Eine Einführung in OpenGL

Inhaltsverzeichnis

1 Übersicht

1.1 Was ist OpenGL?

OpenGL ist eine plattformunabhängige Grafikbibliothek, die inzwischen zu einem weltweiten Standard für 3D-Grafikprogrammierung geworden ist. Sie wird sehr vielfältig verwendet: In der Industrie, in Forschungszentren, in der Spieleprogrammierung und in der Lehre.
Die Plattformunabhängigkeit wird dadurch erreicht, dass OpenGL tatsächlich nur die *Spezifikation* einer Bibliothek ist, für die verschiedenste Implementierungen existieren, teilweise direkt von Graphikkarten unterstützt[1]. Obwohl es auch z. B. Pascal-artige Varianten (etwa für die Verwendung mit Borland Delphi[2]) gibt, verwendet man meist eine C-artige (und damit auch Java-artige) Syntax. Dabei werden aber keine fortgeschrittenen Konzepte wie etwa die Zeigerarithmetik verwendet. Bis auf den Typ der Wahrheitswerte sehen die Programme weitgehend wie Ausschnitte aus Java-Programmen aus. In Beispielen finden sich gelegentlich auch Elemente der objektorientierten Erweiterung C++ von C, aber ebenfalls keine fortgeschrittenen.

1.2 Was leistet OpenGL?

Hier sind einige charakteristische Elemente:

- Es werden grundlegende geometrische Objekte bereitgestellt wie Geraden/Strecken, Polygone, Dreiecksnetze, Kugeln, Quader, Quadriken (Ellipsoide, Paraboloide, Hyperboloide), spezielle Kurven und Flächen.
- Dazu kommen die besprochen Transformationen (Streckung, Drehung, Spiegelung, Scherung), Projektionen (Parallel- und Zentralprojektion) sowie Kameradefinitionen.
- OpenGL unterstützt Bilddarstellung (Rendering) in hoher Qualität, Entfernen verdeckter Flächen, mehrfache Lichtquellen, Oberflächenmaterialien und -texturen, Farbverläufe und Nebel.
- Es gibt Datenstrukturen zur gepufferten Verwaltung von Objektsammlungen und zum Aufbau hierarchischer Modelle; schließlich können Objekte interaktiv ausgewählt werden.
- Auch Implementierungsdetails wie Pixelansteuerung, Anti-Alias, Bewegungsunschärfe, Tiefeneffekte und weiche Schatten werden unterstützt.

In OpenGL werden die Bereiche der *Bildsynthese* (Rendering) und der *Interaktion* (Eingabe und Fensteransteuerung) zunächst getrennt:

- OpenGL in engeren Sinn befasst sich nur mit der Synthese; alle zugehörigen Funktionen beginnen mit „`gl`“.
- Die Zusatzbibliothek GLU (OpenGL Utility Library) stellt reichhaltigere Zeichenprimitive zur Verfügung als OpenGL selbst, z. B. Kurven und Flächen und Funktionen zur Spezifikation von 3D-Szenerien. Alle GLU-Funktion beginnen mit „`glu`“.

[1] dann ist es natürlich schnell
[2] OO-Erweiterung von Pascal

- Eine weitere Zusatzbibliothek GLUT (OpenGL Utility Toolkit) behandelt die Interaktion. Sie stellt Funktionen zum Aufbau von Fenstern und zum Behandeln von Tastatur- und Mausereignissen zur Verfügung. Außerdem enthält sie einige rudimentäre Möglichkeiten zum Aufbauen grafischer Benutzeroberflächen. Schließlich definiert GLUT noch einige komplexere geometrische Objekte – Polyeder und die „klassische“ Grafik-Teekanne. Alle GLUT-Funktionen beginnen mit „`glut`“.

Meist unterscheidet man nicht so fein und bezeichnet mit „OpenGL“ das Gesamtpaket dieser drei Bibliotheken.

2 Erste Schritte mit OpenGL

2.1 Grundstrukturen

- Ein *Fenster* in OpenGL ist eine rechteckige Fläche auf einem physikalischen Darstellungsmedium, in das OpenGL Grafiken zeichnet. Häufig entspricht ein OpenGL-Fenster einem Fenster des Fenstermanagers im verwendeten Betriebssystem.

