Grundlagen der 3D-Computergrafik Programmieren mit OpenGL

Marc Wagner
mcwagner@cip.informatik.uni-erlangen.de
13. November 1998

Was ist OpenGL?

OpenGL (Open Graphics Library) ist ein Softwareinterface zum Erzeugen schneller, bewegter 3D-Computergrafiken, das in etwa 120 verschiedene Funktionen zur Verfügung stellt. OpenGL ist hardware- und betriebssystem-unabhängig, was den Vorteil hat, daß ein OpenGL-Programm auf verschiedenen Rechnern beziehungsweise Betriebssystemen lauffähig ist. Als Konsequenz daraus ergibt sich jedoch, daß OpenGL keine Funktionen zur Verwaltung von zum Beispiel Fenstern oder Benutzereingaben zur Verfügung stellt. Es ist jedoch möglich zu diesem Zweck selbst eine Library zu entwerfen oder eine bereits existierende zu verwenden (zum Beispiel glaux.lib für Windows 95). Darüber hinaus gibt es noch die hardware- und betriebssystem-unabhängige OpenGL Utility Library (GLU), die weitere nützliche Grafikfunktionen zur Verfügung stellt, und somit für mehr Benutzerfreundlichkeit sorgt. Es existieren OpenGL-Implementationen für verschiedene Programmiersprachen. Die Beispiele in diesem Skript wurden jedoch ausschließlich mit C++ erstellt.

Die OpenGL-Command-Syntax

Alle OpenGL-Funktionen beginnen mit *gl*, alle OpenGL-Konstanten mit *GL_*. Manche Funktionen bestehen in mehreren Versionen, die sich in der Anzahl und der Art ihrer Parameter unterscheiden, im Wesentlichen jedoch das gleiche leisten. In einem solchen Fall wird nach dem Funktionsnamen noch die Anzahl der Parameter, gefolgt von einer Abkürzung des Parametertyps angegeben. Ein eventuell folgendes *v* signalisiert, daß die sogenannte Vektorversion verwendet wird, das heißt, die Parameter werden nicht einzeln angegeben sondern in einem Array gespeichert, wobei der Funktion ein Zeiger auf dieses Array übergeben wird. Zum Beispiel erzielen die drei Zeilen

```
glVertex2i(1, 3);   // Funktion erwartet zwei 32-bit-integer Werte
glVertex3f(1.0, 3.0, 0.0);   // Funktion erwartet drei 32-bit-floating-point Werte
GLfloat array[2] = {1.0, 3.0}; glVertex2fv(array);   // Funktion erwartet einen Zeiger
```

das gleiche Resultat. Darüber hinaus stellt OpenGL Äquivalente zu den meisten Grunddatentypen zur Verfügung, wie zum Beispiel *GLfloat* oder *GLint*, die ebenfalls alle mit den Buchstaben *GL* beginnen. Diese sollten im Sinn der problemlosen Portierbarkeit von OpenGL-Programmen wann immer möglich verwendet werden.

Das Erstellen dreidimensionaler Objekte

Der erste Schritt auf dem Weg zu einem 3D-Computerbild besteht darin, die Objekte, die später zu sehen sein sollen, in einem, dem Computer verständlichen, Format zu definieren. Diese dreidimensionalen Objekte, setzen sich in OpenGL ausschließlich aus Polygonen zusammen. Polygone wiederum werden durch Angabe ihrer Eckpunkte spezifiziert. Ein Würfel zum Beispiel besteht aus sechs Polygonen, definiert jeweils durch vier Eckpunkte.

Wie werden Punkte im dreidimensionalen Raum definiert?

Jeder dreidimensionale Punkt in OpenGL wird durch einen Vektor, bestehend aus vier Komponenten beschrieben, sogenannten homogenen Koordinaten. Der Vektor $(x, y, z, w)^T$ beschreibt den Punkt, mit x-Koordinate x/w, y-Koordinate y/w und z-Koordinate z/w im dreidimensionalen Raum. OpenGL verwendet das rechtshändige

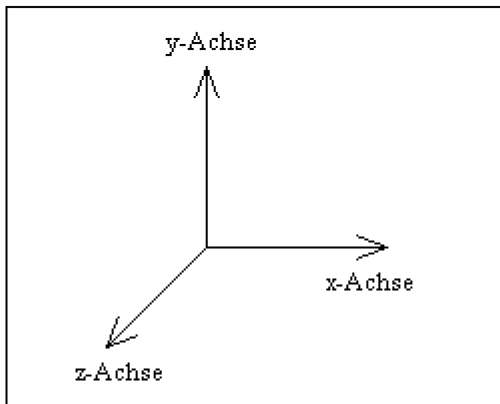

Bild 1: Das rechtshändige Koordinatensystem

Koordinatensystem, bei dem die x-Achse nach rechts, die y-Achse nach oben und die z-Achse auf den Betrachter zeigt. Die w-Koordinate ist bei verschiedenen Matrizenabbildungen notwendig, wie zum Beispiel der Translation, hat jedoch bei der Angabe von Punkten im Normalfall den Wert 1. Um einen Vektor in OpenGL anzugeben verwendet man die Funktion *glVertex*()*.

void glVertex{2 3 4}{s i f d}[v](TYPE coords);
Dient der Angabe eines Punktes durch zwei bis vier Koordinaten (x, y, z, w). Bei Verwendung von weniger als vier Parametern wird w auf 1, bei Verwendung von weniger als drei Parametern z auf 0 gesetzt. *glVertex*()* muß zwischen *glBegin()* und *glEnd()* aufgerufen werden.

Zum Beispiel gibt der Befehl

glVertex2f(1.5, 4.0);

den Vektor $(1.5, 4.0, 0.0, 1.0)^T$ an.

Was muß bei der Definition von Polygonen beachtet werden?

Polygone in OpenGL müssen konvex sein. Ein Polygon ist konvex genau dann, wenn jede Linie zwischen zwei belieben Punkten im Inneren des Polygons ebenfalls in dessen Inneren liegt. Natürlich ist es möglich nicht-konvexe Polygone zu erstellen, indem man verschiedene konvexe Polygone in geeigneter Weise zusammensetzt.

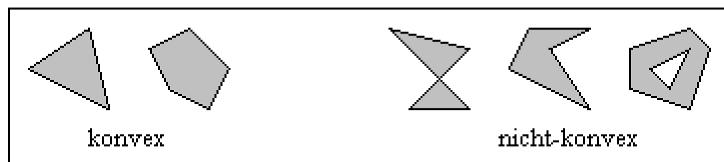

Bild 2: Konvexe und nicht-konvexe Polygone

Ferner müssen die Eckpunkte eines Polygons in einer Ebene liegen. Ist dies nicht der Fall, würde ein konvexes Polygon von einem bestimmten Blickwinkel aus betrachtet zu einem nicht-konvexen Polygon werden. Eine einfache Methode, dieser Schwiergkeit zu entgehen, besteht darin, Objekte nur aus Dreiecken aufzubauen, da drei Punkte stets in einer Ebene liegen.

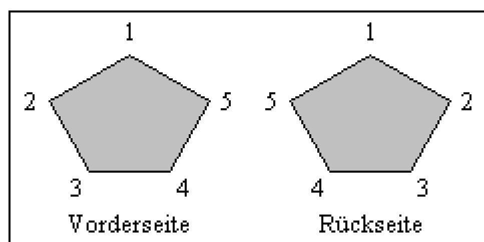

Bild 3: Vorder- und Rückseite eines Polygons

Außerdem sollte beachtet werden, daß Polygone eine Vorder- und eine Rückseite haben. Erscheinen die Vektoren eines Polygons gegen den Uhrzeigersinn, betrachtet man die Vorderseite, erscheinen sie im Uhrzeigersinn, die Rückseite. Bedeutung gewinnt dieser Sachverhalt dadurch, daß man verschiedene Einstellungen, die grafische Darstellung betreffend, für Vorder- und Rückseiten vornehmen kann. Es ist daher zu empfehlen, bereits bei der Erstellung von Objekten die Ausrichtung der Polygone einheitlich zu gestalten, da es meistens sehr zeitaufwendig ist, dies im nachhinein zu korrigieren.

Wie werden Polygone definiert?

Polygone werden durch Angabe Ihrer Eckpunkte in Vektorform zwischen den Funktionen *glBegin()* und *glEnd()* definiert.

void glBegin(GLenum mode);
Markiert den Anfang einer Liste von Vektoren. Der Parameter *mode* gibt die Art und Weise an, in der diese Vektoren verarbeitet werden.

