3D Echtzeit Grafik am Computer - Mathematischer Hintergrund

Sebastian Maisch

02.02.2004

Inhaltsverzeichnis

1 Einführung

Dreidimensionale Grafiken am Computer werden in der heutien Zeit immer bedeutender. Sie werden in den unterschiedlichsten Gebieten eingesetzt. So stellen Architekten und Designer ihre Produkte heute zuerst in so genannten CAD[1]-Programmen am Computer. In den Naturwissenschaften werden simulierte 3D-Umgebungen benutzt, um in der Realität schwer durchführbare Versuche darzustellen. Sowohl Mediziner, als auch Piloten üben ihren Beruf an Simulatoren, die eine 3D-Darstellung benötigen. Weder Filme, noch Computerspiele würden ohne 3D-Grafik in ihrer heutigen Form existieren.
Die Simulation und Darstellung dreidimensionaler Objekte am Computer ist nicht mehr wegzudenken. Man kann sich ein Haus auf dem Computer anschauen, schon bevor es gebaut ist. Bei Nichtgefallen kann es relativ einfach geändert werden, was in der Realität unter Umständen sehr kostenintensiv wäre.
Dabei wird im Allgemeinen immer über „3D-Grafik“ geredet. Es gibt aber mehrere verschiedene Ansätze, räumliche Strukturen auf den Bildschirm zu bringen. Hierbei sollte man anhand der Vort- und Nachteile abwägen, welcher Ansatz sich am besten für das jeweilige Anwendungsgebiet eignet.
Eine dieser Techniken wird Raytracing[2] genannt. Hierbei wird für jeden Bildpunkt des fertigen Bildes ein Lichtstrahl berechnet, über den dann die Farbe des Punktes bestimmt wird. Auf der einen Seite ist dieses Verfahren sehr rechenaufwendig und die Ermittlung eines einzelnen Bildes kann an einem normalen PC mehrere Minuten dauern, auf der anderen Seite liefert diese Technik Bilder, die qualitativ von Fotos nicht mehr zu unterscheiden sind. In Filmen wird diese Methode verwendet um Special Effects darzustellen.
Wenn man allerdings Grafiken in Echtzeit darstellen will, benötigt man ein Verfahren, das schnell genug ist, mindestens 15 Bilder in der Sekunde berechnen zu können, denn erst ab diese Ablaufgeschwindigkeit der Bilder, kann das Auge Einzelbilder nicht mehr als solche verarbeiten, man sieht fließende Bewegungen. Man sollte beachten, dass diese Zahl als absolute Untergranze gesehen werden muss.
Im Rahmen dieser Arbeit soll die Mathematik, die hinter letztgenanntem Verfahren steht, erläutert werden. Insbesondere wird der Schwerpunkt auf Computerspiele gesetzt, da es hierbei sehr darauf ankommt, dass die entstandenen Bilder möglichst realistisch aussehen, wodurch oft eine stärkere Optimierung benötigt wird als bei anderen Anwendungen.

[1] CAD = Computer Aided Design

[2] Raytracing: Ray = Strahl, tracing = den Verlauf nachvollziehen; bedeutet, dass bei dieser Technik der Verlauf von Lichtstrahlen berechnet und verfolgt wird, um eine korrekte Beleuchtung der Szene zu ermöglichen.

2 Mathematische Grundlagen

An dieser Stelle soll ein kurzer Überblick über die mathematischen Grundlagen gegeben werden, die für weiteres Verständnis wichtig sind. Dies soll keine vollständige Beschreibung der genannten Konzepte darstellen, stattdessen werden die für den Verlauf der Facharbeit wichtigen Informationen kurz vermittelt.

2.1 Vektoren

Die wohl wichtigste mathematische Struktur, die benötigt wird, ist der Vektor. Man bedient sich nicht nur der Vektoren im $\mathbb{R}^3$, sondern insbesondere auchc derer im $\mathbb{R}^2$ und $\mathbb{R}^4$. Letzterer wird hierbei nur benötigt um Multiplikationen mit 4x4 Matrizen zu ermöglichen. Nach einer Berechnung im $\mathbb{R}^4$ werden, damit der Vektor auch im $\mathbb{R}^3$ vergleichbar bleibt, alle Komponenten des Vektors durch dessen vierte Koordinate, den w-Wert geteilt, wodurch alle Vektoren an dieser Stelle den Wert 1 haben. Dies entspricht einer Projektion auf die Hyperebene $w = 1$. Da nun alle Vektoren im $\mathbb{R}^3$ vergleichbar sind, kann man die vierte Komponente vernachlässigen und mit dem dreidimensionalen Vektor arbeiten.[3]
Vektoren im $\mathbb{R}^3$ werden benötigt um Koordinaten auf Ebenen darzustellen. Dies können sowohl Punkte im Bildschirmkoordinatensystem als auch Texturkoordinaten sein. Auf beides werde ich später noch genauer eingehen.
Am häufigsten verwendet man den Vektor im $\mathbb{R}^3$. Er kann Positionen im Raum, aber auch Verschiebungen, Lichteinfallsrichtung und ähnliches festlegen. Auch viele physikalische Größen, die eine Rolle spielen (z.B. Impuls, Geschwindigkeit), werden durch diese Vektoren repräsentiert.
Die Rechenregeln für Vektoren, die im Rahmen dieser Facharbeit benötigt werden, gehen nicht über die Schulmathematik hinaus.

2.2 Matrizen

Die anderen beiden Strukture, die nötig sind, werden in der Schule nicht behandelt. Die erste ist die Matirx. Matrizen sind Zahlengebilde, die sich im Aussehen nicht von Determinanten unterscheiden. Sie haben eine gewisse Anzahl an Reihen und Spalten, und an jeder Position, die sich daraus ergibt steht eine Zahl. Die Matrizen, die zur Darstellung von 3D-Grafiken benötigt werden, sind 4x4 Matrizen. Das heißt sie haben sowohl vier Reihen als auch vier Spalten.[4]

Beispiel für eine 4x4 Matrix: $\begin{bmatrix} 15 & 2 & 5 & 6 \\ 1 & 58 & 3 & 1 \\ 2 & 8 & 9 & 2 \\ 26 & 4 & 38 & 4 \end{bmatrix}$

Matrizen sind deshalb so wichtig, da sie jede beliebige Transformation, die an Objekten im dreidimensionalen Raum ausgeführt werden soll, darstellen können. Diese Transformationen sind Translation[5], Skalierung[6] und Rotation[7]. Eine Matrix kann alle darstellen und auch jede beliebigt Kombination erzeugen.[8]
Das bedeutet, wenn man zum Beispiel zwei Matrizen betrachtet, die eine rotiert ein Objekt um einen bestimmten Winkel und eine bestimmte Achse, und die andere verschiebt das Objekt, dann ist das Produkt der beiden Matrizen eine Matrix die beides gleichzeitig durchführt.
Das Produkt C zweier Matrizen A und B ist folgendermaßend definiert:

[3] Siehe hierzu auch [Ebe01, S. 9 (2.1.4 Homogeneous Transformations)].
[4] Vgl. zu dem vorangegangenen Absatz [ZDA04, S. 153f].
[5] Translation: Verschiebung
[6] Skalierung: maßstabsgetreue Vergrößerung
[7] Rotation: Drehung
[8] Vgl. zu dem vorangegangenen und folgenden Absatz [ZDA04, S. 154].

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk} \tag{1}$$

Hierbei mussa A ebenso viele Spalten haben, wie B Zeilen. Diese Anzahl wird durch das n in obiger Gleichung repräsentiert. Zu beachten ist, dass diese Operation nicht kommutativ ist. Also $A \cdot B \neq B \cdot A$. Dies wiederum hat Auswirkungen auf die Reihenfolge, in der Transformationen angewandt werden. Wenn ein Objekt, das sich am Ursprung des Koordinatensystems befindet, zuerst um die Koordinaten-Achsen rotiert und danach an eine andere Stelle verschoben werden soll, muss der erste Faktor die rotierende Matrix sein, andernfalls würden die Transformationen in anderer Reihenfolge ausgeführt werden, was nicht zum gewünschten Ergebnis führt.
Bei der Matrizenmultiplikation gibt es , analog zur Multiklikation zweier Skalare, ein neutrales Element. Was bei letzterer die 1 ist, ist in der Matrizenmultiplikation die so genannte Einheitsmatrix.

4x4 Einheitsmatrix: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Die letzte wichtige Operation ist die Multiplikation einer Matrix mit einem Vektor. Dies wird auch Transformation genannt. Die Matrix muss hierbei ebenso viele Reihen haben wie der Vektor Koordinaten. Das Produkt hat dann genauso viele Koordinaten wie die Matrix Zeilen hat. Die Durchführung entspricht für jede Koordinate des Produktvektors einem Skalarprodukt einer Zeile der Matrix mit dem Ausgangsvektor. Die Formel für die Multiplikation einer 4x4 Matrix mit einem Vektor lautet wie folgt:

$$\begin{aligned} v_i &= \sum_{j=1}^{4} v_j a_{ij} \; ; \; i \in [1,4] \\ &\Rightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} = \begin{pmatrix} v_1 a_{11} + v_2 a_{12} + v_3 a_{13} + v_4 a_{14} \\ v_1 a_{21} + v_2 a_{22} + v_3 a_{23} + v_4 a_{24} \\ v_1 a_{31} + v_2 a_{32} + v_3 a_{33} + v_4 a_{34} \\ v_1 a_{41} + v_2 a_{42} + v_3 a_{43} + v_4 a_{44} \end{pmatrix} \end{aligned} \tag{2}$$

Auch hier ist wieder die Einheitsmatrix das neutrale Element und bildet den Vektor auf sich selbst ab.

2.3 Quaternions

Als letzte Grundlage sind die Quaternions vorzustellen. Sie sind eine weitere Möglichkeit eine Rotation im 3D-Raum darzustellen. Da sich in der Praxis ein Problem beim direkten Ausrechnen einer Rotationsmatrix ergibt, berechnet man zuerst ein Quaternion für eine Rotation und daraus die Matrix.[9] Dies wird später genauer behandelt. Der größte Vorteil zeigt sich aber erst dann, wenn man zwischen zwei verschiedenen Rotationszuständen interpolieren muss. Falls in diesem Fall eine Matrix benutzt würde, wäre der Vorgang sehr zeitaufwendig. Quaternions dagegen lassen sich sehr einfach interpolieren[10]. Dies soll hier aber nur der Vollständigkeit halber angesprochen werden und ist nicht Teil dieser Facharbeit.
Quaternions sind kurz gesagt vierdimensionale Zahlen.[11] Dabei darf man sie nicht mit einem Vektor vergleichen, der aus vier eindimensionalen Zahlen aufgebaut ist. Sie weisen vielmehr große Analogien zu den komplexen Zahlen auf, die auch in der Schule behandelt werden. Diese bestehen aus einem Real- und einem Imaginärteil, und können als zweidimensionale Zahlen aufgefasst werden.
Bei Quaternions wird dieses Konzept um 2 weitere Imaginärteile ergänzt. Ein Quaternion

[9] Vgl. [ZDA04, S. 205].
[10] interpolieren: Einen Wert an einem unbekannten Punkt zwischen mindestens zwei bekannten schätzen.
[11] Siehe hierzu auch [ZDA04, S. 203].

