

Übungen zu Graphikprogrammierung in C++

Abgabetermin: 23. Mai 2005

Was man wissen muss:

- *Erzeugen eines GL Render Contexts mit glut*
- *Mouse- und Keyboard-Handler in OpenGL*
- *Darstellung und Manipulation von 2D Objekten*
- *Klassenkonzept in C++*

Aufgabe 14 (H) Es werde 3D

Ähnlich wie in Aufgabe 12 soll eine statische Bibliothek (.lib) erzeugt werden, dieses Mal aber mit dem Namen `shape3D`. Hierfür soll das UML Diagramm aus Abbildung ?? ohne die Klasse `Transform3D` implementiert werden. Die Erzeugung einer statischen Bibliothek mit Visual Studio oder mit `g++` sollte bekannt sein. Für die Einstellungen unter Visual Studio bzw. im Makefile kann man sich allerdings an die schon bekannte `imagelib` halten.

Um die Ecken der 3D Objekte zu speichern, soll kein Pointer verwendet werden, sondern die von der STL (Standard Template Library) zur Verfügung gestellte `vector`-Klasse. Dies ist eine Templateklasse, d.h. sie kann mit verschiedenen Datentypen initialisiert werden. Der große Vorteil der Vektorklasse ist, dass dynamisch Elemente hinzugefügt und gelöscht werden können. Es folgt ein kleines Beispielprogramm zur Verwendung des Vektors:

```
#include <vector>
#include <iostream>
using namespace std;
int main(){
    vector<int> vectorOfInt;
    vectorOfInt.push_back(2);
    vectorOfInt.push_back(3);
    cout << "Size of Vector: " << vectorOfInt.size() << endl;
    cout << "Content: " ;
    vector<int>::const_iterator it;
    for (it=vectorOfInt.begin();it!= vectorOfInt.end(); it++){
        cout << *it;
    }
    cout << endl;
}
```

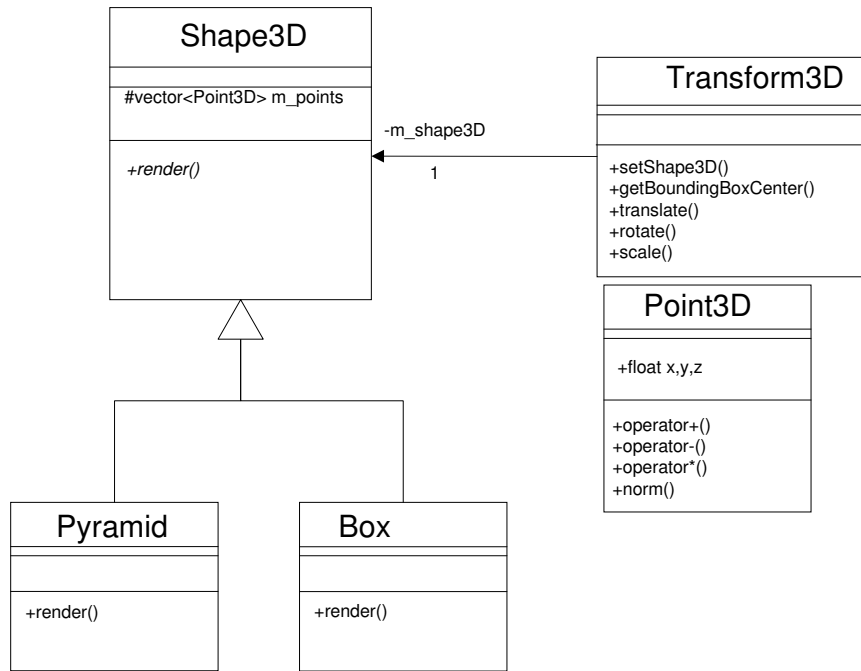


Abbildung 1: UML-Klassendiagramm für 3D Objekte

Mit einem Iterator, den es ebenfalls in der `vector`-Klasse gibt, kann durch den Inhalt des Vektors navigiert werden.

Es sollen die Klassen für eine dreiseitige Pyramide (`Pyramid`) und einen Quader (`Box`) implementiert werden. Als Zusatzaufgabe können weitere 3D Objekte implementiert werden. Vorerst sollen die Objekte noch nicht transformiert werden.

In der Datei `mainA14.cpp` können die implementierten 3D Objekte getestet und mit OpenGL gerendert werden. Diese Datei soll nun erweitert werden, um die 3D Objekte durch die Eingabe des Zeichens 'w' als Drahtmodell (wireframe) und durch das Drücken von 's' als Oberflächenmodell zu rendern. OpenGL stellt hierfür mit `glPolygonMode` eine Funktion zur Verfügung, die es erlaubt, mit zwei Parametern die Repräsentation des Objektes zu bestimmen. Im ersten Parameter wird bestimmt, welche Seite des Polygons mit diesem Modus dargestellt wird (z.B. `GL_FRONT_AND_BACK` oder `GL_FRONT`), der zweite Parameter bestimmt den Modus (`GL_POINT`, `GL_LINE` oder `GL_FILL`).