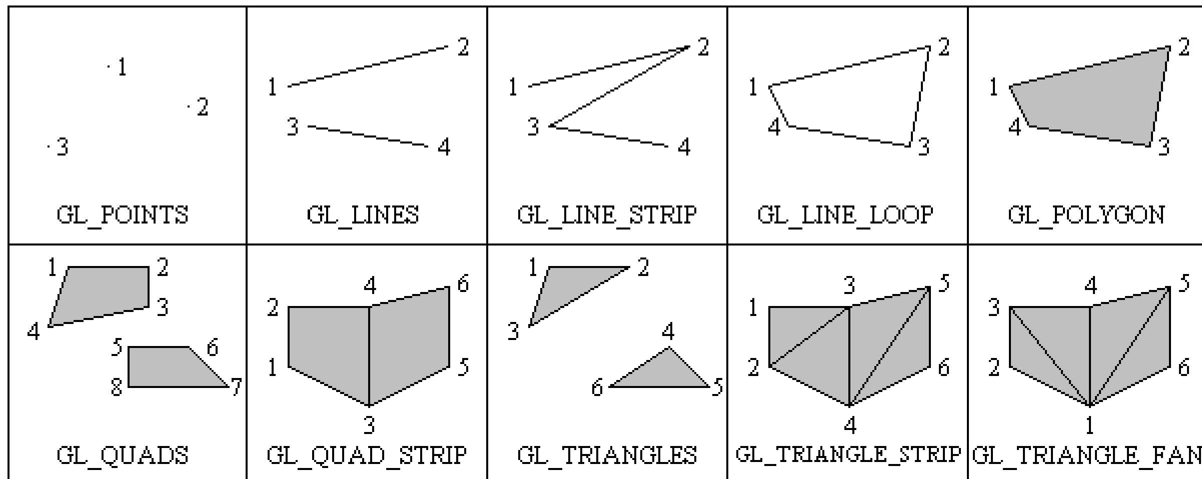

Bild 4: Die verschiedenen Möglichkeiten des Parameters *mode*

void glEnd(void);
Markiert das Ende einer Liste von Vektoren.

Zum Beispiel definiert die Befehlsfolge

```
glBegin(GL_POLYGON);
        glVertex3f(1.0, 0.0, 0.0);
        glVertex3f(0.0, 1.0, 0.0);
        glVertex3f(2.0, 0.0, 1.0);
glEnd();
```

ein Dreieck mit den Eckpunkten $(1.0, 0.0, 0.0)^T$, $(0.0, 1.0, 0.0)^T$ und $(2.0, 0.0, 1.0)^T$.

Wie werden Objekte mit gekrümmten Oberflächen dargestellt?

Da sich die Oberfläche eines Objekts in OpenGL ausschließlich aus Polygonen zusammensetzt, scheint es auf den ersten Blick unmöglich, Objekte mit gekrümmten Oberflächen, wie zum Beispiel eine Kugel darzustellen. Dies wäre jedoch ein gewaltiger Nachteil, da in der realen Welt die meisten Objekte irgendwelche Rundungen aufweisen. Man behilft sich, indem man sehr viele kleine Polygone verwendet. Diese approximieren die gewünschte Oberfläche ziemlich genau, und erwekken dadurch den Anschein einer Rundung.

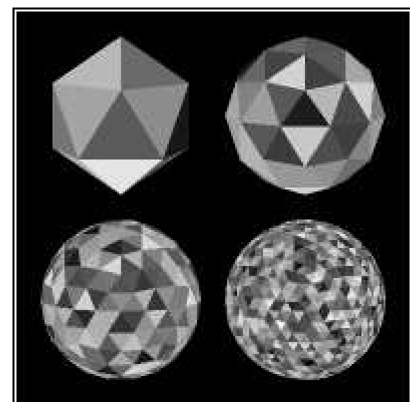

Bild 5: Approximieren einer Kugel

Wie wird die Farbe eines Polygons ausgewählt?

In OpenGL existieren zwei grundlegend verschiedene Methoden, eine gewünschte Farbe zu spezifizieren, der RGBA-Mode und der Color-Index-Mode. Um hier die richtige Entscheidung treffen zu können, ist ein gewisses Basiswissen, Computerfarben betreffend, erforderlich.

Farben auf einem Computerbildschirm

Ein Computerbild besteht aus einer Ansammlung von Bildpunkten, sogenannten Pixeln. Ein Pixel ist der kleinste Bereich auf einem Computerbildschirm, dem eine bestimmte Farbe zugewiesen werden kann. Jeder Pixel auf dem Monitor sendet eine gewisse Menge an rotem, grünem und blauem Licht aus. Durch geeignete Kombination der einzelnen Farbanteile kann jede vom menschlichen Auge wahrnehmbare Farbe dargestellt werden. Eine gewisse Vorstellung, welche Kombinationen welche Farben erzeugen, vermittelt ein Blick auf den hier abgebildeten Farbwürfel. Zum Beispiel ergibt eine Mischung von rotem und grünem Licht die Farbe gelb.

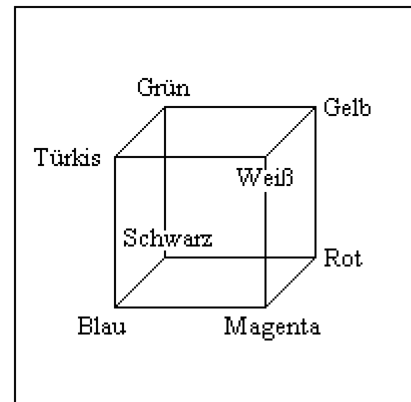

Bild 6: Der Farbwürfel

Der RGBA-Mode

Der RGBA-Mode ist der wesentlich häufiger verwendete Modus. Eine gewünschte Farbe wird spezifiziert, indem man den jeweiligen Anteil an rotem (R), grünem (G) und blauem (B) Licht direkt angibt. Der vierte Wert, der sogenannte Alpha Wert (A), dient vorwiegend dem Blending, einer Technik, die einem Objekt eine gewisse Transparenz zuweist. Mit der Funktion *glColor*()* legt man die gegenwärtige Objektfarbe fest.

void glColor{3 4}{b s i f d ub us ui}[v](TYPE colors);
Setzt die gegenwärtige Objektfarbe gemäß der angegebenen Parameter (R, G, B, A), die bis zum nächsten Aufruf dieser Funktion beibehalten wird. Bei Gleitkommazahlen bedeutet 1.0 das Maximum, 0.0 das Minimum des entsprechenden Farbanteils. Bei Angabe von nur drei Parametern wird der Alpha-Wert automatisch auf 1.0 gesetzt.

Zum Beispiel werden nach dem Befehl

glColor3f(1.0, 1.0, 0.0);

alle Objekte gelb gezeichnet, solange bis die nächste Farbänderung stattfindet. Einige Techniken, wie zum Beispiel Lighting, Texture-Mapping oder Fog, können nur bei Verwendung des RGBA-Mode voll ausgenutzt werden.

Der Color-Index-Mode

Der Color-Index-Mode verwendet eine sogenannte Palette. Eine Palette ist eine Ansammlung von Farben, wobei jeder Farbe eine eindeutige Zahl zugeordnet ist, ihr Index. Paletten werden vom jeweiligen Betriebssystem verwaltet. OpenGL stellt daher auch keine Funktionen, das Erstellen und Verändern von Paletten betreffend, zur Verfügung. Eine Farbe wird im Color-Index-Mode einfach über ihren Palettenindex angegeben, mittels der Funktion *glIndex*().*

void glIndex{s i f d}[v](TYPE index);
Zum Zeichnen von Objekten wird nach diesem Aufruf die dem Parameter *index* entsprechende Palettenfarbe verwendet. Sie wird bis zum nächsten Aufruf dieser Funktion beibehalten.

Es gilt zu beachten, daß man durch Ändern einer Palettenfarbe, mit der bereits Objekte gezeichnet wurden, auch deren Farben verändert. Im Color-Index-Mode können manche Möglichkeiten von OpenGL, wie zum Beispiel Lighting, Texture-Mapping oder Fog nur eingeschränkt oder gar nicht verwendet werden.

