



EINFÜHRUNG IN DIE GRAPHIKSCHNITTSTELLE OPENGL

Jan Fischer

Reader zur Vorlesung Computergraphik
im Sommersemester 2001

ULM 2001

Inhaltsverzeichnis

INHALTSVERZEICHNIS	2
ÜBER DIESEN READER	3
1. OPENGL - EINE MODERNE GRAPHIKSCHNITTSTELLE	3
2. ALLGEMEINE HINWEISE ZU OPENGLS API.....	5
2.1 DATENTYPEN	5
2.2 FUNKTIONEN	5
2.3 KONSTANTEN	6
2.4 HEADERDATEIEN	6
3. ZUSATZBIBLIOTHEKEN UND ANBINDUNG AN DAS FENSTERSYSTEM.....	7
3.1 SYSTEMABHÄNGIGE ANBINDUNGEN.....	7
3.2 SYSTEMUNABHÄNGIGE ANBINDUNGEN	7
3.3 ZUSATZBIBLIOTHEKEN ZU OPENGL	9
4. PRIMITIVE IN OPENGL.....	10
4.1 GLBEGIN [1,SEITE 73]	11
4.2 GLVERTEX* [1,SEITE 78]	11
4.3 GLEND [1,SEITE 73]	12
4.4 BEISPIEL ZUR DEFINITION VON PRIMITIVEN	12
5. BEEINFLUSSUNG DES ZEICHENSTILS	13
5.1 DIE PUNKTAUSDEHNUNG [1,SEITE 79]	13
5.2 DIE LINIENBREITE [1,SEITE 80]	14
5.3 LINIEN- UND POLYGONMUSTER [1,SEITE 80 UND 84]	14
5.4 FARBE [1,SEITE 137]	14
6. BESONDERHEITEN BEI POLYGONEN	15
7. DISPLAY LISTS.....	17
7.1 GLGENLISTS [1,SEITE 58].....	17
7.2 GLNEWLIST [1,SEITE 56].....	18
7.3 DEFINITION EINER DISPLAY LIST	18
7.4 AUFRUF EINER DISPLAY LIST	19
7.5 EIN BEISPIEL FÜR DISPLAY LISTS	19
8. EINFACHE TRANSFORMATIONEN.....	21
8.1 „LÖSCHEN“ DER AKTUELLEN MATRIX [1,SEITE 109].....	21
8.2 SKALIERUNG [1,SEITE 110]	22
8.3 TRANSLATION [1,SEITE 110]	22
8.4 ROTATION [1,SEITE 110]	22
8.5 DIE REIHENFOLGE DER TRANSFORMATIONEN	22
8.6 DIE WIRKUNG VON TRANSFORMATIONEN	23
8.7 EIN BEISPIEL ZU TRANSFORMATIONEN	23
9. PROJEKTION	24
10. DER MATRIZENSTAPEL VON OPENGL.....	25
11. ERGÄNZUNGEN UND AUSBLICKE.....	26
QUELLENVERZEICHNIS.....	27

Über diesen Reader

Dieser Reader ist als Einführung in die grundlegenden Konzepte der Graphik-schnittstelle OpenGL gedacht. Dabei wurde vor allem auf anwendungsorientierte Aspekte Wert gelegt. Aufgrund des immensen Umfangs von OpenGL ist es hier nur möglich, einen Ausschnitt seiner gesamten Funktionalität darzustellen. Die Beschreibung von fortgeschrittenen Funktionen und Eigenschaften findet sich in den im Anhang dieses Dokuments aufgelisteten Sekundärquellen.

Da auch eine Diskussion der einer Graphik-schnittstelle zugrunde liegenden mathematischen und informatischen Verfahren den angemessenen Rahmen sprengen würde, seien ausreichende Grundkenntnisse im Bereich Computergraphik vorausgesetzt. Des weiteren sind für den Leser Kenntnisse einer Programmiersprache, vorzugsweise C oder C++, von großem Vorteil.

Dieser Reader ist - soweit das in der gebotenen Kürze überhaupt möglich ist - nach didaktischen Gesichtspunkten aufgebaut und sollte daher auch in der gegebenen Reihenfolge durchgelesen werden. In jedem Abschnitt befindet sich auch ein Beispielprogramm oder zumindest ein Code-Fragment, das das Verstehen erleichtern sollte.

1. OpenGL - eine moderne Graphik-schnittstelle

In den letzten Jahren hat die rasante Entwicklung der Computertechnologie einer immer größeren Zahl von Anwendern den Zugang zu schnellen Rechnern ermöglicht. Rechnern, die auch schnell genug sind, um für die graphische Datenverarbeitung geeignet zu sein. Dies hat unter anderem natürlich auch zur Folge, daß die Nachfrage nach entsprechenden Anwendungsprogrammen stark gestiegen ist. Die Anwendungsprogrammierer ihrerseits sollen nun von der oft komplexen Aufgabe befreit werden, die Methoden der Computergraphik algorithmisch zu erfassen und zu implementieren. Aus diesem Grund gibt es schon seit einiger Zeit Bemühungen um einheitliche Graphik-schnittstellen (APIs) für Programmierer [1,Seite 1ff]. Als einer der erfolgreichsten und langlebigsten Schnittstellen-Standards ragt das 1992 vorgestellte *OpenGL* dabei aus der Masse seiner Konkurrenten heraus.



Abb. 1-1: Das OpenGL-Logo

OpenGL steht für "Open Graphics Library" - und das trifft genau den Charakter dieser Schnittstelle. Die Anwendung von OpenGL ist nicht nur unabhängig von Betriebssystem oder Rechner-typ, sie ist sogar noch nicht einmal an eine bestimmte Programmiersprache gebunden. Vielmehr handelt es sich bei OpenGL um einen recht abstrakten Schnittstellen-Standard. Er beinhaltet hauptsächlich die Definition der API sowie einige Aspekte des Laufzeitverhaltens - z.B. das Client-Server-Prinzip [1,Seite 15] - einer konkreten Implementierung. (Weitere Informationen zu den Grundlagen von OpenGL finden sich beispielsweise in "OpenGL

OpenGL ist eine plattformunabhängige Graphik-schnittstelle.

Overview” [2].)

Das bedeutet natürlich unter anderem, daß sich reale OpenGL-Implementierungen vom Aufbau her stark voneinander unterscheiden können. Daher rührt auch die Tatsache, daß bestehende Programme für verschiedene OpenGL-Implementierungen in aller Regel neu kompiliert werden müssen. Es besteht hier also keine Binär-, sondern nur eine Quellcode-Portabilität.

In der Tat existieren Implementierungen für die verschiedensten Betriebssysteme:

- Microsoft Windows [1,Seite 2][5]
- Linux [3]
- Solaris [3]
- MacOS [3]
- BeOS [6]
- (...)

für unterschiedliche Rechnertypen:

- Intel-PCs
- Sun Workstations
- Silicon Graphics Workstations
- (...)

und verschiedene Programmiersprachen:

- C/C++ [2]
- Fortran [2]
- Ada [2]
- Java [2]
- Perl [2]
- Delphi [7]
- (...)

Die meisten OpenGL-Implementierungen sind kostenlos verfügbar, dazu zählen unter anderem die zu Microsoft Windows gehörigen [1,Seite 2] und die Freeware-Library “Mesa” [3].

Ein weitere Besonderheit von OpenGL ist die Tatsache, daß seine Architektur von Anfang an auf die Unterstützung von Graphik-Beschleunigerhardware ausgelegt war. Schon die ersten Versionen, die vor allem für die Verwendung auf Rechnern von Silicon Graphics konzipiert waren, arbeiteten eng mit vorhandener Graphikhardware zusammen. Inzwischen gibt es auch im PC-Bereich oder beispielsweise für den Apple Macintosh eine breite Palette von 3D-Hardware, für die spezielle OpenGL-Implementierungen existieren [8].

OpenGLs großer Vorteil ist dabei, daß die Aktivierung von Hardware-Funktionalität weitestgehend transparent für den Programmierer stattfindet. Vielmehr wird von einer korrekten OpenGL-Implementierung verlangt, daß sie entsprechende Vorgänge intern automatisch tätigt. Dies steht im Gegensatz zu vielen Konkurrenz-APIs, bei denen vom Anwendungsentwickler verlangt wird,

die Hardware-Eigenschaften explizit zu erfragen und auszuwählen.

2. Allgemeine Hinweise zu OpenGLs API

Im Folgenden soll vorrangig OpenGLs Schnittstelle für die Sprachen C und C++ betrachtet werden, für die auch die meisten Implementierungen existieren. OpenGL-Varianten für andere Programmiersprachen sind aber in aller Regel sehr ähnlich aufgebaut.

Für alle Elemente (Bezeichner, Konstanten, Funktionen etc.) der Schnittstelle gibt es gewisse Grundregeln, die hier erläutert werden:

2.1 Datentypen

Um eine möglichst umfassende Unabhängigkeit von einer bestimmten Programmiersprache zu erreichen, werden in OpenGL nicht die Standard-Datentypen von C/C++ verwendet, sondern ausschließlich die folgenden:

OpenGL verwendet eigene Datentypen für die Deklaration von Funktionsparametern und –rückgabewerten.