Aufgabe 15 (H) Transformationen in 3D

Eine Transformation in 3D besteht aus einer Rotation (Drehung), einer Translation (Verschiebung) und einer Skalierung.

Dreidimensionale Rotationen können durch eine 3×3 Matrix repräsentiert werden. Z.B. wird mit

$$P_1 = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} * P_2$$

der originale 3D Punkt $P_2 = (x_2, y_2, z_2)^T$ in den 3D Punkt $P_1 = (x_1, y_1, z_1)^T$ rotiert.

In 3D ist eine beliebige Rotation geometrisch einfach durch eine Rotation um die drei Koordinatenachsen x , y und z vorzustellen. Doch wie bekommt man diese elementaren Rotationen um die drei Achsen in eine einheitliche Matrixrepräsentation?

Im Folgenden stellt $R_x(\phi)$ eine Rotationsmatrix um die x-Achse dar. Respektive sind $R_y(\theta)$ und $R_z(\eta)$ zu verstehen.

$$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{pmatrix}, R_y(\theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix}, R_z(\eta) = \begin{pmatrix} \cos(\eta) & -\sin(\eta) & 0 \\ \sin(\eta) & \cos(\eta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

wobei ϕ der Winkel der Rotation um die x-Achse, θ der Winkel der Rotation um die y-Achse und η der Winkel der Rotation um die z-Achse ist.

Bei einer beliebigen Rotation im Raum spielt die Rotationsreihenfolge um die einzelnen Achsen eine wichtige Rolle. Die Rotationsmatrix, die sich aus den elementaren Rotationsmatrizen um die einzelnen Achsen zusammensetzt, wobei zuerst um die z-Achse, dann um y und zuletzt um die x-Achse rotiert wird, wird wie folgt als Produkt von Matrizen berechnet. $R = R_x() * R_y() * R_z()$

$$R = \begin{pmatrix} \cos(\eta)\cos(\theta) & -\sin(\eta)\cos(\theta) & \sin(\theta) \\ \sin(\eta)\cos(\phi) + \cos(\eta)\sin(\theta)\sin(\phi) & \cos(\eta)\cos(\phi) - \sin(\eta)\sin(\theta)\sin(\phi) & -\cos(\theta)\sin(\phi) \\ \sin(\eta)\sin(\phi) - \cos(\eta)\sin(\theta)\cos(\phi) & \cos(\eta)\sin(\phi) + \sin(\eta)\sin(\theta)\cos(\phi) & \cos(\theta)\cos(\phi) \end{pmatrix}$$

Unbedingt zu beachten ist, dass die Rotation nicht kommutativ ist, d.h. $R_x * R_y * R_z \neq R_z * R_y * R_x$. Wichtige Eigenschaften der 3×3 Rotationsmatrix R sind:

- $R^T = R^{-1}$ (die Transponierte der Matrix ist gleich dem Inversen)
- $R * R^T = I$ (wobei I die Einheitsmatrix ist)
- $rank(R) = 3$
- die Spalten und Zeilen aus R bilden eine orthonormal Basis

Eine Translation eines Punktes in 3D wird durch einen 1×3 Vektor $t = (t_x, t_y, t_z)^T$ durch $P_1 = P_2 + t$ berechnet, wobei $P_1 = (x_1, y_1, z_1)^T$ der verschobene und $P_2 = (x_2, y_2, z_2)^T$ der originale Punkt ist.

Eine Skalierung wird in 3D durch eine 3×3 Diagonalmatrix $S = diag(s_x, s_y, s_z)$ mit $P_1 = S * P_2$ durchgeführt.

Eine beliebige Transformation kann man durch eine Kombination aus Rotation, Translation und Skalierung darstellen.

$$P_1 = S * R * P_2 + t = \begin{pmatrix} s_x r_{11} & r_{12} & r_{13} \\ r_{21} & s_y r_{22} & r_{23} \\ r_{31} & r_{32} & s_z r_{33} \end{pmatrix} * P_2 + \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$$

Obige Matrizen sind alle in der sogenannten Zeilenschreibweise (row major), wie sie auch meistens in Mathematikbüchern verwendet werden. Im Gegensatz dazu gibt es auch die Spaltenschreibweise (column major), welche von OpenGL intern verwendet werden. Z.B. werden die Elemente der Matrix

$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ in C++ Arrays üblicherweise in der Reihenfolge 1,2,3,4,5,6,7,8,9 gespeichert, in OpenGL dagegen 1,4,7,2,5,8,3,6,9. Um deswegen von obiger Darstellungsweise auf die in OpenGL verwendete zu kommen, müssen die Matrizen einfach transponiert werden.

Anstatt eine Column-major Matrix von rechts mit einem transponierten 3D Punkt zu multiplizieren kann man auch eine Row-major Matrix von links mit dem Punkt multiplizieren, was das gleiche Ergebnis liefert:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} [x y z] \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix} \end{pmatrix}^T$$

In dieser Aufgabe sollen nun Datentypen für 1×3 Vektoren (Punkt3D) und 3×3 Rotationsmatrizen (RotMat) implementiert werden sowie alle für eine Transformation benötigten Operatoren. In der Applikation soll das Kommando $S * R * P_2 + t$ als Ergebnis den transformierten Punkt geben.