wird durch

$$q = w + xi + yj + zk \text{ [12]} \tag{3}$$

definiert, wobei w, x, y, und z Reelle Zahlen sind. Für i, j und k gilt:[13]

$$\begin{aligned} i^2 &= j^2 = k^2 = -1 \\ ij &= -ji = k \\ jk &= -ki = i \\ ki &= -ik = j \end{aligned}$$

In der Anwendung sind Quaternions Matrizen sehr ähnlich. Zwei Rotationen, die in Quaternions vorliegen, können analog zu den Matrizen dadurch kombiniert werden, dass man die beiden Quaternions multipliziert. Bei Beachtung der obigen Definitionen kann eine Multiplikation nach den normalen Multiplikationsregeln durchgeführt werden.[14]

$$\begin{aligned} q_0 q_1 &= (w_0 + x_0 i + y_0 j + z_0 k) \cdot (w_1 + x_1 i + y_1 j + z_1 k) \\ &= (w_0 w_1 - x_0 x_1 - y_0 y_1 - z_0 z_1) + (w_0 x_1 + x_0 w_1 + y_0 z_1 + z_0 y_1) \cdot i + \\ &\quad (w_0 y_1 + y_0 z_1 + y_0 w_1 + z_0 x_1) \cdot j + (w_0 z_1 + x_0 y_1 + y_0 x_1 + z_0 w_1) \cdot k \end{aligned} \tag{4}$$

Analog zur Multiplikation zweier Matrizen ist auch die Multiplikation von Quaternions nicht kommutativ. Auch hier existiert ein neutrales Element. Es hat die Eigenschaften $x = y = z = 0; w = 1$.[15]

[12] Vgl. [Ebe01, S. 11].
[13] Vgl. zu den vorangegangenen Definitionen [Ebe01, S. 11].
[14] Vgl. [Ebe01, S. 11].
[15] Vgl. [ZDA04, S. 206f].

3 Format und Verarbeitung der Daten im PC

Um zu verstehen, warum bei Echtzeitgrafik auf die eine oder andere Weise gerechnet wird, ist es wichtig einige Grundlagen zu klären. Dabei ist es nützlich zu wissen, in welchem Format die zu bearbeitenden Daten vorliegen. Des Weiteren soll an dieser Stelle der Weg dieser Daten bis zur Anzeige auf dem Bildschirm dargestellt werden. Gegen Ende des Kapitels wird noch erläutert, welche Operationen sich für Echtzeitberechnungen weniger gut eignen und deshalb vermieden werden müssen.

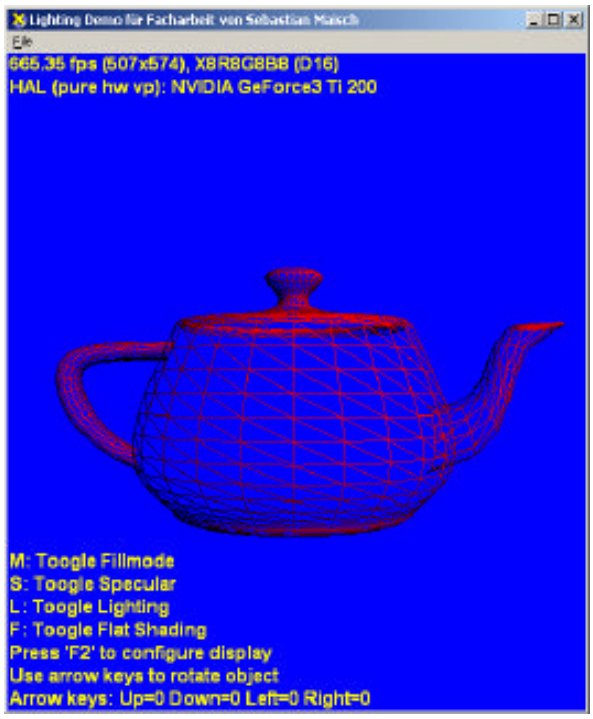


Abbildung 1: Aufbau einer Teekanne aus Dreiecken

3.1 Format und Darstellung der 3D Objekte

Doch zuerst soll das Format der Daten beschrieben werden. Die Technik, die sich heute durchgesetzt hat, beschreibt jedes Objekt durch beliebig viele Polygone[16]. Diese werden oft noch in Dreiecke zerlegt, da sie in gefüllter Form einfacher auf dem Bildschirm gezeichnet werden können als Vielecke. In Abbildung 1 kann man die Aufteilung einer Teekanne in Dreiecke sehen. Ein Dreieck ist genau durch seine Eckpunkte definiert. Diese werden als Vertices bezeichnet.[17] Ein Objekt, bei dem nur die Eckpunkte sichtbar sind, ist in Abbildung 2 zu sehen. Jeder Vertex hat aber noch weitere Attribute zusätzlich zu seiner Position im Raum. Denkbar wäre zum Beispiel eine Farbe, welche dann die des späteren Dreiecks bestimmt. Hierbei können in Dreiecken auch mehrere Farben auftauchen. Wenn dies der Fall ist, wird durch Interpolation der Farbwerte an den Eckpunkten des Dreiecks auf diesem ein Farbverlauf erzeugt.

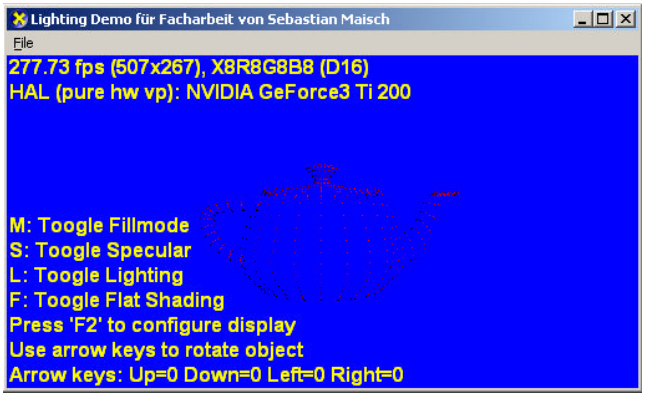


Abbildung 2: Vertices einer Teekanne

[16] Vielecke
[17] Vgl. [Zer00, S. 378].

Dadurch kann man die Objekte aber noch nicht mit Bildern versehen. Hierfür benutzt man so genannte Texturen. Dies sind Bilder, die um das Objekt herum gelegt werden. Man kann sie mit Tapete vergleichen, die auf eine Wand geklebt wird, um dieser Farbe zu verleihen. Um die dahinter stehende Technik, sie wird Texture Mapping genannt, benutzen zu können muss dem Rechner mitgeteilt werden, welche Textur für das Objekt, welches gerade gezeichnet wird, benutzt werden soll. Außerdem benötigen die Vertices dann einen zweiten Vektor, der die gedachte Position des Vertex auf der Textur festlegt. Dieser ist zweidimensional und wird als Texturkoordinate bezeichnet. Später wird aus ihnen und der Textur für jeden Punkt des Dreiecks, der auf dem Bildschirm dargestellt wird, ein Farbwert berechnet.[18]
Oft besitzt ein Vertex einen weiteren Vektor, der für die Lichtberechnung benötigt wird. Dieser wird Vertex Normal[19] genannt. Streng genommen ist es allerdings nicht möglich einen Normalvektor für einen Vertex zu berechnen, da ein solcher nur für Flächen, nicht aber für Punkte definiert ist. Deswegen werden hierfür die Normalvektoren aller angrenzenden Flächen herangezogen und der Durchschnitt gebildet. Dieser beschreibt den Vertexnormalvektor.[20]
Je nach Anwendung können in einem Vertex noch weitere Daten gespeichert sein. Da es aber im Normalfall ausreicht ihn durch seine Position, Texturkoordinaten, oder wahlweise eine Farbe, und einen Normalenvektor zu beschreiben, sollen diese Angaben an dieser Stelle genügen.

3.2 Architektur der 3D-Pipeline

Der folgende Teil beschäftigt sich mit der Architektur der für 3D-Grafik relevanten Hardware im PC. Das ist heute meist die Grafikkarte. Aktuelle Grafikkarten sind sehr stark auf den Umgang mit dreidimensionalen Objekten spezialisiert und können bestimmte Berechnungen sehr schnell durchführen. Man kann die Grafikkarte recht willkürlich in zwei Bereiche aufteilen, die so genannte 3D-Pipeline und den Grafikspeicher. Im Grafikspeicher werden sämtliche Daten, die gerade benötigt werden, gespeichert. Wenn sich ein Objekt nicht im Grafikspeicher befindet, kann es nicht ohne weiteres dargestellt werden. Neben den Vertexdaten für die Objekte beinhaltet der Grafikspeicher auch die Texturen.
Die 3D-Pipeline ist der Teil der Grafikkarte, auf dem alle nötigen Berechnungen ausgeführt werden. Man kann sie sich wie ein Fließband vorstellen, bei dem an jeder Station eine spezielle Rechnung durchgeführt wird. Zur Verdeutlichung soll hier Abbildung 3 dienen, welche die Pipeline von Direct3D darstellt. Letzteres ist eine Sammlung von Datenstrukturen und Funktionen, die es einem Programmierer ermöglichen ohne genaue Kenntnis der Grafikhardware eines PCs diese trotzdem effizient zu nutzen. Die 3D-Pipeline beinhaltet einige Bereiche, die den Rahmen der Facharbeit sprengen würden, und deshalb nicht berücksichtigt werden. Man kann aber sehr gut deren Aufbau und die Reihenfolge der Berechnungen entnehmen. Am Beginn stehen die Vertexdaten, die sich im Grafikspeicher befinden. Diese werden in die Geometry Pipeline gegeben, in der schon die erste Besonderheit auftritt.
Der Programmierer muss vor der Bearbeitung der Daten in der 3D-Pipeline entscheiden ob er auf die so genannte Fixed Function Pipeline oder auf so genannte Shader zurückgreifen will, um seine Daten berechnen zu lassen. Die Fixed Function Pipeline bietet, wie ihr Name schon andeutet eine feste Funktion zum Bearbeiten der Daten. Diese kann nur durch einige Parameter variiert werden. Shader ermöglichen es im Gegensatz dazu eigene Berechnungen und Funktionen zu schreiben. Sie werden nur auf neuen Grafikkarten unterstützt, auf älteren muss die Fixed Function Pipeline benutzt werden.
In der Geometry Pipeline werden die Vertexdaten nun je nach Entscheidung des Programmierers zur „Transformation and Lightning Engine“ oder zu einem „Vertex-Shader“ weitergereicht. In beiden Fällen passiert nahezu das gleiche. Die Daten werden durch bestimmte Matrizen transformiert und gegebenenfalls beleuchtet. Am Ende dieser Station liegen sie in

[18] Vgl. zu dem vorangegangenen Abschnitt [Zer00, S. 412f].
[19] Vertex Normalvektor
[20] Vgl. zu dem vorangegangenen Absatz [Zer00, S. 411f].

