Hochschule Mittweida (FH)
University of Applied Sciences

MITTWEIDA
ROSSWEIN

Monse, Axel

Konzeption und Implementierung eines Systems zur Darstellung von X3D-Objekten in Direct3D

DIPLOMARBEIT

HOCHSCHULE MITTWEIDA (FH)
UNIVERSITY OF APPLIED SCIENCES

Fachbereich Informations- und Elektrotechnik

Mittweida, 2009

Hochschule Mittweida (FH)
University of Applied Sciences

MITTWEIDA
ROSSWEIN

Monse, Axel

Konzeption und Implementierung eines Systems zur Darstellung von X3D-Objekten in Direct3D

eingereicht als

DIPLOMARBEIT

an der

HOCHSCHULE MITTWEIDA (FH)
UNIVERSITY OF APPLIED SCIENCES

Fachbereich Informations- und Elektrotechnik

Mittweida, 2009

Erstprüfer: Prof. Dr.-Ing. Mario Geißler
Zweitprüfer: Dipl.-Ing. Norbert Göbel

Vorgelegte Arbeit wurde verteidigt am:

Bibliographische Beschreibung:

Monse, Axel: Konzeption und Implementierung eines Systems zur Darstellung von X3D-Objekten in Direct3D. - 2009. - 77 S. Mittweida, Hochschule Mittweida (FH), Fachbereich IT/ET, Diplomarbeit, 2009

Referat:

Diese Diplomarbeit beschäftigt sich mit der Untersuchung der Möglichkeiten zur Visualisierung der Modellierungssprache X3D in einer Echtzeit-3D-Renderumgebung auf Basis von Direct3D. Im Rahmen dieser Arbeit werden zunächst ein grundlegendes Verständnis zu Computergrafik und die Arbeit im virtuellen Raum erarbeitet. Danach erfolgt eine kurze Einführung in X3D, sowie DirectX - im speziellen Direct3D. Im Anschluss wird die Herangehensweise beim Durchlaufen und Interpretieren von Daten in X3D-Dateien betrachtet. Darauf aufbauend erfolgt die Implementierung eines Konverters und ein Konzept einer Renderumgebung wird vorgestellt. Abschließend gibt die Arbeit einen kurzen Ausblick auf die Zukunft von X3D und Erweiterungsmöglichkeiten des Konverters werden präsentiert.

Inhaltsverzeichnis

1 Einleitung

Seit dem erstmaligen Zusammentreffen der SIGGRAPH[1] im Jahr 1974, ist die 3D-Computergrafik auf dem Vormarsch. Zuerst entwickelten Wissenschaftler mathematische Formeln zur Beschreibung von 3D-Räumen und Techniken zur Darstellung von Grafiken. Dabei waren deren visuellen Resultate nur wenigen Menschen zugänglich.

Doch im Laufe der 90er Jahre wurden die Digitalrechner immer leistungsfähiger und somit war es möglich immer realistischere 3D-Szenen darzustellen. In dieser Zeit kam auch die Öffentlichkeit erstmals durch Computerspiele oder den ersten computeranimierten Kinofilmen, wie ToyStory oder Shrek mit der 3D-Computergrafik in Kontakt. Im Laufe der Jahre wurde es immer einfacher, 3D-Objekte zu erstellen. Zunächst war dies nur Profis mit Programmen wie CAD, 3dsMax oder Maya möglich. In den letzten Jahren wird die 3D-Computergrafik immer mehr durch kostenlose und kinderleicht zu bedienende Programme wie SketchUp oder TrueSpace der breiten Öffentlichkeit zugänglich gemacht. So ist es z.B. jeder Person die über einen PC verfügt möglich, ihr Wohnhaus in 3D nachzubauen und in GoogleEarth bzw. VirtualEarth zu importieren und zu betrachten.

Doch schon Mitte der 90er Jahre erregte eine Beschreibungssprache Aufsehen, mit der man 3D-Welten programmieren und im Browser durch diese navigieren konnte - VRML. Auch der Nachfolger - X3D macht es möglich, einfache 3D-Szenen mit Editoren wie X3DEdit zu erstellen. Da es sich bei X3D um eine Beschreibungssprache handelt, stellt sich die Frage, wie man die beschriebenen Szenen grafisch darstellt. Ein breite Nutzergruppe verwendet heutzutage Windows-Betriebssysteme und auf fast jedem PC ist DirectX installiert. Damit würde es sich anbieten, X3D-Daten mit Direct3D darzustellen. In dieser Diplomarbeit möchte ich diesen Gedanken weiter vertiefen und ein Konzept für die Darstellung von X3D-Objekten in einer Direct3D-Umgebung erstellen. Aus diesem Konzept heraus werde ich einen X3D-Viewer implementieren.

Bevor ich zum Hauptteil meiner Arbeit komme, möchte ich auf ein paar Grundlagen eingehen. Ich beginne mit den mathematischen Grundlagen. Anschließend stelle ich die wichtigsten Begriffe und Konzepte der Computergrafik vor. Mit diesem Wissen wird es leichter sein, in den Kernbereich meiner Diplomarbeit vorzudringen.

[1] Special Interest Group on Graphics

2 Mathematische Grundlagen

2.1 Koordinatensysteme

In der 3D-Computergrafik wird das aus drei Achsen (X-/Y-/Z-Achse) bestehende kartesisches Koordinatensystem verwendet. Dabei kann jedem Punkt ein eindeutiges *Zahlentripel* (x, y, z) zugeordnet werden. Durch das Koordinatensystem wird der dreidimensionale Raum in acht Oktanten zerlegt.

Üblicherweise zeigt die positive X-Achse von links nach rechts. Die positive Y-Achse von unten nach oben. Nur für die Richtung der Z-Achse werden in der 3D-Softwarebranche zwei Varianten verwendet:

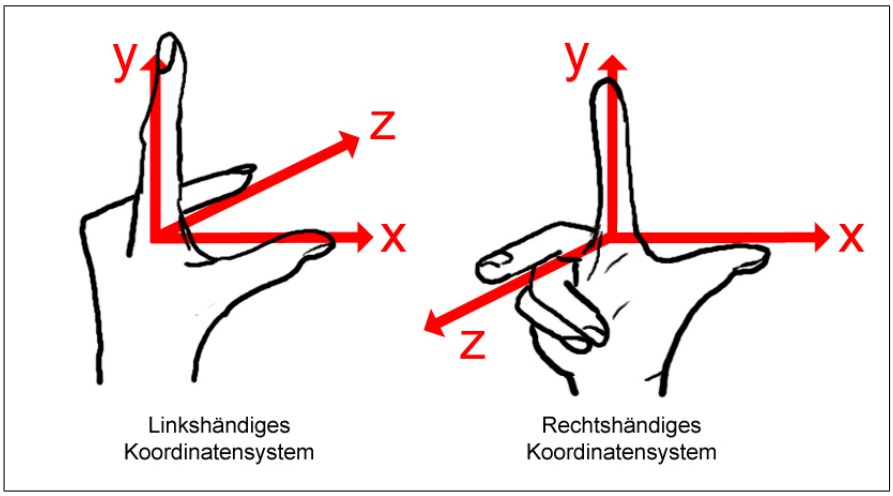


Abbildung 2.1: Links- und Rechtshändiges Koordinatensystem

Bei der als *rechtshändiges Koordinatensystem* bezeichneten Variante zeigt die positive Z-Achse vom Bildschirm aus zum Betrachter. Somit zeigt beim *linkshändigen Koordinatensystem* die positive Z-Achse vom Betrachter aus zum Bildschirm. X3D benutzt das rechtshändiges Koordinatensystem[1], während Microsoft mit Direct3D auf das linkshändige Koordinatensystem zurückgreift[2]

[1] vgl. [2] S.69
[2] vgl. [3] S.449/450 oder [7] S. 117

2.2 Vektoren

2.2.1 Allgemein

Ein Vektor ist eine Größe, die durch Betrag und Richtung festgelegt ist. Sie wird grafisch als gerichtete Strecke einer bestimmten Länge dargestellt. Im dreidimensionalen Raum wird der Vektor durch x-,y-, und z-Koordinate gekennzeichnet:

$$v(x, y, z)$$

Zur Berechnung eines Vektors AB zwischen Anfangspunkt A und Endpunkt B gilt folgende Regel:

$$v_{AB} = (x_B - x_A, y_B - y_A, z_B - z_A) = (x, y, z)$$

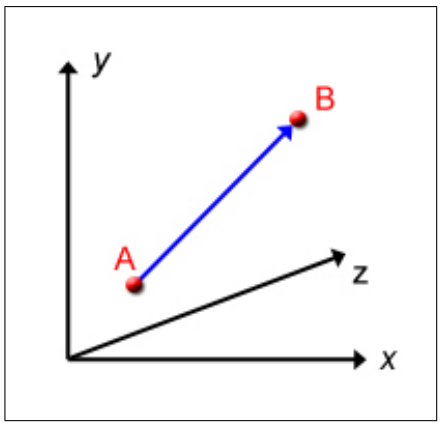


Abbildung 2.2: Ein Vektor (blauer Pfeil) zwischen zwei Punkten A und B

Auf Vektoren trifft man in der 3D-Computergrafik z.B. bei Beleuchtung der Szene, Kollisionserkennung oder bei der Berechnung von Normalen.

2.2.2 Betrag eines Vektors

Der Betrag oder die Länge $|v|$ eines Vektors $v(x, y, z)$ wird berechnet durch:

$$|v| = \sqrt{x^2 + y^2 + z^2}$$

Beträgt die Länge des Vektors Eins handelt es sich um einen Einheitsvektor. Jeder Vektor kann zu einem Einheitsvektor transformiert werden. Dies erreicht man,

indem der Vektor durch seinen Betrag dividiert wird:

$$\vec{n}= \frac{v(x,y,z)}{|v|}$$

Man spricht davon, dass somit der Vektor *normiert* ist.

2.2.3 Addition/Subtraktion von Vektoren

Die Addition oder Subtraktion von zwei Vektoren v_1 und v_2 erfolgt durch addieren oder subtrahieren der einzelnen Richtungskomponenten.

$$v_3 = v_1 + v_2 = \begin{pmatrix} x_1 + x_2 \\ y_1 + y_2 \\ z_1 + z_2 \end{pmatrix}$$

$$v_4 = v_1 - v_2 = \begin{pmatrix} x_1 - x_2 \\ y_1 - y_2 \\ z_1 - z_2 \end{pmatrix}$$

2.2.4 Multiplikation/Division von Vektoren

Die Multiplikation oder Division von zwei Vektoren v_1 und v_2 erfolgt durch multiplizieren oder dividieren der einzelnen Richtungskomponenten.

$$v_3 = v_1 * v_2 = \begin{pmatrix} x_1 * x_2 \\ y_1 * y_2 \\ z_1 * z_2 \end{pmatrix}$$

$$v_4 = v_1 \div v_2 = \begin{pmatrix} x_1 \div x_2 \\ y_1 \div y_2 \\ z_1 \div z_2 \end{pmatrix}$$

2.2.5 Skalarprodukt eines Vektors

Das Skalarprodukt v_3 von zwei Vektoren v_1 und v_2 ergibt sich aus dem Produkt der Längen beider Vektoren multipliziert mit dem Kosinus des Winkels φ zwischen beiden Vektoren. Dabei ist zu beachten, dass es sich bei φ nicht um den überstumpfen Winkel (Winkel > 180 Grad) zwischen den Vektoren handelt.

$$v_3 = v_1 \bullet v_2 = |v_1| * |v_2| * \cos\varphi$$

Ebenfalls kann das Skalarprodukt über die Richtungskomponenten berechnet werden:

$$v_3 = v_1 \bullet v_2 = (x_1 * x_2 + y_1 * y_2 + z_1 * z_2)$$

Durch gleichsetzen dieser beiden Gleichungen ergibt sich die Berechnung des Winkels φ zwischen zwei Vektoren:

$$\cos\varphi = \frac{x_1 * x_2 + y_1 * y_2 + z_1 * z_2}{|v_1| * |v_2|}$$

$$\varphi = \arccos\left(\frac{x_1 * x_2 + y_1 * y_2 + z_1 * z_2}{|v_1| * |v_2|}\right)$$

2.2.6 Kreuzprodukt eines Vektors

Aus dem Kreuzprodukt von zwei Vektoren v_1 und v_2 ergibt sich ein Vektor v_3, der senkrecht auf den beiden Vektoren steht. Die Länge des Kreuzproduktes ist gleichzeitig die Fläche des von v_1 und v_2 aufgespannten Parallelogramms.

Wie beim Skalarprodukt gibt es beim Kreuzprodukt ebenfalls zwei Berechnungsvarianten. Zuerst die Berechnung durch die Multiplikation der Beträge der Vektoren, multipliziert mit dem $\sin$ des Winkels φ zwischen den Vektoren:

$$v_3 = v_1 \times v_2 = |v_1| * |v_2| * \sin\varphi$$

Die andere Variante

$$v_3 = v_1 \times v_2 = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \times \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} y_1 * z_2 - z_1 * y_2 \\ z_1 * x_2 - x_1 * z_2 \\ x_1 * y_2 - y_1 * x_2 \end{pmatrix}$$

ist der Weg über die Richtungskomponenten.[3]

2.3 Matrizen

2.3.1 Allgemein

Als Matrix A (Mehrzahl: Matrizen) bezeichnet man ein System mit m mal n Elementen, aus z.B. reellen oder komplexen Zahlen, die in m Zeilen und n Spalten

[3] vgl. [9] S. 186-193

angeordnet sind:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \dots & \dots & \dots & \dots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{pmatrix}$$

Die Elemente der Matrix definiert man mit $a_{y,x}$. Dabei bezeichnet x die Spalte und y die Zeile in dem sich das Element befindet. Handelt es sich um eine quadratische Matrix ($m = n$) formen die Elemente bei denen x und y identisch sind, die Diagonale der Matrix. Als Identitätsmatrix wird die Matrix bezeichnet, wenn die Elemente auf der Diagonalen „1“ und die restlichen Elemente „0“ betragen:

$$E = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

In der 3D-Computergrafik werden hauptsächlich 3x3 bzw. 4x4 Matrizen, z.B. für Transformationen (Rotation, Skalierung und Translation), verwendet. Deshalb wird im weiteren Verlauf speziell nur auf diese Matrizengrößen eingegangen.

2.3.2 Addition/Subtraktion von Matrizen

Die Addition und Subtraktion von zwei Matrizen A und B gleichen Typs erfolgt, indem elementweise die gleichgestellten Elemente $a_{y,x}$ und $b_{y,x}$ addiert oder subtrahiert werden.

$$C = A + B = \begin{pmatrix} a_{1,1} + b_{1,1} & \dots & a_{1,n} + b_{1,n} \\ a_{2,1} + b_{2,1} & \dots & a_{2,n} + b_{2,n} \\ \dots & \dots & \dots \\ a_{m,1} + b_{m,1} & \dots & a_{m,n} + a_{m,n} \end{pmatrix}$$

$$C = A - B = \begin{pmatrix} a_{1,1} - b_{1,1} & \dots & a_{1,n} - b_{1,n} \\ a_{2,1} - b_{2,1} & \dots & a_{2,n} - b_{2,n} \\ \dots & \dots & \dots \\ a_{m,1} - b_{m,1} & \dots & a_{m,n} - a_{m,n} \end{pmatrix}$$

2.3.3 Multiplikation von Matrizen

Die Multiplikation zweier Matrizen ist nur möglich, wenn die Spaltenanzahl der Matrix A gleich der Zeilenanzahl der Matrix B beträgt, d.h.: Wenn A eine Matrix

vom Typ (m, n) ist, dann muss die Matrix B von Typ (n, p) sein, und das Produkt der Multiplikation ergibt eine Matrix C vom Typ (m, p).

$$C = AB = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \times \begin{pmatrix} j & k & l \\ m & n & o \\ p & q & r \end{pmatrix}$$

$$C = \begin{pmatrix} a*j+b*m+c*p & a*k+b*n+c*q & a*l+b*o+c*r \\ d*j+e*m+f*p & d*k+e*n+f*q & d*l+e*o+f*r \\ g*j+h*m+i*p & g*k+h*n+i*q & g*l+h*o+i*r \end{pmatrix}$$

2.3.4 Multiplikation von Matrizen mit Vektoren

Um einen Vektor mit einer Matrix zu multiplizieren, muss man den Vektor als eine Matrix A betrachten:

$$v(x, y, z) = A = \begin{pmatrix} x & y & z \end{pmatrix}$$

Dadurch kann die Multiplikation des Vektors mit der Matrix, wie bei zwei gewöhnlichen Matrizen erfolgen:

$$C = AB = \begin{pmatrix} x & y & z \end{pmatrix} \times \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

$$C = \begin{pmatrix} x*a+y*d+z*g & x*b+y*e+z*h & x*c+y*f+z*i \end{pmatrix}$$

Im weiteren Verlauf meiner Arbeit trifft man auch auf Multiplikationen von Vektoren mit 4x4 Matrizen. Dabei ist es nötig, den Vektor um ein Element W, das i.R. den Wert 1 besitzt, zu erweitern.

$$v = A = \begin{pmatrix} x & y & z & w \end{pmatrix}$$

Durch diese Erweiterung ist es möglich, die Multiplikation durchzuführen:

$$C = AB = \begin{pmatrix} x & y & z & 1 \end{pmatrix} \times \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix}$$

$$C = \begin{pmatrix} xa+ye+zi+m & xb+yf+zj+n & xc+yg+zk+o & xd+yh+zl+p \end{pmatrix}$$

Falls das zusätzliche Element W nicht den Wert 1 besitzt, muss nach der Multiplikation jedes Element durch den Wert von W dividiert werden, um wieder die Form $v(x, y, z)$ zu erhalten.[4]

[4] vgl. [9] S. 261-264

3 3D-Konzepte

3.1 Objektbeschreibungen

Die 3D-Objekte in der 3D-Computergrafik werden in der Regel als *Mesh* (zu dt. 3D-Modell) bezeichnet. Ein Mesh besteht aus *Vertices* (dt. Eckpunkte) die durch *Edges* (dt. Kanten) miteinander verbunden sind. Das Gebilde, bei der Edges eine in sich geschlossene Fläche bilden, nennt man *Polygon* (dt. Vieleck) oder auch *Face* (dt. Fläche). Eine Edge kann von maximal zwei aneinanderliegenden Polygonen genutzt werden. Das häufigste Objekt in der 3D-Computergrafik ist das *Triangle* (dt. Dreieck). Der Vorteil von Triangles ist, dass sie durch ihren einfachen Aufbau schnell verarbeitet und dargestellt werden. An dieser Stelle unterscheiden sich OpenGL und Direct3D. Direct3D ist auf Triangles angewiesen, während OpenGL mit Polygonen - also mit „echten“ Vielecken - arbeiten kann.

3D-Objekte die als *Primitives* (dt. Grundkörper) bezeichnet werden, sind Objekte (z.B.: Würfel, Zylinder, Kugel,...) die durch Parameter (z.B.: Höhe, Durchmesser,...) beschrieben und durch Algorithmen erst in ein Mesh mit Polygonen, Edges und Vertices zerlegt werden.

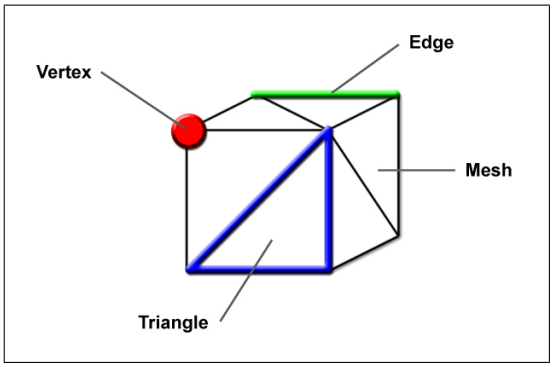


Abbildung 3.1: Objektbeschreibung

3.2 Transformationen

Damit das 3D-Objekt nicht statisch wie ein Foto wirkt, können auf 3D-Objekte oder Kameras Transformationen angewandt werden. Durch Transformationen ist es möglich, Lage, Position und Größe der Objekte zu verändern. Es gibt drei Varianten der Transformation: *Translation* (Verschiebung), *Rotation* und *Skalierung* (Verändern der Größe).