Datentyp	OpenGL	Kürzel	(entspricht in ANSI C)
8 Bit Integer ohne Vz.	GLubyte	ub	(unsigned char)
„Boolean“	GLboolean	ub	(unsigned char)
8 Bit Integer mit Vz.	GLbyte	b	(signed char)
16 Bit Integer mit Vz.	GLshort	s	(short)
32 Bit Integer mit Vz.	GLint	i	(int)
32 Bit Fließkomma	GLfloat	f	(float)
64 Bit Fließkomma	GLdouble	d	(double)

Dies ist nur eine Auswahl der wichtigsten OpenGL-Datentypen. Eine vollständige Auflistung ist [1,Seite 21] zu entnehmen.

Korrekterweise sollte also eine Variablendeklaration zur Verwendung mit OpenGL beispielsweise so aussehen:

```
GLfloat winkel = 0.0;
```

2.2 Funktionen

Funktionsbezeichner der OpenGL-Schnittstelle beginnen immer mit den zwei Buchstaben „gl“, gefolgt von einem Großbuchstaben.

OpenGL-Funktionen beginnen stets mit den Buchstaben „gl“.

```
glVertex      glColor      glClear      glTranslate
```

Selbstredend werden von OpenGL-Funktionen als Argumente und Rückgabewerte nur die oben angegebenen Datentypen verwendet. Bei den Argumenten ist auch noch auf folgendes zu achten: Viele Funktionen unterstützen mehrere Arten der Parameterübergabe (z.B. als Parameterliste oder als Pointer auf einen Parametervektor). Da C das Überladen von Funktionen nicht kennt, ist für jede mögliche Kombination von Parametern eine eigene Funktion vorhanden. Zur Unterscheidung der Funktionen wird der benötigte Parametertyp als Buchstabenkombination an den Funktionsnamen angehängt [1,Seite 22ff]. Dazu dienen die in 2.1 angege-

benen Kürzel für die Datentypen. Zusätzlich gibt eine Ziffer die Anzahl der Parameter an. So verlangen

Art und Zahl der Parameter sind in den Funktionsnamen kodiert.

```
GLvoid glColor3f(GLfloat, GLfloat, GLfloat); [1,Seite 137]
```

```
GLvoid glColor4f(GLfloat, GLfloat, GLfloat, GLfloat);
```

drei bzw. vier Fließkomma-Variablen als Parameter. Bei Funktionen mit konstanter Parameterzahl wird nur der Variablentyp an den Namen angehängt:

```
GLvoid glRecti(GLint, GLint, GLint, GLint);
```

```
GLvoid glRectf(GLfloat, GLfloat, GLfloat, GLfloat); [1,Seite 85]
```

Soll ein Pointer auf einen Vektor als Parameter übergeben werden, so wird dies durch ein angehängtes "v" ausgedrückt:

```
glVertex4fv(const GLfloat *); [1,Seite 78]
```

Diese Funktion arbeitet mit einem Zeiger vom Typ

```
GLfloat array[4] = {0.0, 1.4, -0.4, 1.0};
```

2.3 Konstanten

Konstanten von OpenGL beginnen stets mit der Zeichenfolge "GL_" und werden weiterhin mit Großbuchstaben fortgesetzt. Üblicherweise werden Leerzeichen im Namen durch einen Unterstrich ersetzt.

```
GL_TRUE      GL_FALSE      GL_FRONT      GL_COLOR_BUFFER_BIT
```

2.4 Headerdateien

Bei der Erstellung eines OpenGL-Programms in C oder C++ ist es stets nötig, einige Header-Dateien mit einzubinden. Dabei gibt es für OpenGL und die GLU- und GLUT-Bibliotheken (s.u.) je eine eigene Datei. Die entsprechenden Headers befinden sich üblicherweise im Unterverzeichnis "GL" im entsprechenden Verzeichnis des Compilers. Die nötigen Anweisungen zu Beginn des Programms könnten also beispielsweise so aussehen:

Unter Windows ist auch noch die Angabe der Header-Datei windows.h notwendig.

```
#include <GL/gl.h>
#include <GL/glut.h>
#include <GL/glu.h>
```

Wenn OpenGL-Software unter dem Betriebssystem Microsoft Windows entwickelt wird, muß vor dem Einbinden der OpenGL-Headerdateien in aller Regel zusätzlich die Datei "windows.h" eingebunden werden, um Compiler-Fehlern vorzubeugen.

3. Zusatzbibliotheken und Anbindung an das Fenstersystem

Wie bereits beschrieben, stellt OpenGL eine systemunabhängige Schnittstelle für die Graphikprogrammierung zur Verfügung. Der Vorteil der Systemunabhängigkeit wird aber dadurch erkauft, daß in OpenGL selber eine Reihe notwendiger Funktionen fehlt. Diese werden als die Anbindung an das Fenstersystem bezeichnet und sind üblicherweise in einer eigenen - eben einer systemabhängigen - Schnittstelle zusammengefaßt [1,Seite 18-19]. Die Anbindung an das Fenstersystem ist beispielsweise für die Fensterverwaltung und die Behandlung von verschiedenen Ereignissen (z.B. Refresh oder Reshape) verantwortlich. Vor allem aber stellt eine solche Anbindung sogenannte "Kontexte" für OpenGL zur Verfügung. Erst mit Hilfe dieser Kontexte wird sichergestellt, daß die von OpenGL erzeugten Graphiken auch tatsächlich auf dem Bildschirm dargestellt werden.

Zum Betrieb von OpenGL ist eine sogenannte Fenstersystem-Anbindung notwendig.

3.1 Systemabhängige Anbindungen

Zur Anbindung an das Fenstersystem wird meistens eine vom verwendeten System abhängige Schnittstelle benutzt. Besonders wichtig sind hier die Anbindung an Microsoft Windows "WGL" [1,Seite 287ff] und die Anbindung an X-Windows "GLX" [1,Seite 302ff].

Der Vorteil einer systemabhängigen Anbindung liegt darin, daß sie die speziellen Fähigkeiten und Funktionen eines bestimmten Fenstersystems ausnutzen kann (z.B. Vollbildmodus oder Umschalten der Bildschirmauflösung). Daher ist eine solche Anbindung vor allem für umfangreichere und fortgeschrittene Projekte geeignet.

Der Nachteil liegt ebenso auf der Hand: Durch das Verwenden einer solchen Schnittstelle verliert die erstellte Software an Portabilität. Außerdem muß der Programmierer neben OpenGL-Kenntnissen auch Wissen über die Programmierung des entsprechenden Fenstersystems besitzen.

3.2 Systemunabhängige Anbindungen

Um die vorgenannten Nachteile systemabhängiger Anbindungen an das Fenstersystem umgehen zu können, existieren auch Bibliotheken, die diese Aufgabe wiederum systemunabhängig übernehmen. Es handelt sich dabei zum einen um die OpenGL Auxiliary Library "GLAUX" [1,Seite 279ff]. Da der Funktionsumfang und die Fähigkeiten der GLAUX aber sehr eingeschränkt sind, hat inzwischen das umfangreichere und leistungsfähigere OpenGL Utility Toolkit "GLUT" [4] sehr weite Verbreitung gefunden.

Die systemunabhängige GLUT-Library ist sehr weit verbreitet.