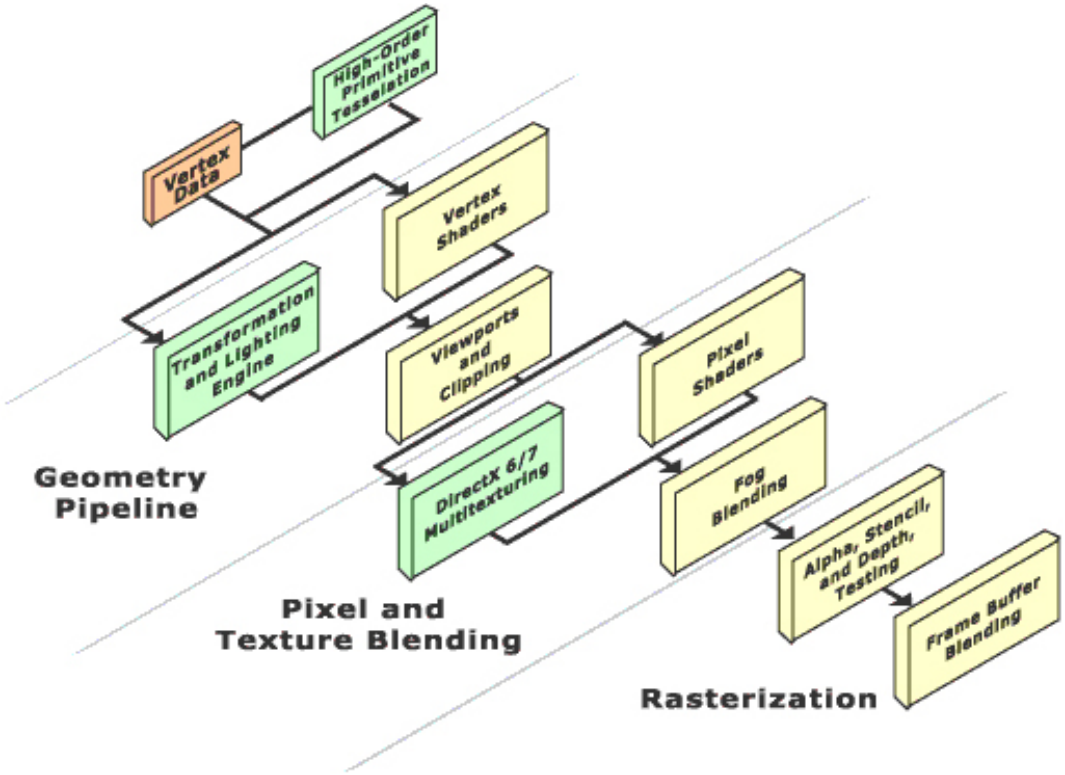


Abbildung 3: Die 3D-Pipeline am Beispiel von Direct3D

den Koordinaten vor, in denen sie auf dem Bildschirm auftauchen. Damit nicht Objekte, die außerhalb des Bildschirms liegen, weiterberechnet werden, müssen diese in der „Clipping Unit“ aussortiert werden.
Da nach diesem Schritt sicher ist, welcher Pixel[21] auf dem Bildschirm zu welchem Dreieck im Grafikspeicher gehört, läuft die Pipeline nun auf Pixelebene weiter. Beim „Pixel and Texture Blending“ steht nun ebenfalls die Entscheidung an, mit „Pixel-Shadern“ oder dem „Multitexturing“zu arbeiten. Auch hier werden von beiden Modulen ähnliche Berechnungen durchgeführt. Dem Pixel wird, über eine oder mehrere Texturen ein Farbwert zugewiesen. Bei der Benutzung von Pixel-Shadern besteht ein weiteres Mal die Möglichkeit Beleuchtungsberechnungen durchzuführen. In den letzen drei Schritten werden noch einige Tests und auf Wunsch eingeschränkte Berechnungen mit dem alten Farbwert auf dem Bildschirm durchgeführt.[22]

3.3 Operationen, die bei der Verarbeitung der Daten vermieden werden sollen

Da die Darstellung der Grafik sehr aufwendig ist, ist es wichtig prozessorlastige Operationen zu vermeiden, wo dies möglich ist. Besonders aufwendig sind für einen Prozessor alle nichtlinearen Verknüpfungen, also alles was nicht über Addition, Subtraktion, Multiplikation oder Division berechnet werden kann. Dazu gehören zum Beispiel trigonometrische Funktionen oder Wurzeln. Der Computer kann diese zwar in Bruchteilen von Millisekunden berechnen, aber auch das ist in den Zeitdimensionen, die bei Echtzeitgrafik erforderlich sind, noch sehr viel. Sinus und Kosinus sind für Rotationen unabdingbar, doch wenn für viele Vektoren die gleiche Rotation durchgeführt werden muss, kann man sehr viel Rechenzeit sparen, indem

[21] Pixel: einzelner Punkt auf dem Monitor
[22] Vgl. zu den vorangegangenen Absätzen [ZDA04, S. 362f].

man für diese Rotation eine Matrix erstellt und sie auf die einzelnen Vektoren anwendet. Wenn man nun eine solche Matrix hat und diese mit dem Vektor multipliziert, kann man alle anderen Transformationen in diese mit einberechnen und dadurch noch mehr Rechenzeit sparen, die durch die Multiplikation des Vektors mit eventuellen Matrizen für Translation oder Skalierung verloren gegangen wäre. Wurzeln sind oft bei Längenberechnungen nötig. Allerdings kann man meist auf den genauen Betrag verzichten, da nicht die absolute Länge einer Strecke gewünscht ist, sondern nur ein Vergleich zweier Strecken. Hier kann man auch das Quadrat der Länge vergleichen, um auf die richtigen Ergebnisse zu kommen, und spart dabei wertvolle Rechenzeit.

4 Grundtransformationen und deren Matrizen

Weiter oben wurden schon die Grundregeln für Matrizen und Quaternions besprochen und erwähnt, dass man beide für Transformationen im dreidimensionalen Raum verwenden kann. Wie dies genau funktioniert, soll in diesem Kapitel erklärt werden. Zu den Transformationen gehören Skalierung, Rotation und Translation. Diese können alle durch Matrizen dargestellt werden, Rotationen auch durch Quaternions.

4.1 Skalierung

Die einfachste der drei Transformationen ist die Skalierung. Hierbei soll jeder Vektor, der mit der Matrix multipliziert wird, mit einem bestimmten Faktor für jede Achse (s_x, s_y, s_z) vervielfacht werden. Es gilt also folgende Beziehung:

$$M_S \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x x \\ s_y y \\ s_z z \\ 1 \end{pmatrix} \quad [23] \tag{5}$$

M_S ist hierbei die gesuchte Matrix. Über die Formel für Multiplikationen zwischen Matrizen und Vektoren kann man sie sehr leicht herleiten. Für die erste Koordinate des Vektors möchte ich dies zur Erläuterung durchführen:

$$m_{11}x + m_{12}y + m_{13}z + m_{14} \cdot 1 = s_x x$$

Diese Gleichung ist nur dann erfüllt wenn

$$\begin{aligned} m_{12} &= m_{13} = m_{14} = 0 \\ m_{11} &= s_x \end{aligned}$$

Analog kann dies für alle anderen Koordinaten des Vektors durchgeführt werden, bis man für M_S folgendes Ergebnis erhält:[24]

$$M_S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad [25] \tag{6}$$

4.2 Translation

Als nächstes soll auf die Translation eingegangen werden. Der Ausgangsvektor wird bei dieser um einen beliebigen anderen Vektor $(\vec{T})$ verschoben. Auch hier lässt sich wieder eine Gleichung aufstellen:

$$M_T \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{pmatrix} \quad [26] \tag{7}$$

Man kann für die gesuchte Matrix M_T nun genauso vorgehen wie bei der Skalierung. Auch hier möchte ich am Beispiel der ersten Koordinate die Rechenschritte erläutern:

$$m_{11}x + m_{12}y + m_{13}z + m_{14} = x + t_x$$

[23] Vgl. [BW99, Representing points and transformations].
[24] Vgl zu der vorangegangenen Herleitung [Zer].
[25] Vgl. [BW99, Representing points and transformations].
[26] Vgl. [BW99, Representing points and transformations].

Diese Gleichung lässt sich mit

$$\begin{aligned} m_{12} &= m_{13} = 0 \\ m_{11} &= 1 \\ m_{14} &= t_x \end{aligned}$$

lösen.[27] Die Matrix die sich letztendlich für M_T ergibt, ist folgendermaßen aufgebaut:

$$M_T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad [28] \tag{8}$$

4.3 Rotation

Die letzte Transformation ist die Rotation. Sie ist etwas komplizierter als die beiden vorhergehenden und ist am einfachsten zu verstehen, wenn man sie in drei verschiedene Fälle aufteilt. Es geht hierbei um die Rotation um die Achsen des Koordinatensystems. Der allgemeine Fall kann dann durch eine Multiplikation der drei Teilrotationen erreicht werden. Repräsentativ soll hier die Rotation um die x-Achse hergeleitet werden. Hierbei bleibt bei der Transformation die x-Komponente des Vektors konstant. Dadurch reduziert sich das Problem auf eine Rotation in der Ebene.

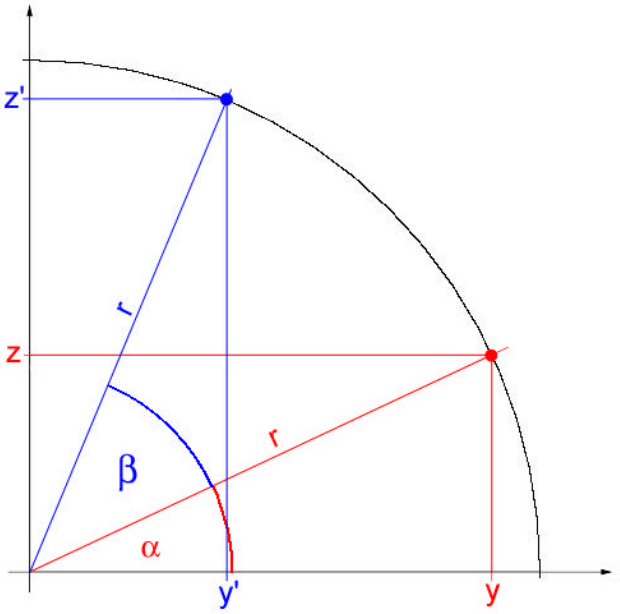


Abbildung 4: Rotation eines Punktes um den Winkel β auf der YZ-Ebene

Abbildung 4 zeigt den Ausgangs- (rot) und Endvektor (blau) der Rotation. Der Rotationswinkel ist β. Gesucht werden für die Rotation Gleichungen, in denen y' und z' durch y

[27] Vgl. zu der vorangegangenen Herleitung [Zer].
[28] Vgl. [BW99, Representing points and transformations].

und z ausgedrückt ist. Diese lassen sich wie folgt herleiten: [29]

$$
\begin{aligned}
z &= r \cdot \sin\alpha && (9)\\
y &= r \cdot \cos\alpha && (10)\\
z' &= r \cdot \sin\alpha + \beta \\
&= r \cdot (\sin\alpha \cdot \cos\beta + \cos\alpha \cdot \sin\beta) \\
&= r \cdot \sin\alpha \cdot \cos\beta + r \cdot \cos\alpha \cdot \sin\beta \\
&= z \cdot \cos\beta + y \cdot \sin\beta && (11)\\
y' &= r \cdot \cos\alpha + \beta \\
&= r \cdot (\cos\alpha \cdot \cos\beta - \sin\alpha \cdot \sin\beta) \\
&= r \cdot \cos\alpha \cdot \cos\beta - r \cdot \sin\alpha \cdot \sin\beta \\
&= y \cdot \cos\beta - z \cdot \sin\beta && (12)
\end{aligned}
$$

Dadurch ergibt sich folgende Gleichung, die die Rotationsmatrix erfüllen muss:

$$
M_{Rx} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \cdot \cos\beta - z \cdot \sin\beta \\ y \cdot \sin\beta + z \cdot \cos\beta \\ 1 \end{pmatrix} \quad [30] \qquad (13)
$$

Hieraus kann man wiederum M_{Rx} bestimmen:

$$
M_{Rx} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\beta & -\sin\beta & 0 \\ 0 & \sin\beta & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad [31] \qquad (14)
$$

Analog lassen sich die Rotationsmatrizen für die anderen beiden Achsen herleiten:

$$
M_{Ry} = \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad [32] \qquad (15)
$$

$$
M_{Rz} = \begin{bmatrix} \cos\beta & -\sin\beta & 0 & 0 \\ \sin\beta & \cos\beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad [33] \qquad (16)
$$

Eine Weitere Operation die bei Rotationsmatrizen oft benötigt wird, ist das Rückgängigmachen einer Rotation. Hierbei kann man sich vorstellen, dass nun anstatt des Vektors die Koordinatenachsen um die Winkel/Achsen Kombination rotiert werden, deren Drehung rückgängig gemacht werden soll. Wenn man nun die Einheitsvektoren betrachtet, die in Richtung der

[29] Vgl. [Kwo98].
[30] Vgl. [BW99, Representing points and transformations].
[31] Vgl. [Kwo98].
[32] Vgl. [Kwo98].
[33] Vgl. [Kwo98].

