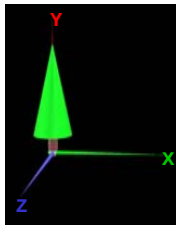
Graphische Datenverarbeitung II

Graphische Programmierung

Prof. Dr. Elke Hergenröther

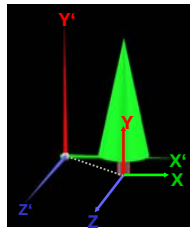
Anordnung der Objekte im Raum



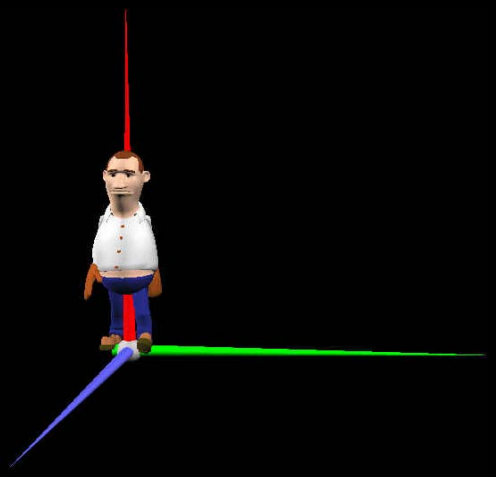
Modellierung des Objekts im lokalen Koordinatensystem

(Modellierungs-koordinatensystem oder körpereigenes Koordinatensystem)

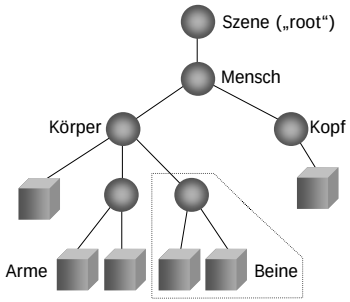
Transformation des Objekts ins Weltkoordinaten-system



Aufbau eines komplexeren Modells



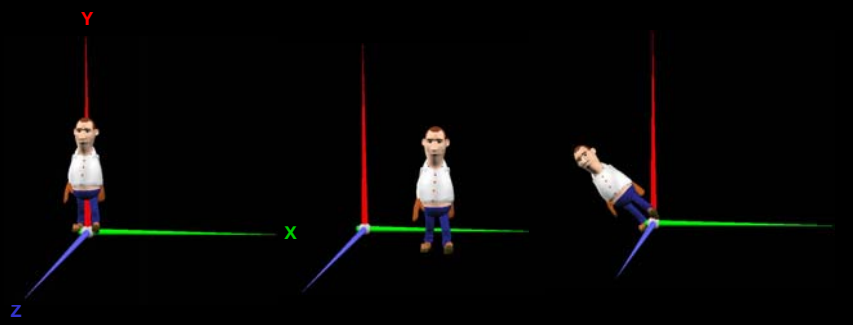
Szenengraph



Szenengraph zur Konstruktion einer Szene:

- Gruppierung von Geometrien zu Gruppen
- Gruppierung von Gruppen zu Gruppen
- Gruppierung von Gruppen zu einer Szene

Beispiele für Transformationen im 3D-Raum

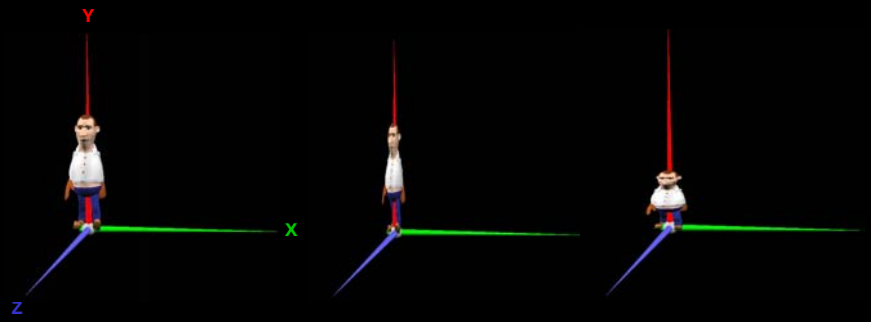


Rechthändiges Koordinatensystem

Translation

Rotation um die Z-Achse

Transformation im 3D-Raum: Skalierung



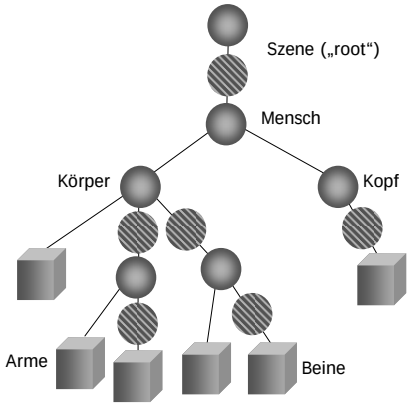
Skalierung entlang der x- und der y-Achse

Szenengraph

Szenengraph besteht aus mindestens 3 Knotentypen:

- Gruppen
- Geometrien (inkl. Materialeigenschaften)
- Transformationen

Szenengraphstruktur:

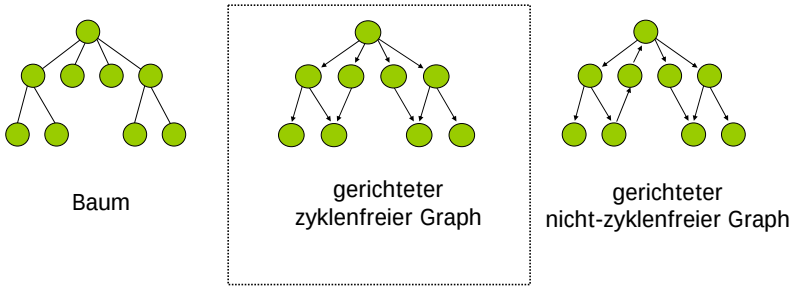


- gerichtet
- **azyklisch**
- heterogen
- Geometrie / darstellbare Primitive in den Blättern

Welchen Vorteil hat ein gerichteter azyklischer Szenengraphen gegen-über Szenengraphen, die als Bäume realisiert wurden?