Transformationen werden oft als 4x4 Matrizen dargestellt, da sich so mehrere Transformationen durch Multiplikation der Transformationsmatrizen (s. 2.3.3) kombinieren lassen. Die Reihenfolge der Multiplikation der Matrizen ist von entscheidender Bedeutung, denn unterschiedliche Reihenfolgen führen zu unterschiedlichen Ergebnissen (s. Abb. 3.2).

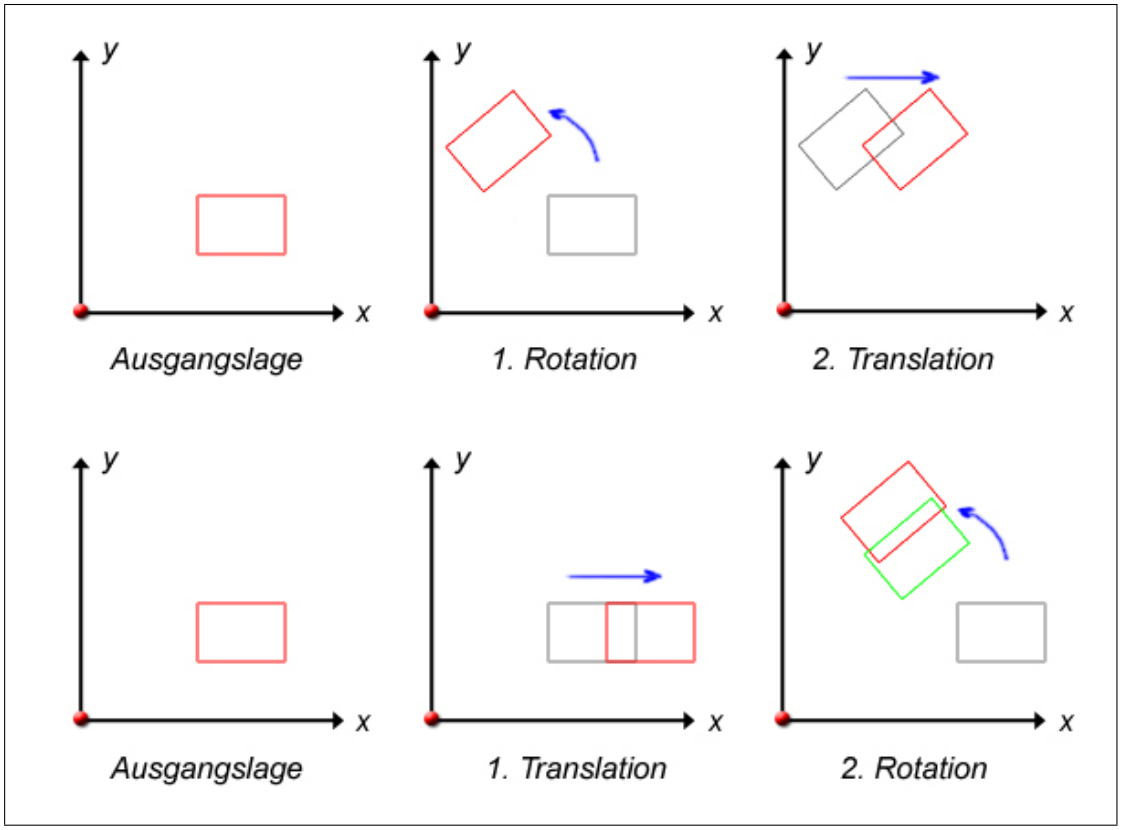


Abbildung 3.2: Reihenfolge der Transformationen entscheidend: 1. Durchlauf: Rotation dann Translation / 2. Durchlauf: Translation dann Rotation (Zum Vergleich: Endposition des 1. Durchlaufes als grünes Viereck im Koordinatensystem des 2. Durchlaufs dargestellt)

Eine Transformation eines Objektes erfolgt, indem alle Vertices des Objektes mit

der entsprechenden Transformationsmatrix multipliziert werden (Vektor-Matrix-Multiplikation)(s. 2.3.4).

Die folgenden Unterpunkte behandeln die Erzeugung der entsprechenden Matrizen für die einzelnen Transformationen.

3.2.1 Translation

Die Matrix für die Translation sieht wie folgt aus:

$$(\acute{x}\ \acute{y}\ \acute{z}\ 1) = (x\ y\ z\ 1) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{pmatrix}$$

Damit wird der Punkt (x, y, z) zu den neuen Koordinaten $(\acute{x}, \acute{y}, \acute{z})$ verschoben. T_x, T_y und T_z definieren die Verschiebungsweite entlang der entsprechenden Achsen.[1]

3.2.2 Rotation

Für jede der drei Rotationsachsen (X-, Y- und Z-Achse) gibt es eine spezielle Matrix.

Rotation um X-Achse:

$$(\acute{x}\ \acute{y}\ \acute{z}\ 1) = (x\ y\ z\ 1) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation um Y-Achse:

$$(\acute{x}\ \acute{y}\ \acute{z}\ 1) = (x\ y\ z\ 1) \begin{pmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation um Z-Achse:

$$(\acute{x}\ \acute{y}\ \acute{z}\ 1) = (x\ y\ z\ 1) \begin{pmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

[1] vgl. [7] S. 170/171

Der jeweilige Rotationswinkel in Radient ist θ. Beachten sollte man aber, dass die Rotation jeweils um den Ursprung des Koordinatensystems des Objektes stattfindet (s. Abb. 3.3). Soll ein Objekt um einen bestimmten Punkt gedreht werden, so muss dieser Punkt in den Ursprung verschoben werden. Entsprechend wird auch das Objekt verschoben. Nachdem dann die Rotation durchgeführt ist, wird das Objekt um den gleichen Betrag zurück bewegt.[2]

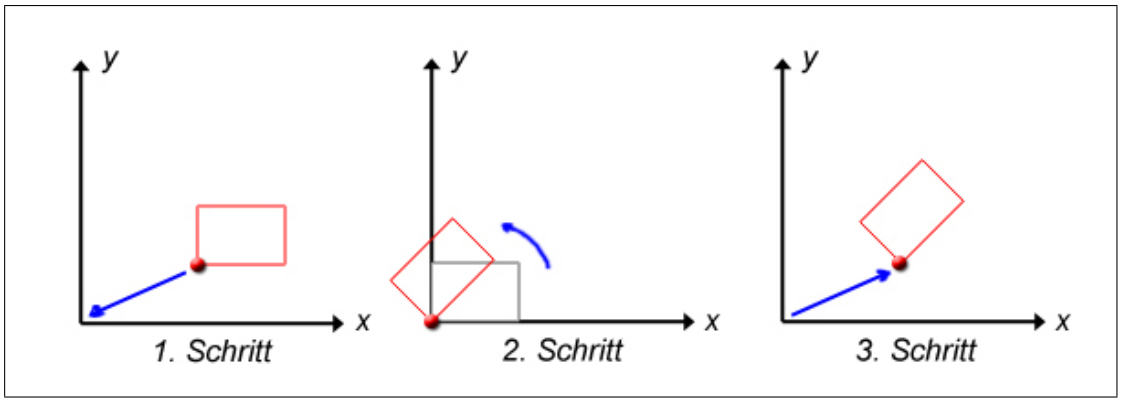


Abbildung 3.3: Rotation um einen bestimmten Punkt

3.2.3 Skalierung

Die Matrix für die Skalierung sieht wie folgt aus:

$$(\acute{x}\ \acute{y}\ \acute{z}\ 1) = (x\ y\ z\ 1) \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Die Skalierungsfaktoren für die entsprechenden Koordinatenachsen sind hierbei s_x, s_y und s_z. Wie bei der Rotation bezieht sich auch die Skalierung auf den Koordinatenursprung. Soll also ein anderer Punkt als Zentrum der Skalierung dienen, muss genauso wie oben bei der Rotation beschrieben, verfahren werden.[3]

[2] vgl. [7] S. 172/173
[3] vgl. [7] S. 171/172

3.3 3D-Räume und deren Transformationen

3.3.1 Räume in der 3D-Computergrafik

Bei den gerade betrachteten Transformationen wurde als Basis für die Transformationen das jeweilige Koordinatensystem des Objektes verwendet. Die 3D-Szene, in der sich möglicherweise das Objekt befindet, besitzt ein anderes Koordinatensystem. Die unterschiedlichen Koordinatensysteme werden auch *Räume* (engl. Space) genannt.

So gibt es, wie oben erwähnt, einen *Model Space* - das eigene (lokale) Koordinatensystem jedes Objektes. In diesem Raum wird das Objekt konstruiert, in welchem man auch arbeiten sollte, um das einzelne 3D-Objekt zu transformieren.

Der *World Space* ist das Koordinatensystem der Szene. In diesem Raum werden die Objekte platziert und Lichtberechnungen ausgeführt.

Da alle 3D-Objekte durch die Linse der Kamera betrachtet werden, besitzt diese auch einen eigenen Raum - den *View Space*. Entsprechend der Position und der Blickrichtung der Kamera, sieht man die 3D-Objekte aus einen bestimmten Blickwinkel. Dieser Raum verändert sich, wenn die Kamera durch die Szene bewegt wird.

Als *Screen Space* wird die zweidimensionale Darstellung der Szene auf dem Bildschirm beschrieben[4]. Da die unterschiedlichen Räume benötigt werden, um eine 3D-Szene realistisch darzustellen, benötigt man Transformationen, um zwischen den Räumen zu wechseln. Auf diese möchte ich kurz eingehen.

3.3.2 Welt-Transformation

Die Welt-Transformation ermöglicht, ein 3D-Modell aus dem *Model Space* in den *World Space* zu transformieren. Dabei wird für jedes Objekt eine Transformationsmatrix gebildet. Dadurch ist es möglich, an dem Objekt die im Abschnitt 3.2 genannten Grund-Transformationen (Translation, Rotation und Skalierung) durchzuführen.

3.3.3 Kamera-Transformation

Den Vorgang der Transformation vom *World Space* zum *View Space* nennt man Kamera- oder auch Sicht-Transformation. Bei dieser Transformation wird der Stand-

[4] vgl. [3] S. 459

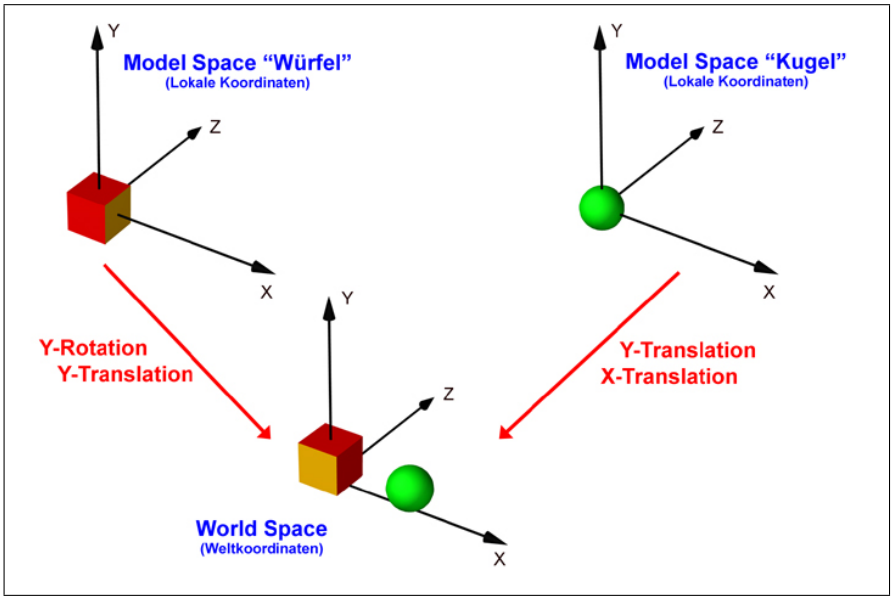


Abbildung 3.4: Ablauf der Welt-Transformation

punkt und Blickwinkel der Kamera einbezogen. Der Kamerastandpunkt (im World Space) wird in den Ursprung des Koordinatensystems verschoben. Danach wird das Koordinatensystem entsprechend des Blickwinkels der Kamera, um diesen Punkt gedreht.

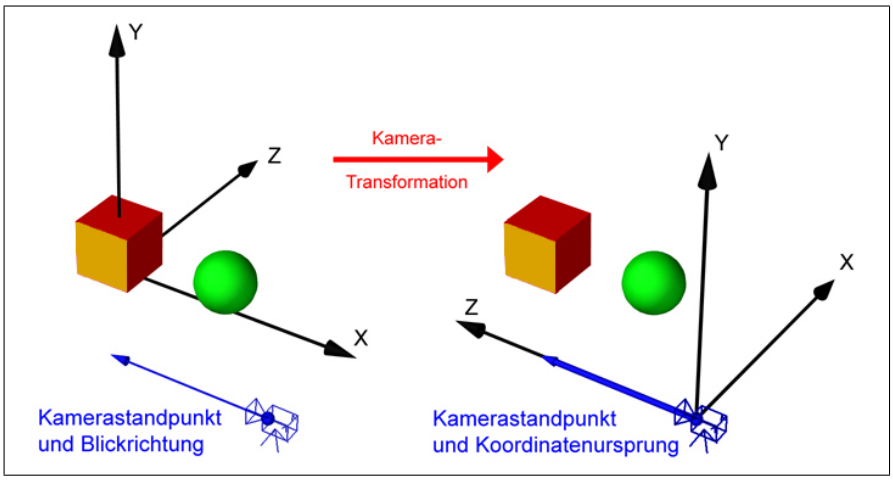


Abbildung 3.5: Ablauf der Kamera-Transformation

3.3.4 Projektions-Transformation

Durch diese Transformation wird die 3D-Szene auf den zweidimensionalen Bildschirm projiziert. Dabei wird üblicherweise eine perspektivische Projektion verwendet, bei der Objekte nahe der Kamera größer erscheinen, als von der Kamera weiter entfernte Objekte.

Um zu entscheiden, welche Objekte sichtbar sind, wird ein sogenanntes *Viewing Volume* verwendet. Es beschreibt den sichtbaren Ausschnitt der 3D-Szene. Die maximale Sichtweite wird dabei durch die *Far Clipping Plane* und die minimale Sichtweite durch die *Near Clipping Plane* begrenzt. Das Sichtfeld (engl. *Field of View* - kurz FoV) entspricht dem Blickfeld der Kamera.

Die Projektions-Transformation transformiert das *Viewing Volume* in einen Quader und der Koordinatenursprung wandert in die Mitte des Sichtbereichs der Kamera[5].

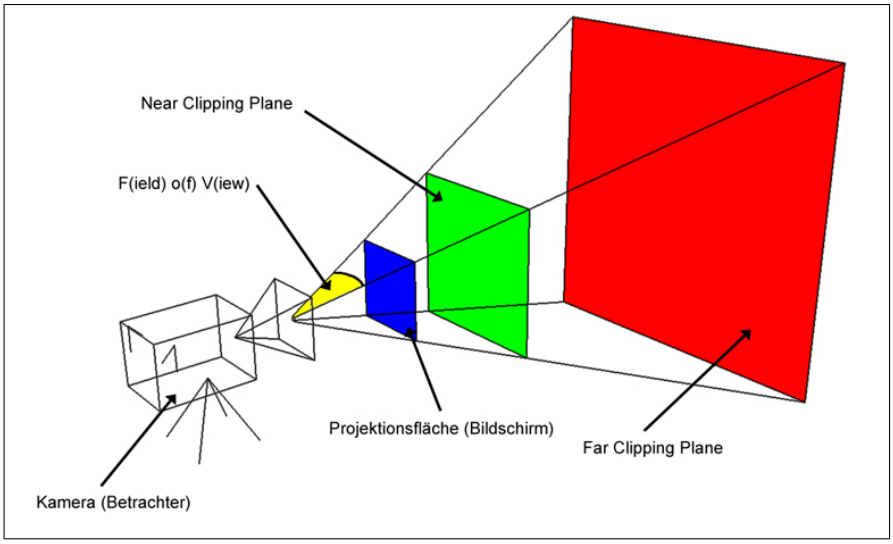


Abbildung 3.6: Die Viewing Volume

3.4 Shading

Durch *Shading*, im deutschen auch Schattierung genannt, bestimmt der Nutzer, wie Lichteffekte oder Farben auf den 3D-Objekten dargestellt werden. Die Qualität des Shading wird durch die unterschiedliche Berechnung der Normalen

[5] vgl. [4] S. 41-44 und [8] S. 14-16

und den Winkel der einfallenden Lichtstrahlen beeinflusst. Die drei bekanntesten Shading-Verfahren möchte ich an dieser Stelle kurz vorstellen.

3.4.1 Flat-Shading

Flat-Shading ist die am wenigsten rechenintensive und einfachste Art des Shading.

Dabei wird bei diesem Verfahren für jedes Polygon nur ein Normalenvektor berechnet. Somit besitzen die Vertices des Polygons den gleichen Normalenvektor und das Polygon wird entsprechend seines Winkels zum Licht nur ein Farbwert zugewiesen.

Der Nachteil bei dem Verfahren ist, dass bei runden Objekten die Übergänge zwischen den einzelnen Polygone sichtbar werden.

Abbildung 3.7: Teapot mit Flat Shading

3.4.2 Gouraud Shading

Bei *Gouraud Shading* werden hingegen die Objekte viel weicher dargestellt und die scharfen Kanten zwischen den Polygonen verschwinden.

Dies wird erreicht indem jedem Vertex ein Normalenvektor und somit ein Farbwert zugewiesen wird. Innerhalb des Polygons werden dann Zwischenwerte (Interpolation) gebildet, die einen Übergang zwischen den Farbwerten der Vertices erzeugen.

Abbildung 3.8: Teapot mit Gouraud Shading

3.4.3 Phoung Shading

Bei diesem Shading wird wie bei Gouraud für jeden Vertex ein Normalenvektor berechnet. Es werden jedoch nicht innerhalb des Polygons die Farbwerte, sondern die Normalen interpoliert. So erhält jedes Pixel des Polygons einen interpolierten Vektor.

Durch dieses Verfahren erhält das 3D-Modell die realistischste Farbdarstellung, gleichzeitig ist dies aber auch die rechenintensivste Variante.

Abbildung 3.9: Teapot mit Phoung Shading

3.5 Materialien

Damit ein 3D-Objekt nicht nur als Drahtgittermodell dargestellt wird, ist es möglich, diesem Materialeigenschaften zu zuweisen. Somit kann ermittelt werden,

in welcher Weise das Objekt Licht reflektiert. Dazu gehört beispielsweise, ob die Oberfläche glatt oder glänzend ist oder in welcher Farbe das Objekt leuchten soll.

3.6 Texturen

Eine *Textur* ist eine zweidimensionale Grafik, die auf das 3D-Objekt projiziert werden kann. Das Aufbringen von Texturen auf Objekte bezeichnet man als *Texture Mapping*. Ein Pixel in der Textur wird als *Texel* bezeichnet. Damit ein 3D-Objekt texturiert werden kann, muss jeder Vertex Texturkoordinaten besitzen. Die Koordinaten von Vertices auf der Textur werden durch die Zeilen- und Spaltennummer u und v angegeben, deshalb auch oft als *UV Mapping* bezeichnet. Dies sind Faktoren für die Höhe und Breite der Textur und nehmen deshalb nur Werte zwischen 0.0 und 1.0 an.