- Ein *Ereignis* in OpenGL tritt ein, wenn der Benutzer ein Eingabemedium betätigt. Für jede Art von Ereignis, die auftreten kann, muss das OpenGL-Programm eine *Behandlungsfunktion* (*callback function*) bereitstellen; der Aufruf wird dann automatisch von OpenGL veranlasst.

- Die *Hauptereignisschleife* von OpenGL prüft ständig, ob Änderungen oder Ereignisse stattgefunden haben und ruft ggf. die zugehörigen Behandler auf. Sie endet erst, wenn das gesamte Programm endet.

2.2 Beispiel: Ein einfaches Fenster

```
#include <GL/glut.h>              // einbinden von GLUT

void display (void) {             // ähnlich repaint in Java
  glClear(GL_COLOR_BUFFER_BIT);   // Fenster löschen
  glFlush();                      // erzwinge Neuanzeige
}

int main (int argc, char **argv) { // Kommandozeilenparameter
  glutInit(&argc, argv);           // initialisiere OpenGL
  glutCreateWindow("Beispiel: Fenster");     // Fenster erzeugen
  glutDisplayFunc(display);        // registiere obige Fkt. als
                                   // Behandler für Auffrischen
  glutMainLoop();                 // Hauptereignisschleife starten
  return 0;
}
```

Der Aufruf von `glutInit` muss stets vor allen weiteren Aufrufen von GLUT- bzw. OpenGL-Funktionen erfolgen:

```
void glutInit (int *argc, char **argv);
```

Die Funktion

```
int glutCreateWindow (char *name);
```

erzeugt ein Fenster mit Titel[3]. Die voreingestellte Größe ist 300×300 Pixel. GLUT nummeriert die Fenster durch; die Nummer des neu erstellten Fensters ist das Ergebnis von `glutCreateWindow`.
Als Nächstes wird OpenGL durch

```
void glutDisplayFunc (void (*func)(void));
```

mitgeteilt, welche Funktion beim Ereignis „Fenster soll aufgefrischt werden“ aufzurufen ist. Die Parametrisierung zeigt, dass *glutDisplayFunc* eine Funktion höherer Ordnung (wie in Haskell) ist: Sie hat als Parameter eine andere Funktion. Da Funktionen aber in C/C++ keine echt eigenständigen Größen sind, wird hier ersatzweise ein *Zeiger* auf eine Funktion (nämlich die Adresse des jeweiligen übersetzten Maschinencodes der Funktion) übergeben. Genau bedeutet also die Parametrisierung `void(*func)(void)`, dass der Parameter von `glutDisplayFunc`

- `func` heißt
- ein Zeiger ist (`*`)
- und auf eine Funktion vom Typ `void (void)`, d. h. ein Parameter und ein Ergebnis, jeweils vom Typ `void`, zeigen muss.

Beim Aufruf genügt es, den Namen der aktuellen Funktion anzugeben, der Adressoperator `&` ist nicht nötig.

[3]Strings werden in C als Zeiger auf das erste Zeichen, also als `char*`, übergeben

Die Hauptschleife

```
void glutMainLoop (void)
```

verhält sich in Pseudocode wie folgt:

```
while(1)// Endlosschleife
{if (Grafik geändert)
       rufe den DISPLAY-Behandler auf;
 if (Fenster verschoben oder in der Größe geändert)
       rufe den RESHAPE-Behandler auf;
 if(Tastatur- oder Mausereignis)
       rufe den KEYBOARD- oder MOUSE- Behandler auf;
 rufe den IDLE-Behandler auf;
}
```

Schließlich müssen wir noch die Funktion `display` besprechen. Zuerst wird

```
void glClear (GLbitfield mask);
```

aufgerufen. Der Parameter `mask`[4] gibt an, welche Puffer gelöscht werden sollen. Meist ist das nur der Bildpuffer (*frame buffer*), in dem das Pixelbild aufgebaut wird, das dann ins Fenster kopiert wird; er hat die spezielle Maske `GL_COLOR_BUFFER_BIT`. Beim Aufruf von `glClear` wird jedes Pixel der angesprochenen Puffer auf die aktuelle Löschfarbe (Schwarz per Voreinstellung) gesetzt. Man kann die Löschfarbe mit `glClearColor()` ändern.
Der Aufruf von

```
void glFlush(void)
```

veranlasst OpenGL, die momentanen Pufferinhalte auf das Fenster zu übertragen. Ein expliziter Aufruf dieser Funktion ist nur nötig, wenn nicht mit Doppelpufferung gearbeitet wird.