Ein Beispiel: Definition eines mehrfarbigen Würfels

Die folgende Funktion *Cube()* erstellt einen mehrfarbigen Würfel im RGBA-Mode. Die Ausrichtung der sechs Polygone wurde auf eine Weise konstant gehalten, daß ihre Vorderseiten nach außen, ihre Rückseiten ins Innere des Würfels weisen.

```
void Cube(void)
{
	glColor3f(1.0, 0.0, 0.0);   // Vorderseite in Rot
	glBegin(GL_POLYGON);
		glVertex3f(-0.5, -0.5, 0.5); glVertex3f(0.5, -0.5, 0.5);
		glVertex3f(0.5, 0.5, 0.5); glVertex3f(-0.5, 0.5, 0.5);
	glEnd();

	glColor3f(0.0, 1.0, 0.0);   // Rückseite in Grün
	glBegin(GL_POLYGON);
		glVertex3f(0.5, -0.5, -0.5); glVertex3f(-0.5, -0.5, -0.5);
		glVertex3f(-0.5, 0.5, -0.5); glVertex3f(0.5, 0.5, -0.5);
	glEnd();

	glColor3f(0.0, 0.0, 1.0);   // Linke Seite in Blau
	glBegin(GL_POLYGON);
		glVertex3f(-0.5, -0.5, -0.5); glVertex3f(-0.5, -0.5, 0.5);
		glVertex3f(-0.5, 0.5, 0.5);  glVertex3f(-0.5, 0.5, -0.5);
	glEnd();

	glColor3f(1.0, 1.0, 0.0);   // Rechte Seite in Gelb
	glBegin(GL_POLYGON);
		glVertex3f(0.5, -0.5, 0.5); glVertex3f(0.5, -0.5, -0.5);
		glVertex3f(0.5, 0.5, -0.5); glVertex3f(0.5, 0.5, 0.5);
	glEnd();

	glColor3f(1.0, 0.0, 1.0);   // Oberseite in Magenta
	glBegin(GL_POLYGON);
		glVertex3f(-0.5, 0.5, 0.5); glVertex3f(0.5, 0.5, 0.5);
		glVertex3f(0.5, 0.5, -0.5); glVertex3f(-0.5, 0.5, -0.5);
	glEnd();

	glColor3f(0.0, 1.0, 1.0);   // Unterseite in Cyan
	glBegin(GL_POLYGON);
		glVertex3f(0.5, -0.5, -0.5); glVertex3f(0.5, -0.5, 0.5);
		glVertex3f(-0.5, -0.5, 0.5); glVertex3f(-0.5, -0.5, -0.5);
	glEnd();
}
```

Die Kamera-Analogie

Ein Foto zu machen hat viel gemeinsam mit dem Erstellen eines dreidimensionalen Computerbildes. Man plaziert die Kamera und die zu fotografierenden Objekte an geeigneter Stelle und richtet die Kamera auf den gewünschten Zielpunkt aus. Diesem Vorgang entsprechen bei OpenGL die sogenannten Viewing- und Modeling-Transformations, die die Objekte relativ zu einer imaginären Kamera anordnen. Dann ist es bei einer echten Fotografie noch wichtig, ein entsprechendes Objektiv zu wählen, damit später genau der gewünschte Ausschnitt auf dem Bild zu sehen ist. Das Pendant in OpenGL ist die Projection-Transformation. Nachdem man den Auslößer betätigt hat, muß das Foto noch in der gewünschten Größe entwickelt werden. Dieser Vorgang wird in OpenGL durch die Viewing-Transformation repräsentiert. Die vier eben genannten Transformations haben die Aufgabe, jedem vorhandenen Vektor einen entsprechenden Punkt auf dem Computerbildschirm derart zuzuordnen, daß der Anschein einer realistischen, dreidimensionalen Szene entsteht. Die Viewing- und Modeling-Transformations sowie die Projection-Transformation werden, da es sich um lineare Abbildungen handelt, durch Matrizen dargestellt.

Allgemeines über Matrizenoperationen in OpenGL

Die Viewing- und Modeling-Transformations werden in der sogenannten Modelview-Matrix, die Projection-Transformation in der Projection-Matrix gespeichert. Es existiert in OpenGL sowohl ein Modelview-Matrix-Stack als auch ein Projection-Matrix-Stack, wobei beide zu Programmbeginn jeweils nur ein Element enthalten. Die beiden aktuellen Matrizen sind die jeweils obersten der beiden Stacks. Darüber hinaus existiert noch ein dritter Stack, der Texture-Matrix-Stack, auf dessen Bedeutung hier nicht näher eingegangen wird. Sämtliche Funktionen, die der Manipulation von Matrizen dienen, beeinflussen nur die oberste Matrix des aktuellen Stacks. Der aktuelle Stack läßt sich mit Hilfe der Funktion *glMatrixMode()* auswählen.

void glMatrixMode(GLenum mode);
Wählt den aktuellen Matrix-Stack aus, der solange als aktueller Stack gilt, bis die Funktion erneut mit einem anderen Parameter aufgerufen wird. Als Parameter sind zulässig *GL_MODELVIEW*, *GL_PROJECTION* und *GL_TEXTURE*.

Mit Hilfe der Matrix-Stacks ist es möglich, Matrizen zu speichern, um sie später erneut zu verwenden. Der Befehl *glPushMatrix()* kopiert die oberste Matrix des aktuellen Stacks und legt diese Kopie oben auf dem Stack ab. Da immer nur die oberste Matrix eines Stacks verändert werden kann, bleibt dessen zweites Element unverändert erhalten. Der Befehl *glPopMatrix()* entfernt das oberste Element des aktuellen Stacks und macht damit die Matrix darunter wieder zur aktuellen Matrix. Der Modelview-Matrix-Stack ist besonders bei der Konstruktion von Objekten hilfreich, die eine gewisse Hierarchie besitzen, das heißt deren Position voneinander abhängig ist. Ein gutes Beispiel hierfür ist ein mehrgliedriger Roboterarm.

void glPushMatrix(void);
Kopiert die oberste Matrix des aktuellen Stacks, und legt die Kopie oben auf dem Stack ab.

void glPopMatrix(void);
Entfernt die oberste Matrix des aktuellen Stacks.

Eine weitere, sehr wichtige Funktion, Matrizen betreffend, ist *glLoadIdentity()*. Dadurch wird die aktuelle Matrix durch die Identität ersetzt. Da die meisten anderen Matrizen-Funktionen Matrizen-Multiplikationen durchführen, ist es wichtig *glLoadIdentity()* zu benutzen, bevor man beginnt, eine neue Transformation zu definieren.

void glLoadIdentity(void);
Ersetzt die gegenwärtige Matrix durch die Identität.

Die Viewing- und Modeling-Transformations

Es ist unbedingt notwendig, die Viewing- und die Modeling-Transformations als eine Einheit zu betrachten. Unter Viewing-Transformations versteht man das Ausrichten und Positionieren der imaginären Kamera, unter Modeling-Transformations das Anordnen der abzubildenden Objekte. Ein einfaches Beispiel macht den engen Zusammenhang deutlich. Bewegt man die Kamera eine bestimmte Strecke in x-Richtung, hat man dadurch den selben Effekt erzielt, den ein Verschieben aller Objekte um die gleiche Strecke in negative x-Richtung bewirkt hätte. OpenGL kombiniert daher die Viewing- und Modeling-Transformations zu einer einzigen sogenannten Modelview-Matrix. Sämtliche mit *glVertex*()* angegebenen Vektoren werden mit der, zum jeweiligen Zeitpunkt des Funktionsaufrufs, obersten Matrix des Modelview-Stacks abgebildet. Die imaginäre Kamera befindet sich stets im Nullpunkt mit Blick in negative z-Richtung. Obwohl ihre Position konstant ist, ist es möglich, den Effekt einer Kamerabewegung zu erzielen, in dem man die Modelview-Matrix bereits vor der Angabe des ersten Vektors verändert, und damit jeden Vektor beeinflußt. Die Funktionen *glTranslate*()*, *glRotate*()* und *glScale*()* führen jeweils eine Matrizen-Multiplikation mit der obersten Matrix des aktuellen Stacks durch und ersetzen diese durch das Ergebnis. Ist M die aktuelle Matrix und N die durch den Funktionsaufruf beschriebene Matrix, wird M durch MN ersetzt. Es ist wichtig, sich darüber im klaren zu sein, daß die Reihenfolge von Matrizen-Multiplikationen entscheidend ist, das heißt die Matrix AB ist im Allgemeinen nicht gleich der Matrix BA. Als Konsequenz daraus ergibt sich, daß die zuletzt angegebene Matrix zuerst auf die entsprechenden Vektoren angewendet wird.