Wiederholung: Szenengraph

Unterschied Baum und Graph



Baum

gerichteter zyklenfreier Graph

gerichteter nicht-zyklenfreier Graph

Allgemeines zu OpenGL

- OpenGL (Open Graphics Library) ist eine Programmbibliothek für 3D-Graphik
- unabhängig von einem Window-System
- dünne Schicht über der Graphikhardware
- OpenGL ist es Renderingsystem: komplexe Modelle müssen aus einfachen graphischen Primitiven aufgebaut werden.
- immediate mode:
  - sehr einfache Befehlte,
  - keine interne Datenstruktur der Szene
- Alle Namen fangen mit gl... an, Konstanten mit GL_

Allgemeines zu OpenGL

OpenGL ist eine State-Machine:

- Funktionen verändern den internen Zustand, bzw. verwenden ihn zur Darstellung.
- Das heißt einmal angeschaltet, bleibt der betreffende Zustand aktiv bis er wieder ausgeschaltet oder umgeschaltet wird.

OpenGL ist sehr „explizit“:

- Was nicht explizit aktiviert wurde, bleibt aus.
- Beispiel: Es nutzt nichts die Transparenz zu setzen, wenn man nicht explizit gesagt hat, dass Transparenzen berechnet werden sollen.

Higher-level Toolkit: GLUT

Initialisierung über:

```
glutInit(&argc, argv);
glutInitDisplayMode( GLUT_DEPTH |GLUT_RGB );
glutCreateWindow(„Name");
```

Zeichenfunktion wird automatisch von der glutMainLoop aufgerufen.

Welche Funktion als Zeichenfunktion genutzt werden soll, wird mit dieser Funktion festgelegt:

```
glutDisplayFunc(display);
```

Aufruf der Hauptschleife:

```
glutMainLoop();
```

1. OpenGL Programm

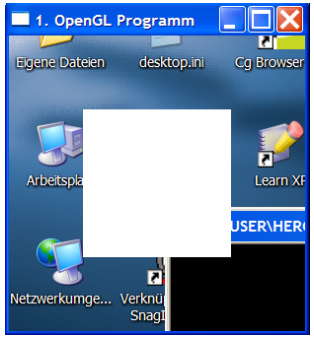
```
#include <GL/glut.h> //GLUT .h-Datei, lädt auch GL .h-Dateien

void display() //Zeichenfunktion
{
   glBegin( GL_POLYGON );
      glVertex3f( -0.5, -0.5, 0);
      glVertex3f(  0.5, -0.5, 0);
      glVertex3f(  0.5,  0.5, 0);
      glVertex3f( -0.5,  0.5, 0);
   glEnd();
   glFlush(); //Buffer leeren
}

int main(int argc, char **argv)
{
   glutInit(&argc, argv); // GLUT initialisieren
   glutInitDisplayMode( GLUT_RGB ); // Fenster Konfiguration
   glutCreateWindow("1. OpenGL Programm"); // Fenster Erzeugung
   glutDisplayFunc(display); // Zeichenfunktion bekannt machen
   glutMainLoop();
   return 0;
}
```


Ausgabe des 1. OpenGL Programms:

Keine Farbe und der Hintergrund ist nicht gesetzt:



Erweiterung des 1. OpenGL Programm

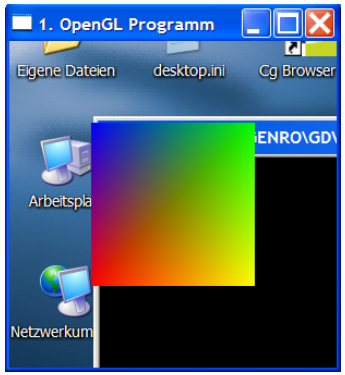
```
#include <GL/glut.h>

void display()
{
   glBegin( GL_POLYGON );
      // State- Machine: Wenn nur der erste Farbwert angeben wird haben
      // alle folgenden Eckpunkte den gleichen Farbwert
      glColor4f( 1., 0., 0., 1.);
      glVertex3f( -0.5, -0.5, 0);
      glColor4f( 1., 1., 0., 1.); // Gelb
      glVertex3f(  0.5, -0.5, 0);
      glColor4f( 0., 1., 0., 1.);
      glVertex3f(  0.5,  0.5, 0);
      glColor4f( 0., 0., 1., 1.);
      glVertex3f( -0.5,  0.5, 0);
   glEnd();
   glFlush(); //Buffer leeren
}

int main(int argc, char **argv)
{
   ... // wie vorher
}
```


Ausgabe des 1. OpenGL Programm

Der Hintergrund ist noch nicht gelöscht worden:



Erweiterung des 1. OpenGL Programm

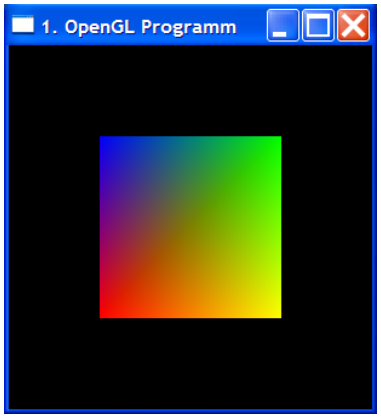
```
#include <GL/glut.h>

void display()
{
   glClear(GL_COLOR_BUFFER_BIT );

   glBegin( GL_POLYGON );
      // State- Machine: Wenn nur der erste Farbwert angeben wird haben
      // alle folgenden Eckpunkte den gleichen Farbwert
      glColor4f( 1., 0., 0., 1.);
      glVertex3f( -0.5, -0.5, 0);
      glColor4f( 1., 1., 0., 1.); // Gelb
      glVertex3f(  0.5, -0.5, 0);
      glColor4f( 0., 1., 0., 1.);
      glVertex3f(  0.5,  0.5, 0);
      glColor4f( 0., 0., 1., 1.);
      glVertex3f( -0.5,  0.5, 0);
   glEnd();
   glFlush(); //Buffer leeren
}

int main(int argc, char **argv) // wie vorher
...
```


Ausgabe des 1. OpenGL Programm



Allgemeines zu OpenGL

OpenGL ist eine State-Machine:

- Funktionen verändern den internen Zustand, bzw. verwenden ihn zur Darstellung.
- Das heißt, einmal angeschaltet, bleibt der betreffende Zustand aktiv, bis er wieder ausgeschaltet oder umgeschaltet wird.