2.3 Beispiel: Tastaturereignisse

```
#include <GL/glut.h>
#include <stdio.h>
#include <iostream>           // damit exit definiert ist
using namespace std;

// void display (void) wie im vorigen Beispiel

void keyboard (unsigned char key, int x, int y)
{ if (key==27)                       // ESC
     exit(0);
   else printf ("Sie haben %c gedrückt.\n", key);
                                     // braucht <stdio.h>
}
```

[4] OpenGL definiert zur Sicherheit (Plattformunabhängigkeit) eigene Varianten der Grunddatentypen, wie z. B. `GLbitfield`, `GLfloat`, `GLdouble`, deren Namen alle mit „`GL`“ beginnen

```
int main(...)
{ // wie zuvor, nur nach dem Registrieren von display noch
  glutKeyboardFunc(Keyboard);
}
```

Die Keyboard-Funktion hat drei Argumente, die von OpenGL intern beim Aufruf übergeben werden: die gedrückte Taste sowie die Mauskoordinaten beim Aufrufzeitpunkt. Sie wird nur beim Drücken einzelner Tasten mit ASCII-codierten Zeichen aufgerufen. Für weitere – wie Pfeil- oder Funktionstasten – verwendet man die `glutSpecialFunc()` .

2.4 Beispiel: Fensteränderung

```
#include <GL/glut.h>
#include <iostream>           // damit exit definiert ist
using namespace std;

void display(void)
{ glClearColor (1.0,1.0,1.0,0.0);  //Löschfarbe setzen
  glClear(GL_COLOR_BUFFER_BIT);
  glFlush();
}

void keyboard (unsigned char key, int x, int y)
{ if (key==27) exit (0);
}

int main (int argc, char **argv)
{ glutInit(&argc, argv);
  glutInitWindowSize(500,500);
  glutInitWindowPosition(100,100);
  glutCreateWindow("Fensterln");
  //Rest wie gehabt
}
```

Die Funktion

```
void glClearColor (GLclampf red, green, blue, alpha);
```

setzt eine RGB-Farbe mit Transparenz `alpha` als Löschfarbe. Der Typ `GLclampf`[5] beschreibt das Intervall $[0.0, 1.0]$; die Farbanteile werden also nicht absolut in Binärwerten, sondern in Bruchteilen angegeben. Der verwendete Aufruf setzt die Farbe auf Weiß.
Für Änderungen am Fenster während des Programmlaufs lässt sich mittels

```
void glutReshapeFunc (void (*func)(int width, int height));
```

ein weiterer Behandler registrieren. Häufig wird in ihm auch die Projektionsart eingestellt.

[5] **col**or **amp**lification (**f**loat)

3 Kamera und Projektionen

3.1 Modellmatrix und Projektionsmatrix

OpenGL hält intern zwei 4D-Matrizen:

- Die *Modellmatrix* M enthält die Gesamttransformation, der die ganze Modellwelt unterworfen wird, bevor die Projektion auf die 2D-Bildfläche stattfindet. Insbesondere ist dabei der Wechsel ins Kamera-Koordinatensystem enthalten.
- Die *Projektionsmatrix* P gibt an, ob Parallel- oder Zentralprojektion stattfindet.

Zum Anzeigen des Bildes wird dann $P \cdot M$ auf alle Objektkoordinaten angewendet.
Beide Matrizen können schrittweise aus Grundmatrizen durch Matrixmultiplikation zusammengesetzt werden; auch explizite Angabe ist möglich. Die jeweils aktuelle Modell- oder Projektionsmatrix wird mit C („current“) bezeichnet. Viele OpenGL-Funktionen verändern C, und wir werden oft die Veränderung in algebraischer Form angeben.
Mit

```
void glMatrixMode (GLenum mode);
```

wird ausgewählt, ob mit C die aktuelle Modell- oder Projektionsmatrix gemeint ist; die entsprechenden Parameter sind `GL_MODELVIEW` bzw. `GL_PROJECTION`; die Voreinstellung ist $C = M$.
Zur Initialisierung der jeweiligen Matrix dient

```
void glLoadIdentity (void);
```

mit der Wirkung $C \leftarrow E$ (E ist die Einheitsmatrix).
Einfache Transformationen werden bewirkt durch folgende Funktionen:

`void glTranslatef (GLfloat x, y, z);`	$C \leftarrow C \cdot T(x, y, z)$
`void glScalef (GLfloat x, y, z);`	$C \leftarrow C \cdot S(x, y, z)$
`void glRotatef (GLfloat angle, x, y, z);`	$C \leftarrow C \cdot R(angle, x, y, z)$

3.2 Die Kamera

Sie wird definiert durch

```
void gluLookAt (GLdouble eyex, eyey, eyez, // Augpunkt
                centerx, centery, centerz, // Hauptpunkt
                upx, upy, upz);            // Obenvektor
```

Diese Funktion wird üblicherweise im Behandler `display` aufgerufen.