Koordinaten laufen erhält man für eine Rotation um die z-Achse:

$$\begin{aligned}\vec{X} &= \begin{pmatrix}1\\0\\0\end{pmatrix};\ \vec{Y} = \begin{pmatrix}1\\0\\0\end{pmatrix}\\ \vec{X'} &= \begin{pmatrix}\cos\beta\\\sin\beta\\0\end{pmatrix};\ \vec{Y'} = \begin{pmatrix}-\sin\beta\\\cos\beta\\0\end{pmatrix}\end{aligned}$$

Jetzt kann man auf dem oben beschriebenen Weg die entsprechende Rotationsmatrix berechnen und kommt auf folgendes Ergebnis:

$$M_{Rz} = \begin{bmatrix}\cos\beta & \sin\beta & 0 & 0\\ -\sin\beta & \cos\beta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix} = \begin{bmatrix}x'_1 & x'_2 & x'_3 & 0\\ y'_1 & y'_2 & y'_3 & 0\\ z'_1 & z'_2 & z'_3 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\ ^{34} \tag{17}$$

Dies gilt ebenfalls für jede beliebige andere Rotationsmatrix. Daher kann sie in manchen Fällen schneller erstellt werden, wenn Achsen bekannt sind. Über diese Vereinfachung ist auch eine direkte Rotation um alle drei Koordinatenachsen gleichzeitig möglich.

4.4 Rotation durch Quaterions

Bei Rotationen über drei Achsen tritt aber oft ein Problem auf. Dies hängt damit zusammen, dass Rotationen immer an globalen Achsen durchgeführt werden, und nicht berücksichtigen, dass das rotierende Objekt eventuell schon einmal rotiert wurde, und damit die lokalen Achsen des Objekts nicht mehr mit den globalen übereinstimmen. Schlimmstenfalls kann es soweit kommen, dass eine lokale Achse genau auf einer anderen globalen liegt, also z.B. die lokale x-Achse auf der globalen y-Achse. Das Problem das nun besteht ist, dass alle weiteren Rotationen auf x- und y-Achse von nun an identisch sind. Dieser Zustand wird „Gimbal Lock“ oder Kardanring-Blockade genannt und tritt bei manchen Trgheitsnavigationssystemen in Flugzeugen oder Schiffen ebenfalls auf. Der Gimbal Lock kann dadurch verhindert werden, dass Rotationen nicht über Matrizen, sondern über Quaternions berechnet werden.[35]
Da aber in der weiter oben beschriebenen 3D-Pipeline Quaternions nicht unterstützt werden, müssen diese, nachdem der endgültige Rotationszustand bekannt ist, in eine Rotationsmatrix umgewandelt werden.[36]
Auf den Beweis wird an dieser Stelle verzichtet, da er fundierte Kenntnisse der Quaternionalgebra erfordert, die zu umfangreich ist um sie hier wiederzugegeben. Dies betrifft auch die Umwandlung eines Quaternions in eine Matrix.[37]
Der Beweis ist im Anhang A dieser Facharbeit zu finden. Quaternions können ebenso wie Matrizen durch Multiplikation kombiniert werden. Dies wird allerdings im Gegensatz zu Matrizen weniger dazu verwendet eine Rotation um mehrere Koordinatenachsen zu erstellen, denn dies ist mit Quaternions sehr einfach, sondern um eine Rotation schrittweise ablaufen zu lassen.
Ein Quaternion, das eine Rotation um eine beliebige Achse $\vec{U}$ der Länge 1 um den Winkel 2θ darstellt ist folgendermaßen aufgebaut:

$$q = \cos\theta + \sin\theta \cdot (u_0 i + u_1 j + u_2 k)\ ^{38} \tag{18}$$

[34] Vgl. [Kwo98] und [BW99, Representing points and transformations].
[35] Vgl. [ZDA04, S. 205f].
[36] Vgl. [ZDA04, S. 203].
[37] Siehe hierzu [Ebe01, S. 17].
[38] Vgl. [Ebe01, S. 12].

Es gilt also:

$$\begin{aligned} w &= \cos\theta \\ x &= u_0 \cdot \sin\theta \\ y &= u_1 \cdot \sin\theta \\ z &= u_2 \cdot \sin\theta \end{aligned}$$

5 Koordinatensysteme und Projektion

Als nächstes sollen die Koordinatensysteme, die auftauchen, wenn man sich mit dem Thema 3D-Grafik beschäftigt, näher betrachtet werden. Hier soll ein Überblick über die wichtigsten gegeben werden, der auch Umwandlungen von Objekten des einen Systems in ein anderes nicht ausschließt.

5.1 Das Bildschirmkoordinatensystem

Das offensichtlichste Koordinatensystem, weil das einzig „reale“ ist das Bildschirmsystem. Der Ursprung dieses Systems liegt in der oberen linken Ecke des Bildschirms, die Achsen verlaufen dementsprechend nach rechts bzw. unten. Für die Berechnungen wird allerdings davon ausgegangen, dass der Ursprung in der Mitte des Bildschirms liegt. Nachdem ein Punkt auf dem Bildschirm feststeht, muss dieser nur noch um die halbe Bildschirmbreite nach rechts und um die halbe Bildschirmhöhe nach unten verschoben werden, damit die Punkte auch wirklich da auftauchen wo die sollen. Eine Einheit dieses Systems ist durch die Bildschirmauflösung festgelegt und genau ein Pixel[39] groß. Dieses System taucht bei der Arbeit mit 3D-Grafik nur selten auf.[40]

5.2 Das Weltkoordinatensystem

Ein ebenfalls sehr offensichtliches System ist das Weltkoordinatensystem. Es ist das System in dem die meisten Operationen ausgeführt werden. In ihm sind die Positionen der einzelnen Objekte festgelegt. Der Ursprung dieses Systems ist nicht direkt festgelegt, doch er wird sich oft in der Mitte der darzustellenden „Welt“ befinden. Auch die Einheit ist frei wählbar, und kann durch die Größe der Objekte auf in der Realität bekannte Maße gesetzt werden. Das heißt, wenn die einzelnen Objekte sehr klein gewählt werden, wird eine Einheit in diesem System größer und umgekehrt. Diese Festlegung ist aber oft nicht nötig. Sogar die Achsen können willkürlich gewählt werden. Sie müssen nur immer im rechten Winkel zueinander stehen. Selbst die Frage ein Rechts- oder Linkshändersystem zu wählen, ist nicht weiter bedeutend, man muss sich nur durchgängig daran halten.[41]

5.3 Das Kamerakoordinatensystem

Um Objekte sinnvoll auf den Bildschirm zu projizieren ist es nötig ein weiteres System einzuführen. Dieses hat die gleichen Einheiten wie das Weltkoordinatensystem, aber der Ursprung ist verschoben, und die Achsen können im Gegensatz zum Weltkoordinatensystem rotiert sein. Der Ursprung befindet sich in diesem System an der Position des Betrachters. Man kann den Betrachter auch als imaginäre Kamera auffassen, und deshalb wird das System Kamerakoordinatensystem genannt. Die z-Achse zeigt in diesem System entweder in Blickrichtung der Kamera oder entgegen dieser. Hierdurch wird festgelegt, ob man ein Links- oder Rechtshändersystem benutzt. Im weiteren Verlauf werden wir nur Linkshändersysteme benutzen, da in diesem Fall die Werte der z-Achse vor dem Betrachter positiv sind, was sinnvoll ist. Theoretisch könnte man sich aber auch anders entscheiden. Die beiden anderen Achsen des Systems gehen vom Betrachter aus gesehen nach rechts (x-Achse) bzw. oben (y-Achse).[42]

5.4 Das Objektkoordinatensystem

Das letzte System ist das Objektkoordinatensystem. Hierbei bekommt jedes Objekt ein lokales Koordinatensystem, das ihm ermöglicht relativ einfach um die eigene Achse zu rotieren.

[39] Bildpunkt
[40] Vgl. zu dem vorangegangenen Absatz [Zer00, S. 381].
[41] Vgl. zu dem vorangegangenen Absatz [Zer00, S. 381].
[42] Vgl. zu dem vorangegangenen Absatz [Zer00, S. 392].

Der Ursprung dieses Systems befindet sich meist in der Mitte des Objekts. Die Achsen sind relativ beliebig, ebenso die Einheiten.[43]

5.5 Das Umwandlung zwischen den Systemen

Anhand dieser Koordinatensysteme lässt sich der „Weg“, den ein Objekt gehen muss bis es korrekt am Bildschirm angezeigt wird, erkennen. Zu Beginn liegen die Daten im Objektkoordinatensystem vor, werden dann über verschiedene Matrizen zuerst ins Welt-, dann ins Kamerazuletzt ins Bildschirmkoordinatensystem transformiert. Hier treten drei Matrizen auf, die in der Geometry Pipeline mit den Vertex-Daten verrechnet werden.

5.5.1 Die Weltmatrix

Um ein Objekt vom Objekt- ins Weltkoordinatensystem zu transformieren sind keine neuen Berechnungen nötig. Die benötigte Matrix wird „World-Matrix“ genannt und ist eine Kombination aus beliebig vielen Skalierungs-, Translations- und Rotationsmatrizen. Diese werden miteinander multipliziert um die World-Matrix zu ergeben. Dabei ist es wichtig, dass Rotationen um die objekteigenen Achsen nur durchgeführt werden können, bevor das Objekt verschoben ist.