Beispiel: `glColor4f`

Erweiterung des 1. OpenGL Programm

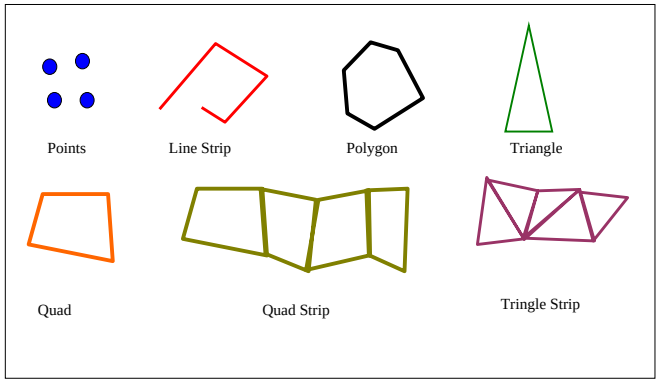
```
#include <GL/glut.h>

void display()
{
   glClear(GL_COLOR_BUFFER_BIT );

   glBegin( GL_POLYGON );
      // State- Machine: Wenn nur der erste Farbwert angeben wird haben
      // alle folgenden Eckpunkte den gleichen Farbwert
      glColor4f( 1., 0., 0., 1.);
      glVertex3f( -0.5, -0.5, 0);
      glColor4f( 1., 1., 0., 1.); // Gelb
      glVertex3f(  0.5, -0.5, 0);
      glColor4f( 0., 1., 0., 1.);
      glVertex3f(  0.5,  0.5, 0);
      glColor4f( 0., 0., 1., 1.);
      glVertex3f( -0.5,  0.5, 0);
   glEnd();
   glFlush(); //Buffer leeren
}

int main(int argc, char **argv) // wie vorher
...
```

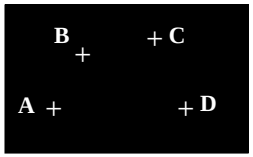
OpenGl Primitive



Definition der OpenGL Primitive

```
glBegin ( Typ des Primitivs )
   Liste von Eckpunkten = Vertices
glEnd ( );
```

```
glBegin ( GL_POINTS );
   glVertex3f ( xA, yA, zA );
   glVertex3f ( xB, yB, zB );
   glVertex3f ( xC, yC, zC );
   glVertex3f ( xD, yD, zD );
glEnd ( );
```

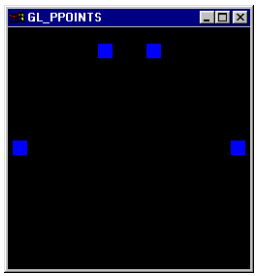


Achtung:
Dies ist nicht die Bildschirmausgabe.
Wegen der Kleinheit der Punkte
(1 Pixel) wird auf dem Schirm
nur sehr wenig zu sehen sein

Merke: 1 Vertex, aber mehrere Vertices!

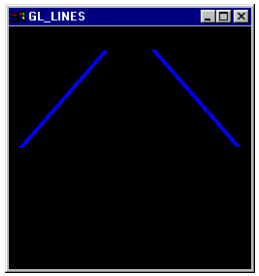
OpenGL Primitiv: Punkt

```
glPointSize (  15.0 );
glBegin ( GL_POINTS );
   glColor4f  ( 0.0f, 0.0f, 1.0f, 1.0f );
   glVertex3f ( -0.9f, 0.0f, 0.0f );
   glVertex3f ( -0.2f, 0.8f, 0.0f );
   glVertex3f ( +0.2f, 0.8f, 0.0f );
   glVertex3f ( +0.9f, 0.0f, 0.0f );
glEnd ( );
```



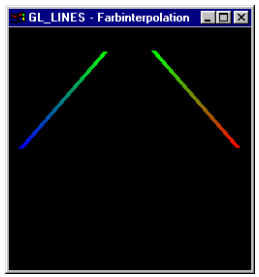
OpenGL Primitiv: Linie

```
glLineWidth (  5.0 );
glBegin ( GL_LINES );
   glColor4f  ( 0.0f, 0.0f, 1.0f, 1.0f );
   glVertex3f ( -0.9f, 0.0f, 0.0f );
   glVertex3f ( -0.2f, 0.8f, 0.0f );
   glVertex3f ( +0.2f, 0.8f, 0.0f );
   glVertex3f ( +0.9f, 0.0f, 0.0f );
glEnd ( );
```



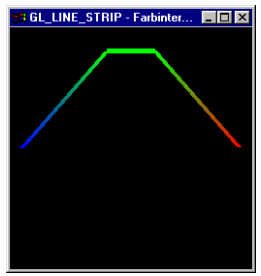
OpenGL Primitiv: Linie

```
glLineWidth (  5.0 );
glBegin ( GL_LINES );
   glColor4f  ( 0.0f, 0.0f, 1.0f, 1.0f );
   glVertex3f ( -0.9f, 0.0f, 0.0f );
   glColor4f  ( 0.0f, 1.0f, 0.0f, 1.0f );
   glVertex3f ( -0.2f, 0.8f, 0.0f );
   glVertex3f ( +0.2f, 0.8f, 0.0f );
   glColor4f  ( 1.0f, 0.0f, 0.0f, 1.0f );
   glVertex3f ( +0.9f, 0.0f, 0.0f );
glEnd ( );
```



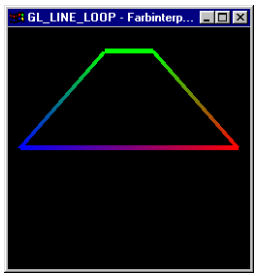
OpenGL Primitiv: Linienzug

```
glLineWidth (  5.0 );
glBegin ( GL_LINE_STRIP );
   glColor4f  ( 0.0f, 0.0f, 1.0f, 1.0f );
   glVertex3f ( -0.9f, 0.0f, 0.0f );
   glColor4f  ( 0.0f, 1.0f, 0.0f, 1.0f );
   glVertex3f ( -0.2f, 0.8f, 0.0f );
   glVertex3f ( +0.2f, 0.8f, 0.0f );
   glColor4f  ( 1.0f, 0.0f, 0.0f, 1.0f );
   glVertex3f ( +0.9f, 0.0f, 0.0f );
glEnd ( );
```



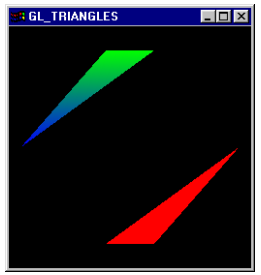
OpenGL Primitiv: Geschlossener Linienzug

```
glLineWidth (  5.0 );
glBegin ( GL_LINE_LOOP );
   glColor4f  ( 0.0f, 0.0f, 1.0f, 1.0f );
   glVertex3f ( -0.9f, 0.0f, 0.0f );
   glColor4f  ( 0.0f, 1.0f, 0.0f, 1.0f );
   glVertex3f ( -0.2f, 0.8f, 0.0f );
   glVertex3f ( +0.2f, 0.8f, 0.0f );
   glColor4f  ( 1.0f, 0.0f, 0.0f, 1.0f );
   glVertex3f ( +0.9f, 0.0f, 0.0f );
glEnd ( );
```