Abbildung ?? stellt das UML-Diagramm mit allen benötigten Klassen, Funktionen und Variablen dar. Die Bibliothek aus der vorigen Aufgabe muss also um die Klasse Transform3D erweitert werden. Mit der Datei mainA15.cpp soll getestet werden, ob alle implementierten Funktionen in der Bibliothek das gewünschte Ergebnis liefern.

Aufgabe 16 (H) Mehr Rotationen mit OpenGL - Homogene Koordinaten

In dieser Aufgabe soll eine Methode mainA16.cpp geschrieben werden, die ohne die in Aufgabe 15 implementierten Transformationsklassen auskommt. Es wird die von OpenGL verwendete Modell-Matrix verwendet, um die Objekte zu transformieren.

Homogene Koordinaten stellen die gesamte Transformation, welche Rotation, Translation und Skalierung beinhaltet, in einer einzigen Matrix dar. Dies ermöglicht, dass ein Punkt (oder eine Linie bzw. Objekt) durch eine Multiplikation mit einer 4×4 Matrix transformiert wird. Dafür muss die Repräsentation eines 3D Punktes geändert werden.

$$P_{hom} = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

Um diesen Punkt wie gewohnt im 3D Raum interpretieren zu können, skalieren wir ihn so, dass $w = 1$ ist. Der skalierte Punkt stellt exakt den gleichen Punkt in 3D dar.

$$P_{hom} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \frac{1}{w} \begin{pmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \\ w \end{pmatrix}$$

Ein Punkt P_2 wird nun durch die Multiplikation mit nur einer 4×4 Matrix T in einen Punkt P_1 transformiert.

$$P_1 = T * P_2 = \begin{pmatrix} s_x r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & s_y r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & s_z r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \\ z_2 \\ 1 \end{pmatrix},$$

wobei die s_i die Skalierung, die r_{jk} die Rotation und die t_l die Translation wie zuvor verwendet darstellen.

Für eine grundlegende Einführung in Homogene Koordinaten und Projektive Geometrie sei der Interessierte auf das Buch von Hartley und Zisserman "Multiple View Geometry in Computer Vision" oder die Vorlesung von Prof. Navab "3D Computer Vision" verwiesen. Eine Einführung, wie man Szenen in OpenGL u.a. mit Hilfe der Modell-Matrix anzeigt, findet sich in Kapitel 3 im Red Book, welches auf der Kurswebseite verlinkt ist.

Den Zeiger auf die aktuelle Modell-Matrix kann in OpenGL mit `glGetfv(GL_MODELVIEW, m)` geholt werden. Dies ist ein Array von 16 float Werten. Wie schon in der letzten Aufgabe erwähnt, ist **diese Matrix die Transponierte der obigen 4×4 Matrix T** .

Mit `glmMultMatrix` kann eine beliebige Transformationsmatrix mit der aktuellen Modell-Matrix multipliziert werden.

Mit `glmLoadMatrixf` kann eine beliebig erzeugte Matrix gesetzt werden. Dies überschreibt die aktuelle Matrix. Mit `glmLoadIdentity` wird die Einheitsmatrix geladen.

Mit `glPushMatrix` kann die aktuelle Matrix gesichert werden, dann eine neue Matrix geladen werden um später mit `glPopMatrix` die alte Version wiederherzustellen. Dies ist speziell für eine unabhängige Bewegung zweier Objekte wichtig.

Das Programm soll die beiden Objekte (Box und Pyramide) nun unabhängig voneinander bewegen. Hierfür sollen zwei Matrizen verwendet werden, die mit `glPushMatrix`, `glPopMatrix` und `glMultMatrix` die Pyramide im Intervall von 0.5° um die x-Achse und die Box im Intervall von 1° um die y-Achse und im Intervall von 0.2° um die z-Achse drehen.

Aufgabe 17 (H) Noch mehr Rotationen mit OpenGL - Achse und Drehwinkel

Die `mainA17.cpp` soll die in `mainA16.cpp` eingesetzten Operationen, welche die Modell-Matrix manipulieren, ersetzen.

OpenGL stellt im `MatrixMode` die Funktionen `glRotatef`, `glTranslatef` und `glScalef` zur Verfügung. Die Translation und Skalierung wird wie in den vorhergehenden Aufgaben beschrieben angewendet. Die Rotation unterscheidet sich jedoch. Die Funktion `glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z)` hat als Eingabe vier Parameter. Dabei ist `GLfloat` der Datentyp, der wie `float` verwendet werden kann (Das Programm kompiliert auch wenn es Parameter vom Datentyp `float` übergeben bekommt). Die Parameter `x,y,z` beschreiben einen Vektor in 3D und `angle` den Winkel, um den gedreht werden soll.

Die Funktionalität aus Aufgabe 16 soll nun in `mainA17.cpp` ohne Verwenden der `glMultMatrix` Funktion implementiert werden.