5.5.2 Die Viewmatrix

Interessanter ist hier die so genannte „View-Matrix“. Ihre Aufgabe ist es alle Objekte in das Koordinatensystem der Kamera zu übertragen. Dies geschieht dadurch, dass zunächst die lokalen Achsen der Kamera an denen des Weltkoordinatensystems ausgerichtet werden. Dafür müssen alle Rotationen rückgängig gemacht werden, die auf die Kamera angewendet wurden. Weiter oben wurde schon erwähnt, dass man Rotationen sehr einfach über die Koordinatenachsen des lokalen Systems (in diesem Fall das der Kamera) rückgängig manchen kann. Da diese Achsen in den meisten Fällen sowieso vorliegen, da sie für Bewegungen benötigt werden, kann die benötigte Rotationsmatrix sehr einfach über die Vektoren $\vec{U}$, dem so genannten „Up-Vector“, der die lokale y-Achse beschreibt, $\vec{R}$, dem „Right-Vector“, der die lokale x-Achse beschreibt und $\vec{D}$, dem so genannten „Direction-Vector“, der die lokale z-Achse beschreibt, erstellt werden.[44]

$$M_R = \begin{bmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ d_x & d_y & d_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad [45] \tag{19}$$

Nun muss die Kamera noch zum Ursprung verschoben werden. Dies geschieht durch Translation um den negativen Positionsvektor der Kamera ($\vec{P}$).

$$M_T = \begin{bmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad [46] \tag{20}$$

[43] Vgl. zu dem vorangegangenen Absatz [Zer00, S. 378].
[44] Vgl. zu dem vorangegangenen Absatz [BW99, Viewing transformation].
[45] Vgl. [BW99, Viewing transformation].
[46] Vgl. [BW99, Viewing transformation].

Damit ist die Kamera am Ursprung. Die View-Matrix kann nun über eine Multiplikation dieser beiden Matrizen erstellt werden.

$$M_V = M_R \cdot M_T = \begin{bmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ d_x & d_y & d_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} r_x & r_y & r_z & -\vec{R} \circ \vec{P} \\ u_x & u_y & u_z & -\vec{U} \circ \vec{P} \\ d_x & d_y & d_z & -\vec{D} \circ \vec{P} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad [47] \quad (21)$$

5.5.3 Perspektivische Projektion und Projektionsmatrix

Jetzt können die einzelnen Vektoren auf den Bildschirm transformiert werden. Auch dies wird wieder in zwei Schritte aufgeteilt. Um herauszufinden welche Punkte später auf dem Bildschirm dargestellt werden, gibt es eine sehr einfache Bedingung. Jeder dieser Punkte muss sich nach der Transformation mit der so genannten „Projection-Matrix“ im Würfel $[-1;+1]^3$ befinden. Aus den x- und y-Koordinaten wird dann die Position auf dem Bildschirm berechnet. Aber auch die z-Koordinate ist wichtig.
Da nämlich alle Punkte, die auf einem Strahl durch den Ursprung liegen auf einen einzelnen Punkt abgebildet werden, ist es nötig zu wissen, welcher dieser Punkte am nächsten am Ursprung liegt. Dafür gibt es den so genannten „Z-Buffer“ Algorithmus. Hierfür wird im Speicher ein Feld angelegt, das ebenso viele Werte aufnehmen kann, wie es Bildpunkte auf dem Bildschirm gibt. Wird nun ein Punkt auf den Bildschirm gemalt, wird dessen z-Wert mit dem z-Wert im Z-Buffer an der Position des Punktes verglichen. Ist der z-Wert des zu malenden Punktes kleiner als der schon vorhandene, wird der Punkt auf den Bildschirm gemalt, und sein z-Wert in den Z-Buffer geschrieben.
Für die Projektionsmatrix wird also für die y- und x-Koordinate die Projektion des Aus-

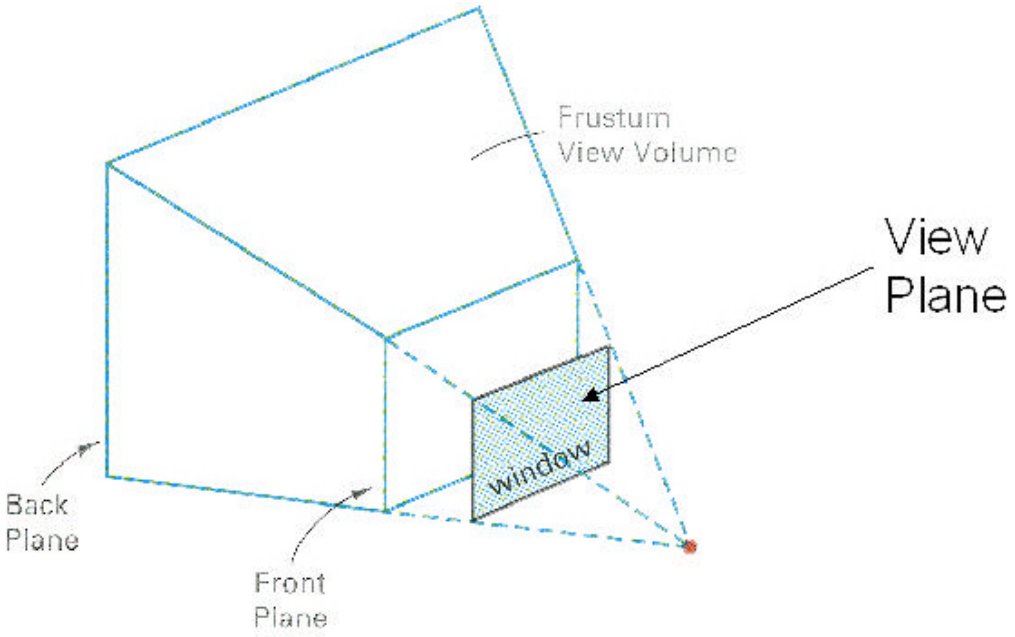


Abbildung 5: Das Viewfrustum

gangspunktes auf eine Ebene, die den Bildschirm darstellt, und für die z-Koordinate ein Wert, der je nach Abstand zum Ursprung zunimmt, gesetzt. In Abbildung 5 sieht man, in welchem Bereich sich die Punkte vor der Transformation befinden. Dieser Bereich wird durch

[47] Vgl. [BW99, Viewing transformation] und [Mic03, D3DXMatrixLookAtLH].

einen Pyramidenstumpf gebildet, dessen gedachte Spitze sich an der Position des Betrachters befindet. Dieser wird vorne und hinten durch die „Front Plane“ und die „Back Plane“ begrenzt. Beide Ebenen stehen senkrecht auf der z-Achse und schneiden diese an bestimmten Positionen, die mit z_{near} und z_{far} bezeichnet werden. Die Punkte, die sich in diesem so genannten „View-Frustum“ befinden, müssen nun auf die Ebene, die in der Abbildung mit „window“ bezeichnet ist, transformiert werden.

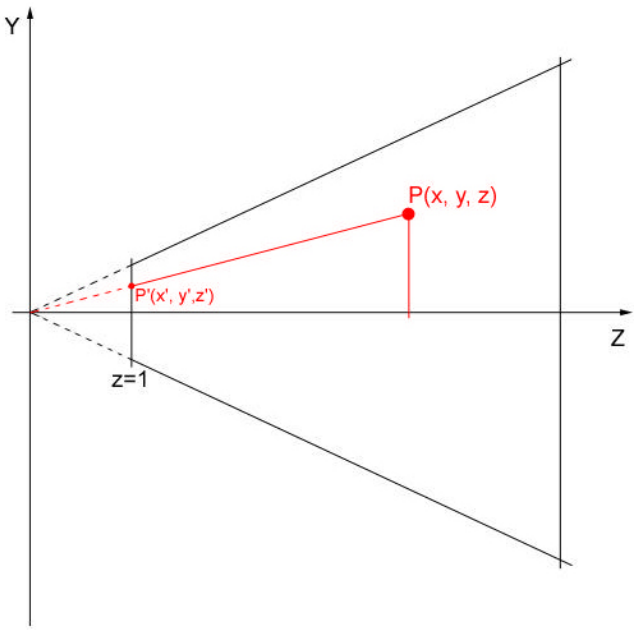


Abbildung 6: Projektion von P auf die Ebene $z = 1$

Zunächst sollen hier einige Vereinfachungen vorgenommen werden, die zum Teil später wieder korrigiert werden. Um den Gedankenweg übersichtlich zu halten, ist es einfacher anfangs von einer quadratischen Projektionsebene auszugehen. Die Position dieser Ebene ist so gewählt, dass sie auf der z-Achse senkrecht steht und diese bei $z = 1$ schneidet. Wenn man nun die YZ-Ebene betrachtet, gelten die Verhältnisse aus Abbildung 6. Aus dieser Abbildung kann man über ähnliche Dreiecke folgende Beziehungen ableiten:[48]

$$\frac{y}{y'} = \frac{z}{z'} \Rightarrow y' = \frac{y \cdot z'}{z}$$

Da als Vorraussetzung gilt $z' = 1$, lautet die endgültige Formel:

$$y' = \frac{y}{z} \quad [49] \tag{22}$$

Ebenso kann man x' herleiten, so dass gilt:

$$x' = \frac{x}{z} \quad [50] \tag{23}$$

Um dies in einer Matrix darzustellen, sollen einige Voraussetzungen nochmals aufgegriffen werden. Um einen vierdimensionalen Punkt in die dritte Dimension zu projizieren, werden sämtliche anderen Koordinaten des Punktes durch den Wert der vierten Koordinate geteilt.[51] Da in diesem Fall sowohl x, als auch y durch z geteilt werden müssen, kann man eine Matrix erstellen, in der z als vierte Koordinate auftaucht. Eine solche Matrix wäre diese:

$$M_Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad [52] \tag{24}$$

[48] Vgl. zu den vorangegangenen Absätzen [BW99, Perspective projection].
[49] Vgl. [BW99, Perspective projection].
[50] Vgl. [BW99, Perspective projection].
[51] Vgl. [BW99, Perspective projection].
[52] Vgl. [BW99, Perspective projection] und [Mic03, D3DXMatrixPerspectiveFovLH].

Bei der Multiplikation mit einem Vektor erhält man das folgende Ergebnis:

$$M_Q \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \cdot \begin{pmatrix} x \\ y \\ z \\ z \end{pmatrix}$$

Wenn man den Ergebnisvektor in die dritte Dimension projiziert, erhält man für den Endpunkt:

$$P' = (\frac{x}{z} \; \frac{y}{z} \; 1)$$

Nun besitzt allerdings die z-Koordinate einen konstanten Wert, und nicht, wie benötigt, einen Wert, der je nach Abstand vom Ursprung aus in z-Richtung ansteigt. Um diesen dennoch zu erhalten, muss die Matrix um zwei Parameter erweitert werden:

$$M_Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad [53] \tag{25}$$

Dadurch ergibt sich für z' folgendes Ergebnis:

$$z' = \frac{a \cdot z + b}{z} \tag{26}$$

Um a und b zu bestimmen stellt man nun zwei Gleichungen auf. Dabei wird festgelegt, dass bei Projektion eines Punktes, für den gilt $z = z_{far}$, $z' = 1$ gelten soll. Ebenso soll für einen Punkt, der den z-Wert z_{near} besitzt, $z' = 0$ sein. Die sich ergebenden Gleichungen können dann wie folgt aufgelöst werden:

$$\begin{aligned} \frac{a \cdot z_{near} + b}{z_{near}} &= 0 \\ a \cdot z_{near} + b &= 0 \\ b &= -a \cdot z_{near} \end{aligned}$$

$$\begin{aligned} \frac{a \cdot z_{far} + b}{z_{far}} &= 1 \\ a \cdot z_{far} + b &= z_{far} \\ a &= \frac{z_{far} - b}{z_{far}} \end{aligned}$$

a in b eingesetzt ergibt: [54]

$$\begin{aligned} b &= -\frac{(z_{far} - b) \cdot z_{near}}{z_{far}} \\ z_{far} \cdot b &= -(z_{far} \cdot z_{near} - z_{near} \cdot b) \\ z_{far} \cdot b - z_{near} \cdot b &= -z_{far} \cdot z_{near} \\ b &= \frac{-z_{far} \cdot z_{near}}{z_{far} - z_{near}} \end{aligned} \tag{27}$$

[53] Vgl. [BW99, Perspective projection].
[54] Vgl. [BW99, Perspective projection] und [Mic03, D3DXMatrixPerspectiveFovLH].