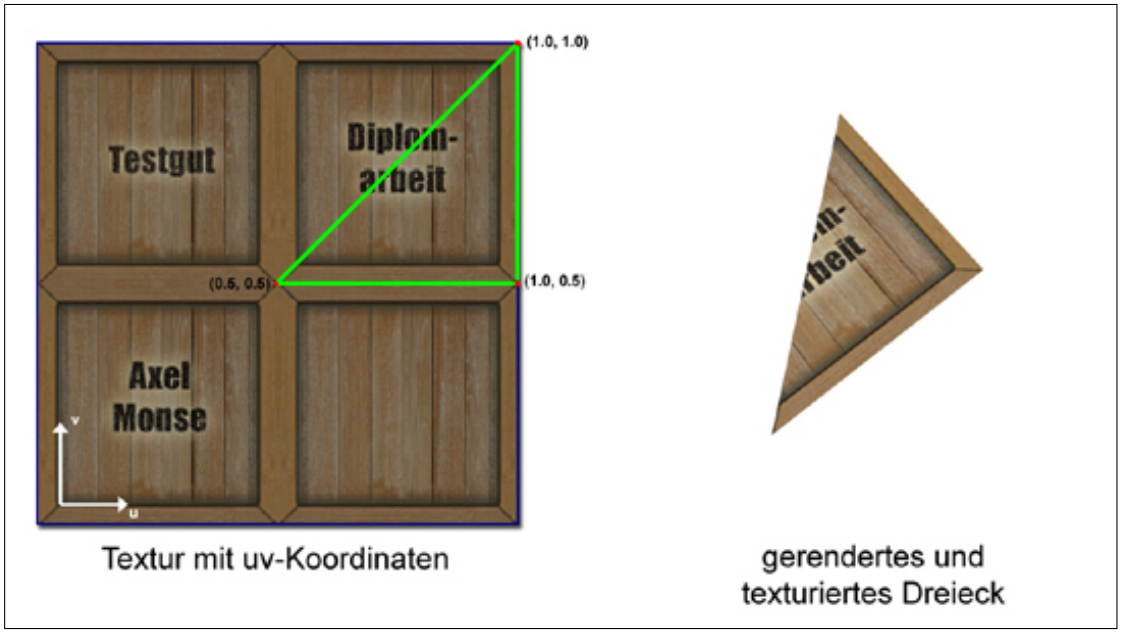


Abbildung 3.10: Texturierung mit uv-Koordinaten

Mapping-Arten

Es gibt vier bekannte Arten von UV-Mapping (Abb. 3.11). Mit ihnen lassen sich leicht Grundkörper texturieren.

Planar Mapping

Das *Planar Mapping* arbeitet wie ein Projektor. Die Textur wird von einer Richtung auf das 3D-Objekt projiziert. Diese Mapping-Art wird häufig für eindimensionale Wände oder Ebenen genutzt.

Box Mapping

Bei dieser Mapping-Art wird die Textur von sechs verschieden Seiten auf das 3D-Modell projiziert. Diese Variante bietet sich zum Texturieren von Quader oder Würfel an.

Spherical Mapping

Spherical Mapping umhüllt das 3D-Objekt und projiziert die Textur von allen Seiten auf das Objekt. Diese Mapping-Art eignet sich hervorragend für kugelförmige Objekte.

Cylindrical Mapping

Schlussendlich wird bei *Cylindrical Mapping* das 3D-Objekt in einer zylindrischen Form umhüllt. Die Deckel des Zylinders werden durch *Planar Mapping* texturiert. Diese Art wird für Pfeiler, Baumstämme, Rohre oder ähnliche Objekte verwendet.[6]

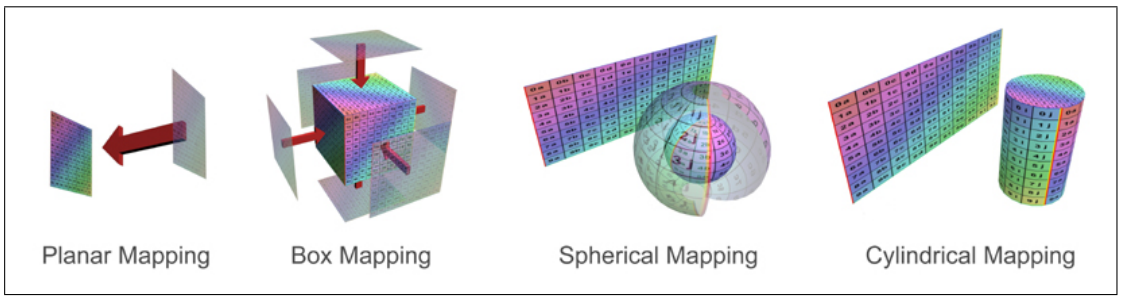


Abbildung 3.11: Mapping-Arten im Vergleich

[6] vgl. [10] S. 61-64

3.7 Polygon-Sortierung und Z-Buffer

Bei komplexen Szenen mit mehreren sich gegenseitig verdeckenden 3D-Elementen, wird ein Verfahren benötigt, welches die Elemente in der Szene richtig darstellt.

Auf den ersten Blick würde sich ein Verfahren anbieten, dass berechnet welches Objekt zuerst und welches Objekt zuletzt dargestellt wird. Dabei wird der Szenenaufbau beim vom Betrachter entferntesten Objekt begonnen und endet mit der Darstellung des vom Betrachter am nächsten liegenden Objekt. Doch dieses Verfahren führt bei sich ineinander überschneidenden Polygonen zu Problemen, da man nicht genau entscheiden kann, welches Polygon zuerst gezeichnet werden soll. Deshalb bietet sich eine andere Methode an.

Dieses Verfahren ist auch als *Z-Buffering* bekannt. In dem Z-Buffer - ein zweidimensionaler Array (dt. Feld) in der Größe der Bildschirmauflösung - wird die z-Koordinate jedes Pixel eines Objektes gespeichert, dass dargestellt wird.

Wenn also das nächste Objekt dargestellt werden soll, wird die z-Koordinate des darzustellenden Pixel mit der im Z-Buffer verglichen. Falls das Pixel, eine kleinere z-Koordinate besitzt, demzufolge näher am Betrachter ist, wird es dargestellt und ihre z-Koordinate an die entsprechende Stelle des Z-Buffer-Arrays eingetragen. Dabei werden die Werte von weiter entfernten Pixeln überschrieben.

Durch das Z-Buffering lässt sich demzufolge auch das oben genannte Problem lösen.

4 Einführung in X3D

4.1 Allgemein

Der Beginn von Beschreibungssprachen für 3D-Szenen war das Jahr 1994. Mark Pesce und Tony Parisi entwickelten nach der ersten World Wide Web Konferenz in Genf ein Konzept für eine eigene Programmiersprache zur Darstellung von 3D-Szenen. Als Grundlage für diese neue Sprache ging aus einem Wettbewerb das ASCII-Format des OpenInventors von *Silicon Graphics Inc.* hervor. Aus dieser entstand kurze Zeit später *VRML 1.0*[1].

Dessen Nachteil bestand darin, dass nur statische 3D-Szenen beschrieben werden konnten. So folgte 1997 das verbesserte und erweiterte *VRML 2.0* und wurde zum internationalen Standard erklärt. Durch diesen Standard ist es auch unter dem Namen *VRML97* bekannt.

Um VRML als offenen 3D-Grafik-Standard zu erhalten und weiter zu verbessern, wurde durch Initiative von Firmen und Hochschulen, das *Web3D Consortium* gegründet. Dieses entwickelte die dritte Generation der 3D-Beschreibungssprache - mit dem Namen *X3D*. Dabei steht X3D für eXtensible 3D Graphics. In dieser neuen Version wurde die XML-Syntax integriert, um leichter mit anderen Internet-Technologien zu interagieren. Gleichzeitig ist es weiterhin möglich, in der alten VRML-Syntax zu programmieren.

Im Jahr 2004 wurde X3D in der Version 3.0 als internationaler Standard aufgenommen. [2] [3]

4.2 Profile

Die 3D-Beschreibungssprache X3D ist modular aufgebaut, d.h. dass die gesamte Spezifikation in sogenannte *Profile* unterteilt ist. Die aktuelle X3D-Version unterscheidet zwischen sieben Profilen.[4] Die Profile weichen in Komplexität und Funktionalität erheblich voneinander ab:

[1] Virtual Reality Modelling Language
[2] vgl. http://www-lehre.informatik.uni-osnabrueck.de/~okrone/DIP/node33.html [15.01.2009]
[3] vgl. [2] S.3
[4] vg. [2] S. 14

- **Core Profile:** Minimale Dateiinformationen (Meta-Daten), benötigte Komponenten und Level müssen explizit angegeben werden
- **Interchange Profile:** Basisprofil zum Austausch zwischen 3D-Applikationen, alle Basisgeometrie-, Aussehens-(Material und Textur) und Keyframe-Animationsknoten sind enthalten
- **Interactive Profile:** Erweiterte Implementation, die mehr Interaktion in der Szene erlaubt
- **MPEG-4 interactive Profile:** Angepasstes Profil für die Bedürfnisse der MPEG4-Multimedia-Spezifikation zur Beschreibung von 3D-Elementen.
- **CADInterchange:** Unterstützung des Imports von CAD-Modellen, alle Knoten vom Interchange Profile und zusätzliche Knoten für CAD
- **Immersive Profile:** Entspricht dem VRML97-Standard
- **Full Profile:** Komplette Implementation der X3D-Spezifikation

Im Rahmen dieser Diplomarbeit werde ich mich, wenn nicht anders ausdrücklich erwähnt, auf das *Interchange Profile* konzentrieren.

5 Einführung in DirectX und Direct3D

5.1 Allgemein

Die Ära von *DirectX* begann Ende 1995. In dieser Zeit führte Microsoft die erste Version von DirectX ein. Bis zu diesem Zeitpunkt dominierte MS-DOS den Grafikbereich. Denn die Stärke von DOS lag darin, dass Programmierer mit ihren geschriebenen Anwendungen direkt und schnell die Hardware ansprechen konnten. Dagegen hatte Windows erhebliche Schwierigkeiten, schnell auf Sound- und Grafikkarten zu zugreifen.

Microsoft benötigte deshalb für ihr damals neues Betriebssystem eine API[1], um mit Anwendungen beschleunigt, direkt und einheitlich auf die Hardware der PCs zugreifen zu können. Nur so war es möglich die neuen Fähigkeiten von Windows, wie Multitasking-Fähigkeit und zentrale Hardware-Verwaltung, im vollem Umfang zu verwenden.

Gleichzeitig wollte man mit der neuen API auch die Schwäche von MS-DOS auskurieren. In MS-DOS hatte man die Möglichkeit mit Programmen direkt die Hardware anzusprechen. Es konnten jedoch nur Funktionen genutzt werden, von denen man sich sicher war, dass dies auch alle Grafikkarten verarbeiten können. Wollte man soviel wie möglich Funktionen nutzen, wäre die einzige Alternative gewesen, jede Karte einzeln zu unterstützen. Dies hätte wiederum zu sehr viel Programmieraufwand geführt[2][3].

5.2 Architektur

Die neue API namens DirectX mit seiner gut organisierten Architektur sollte nun die oben genannten Probleme beheben. Die Architektur besteht aus zwei Schichten. Der *HAL*[4] und der *HEL*[5]. Dabei repräsentiert die HAL Funktionen, die von der verfügbaren Hardware unterstützt werden. Falls jedoch eine Funktion nicht unter-

[1] Application Programming Interface
[2] vgl. [3] S. 288
[3] vgl. [7] S. 113
[4] Hardware Abstraction Layer
[5] Hardware Emulation Layer

stützt wird, springt HEL ein. Es versucht die Funktionen bis zu einem gewissen Grad zu emulieren.

So wird z.B. eine Grafikfunktion statt von der Grafikkarte (via HAL) von der GDI[6] ausgeführt. Dies führt dann natürlich zu Geschwindigkeitsverlusten gegenüber der Darstellung mit einer Grafikkarte.[7]

5.3 Komponenten

DirectX besteht heutzutage aus vielen verschiedenen Komponenten. Die bekanntesten unter ihnen sind.:

Direct3D und DirectDraw (DirectX Graphics), DirectInput und DirectSound.

Da ich in meiner Diplomarbeit keine komplexe Multimediaanwendung, sondern nur einen 3D-Viewer erstelle, möchte ich nur kurz auf die DirectX Graphics-Komponente eingehen.

5.4 DirectX Graphics und Direct3D

Direct3D ging aus der 3D-API „Relatiy Labs“ von der Firma *RenderMorphic* hervor und ist seit der dritten Version von DirectX in der API integriert. Vor DirectX3 gab es keine direkte 3D-Unterstützung von Microsoft. Seit DirectX8 hat Microsoft Direct3D und DirectDraw zu einer Komponente namens DirectX Graphics zusammengelegt. Diese Komponente nimmt den größten und komplexesten Teil von DirectX ein. DirectX Graphics beinhaltet Funktionen für Geometrie-Transformationen, Beleuchtung, Texturierung, Render-Einstellungen usw. und erleichtert somit dem Anwender, eine Direct3D-Renderumgebung zu erstellen.

[6] Graphic Device Interface
[7] vgl. [3] S. 289

6 Vorüberlegungen zur Aufgabenstellung

6.1 Allgemein

Mein Aufgabenstellung ist, wie in der Einleitung kurz erwähnt, die Darstellung von X3D-Objekten in einer Direct3D-Umgebung. Hier möchte ich ein paar Vorüberlegungen zu dieser Aufgabe anstellen.

6.2 Was gibt es schon?

Um einen besseren Blick auf die Herangehensweise zur Bearbeitung der Aufgabe zu bekommen, ist es hilfreich zu recherchieren, ob schon Ähnliches realisiert wurde. Ebenso möchte ich untersuchen, wie die schon existierenden Programme diese Aufgabe abdecken und welche weiteren Funktionalitäten sie aufweisen. Dabei bin ich auf folgende Programme gestoßen, die Untermengen meiner Aufgabe schon gelöst haben (siehe auch Tabelle Anhang A):

BS Contact

Das Programm „BS Contact“ wurde von der Firma *Bitmanagment Software GmbH* entwickelt und ist ein Browser-Plug-in zur Darstellung von X3D-Objekten. Dabei werden die X3D-Szenen wahlweise in Direct3D oder in OpenGL dargestellt. Lesbar sind X3D- und VRML-Dateien. Das Programm ist als technisch eingeschränkte Version kostenlos verfügbar.[1]

Octaga Player

Die Firma *Octaga* präsentiert einen Player der als Browser-Plug-in sowie als Stand-Alone-Programm verwendet werden kann. Die X3D-Szenen werden jedoch nur in OpenGL dargestellt. Dafür können wiederum X3D- und VRML-Dateien eingelesen werden. Eine kostenlose Testversion kann man auf der Firmenwebsite

[1] http://www.bitmanagement.de

herunterladen. Diese Version ist in der Funktionalität im Gegensatz zur kostenpflichtigen Vollversion eingeschränkt.[2]

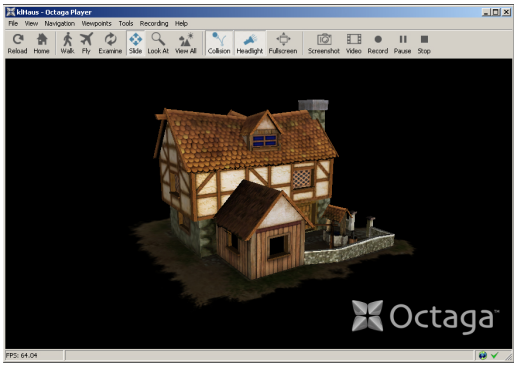


Abbildung 6.1: Screenshot des Octaga Players

AccuTrans3D

Bei „AccuTrans3D“ handelt es sich nicht im eigentlichen Sinne um einen Viewer für X3D-Dateien. Das Programm von *Micro Mouse Productions* ist ein Konverter für verschieden 3D-Dateien. Dabei werden in dem Stand-Alone-Programm alle 3D-Daten in OpenGL dargestellt. Bei den für uns relevanten Dateiformaten sind nur unkomprimierte VRML-Dateien lesbar. Dafür ist es möglich, unzählige 3D-Formate in VRML- und X3D-Dateien zu konvertieren. Es ist eine voll funktionsfähige, aber zeitlich eingeschränkte Testversion des Programms verfügbar.[3]

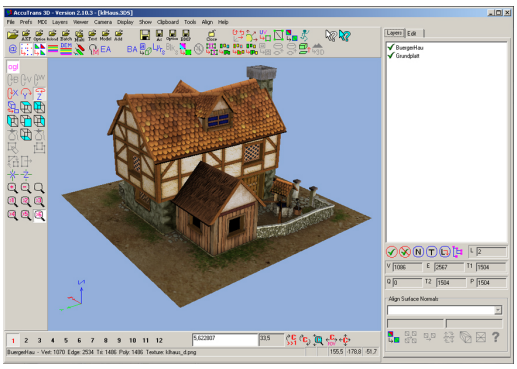


Abbildung 6.2: Screenshot des AccuTrans3D

[2] http://octaga.com
[3] http://www.micromouse.ca/

SwirlViewer

Der „SwirlViewer“ von *Pinecoast Software* ist ein einfacher Viewer. Er ähnelt in seinem Umfang stark dem „Octaga Player“. Das Tool ist ebenso als Plug-in und Stand-Alone-Programm nutzbar und stellt die X3D-Szenen in OpenGL dar. Es sind VRML- und X3D-Dateien lesbar, zeigte jedoch einige Schwächen beim Darstellen von komplexen VRML- und X3D-Dateien. Die Anwendung ist kostenlos und zeitlich unbegrenzt als Vollversion verfügbar.[4]

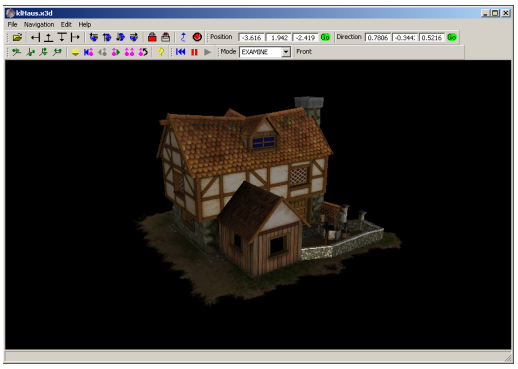


Abbildung 6.3: Screenshot des SwirlViewer

Xj3D Browser

Wie der Name des Programm schon verrät, handelt es sich um ein Browser-Plug-in des *Web3D Consortiums* - den Entwicklern von X3D. Das Programm wurde mit Java programmiert und die X3D-Objekte werden in Java3D dargestellt. Deshalb muss die Java J2SE Laufzeitumgebung auf dem System installiert sein, um diesen Viewer zu verwenden. „Xj3D“ kann VRML- und X3D-Szenen darstellen und legt dabei einen sehr hohen Wert auf die Wohlgeformtheit der X3D-Dateien.[5]

Enceladus

Auf meinem Besuch der *Hannover Messe* 2009 hatte ich die Möglichkeit diesen kostenfreien X3D-Browser der Firma *dssd* zu testen und mit dem Entwickler Dirk Schulz zu sprechen. Der Viewer ist ein Stand-Alone-Programm, dass die X3D-Szenen in OpenGL darstellt. In seinem Umfang ähnelt er stark dem „SwirlViewer“. In seiner erweiterten aber kostenpflichtigen Version namens „Calypso“

[4] http://www.pinecoast.com
[5] http://www.xj3d.org

ist es möglich, mit dem Programm mittels Datenhandschuh oder -helm durch die 3D-Szenen zu navigieren. Die X3D-Szenen können mit Hilfe des Autorentools „Dione“ vom gleichen Software-Entwickler sehr leicht erstellt werden. [6]

Zielsetzung

Ich denke, dass viele Benutzer Probleme sehen, wenn sie entsprechende Plug-ins oder Applets in ihrem Browser installieren müssen. Sie nutzen eher ein Stand-Alone-Programm, das sie getrennt vom Internet und in einem selbständigen Programm bedienen können. Außerdem nutzen viele PC-Anwender das Windows-Betriebssystem und haben DirectX und somit auch Direct3D installiert.

In dieser vorliegenden Arbeit werde ich einige leistungsstarke Funktionen der analysierten Programme übernehmen, aber auch einen Schritt weitergehen und den Benutzern die Arbeit mit X3D-Objekten erleichtern. Dies möchte ich erreichen, indem ein Viewer entsteht, der als Stand-Alone-Programm läuft und X3D-Dateien in einer Direct3D-Umgebung darstellt.