GLUT selber besitzt eine umfangreiche Schnittstelle, deren Funktionalität von der Fensterverwaltung über Font-Rendering bis hin zu Eingabe- und Timer-Ereignisbehandlung reicht. Daher würde selbst ein bloßer Überblick über GLUT den Rahmen dieses Readers bei weitem sprengen. Vielmehr soll hier nur eine kommentierte "Code-Schablone" vorgestellt werden, die für den Leser ein Ausgangspunkt für praktisches Arbeiten mit OpenGL sein kann:

```

// g l u t . c p p
//
// GLUT Code-Schablone.
// Dieser Code stellt den "Rahmen" für ein OpenGL-Programm dar.
// Diese Schablone verwendet die perspektivische Projektion.
//
// Von Jan Fischer
//

//----- Zu OpenGL gehörige Header-Dateien -----

#include <GL/gl.h>
#include <GL/glut.h>
#include <GL/glu.h>

//----- Funktionsprototypen -----

void display();
void reshape(int w, int h);
void keyboard(unsigned char key, int x, int y);
void initialize();
int main(int argc, char **argv);

//----- Globale Funktionen -----

// d i s p l a y
// Diese Routine wird aufgerufen, wenn ein Neuzeichnen des Fensterinhaltes
// notwendig wird. Hier sollten sich also die Befehle zur Darstellung der
// OpenGL-Szene befinden.
//
void display()
{
    // Lösche den Bildspeicher
    glClear(GL_COLOR_BUFFER_BIT);

    // <Hier eigene OpenGL-Befehle einsetzen>
    // <Beispielsweise kann man hier einige Primitive zeichnen lassen>

    // Zeichnen beenden
    glFinish();
}

// r e s h a p e
// Diese Routine wird aufgerufen, wenn die Fenstergröße verändert wurde.
// Wichtig: hier wird die "aspect ratio" wieder richtig eingestellt.
// Die Parameter w und h geben die neue Breite und Höhe des Fensters an.
//
void reshape(int w, int h)
{
    // Setze den OpenGL-Viewport (der sichtbare Bereich, in den gezeichnet wird)
    glViewport(0, 0, (GLint)w - 1, (GLint)h - 1);

    // Die aktuelle Projektionsmatrix verändern
    glMatrixMode(GL_PROJECTION);
    // Die Projektionsmatrix auf die Einheitsmatrix setzen
    glLoadIdentity();
    // Eine Projektionsmatrix für die perspektivische Projektion erstellen
    // Der dargestellte Blickwinkel umfasst einen Bereich von 65 Grad
    // Der Clipping-Bereich ist -1.0 bis 20.0
    gluPerspective(65.0, (GLfloat)w / (GLfloat)h, 1.0, 20.0);
}

// k e y b o a r d
// Diese Routine dient der Behandlung von Tastatureingaben.
// Vor allem stellt sie sicher, daß das laufende Programm beendet werden kann.
// Der Parameter key gibt die gedrückte Taste an, x und y geben die Maus-
// position zum Zeitpunkt des Tastendrucks an.
//
void keyboard(unsigned char key, int x, int y)
{
    // Wenn ESC oder q gedrückt wurde -> Programm beenden
    if(key == 27 || key == 'q') { exit(0); }
}

// i n i t i a l i z e
// Hier werden eventuell benötigte Benutzereinstellungen gesetzt.
// Kann beispielsweise zur Einrichtung von Display Lists verwendet werden.
//
void initialize()
{
    // Setze die Farbe zum Löschen des Bildspeichers
    glClearColor(0.0, 0.0, 0.0, 0.0);
}

```

```

// Setze OpenGL-Voreinstellungen (hier:Schattierungs-Modell)
glShadeModel(GL_SMOOTH);
}

//----- Das Hauptprogramm -----

int main(int argc, char **argv)
{
    // GLUT initialisieren
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA);

    // Öffne ein Fenster
    glutCreateWindow("GLUT Fenster");

    // Registrieren der Ereignisbehandlung (Callbacks)
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);

    // Initialisierungen durchführen
    initialize();

    // Starten der Haupt-Ereignisbehandlungsschleife von GLUT
    glutMainLoop();

    // Programmende
    return 0;
}

```

Quellcode 3-1: Codeschablone für GLUT (glut.cpp)

Im dargestellten Quellcode mußten einige OpenGL-Befehle vorweggenommen werden, die erst weiter unten erklärt werden. Eine detaillierte Beschreibung von GLUT findet der interessierte Leser in der “GLUT Specification” [4].

Neben GLUT und GLAUX existieren noch weitere systemunabhängige Anbindungen für OpenGL. Besonders hervorzuheben sind hierbei das Qt Toolkit [9], das Fast Light Toolkit (fltk) [10] und wxWindows [11]. Alle drei bieten nicht nur die Möglichkeit, OpenGL-Anwendungen für die verschiedensten Plattformen zu entwickeln, sondern können auch noch zur Erstellung von Benutzeroberflächen verwendet werden. Somit können mit Qt, fltk und wxWindows also „vollwertige“ OpenGL-Anwendungen erstellt werden. Allerdings sind diese Bibliotheken auch dementsprechend umfangreicher und schwerer anzuwenden, so daß sie sich erst für aufwendigere Projekte empfehlen.

Für plattformunabhängige OpenGL-Software, bei der nicht die Erstellung von Benutzeroberflächen, sondern Multimedia-Fähigkeiten und hohe Echtzeit-Leistung im Vordergrund steht, empfiehlt sich beispielsweise die Simple Direct-Media Layer Bibliothek [12].

3.3 Zusatzbibliotheken zu OpenGL

Neben den Anbindungen an das Fenstersystem werden bei der Arbeit mit OpenGL in aller Regel noch einige andere zusätzliche Bibliotheken verwendet. Diese dienen vor allem dazu, um auf Basis von OpenGL weitere nützliche Funktionen zur Verfügung zu stellen. Dies geschieht vor allem im Bereich der Modellierung (durch zusätzliche Primitive), der Erzeugung von Projektionsmatrizen (durch einfachere Funktionen als die von OpenGL) und in verschiedenen anderen Bereichen wie beispielsweise der Darstellung von Text (Font-Rendering).

Wichtige Zusatzbibliotheken sind vor allem die GLU, sowie auch GLAUX und

Funktionen der GLU-Zusatzbibliothek beginnen mit den Buchstaben „glu“.

GLUT, die hier also eine »Doppelrolle« spielen [1,Seite 273-286] [4].

In den Schnittstellen der Zusatzbibliotheken gelten üblicherweise die gleichen Regeln wie in Punkt 2 dargelegt. Der einzige Unterschied ist, daß Funktionsnamen nicht mit dem Kürzel “gl”, sondern mit dem entsprechenden Bibliotheksnamen eingeleitet werden.

`gluPerspective`

`glutInit`

`auxWireTorus`

4. Primitive in OpenGL

Nachdem bisher die Vorbedingungen für die Arbeit mit OpenGL diskutiert wurden, soll von nun an die Schnittstelle selber erläutert werden. Der erste und grundlegende Schritt zur Graphikdarstellung unter OpenGL ist die Definition sogenannter *Primitive*.

Unter Primitiven versteht man die *einfachsten Bestandteile* einer Szene, aus denen dann Schritt für Schritt mit Hilfe der hierarchischen Modellierung (s.u.) die darzustellenden Modelle aufgebaut werden [1,Seite 63ff]. OpenGL kennt die folgenden Arten von Primitiven [1,Seite 74-78]:

- Punkte
- Linien
- Polygone (dabei insbesondere auch Dreiecke und Vierecke)
- indirekt auch Kurven und gekrümmte Flächen per Bézierapproximation [1, Seite 97ff]

Zur Polygondefinition gibt es dabei eine ganze Reihe von Möglichkeiten.

Primitive – die Grundbausteine einer OpenGL-Szene – werden innerhalb einer `glBegin()` / `glEnd()` - Klammer definiert.

Allgemein geht man bei der Definition von Primitiven wie folgt vor:

1. Zunächst wird die Primitivdefinition durch den Befehl

```
glBegin(GLenum mode);
```

begonnen.

2. Im Anschluß werden die einzelnen Eckpunkte des Primitivs durch

```
glVertex*(...);
```

angegeben.

3. Die Definition des Primitives wird durch

```
glEnd();
```

abgeschlossen.

Ein Beispiel soll die Definition eines Dreiecks verdeutlichen:

```
glBegin(GL_TRIANGLES);
glVertex3d(0.0, 0.0, 0.0);
glVertex3d(0.5, 0.0, 0.0);
glVertex3d(0.25, 0.5, 0.0);
glEnd();
```

Quellcode 4-1: Beispiel für eine Primitivdefinition

4.1 glBegin [1,Seite 73]

Dieser Befehl leitet die Definition ein. Sein Parameter gibt an, was für ein Primitiv erzeugt werden soll. Er kann u.a. folgende Werte annehmen:

Bei glBegin() können verschiedene Primitiv-Modi angegeben werden.

Punkte:

- GL_POINTS: Jede Eckpunktdefinition erzeugt einen einzelnen Punkt

Linien:

- GL_LINES: Je zwei Eckpunkte definieren eine Linie
- GL_LINE_STRIP: Die Eckpunkte definieren einen zusammenhängenden Linienzug
- GL_LINE_LOOP: Wie GL_LINE_STRIP; der erste und letzte Punkt werden aber auch noch miteinander verbunden („Schleife“)

Polygone:

- GL_TRIANGLES: Jeweils drei Eckpunkte definieren ein Dreieck
- GL_TRIANGLE_STRIP: Jeder Eckpunkt nach den ersten beiden wird mit den jeweils vorhergehenden zu einem Dreieck verbunden
- GL_QUAD: Je vier Eckpunkte geben ein Viereck an
- GL_POLYGON: Alle Punkte in der Primitivdefinition ergeben ein Polygon. So definierte Polygone müssen konvex sein!

Die komplette Auflistung der möglichen Modi für die Primitivdefinition findet sich in [1]. Hier befindet sich ebenfalls eine grafische Darstellung der einzelnen Primitive.