3.3 Projektionen

Eine Orthogonalprojektion wird spezifiziert durch

```
void glOrtho (GLdouble left, right,
                       bottom, top,
                       near, far);
```

Eine Zentralprojektion ist gegeben durch

```
void gluPerspective (GLdouble alpha, aspect, near, far);
```

3.4 Der Bildausschnitt (Viewport)

Manchmal will man das Frustum nicht auf dem Gesamtschirm anzeigen, sondern nur auf einem Ausschnitt daraus. Dazu dient

```
void glViewport (GLint x, y,
                 GLsizei width, height);
```

x und y geben die linke untere Ecke des Ausschnitts an. Im Gegensatz zu Welt- und Bildkoordinaten werden hier Pixelkoordinaten verwendet.
Achtung: Stimmt das Verhältnis von `width` und `height` nicht mit dem des Frustums überein, kommt es zu Verzerrungen.
Bildausschnitt und Projektion werden meist im Behandler `reshape` gesetzt.

3.5 Beispiel: Orthogonalprojektion eines Dreiecks

```
#include <GL/glut.h>
#include <iostream>
using namespace std;

void display(void) {
  // glClearColor(1,1,1,0);
  glClear(GL_COLOR_BUFFER_BIT);
  glLoadIdentity();
  gluLookAt(0,0,0.5,0,0,0,0,1,0);
  glBegin(GL_LINE_LOOP);
    glVertex3f(-0.3,-0.3,0);
    glVertex3f(0,0.3,0);
    glVertex3f(0.3,-0.3,0);
  glEnd();
  glFlush();
}

void keyboard (unsigned char key, int x, int y) {
  if(key==27) exit(0);
}
```

(Fortsetzung nächste Seite)

```
void reshape(int width, int height) {
  glViewport(0,0,(GLsizei) width, (GLsizei) height);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-1,1,-1,1,-1,1);
  glMatrixMode(GL_MODELVIEW);
}

int main(int args, char **argv) {
  glutInit(&args,argv);
  glutInitWindowSize(500,500);
  glutInitWindowPosition(100,100);
  glutCreateWindow("Beispiel4");
  glutDisplayFunc(display);
  glutKeyboardFunc(keyboard);
  glutReshapeFunc(reshape);
  glutMainLoop();
  return 0;
}
```

3.6 Beispiel: Zentralprojektion eines Würfels

Der Würfel wird in die Objektwelt eingefügt durch den Aufruf

```
glutWireCube(2.0);      // Seitenlänge 2.0
```

in der `display`-Funktion. In `reshape` wird nun

```
gluPerspective(60, (GLfloat)width / (GLfloat)height,
               10, 100.0);
```

aufgerufen. Alles andere ist wie gehabt.

4 Animation

4.1 Prinzip

Animationen werden durch die Hauptschleife von `GL` getrieben. Dazu wird ein Behandler registriert mit

```
void glIdleFunc (void (*func)(void));
```

In diesem Behandler wird dann die Bildänderung von einem Schleifendurchlauf zum nächsten programmiert; am Ende des Behandlers sollte `OpenGL` mit

```
void glutPostRedisplay (void);
```

aufgefordert werden, bei der nächstmöglichen Gelegenheit wieder den Behandler `display` aufzurufen.
Sollten innerhalb eines Schleifendurchlaufs mehrere Aufrufe von `glutPostRedisplay` auftreten, werden sie von `OpenGL` gesammelt und am Ende des Durchlaufs wie ein einziger gewertet.