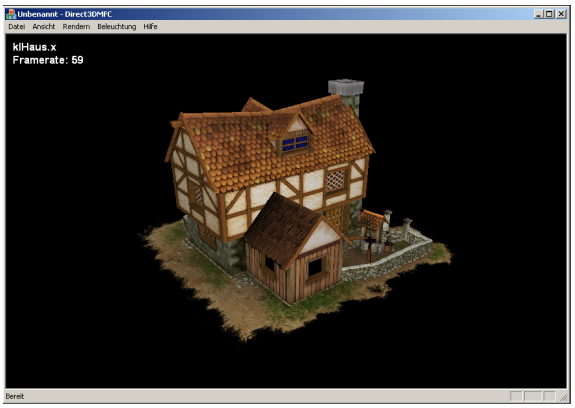


Abbildung 6.4: Screenshot des Viewers der Diplomarbeit

6.3 Wie lese ich X3D-Daten ein?

Da die Beschreibung der Daten von X3D in einer XML-Syntax möglich ist, bietet sich ein *XML-Parser* zum Einlesen der X3D-Daten an. Ein *XML-Parser* ist ein Programm oder eine Klasse, der eine in XML geschriebene Datei analysieren und deren enthaltene Daten (Knoten, Attribute) für die weitere Verarbeitung zur

[6] http://www.3dione.com

Verfügung stellen kann. Dass entspricht genau den Anforderungen, um die X3D-Daten einzulesen und später zu verarbeiten.

Da ich als Entwicklungsumgebung Visual Studio 2003 wähle, in dem das .NET Framework enthalten ist, besteht die Möglichkeit, auf den Namespace `System::Xml` zu zugreifen.

Der Namespace umfasst vier Klassen - `XmlTextReader`, `XmlValidatingReader`, `XmlTextWriter` und `XmlDocument`.

Da ich beim Zugriff auf die X3D-Dateien annehme, dass diese auch wohlgeformt sind, fällt aus meiner Betrachtung der `XmlValidatingReader` zum Lesen und Prüfen auf Gültigkeit von XML-Dateien heraus. Ebenso kann der `XmlTextWriter` von der Auswahlliste gestrichen werden, da ich im Rahmen der Diplomarbeit keine X3D-Dateien erstellen möchte.

Die Klasse `XmlDocument` wiederum implementiert ein W3C-DOM[7] und ermöglicht somit die Darstellung der kompletten X3D-Datei im Speicher. In der im Arbeitsspeicher nachgebildeten XML-Datei wäre es dann möglich, mit XPath Knoten zu selektieren oder schnell zwischen Knoten hin und her zu springen. Doch der Nachteil liegt darin, dass zu große X3D-Dateien nicht in einer DOM aufgenommen werden können.

Schlussendlich bietet die Klasse `XmlTextReader` die beste Wahl. Sie bietet die Möglichkeit, X3D-Dateien mit minimaler Ressourcenbeanspruchung zu analysieren. Die Daten der Dateien werden von Anfang bis Ende durchlaufen und erkennen Elemente während des Lesens. Der `XmlTextReader` arbeitet nach dem „Pull“-Methode. Um einen Knoten zu erkennen oder zu lesen, muss man die jeweilige Funktion aufrufen. Dies entspricht dem gegenteiligen Prinzip der „Push“-Methode des bekannten SAX[8]-Parsers. Dabei werden Ereignisse ausgeworfen, wenn der Parser auf einen bestimmten Knoten trifft. Der einzige Nachteil beim `XmlTextReader` zu DOM besteht darin, dass man nicht zu einem früheren Punkt zurückspringen kann, außer man liest die Datei erneut ein.

Da ich aber die gesamte X3D-Datei parse und sie in einem Durchlauf in eine X-Datei konvertiere, bietet sich die Klasse `XmlTextReader` hervorragend an.

6.4 Wie versteht Direct3D die X3D-Daten?

Um X3D-Szenen in Direct3D darzustellen, muss man die Frage klären, in welcher Form man in Direct3D die X3D-Daten präsentiert, damit die Szenen fehlerfrei wiedergegeben werden.

[7] Document Object Modell
[8] Simple API for XML

Zuerst ist dabei zu beachten, dass beide Systeme unterschiedliche Koordinatensysteme (rechtshändiges, linkshändiges) benutzen. Dieser Unterschied wirkt sich besonders auf Transformationen, Vertex- und Texturkoordinaten, sowie beim Polygon-Aufbau aus. Die nötigen Konvertierungsmethoden von X3D in Direct3D-Daten sollte in Form einer Klasse enthalten sein. So stellt sich nur noch die Frage, wie der Renderer die konvertierten Daten liest.

Eine Möglichkeit wäre, den Konverter und die Renderumgebung in ein Programm zu fassen. Dabei müsste man viele Methoden schreiben, damit die Daten in Direct3D dargestellt werden. Eine andere Variante ist, den Konverter von der Renderumgebung zu trennen. Hierbei stellt sich aber die Frage, in welcher Form die beiden Programme kommunizieren. Doch diese Frage ist schnell beantwortet: Als Lösung bietet sich an, die X3D-Datei in eine DirectX-Datei zu konvertieren. Anschließend lädt die Renderumgebung die konvertierte Datei mit der DirectX-Standardmethode `D3DXLoadMeshFromX`.

Damit spart man auch eine Menge Zeit- und Programmieraufwand, die beim Programmieren einer eigenen Methode zum Laden und Interpretieren der konvertieren X3D-Daten notwendig würde.

6.5 Welche Werkzeuge benötige ich?

Um die Möglichkeit zu geben meine Arbeit, durch Programmieren und Testen nachvollziehen zu können, werden im Folgenden die Software- und Hardwarespezifikationen meiner Testumgebung vorgestellt.

Die wichtigste Komponente die auf dem System installiert sein sollte ist das DirectX 9 SDK mit dem Release vom August 2003. Man wird sich jetzt fragen, warum die Version 9 und nicht die aktuelle Version 10 von DirectX benutzt wurde. Ich hatte mich für die ältere Version entschieden, da diese auf dem Betriebssystem „Windows XP“ stabil läuft und zu diesem Zeitpunkt eine der ausgereiftesten Versionen ist. Ebenso sprach für diese DirectX-Version die Möglichkeit, sich in Publikationen, auf Webseiten und -foren in dieses Thema zu vertiefen.

Im Folgenden meine Testumgebung:
- Dell Precision NM6300 (Notebook)
- Intel Core 2 Duo (2x2GHz)
- 1GByte RAM
- Windows XP SP2
- DirectX 9 SDK Release August 2003
- Visual Studio.NET 2003

7 Vergleich von X3D- und X-Format

7.1 X3D-Format

Die 3D-Beschreibungssprache kennt zwei Dateiformate in der man die 3D-Szene abspeichern kann: `*.x3dv` und `*.x3d`. Dabei wird die Datei-Endung `*.x3dv` für Dateien benutzt die 3D-Szenen in der klassischen VRML-Syntax enthalten. Dateien mit der Endung `*.x3d` beinhalten in der XML-Syntax von X3D verfasste 3D-Szenen. Ist das jeweilige Format komprimiert, wird der Buchstabe `z` (für zipped) angehängt. Bei Dateien mit der Endung `*.wrl` handelt es sich in der Regel um VRML-Szenen des Standards VRML2.0 oder älter. Deshalb sind es streng gesehen keine X3D-Dateien.

Im weiteren Verlauf der Diplomarbeit werde ich mich mit dem X3D-Format in der XML-Syntax (`*.x3d`) befassen.

Der übliche Aufbau einer solchen X3D-Datei mit einem Interchange-Profil zur Darstellung eines roten, in der Szene verschobenen Würfels sieht wie folgt aus:

```
<X3D version='3.0' profile='Interchange'>
<head>Meine kleine X3D-Datei</head>
  <Scene>
    <Group>
      <Transform translation='3.0 0.0 1.0'>
        <Shape>
          <Box scale='2.3 4 5.3'/>
          <Appearance>
            <Material diffuseColor='1.0 0.0 0.0'/>
          </Appearance>
        </Shape>
      </Transform>
    </Group>
  </Scene>
</X3D>
```

Listing 7.1: Typische X3D-Datei

Die wesentlichen Knoten und deren Attribute, auf die man bei der Durchmusterung einer X3D-Datei im Interchange-Profil treffen kann, sind in Abb. 7.1 zu finden. Diese Grafik gibt aber nicht die zwingende Reihenfolge der Knoten an.

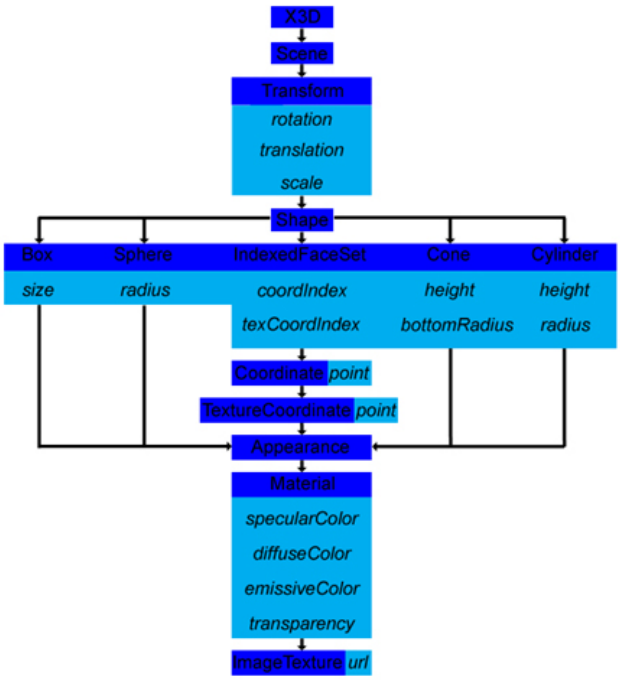


Abbildung 7.1: Zu parsende Knoten und ihre möglichen Attribute

Den Parser muss man für diese genannten Knoten und Attribute sensibilisieren. Nur so kann man die Knoten finden, auswerten, konvertieren und später grafisch darstellen.

Als Nächstes möchte ich das Zielformat der X3D-Daten - das DirectX-Format - vorstellen.

7.2 X-Format

Das DirectX-Format wurde erstmals in DirectX Version 3 eingeführt. Es trägt die Datei-Endung *.x wenn es für DirectX SDK genutzt wird. Bis heute konnte es sich noch nicht als Standardformat für 3D-Objekte etablieren.

Das Format an sich ist ein architektur- und kontextfreies Dateiformat und basiert hauptsächlich auf einer Ansammlung von Templates. Für meine Diplomarbeit benutze ich hauptsächlich folgende Templates: `Vector`, `MeshFace`, `Mesh`, `MeshNormals`, `MeshTextureCoordinates` und `MeshMaterialList`.

Der übliche Aufbau einer solchen DirectX-Datei mit dem gleichen Würfel, wie im vorherigen Kapitel über X3D ist im Listing 7.2 dargestellt.

```
xof 0302txt 0032

Header
{
  1;
  0;
  1;
}

Frame Frame_Translation
{
    FrameTransformMatrix
  {
    1.0, 0, 0, 0,
    0, 1.0, 0, 0,
    0, 0, 1.0, 0,
    3.0000, 0.0000, -1.0000, 1.0;;
  }

  Frame Frame_Box
  {
    FrameTransformMatrix
    {
      2.3000, 0, 0, 0,
      0, 4.0000, 0, 0,
      0, 0, 5.3000, 0,
      0, 0, 0, 1.0;;
    }

    Mesh_Box  {
      36;
      0.5000; -0.5000; 0.5000;,
      ...
      -0.5000; 0.5000; -0.5000;;

      12;
      3;0,1,2;,
      ...
      3;33,34,35;;

      MeshMaterialList
      {
        1;
        12;
        ...
        0;

        Material Mat_Box
        {
          1.0000;0.0000;0.0000;1.0000;;
          0;
          0.0827;0.0827;0.0827;;
          0.0000;0.0000;0.0000;;
        }
      }

      MeshNormals
      {
        36;
        0.0000; -1.0000; 0.0000;,
        ...
        -1.0000; 0.0000; 0.0000;;
```

```
        12;
        3;0,1,2;,
        ...
        3;33,34,35;;
      }
    }
  }
}
```

Listing 7.2: Typische DirectX-Datei

Man sollte hier beachten, dass es sich trotz der gleichen Szene um eine vielfach längere Datei handelt. Wie man sieht, habe ich die Datei an vielen Stellen durch „...“ gekürzt. Somit kommen wir zum Vergleich.

7.3 Vergleich

Nachdem ich beide Dateiformate vorgestellt habe, möchte ich auf die Gemeinsamkeiten, Unterschiede, Vor- und Nachteile eingehen. Wie schon erwähnt wird eine DirectX-Datei sehr schnell in Form von Dateilänge und -größe aufgeblasen. Die *.x-Datei ist von Laien schwerer zu verfassen, als eine *.x3d-Datei.

Wie man sieht, benutzen beide Dateiformate eine unterschiedliche Syntax. Das Dateiformat *.x3d nutzt die sauber lesbare XML-Syntax. Nur im *.x3dv - Format ähneln sich X3D- und DirectX-Dateiinhalt, wie z.B. bei den geschweiften Klammern.

Im X-Format sind es die Templates, im X3D die Knoten und Attribute, die den Dateiinhalt strukturieren.

Einen großen Vorteil bringt X3D mit: Man kann Grundkörper zum Erstellen einer 3D-Szenen verwenden. Auch in der Direct3D-Umgebung werden Primitives (Quader, Kugel, Teekanne, ...) zur Verfügung gestellt, lassen sich aber in einer *.x-Datei leider nicht definieren. Deshalb muss ich später einen Weg suchen um die Grundkörper als Mesh mit Faces und Vertices im DirectX-Format zu beschreiben. Durch Beschreiben der Grundkörper in Faces und Vertices wächst natürlich auch die Dateilänge enorm.

Ebenfalls fällt auf, dass im X-Format die Anzahl der Faces und der Vertices mit angegeben wird. Für eine erfolgreiche Konvertierung muss man also einen Zähler schreiben, der Polygon- und Vertices-Anzahl für DirectX festhält.

Im X3D-Format ist es nicht möglich einem `Shape`, das in DirectX einem `MeshTemplate` entspricht, mehrere Materialien zu zuweisen. Im X-Format wäre es möglich, jedes Face in einer anderen Farbe oder Textur darzustellen.

Interessant sind auch die weiteren Materialangaben. Während X3D von Transpa-

renz spricht, wird bei Direct3D mit der Opazität[1] gearbeitet. Den richtigen Wert für das DirectX-Format erhält man, indem von der Zahl Eins der Transparenz-Wert abgezogen wird.

Ebenfalls sollte man bei der späteren Konvertierung nicht aus den Augen verlieren, dass die beiden Formate unterschiedliche Koordinatensysteme (links- und rechtshändig) benutzen.

Auch die Transformationen von Objekten sind in X3D leichter zu beschreiben. Denn durch Nutzen eines `Transform`-Knotens und dem jeweiligen Attribut `scale`, `rotation`, `translation` werden z.B. nur Winkel und Rotationsachse, oder Skalierung in X-, Y- und Z-Achse beschrieben. Somit sind die Transformationen sofort zu erkennen. Im DirectX-Format werden alle Transformationen in der `FrameTransformMatrix` ausgeführt. Somit ist für einen Laien nicht sofort erkennbar um welche Transformationen es sich handelt. Gleichzeitig wächst bei vielen Transformationen, durch die großen Matrizen, sehr schnell die Dateigröße.

[1] Maß für die Lichtundurchlässigkeit (Trübung) von Materialien

8 Implementierung des X3D-Konverters

8.1 Allgemein

Um die verarbeitenden Methoden zur Darstellung der X3D-Objekte besser zu kapseln und es zu ermöglichen diese später in einem anderen Programmteil wiederzuverwenden, ist es notwendig Klassen zu bilden. Ich möchte deshalb die Methoden des Konverters in zwei Klassen namens `CX3DInterpret` und `CX3DInOutput` zusammenfassen.

Die Klasse `CX3DInOutput` ist für das Laden der X3D-Datei, sowie Erstellen und Schreiben der DirectX-Datei zuständig. Das Konvertieren übernehmen die Methoden der Klasse `CX3DInterpret`. Der Konverter ist somit wie in Abbildung 8.1 (am Ende des Kapitels) aufgebaut.

8.2 Klasse X3DInterpret

Die Klasse `CX3DInterpret` beinhaltet alle wichtigen Methoden zum Transformieren der X3D-Daten in eine für Direct3D verständliche Sprache. Das heißt, dass vor allem Zeichenketten umstrukturiert oder in Zahlen umgewandelt werden. Denn der XML-Parser gibt die Werte der X3D-Knoten nur in Form von Zeichenketten (engl. Strings) zurück. Ebenso müssen die Werte so umgewandelt werden, dass die Konvertierung vom rechtshändigen (X3D) zum linkshändigen Koordinatensystem (D3D) erfolgreich verläuft. Nun stelle ich die Methoden der Klasse `CX3DInterpret` vor.

Methode *MakeStringToFloatVector3*

```
void MakeStringToFloatVector3(String* sValue, floatVector3* VecAdr);
```

Diese Methode wandelt eine Zeichenkette in einen von mir definierten Zahlentripel des Typs *Float* (s. Anhang B) um. Die Zeichenkette muss drei durch ein Leerzeichen getrennte Fließkommazahlen enthalten. Dadurch können in X3D-Knoten angegebene Farben (Rot, Grün, Blau - Tripel), Dimensionen von Quadern (Breite,

Höhe, Tiefe - Tripel) oder die Transformationen Skalierung und Translation ausgewertet werden. Das Tripel muss vor Aufruf der Methode erstellt worden sein. In der Methode selbst wird es nur mit Werten aus der Zeichenkette gefüllt.

Methode *MakeStringToRotVector4*

```
void MakeStringToRotVector4(String* sValue, rotVector4* VecAdr);
```

Auch hier wird eine Zeichenkette zerlegt. Jedoch handelt es sich nicht nur um die Strukturierung in ein Zahlentripel, sondern es wird noch ein vierter Wert aufgenommen. Ein von mir definierter Rotationsvektor (s. Anhang B).

Diese Methode ist speziell für die Verwendung zur Interpretation von Transformationen (X3D-Attribut `rotation`) in Form von Rotationen. Das Tripel nimmt somit die Rotationsachsen auf, der vierte Wert den Rotationswinkel θ in Radient. Auch hier muss vor Aufruf der Methode, der Rotationsvektor erstellt werden.

Methode *FillCoordIndex*

```
int FillCoordIndex(String* sValue, intVector3** ArrayAdr);
```

Die Methode `FillCoordIndex` liest eine Zeichenkette ein, in der die Beschreibungen der einzelnen Faces enthalten sind. Die Faces dürfen in diesem Fall nicht mehr als drei Eckpunkte besitzen. Dies ist darauf zurückzuführen, dass Direct3D, wie schon erwähnt, nur mit Dreiecken und nicht mit Vielecken arbeiten kann. Eine Möglichkeit wäre die Methode `FillCoordIndex` durch eine weitere Methode zur Triangulation (Zerlegung von Vierecken in Dreiecke) in Zukunft zu erweitern (s. 10.3).

Die Punkte in der Zeichenkette müssen durch Kommas getrennt angegeben werden. Die Beschreibung der Dreiecke wird in einen von mir definierten aus Tripel bestehenden Array abgespeichert. Da die Beschreibung der Dreiecke nur in ganzen Zahlen geschieht, kann auch das Tripel immer nur ganze Zahlen annehmen.

In der Methode wird der dritte mit dem ersten Punkt vertauscht. Dies ist damit begründet, dass das mit dem rechtshändigen Koordinatensystem arbeitenden X3D, die Dreiecke nach dem Uhrzeigersinn aufbaut. Man benötigt aber eine Beschreibung, die die Dreiecke gegen den Uhrzeigersinn aufbauend beschreibt.