4.2 glVertex* [1,Seite 78]

Dieser Befehl kann in sehr vielen Varianten aufgerufen werden. Man kann ihm Koordinaten für einen zweidimensional definierten Eckpunkt übergeben (glVertex2*), für einen dreidimensionalen (glVertex3*), und man kann sogar den Wert der homogenen Koordinate manipulieren (glVertex4*). Als Variablentypen akzeptiert er u.a. Integers (glVertex*i) und Fließkommazahlen (glVertex*f). Auch die Übergabe eines Arrays von Koordinaten ist möglich (glVertex*v). Gültige Beispiele für den Aufruf dieser Routine wären also:

Einzelne Eckpunkte werden mit der OpenGL-Funktion glVertex definiert.

```
GLint x, y, z, w;
GLdouble fx, fy, fz, fw;
GLfloat koordinaten2[2], koordinaten3[3];
(...)
glVertex2i(x, y); glVertex3i(x, y, z); glVertex3i(1, -4, 2);
glVertex4d(fx, fy, fz, fw); glVertex3d(1.0, 2.0, -2.4);
glVertex2fv(koordinaten2); glVertex3fv(koordinaten3);
```

Quellcode 4-2: Aufruf von glVertex

Ab OpenGL 1.1 ist es auch möglich, die verwendeten Eckpunkte in einem Array abgespeichert zu halten [1, Seite 313ff]. Diese sogenannten „Vertex Arrays“ haben vor allem den Vorteil, daß Eckpunkte, die in mehreren verschiedenen Primitiven vorkommen, nur einmal den Transformations- und Projektionsrechnungen unterzogen werden. Gerade bei großen Modellen ist dies mit einem nicht zu vernachlässigenden Geschwindigkeitsgewinn verbunden.

4.3 glEnd [1,Seite 73]

Der Befehl glEnd() veranlaßt die Darstellung eines Primitivs.

Dieser Befehl beendet eine Primitivdefinition. Wenn die Definition nicht innerhalb einer *Display List* stattfindet (s.u.), und wenn das Primitiv sichtbar ist, **wird es jetzt sofort gezeichnet!**

Wichtig: Innerhalb eines glBegin()/glEnd() - Paares haben die meisten OpenGL-Befehle keine Gültigkeit! Zugelassen sind nur Befehle zur Eckpunktdefinition (glVertex*), zum Einstellen der aktuellen Farbe (glColor*) und einige andere, die direkt mit der Primitivdefinition zu tun haben [1,Seite 74].

4.4 Beispiel zur Definition von Primitiven

Abschließend werden einige Beispiele zur Primitivdefinition vorgestellt. Sie befinden sich im Kontext der oben vorgestellten GLUT-Codeschablone:

```
// d i s p l a y
// Diese Routine wird aufgerufen, wenn ein Neuzeichnen des Fensterinhaltes
// notwendig wird. Hier sollten sich also die Befehle zur Darstellung der
// OpenGL-Szene befinden.
void display()
{
    // Setze die Farbe zum Löschen des Bildspeichers
    glClearColor(0.0, 0.0, 0.0, 0.0);
    // Lösche den Bildspeicher
    glClear(GL_COLOR_BUFFER_BIT);

    // Hier werden einige Punkte gezeichnet
    glBegin(GL_POINTS);
        glVertex3d(0.0, 0.0, 0.0);
        glVertex3d(0.5, 0.0, 0.0);
        glVertex3d(0.25, 0.5, 0.0);
    glEnd();

    // Das folgende Primitiv besteht aus einem Linienzug
    glBegin(GL_LINE_STRIP);
        glVertex3d(1.0, 1.2, 0.0);
        glVertex3d(1.1, 1.3, 0.0);
        glVertex3d(0.1, 1.7, 0.0);
        glVertex3d(1.9, 1.4, 0.0);
        glVertex3d(1.2, 5.3, 0.0);
    glEnd();

    // Das folgende Primitiv ist ein Polygon
    glBegin(GL_POLYGON);
        glVertex3d(2.0, 0.0, 0.0);
        glVertex3d(3.0, 1.5, 0.0);
        glVertex3d(1.5, 2.7, 0.0);
        glVertex3d(0.9, 2.7, 0.0);
        glVertex3d(0.0, 1.0, 0.0);
    glEnd();

    // Zeichnen beenden
    glFinish();
}
```

Quellcode 4-3: Primitivdefinitionen (primitiv.cpp)

5. Beeinflussung des Zeichenstils

Bisher haben wir Primitive nur definiert, ohne Einfluß auf ihre Darstellung zu nehmen. Selbstverständlich ist darüber hinaus auch möglich, den Zeichenstil von Punkten, Linien und Polygonen unter OpenGL einzustellen.

5.1 Die Punktausdehnung [1,Seite 79]

Standardmäßig werden Punkte in einer „ideal kleinen“ Ausdehnung auf dem Bildschirm dargestellt, d.h. als genau ein Pixel. Wenn man einzelne Punkte größer zeichnen will, geschieht dies einfach durch die Funktion:

```
void glPointSize(GLfloat size);
```

Der Parameter size gibt dabei die Punktgröße als ein Vielfaches eines Pixels an. Wenn Antialiasing [1,Seite 263] aktiviert ist, sind für size auch Dezimalbrüche zulässig, ansonsten wird einfach auf den nächsten ganzzahligen Wert gerundet.

Hinweis: glPointSize darf nicht innerhalb einer Primitivdefinition stehen! Vielmehr kann die aktuelle Punktausdehnung nur außerhalb von glBegin()/glEnd() - Paaren verändert werden. Andernfalls wird der Aufruf von glPointSize ignoriert.

Im folgenden wird die Veränderung der Punktausdehnung in der Praxis gezeigt:

```
// d i s p l a y
// Diese Routine wird aufgerufen, wenn ein Neuzeichnen des Fensterinhaltes
// notwendig wird. Hier sollten sich also die Befehle zur Darstellung der
// OpenGL-Szene befinden.
void display()
{
    // Setze die Farbe zum Löschen des Bildspeichers
    glClearColor(0.0, 0.0, 0.0, 0.0);
    // Lösche den Bildspeicher
    glClear(GL_COLOR_BUFFER_BIT);

    // Setze die Punktausdehnung auf 2 Pixel
    glPointSize(2.0);

    // Hier werden einige Punkte (Ausdehnung 2 Pixel) gezeichnet
    glBegin(GL_POINTS);
        glVertex3d(0.0, 0.0, 0.0);
        glVertex3d(0.5, 0.0, 0.0);
        glVertex3d(0.25, 0.5, 0.0);
    glEnd();

    // Setze die Punktausdehnung auf 3,6 Pixel
    glPointSize(3.6);

    // Hier werden einige Punkte (Ausdehnung 3,6 Pixel) gezeichnet
    glBegin(GL_POINTS);
        glVertex3d(0.5, 0.9, 0.0);
        glVertex3d(1.0, 0.9, 0.0);
        glVertex3d(0.85, 0.7, 0.0);
    glEnd();

    // Setze die Punktausdehnung auf 0,6 Pixel
    glPointSize(0.6);

    // Hier werden einige Punkte (Ausdehnung 0,6 Pixel) gezeichnet
    glBegin(GL_POINTS);
        glVertex3d(0.3, 0.6, 0.0);
        glVertex3d(0.3, 0.8, 0.0);
    glEnd();

    // Zeichnen beenden
    glFinish();
}
```

Quellcode 5-1: Verändern der Punktausdehnung (pointsize.cpp)

5.2 Die Linienbreite [1,Seite 80]

Analog zur Punktausdehnung kann auch die Breite zu zeichnender Linien unter OpenGL bestimmt werden. Der entsprechende Befehl lautet:

```
void glLineWidth(GLfloat width);
```

Für diesen Befehl gelten ebenfalls die unter 5.1 zu glPointSize gemachten Anmerkungen.

5.3 Linien- und Polygonmuster [1,Seite 80 und 84]

Der Vollständigkeit halber soll hier noch erwähnt werden, daß unter OpenGL Linien wie auch auch Polygone mit einem (Schwarzweiß-)Muster versehen werden können. Da diese Anforderung in der Praxis aber wohl eher selten auftritt, sei hier für den interessierten Leser einfach auf [1,Seite 80] bzw. [1,Seite 84] verwiesen.

5.4 Farbe [1,Seite 137]

OpenGL kennt beim Zeichnen von Primitiven stets die „aktuelle Farbe“.

Selbstverständlich ist es in OpenGL möglich, graphische Primitive in unterschiedlichen Farben zu zeichnen. Dabei spielt das Konzept der aktuellen Farbe eine wichtige Rolle. Mit Hilfe des Befehls glColor kann die aktuelle Farbe eingestellt werden. Alle danach definierten Primitive werden in dieser Farbe gezeichnet, solange bis eine neue aktuelle Farbe festgelegt wird. Der Befehl glColor wird üblicherweise in einer der folgenden Varianten aufgerufen:

```
void glColor3f(GLfloat red, GLfloat green, GLfloat blue);
void glColor3d(GLdouble red, GLdouble green, GLdouble blue);
void glColor3ub(GLubyte red, GLubyte green, GLubyte blue);
```

siehe je [1,Seite 137]

Dabei stehen die Parameter natürlich für den RGB-Vektor der gewünschten Farbe. Fließkomma-Werte werden dabei im Bereich [0.0; 1.0], Ganzzahlwerte im Bereich [0; 255] erwartet.