void glTranslate{f d}(TYPE x, TYPE y, TYPE z);

Multipliziert die aktuelle Matrix mit einer Matrix T, die einen Vektor, entsprechend der angegebenen Parameter, in x-, y-, und z-Richtung verschiebt.

$$\text{glTranslate*}(x, y, z) \longrightarrow T = \begin{vmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

void glRotate{f d}(TYPE angle, TYPE x, TYPE y, TYPE z);

Multipliziert die aktuelle Matrix mit einer Matrix R, die einen Vektor um den Winkel *angle* gegen den Uhrzeigersinn um die Achse dreht, die durch den Vektor $(x, y, z)^T$ definiert wird.

$$\text{glRotate*}(a, 1, 0, 0) \longrightarrow R = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos a & -\sin a & 0 \\ 0 & \sin a & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

$$\text{glRotate*}(a, 0, 1, 0) \longrightarrow R = \begin{vmatrix} \cos a & 0 & \sin a & 0 \\ 0 & 1 & 0 & 0 \\ -\sin a & 0 & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

$$\text{glRotate*}(a, 0, 0, 1) \longrightarrow R = \begin{vmatrix} \cos a & -\sin a & 0 & 0 \\ \sin a & \cos a & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

void glScale{f d}(TYPE x, TYPE y, TYPE z);

Multipliziert die aktuelle Matrix mit einer Matrix S, die einen Vektor, entsprechend der angegebenen Parameter, in x-, y-, und z-Richtung skaliert.

$$\text{glScale*}(x, y, z) \longrightarrow S = \begin{vmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Man sollte sich vor jedem dieser Funktionsaufrufe vergewissern, daß der Modelview-Stack auch der aktuelle ist, und falls nicht *glMatrixMode()* mit dem Parameter *GL_MODELVIEW* anwenden.

Ein Beispiel: Aufbau einer Szene, die mehrere Objekte enthält

Die folgende Funktion *Modelview()* erzeugt geeignete Viewing- und Modeling-Transformations, um mittels der oben definierten Funktion *Cube()* drei Würfel an verschiedenen Positionen in die Szene einzufügen.

```
void Modelview(void)
{
        glMatrixMode(GL_MODELVIEW);   // Macht die Modelview-Matrix zur aktuellen Matrix
        glLoadIdentity();   // Setzt die Modelview-Matrix gleich der Identität

        glPushMatrix();
                glTranslatef(0.0, 0.0, -5.0);   // Verschiebt Cube1 und Cube2 nach hinten

                glPushMatrix();
                        glTranslatef(-1.25, -1.25, 0.0);   // Verschiebt Cube1 nach links-unten
```

```
            glRotatef(210.0, 0.0, 1.0, 0.0);   // Dreht Cube1 210 Grad um die y-Achse
            Cube(); // Zeichnet Cube1
        glPopMatrix();

        glPushMatrix();
            glTranslatef(1.25, 1.25, 0.0);   // Verschiebt Cube2 nach rechts-oben
            Cube();   // Zeichnet Cube2
        glPopMatrix();
    glPopMatrix();   // Die aktuelle Modelview-Matrix ist nun wieder die Identität

    glTranslatef(0.0, 0.0, -8.0);   // Verschiebt Cube3 nach hinten
    glRotatef(35.0, 0.0, 1.0, 0.0);   // Dreht Cube3 40 Grad um die y-Achse
    glRotatef(-35.0, 0.0, 0.0, 1.0);   // Dreht Cube3 –35 Grad um die z-Achse
    glScalef(6.0, 3.5, 2.0);   // Skaliert Cube3
    Cube();   // Zeichnet Cube3
}
```

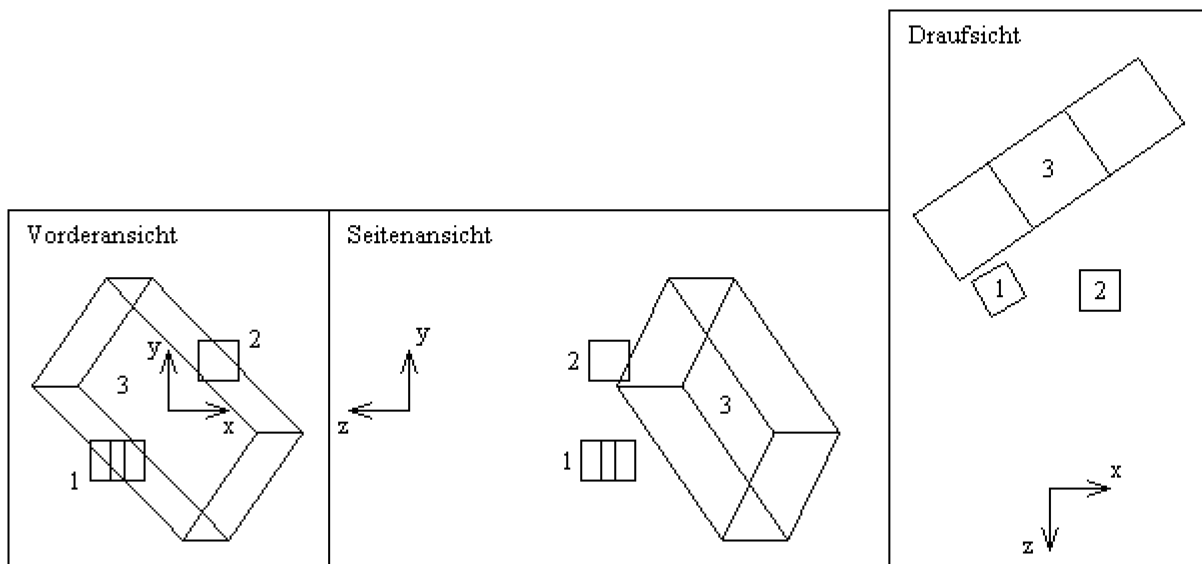

Bild 7: Vorder-, Seitenansicht und Draufsicht der durch *Modelview()* erzeugten Szene

Die Projection-Transformation

Mittels der Projection-Transformation definiert man, welcher Ausschnitt der Szene auf dem Bild zu sehen sein wird, und in welcher Art und Weise die Objekte erscheinen werden. Die Projection-Transformation beschreibt ein sogenanntes Viewing-Volume, einen von sechs Ebenen, sogenannten Clipping-Planes, begrenzten Bereich, dessen Inhalt später auf dem Monitor abgebildet wird. Jede Clipping-Plane teilt den dreidimensionalen Raum in zwei Bereiche auf, wobei für einen von beiden gilt, daß Objekte und Vektoren darin nicht weiter beachtet werden. Das bedeutet, alles was sich auf dieser Seite der Clipping-Plane befindet, wird auf dem 3D-Bild nicht erscheinen. Eine Projection-Transformation kann entweder eine Perspective-Projection oder eine Orthographic-Projection sein, zwei Methoden die sich grundlegend unterscheiden.

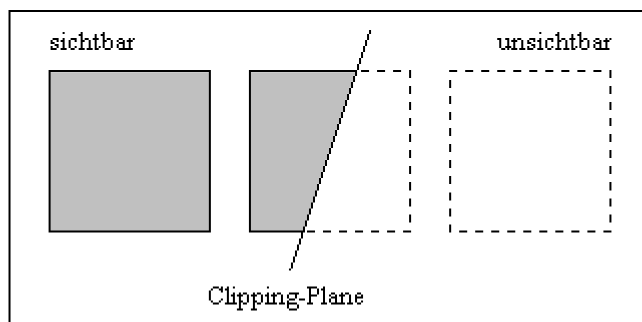

Bild 8: Funktionsweise einer Clipping-Plane

Die Perspective-Projection

Die Perspective-Projection erzeugt wesentlich realistischere Bilder, da Objekte, die weiter entfernt sind, kleiner erscheinen, als solche, die sich nahe beim Betrachter befinden. Die horizontale und vertikale Ausdehnung des Viewing Volumes nimmt daher proportional zur Entfernung von der imaginären Kamera zu. Das Viewing-Volume definiert gleichzeitig den Blickwinkel der Kamera, den man nicht zu groß wählen sollte, da die Objekte sonst stark verzerrt dargestellt werden. Mit der Funktion *glFrustum()* erzeugt man eine entsprechende Matrix für eine Perspective-Projection.