Während des Zerlegens der Zeichenkette werden die bisher verarbeiteten Dreiecke gezählt. Damit erhält man am Ende der Methode als Rückgabewert die Anzahl der Dreiecke des aktuellen Drahtgittermodells. Den Wert der Dreieckszählung werde ich im weiteren Verlauf auch als *Polycount* bezeichnen.

Die Methode wird zur Interpretation der Attribute `coordIndex` sowie `texCoordIndex` des X3D-Knotens `IndexedFaceSet` verwendet.

Methode *FillPoint*

```
int FillPoint(String* sValue, floatVector3** ArrayAdr);
```

Das Attribut `point` im X3D-Knoten `Coordinate` beinhaltet die Koordinaten der Vertices des Drahtgittermodells. Um auch diese richtig zu verarbeiten, steht die Methode `FillPoint` zur Verfügung.

Dabei werden wieder die Punkte aus einer Zeichenkette herausgelesen. Die Koordinaten x, y und z sind durch Leerzeichen getrennt. Das Ende eines Koordinaten-Sets für ein Vertex wird durch ein Komma angegeben.

Die Koordinaten der Punkte werden wiederum in ein von mir definiertes Array aus Tripeln gespeichert. In diesem Fall können die Tripel Fließkommazahlen aufnehmen. Das liegt daran, dass die Koordinaten der Vertices keine ganzen Zahlen sein müssen.

Wenn man die Punkte unverändert in meinem Array speichert und später in Direct3D darstellen würden, könnte man einen seltsamen Effekt erleben. Das 3D-Modell würde spiegelverkehrt dargestellt. Dies liegt wiederum an den unterschiedlichen Koordinatensysteme von X3D und D3D. Damit das Drahtgittermodell jedoch im Viewer korrekt dargestellt wird, invertiert man die z-Koordinate jedes einzelnen Punktes.

Man sollte vor dem Aufruf der Methode bedenken, das Array für die Punkte vorher zu erstellen und nur die Adresse an die Methode zu übergeben. Der Rückgabewert von `FillPoint` ist die Anzahl der Vertices des Drahtgittermodells.

Methode *FillTexPoint*

```
int FillTexPoint(String* sValue, floatVector3** ArrayAdr);
```

Die Methode ist zur Interpretation des Attributs `point` im X3D-Knoten `TextureCoordinate`. Dabei gleicht die Methode der vorher beschriebenen Methode `FillPoint` nur auf den ersten Blick. Der große Unterschied liegt im Detail.

Bei dieser Methode werden die Texturkoordinaten des Drahtgittermodells aus einer Zeichenkette gelesen. Dabei handelt es sich, wie schon im Kapitel 3.6 erwähnt, um 2D-Koordinaten. Die Achsen werden als U und V bezeichnet.

Den Betrag der v-Koordinate muss man von Eins abziehen, da der Koordinatenursprung bei X3D in der oberen linken Ecke definiert ist, während der Ursprung sich

bei X3D und gängigen 3D-Programmen unten links befindet.

Bei dem Rückgabewert der Methode, handelt es sich um die Anzahl der gezählten Koordinatenpunkte der Textur.

Methode *CalculateNormals*

```
void CalculateNormals(int iAnz, floatVector3** PointAdr, intVector3** FaceAdr,
    floatVector3** ArrayAdr);
```

Diese Methode ist, wie der Name schon verrät, zur Berechnung der fehlenden Normalen der Dreiecke des Drahtgittermodells.

Sie benötigt die Adressen der Arrays mit den Vertices-Koordinaten und den Beschreibungen der Triangle. Ebenso muss die Anzahl der Dreiecke des 3D-Modells (Polycount) übergeben werden. Gleichzeitig ist zu beachten, den Array für die Normalen vor dem Aufruf der Methode zu erstellen.

Die Normale jedes Dreiecks wird berechnet, indem jeweils zwei Differenzvektoren vom ersten Vertex zum zweiten Vertex, sowie vom ersten zum dritten Vertex des jeweiligen Dreiecks erstellt werden. Diese beiden Vektoren spannen eine Fläche auf. Zu dieser Fläche steht die gesuchte Normale senkrecht. Mit den beiden Differenzvektoren wird das Skalarprodukt gebildet. Durch Normalisierung des Skalarprodukts erhält man die gesuchte Normale des jeweiligen Dreiecks.

Diese Operation wird an jedem Dreieck des 3D-Modells durchgeführt. Die Normalen werden dann in dem Array abgespeichert.

Die Verbindung der einzelnen Normalen zu den Vertices erfolgt erst später in der Ausgabe.

8.3 Klasse CX3DInOutput

In der Klasse `CX3DInOutput` befinden sich alle Methoden, die wichtig sind für das Laden und Parsen der X3D-Datei, sowie zum Erstellen der DirectX-Datei. Die wichtigsten Methoden werden jetzt kurz erläutert:

Methode *LoadNParseX3DFile*

```
bool LoadNParseX3DFile(String* sPath, bool bIsAusgabe);
```

Diese Methode ist die größte und entscheidendste Methode in dieser Klasse. An sie muss man den Pfad und den Namen der zu konvertierenden X3D-Datei

übergeben. Ebenfalls kann man dem Parameter `bIsAusgabe` den Wert `TRUE` übergeben, um ein Ausgabeprotokoll des Konvertierungsvorgangs anzufordern.

In der Methode selbst wird ein `FileStream` erzeugt, um eine neue, der X3D gleichnamigen Datei im DirectX-Format anzulegen. Mit einem dazugehörigen `Streamwriter` wird das Schreiben in der neuen Datei ermöglicht. Als nächstes werden in der Methode alle wichtigen Variablen und Zeiger initialisiert. Anschließend wird versucht ein `XmlTextReader`-Objekt zu erstellen. Bei diesem Objekt handelt es sich um unseren Parser zum Durchlaufen der X3D-Datei. Da man davon ausgeht, dass die X3D-Datei wohlgeformt ist, wird `XmlResolver` ein Nullwert zugewiesen und damit ein Internet-Zugriff auf DTDs[1] und andere externe URLs unterbunden.

Ist dies geschehen, ist der `XmlTextReader` bereit, die gesamte X3D-Datei durchzumustern. Dabei springt der Parser von einem Knoten zum anderen. Durch Vergleich der Typen und Namen der aktuellen Knoten, ist es möglich die uns wichtigen Knoten herauszufiltern, diese zu interpretieren und zu konvertieren.

Trifft der Parser auf einen `Transformation`-Knoten, werden dessen Daten sofort konvertiert und in die DirectX-Datei geschrieben.

Wenn der `XmlTextReader` auf einen `Shape`-Knoten stößt, werden die folgenden Daten gesammelt und erst am Ende des `Shape` in die DirectX-Datei ausgegeben. Hat der Parser einen `Shape`-Knoten gefunden, merkt er sich dessen Namen. Nach diesem Knoten können nun Primitives - die Grundkörper in X3D - folgen oder ein komplexes 3D-Modell mit seinen Geometriedaten in `IndexedFaceSet`-,`Coordinate`- und `TextureCoordinate`-Knoten. Vor oder nach den Geometriedaten kann der Parser auf einen `Appearance`-Knoten treffen, der einen `Material`-Knoten und/oder einen `ImageTexture`-Knoten für die farbliche Darstellung des 3D-Objektes, umschließt.

Findet der Parser das Ende eines `Shape`-Knoten, werden die bisher gesammelten Daten in der DirectX-Datei ausgegeben und Variablen zurückgesetzt, sowie Arrays geleert. Somit ist der Parser bereit, weitere `Shape`-Knoten zu verarbeiten.

Ist der Parser am Ende der Datei angelangt, wird die DirectX-Datei geschlossen, alle Ressourcen wieder freigegeben und die Methode `LoadNParseX3DFile` gibt `TRUE` bei erfolgreicher Konvertierung zurück.

[1] Document Type Definition

Methode *InitMaterials*

```
void InitMaterials(strMaterial** MatAdr);
```

Findet der Parser keinen Knoten mit Materialien-Informationen für das 3D-Modell oder enthalten diese Knoten nicht alle Attribute zur richtigen Farbdarstellung, werden diese durch festgelegte Standard-Farbwerte ergänzt. Diese Methode dient dazu, die Materialien-Variablen auf diese Standardwerte zu setzten.

Methode WriteHeader

```
void WriteHeader(StreamWriter* sw, String* sName);
```

Diese Methode schreibt in die neu angelegte DirectX-Datei den spezifischen 16-Byte langen Header. Durch diesen wird die Datei später als DirectX-Datei erkannt.

Der Header beginnt mit den Buchstaben `xof`, gefolgt von der Versionsnummer der Datei-Spezifikation. In meiner Arbeit wird die Version `0302` verwendet. Danach erfolgt mit `txt` der Hinweis auf eine unkomprimierte ASCII-Textdatei. Mit den letzten vier Ziffern wird die Bitbreite unserer Float-Variablen beschrieben. Ich verwende ausschließlich den Typ *Single* - dessen Größe 32 Bit beträgt.

Nach der ersten Zeile folgt das Header-Template. In älteren DirectX-Dateiformaten wurde es genutzt, um die verwendete Version anzugeben. Verschiedene Quellen geben an, dass es nicht mehr notwendig ist dieses Template zu verwenden[2]. Um eine hohe Kompatibilität der erstellten DirectX-Datei zu erreichen, möchte ich dieses Header-Template trotzdem in den Kopf der zu erstellenden Datei einfügen.

Methode WriteObjectTransformMatrix

```
void WriteObjectTransformMatrix(StreamWriter* sw, float fScWidth, float fScHeight,
    float fScDepth, String* sMeshName);
```

Damit die Standard-3D-Primitives (Kugel, Quader, Kegel, Zylinder) in ihre von der X3D-Datei geforderten Dimensionen in der *World Scene* transformiert werden, nutzt man diese Methode. Deshalb auch die Übergabe der drei Skalierungsparameter `fScWidth`, `fScHeight`, `fScDepth`. Die Methode erzeugt eine Transformation-Matrix, die das spätere Mesh umklammert und somit in die richtigen Dimensionen skaliert.

[2] vgl. [3] S. 517-519

Methode WriteTranslationMatrix

```
void WriteTranslationMatrix(StreamWriter* sw, String* sTranslation);
```

Wenn der Parser in X3D ein Attribut `translation` (Verschiebung) in einem `Transform`-Knoten findet, wird diese Methode aufgerufen. In dieser Methode werden die Translation-Parameter ausgelesen und in eine für DirectX verständliche Transformationsmatrix geschrieben. In der Methode wird die Parameterangabe für die Verschiebung auf der z-Koordinate invertiert. So wird der Wechsel vom rechts- zum linkshändigen Koordinatensystem erreicht.

Methode WriteRotationMatrix

```
int  WriteRotationMatrix(StreamWriter* sw, String* sRotation);
```

Eine weitere Methode zum Verarbeiten von Transformationen ist die Methode `WriteRotationMatrix`. Diese ist für den Fall, dass der XML-Parser ein Attribut `rotation` in einem `Transform`-Knoten findet.

Tritt dieser Fall ein, wird eine Transformation-Matrix geschrieben, die das darauf folgende Mesh umschließt und somit um die angegebenen Parameter rotiert. Diese Methode erfasst zu aller erst, um welche Achse die Rotation stattfinden soll. Nach diesem Kriterium wird auch die geeignete Rotationsmatrix ausgewählt.

Für die Rotation nutzt man die Transformationsmatrizen wie in Kapitel 3.2.2. Doch die Matrizen werden nicht 1:1 übernommen, denn auch hier muss der Koordinatensystemwechsel vollzogen werden. Dies erreicht man, indem man bei X- und Y-Achsenrotation die jeweilige Matrix um die Hauptdiagonale spiegelt. Die Z-Achsenrotation bleibt jedoch unangetastet, weil bei dieser Transformation um die Z-Achse rotiert wird und damit die Ausrichtung dieser Achse keine Rolle spielt.

Methode WriteScaleMatrix

```
void WriteScaleMatrix(StreamWriter* sw, String* sScale);
```

Diese Methode stellt bei der Konvertierung von X3D in Direct3D eine Seltenheit dar. Denn die Parameter der Transformation „Skalierung“ müssen nicht speziell konvertiert werden. Der Grund liegt darin, dass bei Skalierung die Ausrichtung der Z-Achse keine Rolle spielt. Deshalb verarbeitet die Methode nur den übergebenen String, der die Skalierungsparameter beinhaltet, und erstellt eine für Direct3D verständliche Skalierungsmatrix.

Methode WriteVertexList

```
void WriteVertexList(StreamWriter* sw, int iPCount, floatVector3** PointAdr,
    intVector3** FaceAdr);
```

Die Methode `WriteVertexList` schreibt den größten Teil des Mesh-Templates der DirectX-Datei. Das beinhaltet die Beschreibung der Polygone, sowie Koordinaten der einzelnen Vertices.

Beim Schreiben der *.x-Datei muss man - im Gegensatz zu X3D - verhindern, dass sich angrenzende Polygone einen Vertex teilen. Dieser Schritt muss erfolgen, da ein Vertex in der DirectX-Datei nur eine Textur-Koordinate und eine Normale besitzen darf. Dies wird beim „einwickeln“ des 3D-Objekts durch eine der in Kapitel 3.6 genannten Mapping-Verfahren mit einer Textur eingehalten. Beim Texturieren eines komplexeren Objektes kann es passieren, dass angrenzende Polygone jeweils andere Texturbereiche nutzen. Somit würden von den Polygonen gemeinsam genutzte Vertices mehr als eine zugelassene Texturkoordinate beinhalten. Dies würde zu Texturierungsfehlern führen. Die Abbildung 8.2 soll dieses Problem verdeutlichen.

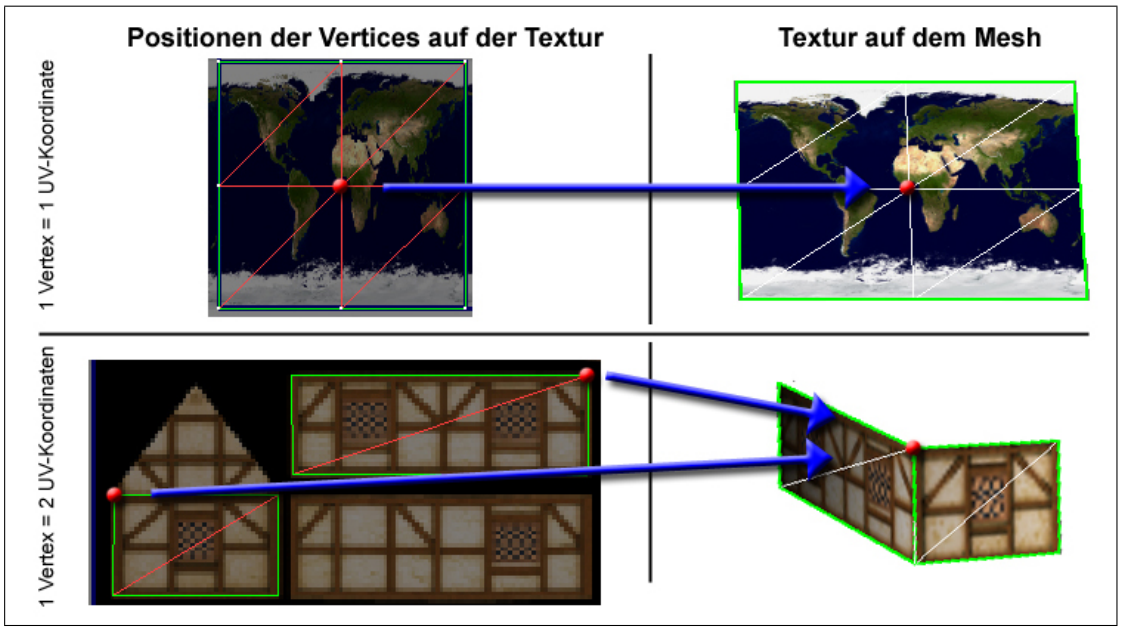


Abbildung 8.2: Vertices mit gleichen und unterschiedlichen uv-Koordinaten

Das Unterbinden von gemeinsam genutzten Vertices ist ebenfalls notwendig für die spätere Normalenberechnung. Bei ihr erhält jedes Polygon und somit auch jeder Vertex des Polygons, seine eigene Normale. Dadurch entsteht später in der Renderumgebung bei runden Objekte der Flat Shading - Effekt. Würde jedoch die

Interpolation der Normalen gemeinsam genutzter Vertices durchgeführt, würde z.B. ein Würfel ein nicht gewolltes geglättetes Aussehen erhalten (Abb. 8.3).

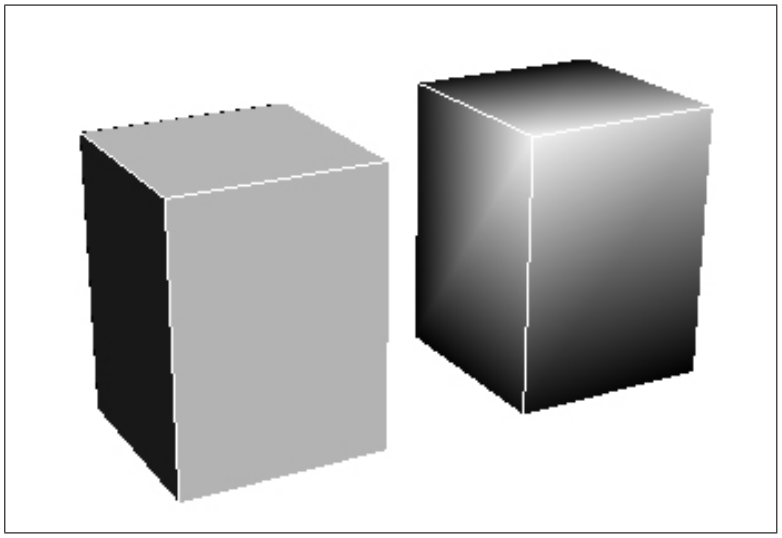


Abbildung 8.3: Triangles ohne (links) und mit (rechts) gemeinsam genutzten Vertices und Normalen

Aus den beiden genannten Gründen gilt die Einschränkung, dass die Polygone eines Mesh keine gemeinsamen Vertices aufweisen dürfen.

Die Methode realisiert diesen Weg, indem sie das von jedem Polygon verwendete Vertex als eigenständiges Vertex in die DirectX-Datei schreibt. Die Anzahl der Vertices beträgt nun das Dreifache des PolyCount (Jedes Triangle mit seinen drei eigenen Vertices).

Die Vertices werden neu nummeriert. So werden den drei Vertices des n-ten Triangles ($(n = (0...(PolyCount-1))$) die Nummern $(n*3)$, $((n*3)+1)$, $((n*3)+2)$ zugewiesen.

Methode WriteTexCoords

```
void WriteTexCoords(StreamWriter* sw, int iPCount, floatVector3** PointAdr,
    intVector3** FaceAdr);
```

Nachdem die Vertices neu angeordnet wurden, folgt nun die Methode zur Neuordnung und Ausgabe der Texturkoordinaten. Dabei durchläuft die Methode abermals alle Polygone und deren Vertices und schreibt in die DirectX-Datei die den Vertices zugehörigen Texturkoordinaten.

Methode WriteMaterials

```
void WriteMaterials(StreamWriter* sw, int iPCount, strMaterial** MatAdr, String*
    sMatName, String* sTexture, bool bTexture);
```

Mit dieser Methode wird das `MeshMaterialList`-Template für das aktuelle Mesh in der DirectX-Zieldatei erstellt. In dem Template werden die vom Parser gesammelten und in dem Material-Array abgelegten Materialien-Informationen geschrieben.

Methode WriteNormals

```
void WriteNormals(StreamWriter* sw, int iPCount, floatVector3** NormalsAdr);
```

Diese Methode ist wichtig für die spätere Darstellung von beleuchteten 3D-Objekten. In der Methode wird das `meshNormals` Template mit den dazugehörigen Daten der Normalen in der DirectX-Datei erstellt.

Methode WriteFoot

```
void WriteFoot(StreamWriter* sw);
```

Mit dieser Methode wird das Schreiben der DirectX-Datei abgeschlossen. Es wird sichergestellt, dass alle geschweiften Klammern (die, die einzelnen Templates abschließen) geschlossen werden. Nur dies ermöglicht die Erzeugung einer korrekten X-Datei.