Wenn man dies wiederum in a einsetzt ergibt sich: [55]

$$\begin{aligned} a &= \frac{z_{far} - \frac{(z_{far} - b) \cdot z_{near}}{z_{far}}}{z_{far}} \\ &= \frac{z_{far} \cdot (z_{far} - z_{near}) + z_{far} \cdot z_{near}}{z_{far} \cdot (z_{far} - z_{near})} = \frac{z_{far}^2 - z_{far} \cdot z_{near} + z_{far} \cdot z_{near}}{z_{far} \cdot (z_{far} - z_{near})} \\ &= \frac{z_{far}^2}{z_{far} \cdot (z_{far} - z_{near})} = \frac{z_{far}}{z_{far} - z_{near}} \end{aligned} \tag{28}$$

Damit ergibt sich für die Matrix:[56]

$$M_Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{z_{far}}{z_{far} - z_{near}} & -\frac{z_{far} \cdot z_{near}}{z_{far} - z_{near}} \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{29}$$

Dies ist aber noch nicht die endgültige Matrix. Zu Anfang war als Voraussetzung eine quadratische Projektionsfläche gegeben. Da der Bildschirm aber nicht quadratisch ist würde diese Annahme zu Verzerrungen führen. Daher werden nun alle Punkte noch mit einem Faktor in x- und y-Richtung skaliert.
Sowohl für die Höhe, als auch für die Breite des View-Frustums kann man sich Funktionen vorstellen, die diese Werte in Abhängigkeit vom z-Wert der gesuchten Stelle liefern. Man kann also davon ausgehen, dass eine Fläche, die durch dieses View-Frustum begrenzt wird und auf der z-Achse senkrecht steht, die Fläche $H(z) \cdot W(z)$ besitzt. Da aber, wie weiter oben schon erwähnt nur Werte zwischen -1 und $+1$ auf den Bildschirm kommen, müssen alle x-Werte mit dem Faktor $\frac{2}{W}$ und alle y-Werte mit dem Faktor $\frac{2}{H}$ multipliziert werden.[57]
Was nun noch fehlt ist eine Darstellung von W und H, die unabhängig ist von z. H kann man dadurch erhalten, dass man einen Sichtwinkel einführt. Dieser bestimmt den Winkel des Sichtfeldes, durch das der Betrachter die Ebene sieht. Dieser Sichtwinkel gilt aber nur für die YZ-Ebene. Man könnte nun einen weiteren Winkel für die XZ-Ebene einführen, aber dies ist nicht nötig. Die Funktionen $H(z)$ und $W(z)$ sind zwar abhängig von z, aber ihr Verhältnis zueinander bleibt immer gleich, weswegen gilt:[58]

$$r = \frac{W(z)}{H(z)} \tag{30}$$ [59]

[55] Vgl. [BW99, Perspective projection] und [Mic03, D3DXMatrixPerspectiveFovLH].
[56] Vgl. [BW99, Perspective projection] und [Mic03, D3DXMatrixPerspectiveFovLH].
[57] Vgl. [BW99, Perspective projection].
[58] Vgl. [BW99, Perspective projection].
[59] Vgl. [BW99, Perspective projection].

Neben den oben erwähnten Parametern z_{near} und z_{far} werden nun zusätzlich ein Winkel θ, der das Sichtfeld in Y-Richtung beschränkt und der Faktor r, der das Verhältnix von Breite zu Höhe des Bildschirms wiedergibt, benötigt. In Abbildung 7 erkennt man, dass der Winkel θ über den Tangens dargestellt werden kann. Es gilt:

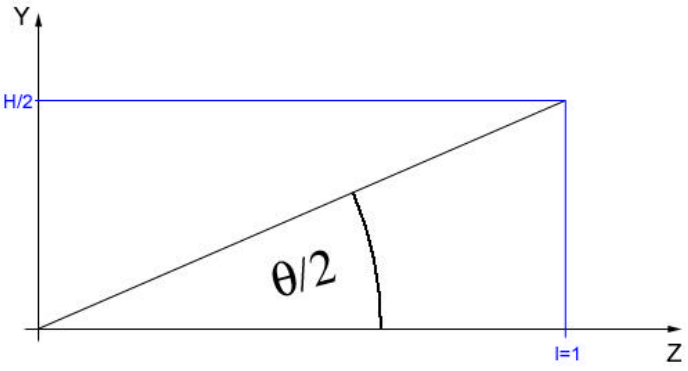


Abbildung 7: Obere Hälfte des Sichtfeldes in z-Richtung

$$\tan\frac{\theta}{2} = \frac{\frac{H}{2}}{l} = \frac{H}{2l}$$
$$H = 2 \cdot l \cdot \tan\frac{\theta}{2}$$

Da, wie oben festgelegt, alle Punkte auf die $z = 1$ Ebene projiziert werden sollen, gilt fr̈ diese Stelle $l = 1$. Damit ergibt sich für H:

$$H = 2 \cdot \tan\frac{\theta}{2} \text{ [60]} \tag{31}$$

Über diese Formel lässt sich nun W berechnen:

$$W = 2 \cdot r \cdot \tan\frac{\theta}{2} \text{ [61]} \tag{32}$$

Nun kann man die endgültigen Faktoren berechnen durch die die Punkte noch skaliert werden müssen und sie gleich in eine Skalierungsmatrix einsetzen:

$$M_S = \begin{bmatrix} r \cdot \tan\frac{\theta}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\frac{\theta}{2}} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ [62]} \tag{33}$$

[60] Vgl. [BW99, Perspective projection].
[61] Vgl. [BW99, Perspective projection].
[62] Vgl. [BW99, Perspective projection].

Die endgültige Projektionsmatrix ergibt sich durch Multiplikation der beiden Teilmatrizen:[63]

$$M_{Proj} = M_Q \cdot M_S \tag{34}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{z_{far}}{z_{far} - z_{near}} & -\frac{z_{far} \cdot z_{near}}{z_{far} - z_{near}} \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} r \cdot \tan\frac{\theta}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\frac{\theta}{2}} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{35}$$

$$= \begin{bmatrix} \frac{1}{r \cdot \tan\frac{\theta}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\frac{\theta}{2}} & 0 & 0 \\ 0 & 0 & \frac{z_{far}}{z_{far} - z_{near}} & -\frac{z_{far} \cdot z_{near}}{z_{far} - z_{near}} \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{36}$$

Mit dieser Matrix lassen sich nun Punkte die sich im Kamerakoordinatensystem befinden auf den Bildschirm projizieren.

[63] Vgl. [BW99, Perspective projection] und [Mic03, D3DXMatrixPerspectiveFovLH].

6 Lichtberechnungen

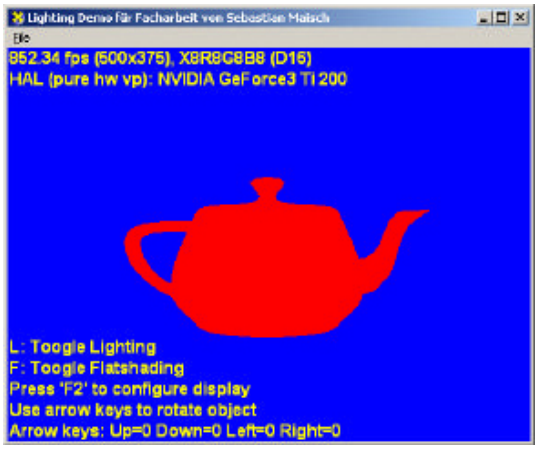


Abbildung 8: Teekanne ohne Beleuchtung

Alles bist jetzt beschriebene hatte mit den Positionen der Vertices als Punkte auf dem Bildschirm zu tun. Was noch nicht angesprochen wurde, ist die Farbe, die diese Punkte haben sollen. Es wurde zwar schon erwähnt, das man jedem Vertex entweder eine Farbe oder eine Textur zuweisen kann, über die dann die Farbe bestimmt wird, aber dabei wird ein wichtiger Aspekt außer Acht gelassen. Was dies ist, erkennt man auf Abbildung 8 recht gut. Man sieht hier eine Teekanne, der einfach die Farbe Rot zugewiesen wurde. Allerdings sieht diese nicht aus wie ein dreidimensionales Objekt, sondern eher wie eine ausgemalte Fläche. Das liegt daran, dass die Lichtberechnung ausgeschaltet wurde. Erst durch den Einfluss von simulierten Licht und Schatteneffekten erkennt das menschliche Auge ein dreidimensionales Bild.

6.1 Lichtarten und Beleuchtungsmodelle

Für die eigentliche Beleuchtung gibt es mehrere Arten von Lichtern, die einzeln oder kombiniert benutzt werden können. Bei jeder Art wird die über eine Textur oder direkt angegebene Ausgangsfarbe mit einem Faktor zwischen 0 und 1 multipliziert. Dadurch ergibt sich bei 1 die Ausgangsfarbe und bei 0 schwarz. Alle anderen Werte dazwischen ergeben einen Farbton der dunkler ist als die Ausgangsfarbe aber noch nicht schwarz. Jede Lichtart berechnet diesen Faktor auf eine andere Art, wodurch sich die Ergebnisse zum Teil stark unterscheiden. Wenn man nun, wie oben angedeutet, mehrere Arten kombinieren will, werden die Faktoren einfach addiert, wobei Werte, die größer sind als eins auf eins abgerundet werden. Neben den verschiedenen Lichtarten gibt es auch noch drei Beleuchtungsmodelle, die bei verschiedenen Voraussetzungen benutzt werden können.[64]

6.1.1 Ambiente Beleuchtung

Das einfachste Modell ist das so genannte ambiente Beleuchtungsmodell. Mit ihm werden Lichtstrahlen simuliert, die keinen fest definierten Ursprung haben. In der Realität wäre das mit Licht vergleichbar, das oft reflektiert wurde. Dieses Licht sorgt dafür, dass auch im Schatten noch eine gewisse Restlichtmenge vorhanden ist. Simuliert wird dieses Licht durch einen konstanten Faktor, der mit dem Farbwert multipliziert wird. Dieses Modell trägt aber nicht dazu bei einen Tiefeneffekt zu sehen, da das Licht von allen Seiten gleichzeitig kommt, und daher jeder Bildpunkt auf dem Bildschirm gleich stark beleuchtet wird. Auf dieses Modell hat die Art des verwendeten Lichts keinen Einfluss.[65]

6.1.2 Diffuse und spekulare Beleuchtung

Das zweite Beleuchtungsmodell ist das diffuse Beleuchtungsmodell. Über dieses Modell werden Objekte beleuchtet, die eine matte Oberfläche haben und daher das Licht streuen. Im Gegensatz hierzu existiert noch ein drittes, das so genannte spekulare Modell. Dieses simuliert glänzende Oberflächen, die das Licht spiegeln, und dadurch den Betrachter blenden