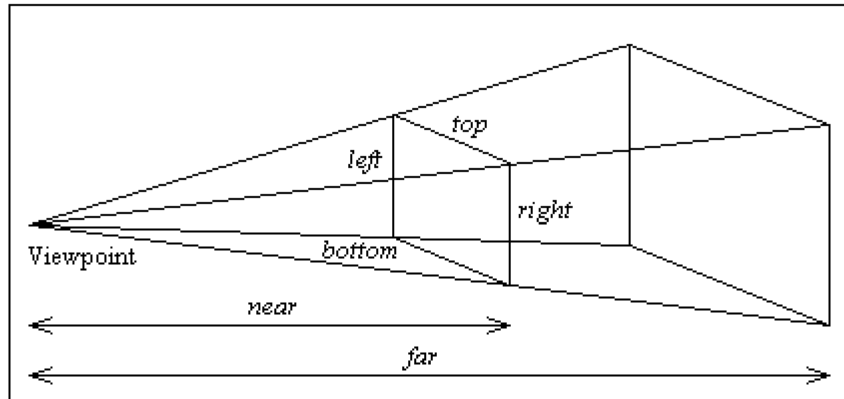

Bild 9: Das Perspective-Viewing-Volume

void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, Gldouble far);
Erzeugt eine Matrix für ein Perspective-Viewing-Volume und multipliziert sie mit der aktuellen Matrix. Das Bild des Perspective-Viewing-Volumes erläutert die Bedeutung der Parameter.

Es ist wichtig daran zu denken, daß beim Aufruf von *glFrustum()* der Projection-Matrix-Stack der aktuelle Stack sein sollte.

Die Orthographic-Projection

Bei der Orthographic-Projection ist die Größe, in der ein Objekt auf dem 3D-Bild erscheint, völlig unabhängig von dessen Entfernung zur imaginären Kamera. Die horizontale und vertikale Ausdehnung des Viewing-Volumes ändert sich daher nicht. Verwendung findet diese Methode vor allem bei schematischen Darstellungen, bei denen es darauf ankommt, Längen- und Winkelverhältnisse konstant zu halten. Der Befehl *glOrtho()* erzeugt eine entsprechende Matrix für eine Orthographic-Projection.

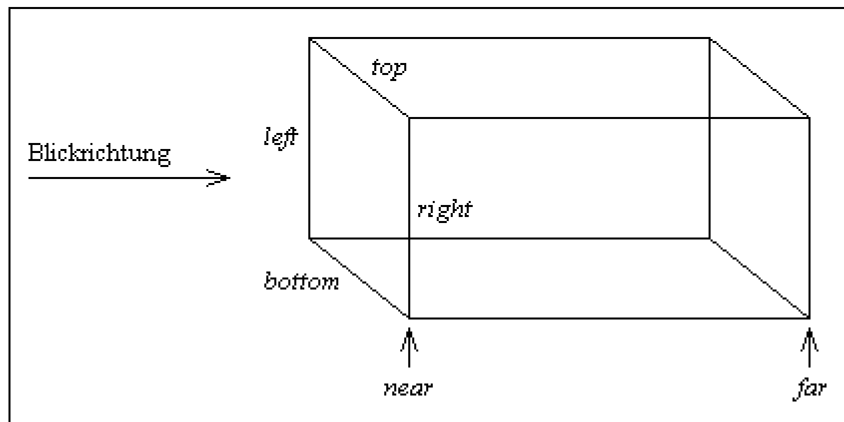

Bild 10: Das Orthographic-Viewing-Volume

void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);
Erzeugt eine Matrix für ein Orthographic-Viewing-Volume und multipliziert sie mit der aktuellen Matrix. Das Bild des Orthographic-Viewing-Volumes erläutert die Bedeutung der Parameter.

Auch für diese Funktion gilt, daß der Projection-Matrix-Stack der aktuelle sein sollte.

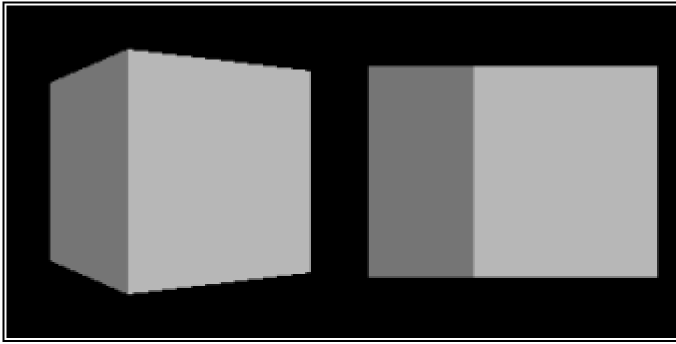

Bild 11: Perspective- und Orthographic-Projection eines Würfels

Die Viewport-Transformation

Die Viewport-Transformation bestimmt die Größe des Bildes auf dem Monitor in Pixeln, wobei Breite und Höhe voneinander unabhängig angegeben werden können. Stimmt das Seitenverhältnis des Viewing-Volumes nicht mit dem des Bildes überein, wird es verzerrt dargestellt. Zur Angabe der Viewport-Transformation verwendet man die Funktion *glViewport()*.

void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
Definiert das Rechteck im betreffenden Fenster, in dem das 3D-Bild zu sehen sein wird. *x* und *y* geben die linke, untere Ecke an, *width* und *height* die Breite und Höhe des Rechtecks.

Zum Beispiel legt der Funktionsaufruf

glViewport(0, 0, 300, 200);

die Größe des Bildes auf eine Breite von dreihundert und eine Höhe von zweihundert Pixeln fest, wobei seine linke, untere Ecke mit der linken, unteren Ecke des betreffenden Fensters übereinstimmt.

Die Reihenfolge der Transformations

Bild 12 zeigt die verschiedenen Schritte einer Vektor-Transformation in der Reihenfolge, in der sie der Computer ausführt. Zuerst wird der entsprechende Vektor mit der Modelview-Matrix abgebildet. Die resultierenden Koordinaten nennt man Eye-Coordinates. Dann folgt die Abbildung mit der Projection-Matrix, die alle Eye-Coordinates innerhalb des Viewing-Volumes in einen Würfel mit einer Kantenlänge von zwei Einheiten projeziert, in desen Mitte sich der Nullpunkt befindet. Das Ergebnis sind sogenannte Clip-Coordinates. Nun findet die Perspective-Division statt, bei der die x-, y-, und z-Koordinaten eines Vektors durch seine w Koordinate geteilt werden. Die dadurch entstandenen Normalized-Device-Coordinates werden jetzt nur noch mit Hilfe der Viewport-Transformation in das gewünschte Fenster projeziert, das Resultat sogenannte Window-Coordinates.

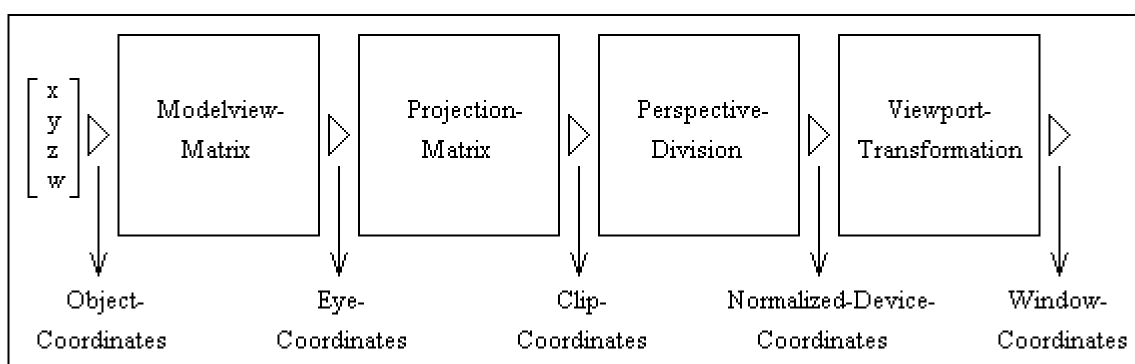

Bild 12: Die Schritte einer Vektor-Transformation

Der Framebuffer

Zu jedem Pixel des entstehenden 3D-Bildes muß eine bestimmte Menge an Informationen, das heißt eine gewisse Anzahl von Bits, gespeichert werden. Diese Bits, für jeden Bildpunkt die gleiche Anzahl, werden in sogenannten Bitplanes gespeichert. Jede Bitplane enthält genau ein Bit pro Pixel, ganz offensichtlich nicht ausreichend für ein ansprechendes Computerbild. Jedes Bild ist daher mit mehreren Bitplanes verbunden. Die Menge aller vorhandenen Bitplanes nennt man den Framebuffer. Der Framebuffer besteht je nach Art und Komplexität des OpenGL-Programms aus verschiedenen kleineren Buffern. Hier sollen nur die beiden wichtigsten näher erläutert werden, der Color-Buffer und der Depth-Buffer. Das Erzeugen eines mit einem entsprechenden Framebuffer verbundenen Fensters ist abhängig vom verwendeten Betriebssystem. OpenGL stellt deshalb dafür keine Funktionen zur Verfügung. Es gibt jedoch Hilfsbibliotheken, zum Beispiel die bereits erwähnte glaux.lib für Windows 95, die Funktionen beinhalten, die diese Aufgabe erleichtern.