Methode CreateSphere

Die folgenden Methoden beschäftigen sich mit den in X3D möglichen Primitives - den Grundkörpern.

```
void CreateSphere(String* sSphereRadius);
```

Die erste Methode behandelt den Grundkörper einer Kugel. Trifft der Parser auf einen Knoten `Sphere`, wird deren Attribut `radius` ausgelesen (falls vorhanden) und die Methode aufgerufen. Diese regelt, dass das Primitive in ein Mesh mit Polygonen, Vertices und Texturkoordinaten umgewandelt und somit für Direct3D verständlich wird.

Die Erstellung des Meshs erfolgt, indem den Methoden `FillCoordIndex`, `FillPoint`, `FillTexPoint` der Klasse `CX3DInterpret` Werte übergeben werden,

die man von einer als 3D-Drahtgittermodell definierten Kugel erwarten würde. Bei den Werten handelt es sich um Zeichenketten in XML-Syntax, die eine Kugel mit 12 Vertikalen, 6 Horizontalen und einem Radius von einer Einheit beschreibt. Als Texturkoordinaten werden Koordinaten zugewiesen, die dem *Spherical Mapping* mit einer Textur entsprechen.

Die der Methode übergebene Zeichenkette `sSphereRadius` enthält den Wert des Attributs `radius` des `Sphere`-Knotens. Dieser Wert kann möglicherweise NULL oder der vom X3D-Datei-Schreiber angegebene Radius der Kugel sein. Dieser String wird in eine Fließkommazahl konvertiert und der Wert in den Variablen `dMeshHeight`, `dMeshWidth`, `dMeshDepth` gespeichert. Alle drei Variablen erhalten den gleichen Wert, da es sich um einen Kugel handelt, die nur über den Radius definiert wird. Diese Variablen werden im späteren Verlauf vom Parser mit der oben angesprochenen Methode `WriteObjectTransformMatrix` genutzt, um die Standard-Kugel auf die gewünschte Dimension zu transformieren.

Um eine Kugel in eine abgeflachte Kugel zu skalieren, müsste der X3D-Datei-Verfasser einen `Transform`-Knoten mit dem Attribut `scale`, vor dem `Shape`-Knoten in der sich die Kugel befindet, erstellen.

Methode CreateBox

```
void CreateBox(String* sBoxSize);
```

Diese Methode behandelt das Primitive „Box“ - den Quader. Im Grundaufbau gleicht sie der vorangegangen Methode.

Der Unterschied besteht darin, dass die oben genannten Methoden der Klasse `CX3DInterpret` mit Zeichenketten eines Standard-Würfels „gefüttert“ werden. Es handelt sich dabei um einen Würfel mit der Kantenlänge Eins und ohne horizontalen oder vertikalen Unterteilungen. Für das richtige *Texture Mapping* werden uv-Koordinaten übergeben, die dem *Box Mapping* mit einer Textur entsprechen. Das heißt, falls der Parser später eine Textur findet, wird diese auf die sechs Seiten des Würfels projiziert.

Die der Methode übergebenen Variable `sBoxSize` enthält eine Zeichenkette mit den vom X3D-Datei-Verfasser gewünschten Dimensionen (Höhe, Breite, Tiefe) des Quaders. Diese werden wie gehabt verarbeitet und in die schon bekannten drei Variablen `dMeshHeight`, `dMeshWidth`, `dMeshDepth` gespeichert. Falls die Variable `sBoxSize` den Wert NULL aufweist, wird allen Kanten des Quaders eine Länge von zwei Einheiten zugewiesen.

Methode CreateCone

```
void CreateCone(String* sBottomRadius, String* sHeight);
```

Mit der Methode `CreateCone` wird der Primitive „Cone“ - der Kegel - in ein 3D-Objekt aus Vertices und Polygonen umgewandelt.

Der Standard-Kegel besitzt 16 Seiten und der Radius seines Bodens, sowie die Höhe des Kegels, beträgt eine Einheit. Die zu übergebenen uv-Koordinaten entsprechen in dieser Methode dem *Cylindrical Mapping*.

Die beiden Zeichenketten beinhalten die vom X3D-Datei-Verfasser gewünschten Dimensionen des Kegels. Die Dimensionen werden durch Kegelhöhe (`sHeight`) und Radius des Kegelbodens (`sBottomRadius`) definiert. Falls diese beiden Zeichenketten NULL sind, wird für die Höhe der Wert von zwei Einheiten angenommen. Der Radius des Kegelbodens bleibt bei Eins. Befinden sich in den Zeichenketten, jedoch Werte, werden sie in Fließkommazahlen konvertiert und in den schon durch die vorangegangenen Methode bekannten drei Variablen `dMeshHeight`, `dMeshWidth` und `dMeshDepth` gespeichert.

Methode CreateCylinder

```
void CreateCylinder(String* sCylRadius, String* sCylHeight);
```

Das Primitive „Cylinder“ wird schlussendlich mit der Methode `CreateCylinder` zu einem Mesh mit Vertices und Polygonen umgewandelt.

Dabei werden wieder Vertex-, Texturkoordinaten, sowie die Polygon- Beschreibungen eines Standard-Zylinders benutzt. Der Zylinder besitzt wie der Kegel zwölf Seiten, hat keine horizontalen Unterteilungen und sein Radius, sowie die Höhe beträgt eine Einheit. Die uv-Koordinaten entsprechen, wie beim Kegel, dem *Cylindrical Mapping*.

Die beiden der Methode zugewiesenen Zeichenketten `sCylRadius` und `sCylHeight` werden, falls sie nicht NULL sind, in Fließkommazahlen umgewandelt. Anschließend werden sie an die drei Variablen `dMeshHeight`, `dMeshWidth` und `dMeshDepth` für die finale Dimension des Zylinders übergeben. Wenn die Zeichenketten leer sind, wird an die Variablen ein Radius von einer Einheit (Breite und Tiefe sind somit zwei Einheiten groß), sowie eine Höhe von zwei Einheiten angegeben.

Methode GetMatDiffuse

Nachdem ich meine Methoden für die Verarbeitung der Primitives vorgestellt habe, möchte ich zu den Methoden für Verarbeitung der Materialien und Texturen kommen.

```
void GetMatDiffuse(String* sDiffuseColor);
```

Die erste Methode namens `GetMatDiffuse` ist dafür zuständig, die Zeichenkette mit den Werten der *Diffuse Color* (dt. Streufarbe) in drei Fließkommazahlen für die Grundfarben Rot, Grün und Blau zu zerlegen. Die RGB-Werte werden in ein von mir erstelltes Array für die Materialien (s. Anhang B) unter der Eigenschaft *DiffuseColor* abgelegt.

In der DirectX-Datei wird dieser Wert später als *FaceColor* verwendet.

Methode GetMatEmissive

```
void GetMatEmissive(String* sEmissiveColor);
```

Mit dieser Methode wird die übergebene Zeichenkette in die RGB-Komponenten für die *Emissive Color* (dt. Ausstrahlungsfarbe) konvertiert und in den Material-Array unter der Eigenschaft *EmissiveColor* abgelegt.

Methode GetMatSpecular

```
void GetMatSpecular(String* sSpecularColor);
```

Bei dieser Methode wird die Konvertierung der übergebenen Zeichenkette in die RGB-Komponenten für die *Specular Color* (dt. Glanzfarbe) durchgeführt. Die drei dabei erzeugten Fließkommazahlen werden in dem Material-Array unter der Eigenschaft *SpecularColor* abgelegt.

Methode GetMatTransparency

```
void GetMatTransparency(String* sTransparency);
```

Damit einfache Materialien z.B. Flüssigkeiten, Glasscheiben oder Folien dargestellt werden können, wird für Direct3D ein Alpha-Wert (auch Opazitätswert genannt) benötigt. Der X3D-Standard kann jedoch nur einen Transparenzwert bereitstellen.

Die Methode `GetMatTransparency` ermöglicht, den als Zeichenkette übergebenen Transparenzwert in einen Alphawert umzuwandeln. Den Alphawert wird ermittelt, wie in 7.3 erwähnt, indem von der Zahl Eins der Transparenzwert abgezogen wird. Unter der Eigenschaft *transparency* im Material-Array wird der Alphawert abgelegt.

Methode GetMatTexture

```
void GetMatTexture();
```

Diese Methode namens `GetMatTexture` ist zuständig, Texturpfadangaben in eine für Direct3D verständliche Form zu konvertieren. Die Ordner-Ebenen in den Pfadangaben müssen nicht wie üblich durch ein Backslash, sondern durch ein doppeltes Backslash getrennt werden. Der Grund liegt darin, dass ein einzelner Backslash in der Direct3D-Umgebung als Escape-Sequenz gedeutet wird.

Die globale Variable `sImageTexture` des Zeiger-Typs String* zeigt auf den Texturpfad des aktuell zu bearbeitenden `Shape`s. Um einen doppelten Backslash zu erreichen, fügt diese Methode jedem gefunden Backslash einfach einen zweiten hinzu. Dadurch kann später das Direct3D-Objekt in der Renderumgebung auch jede Textur finden.

Methode SetBack

```
void SetBack();
```

Diese Methode kümmert sich darum, die Matrizen mit den Face-Beschreibungen, den Textur- und Vertex-Koordinaten, sowie den Materialien zurückzusetzen und zu leeren.

Sie wird aufgerufen, wenn ein `Shape`-Knoten beendet wird und deren Inhalte konvertiert und in die aktuelle DirectX-Datei geschrieben wurde. Somit ist der Konverter wieder initialisiert und bereit, den nächsten `Shape`-Knoten zu verarbeiten.

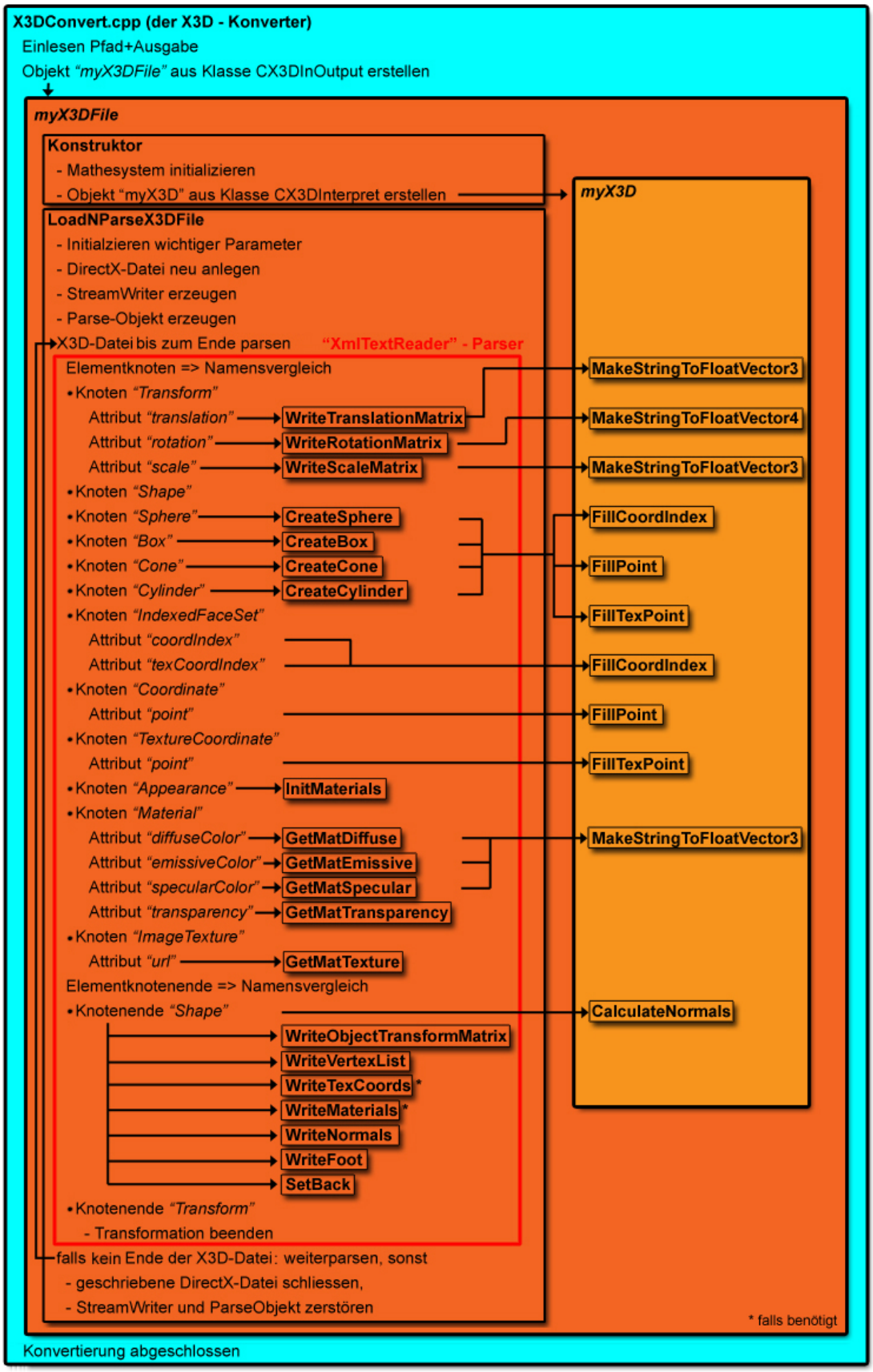


Abbildung 8.1: Aufbau und Ablauf des Konverters

9 Implementierung der Renderumgebung

9.1 Allgemein

Wie bereits in 6.4 erwähnt, ist die Konvertierung von X3D-Dateien und die Renderumgebung getrennt. Dafür gibt es noch weitere Gründe.

Erstens ist es durch diese scharfe Trennung möglich, den Konverter zu bearbeiten und zu erweitern ohne dabei die Struktur der Renderumgebung zu verändern. Umgekehrt kann die Renderumgebung weiterentwickelt werden, ohne sich mit der Programmierung des Konverters auseinander setzen zu müssen.

Ein weiterer Grund der Trennung ist, dass Konverter und Renderumgebung verschiedene Konzepte nutzen. Der Konverter arbeitet verstärkt mit verwaltetem Code des .NET-Konzeptes. Dabei werden die erstellten Objekte und Arrays vom System verwaltet. Der Programmierer braucht sich nicht um die Speicherverwaltung zu kümmern. So werden z.B. nicht mehr benötigte Objekte vom sogenannten *Garbage Collector* automatisch zerstört und der vorher belegte Speicherplatz wieder freigegeben. Der in dieser Arbeit benutzte Parser `XmlTextReader` kann nur in diesem verwalteten Code verwendet werden.

Die Renderumgebung benutzt keinen verwalteten Code, da dieses Konzept nicht auf das DirectX SDK und MFC übertragbar ist. Ich habe mich für die Programmierung der Renderumgebung zur Darstellung der X3D-Objekte in DirectX mit MFC[1] entschieden, weil diese Klasse eine Menge nützlicher C++ Klassen zur Verfügung stellt. Diese vereinfachen vor allem die Interaktion zwischen dem Viewer und dem Betriebssystem. So beinhaltet MFC zum Beispiel vorgefertigte Funktionalitäten für Fenster, Buttons und Menüs. Dadurch nimmt mir die Klasse einen Großteil der Erstellung von Windows-Anwendungen ab und spart Programmierarbeit. Durch MFC ist es möglich, die Funktionalität der Renderumgebung auszubauen und die Oberfläche benutzerfreundlich zu gestalten.

Im Folgenden möchte ich nicht jede einzelne Funktion der Renderumgebung erklären, sondern nur auf die komplexeren Probleme zur Realisierung der Features des Viewers eingehen. Der Quellcode der Features selbst befindet sich in Header- und C++-Datei `Direct3DMFCView` des Projekts `Direct3DMFC`.

[1] Microsoft Foundation Class

9.2 Öffnen von X3D-Dateien

Damit der Benutzer, verschiedene Dateien in die Renderumgebung laden kann, wird beim „Öffnen“ ein Objekt der Klasse `OpenFileDialog` erzeugt, in dem DirectX- oder X3D-Dateien geöffnet werden können. Ist eine Datei ausgewählt, wird geprüft, ob es sich um eine Datei mit der Endung "*.x“ oder „*.x3d“ handelt. Erkennt die Funktion eine X-Datei, wird diese ohne weiteres dargestellt. Ist jedoch eine X3D-Datei ausgewählt, wird der X3D-Konverter *X3DConvert* mit der Klasse *CX3DInterpret* durch einen Prozess aufgerufen und ausgeführt. So wird im Hintergrund des Viewers aus der X3D-Datei eine X-Datei gleichen Namens erstellt.

Nach der Abschluss der Konvertierung der ausgewählten X3D-Datei und Beendigung des Systemprozesses (in dem der Konverter ausgeführt wurde), öffnet die Renderumgebung die neu erstellte DirectX-Datei mit dem konvertierten X3D-Inhalt und stellt diese grafisch dar.

9.3 Objektdarstellungen

Im weiteren Verlauf, trifft man öfter auf die Methode `setRenderState`. Deshalb wird hier kurz auf diese eingegangen. Die Methode ist zur Steuerung des Renderprozesses. Dabei bestimmt der erste Parameter, welche Eigenschaft im Renderprozess geändert wird. Der zweite Parameter ist der neue Wert, der eingestellt werden soll.

Somit ist es dem Benutzer möglich, die Polygone in verschiedene Richtungen zu rendern, indem beim `setRenderState` die Eigenschaft `D3DRS_CULLMODE` verändert wird. Mögliche Werte sind:

- Clockwise `D3DCULL_CW` (im Uhrzeigersinn rendern)
- Counterclockwise `D3DCULL_CCW` (gegen den Uhrzeigersinn rendern)
- Keine Richtungsangabe `D3DCULL_NONE` (doppelseitig rendern)

Es soll dem Benutzer ebenso möglich sein, das Objekt in verschiedenen Darstellungen zu präsentieren. Dafür werden in `setRenderState` dem `D3DRS_FILLMODE` verschiedene Werte zugewiesen. `D3DFILL_WIRESOLID` stellt z.B. das Objekt mit gefüllten Polygonen dar. Ist der Wert `D3DFILL_WIREFRAME` gesetzt, wird das Objekt in Drahtgitterdarstellung visualisiert.

9.4 Beleuchtung der 3D-Szene

Dem Benutzer der Renderumgebung soll es möglich sein, mit verschiedenen Lichtarten die Szene zu beleuchten.

So kann er, ein Umgebungslicht (engl.: `AmbientLight`) an- und ausschalten. Dafür wird via `SetRenderState` dem Parameter `D3DRS_AMBIENT` ein heller D3D-Farbwert zugewiesen. Die Berechnung dieses Lichts verbraucht sehr wenig Ressourcen und wirkt sich deshalb nicht negativ auf die Framerate aus. Dies liegt daran, dass wenig Rechenoperationen durchgeführt werden müssen, denn das Umgebungslicht benötigt keine Normalen von dem zu beleuchtenden Objekt. Es taucht einfach das gesamte Objekt in eine einheitliche Helligkeit und Farbe. Texturierte Objekte können durch dieses Licht bereits ansprechend wirken. Nichttexturierte Objekte wirken jedoch flach und eindimensional. Soll das Licht ausgeschaltet werden, löst man dies mit einem kleinen Trick, indem dem Parameter `D3DRS_AMBIENT` die Farbe schwarz zugewiesen wird.

Mit einem Punktlicht (engl.: `PointLight`) kann ebenfalls die Szene ausgeleuchtet werden. Dafür muss eine Variable vom Typ `D3DLight9` angelegt und dessen Parameter `Type` der Wert `D3DLIGHT_POINT` zugewiesen werden. Damit das Punktlicht im Viewer bei der Kamera positioniert ist, wird dem Lichtparameter `Position` die Kamerakoordinaten zugewiesen. Bei der Rotation der Kamera um das Objekt, dreht sich somit auch das Punktlicht mit.

```
m_lpD3DDevice->SetLight(0,&Light);
m_lpD3DDevice->LightEnable(0, TRUE);
```

Damit das Punktlicht auch angezeigt wird, muss man dem Licht eine Nummer für die Szenenbeleuchtung zuweisen (hier 0) und `LightEnable` auf `TRUE` setzen. Auch im Renderprozess muss mit `SetRenderState` der Parameter `D3DRS_LIGHTING` auf `TRUE` gestellt werden.