[64] Vgl. [Zer00, S. 408] und [eng03].
[65] Vgl. [Zer00, S. 406]; [eng03, Folien 7-9]; [koe03, Folie 8].

können. Beide Modelle sind abhängig von den verschiedenen Lichtarten mit denen sie beleuchtet werden. Diese Lichtarten machen im Großen und Ganzen hauptsächlich Aussagen über die Position der Lichtquelle. Im weiteren Verlauf soll das spekulare Licht außer Acht gelassen werden. Die Grundideen sind hierbei ähnlich, es müssten aber ein paar Anpassungen gemacht werden.[66]

6.1.3 Direktionale Lichter

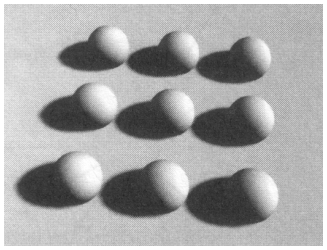


Abbildung 9: Beleuchtung mit direktionalem Licht

Nach dem ambienten Licht, das nur durch eine einzige Zahl repräsentiert werden kann, ist das direktionale Licht ebenfalls sehr einfach zu berechnen. Bei diesem Licht kann man sich vorstellen, dass die Lichtquelle unendlich weit entfernt liegt. Daher kann man davon ausgehen, dass alle Lichtstrahlen parallel sind. Dieses Licht kann also durch seinen Richtungsvektor dargestellt werden. Bei sämtlichen Lichtarten ist zur Bestimmung der Intensität des Lichts der Einstrahlwinkel auf eine Oberfläche interessant. Wenn das Licht senkrecht auf eine Oberfläche fällt, ist die Intensität am höchsten, wenn es dagegen parallel zur Oberfläche verläuft, macht es sich überhaupt nicht bemerkbar. Wie weiter oben habe erwähnt wurde, besitzt zur Lichtberechnung jeder Vertex einen Normalvektor. Zur Vereinfachung soll hier davon ausgegangen werden, dass nur jedes Polygon einen solchen Vektor besitzt. Dieser Vektor steht senkrecht auf dem Polygon und bildet ebenfalls einen Winkel mit dem Richtungsvektor des einfallenden Lichts. Ist dieser gleich null, so steht der Vektor senkrecht, bei 90° ist er parallel. Über das Skalarprodukt zwischen den beiden Vektoren lässt sich dieser Winkel berechnen:

$$\vec{L} \circ \vec{N} = \|\vec{L}\| \cdot \|\vec{N}\| \cdot \cos\alpha \text{ [67]} \tag{37}$$

Wenn man für beide Vektoren voraussetzt, dass diese Einheitsvektoren sind, ist das Skalarprodukt der Kosinus des Zwischenwinkels. Dieser erfüllt auch alle Anforderungen, welche der Faktor, der die Intensität des Lichtes angibt, erfüllen muss. Ist der Winkel zwischen den Vektoren Null, so ist der Kosinus gleich Eins, bei 90° ist er Null. Daher kann man als Intensitätsfaktor eines direktionalen Lichtes das Skalarprodukt zwischen Normalvektor der beleuchteten Oberfläche und dem Richtungsvektor des Lichtes benutzen.[68]

6.1.4 Punktlichter

Eine weitere Möglichkeit, ein Licht zu beschreiben, ist das Punktlicht. In diesem Fall hat das Licht einen festgelegten Ort anstatt einer Richtung. Der Richtungsvektor lässt sich aber sehr einfach bestimmen. Dazu wird der normalisierte Differenzvektor zwischen der Lichtquelle und der Position, für die die Lichtberechnung durchgeführt wird, herangezogen. Die angepasste Gleichung ergibt nun:

$$\frac{\vec{P}_{Licht} - \vec{P}_{Objekt}}{\|\vec{P}_{Licht} - \vec{P}_{Objekt}\|} \circ \vec{N} = \cos\alpha \text{ [69]} \tag{38}$$

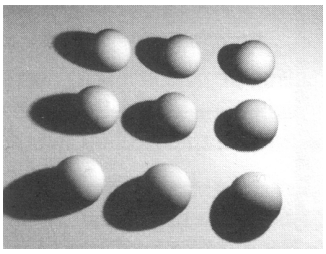


Abbildung 10: Beleuchtung mittels eines Punktlichts

[66] Vgl. [Zer00, S. 410]; [eng03, Folien 10ff]; [koe03, Folien 9f].
[67] Vgl. [Zer00, S. 408f]; [eng03, Folien 11f]; [koe03, Folie 10].
[68] Vgl. [Zer00, S. 408f]; [eng03, Folien 11f]; [koe03, Folie 10]; [Rou01, S. 407].
[69] Vgl. [Zer00, S. 407]; [Rou01, S. 406].

6.1.5 Strahler

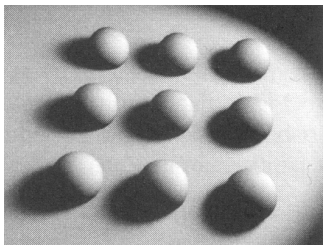


Abbildung 11: Beleuchtung über einen Strahler

Eine sehr komplexe Art des Lichts ist der Strahler. Hier geht von einem Punkt im Gegensatz zum Punktlicht das Licht nicht in alle Richtungen, sondern wird durch eine Austrittsrichtung und einen Winkel, der die Abweichung von dieser Richtung angibt, abgegrenzt. Der Strahler ist hier nur der Vollständigkeit halber erwähnt, wird aber auf Grund seiner Komplexität nicht weiter beschrieben.[70]

6.2 Beleuchtungsarten

Wenn man nun dieselbe Teekanne nochmals betrachtet, aber die Beleuchtung hinzunimmt, kommt man zu einem Ergebnis, dass sich zwar schon ein dreidimensionales Objekt erkennen lässt, es aber immer noch nicht realistisch aussieht. In Abbildung 12 kann man das gut erkennen. Das liegt daran, dass die Normalvektoren, die verwendet wurden, für eine ganze Fläche gelten. Mit dieser Vereinfachung lässt sich zwar schnell rechnen, sie ist aber optisch unschön. Es gibt aber noch andere Möglichkeiten der Beleuchtung, die realistischer aussehen.

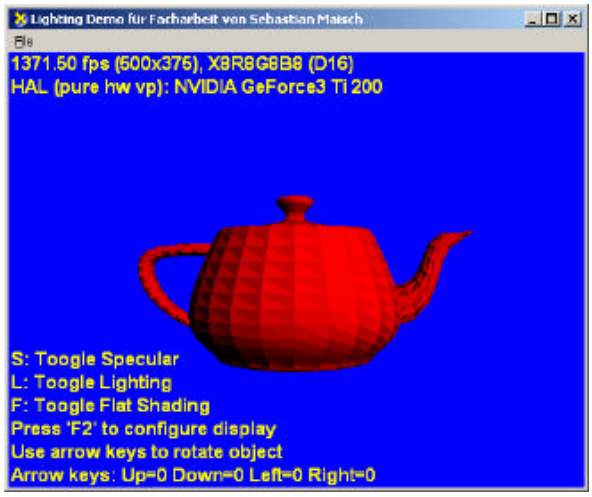


Abbildung 12: Lichtberechnung über die Normalvektoren einer Fläche

6.2.1 Flat-Shading

Die einfachste Möglichkeit ist in Abbildung 12 zu sehen. Sie wird „Flat-Shading“ genannt. Beim Flat-Shading hat jede Fläche einen Normalvektor, und bekommt daher an jeder Stelle exakt die gleiche Farbe zugewiesen. Da sich an den Kanten der Normalvektor ändert, entstehen oft harte Farbübergänge.[71]

[70] Vgl. [Zer00, S. 408]; [Rou01, S. 407].
[71] Vgl. [Rou01, S. 405] und [Zer00, S. 405, 410ff].

6.2.2 Gouraud-Shading

Die im Moment am weitesten verbreitete Technik nennt sich „Gouraud-Shading“. Hier besitzt wie anfangs erwähnt jeder Vertex einen Normalvektor, der als Durchschnitt zwischen den Normalvektoren aller anliegenden Flächen berechnet wird. Bei der Lichtberechnung wird nun für jeden Vertex ein Intensitätsfaktor errechnet, der dann für jeden Bildpunkt einer Fläche interpoliert wird. Die Abbildung zeigt die schon bekannte Teekanne, diesmal mit Gouraud-Shading. Dies liefert fast immer sehr überzeugend aussehende Ergebnisse.[72]

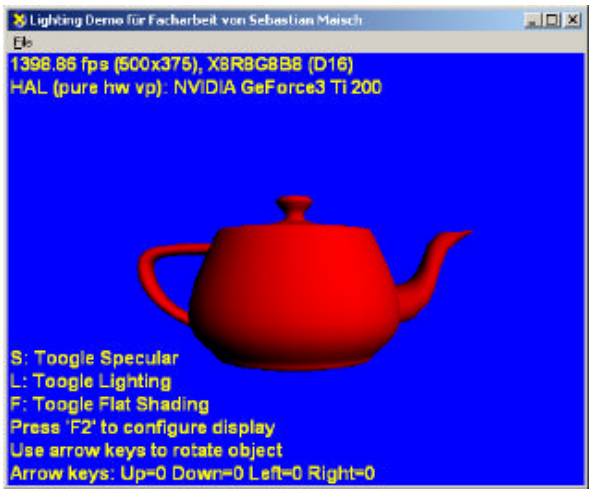


Abbildung 13: Lichtberechnung über interpolierte Vertexnormalvektoren

6.2.3 Per-Pixel Lighting

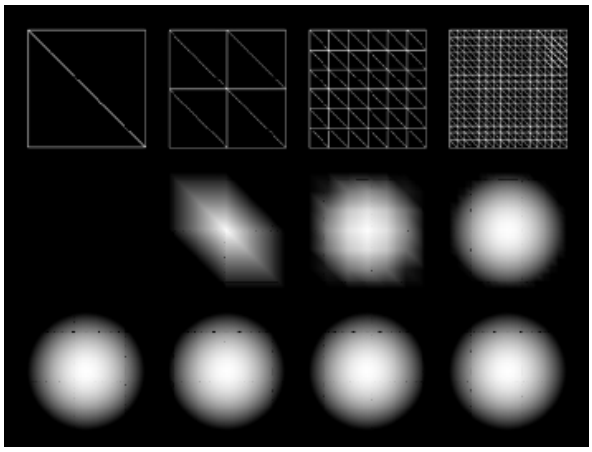


Abbildung 14: Per-Pixel Lighting im Vergleich zu Gouraud-Shading

Eine weitere Technik, die sich in Zukunft wahrscheinlich gegen Gouraud-Shading durchsetzen wird, ist das Per-Pixel Lighting. Bei diesem wird ein wenig anders vorgegangen als beim Gouraud-Shading. Hier werden die Normalvektoren über die gesamte Fläche interpoliert, so dass jeder Bildpunkt einen eigenen Normalvektor besitzt, über den die Farbe berechnet wird. Dies ist etwas rechenaufwendiger als die beiden anderen Techniken. In manchen Fällen ist dies aber nötig. Man stelle sich zum Beispiel eine sehr große Fläche vor, in deren Mitte ein Lichtkegel fällt. Da nun die Eckpunkte der Fläche, also die Vertices, kein Licht abbekommen, wird die Fläche nicht beleuchtet. Die Abbildung macht die Abhängigkeit vor Gouraud-Shading (mittlere Reihe) zur Menge an Vertices (obere Reihe) deutlich. In der unteren Reihe sieht man zum Vergleich Per-Pixel Lighting bei den gleichen Voraussetzungen. Der größte Nachteil von dieser Technik ist allerdings, dass sie nur auf sehr neuer Hardware möglich ist, weswegen es noch einige Zeit dauern wird, bis sie weitreichend Verwendung findet.[73]

[72] Vgl. [Rou01, S. 406] und [Zer00, S. 410ff].
[73] Vgl. [koe03, Folien 4f].