Jeder einzelne Eckpunkt kann eine eigene Farbe besitzen (nicht nur komplette Primitive).

Wichtig für das Verständnis des glColor-Befehls ist die Tatsache, daß in OpenGL potentiell jeder einzelne Eckpunkt eine eigene Farbe zugewiesen bekommen kann. Je nach der eingestellten Schattierung [1,Seite 178] wird dann je die Farbe des ersten Eckpunkts eines Primitivs angewendet, oder es kommt das sogenannte Gouraud-Shading zur Anwendung.

Es folgt ein Beispiel für farbige Primitive in OpenGL:

```
// d i s p l a y
// Diese Routine wird aufgerufen, wenn ein Neuzeichnen des Fensterinhaltes
// notwendig wird. Hier sollten sich also die Befehle zur Darstellung der
// OpenGL-Szene befinden.
void display()
{
    // Setze die Farbe zum Löschen des Bildspeichers
    glClearColor(0.0, 0.0, 0.0, 0.0);
    // Lösche den Bildspeicher
    glClear(GL_COLOR_BUFFER_BIT);

    // Stelle grün als aktuelle Farbe ein
    glColor3ub(0, 255, 0);

    // Zeichne ein Quadrat
    glBegin(GL_QUADS);
        glVertex3d(-0.8, -0.8, 0.0);
        glVertex3d(-0.2, -0.8, 0.0);
```

```

    glVertex3d(-0.2, -0.2, 0.0);
    glVertex3d(-0.8, -0.2, 0.0);
    glEnd();

    // Stelle hellgrau als aktuelle Farbe ein
    glColor3d(0.7, 0.7, 0.7);

    // Zeichne ein Quadrat
    glBegin(GL_QUADS);
    glVertex3d(0.0, 0.0, 0.0);
    glVertex3d(0.7, 0.0, 0.0);
    glVertex3d(0.7, 0.7, 0.0);
    glVertex3d(0.0, 0.7, 0.0);
    glEnd();

    // Zeichnen beenden
    glEnd();
}

```

Quellcode 5-2: Farbige Primitive (color.cpp)

6. Besonderheiten bei Polygonen

Bei der Definition von Polygonen wie in Abschnitt 4 beschrieben müssen noch einige Besonderheiten beachtet werden. Für ein Polygon müssen im Gegensatz zu einem Punkt oder einem Linienzug noch einige zusätzliche Eigenschaften festgelegt werden. Diese haben mit der Rückflächenunterdrückung (backface culling) und der Schattierung zu tun.

Die Vorderseite eines Polygons wird über den Uhrzeigersinn seiner Eckpunkte festgelegt.

Zum einen sollte man OpenGL mitteilen, welche die Vorder- und welche die Rückseite des Polygons ist. Zu diesem Zweck wird zunächst einmal mit Hilfe der Funktion

```
void glFrontFace(GLenum mode); [1,Seite 83]
```

festgelegt, was als Vorderseite eines Polygons betrachtet wird. Der Parameter mode kann hier die Werte GL_CCW und GL_CW annehmen. GL_CCW steht für “counterclockwise” und bedeutet, daß die Seite eines Polygons, auf der die Eckpunkte als gegen den Uhrzeigersinn definiert erscheinen, die Vorderseite ist. GL_CW ist die Abkürzung für “clockwise” und bewirkt eben das genaue Gegenteil, nämlich daß Punkte im Uhrzeigersinn die Vorderseite beschreiben.

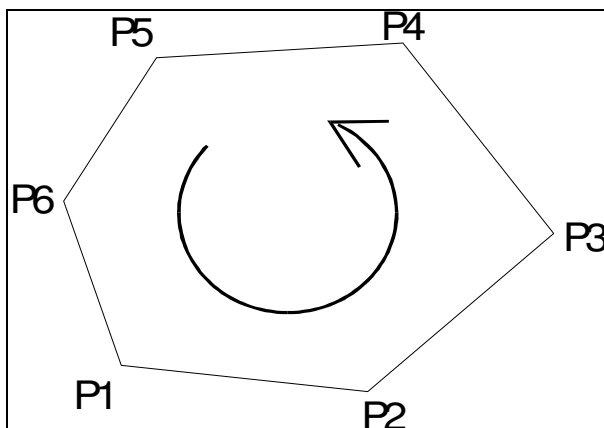


Abbildung 6-1: Die Vorderseite eines gegen den Uhrzeigersinn definierten Polygons

Wie in der Computergraphik üblich, nimmt OpenGL die Einstellung GL_CCW

als Standardwert an. Im Zusammenhang mit der Rückflächenunterdrückung ist es noch interessant, daß auch ihr Aktivierungszustand und ihr Umfang manipuliert werden können. Weitere Informationen dazu finden sich in [1,Seite 83f].

Die Eckpunkt-Normalen spielen bei der Beleuchtungsrechnung eine wichtige Rolle.

Die zweite wichtige zusätzliche Information, die für ein Polygon angegeben werden sollte, ist das Aussehen des Normalenvektors. Hierbei hat der Normalenvektor nicht die gewöhnliche Bedeutung. Ein Normalenvektor in OpenGL muß nicht notwendigerweise senkrecht auf dem Polygon stehen. Der Normalenvektor wird nicht automatisch erzeugt, und im Extremfall kann jeder einzelne Eckpunkt einen eigenen haben. Der Normalenvektor spielt hauptsächlich bei der Schattierung [1,Seite 141ff] und auch bei der Texturierung [1,Seite 183ff] eine große Rolle. Zur Angabe des Normalenvektors genügt es einfach, zu Beginn der Primitivdefinition die Funktion

```
void glNormal*(...); [1,Seite 86]
```

aufzurufen. Diese Funktion kann wie glVertex (s. Abschnitt 4.2) mit einer ganzen Reihe von Parametertypen aufgerufen werden. Beispielsweise kann die Variante

```
void glNormal3d(GLdouble, GLdouble, GLdouble);
```

verwendet werden. Die drei Parameter stehen dabei selbstredend für die x-, die y- und die z-Komponente des Normalenvektors.

Folgendes Beispiel soll nun eine komplette Polygon-Definition demonstrieren:

```
// d i s p l a y
// Diese Routine wird aufgerufen, wenn ein Neuzeichnen des Fensterinhaltes
// notwendig wird. Hier sollten sich also die Befehle zur Darstellung der
// OpenGL-Szene befinden.
void display()
{
    // Setze die Farbe zum Löschen des Bildspeichers
    glClearColor(0.0, 0.0, 0.0, 0.0);
    // Lösche den Bildspeicher
    glClear(GL_COLOR_BUFFER_BIT);

    // Folgenden Aufruf kann man auch weglassen, da GL_CCW in OpenGL sowieso die
    // Standardeinstellung ist
    glFrontFace(GL_CCW);

    // Beginn der Polygondefinition
    glBegin(GL_POLYGON);
    // Der Normalenvektor stehe senkrecht auf dem Polygon und
    // „zeige“ in Richtung Beobachter
    glNormal3d(0.0, 0.0, 1.0);

    // Die einzelnen Eckpunkte des Polygons gegen den Uhrzeigersinn definiert
    glVertex3d(0.5, 0.0, 0.0);
    glVertex3d(2.5, 1.0, 0.0);
    glVertex3d(0.0, 3.0, 0.0);
    glVertex3d(-2.5, 3.0, 0.0);
    glVertex3d(-1.0, -0.8, 0.0);
    // Ende der Polygondefinition
    glEnd();

    // Zeichnen beenden
    glFinish();
}
```

Quellcode 6-1: Eine vollständige Polygon-Definition (polygon.cpp)

7. Display Lists

Unsere bisher besprochenen “direkten” Primitiv-Definitionen werden – wie schon erwähnt – jeweils gleich im Anschluß gezeichnet (sofern sie sichtbar sind). In einem umfangreicheren OpenGL-Programm will man üblicherweise aber die Möglichkeit haben, ein solches Primitiv nicht nur einmal, sondern auch wiederholt darzustellen. Und das noch dazu nicht nur in der ursprünglichen Lage, sondern auch gedreht, skaliert, verschoben usw.

Display Lists dienen der Modularisierung und Effizienzsteigerung von OpenGL-Programmen.

Ein einfacher Ansatz zur Lösung dieses Problems wäre es natürlich, den gesamten `glBegin() / glEnd()` – Block etwa in eine Prozedur zu schreiben. Ein wiederholter Aufruf dieser Prozedur würde dann auch das Primitiv mehrmals auf den Bildschirm bringen. Diese Methode hat jedoch einen gravierenden Nachteil, der in der Architektur von OpenGL begründet liegt. Alle OpenGL-Befehle, die unser Programm (der “OpenGL-Client” [1,Seite 15]) aufruft, werden zunächst an den sogenannten “OpenGL-Server” übertragen, der dann das Rendering an sich durchführt [1,Seite 41]. Diese Übertragung selbst stellt aber schon einen gewissen Overhead dar. Noch dazu optimiert der OpenGL-Server oft intern ein Primitiv zur schnellstmöglichen Darstellung. Beispielsweise werden von vielen Implementierungen Polygone immer in Dreiecke zerlegt. Wird nun eine Primitiv-Definition wie oben beschrieben mehrere Male durchgeführt, finden auch die Übertragung an den OpenGL-Server und die internen Optimierungen mehrmals statt, was sich natürlich unter anderem auf die Darstellungsgeschwindigkeit negativ auswirkt.

