Complex Numbers as Rotations

$$e^{i\theta} = \cos\theta + i\sin\theta$$

$$v = re^{i\varphi} = r\cos\varphi + ri\sin\varphi$$

$$e^{i\theta}v = re^{i\theta}e^{i\varphi} = re^{i(\theta+\varphi)} = r\cos(\theta+\varphi) + ri\sin(\theta+\varphi)$$

Vorbemerkung

- Wir können reelle Zahlen einfach invertieren:

$$x \cdot \frac{1}{x} = 1$$

- Auch für komplexe Zahlen können wir ein Inverses finden:

$$\frac{z \cdot z^*}{|z|^2} = 1$$

- Gibt es etwas Analoges auch in "höheren Dimensionen"?
  - Z.B. eine "dreidimensionale" Verallgemeinerung von   ? $\rightarrow$ Nein!
  - Aber: im 4D klappt es wieder … (fast)

Quaternionen

[Hamilton, 1843]

- Erweiterung der komplexen Zahlen (leider nicht mehr kommutativ):

$$\mathbb{H} = \left\{ q \mid q = w + a \cdot \mathbf{i} + b \cdot \mathbf{j} + c \cdot \mathbf{k} \;,\; w, a, b, c \in \mathbb{R} \right\}$$

- Alternative Schreibweise:

$$q = (\, w, \mathbf{v} \,)$$

- Axiome für die 3 imaginären Einheiten:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$$

$$(\mathbf{ij})\mathbf{k} = \mathbf{i}(\mathbf{jk})$$

- Daraus folgen sofort diese Rechengesetze:

$$\mathbf{ij} = -\mathbf{ji} = \mathbf{k} \qquad \mathbf{jk} = -\mathbf{kj} = \mathbf{i} \qquad \mathbf{jk} = -\mathbf{kj} = \mathbf{i}$$

Eine Algebra über den Quaternionen

- Addition: $q_1 + q_2 = (w_1 + w_2) + (a_1 + a_2)\mathbf{i} + (b_1 + b_2)\mathbf{j} + (c_1 + c_2)\mathbf{k}$
- Skalierung: $s \cdot q = (sw) + (sa)\mathbf{i} + (sb)\mathbf{j} + (sc)\mathbf{k}$
- Multiplikation:

$$\begin{aligned} q_1 \cdot q_2 &= (w_1 + a_1\mathbf{i} + b_1\mathbf{j} + c_1\mathbf{k}) \cdot (w_2 + a_2\mathbf{i} + b_2\mathbf{j} + c_2\mathbf{k}) \\ &= (w_1 w_2 - a_1 a_2 - b_1 b_2 - c_1 c_2) + \\ &\quad (w_1 a_2 + w_2 a_1 + b_1 c_2 - c_1 b_2)\,\mathbf{i} + \\ &\quad (\quad \ldots \quad \ldots \quad)\,\mathbf{j} + \\ &\quad (\quad \ldots \quad \ldots \quad)\,\mathbf{k} \end{aligned}$$

- Konjugation: $q^* = w - a\mathbf{i} - b\mathbf{j} - c\mathbf{k}$
- Betrag (Norm): $|q|^2 = w^2 + a^2 + b^2 + c^2 = q \cdot q^*$
- Inverse eines Einheitsquaternions: $|q| = 1 \ \Rightarrow \ q^{-1} = q^*$

- Behauptung (o. Bew.):

  $\mathbb{H}$ mit der Multiplikation ist eine nicht-kommutative Gruppe.

Einbettung des 3D-Vektorraumes in $\mathbb{H}$

- Den Vektorraum $\mathbb{R}^3$ kann man in $\mathbb{H}$ so einbetten:

$$\mathbf{v} \in \mathbb{R} \;\mapsto\; q_v = (0, \mathbf{v}) \in \mathbb{H}$$

- Definition:
  Quaternionen der Form $(0, \mathbf{v})$ heißen reine Quaternionen (*pure quaternions*)

Darstellung von Rotationen mittels Quaternionen

- Gegeben sei Axis & Angle $(\varphi, \mathbf{r})$ mit $\|\mathbf{r}\| = 1$
- Definiere das dazu gehörige Quaternion als

$$q = \left(\cos\frac{\varphi}{2}, \sin\frac{\varphi}{2}\mathbf{r} \right) = \left(\cos\frac{\varphi}{2}, \sin\frac{\varphi}{2}r_x, \sin\frac{\varphi}{2}r_y, \sin\frac{\varphi}{2}r_z \right)$$

- Beobachtung: $|q| = 1$

- Zurückrechnen:

$$q = (w, a, b, c) \text{ ist gegeben, mit } |q| = 1$$

Dann ist

$$\varphi = 2\arccos(w)$$

$$\mathbf{r} = \frac{1}{\sin\frac{\varphi}{2}} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \frac{1}{\sqrt{1-w^2}} \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

- Satz:

  Jedes Einheitsquaternion kann man in der Form

  $$\left(\cos\tfrac{\varphi}{2}, \sin\tfrac{\varphi}{2}\mathbf{r} \right)$$

  darstellen.