7 Zukunftsaussichten

Was die Zukunft für die Echtzeit 3D-Grafik bringt, lässt sich nur erahnen. Wahrscheinlich wird ihre Bedeutung aber weiter zunehmen. Das hängt von einigen Faktoren ab. Zum einen sind moderne Grafikkarten sehr stark auf die Darstellung von 3D-Grafik ausgelegt, so dass man durch ihre Benutzung oftmals einen Geschwindigkeitszuwachs gegenüber anderer Programme feststellen kann. Dies führt zum Beispiel dazu, dass in Windows Longhorn, dem nächsten Betriebssystem von Microsoft, die Grafikanzeige auf 3D umgestellt wird. In eine andere Richtung geht die Entwicklung der schon angesprochenen Shader. Diese werden in naher Zukunft stark erweitert, so dass der Programmierer viel mehr Einfluss auf Berechnungen nehmen kann, die auf der Grafikkarte ablaufen. Die offensichtlichste Folge daraus sind vor allem Special-Effects, die hauptsächlich in Computerspielen Anwendung finden werden. Aber auch die einfachen Shader von heute können in Bereichen eingesetzt werden, in denen es niemand für möglich gehalten hätte. Sie können bei komplizierten und zeitaufwendigen Berechnungen, bei denen keine Grafikausgabe vonnöten ist, verwendet werden, um den Prozessor zu entlasten, indem ein grossteil der Berechnungen auf einer Grafikkarte abläuft. Interessanterweise haben diese Berechnungen im seltensten Fall etwas mit Grafik zu tun oder finden im dreidimensionalen Raum statt, aber dadurch, dass der Programmierer selbst bestimmen kann, was genau auf der Grafikkarte abläuft, sind ihm sehr große Freiheiten gegeben und die Möglichkeiten sind noch lange nicht ausgeschöpft. Trotz all der anstehenden Veränderungen bleiben allerdings die mathematischen Grundlagen dieselben. Daher ist es auch in Zukunft von Vorteil sich mit den beteiligten Mechanismen auszukennen.

A Erklärung der Quaternion Roatation im euklidischen Raum

[74] von Jason Shankel

Einleitung

Der Zweck dieser Abhandlung ist, zu demonstrieren wie Quaternion Operationen verwendet werden, um Rotationen im euklidischen Raum durchzuführen. Die Rotation wird anhand von nichts als einfachen Vektoroperationen demonstriert (Punktprodukt, Kreuzprodukt und Skalarmultiplikation). An keinem Punkt benutze ich imaginäre Zahlen, Berechnungen oder andere so genannte „nicht-euklidische“ Konzepte.

Definitionen

Quaternions werden folgendermaßen spezifiziert:

$$q = (\vec{v}, s)$$

wobei $\vec{v}$ ein Vektor im gewöhnlichen dreidimensionalen Raum und s ein Skalar ist. Für einen beliebigen Vektor $\vec{V}$, wird das gleichwertige Quaternion wie folgt spezifiziert:

$$V = (\vec{V}, 0)$$

Quaternion Multiplikation wird folgendermaßen durchgeführt:

$$(\vec{v}_1, s_1) \cdot (\vec{v}_2, s_2) = (\vec{v}_1 \times \vec{v}_2 + s_1\vec{v}_2 + \vec{v}_1 s_2, s_1 s_2 - \vec{v}_1 \circ \vec{v}_2)$$

Die Multiplikation ist assoziativ, aber nicht kommutativ.

Multiplikation mit einem Skalar wird folgendermaßen durchgeführt:

$$x \cdot (\vec{v}, s) = (x\vec{v}, xs)$$

Die Multiplikation mit einem Skalar ist assoziativ, und kommutativ.

Das konjugierte eines Quaternion wird folgendermaßen definiert:

$$(\vec{v}, s)^* = (-\vec{v}, s)$$

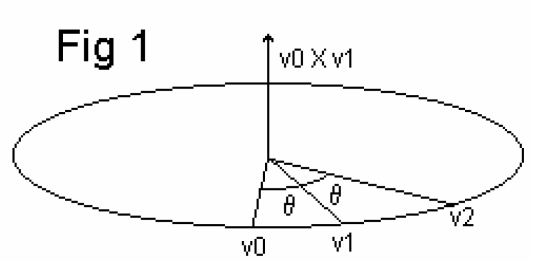
Diese sind die einzigen Definitionen, die wir benötigen um die Rotationsformel zu zeigen.

Quaternion Rotation

$\vec{v}_0$, $\vec{v}_1$ und $\vec{v}_2$ sind Einheitsvektoren im dreidimensionalen Raum. Sie sind komplanar, so dass gilt: $\vec{v}_0 \times \vec{v}_1 = \vec{v}_1 \times \vec{v}_2$ und $\vec{v}_0 \circ \vec{v}_1 = \vec{v}_1 \circ \vec{v}_2$.
$\theta = \cos \vec{v}_0 \times {\vec{v}_1}^{-1}$ soll der Winkel zwischen $\vec{v}_0$ und $\vec{v}_1$ sein. Es ist aus der Abbildung ersichtlich, dass $\vec{v}_2$ eine Rotation von $\vec{v}_0$ um $\vec{v}_0 \times \vec{v}_1$ mit dem Winkel 2θ darstellt.

Behauptung Mit dem gegebenen Quaternion $q = (\vec{v}_0 \times \vec{v}_1, \vec{v}_0 \circ \vec{v}_1)$ ist zu zeigen, dass $q \cdot \vec{v}_0 \cdot q^* = \vec{v}_2$.

[74] Dieser Anhang ist eine Übersetzung von [Sha].



Beweis Zuerst suchen wir einige gleichwertige Formen für q:

$$\begin{aligned}
&q = (\vec{v}_0 \times \vec{v}_1, \vec{v}_0 \circ \vec{v}_1) \\
&v_1 \cdot v_0^* = (-\vec{v}_1 \times \vec{v}_0, \vec{v}_1 \circ \vec{v}_0) = (\vec{v}_0 \times \vec{v}_1, \vec{v}_0 \circ \vec{v}_1) \\
&\Rightarrow q = v_1 \cdot v_0^*
\end{aligned}$$

$$\begin{aligned}
&\vec{v}_1 \times \vec{v}_2 = \vec{0}_0 \times \vec{v}_1 \\
&\vec{v}_1 \circ \vec{v}_2 = \vec{v}_0 \circ \vec{v}_1 \\
&\Rightarrow q = (\vec{v}_1 \times \vec{v}_2, \vec{v}_1 \circ \vec{v}_2) = v_2 \cdot v_1^*
\end{aligned}$$

Wenn man jetzt $q = v_1 \cdot v_0^*$ in $q\vec{v}_0 p^*$ einsetzt erhält man:

$$q\vec{v}_0 p^* = (v_1 \cdot v_0^*)\vec{v}_0 q^* = v_1 \cdot (v_0^* \cdot v_0) \cdot q^*$$

Für Einheitsvektoren gilt $v^* \cdot v = 1$, so dass:

$$q\vec{v}_0 p^* = v_1 \cdot q^*$$

Jetzt drücken wir q^* durch v_1 und v_2 aus:

$$q^* = (v_2 \cdot v_1^*)^* = v_1 \cdot v_2^*$$

Diese Form von q^* in $v_1 \cdot q^*$ eingesetzt ergibt:

$$q\vec{v}_0 p^* = v_1 \cdot q^* = v_1^2 \cdot v_2^*$$

Für Einheitsvektoren gilt $v^2 = -1$, so dass:

$$q\vec{v}_0 p^* = -v_2^*$$

Für Vektoren gilt $v^* = -\vec{v}$, so dass

$$q\vec{v}_0 p^* = \vec{v}_2$$

QED

Wir haben nun bewiesen, dass für Einheitsvektoren $q\vec{v}_0 p^* = \vec{v}_2$ gilt. Dieses für alle Vektoren abzuleiten ist trivial. Einfach beide Seiten der Gleichung mit einem Skalar multiplizieren.

$$\begin{aligned}
&q\vec{v}_0 p^* = \vec{v}_2 \\
&\Rightarrow q \cdot A \cdot \vec{v}_0 \cdot q^* = A \cdot \vec{v}_2
\end{aligned}$$

für jeden beliebigen Skalar A.

Abbildungsverzeichnis

Die Grafiken wurden entweder selbst erstellt, zum Teil in Anlehnung an Grafiken in [Kwo98] und [BW99, Perspective projection], oder entnommen aus [Mic], [Uni99], [Rou01, S. 407f] und [koe03, Folie 5].

Literatur

[BW99] Philippe Bekaert and Yves D. Willems. Viewing in 3d, 1999. http://www.cs.kuleuven.ac.be/cwis/research/graphics/INFOTEC/viewing-in-3d/viewing-engels.html.

[Ebe01] David H. Eberly. *3D game engine design*. Morgan Kaufmann, 2001.

[eng03] *Einführung in die Shader Programmierung mit HLSL*. ZFX, 2003. http://old.zfxcon.info/zfxCON03/Proceedings/zfxCON03_WEngel.pdf, Vortrag auf der zfxCON2003 von W. Engel.

[koe03] *ART - Advanced Rendering Techniques*. ZFX, 2003. http://old.zfxcon.info/zfxCON03/Proceedings/zfxCON03_MKoegler.pdf, Vortrag auf der zfxCON2003 von M. Kögler.

[Kwo98] Young-Hoo Kwon. Rotation matrix, 1998. http://kwon3d.com/theory/transform/rot.html.

[Mic] Microsoft. *DirectX 8 SDK*.

[Mic03] Microsoft. *DirectX SDK - D3DX Reference Manual*, 2003. http://msdn.microsoft.com/en-us/library/bb172972(VS.85).aspx.

[Rou01] Christian Rousselle. *Jetzt lerne ich Spieleprogrammierung mit DirectX und Visual C++*. Markt-und-Technik Verl., 2001.

[Sha] J. Shankel. An explanation of quaternion rotation in euclidean space. http://shankel.best.vwh.net/QuatRot.html, Zugriff am 12.1.2004.

[Uni99] Princeton University. Perspective projection view volume, 1999. http://www.cs.princeton.edu/courses/archive/fall99/cs426/lectures/view/sld024.htm.

[ZDA04] Stefan Zerbst, Oliver Düvel, and Eike Anderson. *3D-Spiele-Programmierung Kompendium. Professionelle Entwicklung von 3D-Engines und -Spielen*. Markt und Technik, 2004.

[Zer] Stefan Zerbst. 3d grundlagen tutorial. http://www-public.tu-bs.de:8080/~y0005571/3dtutorial/main.htm,, Zugriff am 9.1.2004.

[Zer00] Stefan Zerbst. *3D-Spieleprogrammierung mit DirectX in C/C++*. S. Zerbst, Libri Books on Demand, 2000.