4.2 Beispiel: Rotierender Würfel

```
#include <GL/glut.h>
#include <iostream>
using namespace std;

GLfloat angle=0.0;

void spin(void) {
  angle+=1;
  glutPostRedisplay();
}

void display(void) {
  // glClearColor(1,1,1,0);
  glClear(GL_COLOR_BUFFER_BIT);
  glLoadIdentity();
  gluLookAt(0,0,5,0,0,0,0,1,0);
  glRotatef(angle,1,0,0);
  glRotatef(angle,0,1,0);
  glRotatef(angle,0,0,1);
  glutWireCube(2.0);
  glFlush();
}

void keyboard (unsigned char key, int x, int y) {
  if(key==27) exit(0);
}

void reshape(int w, int h) {
  glViewport(0,0,(GLsizei) w, (GLsizei) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  //glOrtho(-1,1,-1,1,-1,1);
  gluPerspective(60, (GLfloat)w/(GLfloat)h, 1,100);
  glMatrixMode(GL_MODELVIEW);
}

int main(int args, char **argv) {
  glutInit(&args,argv);
  glutInitWindowSize(500,500);
  glutInitWindowPosition(100,100);
  glutCreateWindow("Beispiel6: Rotating Cube");
  glutDisplayFunc(display);
  glutKeyboardFunc(keyboard);
  glutReshapeFunc(reshape);
  glutIdleFunc(spin);
  glutMainLoop();
  return 0;
}
```

4.3 Doppelpufferung

Dieses Prinzip ist bereits aus Informatik II bekannt; es dient dazu das Flackern bewegter Bilder zu vermeiden. Die Voreinstellung des einfachen Bildpuffers kann überschrieben werden mit

```
void glutInitDisplayMode (unsigned int mode);
```

die Modi sind `GLUT_SINGLE` und `GLUT_DOUBLE` (es gibt auch noch weitere, die für uns aber vorerst nicht relevant sind.)
Die Funktion muss vor `glutCreateWindow` aufgerufen werden.
Das eigentliche Doppelpuffern wird dann durch einen Aufruf von

```
void glutSwapBuffers (void);
```

anstelle von `glFlush` in `display` ausgelöst.

5 Grundlegende Grafikelemente

5.1 Punkte

Ein Punkt im $\mathbb{R}^3$ wird definiert durch:

```
void glVertex3f (GLfloat x, y, z);
```

Will man nur 2D-Grafik betreiben, benutzt man

```
void glVertex2f (GLfloat x, y);
```

5.2 Gruppierung

Mehrere Punkte können zu einer Figur zusammengesetzt werden mit

```
void glBegin (GLenum mode);
void glEnd (void);
```

Dabei gibt `mode` die Art der Gruppe an:

- `GL_POINTS`: unverbundene Punkte
- `GL_LINES`: bildet Punktepaare, von denen jedes verbunden wird, aber nicht mit den anderen zusammenhängt.
- `GL_LINE_STRIP`: Die Punkte werden zu einem Kantenzug zusammengefügt.
- `GL_LINE_LOOP`: Der Kantenzug wird zu einem Kreis geschlossen.
- `GL_TRIANGLES`: Unverbundene Dreiecke aus Dreiergruppen von Punkten.

- `GL_TRIANGLE_STRIP`: Bildet Dreiecke aus

$$v_0, v_1, v_2, v_3, v_4, \ldots$$

wie folgt:

$$\Delta v_0 v_1 v_2, \Delta v_2 v_1 v_3, \Delta v_2 v_3 v_4, \Delta v_4 v_3 v_5, \ldots$$

Damit wird ein Dreiecksnetz als Ausschnitt einer angenäherten 3D-Fläche gebildet.

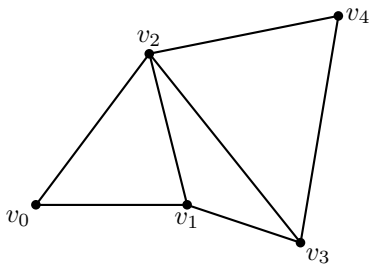


Abbildung 1. *GL_TRIANGLE_STRIP*

- `GL_TRIANGLE_FAN`: Zeichnet Dreiecke $\Delta v_0 v_1 v_2, \Delta v_0 v_2 v_3, \Delta v_0 v_3 v_4$, d.h. einen Fächer mit v_0 im Zentrum.

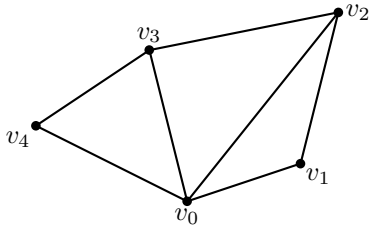


Abbildung 2. *GL_TRIANGLE_FAN*

- `GL_QUADS`: Unverbundene Vierecke aus Vierergruppen von Punkten.