- Beweis: siehe vorige Folie

- Theorem: Rotation mittels eines Quaternions

  Sei $\mathbf{v} \in \mathbb{H}$ ein pures Quaternion und $q \in \mathbb{H}$ ein Einheitsquaternion. Dann beschreibt die Abbildung

  $$\mathbf{v} \mapsto q \cdot \mathbf{v} \cdot q^* = \mathbf{v}'$$

  eine (rechtshändige) Rotation von **v**, wobei Winkel und Achse durch q bestimmt sind.

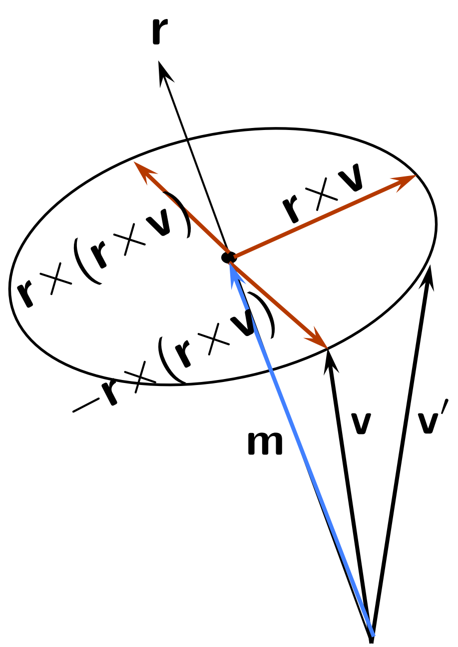
- Beweisskizze:

$$q\mathbf{v}q^* = (c, s\mathbf{r})\cdot(0,\mathbf{v})\cdot(c,-s\mathbf{r}) \qquad \text{mit} \quad c = \cos\frac{\varphi}{2},\ s = \sin\frac{\varphi}{2}$$

$$= \ldots \qquad (*)$$

$$= \Big(\,0,\ \underbrace{\mathbf{v} + \sin\varphi\cdot\mathbf{r}\times\mathbf{v} + (1-\cos\varphi)\cdot\mathbf{r}\times(\mathbf{r}\times\mathbf{v})}_{\shortparallel}\,\Big)$$

$$\underbrace{\mathbf{v} + \mathbf{r}\times(\mathbf{r}\times\mathbf{v})}_{\mathbf{m}} + \sin\varphi\cdot\mathbf{r}\times\mathbf{v} + \cos\varphi\cdot(-\mathbf{r}\times(\mathbf{r}\times\mathbf{v})) \quad = \mathbf{v}'$$



*) Zwischendurch benötigt man diese trigonometrischen Identitäten:

$$\sin\varphi = 2\sin\frac{\varphi}{2}\cos\frac{\varphi}{2} \qquad 1-\cos\varphi = 2\sin^2\frac{\varphi}{2}$$

- Bemerkung: die so definierte Rotationsabbildung ist mit der Quaternionen-Multiplikation verträglich, d.h., dass

$$R_{q_1}(R_{q_2}(\mathbf{v})) = R_{q_1 \cdot q_2}(\mathbf{v})$$

Lineare Interpolation von Quaternionen

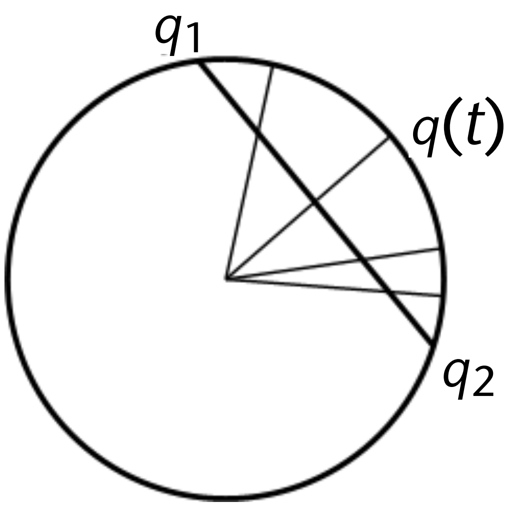
- Gegeben: zwei Orientierungen q_1 , q_2
  (Orienterung = Rotation aus der Null-Lage)
- Aufgabe: dazwischen interpolieren
- Einfachste Lösung:

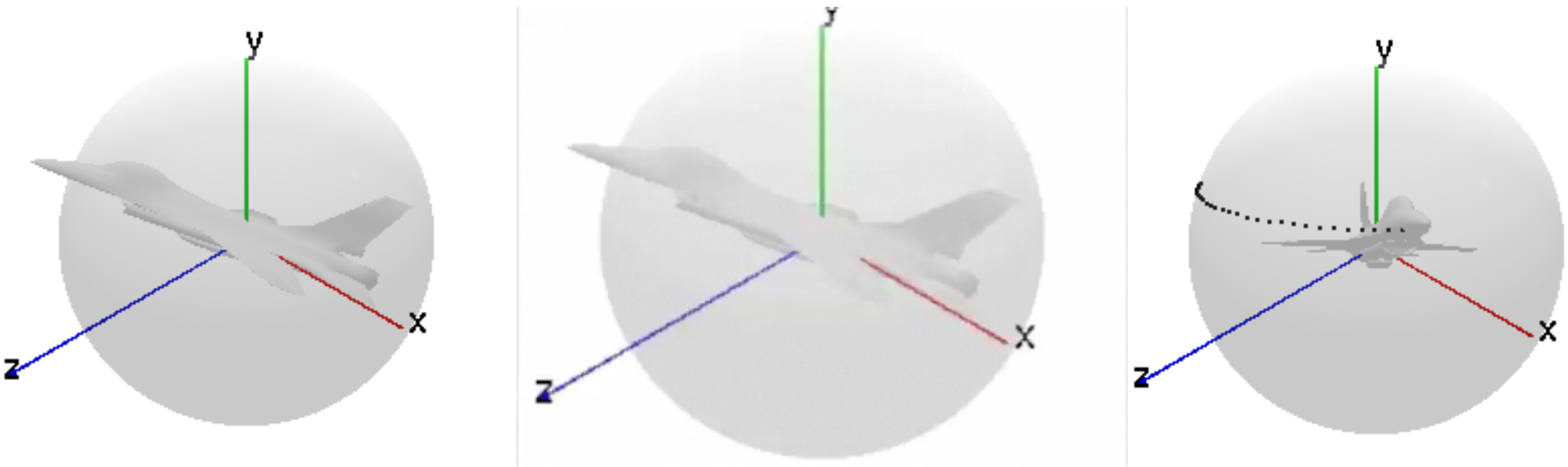
$$q(t) = \mathrm{lerp}(t; q_1, q_2) = (1-t)q_1 + tq_2$$

  - Wichtig: $q(t)$ hinterher immer normieren!

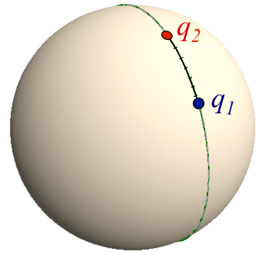
- Vorteil: Kein Gimbal Lock!

- Nachteil (noch): keine konstante Winkelgeschwindigkeit
- Problem: Geschwindigkeit an den "Enden" der Interpolation ist langsamer als in der "Mitte"





Sphärische lineare Interpolation

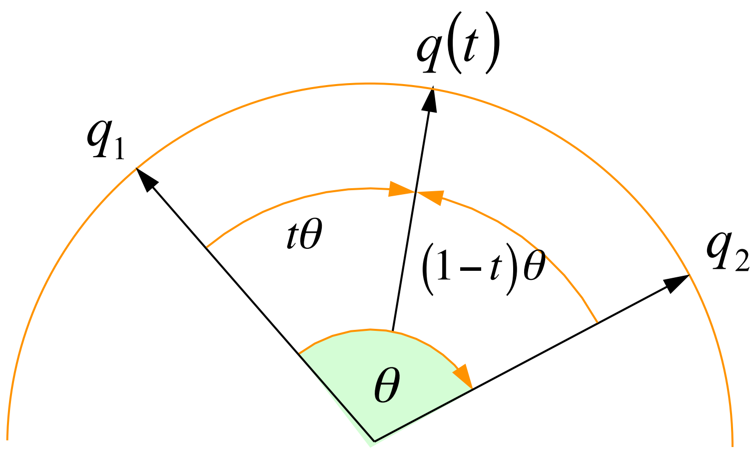


- Besser ist die sphärische lineare Interpolation "*slerp*":

$$q(t) = \text{slerp}(t; q_1, q_2) = \frac{\sin((1-t)\theta)}{\sin\theta} q_1 + \frac{\sin(t\theta)}{\sin\theta} q_2$$

mit $\cos\theta = q_1 \odot q_2$ Skalarprodukt der *Vektoren* q_1 und q_2

- Hier gilt:



Vergleich der verschiedenen Interpolationsarten

Interpolation of Euler angles

Naïve interpolation of matrices

Slerp of quaternions

Lerp of quaternions (with normalization)

