Textschrift. Die in diesem Dokument verwendeten Skizzen wurden in Adobe Illustrator® erstellt, die Abbildungen in Kapitel 3 sind Bildschirmfotos des Beispielprogramms.

Eine PDF-Version der Arbeit ist zusammen mit den Quelldateien dieses Dokuments im Internet unter `http://klickverbot.github.com/3d-mathematics` abrufbar. In digitaler Form ist diese Arbeit auch auf der beiliegenden CD enthalten, wo neben dem Beispielprogramm auch noch Kopien aller online zugänglichen Quellen gespeichert sind. Sollten Sie die CD-ROM nicht erhalten haben, sind meine Kontaktdaten auf der genannten Internetseite zu finden.