Dieses Licht hat im Gegensatz zum Umgebungslicht einen Nachteil: Zur Darstellung des Lichts werden die Normalen des zu beleuchtenden Objekts benötigt. Sind keine Normalen vorhanden, wird das Objekt überhaupt nicht angezeigt. Findet der Renderer Normalen für das beleuchtende Objekt, zeigt diese Lichtart seinen Vorteil. Die ausgeleuchtete Szene wirkt nun dreidimensional und wesentlich lebendiger als mit dem Umgebungslicht. Soll auch dieses Punktlicht ausgeschalten werden, weist man auch hier dem Licht die Farbe schwarz zu.

9.5 Texturen und Alpha-Kanal

Damit Objekte nicht eintönig wirken, tragen viele 3D-Objekte Texturen. Die Texturen beinhalten oft auch einen Alpha-Kanal, um bestimmte Bereiche der Textur transparent darzustellen. Deshalb soll hier auf die Darstellung von Texturen in D3D eingegangen werden.

Wenn sich in der Renderumgebung ein komplexes Objekt befindet oder verschieden Texturen und Materialien beinhaltet, wird es in *Subsets* unterteilt. *Subsets* sind verschiedene Teilobjekte des Meshs, die unterschiedliche Texturen und Materialien besitzen und auch oft nicht durch Edges oder Vertices verbunden sind. Diese Subsets werden von der Renderumgebung einzeln gerendert. Durch Rendern aller Subsets ergibt sich erst das Gesamtbild des 3D-Objektes.

Die Darstellung von Materialien, Texturen und Transparenz wird erreicht, indem für jedes Subset der Direct3D-Device folgende Parameter zugewiesen werden:

```
m_lpD3DDevice->SetRenderState(D3DRS_ALPHAREF, (DWORD)0x00000001);
```

Der Parameter `D3DRS_ALPHAREF` erhält ein Farbreferenzwert für das sogenannte *Alpha Testing*. Alpha Testing ist eine Funktion, um den Direct3D-Renderer anzuweisen jedes Pixel des Objekts auf einen Alpha-Wert zu testen. Ist kein Alpha-Wert vorhanden, wird das Pixel dargestellt, ansonsten wird die Darstellung des Pixel weggelassen.

Abbildung 9.1: Links: Ohne Alpha Testing / Rechts: Mit Alpha Testing

Alpha Testing ist insbesondere in dem Fall wichtig, wenn hinter einem Face das eine mit Alpha-Wert ausgestattete Textur besitzt, sich ein anderes Face befindet, das durch den Alpha-Wert nun sichtbar sein sollte. Ist Alpha Testing nämlich nicht eingeschaltet, würde das dahinter liegende Face nicht dargestellt werden. Es würde durch den Alpha-Wert des darüber liegenden Face ebenfalls ausgeblendet werden (s. Abb. 9.1).

```
m_lpD3DDevice->SetRenderState(D3DRS_ALPHATESTENABLE, TRUE);
m_lpD3DDevice->SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL);
```

Hier wird der Direct3D-Renderer angewiesen, dass Alpha Testing zu starten. Der Wert des zweiten Parameters gibt die Funktion für den Alpha-Test an. Der Parameter für den Vergleich ist `D3DCMP_GREATEREQUAL`, d.h. es wird von den beiden zu vergleichenden Pixeln nur das Pixel dargestellt, dass den gleichen oder größeren Wert aufweist (engl.: greater or equal).

```
m_lpD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
```

Mit dieser Parameterzuweisung wird das Alpha-Blending zugeschaltet. Durch *Alpha-Blending* werden Mischfarben aus der Materialfarbe des Texels des transparenten Objekts und der Farbe des Hintergrunds erzeugt. Man muss darauf achten, dass die weiter entfernten Objekte zuerst gerendert werden, ansonsten funktioniert die Farbberechnung mit Alpha-Blending nicht.

```
m_lpD3DDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
m_lpD3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
```

Die Texelmaterialfarbe des transparenten Objekts ist die *Source = SRC*, die Farbe des Hintergrunds die *Destination = DEST*.

Für die Konstante `D3DBLEND_SRCALPHA` wird der $Alphawert$ bestimmt. Bei der Konstante `D3DBLEND_INVSRCALPHA` hingegen gilt $1 - Alphawert$.

Dadurch ergibt sich für die Mischfarbe:

$$Mischfarbe = Texelmaterialfarbe * Alphawert + Hintergrundfarbe * (1 - Alphawert)$$

```
m_lpD3DDevice->SetMaterial(&g_pMeshMaterials[i]);
m_lpD3DDevice->SetTexture(0, g_pMeshTextures[i]);

g_pMesh->DrawSubset(i);
```

Nachdem die Transparenz der Texturen behandelt wurde, muss dem Subset die behandelten Texturen und Materialien zugewiesen werden. Anschließend kann das Subset gezeichnet werden.

9.6 Steuerung durch den 3D-Raum

Ein sehr wichtiges Feature für einen 3D-Viewer ist, dass der Benutzer das geladene 3D-Objekt von allen Seiten betrachten und auch durch eine Zoom-Funktion an das Objekt heran- oder wegbewegen kann. Die theoretische Realisierung kann wie folgt aussehen:

Bei einer horizontalen Drehung, rotiert das 3D-Objekt um die Y-Achse. Für den Betrachter würde es den Anschein erwecken, dass die Kamera um das 3D-Objekt rotiert. Nur bei der vertikalen Drehung, und beim Zoom wird die Position der Kamera geändert und um den Koordinatenursprung gedreht.

```
D3DXMATRIXA16 matRotY;
D3DXMatrixRotationY(&matRotY, f_angle_alpha);
m_lpD3DDevice->SetTransform(D3DTS_WORLD, &matRotY);
```

Programmtechnisch ist das zu realisieren, indem man in der Render-Pipeline zuerst für das 3D-Objekt eine Welt-Transformation (s. 3.3.2) durchführt und somit eine Transformationsmatrix für die Objektdrehung um die Y-Achse erstellt. Danach wird die Rotationsmatrix als *World Space* definiert.

Der Wert `f_angle_alpha` wird später durch Steuerung der Cursor-Tasten „Links“ und „Rechts“ erhöht bzw. verringert. Damit wäre die „scheinbar“ horizontale Rotation der Kamera um das Objekt realisiert.

Um die Kamera nun auch vertikal um das Objekt zu bewegen, muss die Kamera-Transformation (s. 3.3.3) durchgeführt werden. Dafür werden verschiedene Vektoren für die Kamera definiert.

```
D3DXVECTOR3 vEyePt(f_xAxis, f_yAxis, f_zAxis);
D3DXVECTOR3 vLookatPt(0, 0, 0);
D3DXVECTOR3 vUpVec(0.0f, 1.0f, 0.0f);
D3DXMATRIXA16 matView;
D3DXMatrixLookAtLH(&matView, &vEyePt, &vLookatPt, &vUpVec);
m_lpD3DDevice->SetTransform(D3DTS_VIEW, &matView);
```

Der erste Vektor namens `vEyePt` gibt die Kameraposition an. Da der Benutzer bei dieser Transformation nur den vertikalen Winkel (Winkel β) und die Entfernung zum Objekt (dist) durch die Cursor-Tasten „hoch“ und „runter“ bzw. „X“ und „Y“ steuert, werden durch diese beiden Angaben die 3D-Koordinaten berechnet. Dies erfolgt in einer eigenen Methode namens `CorrectCamera`. Zur Verdeutlichung der Berechnung soll die Abbildung 9.2 dienen.

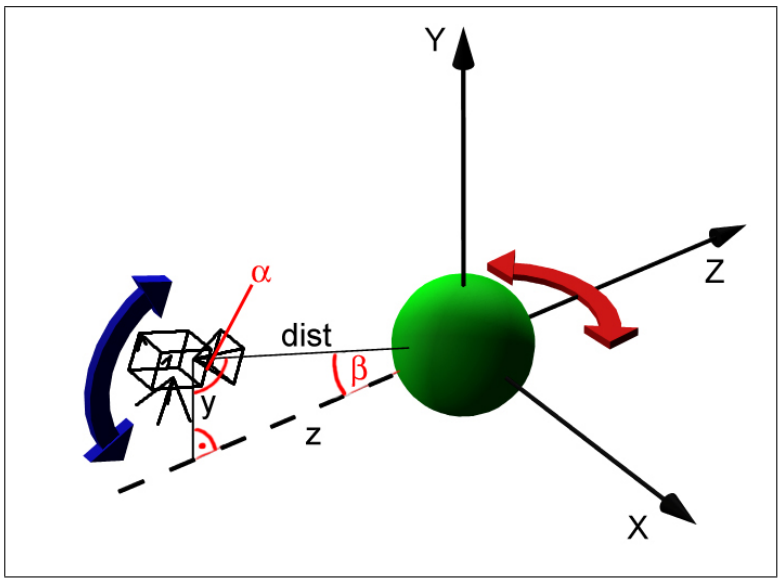


Abbildung 9.2: Rotation der Kamera um 3D-Objekt

Wie zu erkennen ist, bildet die Strecke Kamera bis Koordinatenursprung ($dist$), die Höhe der Kamera über der XZ-Ebene (y), sowie die Strecke z-Koordinate bis Koordinatenursprung ein rechtwinkliges Dreieck. Der fehlende Winkel α errechnet sich durch:

$$\alpha = 180\ Grad - 90\ Grad - \beta$$

Da nun alle Winkel bekannt sind, kann auch die z- und y-Koordinate der Kameraposition ausgerechnet werden:

$$y = dist * \sin(\pi/180\ Grad) * \alpha)$$

$$z = dist * \cos(\pi/180\ Grad) * \alpha)$$

Die x-Koordinate bleibt auf den Wert Null, da die horizontale Drehung schon anderweitig am Anfang dieses Absatzes gelöst wurde.

Der Vektor `vLookatPt` gibt an, wohin die Kamera schauen soll. Da das 3D-Objekt dauerhaft im Koordinatenursprung bleibt, blickt auch die Kamera auf diesen Ursprung. Der letzte Vektor `vUpVec` gibt an, dass die Ausrichtung der Y-Achse für die Kamera als „oben“ definiert wird. Diese drei Vektoren definieren die Kamera-Matrix `matView` die wiederum den *View Space* definiert.

Damit die 3D-Szene auch korrekt auf dem Bildschirm angezeigt wird, ist noch eine Projektions-Transformation (s. 3.3.4) notwendig.

```
D3DXMATRIX matProj;
D3DXMatrixPerspectiveFovLH(&matProj, D3DX_PI * 0.25f, (float)width / (float)height,
     1.0f, 1000.0f);
m_lpD3DDevice->SetTransform(D3DTS_PROJECTION,  &matProj);
```

Für die Projektion erstellt man eine Matrix. Mit der Methode `D3DXMatrixPerspectiveFovLH` wird die Projektion initialisiert. Dabei werden der Zeiger der Projektion-Matrix, die Weite des *FoV* der Kamera (hier 90 Grad), die Größe des 3D-Darstellungsbereichs, sowie die Entfernung von *Near Clipping Plane* und *Far Clipping Plane* übergeben. Am Ende wird dem DirectX-Device diese Transformation zugewiesen.

Die Variation der Parameter für den Winkel der Kamera, die Entfernung der Kamera vom 3D-Objekt, sowie der Rotationswinkel des 3D-Objekts wird, wie oben kurz erwähnt, durch die Cursor-Tasten sowie „X“ und „Y“ für den Zoom ermöglicht.

Die Tasten werden durch die Nachricht `WM_KEYDOWN` ausgewertet. Wird dabei gemeldet, dass eine relevante Taste gedrückt wurde, werden die Werte verändert und bei der vertikalen Drehung die fehlenden y und z-Koordinaten durch die Methode `CorrectCamera` berechnet.

9.7 Berechnung der Framerate

Damit der Benutzer sehen kann, mit wieviel Frames pro Sekunde sein 3D-Objekt dargestellt wird, muss eine Möglichkeit gesucht werden, die Zeit im Programm genau zu messen.

Dafür gibt es drei Möglichkeiten: die Nachricht `WM_TIMER`, die Funktion `timeGetTime` oder durch den *Performence Counter*.

Da auf eine sehr genaue Zeitmessung Wert gelegt wird, fallen die ersten beiden Varianten heraus. Die Nachricht `WM_TIMER` hat in der Verarbeitung eine geringe Priorität. So können ungenaue Messwerte entstehen. Die Funktion `timeGetTime` arbeitet nur bis in den Millisekunden-Bereich. Somit scheint der *Performance Counter* am geeignetsten für eine Berechnung der Framerate zu sein.

Der *Performance Counter* ist ein auf fast jedem System vorhandener Hardware-Baustein der bis 3,19 MHz getaktet werden kann. Das ergibt eine mögliche Genauigkeit von 0,313 Mikrosekunden.[2]

Mit der Funktion `QueryPerformanceFrequency` wird überprüft, ob es möglich ist, auf den *Performence Counter* zu zugreifen. Ist dies möglich, wird ein Zeiger übergeben. Dieser Pointer zeigt auf einen Wert, der die Frequenz des Zählers in Ticks pro Sekunde angibt.

Durch die Funktion `QueryPerformanceCounter` wird der eigentlichen Wert des Zählers angefordert. Bei dem Rückgabewert der Funktion handelt es sich um eine

[2] vgl. [3] S.396/397

fortlaufende Zahl der Ticks. Um die Framerate zu berechnen, wird der Counter mit dieser Funktion initialisiert und der aktuelle Tickwert gespeichert.

Gleichzeitig wird der Framezähler, der die Anzahl der dargestellten Frames speichert, auf Null gesetzt. Nach jedem gezeichneten Frame, das Direct3D gezeichnet hat, wird der aktuelle Tickwert abgefragt.

Ist die Differenz zwischen dem gerade neu aufgenommenen Tickwert und dem Tickwert am Anfang kleiner als die Anzahl der Frequenz (die Ticks pro Sekunde), dann ist noch keine Sekunde vergangen. Somit kann der Framezähler um eins erhöhen werden.

Ist eine Sekunde vergangen, ist mit dem Framezähler die Framerate pro Sekunde ermittelt. Dieser Wert wird mittels Direct3DText im Fenster der Renderumgebung ausgegeben.

In dem Viewer wird dauerhaft die Framerate ermittelt, da die Rate ständigen Schwankungen durch komplexe Operationen, wie Drehungen oder Bewegungen des 3D-Objekts, sowie möglichen zusätzlichen Belastungen durch Rechenoperationen im Hintergrund des Programms, unterliegt.

Sollen in Zukunft mit dem Programm auch Animationen dargestellt werden, möchte ich vorschlagen, den Viewer auf eine maximale Framerate zu begrenzen. Eine Begrenzung ist sinnvoll, da die Zeitangabe von Animationen nicht in Sekunden, sondern in Frames angegeben wird. Somit würde ohne Begrenzung der Framerate, die Animation auf einem Rechner mit hoher Rechnerleistung schneller ablaufen, als auf einem Langsameren. Um dies zu vermeiden, sollte man die Framerate auf einen bestimmten Wert begrenzen. Da für eine flüssige Darstellung mindestens 25 Frames pro Sekunde benötigt werden, würde ich eine Begrenzung auf 30 Frames pro Sekunde empfehlen.

10 Fazit und Ausblick

10.1 Allgemein

In meiner Arbeit habe ich eine Variante betrachtet, die X3D-Objekte in der Direct3D-Umgebung darstellt. Bei der Betrachtung zur Konvertierung der X3D-Objekte für Direct3D bin ich jedoch nur auf die grafische Interpretation von X3D eingegangen. Den größten Teil des X3D-Profils „Interchange“ habe ich dabei konvertiert. Die Implementierung der Funktionen für die Konvertierung des gesamten Profils hätte den zeitlichen Rahmen der Diplomarbeit gesprengt.

In einer weiterführenden Arbeit ist es möglich, Funktionen zur Interpretation der übrigen „Interchange“-Knoten zu implementieren. Da mit X3D auch Animationen und Interaktionen mit den Objekten durch die übrigen X3D-Profile beschrieben werden können, wäre es möglich, die Klasse zur Interpretation von diesen Features zu erweitern. So würde der Viewer zu einem noch leistungsstärkeren Tool werden.

Desweiteren möchte ich speziell auf zwei Punkte hinweisen, die die Kompatibilität des Viewers mit komplexeren X3D-Dateien in Zukunft ermöglichen.

10.2 DEF und USE

Im Rahmen meiner Diplomarbeit habe ich die X3D-Attribute `DEF` und `USE` nicht betrachtet.

In diesen Attributen steckt jedoch ein sehr großes Potenzial, um auf effizienter geschriebene X3D-Dateien zuzugreifen und diese darzustellen. Wenn man einem X3D-Knoten das Attribut `DEF` mit einem eindeutigen Namen als Wert hinzufügt, wird der Knoten eindeutig erkennbar. Ein gleichwertiger Knoten kann nun das Attribut `USE` mit dem gleichen Namen als Wert nutzen. Dadurch wird auf den `DEF`-Knoten verwiesen und dessen Werte und Unterknoten werden von dem mit `USE` ausgewiesen Knoten übernommen. Dadurch spart sich der Urheber der X3D-Datei einen großen Programmieraufwand bei ähnlichen und gleichen Elementen.

Dies bietet sich zum Beispiel bei Elementen des 3D-Modells der Mensa Mittweida an. Hier finden sich wiederholt die gleichen Säulen, Fenster, Stufen und Türen. Somit wäre es möglich, ein `Shape` mit `DEF='Fenster'` zu erstellen. In

dieser würden sich die Daten für das 3D-Modell eines Fensters befinden. Die `Shape`s, die ähnliche Fenster beinhalten sollen, erhielten dann einfach das Attribut `USE='Fenster'`. Dadurch würde der Knoten automatisch den Inhalt von dem Referenzknoten übernehmen.

Zur Realisierung des Features müsste der in dieser Diplomarbeit implementierte Konverter um eine Methode erweitert werden. Diese Methode sollte alle Informationen der Knoten (und dessen Unterknoten) in einem Datenpaket speichern, die mit dem Attribut `DEF` versehen sind. Dem Datenpaket müsste der Wert (Name) von `DEF` zugeordnet werden. Falls dann ein Knoten mit einem Attribut `USE` auftritt, wird nur nach dem Datenpaket mit den gleichen Wert (Namen) von `USE` gesucht und dieses Datenpaktet mit seinen Informationen an der aktuellen Position von `USE` wieder ausgegeben.

10.3 Polygon-Triangulation

Wie schon am Anfang dieser Arbeit erwähnt, kann Direct3D nur Dreiecke verarbeiten. Deshalb kommt es bei dem implementierten Viewer zu Komplikationen, wenn in der X3D-Datei ein Polygon mit mehr als drei Vertices beschrieben wird.

Es ist eine Methode notwendig, die alle Polygone, die aus mehr als drei Eckpunkten bestehen, in mehrere Dreiecke zerlegt. Auch die Implementierung eines solchen Systems war im Rahmen dieser Arbeit nicht möglich. So möchte ich hier nun auf drei Lösungsansätze eingehen.

Erster Ansatz

Der einfachste Ansatz für eine Triangulation wäre, beim ersten Vertex (in der Polygon-Beschreibung von X3D `CoordIndex`) zu beginnen. Den ersten Vertex merkt man sich. Dann folgt man dem Polygon-Rand bis zum dritten Vertex. Beim dritten Vertex beendet man das Polygon, indem man das dritte Vertex mit dem ersten Vertex verbindet. Dadurch erhält man das geforderte Dreieck.

Nun beginnt man beim dritten Vertex, merkt sich dieses Vertex wiederum und geht zwei Vertices weiter. Diese bilden wiederum gemeinsam ein Triangle.