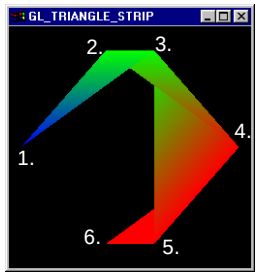
OpenGL Primitiv: Dreieck

```
glBegin(GL_TRIANGLES);
  glColor4f ( 0.0f, 0.0f, 1.0f, 1.0f );
  glVertex3f(-0.9f, 0.0f, 0.0f );
  glColor4f ( 0.0f, 1.0f, 0.0f, 1.0f );
  glVertex3f(-0.2f, 0.8f, 0.0f );
  glVertex3f(+0.2f, 0.8f, 0.0f );
  glColor4f ( 1.0f, 0.0f, 0.0f, 1.0f );
  glVertex3f(+0.9f, 0.0f, 0.0f );
  glVertex3f(+0.2f,-0.8f, 0.0f );
  glVertex3f(-0.2f,-0.8f, 0.0f );
glEnd();
```



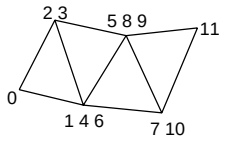
OpenGL Primitiv: Dreiecksstreifen

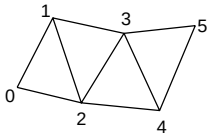
```
glBegin(GL_TRIANGLE_STRIP);
    glColor4f ( 0.0f, 0.0f, 1.0f, 1.0f );
1.  glVertex3f(-0.9f, 0.0f, 0.0f );
    glColor4f ( 0.0f, 1.0f, 0.0f, 1.0f );
2.  glVertex3f(-0.2f, 0.8f, 0.0f );
3.  glVertex3f(+0.2f, 0.8f, 0.0f );
    glColor4f ( 1.0f, 0.0f, 0.0f, 1.0f );
4.  glVertex3f(+0.9f, 0.0f, 0.0f );
5.  glVertex3f(+0.2f,-0.8f, 0.0f );
6.  glVertex3f(-0.2f,-0.8f, 0.0f );
glEnd();
```



OpenGL Primitiv: Dreiecksstreifen

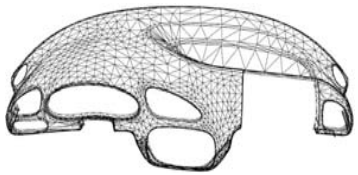
- Ziel ist es, möglichst wenig Elemente anzulegen.
- Eckpunkte können durch zusammenhängende Strips recycelt werden.



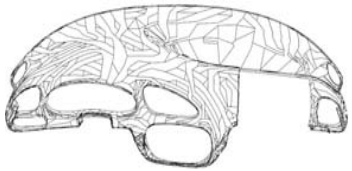


Anzahl der Punkte ohne Streifen- und mit Streifenbildung

OpenGL Primitiv: Dreiecksstreifen



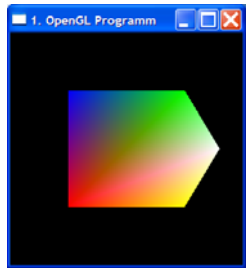
4320 Dreiecke
12960 Eckpunkte



905 Strips
6127 Eckpunkte

OpenGL Primitiv: Polygon

```
glBegin( GL_POLYGON );
  glColor4f( 1., 0., 0., 1.);
  glVertex2f( -0.5, -0.5);
  glColor4f( 1., 1., 0., 1.);
  glVertex2f(  0.5, -0.5);
  glColor4f( 1., 1., 1., 1.);
  glVertex2f(  0.8, 0.);
  glColor4f( 0., 1., 0., 1.);
  glVertex2f(  0.5,  0.5);
  glColor4f( 0., 0., 1., 1.);
  glVertex2f( -0.5,  0.5);
glEnd();
```



Transformationen in OpenGL

Translation:
```
glTranslatef( x, y, z);
```

Rotation:
```
glRotatef( Winkel, x, y, z);
x, y, z: Rotationsachse
```

Skalierung:
```
glScalef( x, y, z);
```

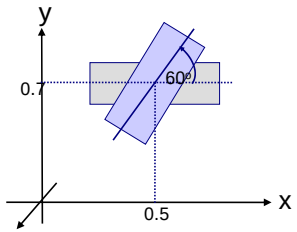

Transformationen in OpenGL

Translation:
```
glTranslatef( x, y, z);
```

Rotationen:
```
glRotatef( Winkel, x, y, z);
x, y, z: Rotationsachse
```

Skalierung:
```
glScalef( x, y, z);
```


Reihenfolge in der die Transformationen angewendet werden ist wichtig!



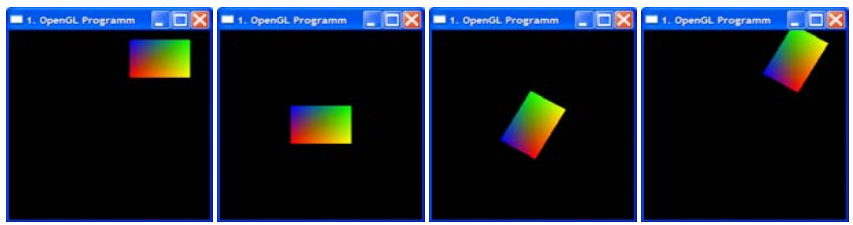
3. Rücktransformation in die Originalposition (M3)
2. Rotation um die Z-Achse (M2)
1. Transformation in den Ursprung (M1)

Transformationen in OpenGL

Transformationen werden in **umgekehrter** Reihenfolge der Angabe angewendet:

Reihenfolge der Anwendung ↑

```
glTranslatef( 0.5, 0.7, 0.); //M3
glRotatef( 60., 0., 0., 1.); //M2
glTranslatef( -0.5, -0.7, 0.); //M1
Quadrat();
```