Zur Lösung dieser Problematik gibt es in OpenGL den Mechanismus der sogenannten *Darstellungslisten (Display Lists)*. In einer Display List können nicht nur ein oder mehrere Primitive, sondern alle OpenGL-Befehle gespeichert werden.

Die Erstellung einer Display List geht in den folgenden Schritten vor sich:

1. Zunächst sollte man mit

```
glGenLists(1);
```

die nächste freie Nummer für eine Display List holen.

2. Die Display List selber wird mit

```
glNewList(GLuint list, GLenum mode);
```

begonnen.

3. Die OpenGL-Befehle, die jetzt folgen – auch der Aufruf anderer Display Lists (s.u.) – werden in der Display List gespeichert.

4. Der Befehl

```
glEndList(); [1,Seite 57]
```

beendet die Liste.

Display Lists werden in einer `glNewList() / glEndList()` – Klammer definiert.

7.1 `glGenLists` [1,Seite 58]

Unter OpenGL hat jede Display List eine eindeutige Nummer zwischen 1 und der maximalen Anzahl von Display Lists (implementierungsabhängig). Diese

Nummer wird später bei der Definition und vor allem auch beim Aufruf der Darstellungsliste wieder benötigt. Während es bei kleineren Programmen noch durchaus sinnvoll ist, feste Nummern für die einzelnen Listen zu wählen, ist das bei größeren Anwendungen nicht mehr praktikabel. Hier werden die Nummern der Display Lists typischerweise in Variablen gespeichert. Um keine Nummer doppelt zu verwenden, holt man sich daher von OpenGL mit Hilfe von

```
GLuint glGenLists(GLsizei range);
```

die nächste noch unbelegte Listennummer (der Rückgabewert der Funktion). Der Parameter range dient dazu, zusammenhängende Nummern für mehrere Display Lists zu “generieren”. Für unsere Zwecke reicht es aus, immer nur einzelne Display Lists zu definieren. Daher kann hier range immer gleich 1 gesetzt werden. Ein entsprechender Aufruf würde folgendermaßen aussehen:

```
GLuint my_list;
my_list = glGenLists(1);
```

7.2 glNewList [1,Seite 56]

Der „Name“ einer Display List ist eine vorzeichenlose ganze Zahl.

Mit glNewList wird eine neue Display List begonnen. Alle OpenGL-Befehle, die nach glNewList und vor dem nächsten glEndList stehen, werden vom OpenGL-Server unter der angegebenen Listennummer abgespeichert.

```
void glNewList(GLuint list, GLenum mode);
```

Die Funktion hat zwei Parameter. Der erste Parameter, list, ist die Nummer der neu zu definierenden Display List (s. 7.1). **Wichtig:** Wenn schon eine Darstellungsliste mit dieser Nummer definiert wurde, wird diese überschrieben!

Der zweite Parameter, mode, kann einen der beiden Werte GL_COMPILE und GL_COMPILE_AND_EXECUTE annehmen. Im ersten Fall werden die zwischen glNewList und glEndList stehenden OpenGL-Befehle nur abgespeichert. Im letzteren werden sie gleichzeitig auch ausgeführt.

7.3 Definition einer Display List

Die Definition einer Darstellungsliste wird auch als “Compilierung” bezeichnet. Während der Compilierung ist es zwar zulässig (und auch sehr sinnvoll), andere Display Lists *aufzurufen*. Es ist aber nicht erlaubt, innerhalb einer Compilierung wiederum eine neue Darstellungsliste zu definieren [1,Seite 57].

Für das Verständnis von Darstellungslisten ist es sehr wichtig, zwischen OpenGL-Befehlen und den Konstrukten der Programmiersprache zu unterscheiden. In einer Display List werden nur die Befehle gespeichert, die beim OpenGL-Server “ankommen”. Schleifen oder bedingte Ausdrücke werden beim Aufruf einer Display List nicht noch einmal ausgewertet. Betrachten wir beispielsweise das folgende Codestück:

```
int schleife, max = 5;
glNewList(1, GL_COMPILE);
glBegin(GL_POINTS);
for(schleife = 0; schleife < max; schleife++) {
    glVertex3d((GLdouble)schleife * 0.1, (GLdouble)schleife * 0.1, 0.0);
}
glEnd();
glEndList();
```

Quellcode 7-1: Eine Schleife in einer Display List

Der OpenGL-Server registriert hier nur die folgenden Befehle:

```
glVertex3d(0.0, 0.0, 0.0);
glVertex3d(0.1, 0.1, 0.0);
glVertex3d(0.2, 0.2, 0.0);
glVertex3d(0.3, 0.3, 0.0);
glVertex3d(0.4, 0.4, 0.0);
```

Ein Verändern der Variable max hätte also beim Aufruf der Display List mit `glCallList(1)` keinerlei Einfluß auf die dargestellten Punkte.

7.4 Aufruf einer Display List

Eine einmal compilierte Display List kann nun jederzeit wieder aufgerufen werden. Dazu dient der Befehl

```
void glCallList(GLuint list); [1,Seite 58]
```

Beim Aufruf einer Display List werden alle in ihr gespeicherten OpenGL-Befehle vom OpenGL-Server so ausgeführt, als ob sie einzeln explizit aufgerufen würden.

Eine Besonderheit, die in großem Maße zur Leistungsfähigkeit, aber auch zur Übersichtlichkeit von OpenGL-Programmen beiträgt, ist die Möglichkeit, innerhalb einer Display List wiederum Display Lists aufzurufen. Das folgende ist also in OpenGL gültig:

Innerhalb einer Display List können andere Display Lists aufgerufen werden.

```
// Display List für eine Linie
glNewList(1, GL_COMPILE);
glBegin(GL_LINES);
    glVertex3d(-0.5, -0.5, 0.0);
    glVertex3d(0.5, -0.5, 0.0);
glEnd();
glEndList();

// Display List für eine Linie mit einem Punkt darüber
glNewList(2, GL_COMPILE);
glCallList(1);
glBegin(GL_POINTS);
    glVertex3d(0.0, 0.0, 0.0);
glEnd();
glEndList();
```

Quellcode 7-2: verschachtelte Display Lists

Diese Möglichkeit ist vor allem im Rahmen der “hierarchischen Modellierung” von unschätzbarem Wert. Wie oben schon erwähnt, werden Modelle in OpenGL dadurch aufgebaut, daß man einfache Primitive wiederholt aneinander setzt. Ganze Szenen werden wiederum dadurch aufgebaut, daß man die einzelnen Modelle (unter Umständen mehrmals) positioniert. Beide Vorgänge sind natürlich mit Hilfe verschachtelter Darstellungslisten einfach und übersichtlich, aber auch höchst effizient durchzuführen.

Es ist darauf zu achten, daß es für Display Lists eine maximale Schachtelungstiefe gibt. Diese ist implementierungsabhängig, laut OpenGL-Standard aber mindestens 64.