Der Color-Buffer

Im Color-Buffer wird die Farbe jedes einzelnen Bildpunktes gespeichert. Da die Farbinformationen die Basis eines jeden Computerbildes bilden, muß jedes OpenGL-Programm einen Color-Buffer erzeugen. Die Anzahl der Bitplanes im Color-Buffer sollte abhängig von der Leistungsfähigkeit der zur Verfügung stehenden Hardware gewählt werden. Jeder Funktionsaufruf zum Zeichnen eines auf dem Bild sichtbaren Polygons verändert den Inhalt des Color-Buffers. Bei Programmstart beziehungsweise bei Zeichenbeginn eines Bildes sollte der Inhalt des Color-Buffers mit der Funktion *glClear()* gelöscht werden.

void glClear(GLbitfield mask);
Löscht alle angegebenen Buffer. Der Parameter *mask* sollte eine beliebige bitweise Kombination folgender Konstanten sein: *GL_COLOR_BUFFER_BIT*, *GL_DEPTH_BUFFER_BIT*, *GL_STENCIL_BUFFER_BIT*, *GL_ACCUM_BUFFER_BIT*.

Die Farbe, mit der der Color-Buffer gelöscht wird, kann mit den Funktionen *glClearColor()* im RGBA-Mode beziehungsweise *glClearIndex()* im Color-Index-Mode angegeben werden.

void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
void glClearIndex(GLfloat index);
Diese Funktionen geben die Farbe an, mit der der Color-Buffer gelöscht wird. Die Löschfarbe bleibt bis zum nächsten Aufruf dieser Funktionen erhalten.

Zum Beispiel bewirkt die Anweisungsfolge

```
glClearColor(0.0, 1.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

daß Color- und Depth-Buffer gelöscht werden, der Color-Buffer mit grüner Farbe.

Der Depth-Buffer

Im Depth-Buffer wird der Abstand eines jeden Pixels in z-Richtung von der imaginären Kamera gespeichert. Der Depth-Buffer wird daher auch häufig als z-Buffer bezeichnet. Bevor der Inhalt des Color-Buffers durch einen Funktionsaufruf zum Zeichnen eines Objekts verändert wird, wird für jeden Pixel ein sogenannter Depth-Test durchgeführt, bei dem die Entfernung des gegenwärtig gespeicherten Bildpunktes mit der des neuen Bildpunktes verglichen wird. Nur wenn sich der neue Pixel näher an der imaginären Kamera

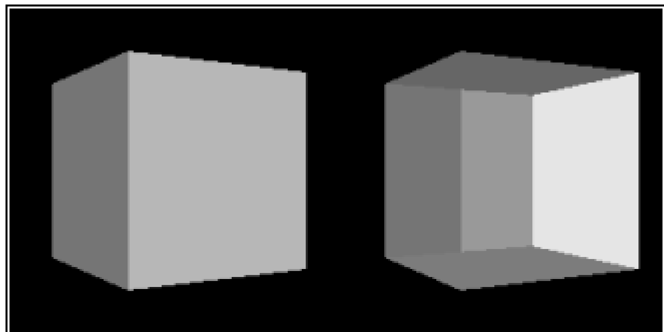

Bild 13: Ein Würfel, gezeichnet mit und ohne Depth-Test

befindet, wird der Wert im Color-Buffer überschrieben. Der Depth-Buffer gewährleistet also, daß nahe Objekte weiter entfernte verdecken, und sollte daher bei fast allen dreidimensionalen Darstellungen verwendet werden. Bei Programmstart beziehungsweise bei Zeichenbeginn eines Bildes sollte der Inhalt des Depth-Buffers mit der Funktion *glClear()* gelöscht werden. Darüber hinaus muß der Depth-Test mit dem Funktionsaufruf

glEnable(GL_DEPTH_TEST);

einmalig aktiviert werden.

Ein Beispiel: Ein vollständiges OpenGL-Programm

Das folgende OpenGL-Programm erstellt ein 3D-Computerbild, auf dem mehrere Würfel zu sehen sind. Die Funktionen *OpenAWindow()*, erzeugt ein Fenster im RGBA-Mode, vierhundert mal vierhundert Pixel groß, verbunden mit einem Color- und einem Depth-Buffer, und *KeepTheWindowOnTheScreenForAWhile()*, verhindert, daß das Fenster nach Ausführung des Programms sofort automatisch geschlossen wird, müssen entsprechend des verwendeten Betriebssystems implementiert werden. Die einzige im Programm vorkommende und bisher noch nicht behandelte OpenGL-Funktion ist *glFlush()*, die gewährleistet, daß alle bis dahin angegebenen Grafik-Befehle augenblicklich ausgeführt werden.

void glFlush(void);
Sorgt dafür, daß alle bisher aufgerufenen und zur Verarbeitung noch ausstehenden OpenGL-Funktionen sofort ausgeführt werden.

Insbesondere wenn ein OpenGL-Programm in einem Netzwerk ausgeführt wird, das heißt das Programm läuft auf einem Rechner, während ein anderer die Grafikausgabe übernimmt, oder wenn eine Grafik-Pipeline zur Verfügung steht, kommt es häufig vor daß nicht jeder Grafikbefehl einzeln und unverzüglich ausgeführt wird. Es ist durchaus möglich, daß der Aufruf von *glFlush()* bei manchen Programmen beziehungsweise Systemen überflüssig ist, sollte aber sicherheitshalber immer verwendet werden.

```
#include <WhateverYouNeed.h>

main()
{
	OpenAWindow();

	glEnable(GL_DEPTH_TEST);   // Aktiviert den Depth-Test

	glViewport(0, 0, 400, 400);   // Die Bildgröße wird auf 400 mal 400 Pixel festgelegt

	glMatrixMode(GL_PROJECTION);   // Macht die Projection-Matrix zur aktuellen Matrix
	glLoadIdentity();   // Setzt die Projection Matrix gleich der Identität
	glFrustum(-1.0, 1.0, -1.0, 1.0, 1.0, 50.0);   // Erzeugt ein Perspective-Viewing-Volume

	glClearColor(0.0, 0.0, 0.0, 0.0);   // Setzt die Löschfarbe des Color-Buffers auf Schwarz
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);   // Löscht Color- und
	                                                      // Frame-Buffer

	Modelview();   // Erzeugt und plaziert die Objekte in der Szene

	glFlush();   // Gewährleistet die Ausführung aller bisher angegebenen Grafikbefehle