Reihenfolge der Transformationen:

```
glTranslatef( 0.5, 0.7, 0.); // M = M3

glRotatef( 60., 0., 0., 1.); // M = M3 * M2
glTranslatef( -0.5, -0.7, 0.);// M = M3 * M2 * M1

glBegin( GL_POLYGON ); // P' = M * P
      glColor4f( 1., 0., 0., 1.);
      glVertex3f( 0.2, 0.5, 0);
      glVertex3f(  0.8, 0.5, 0);
      glVertex3f(  0.8,  0.9, 0);
      glVertex3f( 0.2,  0.9, 0);
glEnd();
```


Transformationen in OpenGL

Matrizenstack: Ermöglicht das temporäre Speichern von Matrizen

Matrix wird auf den Stack gesichert: **`glPushMatrix();`**

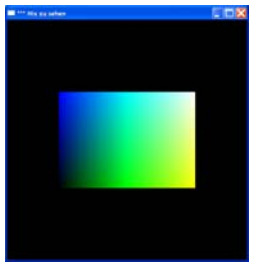
Matrix wird wieder aktiviert mit: **`glPopMatrix();`**

Mit dem Aufruf von **`glPopMatrix()`** werden alle Änderungen der aktuellen Matrix, die zwischen dem Aufruf **`glPushMatrix()`** und **`glPopMatrix()`** gemacht worden sind gelöscht.

Transformation eines Würfels

```
glRotatef(45., 0., 1., 0.);
Wuerfel(0.8);
```

Rotation 45° um die Y-Achse



Transformation eines Würfels

```
glRotatef(320., 1., 0., 0.);
glRotatef(45., 0., 1., 0.);
Wuerfel(0.8);
```

Rotation 320° um die X-Achse

Rotation 45° um die Y-Achse

Transformationen in OpenGL

Matrizenstack: Ermöglicht das temporäre Speichern von Matrizen

Matrix wird auf den Stack gesichert: **`glPushMatrix();`**

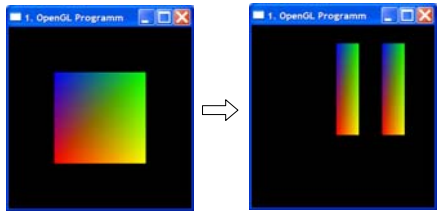
Matrix wird wieder aktiviert mit: **`glPopMatrix();`**

Mit dem Aufruf von **`glPopMatrix()`** werden alle Änderungen der aktuellen Matrix, die zwischen dem Aufruf **`glPushMatrix()`** und **`glPopMatrix()`** gemacht worden sind, gelöscht.

Transformationen in OpenGL

Aufgabe: Der Ausgangswürfel (links) soll so transformiert werden, dass das rechte Bild entsteht.

```
void Quadrat()
{
  glBegin( GL_POLYGON );
    glColor4f( 1., 0., 0., 1.);
    glVertex3f( -0.5, -0.5, 0);
    glColor4f( 1., 1., 0., 1.);
    glVertex3f(  0.5, -0.5, 0);
    glColor4f( 0., 1., 0., 1.);
    glVertex3f(  0.5,  0.5, 0);
    glColor4f( 0., 0., 1., 1.);
    glVertex3f( -0.5,  0.5, 0);
  glEnd();
}
```

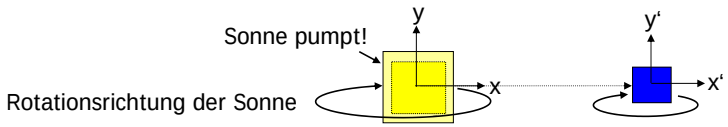


Transformationen in OpenGL

```
glTranslatef(0.3, 0.3, 0);
glPushMatrix();
glTranslatef( -0.25, 0., 0.);
glScalef(0.25, 1., 1.);
Quadrat();
glPopMatrix();
glTranslatef( 0.25, 0., 0.);
glScalef(0.25, 1., 1.);
Quadrat();
```

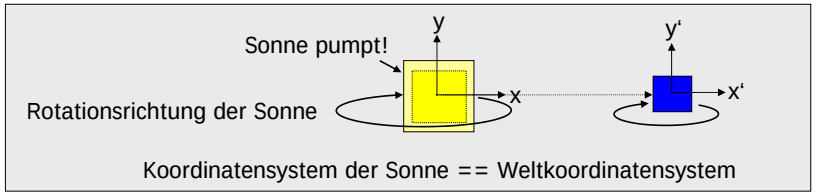

Aufgabe: Miniatursonnensystem

- Sonne rotiert um ihre Y-Achse.
- Während sie rotiert, pumpt sie sich auf und fällt wieder zusammen.
- Die Erde rotiert in einiger Entfernung um die Y-Achse der Sonne und sie rotiert um ihre eigene Z-Achse.
- Erde und Sonne sind eckig ;-)

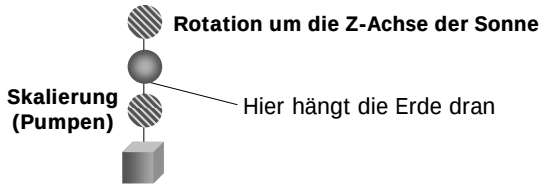


Koordinatensystem der Sonne == Weltkoordinatensystem

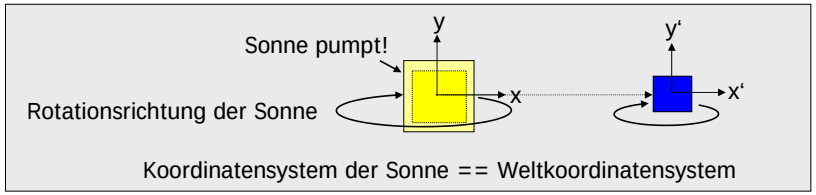
Szenengraph des Miniatursonnensystems



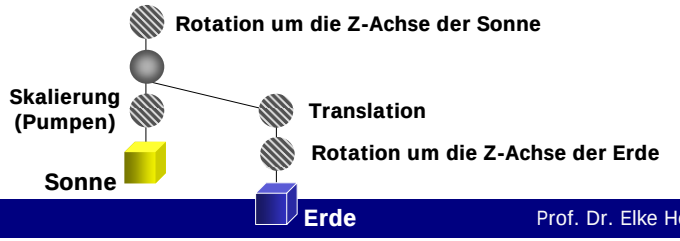
- Sonne und Erde rotieren beide um die Z-Achse der Sonne
- Aber nur die Sonne pumpt sich auf und fällt zusammen



