OpenGL–Transformationen: Modeling

- Punkt-Trfn = Matrizen-Multiplikationen von links (s.o.):
  $\underline{v}_{neu} = \underline{T}_n \bullet (\ldots) \bullet \underline{T}_2 \bullet \underline{T}_1 \bullet \underline{v}_{alt} = \underline{T}_{gesamt} \bullet \underline{v}_{alt}$
- OpenGL: Laden **`mat[16]`**: **`glLoadMatrix{fd}(mat)`**
  Matrizen-Multiplikation: **`glMultMatrix{fd}(mat)`**

⇨ eigene Matrizen (z.B.: OpenGL bietet keine Scherung)

trans-poniert!

Aber: OpenGL multipliziert von rechts (engl.: *postmultiply)*

⇨ Aufbau von $\underline{T}^T_{gesamt}$ in umgekehrter Reihenfolge:

$\underline{C}_0 = \underline{\mathbf{I}}$

$\underline{C}_1 = \underline{C}_0 \bullet \underline{T}^T_n$

(...)

$\underline{C}_i = \underline{C}_{i-1} \bullet \underline{T}^T_{n-i+1}$

(...)

$\underline{C}_n = \underline{C}_{n-1} \bullet \underline{T}^T_1 = \underline{T}^T_n \bullet (\ldots) \bullet \underline{T}^T_2 \bullet \underline{T}^T_1 = \underline{T}^T_{gesamt}$

```
/*Betr.: Matrix Modell-Trf.:*/
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(TN); /*...*/
glMultMatrixf(T1);
```

$\underline{C}$: aktuelle (engl. *current*) Positionierungsmatrix; $\underline{\mathbf{I}}$: Einheitsmatrix

OpenGL–Transformationen: Modeling

Bequemer (oft: schneller) als **`glLoadMatrix`***(),**`glMultMatrix`***():

- Translation eines Objektes (bzw. lokalen Koord.Systems):
  `void` **`glTranslate{fd}(`**TYPE **`x,`**TYPE **`y,`**TYPE **`z);`**
  (Keine Änderung für **`(0.,0.,0.)`**)
- Rotation eines Objektes (bzw. lokalen Koord.Systems) um d. Winkel **`angle`** (in Grad) um eine Achse vom Koord.-Ursprung zum Punkt (**`x, y, z`**) gegen den Uhrzeigersinn:
  `void` **`glRotate{fd}(`**TYPE **`angle,`**TYPE **`x,`**TYPE **`y,`**TYPE **`z);`**
  (Abstand von Achse = Radius „Umlaufbahn“; Keine Änderg für **`(0.,*,*,*)`**; **`(0.,0.,0.,0.)`** zulässig/wirkungslos)
- Skalierungsfaktoren entlang d. Achsen d. Koord.Systems:
  `void` **`glScale{fd}(`**TYPE **`x,`**TYPE **`y,`**TYPE **`z);`**
  (Keine Änderung für **`(1.,1.,1.)`**; Skalierung mit 0. meist problematisch ⇨ stattdessen: Projektion!)
- Alle OpenGL-Trfn sind Multiplikationen von rechts!

OpenGL–Transformationen: Modeling

Code-Beispiele:

Vor Viewing & Modeling sicherstellen (ggf. wiederherstellen):