Gianluca Vatinno, Trinity College Dublin

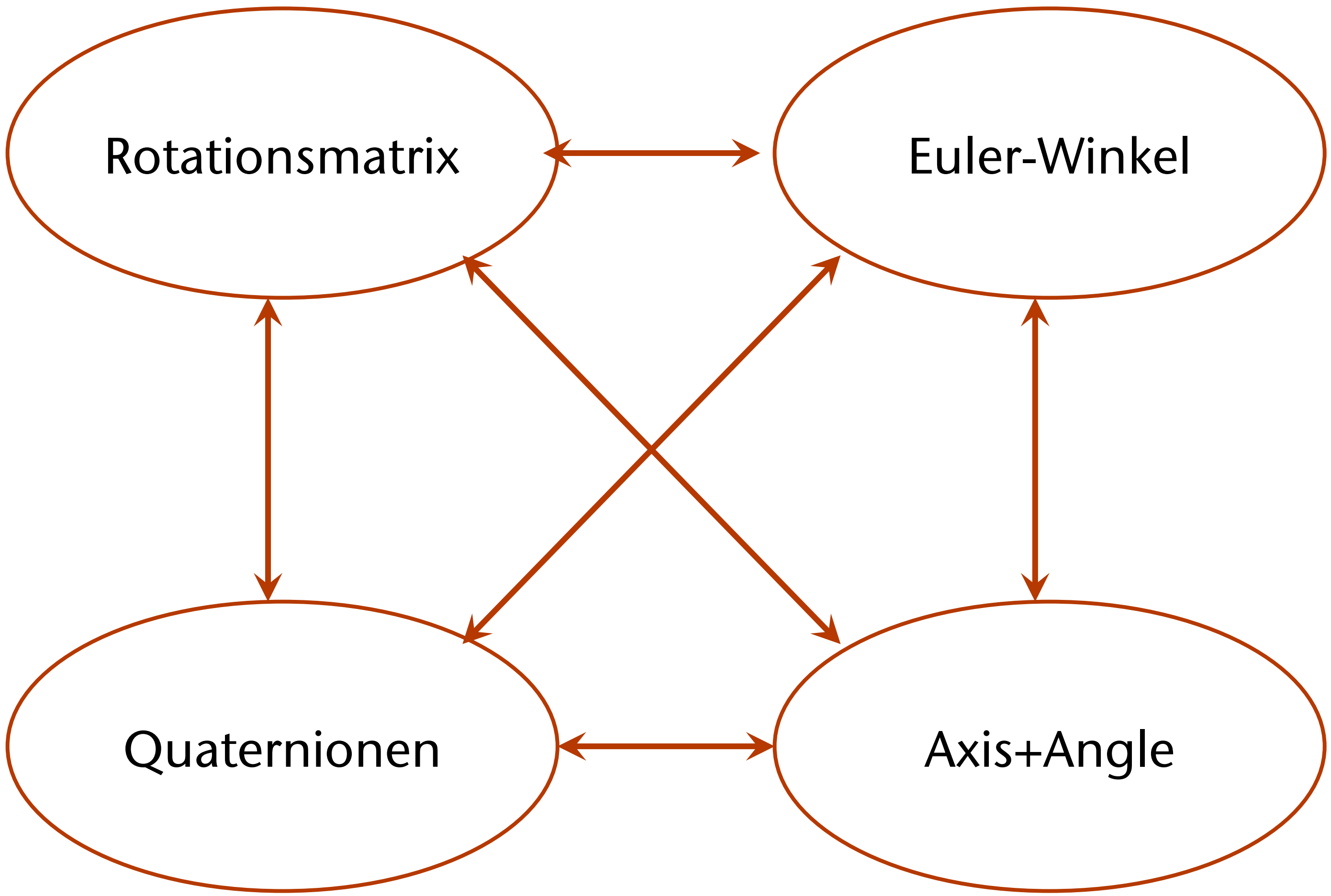
Umwandlung Quaternion → Rot.matrix

- Zunächst das Analogon im 2D:
  wenn a,b, mit $a^2 + b^2 = 1$, gegeben sind, dann ist
  $$M = \begin{pmatrix} a^2 - b^2 & -2ab \\ 2ab & a^2 - b^2 \end{pmatrix}$$
  eine Rotationsmatrix.
- So ähnlich kann man eine Rotationsmatrix im 3D aus einem Quaternion $q = w + a\mathrm{i} + b\mathrm{j} + c\mathrm{k}$, mit $|q| = 1$, bilden:

$$R(q) = \begin{pmatrix} w^2 + a^2 - b^2 - c^2 & 2ab - 2wc & 2ac - 2wb \\ -2ab + 2wc & w^2 - a^2 + b^2 - c^2 & 2bc - 2wa \\ -2ac + 2wb & -2bc + 2wa & w^2 - b^2 - c^2 + d^2 \end{pmatrix}$$

- Überprüfung:
  - Spalte i × Spalte $j = 0 \;\Leftrightarrow\; i \neq j$
  - Spalte i × Spalte $i = (w^2 + a^2 + b^2 + c^2)^2$

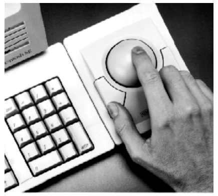
Alle Darstellungen von Rotation lassen sich ineinander umrechnen



Mehr Infos: siehe die Tutorials auf der Homepage der Vorlesung!