	KeepTheWindowOnTheScreenForAWhile();
}
```

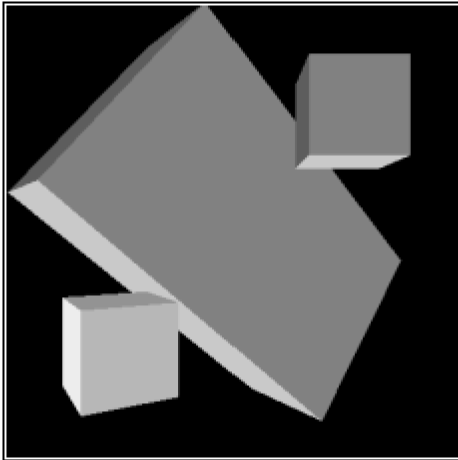

Bild 14: Die Grafikausgabe des Programms

Display-Listen

Display-Listen dienen dem Steigern der Geschwindigkeit eines OpenGL-Programms, besonders dann, wenn dieses in einem Netzwerk ausgeführt wird. Es ist nämlich häufig der Fall, daß das eigentliche Programm auf einem Rechner, dem sogenannten Client, läuft, während ein anderer Computer, der sogenannte Server, die Grafikausgabe übernimmt. Mittels einer Display-Liste wird eine Reihe von OpenGL-Funktionen zusammengefaßt, vom Client zum Server geschickt, und dort in hardware-günstiger Form gespeichert, wo sie nun beliebig oft verwendet werden kann. Anstatt nun jedesmal alle in der Liste enthaltenen Befehle über das Netz zu schicken, wird vom Client lediglich die entsprechende Display-Liste aufgerufen, und dadurch diese, die ja bereits in einem günstigen Format beim Server vorliegt, auf dem Server-Computer ausgeführt. Aber auch wenn man nicht in einem Netzwerk arbeitet, kann durch Display-Listen die Leistung eines Programms gesteigert werden, da ihr Inhalt beim Erzeugen in eine Form umgewandelt wird, die schneller von der entsprechenden Grafik-Hardware verarbeitet werden kann.

Wann sind Display-Listen zu empfehlen?

Display-Listen sollte man vor allem dann verwenden, wenn das OpenGL-Programm in einem Netzwerk eingesetzt werden soll, oder viele zeitintensive Funktionsaufrufe enthält, die eventuell mehrmals auszuführen sind. Ein gutes Beispiel hierfür sind Matrizenoperationen. Beim Speichern in Display-Listen wird bereits ein Teil der notwendigen Berechnungen, wie zum Beispiel das Ermitteln der Inversen, nötig bei den meisten Matrizenoperationen, ausgeführt und abgespeichert. Außerdem werden eventuell hintereinander stattfindende Matrizenmultiplikationen, wie sie häufig bei den Viewing- und Modeling-Transformations vorkommen, zu einer einzigen Matrix kombiniert. Weitere Hauptanwendungsgebiete finden sich bei der Verwendung von Texturen, dem Festlegen von Materialeigenschaften und dem Definieren von Lichtquellen.

Wie verwendet man Display-Listen?

Man erzeugt eine neue Display-Liste, indem man zwischen den Funktionen *glNewList()* und *glEndList()* die zu speichernde Befehlsfolge angibt.

void glNewList(GLuint list, GLenum mode);
Diese Funktion markiert den Beginn einer neuen Display-Liste. Über den Parameter *list* gibt man einen positiven, eindeutigen Index an, mit dessen Hilfe man später die Liste aufrufen kann. Der Parameter *mode* kann entweder den Wert *GL_COMPILE* oder *GL_COMPILE_AND_EXECUTE* annehmen. In letzterem Fall wird die Liste nicht nur erzeugt, sondern auch sofort ausgeführt.

void glEndList(void);
Diese Funktion markiert das Ende einer Display-Liste.

glNewList() erwartet als Parameter einen eindeutigen Index. Durch Einsetzten dieses Werts in die Funktion *glCallList()* wird die sofortige Ausführung der in der Display-Liste gespeicherten Befehle veranlaßt.

void glCallList(GLuint list);
Diese Funktion veranlaßt die Ausführung der in der Display-Liste, mit Index *list*, gespeicherten Grafikbefehle.

In einer Display-Liste werden ausschließlich OpenGL-Funktionen gespeichert. Der Inhalt einer Display-Liste kann nach ihrer Erzeugung nicht mehr verändert werden.

Ein Beispiel: Mehrfache Verwendung einer Display-Liste

Wir fügen obigem Programm die folgende Funktion hinzu, und ersetzen in der Funktion *main()* den Aufruf *von Modelview()* durch einen Aufruf *von DisplayList()*. Der durch *Cube()* definierte Würfel wird in der Display-Liste mit Index 1 gespeichert. Diese wird dann dreiundsechzigmal aufgerufen, um eine aus Würfeln bestehende Ebene zu erzeugen.

```
void DisplayList(void)
{
        int i1, i2;

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        glNewList(1, GL_COMPILE);   // Markiert den Anfang einer neuen Display-Liste mit Index 1
                Cube();
        glEndList();   // Markiert das Ende der Display-Liste