Betr.: Modellierung

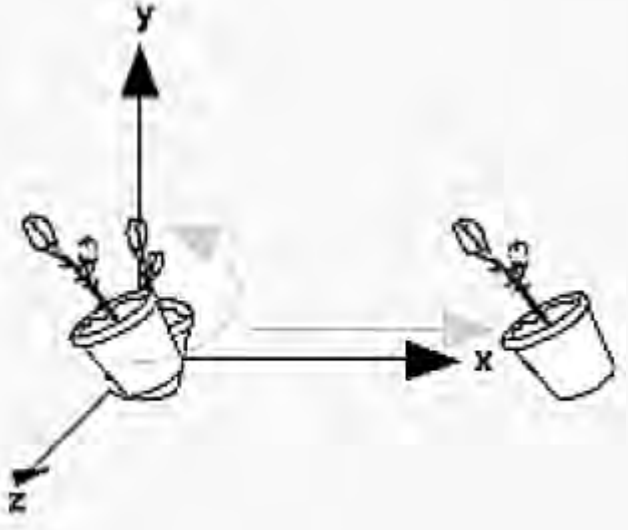
Neuer Start

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

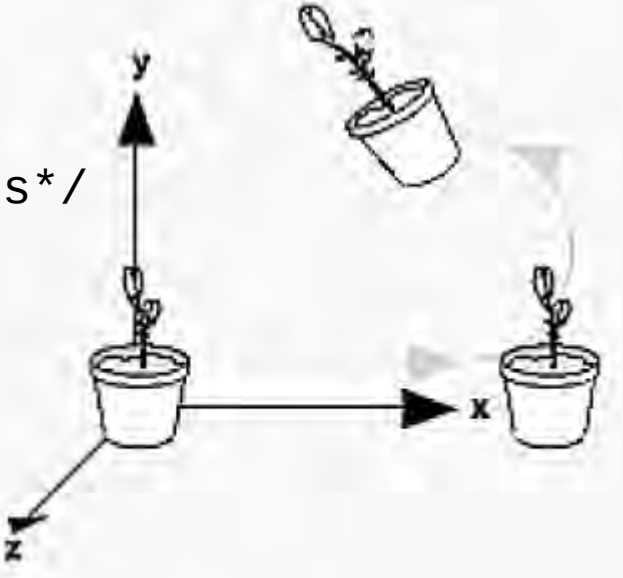
Multiplikationen von rechts mit der Transponierten

```
glTranslatef      (x, 0.,0.);
glRotatef(angle[Z],0.,0.,1.);
```

```
glRotatef(angle[Z],0.,0.,1.);
glTranslatef      (x, 0.,0.);
```




```
glBegin(GL_POLYGON);
/*(...) */
/*transformed vertices*/
glVertex3fv(v1);
/*(...) */
glEnd();
```



Bilder: OpenGL Programming Guide (Addison-Wesley Publishing Company) First Printing, 1993

OpenGL–Transformationen: Viewing

- Sicht-Transformation bestimmt Lage und Orientierung des Sichtvolumens („Augenpunkt“, „Blickrichtung“); sie wirkt in entgegengesetzter Weise wie die Modellierung: „Virtual* Walkthrough“ als Objekt-Trf. in umgekehrter Richtung.
- GLU-Funktion definiert Matrix und multipliziert von rechts:

```
void gluLookAt(
GLdouble eyex,  GLdouble eyey,  GLdouble eyez,
GLdouble centx, GLdouble centy, GLdouble centz,
GLdouble upx,   GLdouble upy,   GLdouble upz);
```

mit: **eyex**/**y**/**z**: Augenpunkt
centx/**y**/**z**: bel. Punkt auf Sichtvolumen-Mittelachse
upx/**y**/**z**: Vektor v. unterer zu oberer Sichtvol.-Kante

OpenGL-Voreinst.: Augenpunkt am Ursprung, Blick entlang neg. z-Achse – äquivalent zu (**centz** willkürlich):

```
gluLookAt (0.,0.,0., 0.,0.,-100., 0.,1.,0.);
```

*engl. *virtual*: „eigentlich“ (vgl.: das Eigentliche)

OpenGL–Transformationen: Projection

- Projektions-Transformation legt Sichtvolumen-Form fest:
  - Art der Projektion (perspektivisch / orthografisch)
  - Objekte (bzw. O.-Teile), die ins Ergebnis-Bild kommen
- Vor Projektions-Transformation sicherstellen:
  `glMatrixMode(GL_PROJECTION);`
  `glLoadIdentity();`
- Festlegung d. Pyramidenstumpfs für die persp. Projektion und Multiplikation mit der aktuellen Positionierungsmatrix:

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

  Eckpunkte der Deckfläche: **`(left, bottom, -near)`** und **`(right, top, -near)`**; **`far`**: Abstand d.Augenpunkts zur Grundfläche des Pyramidenstumpfs. Alle Größen >0.

Tip: Irreal großes Sichtvolumen (z.B. 10^{-3}...10^{6}) kann vorläufig helfen, „verlorene“ Objekte („black screen“) wieder zu finden.

OpenGL–Transformationen: Viewport

- OpenGL-Parallelprojektion (Matrix-Def., -Multiplikation):

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

  Sicht-Quader mit Eckpunkten: **(left, bottom, -near)**, **(right, top, -near)**, **(left, bottom, -far)**, **(right, top, -far)**; alle Größen >0, **near** ≠ **far**.

- Viewport-Festlegung:

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

  mit: **x,y**: linke, untere Fenster-Ecke (def.: 0,0)
  width, **height**: Breite, Höhe des generierten Bildes (def.: Breite, Höhe des Fensters)

  Verzerrungen, falls **width** : **height** ≠ Breite : Höhe!