- `GL_QUAD_STRIP`: analog zu `GL_TRIANGLE_STRIP`:

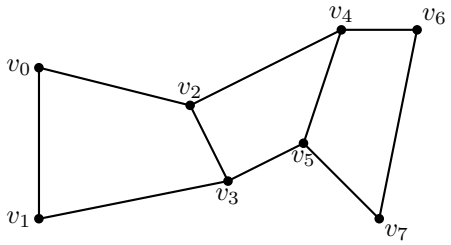


Abbildung 3. *GL_QUAD_STRIP*

- `GL_POLYGON`: verbindet Punkte zu einem Polygonzug. Das Polygon muss konvex sein und darf sich nicht selbst überschneiden. Sonstige Polygone müssen in konvexe zerlegt werden (*Tesselierung*).

Zu Linien und Polygonen können zusätzliche Attribute spezifiziert werden;

- `void glLineWidth (GLfloat width);` stellt die Linienbreite ein.
- `void glLineStipple (GLint factor, GLushort pattern);` definiert die Strichelung (muss zuerst eingeschaltet werden, s. u.); die Möglichkeiten werden binär codiert.
- `void glPolygonMode (GLenum face, mode)` mit `face` $\in$ { `GL_FRONT, GL_BACK, GL_FRONT_AND_BACK` }

  `mode` $\in$ { `GL_FILL, GL_LINE` }

5.3 Einstellen von Optionen

Hierzu dienen
`void glEnable (GLenum capability);`
`void glDisable (GLenum capability);`
Einige mögliche Werte für `capability` sind

- `GL_LINE_STIPPLE`
- `GL_LIGHTING`
- `GL_FOG`
- `GL_DEPTH_TEST`

Aus Effizienzgründen sind sie per Voreinstellung abgeschaltet.

5.4 Grafikfunktionen in GLUT

GLUT stellt einige komplexe Körper vordefiniert zur Verfügung.

- `glutWireCube (GLdouble size)`

  zeichnet einen Würfel der Seitenlänge `size` mit Mittelpunkt im Ursprung (in Weltkoordinaten) als Drahtmodell.

- `glutSolidCube (GLdouble size)`

  analog, aber als festen Körper.

- `glutWireSphere (GLdouble radius, GLint slices, stacks)`

  `glutSolidSphere (GLdouble radius, GLint slices, stacks)`

  Kugel mit Radius `radius` und Mittelpunkt im Ursprung.

  `slices` gibt die Anzahl der „Längengrade“,

  `stacks` die der „Breitengrade“ an.

- `glutWireCone (GLdouble base, height, GLint slices, stacks)`

  `glutSolidCone (GLdouble base, height, GLint slices, stacks)`

  Kegel mit Grundradius `base` und Höhe `height`; die Grundfläche liegt auf der xy-Ebene und die z-Achse ist die Kegelachse.

  `slices, stacks` wirken wie bei den Kugeln.

- ```
  glutWireTorus (GLdouble innerRadius, outerRadius,
                 GLint nsides, rings)
  glutSolidTorus (GLdouble innerRadius, outerRadius,
                  GLint nsides, rings)
  ```

  Torus mit Mittelpunkt im Ursprung und der z-Achse als Achse.

  `nsides` ist die Zahl der Seiten in jedem Radialschnitt,

  `rings` die Zahl der Radialschnitte.

- Die platonischen Polyeder (jeweils mit Mittelpunkt im Ursprung) lassen sich mit folgenden Funktionen zeichnen:

`void glutWireTetrahedron (void)`	Radius $\sqrt{3}$
`void glutSolidTetrahedron (void)`	Radius $\sqrt{3}$
`void glutWireOctahedron (void)`	Radius 1
`void glutSolidOctahedron (void)`	Radius 1
`void glutWireDodecahedron (void)`	Radius $\sqrt{3}$
`void glutSolidDodecahedron (void)`	Radius $\sqrt{3}$
`void glutWireIcosahedron (void)`	Radius 1
`void glutSolidIcosahedron (void)`	Radius 1