Szenengraph des Miniatursonnensystem

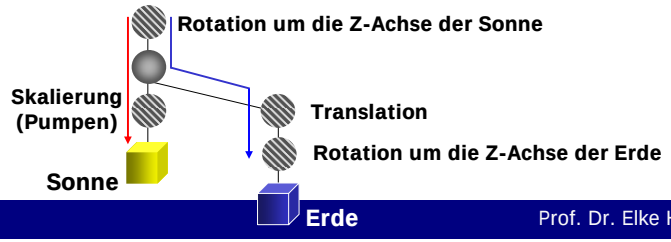


- Erde rotiert um ihre eigne Z-Achse
- und die Erde rotiert in einem gewissen Abstand mit der Rotationsgeschwindigkeit der Sonne um die Z-Achse der Sonne



Prinzipielle Codierung des Szenengraphs

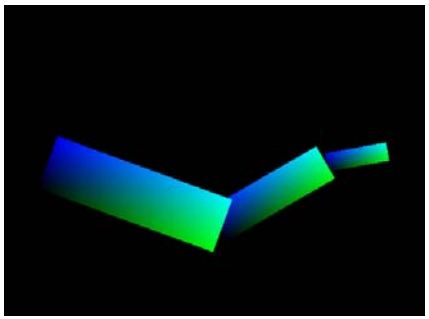
```
//Gruppenknoten zeigen an, wann die Matrix auf dem Stack
//gesichert werden muss.
glLoadIdentity();
glRotatef(fRotSonne, 0., 1., 0.);
glPushMatrix(); //Matrix wird auf den Stack gesichert
glScalef(fX, fY, fZ); // hier wird gepumpt
Wuerfel ( 0.5);
glPopMatrix(); //Matrix wird wieder aktiviert
glTranslatef( 0.5, 0., 0.);
glRotatef( fRotErde, 0., 1., 0.);
Wuerfel ( 0.2);
```



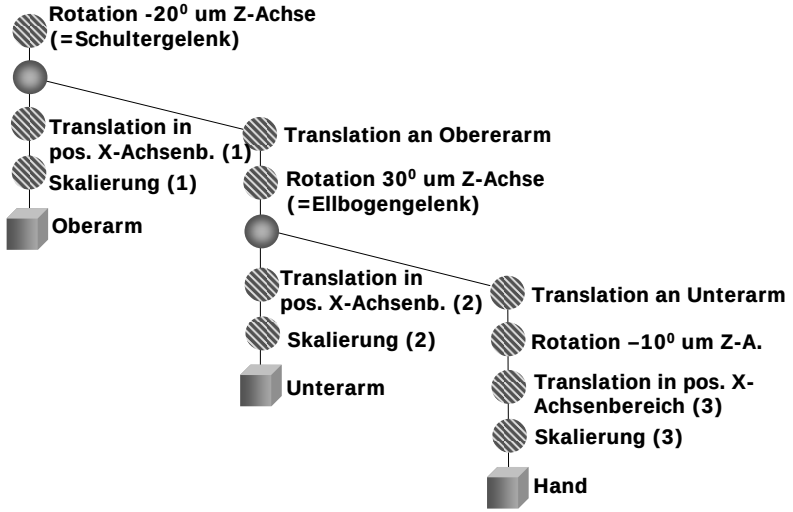

```
void Animate ()
{
   static float fRotSonne = 0., fRotErde=0;
   static float fX=1, fY=1, fZ=1;
   glLoadIdentity();
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);//
   Puffer loeschen
   glRotatef(fRotSonne+= 5., 0., 1., 0.);
   glPushMatrix(); //Matrix wird auf den Stack gesichert
   if( fX == 1){ fX=1.12; fY=1.12; fZ=1.12;}
   else {fX=1; fY=1; fZ=1;}
   glScalef(fX, fY, fZ);
   Wuerfel ( 0.5);
   glPopMatrix(); //Matrix wird wieder aktiviert
   glTranslatef( 0.5, 0., 0.);
   glRotatef( fRotErde+= 25., 0., 1., 0.);
   Wuerfel ( 0.2);
   glFlush();
   Sleep(200);
}
```

Roboterarm

Ausgehend von unserem Würfel, soll dieser Roboterarm erstellt werden:

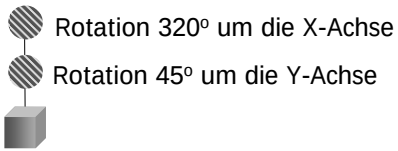


Szenengraph des Roboterarms



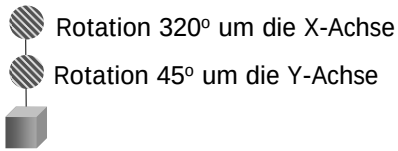
Erinnern Sie sich noch an dieses Beispiel?

```
glRotatef(320., 1., 0., 0.);
glRotatef(45., 0., 1., 0.);
Wuerfel(0.8);
```




```
glRotatef(320., 1., 0., 0.);
glRotatef(45., 0., 1., 0.);
Wuerfel(0.8);
```

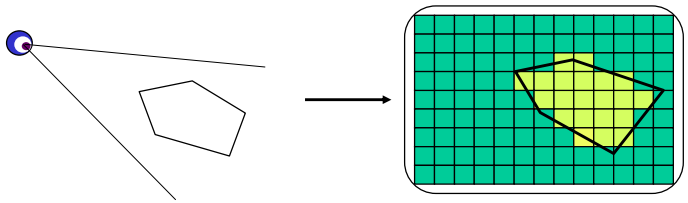
Eigentlich sieht das Ergebnis so aus?!



Z-Buffer

Rendering

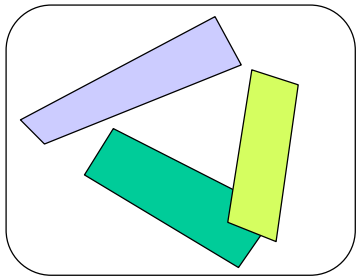
- Die Eckpunkte des Polygons werden aus dem 3D-Raum in den 2D-Raum des Bildschirms transformiert
- Danach wird das 2D-Polygon in einzelne Pixel zerlegt,
- die auf dem Bildschirm dargestellt werden



Z-Buffer

Verdeckungsproblem:

Mögliche Lösung: von hinten nach vorne zeichnen!?