OpenGL–Transformationen: Projection

- GLU-Alternative (definiert Matrix u. multipliziert v. rechts) für perspektivische Projektion:

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

mit: **`fovy`**: Sichtfeld (*field of view*) Sichtwinkel in y-Richtg: $0. \leq$ **`fovy`** $\leq 180.$

`aspect`: Seitenverhältnis d. Sichtfensters (Breite:Höhe)

`near`, **`far`**: Augenpunkt-Abstände (>0) zu Deck- / Grund-Flächen des Sichtvolumens (*clipping planes*).

`glu`-Sichtfeld symmetrisch um Sichtvolumen-Mittelachse!

OpenGL: Matrix-Stack

- Modular-hierarchische Modellierung / Animation erfordern mehrfache Zwischenablage v.Matrizen (In eigenen Koord. Systemen modellierte/animierte Roboter-Hand /-Unterarm /-Oberarm – jeweils an linker / rechter R.-Schulter).
- Matrizen-Stapelspeicher für ≥32 (**`GL_MODELVIEW`**) bzw. ≥2 (**`GL_PROJECTION`**, **`GL_TEXTURE`**, **`GL_COLOR`**) 4x4-Matrizen.
- Ablage einer Kopie der aktuellen **`glMatrixMode()`**-Matrix als 2.im Stapel (Verschiebung gespeicherter Mat. um eine Position; kein Einfluß auf darauffolgende Berechnungen):

  ```
  void glPushMatrix(void);        /*“Merke Dir!“*/
  ```

- Aufgeben der aktuellen Matrix (der 1. im Stapel), weitere Verwendung der bisher 2. Stapel-Matrix (Rück-Verschiebung der anderen Matrizen im Stapel):

  ```
  void glPopMatrix(void);     /*“Erinnere Dich!“*/
  ```

  (Fehlermeldungen, falls keine Speicherung mehr möglich bzw. noch keine Ablage erhältlich)

OpenGL: Farbe u. Schattierung

- Grundsätzlich: gleiche Farbdarstellg. für alle Geräte-Pixel; Farbdaten auf mehreren (Hw-)„Bitebenen“ (*bitplanes*) = Teilspeicher in Bildspeicher-Dimension mit je 1 Bit / Pixel; Bildspeicher (*framebuffer*) mit 8 Bitebenen können 2^8, mit 24 (meist: 8R, 8G, 8B) 2^{24}=16.777.216 Farben darstellen.
- **(Color) Index Mode**: Einmalig (je Fenster) einstellbare Farb-Palette (-Tabelle: engl. *look up table* bzw. *color map*)

  Bildspeicher (*color buffer*), meist für 256 Farben/Grautöne („*8-bit buffer*“), jeweils wählbar über Farb-/Grauton-Index
- **RGBA Mode**: Individuelle Pixel-Farbgebung durch R/G/B-Farbwerte, Alpha (1.=opak, def.) für Transparenz u./o. Farbmischung (*blending*) / lineare Berechnung, ggf. Gamma-Korrektur/ RGBA aufwendiger, aber verbreiteter
- Farbmodus nur beim Start (Initialisierung) einmal wählbar; Farben werden Eckpunkten (*vertices*) zugewiesen

OpenGL: Farbe u. Schattierung

- Farbwerte meist als **`float`** [0. bis 1.] Interne Umrechnung auf System-Ressourcen – z.B. bei 8 Bitplanes für jede Farbe: 0.=0/255; ... ; 1.0=255/255
- Systemabhängig möglich: Bitplane-Aufteilung bei Double Buffering (2x4 Bitplanes je Farbe); OpenGL vorsorglich: Halbton-Darstellung (eng. *dithering*); abstellbar mit:

  ```
  glDisable(GL_DITHER);//def.:glEnable(GL_DITHER)
  ```

  (Denkbare Anwendung: Bilder/Grafiken mit 24 Bitplanes, flimmerfreie Fenster-Verlegung mit Double-Buffering –evtl. in 2 Synchron-Fenstern)
- Wahl des Schattierungsmodells:

  ```
  void glShadeModel (GLenum mode);
  ```

  Zulässige Werte für **`mode`**:

  `GL_FLAT` (Flat – Farbwert einheitlich für ganze Fläche),

  `GL_SMOOTH` (Gouraud – def.: Interpolation zwischen individuell berechneten Eckpkt-Farbwerten)

GLUT / OpenGL-Animation

Grafik-Animation („-Beatmung“, „-Behauchung“):

- Bewegungsillusion durch schnelle Abfolge Bild-Löschung-Bild... (vgl. Kino: Bild-Blende-Bild...); bei sichtbarer Bild-Entstehung: Flimmern des zuletzt berechneten Bildteils

  ➥ Animation meist mit **Double-Buffering**:

  Darbietung erst nach Fertigstellung eines Bildes durch Wechsel (engl. *swap*) des zugeschalteten Bildspeichers
- Aufgrund starker Abhängigkeit von Hardware überläßt es OpenGL dem Fenstersystem: **`glutSwapBuffers();`**
- Typische Bildraten in Hz (Bilder/sec = *fps* [frames per sec]):

  Ch.Chaplin-Filme: 16 Hz bzw. 62,5 ms/Bild

  modernes Kino: 24 Hz bzw. 41,66... ms/Bild

  TV (Interlaced): 50 Hz bzw. 20,0 ms/Halbbild

  Flug-Simulation: 60-120 Hz bzw. 16,66...-8,33... ms/Bild

GLUT / OpenGL-Animation

- Darstellung auf elektr. Ausgabemedien (Monitor, Beamer) flüchtig: Bild wird in konstanten Zeitabständen t_R [msec] aktualisiert (vgl. TV).

  ⇨ Bildauffrischungsrate (engl. *refresh rate*) [Hz] $f_R = 1/ t_R$

- Wichtig für die Illusion kontinuierlicher Bewegung: Konstante Bildrate (engl. *frame rate*) [Hz] $f_F = 1 / t_F$, mit t_F [msec]: Darbietungszeit eines Bildes bzw. Zeit zwischen dem Zeichenbeginn zweier aufeinanderfolgender Bilder.

  ⇨ t_F kann nur ein ganzzahliges Vielfaches von t_R sein:

  $t_F = n \cdot t_R$, mit $n \geq 1, n \in \mathbf{N}$

- Es muß je Bild Rechenzeit $t_A \leq t_F = 1/ f_F$ gewährleistet sein ⇨ bei Eintritt komplexer Objekte in das Sichtvolumen ggf. Auslassen (engl. *drop*) v. Einzelbildern u. Herabsetzg der Bildrate so, daß jedes Bild als Rechenzeit ein ganzzahl. Vielfaches der Sichtsystem-Bildauffrischungszeit erhält.

GLUT / OpenGL-Animation

- Definition: Die **Ganzzahlfunktion** [x] ordnet einer reellen Zahl x eine ganze Zahl z zu:

  „Gauß-Klammer“

  $$z = [x] \quad \Leftrightarrow \quad x-1 < z \leq x \quad (z \in \mathbf{Z},\ x \in \mathbf{R})$$

  Für nicht-positive $x = -|t| \leq 0$, $t \in R$, gilt:

  $$z = [-|t|] \quad \Leftrightarrow \quad -|t| - 1 < z \leq -|t|$$

  bzw.:

  $$\Leftrightarrow \quad |t| + 1 > -z \geq |t|$$

  Mit $n = -z \geq 0$ wird daraus:

  $$n = -z = -[-|t|] \quad \Leftrightarrow \quad |t|+1 > n \geq |t| \quad (n \in \mathbf{N}_0,\ t \in \mathbf{R})$$

  – in Worten: Die Funktion $n_{(t)} = -[-|t|]$ ordnet einer nicht-negativen reellen Zahl $|t|$ die nicht-negative ganze Zahl n mit dem gleichen oder dem nächstgrößeren Betrag zu.

- Daraus Forderung für Animation: $t_F = n \cdot t_R$ mit $n = -[-t_A / t_R]$

  ⇨ $t_F = -[-t_A / t_R] \cdot t_R$

  bzw. wegen $f_F = 1/t_F$ und $f_R = 1/t_R$ und $n = -[-t_A \cdot f_R]$:

  $$f_F = -f_R / [-t_A \cdot f_R]$$

⇨ Berechnung der Bildrate f_F aus der Rechenzeit $\boldsymbol{t_A}$ für ein Bild und der Bildauffrischungsrate f_R:

$f_F = - f_R / [- t_{\boldsymbol{A}} \cdot f_R]$

<u>Beispiel:</u>

Wenn bei einer „*refresh rate*“ von f_R =120 Hz ein Bild $t_{\boldsymbol{A}}$ = 1/50 sec Rechenzeit braucht, beträgt die „*frame rate*“:

f_F = 120 / -[-120 / 50] Hz = 120/ -[-2,4] Hz = 120/ -[-3] Hz = 40 Hz

Die verfügbare (Rechen-,Warte-) Zeit je Bild beträgt dann:

t_F = 1/40 sec = 1000/40 ms = 25 ms

D.h., jede 3. Bildauffrischung bekommt einen neuen Inhalt u.d.Grafik-System bleibt untätig (engl. *idle*) für 20% d. Zeit

$(t_{\boldsymbol{F}} – t_{\boldsymbol{A}}) / t_{\boldsymbol{F}}$ = (1/40–1/50) / 1/40 = 40·(5-4)/200 = 1/5 = 0,2