7.5 Ein Beispiel für Display Lists

Der folgende kommentierte Quellcode demonstriert die Compilierung und den Aufruf mehrerer Display Lists. Er befindet sich wieder im Kontext der oben vorgestellten GLUT-Codeschablone. Dabei findet die Definition der Darstellungslisten selbstredend nicht in der Zeichen-Routine statt, sondern in einer eigenen Funktion namens `initialize`.

```

//----- Globale Variablen -----
GLuint dlist1, dlist2, dlist3; // Die Nummern unserer Display Lists

//----- Globale Funktionen -----

// d i s p l a y
void display()
{
    // Setze die Farbe zum Loeschen des Bildspeichers
    glClearColor(0.0, 0.0, 0.0, 0.0);
    // Lösche den Bildspeicher
    glClear(GL_COLOR_BUFFER_BIT);

    // Durchkreuztes Quadrat zeichnen
    glCallList(dlist2);
    // Polygon zeichnen
    glCallList(dlist3);

    glFinish();
}

// i n i t i a l i z e
// Hier werden eventuell benötigte Benutzereinstellungen gesetzt.
// Kann beispielsweise zur Einrichtung von Display Lists verwendet werden.
void initialize()
{
    // Nummern fuer die Display Lists holen
    dlist1 = glGenLists(1);
    dlist2 = glGenLists(1);
    dlist3 = glGenLists(1);

    // Display List 1 definieren
    // Display List 1 enthaelt ein durchkreuztes Quadrat
    glNewList(dlist1, GL_COMPILE);
    // Das Quadrat
    glBegin(GL_LINE_LOOP);
        glVertex3d(0.0, 0.0, 0.0);
        glVertex3d(0.5, 0.0, 0.0);
        glVertex3d(0.5, 0.5, 0.0);
        glVertex3d(0.0, 0.5, 0.0);
    glEnd();
    // Das Kreuz
    glBegin(GL_LINES);
        glVertex3d(0.0, 0.0, 0.0);
        glVertex3d(0.5, 0.5, 0.0);
        glVertex3d(0.5, 0.0, 0.0);
        glVertex3d(0.0, 0.5, 0.0);
    glEnd();
    glEndList();

    // Display List 2 definieren
    // Display List 2 enthaelt Display List 1 mit einem zusaetzlichen Kreuz
    glNewList(dlist2, GL_COMPILE);
    // Neue Linienbreite setzen und Display List 1 aufrufen
    glLineWidth(2);
    glCallList(dlist1);

    // Neue Linienparameter setzen und Kreuz definieren
    glLineWidth(1);
    glLineStipple(1, 0xAAAA);
    glBegin(GL_LINES);
        glVertex3d(0.25, 0.0, 0.0);
        glVertex3d(0.25, 0.5, 0.0);
        glVertex3d(0.0, 0.25, 0.0);
        glVertex3d(0.5, 0.25, 0.0);
    glEnd();
    glEndList();

    // Display List 3 definieren
    // Display List 3 enthaelt ein Polygon
    glNewList(dlist3, GL_COMPILE);
    // Parameter fuer backface culling setzen
    glCullFace(GL_BACK);
    glFrontFace(GL_CCW);

    // Polygon definieren
    glBegin(GL_POLYGON);
        glNormal3d(0.0, 0.0, 1.0);
        glVertex3d(0.6, 0.6, 0.0);
        glVertex3d(0.8, 0.7, 0.0);
        glVertex3d(0.7, 0.8, 0.0);
}

```

```

    glVertex3d(0.4, 0.6, 0.0);
    glEnd();
    glEndList();
}

```

Quellcode 7-3: Beispiel für Display Lists (dlists.cpp)

8. Einfache Transformationen

Display Lists machen für sich genommen natürlich noch recht wenig Sinn. Denn die Möglichkeit, ein Primitiv mehrmals identisch an ein und der selben Stelle zu zeichnen, bringt noch keinen großen Nutzen. Erst im Zusammenspiel mit Transformationen zeigt sich der große Vorteil von Display Lists.

Unter Transformationen versteht man Operationen, die die Größe und vor allem die Lage eines Primitivs beeinflussen können. Im einzelnen kennt OpenGL die folgenden Transformationen:

- skalieren
- verschieben (Translation)
- drehen (Rotation)

Transformationen verändern Aussehen und Lage eines graphischen Primitivs.

Um die Wirkungsweise von Transformationen unter OpenGL zu verstehen, ist es wichtig zu wissen, daß hier – wie allgemein üblich – *Transformationsmatrizen* zur Anwendung kommen. Das heißt für jede einzelne Transformation wird eine entsprechende Translations-, Skalierungs- oder Rotationsmatrix erzeugt. Schließlich entsteht aus diesen eine Gesamt-Transformationsmatrix, die dann mit jedem Eckpunkt eines Primitivs multipliziert wird. Das genaue Aussehen dieser Transformationsmatrizen ist in [1,Seite 90-93] dargestellt.

Bevor in OpenGL mit Transformationsmatrizen überhaupt gearbeitet werden kann, muß mit dem Aufruf

```
glMatrixMode(GL_MODELVIEW); [1,Seite 108]
```

in dem Transformationsmatrix-Modus geschaltet werden.

Wir wollen zunächst die Erzeugung einzelner Transformationsmatrizen betrachten:

8.1 „Löschen“ der aktuellen Matrix [1,Seite 109]

Bevor man mit einer Kombination der unten stehenden OpenGL-Aufrufe die genauen Transformationen für ein Primitiv festlegt, sollte man sicherstellen, daß dieses zunächst einmal unverändert bleibt. Dazu dient der Aufruf von

```
void glLoadIdentity();
```

Dieser Befehl überschreibt die *aktuelle Matrix* (die Matrix, in der die Transformationen zusammengefaßt sind) mit der Einheitsmatrix. Die Multiplikation mit der Einheitsmatrix läßt einen Punkt bzw. einen Vektor bekanntermaßen unverändert und entspricht somit einer identischen Abbildung – daher auch LoadIdentity.

Die Einheitsmatrix beschreibt die identische Abbildung.

8.2 Skalierung [1,Seite 110]

Die Skalierung bewirkt die Dehnung bzw. Schrumpfung von Primitiven.

Um die Größe bzw. die Proportionen eines Primitivs zu ändern, genügt ein Aufruf von

```
void glScaled(GLdouble x, GLdouble y, GLdouble z);
```

(Selbstverständlich gibt es diesen Befehl auch in Versionen mit Integer-, Float- usw. Parametern. Das gilt auch für die noch folgenden Matrixbefehle.)

Die drei Parameter geben dabei die Skalierungsfaktoren in x-, y- und z-Richtung an. Ein negativer Skalierungsfaktor bewirkt dabei natürlich eine Spiegelung. Intern erzeugt dieser Befehl eine Skalierungs-Matrix und multipliziert diese von rechts an die aktuelle Matrix heran (die Bedeutung dieser Tatsache wird weiter unten erläutert).

8.3 Translation [1,Seite 110]

Primitive können auch verschoben werden.

Mit Hilfe von

```
void glTranslated(GLdouble x, GLdouble y, GLdouble z);
```

erwirkt man eine Verschiebung um die angegebenen Parameter in x-, y- und z-Richtung. Wie üblich erzeugt auch dieser Befehl zunächst eine (Translations-)Matrix und multipliziert diese von rechts an die aktuelle Matrix heran.

8.4 Rotation [1,Seite 110]

Der Befehl glRotate bewirkt die Drehung von graphischen Primitiven.

Die letzte der einfachen Transformationen stellt die Rotation um eine Achse dar, die durch den Aufruf von

```
void glRotated(GLdouble angle,
               GLdouble x, GLdouble y, GLdouble z);
```

durchgeführt wird. Dabei gibt angle den Rotationswinkel an, und x, y und z definieren die Achse, um die rotiert werden soll. Der Aufruf

```
glRotated(90.0, 0.0, 1.0, 0.0);
```

dreht ein Primitiv also 90 Grad um die y-Achse. Auch hier gilt: glRotate erzeugt lediglich eine Rotationsmatrix, die von rechts an die aktuelle Matrix heranmultipliziert wird.

8.5 Die Reihenfolge der Transformationen

Wie aus der Linearen Algebra bekannt, stellt ein Produkt von Matrizen eine Verkettung linearer Abbildungen dar. Dabei ist zu beachten, daß die Abbildungen in diesem Produkt aber von rechts nach links wirken. D.h. diejenige Matrix, die am weitesten rechts steht, ist zuerst von Bedeutung. Für die konkrete Anwendung von Transformationsmatrizen in OpenGL bedeutet diese Tatsache folgendes:

Transformationen wirken in der umgekehrten Reihenfolge ihrer Festlegung.

Wie oben mehrmals erwähnt, multiplizieren die Transformations-Befehle ihre Matrizen von rechts an die aktuelle Matrix heran. Das bedeutet, sie stellen ihre Transformation vor (!) die bisher angegebenen Transformationen.

In einer zusammenhängenden Folge von glScale-, glTranslate- und glRotate – Aufrufen wirken die zugehörigen Transformationen in umgekehrter Reihenfolge!

Betrachten wir beispielsweise dieses Codestück:

```
glLoadIdentity();
glTranslated(5.0, 2.0, 15.0);
glRotated(37.2, 1.0, 0.0, 0.0);
glScaled(2.0, 2.0, 2.0);
```

Quellcode 8-1: Transformationen-Reihenfolge

Diese Code würde nacheinander das folgende bewirken:

1. Verdoppele die Größe des Primitivs in alle Richtungen
2. Drehe das Primitiv 37,2 Grad um die x-Achse
3. Verschiebe das Primitiv um den Vektor (5 2 15)

8.6 Die Wirkung von Transformationen

Direkt bevor ein Primitiv gezeichnet wird – das kann unmittelbar oder in einer Display List geschehen –, werden seine Eckpunkte mit der aktuellen Transformationsmatrix multipliziert. Somit können Transformationen natürlich auch für mehrere Primitive gelten, indem zwischen verschiedenen Primitivdefinitionen die aktuelle Matrix unverändert bleibt.

Andernfalls müßte man zwischen verschiedenen Primitiven die aktuelle Matrix mit `glLoadIdentity` zurücksetzen und wieder neu definieren. Da dieses Verfahren natürlich gerade bei „hierarchischen Transformationen“ an seine Grenzen stößt, kommt hier typischerweise der OpenGL-eigene *Matrizenstapel* zur Anwendung (s.u.).