Der virtuelle Trackball

- Wie gibt man Orientierungen mit der Maus ein?
- Idee:
  - Lege Kugel um das Objekt / die Szene
  - Kugel kann um ihr Zentrum rotieren
  - Maus pickt Punkt auf Oberfläche, den man zieht



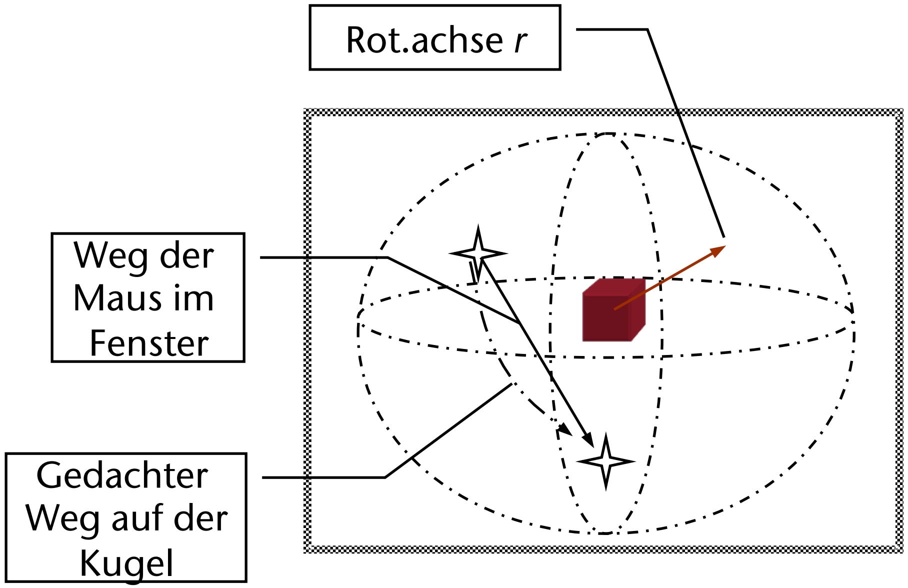
- Geg.: 2D Punkte Startpunkt = (x_1,y_1), Endpunkt = (x_2,y_2)
- Ges.: Rotationsachse r
- Berechnung:
  1. Bestimme 3D Punkte

$$\mathbf{p}_i = (x_i, y_i, z_i)$$

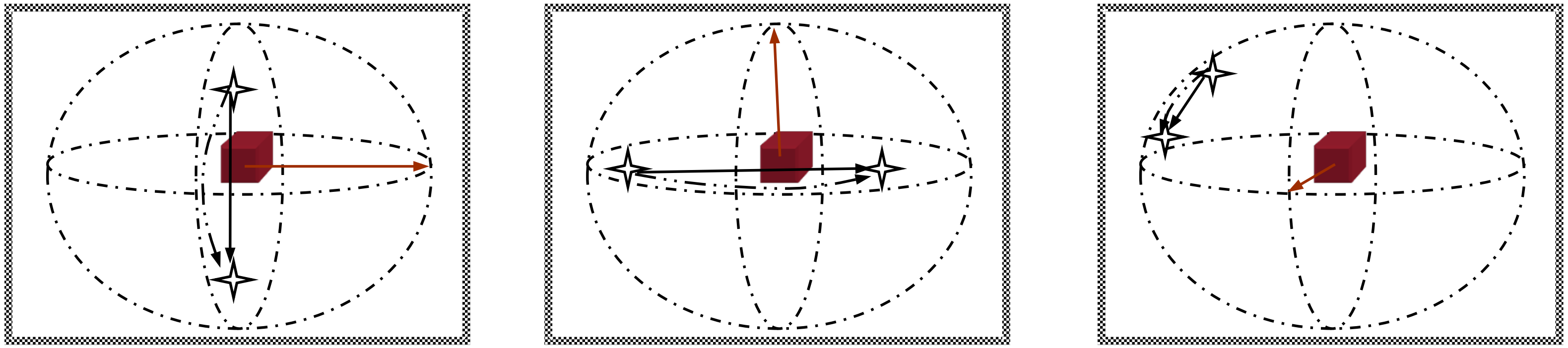
$$z_i = 1 - \sqrt{x_i^2 + y_i^2}$$

  2. Rotationsache

$$\mathbf{r} = \mathbf{p}_1 \times \mathbf{p}_2$$



- Man kann um alle Achsen (bis auf eine) direkt rotieren:



X Y Z

- Verbesserungen:
  - "Spinning trackball" vermeidet häufiges Nachfassen
  - "Locking" für exaktes Rotieren um eine Koord.achse
  - Was macht man, wenn (x,y) die Ellipse verlassen?
    - Nichts(?) → z wird negativ → dann noch x,y am Kreis nach innen spiegeln → p liegt auf der Rückseite der Kugel