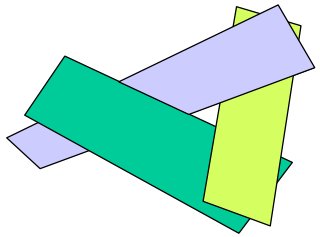


Z-Buffer

Verdeckungsproblem:

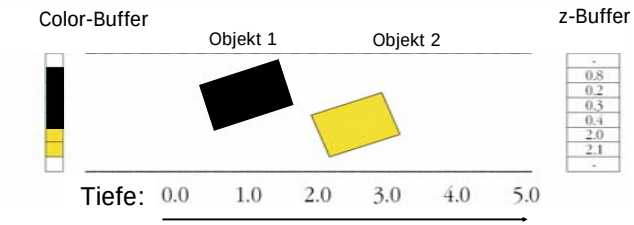
Von hinten nach vorne zeichnen führt zu keiner eindeutigen Lösung

Lösung: Tiefen- oder Z-Buffer



Z-Buffer

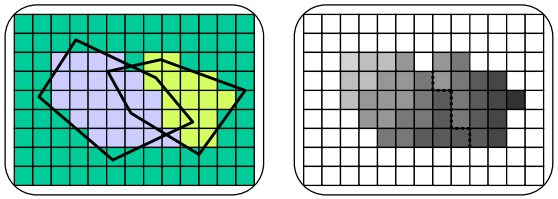
Pixel, die von der Kamera weiter entfernt liegen, als der bereits gespeicherte Z-Bufferwert, werden nicht dargestellt: **Verdeckung**



Visualisierung

Lösung des Verdeckungsproblems mit Z-Buffer:

- Zu jedem Pixel wird die aktuelle Entfernung vom Auge gespeichert
- Nur wenn die Tiefe des aktuell zu schreibenden Pixels kleiner als die gespeicherte ist, wird er geschrieben



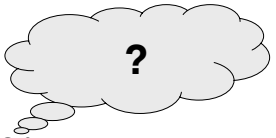
Tiefenwerte pro Pixel:

Dunkler, je näher am Betrachter

Allgemeines zu OpenGL

OpenGL ist sehr „explizit“:

- Was nicht explizit aktiviert wurde, bleibt aus.
- Beispiel: Es nutzt nichts die Transparenz zu setzen, wenn man nicht explizit gesagt hat, dass Transparenzen berechnet werden sollen.



Beispiel: Z-Buffer

OpenGL: Z-Buffer

Initialisierung:

```
glutInitDisplayMode (GLUT_RGB | GLUT_DEPTH);
```

Einschalten:

```
glEnable(GL_DEPTH_TEST);
```

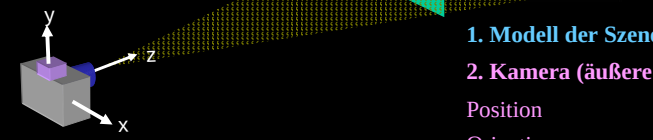
Löschen:

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

Und die Near- und Far-Plane müssen gesetzt werden:

```
z.Bsp.: glOrtho( GLdouble left, GLdouble right,
                 GLdouble bottom, GLdouble top,
                 GLdouble near, GLdouble far);
```


Der Betrachter als Kameramodell



1. Modell der Szene

2. Kamera (äußere Parameter)

Position

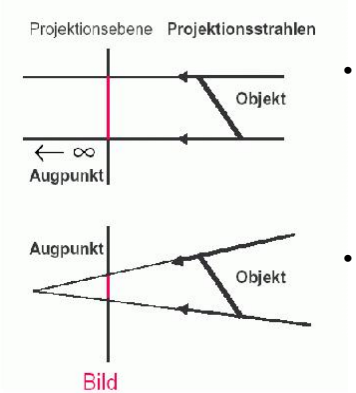
Orientierung

3. Kamera (innere Parameter)

Öffnungswinkel

Clipping - Ebenen

Projektionsverfahren

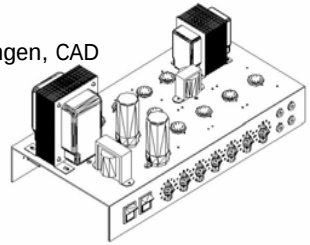


- Parallelprojektion (Isometrische)
- Perspektivische Projektion

Projektionsverfahren

Parallelprojektion

- Parallele Linien bleiben parallel
- Volumenraum der Parallelprojektion entspricht einer **Kiste ;-)**
  Es gibt keinen Fluchtpunkt
- Untergruppe Isometrische Projektion:
  - dieselbe Verkürzung entlang aller drei Achsen
  - Vorteil: Abstände sind in 2D messbar
  - Anwendungsbereich: Technische Zeichnungen, CAD



Projektionsverfahren

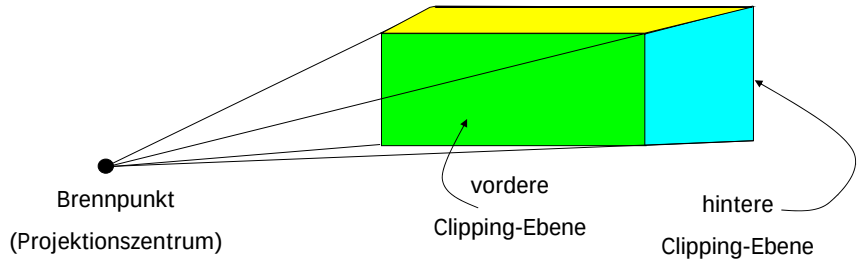
Perspektivische Projektion

- „Natürliche Darstellung“
- Am meisten verwendete Projektionsart
- Sehstrahlen treffen sich in einem Fluchtpunkt

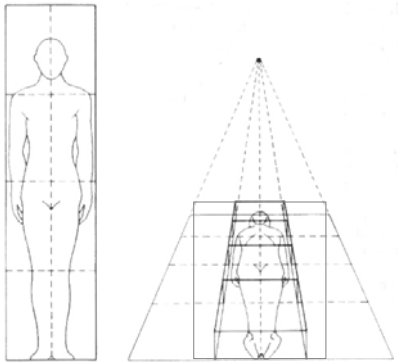
Projektionsverfahren

„Volumenraum“ der perspektivischen Projektion: Pyramidenstumpf

Wie entsteht der Fluchtpunkt?