Sind keine weiteren Vertices mehr im `CoordIndex` vorhanden, zieht man den vorher gemerkten Ausgangsvertex zum Bilden eines Dreiecks hinzu. Gab es bei der Triangulation mehr als drei Ausgangsvertices, werden auch diese zu Dreiecken verbunden.

Die Abbildung 10.1 soll den Ansatz an einem Sechseck verdeutlichen.

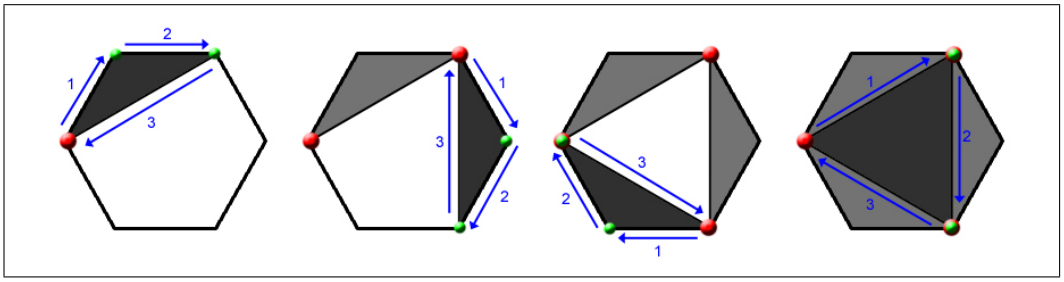


Abbildung 10.1: Erster Ansatz zur Triangulation

Der erste Ansatz ist schnell zu implementieren, wenn die Polygone Vierecke oder Kreise darstellen. Doch bei konkaven Polygonen (z.B.: ein L-förmiges Polygon) zeigen sich bei diesem Ansatz Triangulationsfehler (Abb. 10.2).

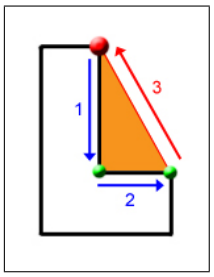


Abbildung 10.2: Triangulations-Fehler

Zweiter Ansatz

Ein zweiter Ansatz ist, dass sogenannte *Museumsproblem* zu nutzen. Es geht von der Idee aus, dass zu triangulierende Polygon sei ein Museum, das durch in Ecken angebrachte Kameras überwacht werden soll. Die Kameras müssen jeden Bereich des Museumsraums überwachen. Das Ziel ist es aber, so wenig wie möglich Kameras zu verwenden

Ein Algorithmus zur Umsetzung ist folgender: Man beginnt wieder bei dem ersten Vertex (das ist die Kameraposition) und bildet ein Dreieck mit dem zweiten und dritten Vertex (sie bilden einen Teil des Kamera-Blickwinkels). Dann versucht man, ein Dreieck mit dem ersten und dem dritten und vierten Vertex zu bilden (wiederum ein Teil des Kamera-Blickwinkels). Dies geht solange, bis das gesamte Polygon trianguliert wurde. Die Abbildung 10.3 verdeutlicht dieses Prinzip. Auch dieses Verfahren führt zu Triangulationsfehlern, wenn es sich um konkave Polygone handelt.

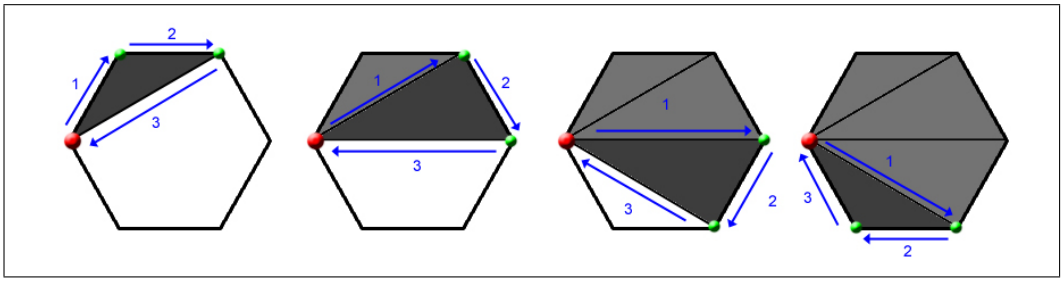


Abbildung 10.3: Zweiter Ansatz zur Triangulation

Dritter Ansatz

Damit es möglich ist, auch konkave Polygone ordnungsgemäß zu triangulieren, bedarf es der Betrachtung eines weiteren Lösungsansatzes. Der Ansatz ist eine Erweiterung des *Museumsproblems* durch eine Kontrollfunktion.

Diese soll kontrollieren, ob die Kamera (der aktuelle Vertex) keine Ecken im Blickfeld hat. Dies wäre der Fall, wenn eine Edge zwischen zwei Vertices gezogen wird und die neue Edge außerhalb des Polygons verläuft oder gar schon vorhandene Edges schneidet. Tritt dieser Fall ein, wird von der aktuellen Kameraposition abgesehen und eine Kamera in der nächsten Ecke (der benachbarte Vertex) platziert und von dieser Position der Triangulierungsprozess fortgesetzt. Die Abbildung 10.4 soll dieses Prinzip verdeutlichen.

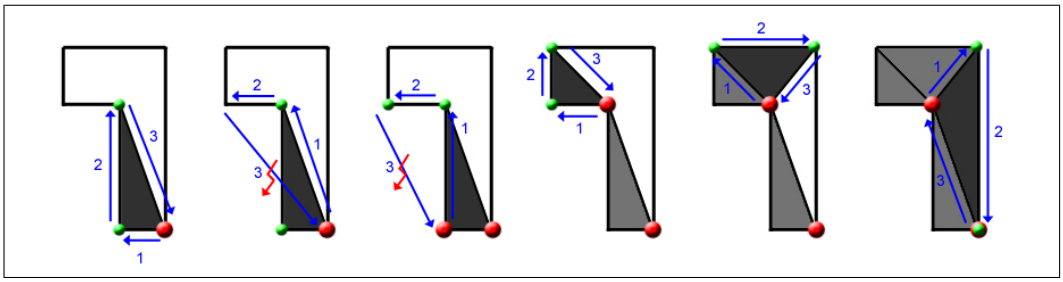


Abbildung 10.4: Dritter Ansatz zur Triangulation

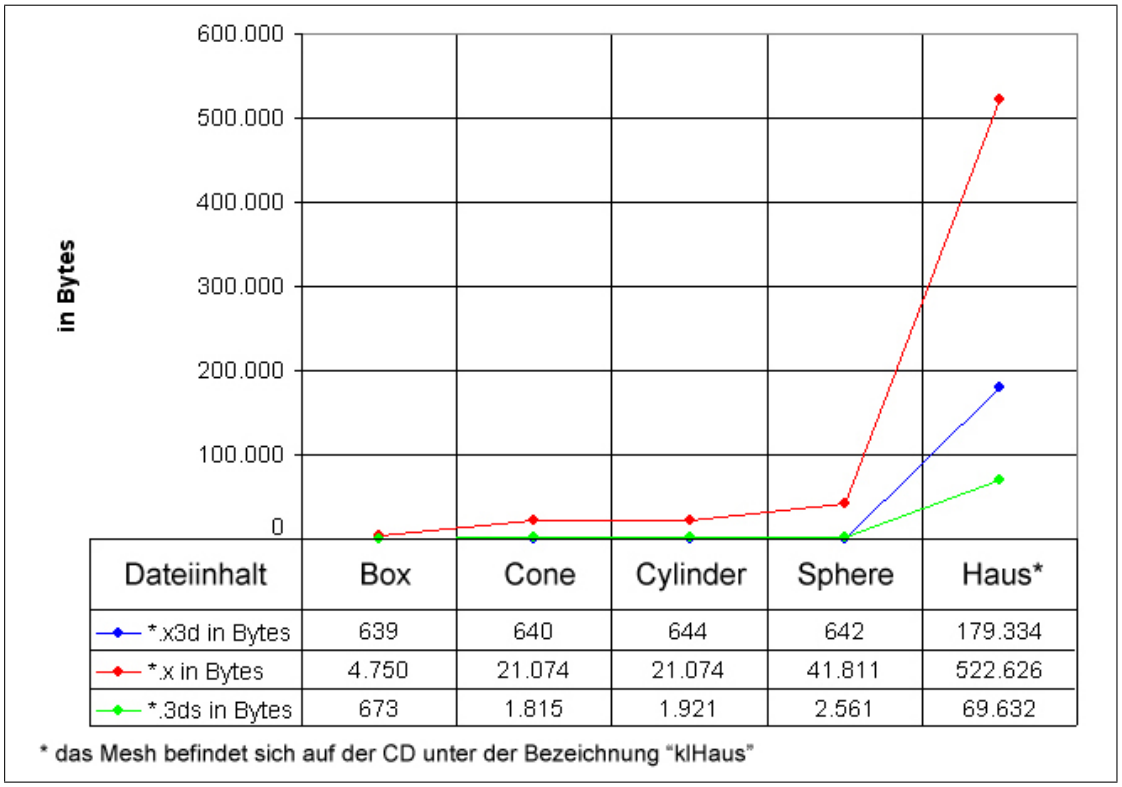
10.4 Größe der Dateien

X3D bietet einen erheblichen Vorteil beim Beschreiben von Szenen mit Primitives und bei der bereits oben angesprochenen Nutzung von DEF und USE.

X3D besitzt mit der XML-Syntax im Gegensatz zu anderen bekannten 3D-Formaten auch eine sehr gute Lesbarkeit und Übersichtlichkeit (auch wenn

XML nicht unbedingt zum Lesen bestimmt ist!).

Ein bedeutender Vorteil ist der sehr geringe Speicherbedarf beim ausschließlichen Benutzen von Primitives in den X3D-Szenen. Der folgende Vergleich soll dies unterstreichen. Dabei wurden vier Primitives und ein „echtes“ 3D-Mesh als X3D-, 3DS- und X-Datei erstellt. Der Graph in Abbildung 10.5 präsentiert das Ergebnis des Vergleichs in Sachen Speicherbedarf.



Dateiinhalt	Box	Cone	Cylinder	Sphere	Haus*
*.x3d in Bytes	639	640	644	642	179.334
*.x in Bytes	4.750	21.074	21.074	41.811	522.626
*.3ds in Bytes	673	1.815	1.921	2.561	69.632

* das Mesh befindet sich auf der CD unter der Bezeichnung “klHaus”

Abbildung 10.5: Vergleich der Dateigröße

X3D zeigt, wie schon vermutet, bei der Verwendung von Primitives seine Stärke. Es übertrifft selbst das renommierte 3D-Modellformat *.3ds. Nur bei komplizierten 3D-Drahtgittermodellen hat X3D gegenüber *.3ds Nachteile. Da im DirectX-Format die Normalen gespeichert und alle 3D-Objekte in Dreiecke zerlegt werden müssen, benötigt es bei weitem mehr Speicherplatz.

Der Vergleich zeigt, dass sich X3D-Dateien durch ihre geringe Größe auch optimal für den Dateitransfer via Internet eignen. Nur beim Darstellen der X3D-Dateien mit dem X3D-Viewer dieser Diplomarbeit, ergibt sich auf dem lokalen Rechner mehr Speicherbedarf (durch die Konvertierung X3D zu DirectX).

10.5 Zukunft von X3D und Fazit

Im Jahr 2004 - zum zehnten Geburtstag von VRML - präsentierten die Erfinder von VRML, Mark Pesce und Tony Parisi, X3D als die Zukunft. Ihr neuer Standard würde das Tor zum Cyberspace aufstoßen. Als die 3D-Beschreibungssprache, „die dort stark sein sollte, wo VRML schwächelte“, sollte sie selbst 3D-Spieleprogrammierer animieren, ihre Spiele mit X3D zu entwickeln. Sie prognostizierten, Millionen Nutzer verwenden in den nächsten Jahren X3D.[1] Nun schreiben wir das Jahr 2009. Abermals ist ein halbes Jahrzehnt vergangen. Doch ein wirklichen Durchbruch von 3D-Inhalten im Web hat es nicht gegeben - zumindest nicht mit X3D. Kein erfolgreiches PC-Spiel ist mit X3D entwickelt worden. Der so erfolgversprechende Standard ist immer noch sehr wenig verbreitet. Viele Firmen verwenden noch immer VRML97 - selbst in den aktuellen Versionen der bekannten 3D-Modellierungsprogramme (TrueSpace, SketchUp, 3dsMax, Maya) lassen sich keine Szenen und Objekte in das X3D-Format exportieren. Auch sie verwenden in den Standardversionen nach wie vor das alte WRL-Format[2][3][4][5].

Als Fazit ist festzuhalten: X3D ist eine der unterbewertesten 3D-Beschreibungssprachen, obwohl es sehr viele Vorteile bietet. Für das Überleben von X3D ist jedoch mehr Support seitens der namenhaften Firmen notwendig. Viel mehr Nutzer von VRML sollten den Schritt von der klassischen VRML-Syntax zur kompatibleren XML-Syntax gehen und damit auch den vollen X3D-Standard akzeptieren.

In meiner Diplomarbeit sehe ich einen Beitrag, durch Verwendung der XML- Syntax und eines XML-Parsers zur Interpretation der X3D-Daten, in diese Richtung zu weisen. Mit der Nutzung von Direct3D zur Darstellung von X3D-Szenen, verstehe ich meine Arbeit auch als Anstoß, die 3D-Beschreibungssprache einem größeren Anwenderkreis zugänglich zu machen.

[1] vgl. http://www.3d-test.com/interviews/mediamachines_2.htm [30.03.2009]
[2] vgl. http://www.caligari.com/products/trueSpace/ts75/Brochure/specification.asp?Cate=BSpecification [30.03.2009]
[3] vgl. http://sketchup.google.com/product/features.html [30.03.2009]
[4] vgl. http://www.autodesk.de/adsk/servlet/index?siteID=403786&id=12354339 [30.03.2009]
[5] vgl. http://www.autodesk.co.uk/adsk/servlet/item?siteID=452932&id=11647061

Literaturverzeichnis

[1] Foley, James; van Dam, Andries; Feiner, Steven; Hughes, John:
Computer Graphics - Principles and Practice. - 2. Aufl. - Chicago: Addison-Wesley, 1995

[2] Brutzman, Don; Daly, Leonhard:
X3D - eXtensible 3D Graphics for Web Authors. - 5. Aufl. - San Francisco: Morgan Kaufmann Publishers, 2007

[3] Rousselle, Christian:
Jetzt lerne ich Spieleprogrammierung mit DirectX. - 8. Aufl. - München: Markt+Technik Verlag, 2006

[4] Rudolph, Alexander:
3D-Effekte für Spieleprogrammierer. - 2. Aufl. - München: Markt+Technik Verlag, 2005

[5] Templeman, Julian; Olsen, Andy:
MS Visual C++.NET Schritt für Schritt. - 1. Aufl. - Unterschleißheim: Microsoft Press Deutschland, 2001

[6] Chapman, Davis:
Visual C++.NET in 21 Tagen - 2. Aufl. - München: Markt+Technik Verlag, 2002

[7] Kettermann, Uwe; Rohde, Andreas:
Spiele effektiv programmieren. - 1. Aufl. - Berlin: Springer, 2004

[8] Stelzer, André:
Visualisierung von skelettgesteuerten 3D-Character-Modellen. - Diplomarbeit - Mittweida, 2000

[9] Bronstein; Semendjajew; Musiol; Mühlig:
Taschenbuch der Mathematik. - 5. Aufl. - Thun und Frankfurt/Main: Verlag Harri Deutsch, 2001

[10] Ahearn, Luke:
3D Game Textures. - 1. Aufl. - Butterworth Heinemann: Oxford und Burlington, 2006

Abbildungsverzeichnis

Alle in dieser Diplomarbeit verwendeten Abbildungen, Texturen und 3D-Modelle sind vom Autor eigenhändig angefertigt wurden.

Anhang A: Vergleich vorhandener X3D-Viewer

Fakt \ Name	BS Contact VRML/X3D	Octaga Player	AccuTrans3D	SwirlViewer	Xj3D Browser	Enceladus	Diplomarbeit
Hersteller	Bitmanagment Software GmbH	Octaga	Micro Mouse Productions	Pinecoast Software	Web3DConsortium	dssd	**FH Mittweida**
Plug-In	X	X	-	-	-	-	-
Stand-alone		X	X	X	X	X	**X**
Import							
*.x3d	X	X	X (nur speichern)	X	X	X	**X**
*.x	-	-	X	-	-	-	**X**
Renderer							
Direct3D	X	-	-	-	-	-	**X**
OpenGL	X	X	X	X	Java3D	X	-
Ansichten							
Walk	X	X	-	X	X	X	
Fly	X	X	X	X	X	X	-
Examine	X	X	X	X	X	X	**X**
LookAt	X	X	-	X	X	X	
Kollisionstest	X	X	-	X	X	X	-
AlphaTexture	fehlerhaft	X	-	X	-	X	**X**
Ladezeiten[2]	1,8s	0,9s	-	6,75s	-	5,3s	**5,12s**
Kosten	100 €	90 €	20 $	Kostenlos	Kostenlos	Kostenlos	**Kostenlos**
Testversion	Eingeschränkt	Eingeschränkt	30 Tage	Keine	Keine	Keine	**Keine**
Website	http://www.bitmanagment.de	http://www.octaga.com	http://www.mircomouse.ca	http://www.pinecoast.com	http://www.xj3d.org	http://www.3dione.de	-

[2]: Lade der X3D-Datei mit dem Modell der Mensa der FH Mittweida mit der vorgegebenen Testumgebung (Kapitel 6.5)

Anhang B: Eigene Datentypen

Vektor für Fließkommazahlen

```
struct floatVector3
{
    float X, Y, Z;
};
```

Vektor für Integer-Zahlen

```
struct intVector3
{
    int A, B, C;
};
```

Rotationsvektor

```
struct rotVector4
{
    int X, Y, Z;
    float Phie;
};
```

Datenstruktur für den Material-Array

```
struct strMaterial
{
    floatVector3 DiffuseColor;
    floatVector3 EmissiveColor;
    floatVector3 SpecularColor;
    float transparency;
};
```

Anhang C: Inhalt der beiliegenden CD-ROM

Wichtiger Hinweis: Der Viewer auf der beiliegende CD ist voll funktionsfähig, wenn sich die zu öffnende X3D-Datei auf einem **nicht** schreibgeschützten Medium befindet!

Die CD-ROM beinhaltet folgende Ordner:

Diplomdokument

Dieser Ordner enthält diese Diplomarbeit im PDF-Format.

Programmierung

Dieser Ordner enthält den implementierten Konverter und Viewer:

X3DConvert

In diesem Ordner befinden sich alle Projektdateien des Konverters, inklusive der Klassen `CX3DInterpret` und `CX3DInOutput`.

Direct3DMFC

Dieser Ordner umfasst die Projektdateien des X3D-Viewers. Der wesentliche Teil der Renderumgebung befindet sich in den gleichnamigen Header- und C++-Datei `Direct3DMFCView`.

In dem Debug-Ordner befindet sich auch die aktuelle EXE-Datei des Konverters, da der Viewer auf diese beim Aufruf einer X3D-Datei zugreift.

Dateiformate

In diesem Ordner finden sich X3D- und DirectX-Dateien von verschiedenen 3D-Objekten.

NewMensa

In diesem Unterordner befindet sich das in dieser Diplomarbeit bearbeitete 3D-Modell der Mensa Mittweida in den Formaten X3D, SCN und COB.

TestSize

Dieser Ordner beinhaltet die beim Dateigrößenvergleich verwendeten 3D-Dateien (s. 10.4).

Abbildungen

In diesem Ordner befinden sich alle in der Diplomarbeit verwendeten Abbildungen.

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig und nur unter Verwendung der angegebenen Literatur, Weblinks und Hilfsmitteln angefertigt habe. Alle Teile, die wörtlich oder sinngemäß einer Veröffentlichung entstammen, sind als solche kenntlich gemacht.
Die Arbeit wurde noch nicht veröffentlicht oder einer anderen Prüfungsbehörde vorgelegt.

Bearbeitungsort, Datum	Unterschrift