Zur Anatomie einer Matrix

- Erst Rotation, dann Translation:

$$P' = (TR)\,P = MP = R_{3x3} \cdot P + T$$

$$M = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{21} & R_{22} & R_{23} & 0 \\ R_{31} & R_{32} & R_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \left(\begin{array}{ccc|c} R_{11} & R_{12} & R_{13} & T_x \\ R_{21} & R_{22} & R_{23} & T_y \\ R_{31} & R_{32} & R_{33} & T_z \\ \hline 0 & 0 & 0 & 1 \end{array}\right) = \begin{pmatrix} R & T \\ 0 & 1 \end{pmatrix}$$

- Erst Translation, dann Rotation:

$$P' = (RT)\,P = MP \cong R\,(P+T) = RP + RT$$

$$M = \begin{pmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{21} & R_{22} & R_{23} & 0 \\ R_{31} & R_{32} & R_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} R_{3x3} & R_{3x3}\,T_{3x1} \\ 0_{1x3} & 1 \end{pmatrix}$$

- Allgemeiner Aufbau (vereinfacht!):

Rotationen

Skalierung

$$
\begin{pmatrix}
a & b & c & t_x \\
d & e & f & t_y \\
g & h & i & t_z \\
0 & 0 & 0 & 1
\end{pmatrix}
$$

Translation

Projektionen (später)

Starre Transformationen (*Rigid-Body Transform*)

- Starre Transformation (Euklidische Transf.) = Hintereinanderausführung von Translationen und Rotationen
- Erhält Längen und Winkel eines Objektes
  - Objekte werden nicht deformiert / verzerrt
- Allgemeine Form:

$$M = T_t R = \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Inverse Rigid-Body Transformation:

$$M^{-1} = (T_t R)^{-1} = R^{-1} T_t^{-1} = R^{\mathsf{T}} T_{-t}$$

$$M = \begin{pmatrix} R & t \\ 0^T & 1 \end{pmatrix} \qquad M^{-1} = \begin{pmatrix} R^{\mathsf{T}} & -R^{\mathsf{T}} t \\ 0 & 1 \end{pmatrix}$$

Klassifikation aller Transformationen

Euklidisch (starre)