- `void glutWireTeapot (GLdouble scale)`

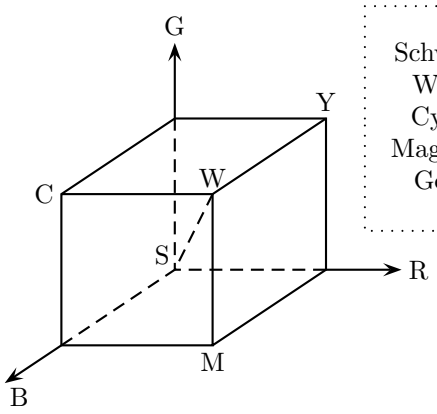
  `void glutSolidTeapot (GLdouble scale)`

  Die klassische Teekanne, um den Faktor `scale` gestreckt.
  ```

6 Weiteres Zubehör

6.1 Farben

Im RGB-Farbmodell wird jede Farbe durch ihren Rot-, Grün- und Blauanteil dargestellt; wie erwähnt gibt OpenGL dazu Bruchzahlen in $[0, 1]$ an.
Der Zusammenhang zwischen den Farben wird durch folgenden Farbwürfel beschrieben:



Schwarz	$S = (0.0, 0.0, 0.0)$
Weiß	$W = (1.0, 1.0, 1.0)$
Cyan	$C = (0.0, 1.0, 1.0)$
Magenta	$M = (1.0, 0.0, 1.0)$
Gelb	$Y = (1.0, 1.0, 0.0)$

Abbildung 4. *Farbwürfel*

Auf der Raumdiagonalen zwischen S und W liegen die Graustufen. Dieses Farbmodell ist die Voreinstellung in OpenGL.
Die aktuelle Zeichenfarbe wird mit

```
void glColor3f (GLfloat red, green, blue);
```

gesetzt.

6.2 Elimination verdeckter Flächen

OpenGL stellt hierfür eigene Befehle zur Verfügung; auf den algorithmischen Hintergrund werden wir später eingehen.
Per Voreinstellung zeichnet OpenGL die Objekte einer Szenerie in der Reihenfolge, wie sie im Programm erscheinen. Das führt insbesondere bei Animationen oft zu grotesken Verdeckungen.
OpenGL unterstützt hier die *Tiefenpufferung (z-buffering)*. Dabei wird für jedes Pixel in einem speziellen Feld Tiefeninformation gespeichert. Bei der Aufbereitung für ein Objekt wird beim Schreiben des entsprechenden Pixels jeweils geprüft, ob es zu einem Punkt geringerer Tiefe gehört, als bisher vermerkt; dann wird der alte, durch das neue Objekt verdeckte, Pixelwert überschrieben, ansonsten ändert er sich nicht.

Die Tiefenpufferung schaltet man so ein:

- Der Aufruf von `glutInitDisplayMode` wird zu

```
void glutInitDisplayMode (GLUT_DOUBLE | GLUT_DEPTH);
```

- In der `display`-Funktion muss zu Beginn auch der Tiefenpuffer gelöscht werden:

```
void glClear (GL_COLOR_BUFFER_BIT |
              GL_DEPTH_BUFFER_BIT);
```

- und die Tiefenpufferung ermöglicht werden:

```
void glEnable (GL_DEPTH_TEST);
```

6.3 Links

Die Quelltexte aus diesem Skript finden sich auch auf `http://www.vislab.de/`. Weitere Links (Dank an Florian Mücke):

1. OpenGL/GLU-Spezifikation:

   `http://www.mevis.de/~uwe/opengl/opengl.html`

2. OpenGL Reference Manual (ähnliche Sammlung wie oben – jedoch etwas aktueller und ausführlicher):

   `http://rush3d.com/reference/opengl-bluebook-1.0/index.html`

3. GLUT API Spezifikation (selbiges für das GLUT):

   `http://www.opengl.org/documentation/specs/glut/spec3/spec3.html`

4. OpenGL-Einführung (eine kleine Einführung): `http://www.robsite.de/daten/tutorials/ogleinfuehrung.pdf`

5. OpenGL RedBook: `http://fly.cc.fer.hr/~unreal/theredbook/`

6. JPot - Java Personal OpenGL Tutorial (interaktives Java Tutorial mit Beispielen mit Schiebereglern – man sieht sofort, wie sich Parameter auswirken; incl. der zugehörigen Quellcodes):

   `http://www.cs.uwm.edu/~grafix2/`

7. NeHe (OpenGL-Tutorials & Quellcode für diverse Sprachen): `http://nehe.gamedev.net/`

Index