Perspektivische Projektion



Projektionsverfahren

Setzen des Volumenraums und Berechnung der Projektionsmatrix geschieht durch:

Perspektivische Projektion:

```
gluPerspective(   GLdouble fovy,
                  /*vertikaler Öffnungswinkel*/
                  GLdouble aspect,
                  /*Breiten zu Höhenverhältnis*/
                  Gldouble near,   /* minimaler und */
                  Gldouble far );  /* maximaler Abstand*/
                                   /* von der Kamera zum
                                      Frustum */
```


Projektionsverfahren

Setzen des Volumenraums und Berechnung der Projektionsmatrix geschieht durch:

Perspektivische Projektion:

- Prinzipiell gleichwertig zu **`gluPerspective:`**

```
glFrustum( GLdouble left,   GLdouble right,
           Gldouble bottom, Gldouble top,
           Gldouble near,   Gldouble far );
```

Diese 4 Werte beziehen sich auf die near-Entfernung

- Während mit **`gluPerspective`** nur ein symmetrisches Viewfrustum erstellt werden kann, kann mit glFrustum ein assymetrisches Frustum erstellt werden, was sich dann auch für Stereoprojektionen eignet.

Projektionen

- Projektionsberechnungen werden auch mit Matrixberechnungen durchgeführt; allerdings liegen diese Matrizen auf einem anderen Matrix-Stack: GL_PROJECTION
- Umschalten kann man mittels:

```
glMatrixMode( { GL_MODELVIEW,
                GL_PROJECTION,
                GL_TEXTURE});
```

Eins von drei muss eingeschaltet sein.

- Unterschieden wird zwischen folgenden Projektionsarten:
  - parallel
  - perspektivisch

Der sich bewegende Betrachter

- Es gibt keinen expliziten Unterschied zwischen Kamera und Betrachter.
- Es bleibt dem Programmierer überlassen, ob der Betrachter sich um das Objekt bewegt oder ob sich das Objekt um den Betrachter bewegt. (Auswahl : Anwendungsbedingt)
- D.h. die Betrachtertransformation wird auch zu den Transformationen hinzugezählt.
- Zuständige Matrix heißt: GL_MODELVIEW)

Simulation des bewegten Betrachters in OpenGL

- Standardmäßig guckt die Kamera von Nullpunkt entlang der negativen Z-Achse.
- Um die Betrachterposition zu ändern, gibt es den folgdenden GLUT-Befehl:

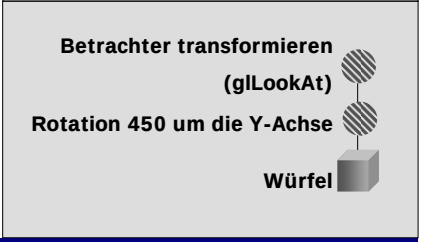
```
gluLookAt(Gldouble eyex, Gldouble eyey, Gldouble eyez,
          /* Betrachterpos. */
          Gldouble centerx, Gldouble centery,
          Gldouble centerz,       /*Blickziel*/
          Gldouble upx, Gldouble upy, Gldouble upz );
          /*Oben-Betrachter*/
```

- gluLookAt setzt die äußeren Kameraparameter und
- gehört damit zur Matrix GL_MODELVIEW – welche Position im Szenegraph?

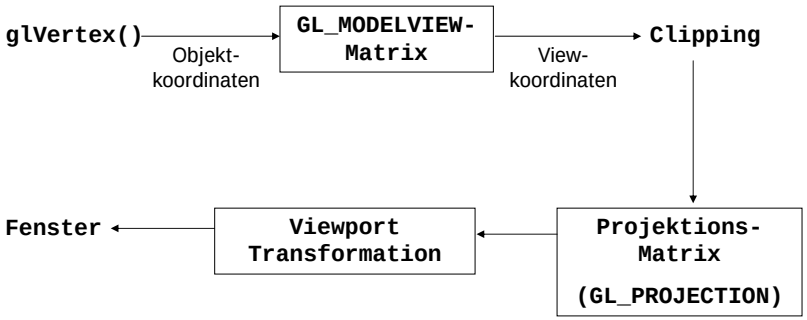
Code mit zugehörigem Szenengraph:

```
void RenderScene(void)
{
  float extent = 0.8;
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho( -1, +1,   /* links, rechts: Aus Sicht der Kamera */
           -1., 1.,  /* unten, oben */
           0.0, 10); /*near, far */
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt( 0, 0, -2*extent,
             0.0, 0.0, 0.0,
             0.0, 1.0, 0.0);
  glRotatef(45, 0., 1., 0.);
  Wuerfel(extent);
  glFlush ();
}
```

Was passiert hier, wenn man die blau markierten Bereiche weglässt?



Transformationspipeline



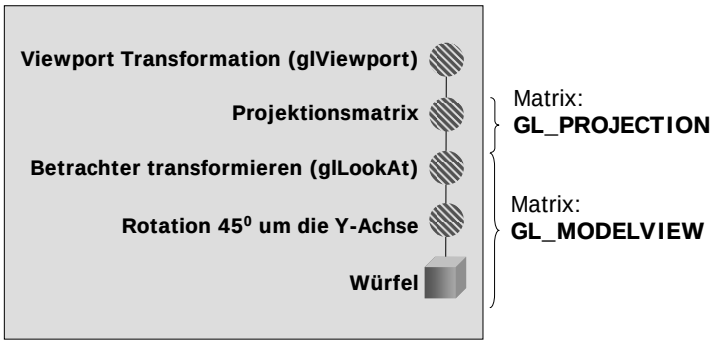
Code mit zugehörigem Szenengraph:

```
void RenderScene(void)
{
  float extent = 0.8;
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho( -1, +1,   /* links, rechts: Aus Sicht der Kamera */
           -1., 1.,  /* unten, oben */
           0.0, 10); /*near, far */
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt( 0, 0, -2*extent,
             0.0, 0.0, 0.0,
             0.0, 1.0, 0.0);
  glRotatef(45, 0., 1., 0.);
  Wuerfel(extent);
  glFlush ();
}
```

Also: was passiert hier, wenn man die blau markierten Bereiche weglässt?

„Vollständiger“ Szenengraph:

Auf Basis der Transformationspipeline erstellt



SwapBuffer

```
// 2 Buffer werden bereitgestellt
glutInitDisplayMode ( GLUT_DOUBLE| ...);
// Standardmäßig: Single-Modus eingeschaltet
// glutInitDisplayMode ( GLUT_SINGLE | ...);

// Jetzt wird der im Hintergrund gefüllte
// Buffer ausgegeben und der andere
// Buffer steht bereit, um für den Betrachter
// unsichtbar gefüllt zu werden.
glutSwapBuffers();
```