OpenGL speichert die aktuelle Transformations-Matrix, die also auch für mehrere Primitive gelten kann.

Für mathematisch versierte OpenGL-Anwender ist es eventuell interessant, daß man die aktuelle Transformationsmatrix auch von Hand manipulieren kann. Dazu dient der Befehl `glLoadMatrix` [1,Seite 109]. Durch ihn kann man die aktuelle Transformationsmatrix mit Werten aus einem selbsterstellten Array überschreiben.

8.7 Ein Beispiel zu Transformationen

Folgendes Beispiel demonstriert einfache Transformationen im Zusammenspiel mit Display Lists in OpenGL:

```
// d i s p l a y
void display()
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    // Lösche den Bildspeicher
    glClear(GL_COLOR_BUFFER_BIT);

    // Matrixmodus Modellierung einstellen
    glMatrixMode(GL_MODELVIEW);

    // Stelle das 1. Polygon dar
    // Groesse 1:1, links unten
    glLoadIdentity();
    glTranslated(-0.5, -0.5, 0.0);
    glCallList(1);

    // Stelle das 2. Polygon dar
    // Groesse halbiert, rechts oben, um 45 Grad gedreht
    glLoadIdentity();
    glTranslated(0.5, 0.5, 0.0);
    glRotated(45.0, 0.0, 0.0, 1.0);
    glScaled(0.5, 0.5, 0.5);
    glCallList(1);

    // Stelle das 3. Polygon dar
    // Rechts unten, gespiegelt
    glLoadIdentity();
```

```

glTranslated(0.5, -0.5, 0.0);
glScaled(-1.0, -1.0, 0.0);
glCallList(1);

glFinish();
}

// i n i t i a l i z e
void initialize()
{
// Definiere die Display List fuer ein Polygon
// Da sowieso nur eine Display List definiert wird, wird die feste
// Nummer 1 verwendet
glNewList(1, GL_COMPILE);
glBegin(GL_POLYGON);
glNormal3d(0.0, 0.0, 1.0);
glVertex3d(-0.2, -0.2, 0.0);
glVertex3d(-0.1, 0.3, 0.0);
glVertex3d(0.2, 0.7, 0.0);
glVertex3d(0.0, 0.8, 0.0);
glVertex3d(-0.5, 0.1, 0.0);
glEnd();
glEndList();
}

```

Quellcode 8-2: Transformationen (transform.cpp)

9. Projektion

Bisher haben wir in den Beispielen immer nur zweidimensionale Primitive und Transformationen betrachtet. Leider ist hier nicht genügend Platz, um alle Aspekte dreidimensionaler Graphik zu behandeln.

Ein besonders wichtiger Vorgang jedoch soll hier nicht ausgespart werden: *die perspektivische Projektion*. Um für eine OpenGL-Szene die perspektivische Projektion zu aktivieren, verwendet man die folgenden Aufrufe:

```

glMatrixMode(GL_PROJECTION);           [1,Seite 108]
glLoadIdentity();                       [1,Seite 109]
gluPerspective(fovy, aspect, zNear, zFar); [1,Seite 275]

```

Der letzte Befehl, gluPerspective, verwendet dabei Variablen vom Typ GLdouble als Parameter. Sie bedeuten im einzelnen:

fovy gibt den Blickwinkel in y-Richtung an
aspect gibt die „aspect ratio“ an, als Ergebnis der Berechnung Breite (x) durch Höhe (y)
zNear gibt die nahegelegene Clip-Ebene an
zFar gibt die entfernte Clip-Ebene an

Der Beobachter blickt üblicherweise entlang der negativen z-Achse.

Nach diesen Aufrufen werden alle folgenden Primitive perspektivisch projiziert. Dabei ist stets darauf zu achten, **daß OpenGL (anders als üblich) als Blickrichtung die negative z-Achse annimmt**. Soll ein Primitiv jetzt noch sichtbar sein, muß es also in den Bereich der negativen z-Achse verschoben werden. Genaugenommen werden nur noch Primitive gezeichnet, die sich **zwischen -zNear und -zFar** befinden.

Da der aspect-Parameter von gluPerspective von den Ausmaßen des Fensters abhängt, in dem die OpenGL-Szene dargestellt wird, sollte gluPerspective sinnvoll-

erweise im reshape-Callback von GLUT aufgerufen werden. (Das reshape-Callback wird dann aktiviert, wenn sich die Größe des GLUT-Fensters ändert.)

Neben gluPerspective gibt es in OpenGL noch weitere Möglichkeiten, um gültige und sinnvolle Projektionsmatrizen zu erzeugen. Dies sind beispielsweise die Funktionen glOrtho [1,Seite 111], glFrustum [1,Seite 112] und gluOrtho2D [1,Seite 275].

10. Der Matrizenstapel von OpenGL

Die bislang besprochenen Möglichkeiten, um die aktuellen Projektions- und Transformationsmatrizen zu verändern, reichen für die Modellierung komplexer Szenen nicht aus. Denn bei umfangreicheren Szenen gerät man häufig in die Situation, eine der Matrizen kurzzeitig verändern und dann wieder herstellen zu müssen. Besonders oft ist dies für die aktuelle Transformationsmatrix nötig, da auf diese Art und Weise die sogenannte “hierarchische Modellierung” verwirklicht wird. Dabei baut man zunächst möglichst allgemeine Transformationsmatrizen (z.B. die Kameraperspektive für die ganze Welt) auf. Später manipuliert man diese allgemeinen Matrizen dann für jedes einzelne Teilobjekt oder –primitiv. Für das jeweils nächste Teilobjekt muß die spezielle Änderung der Matrix für das vorige dann wieder rückgängig gemacht werden.

Um nun dieses temporäre (und hierarchische) Verändern der aktuellen Matrizen möglich zu machen, stellt OpenGL die sogenannten Matrizenstapel zur Verfügung. Auf diesen kann man Matrizen nach dem LIFO-Prinzip ablegen (push) und bei Bedarf wieder herunterholen (pop).

Für die hierarchische Modellierung gibt es in OpenGL den *Matrizenstapel*.

Es existieren sowohl für Transformations- wie auch für Projektionsmatrizen jeweils eigene Stapel. Zwischen diesen kann mit dem Befehl

```
void glMatrixMode(GLenum mode); [1,Seite 108]
```

umgeschaltet werden. Bekommt er die Konstante GL_MODELVIEW übergeben, so wird auf den Transformationsmatrizen-Stapel umgeschaltet. Um auf Projektionsmatrizen umzuschalten, gibt es den Parameter GL_PROJECTION.

Nach dem Aufruf von glMatrixMode beziehen sich alle Matrizenbefehle von OpenGL auf den eingestellten Modus (wie glLoadIdentity, glTranslate*, gluPerspective etc). Deswegen ist es äußerst wichtig, vor der Anwendung solcher Befehle immer auf den richtigen Modus umzuschalten.

Schließlich kann noch mit den Befehlen

```
void glPushMatrix(); [1,Seite 108]
```

```
void glPopMatrix(); [1,Seite 108]
```

die jeweils aktuelle Projektions- oder Transformationsmatrix auf den Stapel gepusht oder von ihm gepoppt werden.

11. Ergänzungen und Ausblicke

Leider ist es aufgrund der gebotenen Kürze in diesem Reader selbstredend nicht möglich, einen Überblick über alle wichtigen Konzepte von OpenGL zu bieten. Nicht behandelt wurden unter anderem die folgenden grundlegenden Themen:

- OpenGL-Zustände [1,Seite 47-54]
- Licht und Schattierung [1,Seite 141ff]
- Texturierung [1,Seite 183ff]
- sowie geometrische Modellierung, Animation usw.

Dem interessierten Leser seien daher zur vertiefenden Lektüre vor allem Ute Claussens Buch „Programmieren mit OpenGL“ [1], aber auch die OpenGL-Website [5], auf der sich unzählige wertvolle Informationen finden, ans Herz gelegt.

Quellenverzeichnis

- [1] Claussen, Ute. *Programmieren mit OpenGL*. Springer Verlag 1997
- [2] *OpenGL Overview* bei <http://www.opengl.org/developers/about/overview.html>
- [3] *The Mesa 3-D graphics library* bei <http://www.mesa3d.org>
- [4] *GLUT API, version 3* bei <http://trant.sgi.com/opengl/docs/Specs/glutspec3/spec3.html>
- [5] *OpenGL – The Industry’s Foundation for High Performance Graphics* bei <http://www.opengl.org>
- [6] *BeOS Specifications* bei <http://www.be.com/products/freebeos/beosspecs.html>
- [7] *Delphi3D* bei <http://www.delphi3d.net>
- [8] *OpenGL 3D accelerators* bei http://www.opengl.org/users/apps_hardware/accelerators.html
- [9] *Troll Tech* bei <http://www.troll.no>
- [10] *The FLTK Home Page* bei <http://www.fltk.org>
- [11] *wxWindows Home* bei <http://www.wxwindows.org>
- [12] *Simple DirectMedia Layer* bei <http://www.libsdl.org>