        // Hier wird eine Ebene, bestehend aus sieben mal neun Würfeln, erzeugt
        for(i1 = -4; i1 < 5; i1++)
                for(i2 = -7; i2 < 0; i2++)
                {
                        glPushMatrix();
                                glTranslatef((GLfloat)i1*2.0, -1.5, (GLfloat)i2*2.0);
                                glCallList(1);   // Ruft die Display-Liste auf
                        glPopMatrix();
                }
}
```

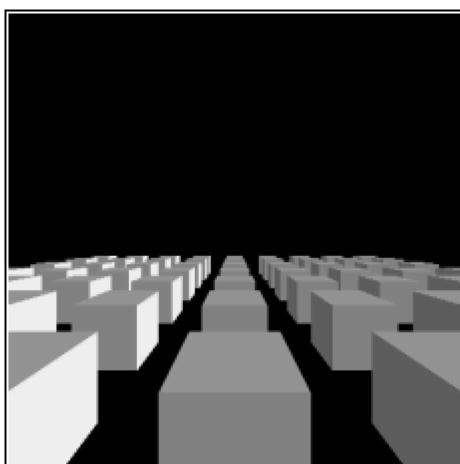

Bild 15: Die Grafikausgabe des Programms

Welche Möglichkeiten bietet OpenGL noch?

Da selbst bei den bisherigen Grundlagen nur die wichtigsten Punkte angesprochen werden konnten, ist es ausgeschlossen, im Rahmen dieses Vortrags sämtliche Möglichkeiten detailliert zu erläutern, die OpenGL bietet. Zumindest einige davon sollen aber noch in Ansätzen erwähnt werden. Besteht der Wunsch, diese Methoden ins eigene Programm einfließen zu lassen, wird man am Studium weiterführender Literatur nicht vorbeikommen. Empfohlen sei hier der OpenGL Programming Guide, der als Hauptreferenz bei der Ausarbeitung dieses Vortrags diente.

Lighting

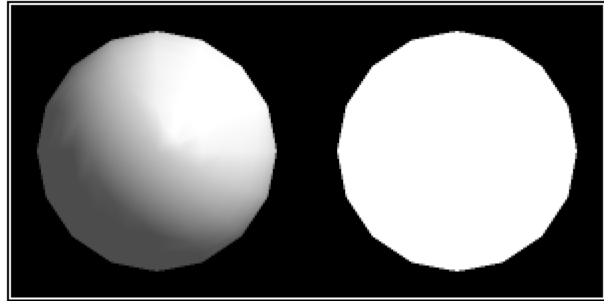

Bild 16: Eine Kugel, gezeichnet mit und ohne Lighting

Um einigermaßen realistische Szenen mit OpenGL zu erstellen, ist es fast zwangsläufig notwendig, Beleuchtungseffekte, das sogenannte Lighting, einzusetzen. OpenGL gestattet es, bis zu acht Lichtquellen gleichzeitig in einer Szene zu plazieren. Jede von ihnen kann mit verschiedenen Eigenschaften versehen werden, mit deren Hilfe unterschiedliche Lichtquellen simuliert werden können, wie zum Bespiel die Sonne, ein Scheinwerfer oder eine Glühlampe. Darüber hinaus weißt man Polygonen, genauer gesagt ihren Vektoren, nicht mehr nur eine einzige Farbe, sondern sogenannte Materialeigenschaften zu. Diese definieren sein Aussehen bei verschiedenen Lichtverhältnissen. Geschickt eingesetzt kann man dadurch den Eindruck erwecken, die Oberfläche bestünde aus einem bestimmten Material wie zum Beispiel Metall. Weder Schatten noch Spiegeleffekte werden von OpenGL automatisch erzeugt.

Blending

Unter Blending versteht man, den Farbwert eines neuen Objekts mit dem des Untergrunds in einer vorher definierten Art und Weise zu kombinieren. Bei dieser Technik findet häufig der bereits erwähnte Alpha-Wert Verwendung, mit dessen Hilfe man zum Beispiel festlegen kann, wie durchscheinend ein bestimmtes Objekt ist. In OpenGL wird die anzuwendende Blending-Funktion mittels *glBlendFunc()* spezifiziert.

void glBlendFunc(GLenum sfactor, GLenum dfactor);
Der Parameter *sfactor* beschreibt die Modifikation der Objektfarbe, der Parameter *dfactor* beschreibt die Modifikation der Untergrundfarbe. Die beiden Farbwerte werden dann addiert, das Ergebnis in den Color-Buffer geschrieben.

Um einem neuen Objekt eine gewisse Transparenz zuzuweisen, würde man also den Funktionsaufruf

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

verwenden. Eine weitere Möglichkeit wäre aus einem bestehenden Bild einen bestimmten Anteil einer Farbe, zum Beispiel 60% der Farbe Rot, herauszufiltern. Man würde die Blending-Funktion mittels

glBlendFunc(GL_ZERO, GL_SRC_COLOR);

spezifizieren und ein Polygon mit dem RGBA-Wert (0.4, 1.0, 1.0, 1.0) über die ganze Szene zeichnen. Für eine genaue Beschreibung sämtlicher Parameter siehe den OpenGL Programming Guide.

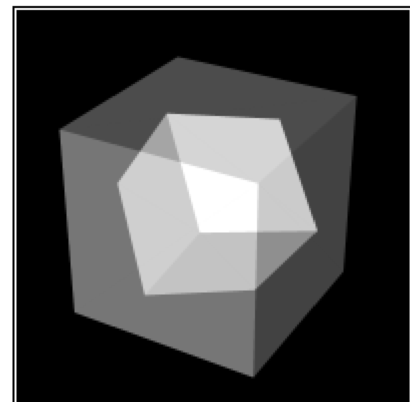

Bild 17: Blending zweier Würfel

Antialiasing

Da der Pixel die kleinste Einheit auf dem Monitor ist, dem eine bestimmte Farbe zugewiesen werden kann, erscheinen Linien oder Kanten oft etwas gezackt und daher unnatürlich. Verwendet man Antialiasing, wird ermittelt, welcher Bruchteil eines Pixels von einem Objekt tatsächlich eingenommen wird. Gemäß dessen wird eine Farbkombination gebildet, eine Mischung aus Objekt- und Untergrundfarbe, deren Wert der betreffende Bildpunkt dann erhält.

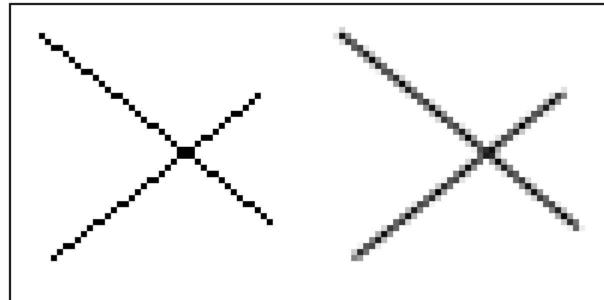

Bild 18: Aliased und antialiased Lines

Fog

Durch die Verwendung von Fog, erreicht man, daß weiter entfernte Objekte zunehmend verblassen und schließlich ganz verschwinden. Fog wird häufig bei der Darstellung von Außenszenen eingesetzt, wobei dadurch der Anschein von Nebel, Rauch oder Wolken erweckt wird. Insbesondere bei großen Szenen kann sich Fog positiv auf die Geschwindigkeit des OpenGL-Programms auswirken, da weit entfernte Objekte nicht mehr gezeichnet werden müssen.

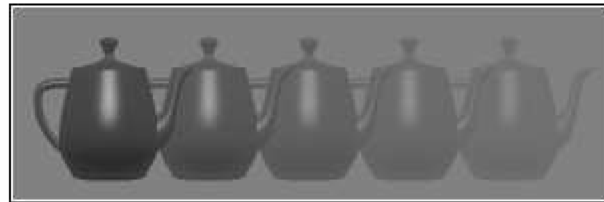

Bild 19: Teekannen, die im Nebel verschwinden

Texturen

Texturen sind unabdingbar beim Erstellen realistischer 3D-Bilder. Eine Textur ist im Normalfall ein zweidimensionales, rechteckiges Bild von einer bestimmten Oberfläche wie zum Beispiel einer Steinwand, einer Tapete oder einer Holzplatte. Eine Textur wird durch Angabe entsprechender Textur-Koordinaten auf eine bestimmte Oberfläche projeziert, ähnlich dem Aufkleben eines Abziehbilds. Die linke untere Ecke einer Textur hat die Koordinaten (0.0, 0.0), die rechte obere Ecke die Koordinaten (1.0, 1.0). Jedem Vektor weißt man mit Hilfe der Funktion *glTexCoord*()* die gewünschten Textur-Koordinaten zu.

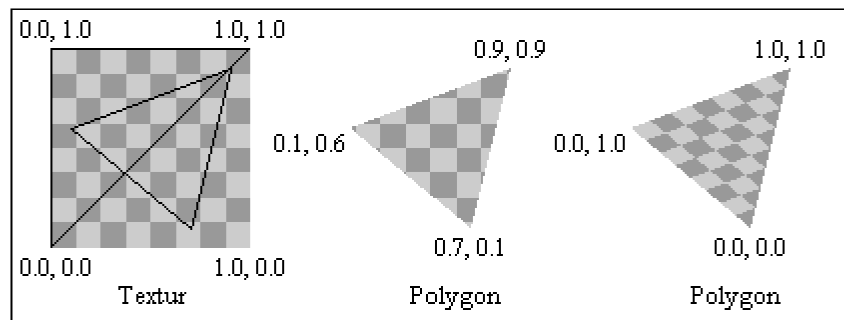

Bild 20: Die Verwendung von Textur-Koordinaten

void glTexCoord{1 2 3 4}{s i f d}{v}(TYPE coords);

glTexCoord()* weißt einem Vektor die Texturkoordinaten (s, t, r, q) zu. r hat momentan noch keine Bedeutung und wird daher ignoriert, q entspricht der w-Koordinate bei der Defininition normaler Vektoren. Ein Vektor erhält also effektiv die beiden Texturkoordinaten (s/q, t/q).

Der diesen Koordinaten entsprechende Teil der Textur wird dann auf das Polygon projeziert. Dadurch lassen sich auf einfache Weise sehr realistische Effekte erzielen, wie sie bei ausschließlicher Verwendung von Materialeigenschaften gar nicht möglich wären.

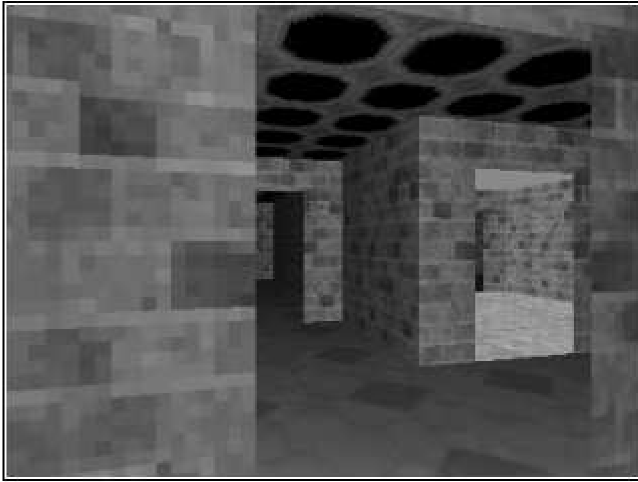

Bild 21: Texturen für Wände, Fußboden und Decke

Animierte Szenen

Um eine flüssig wirkende, animierte Szene zu erzeugen, sollte ein OpenGL-Programm mindestens zwanzig Bilder pro Sekunde erzeugen. Fällt die Leistung unter diesen Wert, wirkt die Bewegung der Objekte ruckartig. Zeichnet man diese Bilder der Reihe nach in den sichtbaren Color-Buffer, auch genannt Front-Buffer, erhält man ein höchst unbefriedigendes Resultat. Da das aktuelle Bild komplett gelöscht bevor das darauf folgende aufgebaut wird, und dieser Aufbau ebenfalls eine gewisse Zeit in Anspruch nimmt, sieht der Betrachter je nach Geschwindigkeit des Systems eine Mischung von halbfertigen und fertigen Szenen. Im schlimmsten Fall blitzen die zum Schluß gezeichneten Objekte nur ganz kurz auf. Man schafft Abhilfe, in dem man einen zusätzlichen, nicht sichtbaren Color-Buffer, den sogenannten Back-Buffer verwendet. In diesen Back-Buffer werden der Reihe nach alle Bilder der Szene gezeichnet. Nachdem ein Bild fertiggestellt ist, tauscht man mittels einer bestimmten Funktion die Inhalte von Front- und Back-Buffer einfach aus. Das Ergebnis ist eine fließend wirkende Animation.