Linear

Ähnlich-keiten

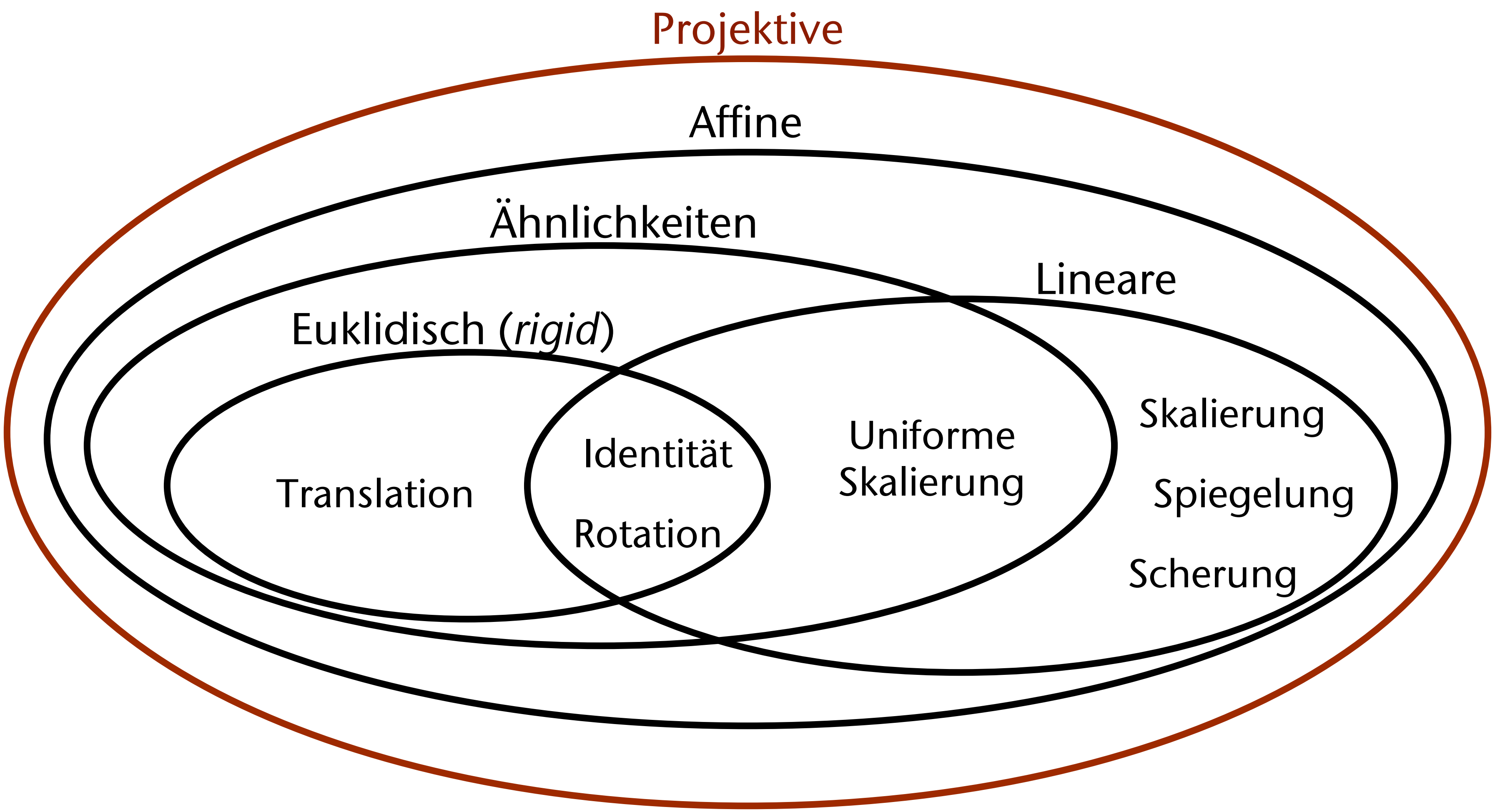
Projektive

Affine

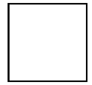
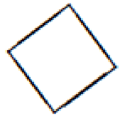
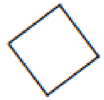
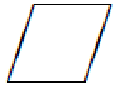
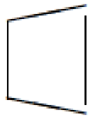
Erhält Winkel und Verhältnisse von Strecken,
kann aber die Länge von Strecken ändern

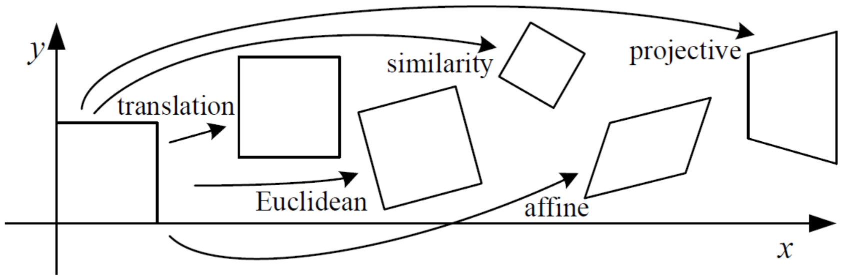
Translation | Identität | Rotation | Uniforme Skalierung | Skalierung | Spiegelung | Scherung

Klassifikation aller Transformationen



Eine Hierarchie von Transformationen (hier in 2D)

Transformation	Matrix	# DoF	Preserves	Icon
translation	$\left[\ \boldsymbol{I} \mid \boldsymbol{t}\ \right]_{2\times 3}$	2	orientation	
rigid (Euclidean)	$\left[\ \boldsymbol{R} \mid \boldsymbol{t}\ \right]_{2\times 3}$	3	lengths	
similarity	$\left[\ s\boldsymbol{R} \mid \boldsymbol{t}\ \right]_{2\times 3}$	4	angles	
affine	$\left[\ \boldsymbol{A}\ \right]_{2\times 3}$	6	parallelism	
projective	$\left[\ \tilde{\boldsymbol{H}}\ \right]_{3\times 3}$	8	straight lines	



Transformationen in OpenGL

- Einfache Befehle zur Objekttransformation:

```
glRotate{fd}( TYPE angle, x, y, z );
```

rotiert um **`angle`** Grad(!) um die angegebene Achse;

```
glTranslate{fd}( TYPE x,y,z );
```

transliert um den angegebenen Betrag;

```
glScale{fd}( TYPE x,y,z );
```

skaliert um die angegebenen Faktoren.

- Ein **glRotate** / **glTranslate** (u.ä.) wirkt sich nur auf die nachfolgende Geometrie aus!

Matrizen in OpenGL

- Es gibt eine „globale" Matrix „**`MODELVIEW`**", die anfangs mit der Einheitsmatrix besetzt ist
- Jeder Aufruf von **`glRotate`**, **`glScale`** etc. resultiert in der Multiplikation der entsprechenden Matrix mit der „globalen" Matrix von **rechts**, z.B.

```
glScalef( sx,sy,sz )
```

$$\Longleftrightarrow \quad M_{\text{MODELVIEW}} \cdot \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```
glTranslatef( tx,ty,tz )
```

$$\Longleftrightarrow \quad M_{\text{MODELVIEW}} \cdot \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Beachte die Reihenfolge in einer Matrixkette:

Reihenfolge der OpenGL-Befehle

$$p' = M_n \cdot \ldots \cdot M_2 \cdot M_1 \cdot p$$

Reihenfolge der Ausführung

- Die Anordnung entspringt aus dem Programmablauf
- Konzeptionell kann man es sich wie folgt vorstellen:

```
glScalef(1.5,1,1);
glTranslatef(.2,0,0);
glRotatef(30,0,0,1);
render geometry
```

„Die Geometrie wandert rückwärts durch das Programm und sammelt die Transformationen ein“

Direkte Matrizenspezifizierung

- Man kann auch direkt Matrizen als Trafo's angeben:

```
glMultMatrix{fd}( TYPE * m );
```

multipliziert die Matrix auf die aktuelle **`MODELVIEW`**-Matrix;

```
glLoadMatrix{fd}( TYPE * m );
```

ersetzt die aktuelle **`MODELVIEW`**-Matrix durch die angegebene;

```
glLoadIdentity();
```

Spezialfall: lädt die Einheitsmatrix.

- Matrixabfrage (sehr langsam):

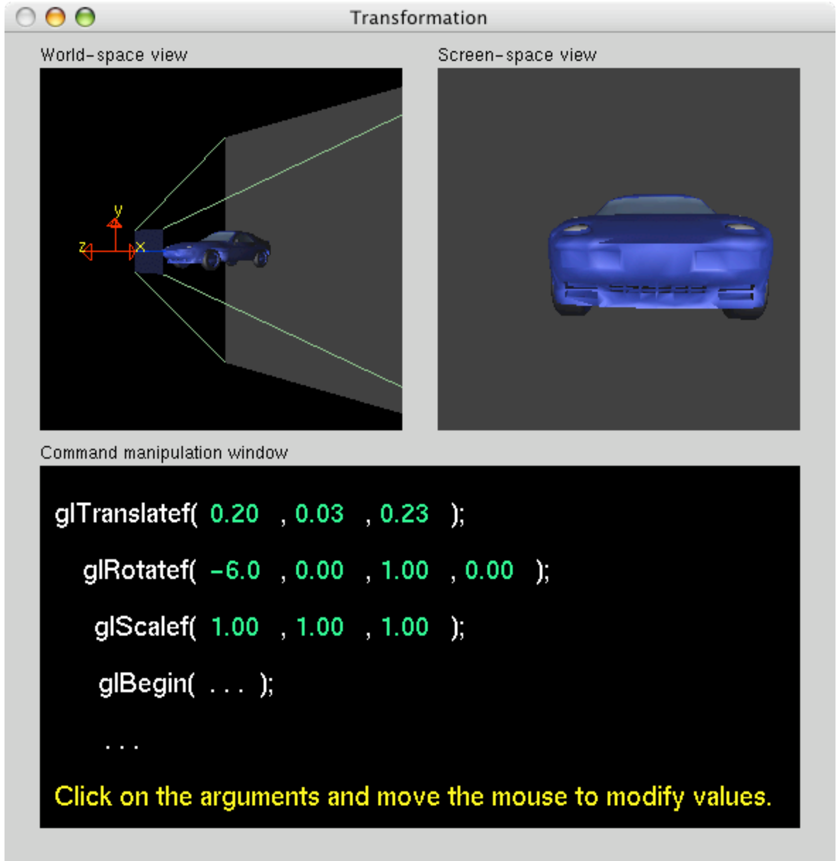
```
glGetFloatv( GL_MODELVIEW_MATRIX, float * m );
```

- Achtung: Matrizen werden **spaltenweise** abgelegt, nicht — wie in C üblich — zeilenweise!
  - Das nent sich *"column-major order"* (der Standard, z.B. in C, ist *row-major order*)

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \Longleftrightarrow$$

```
GLfloat matrix[] =
{
        1, 0, 0, 0,
        0, 1, 0, 0,
        0, 0, 1, 0,
        tx, ty, tz, 1
};
```

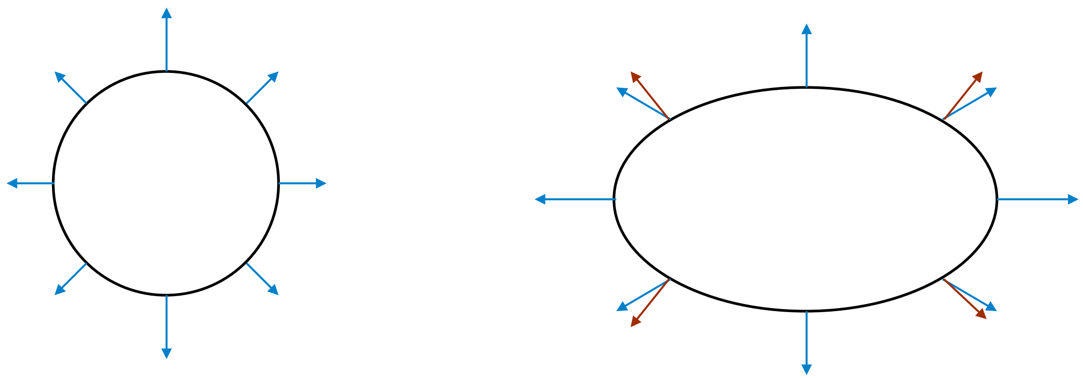
Demo ...



http://www.xmission.com/~nate/tutors.html

Transformation von Normalen

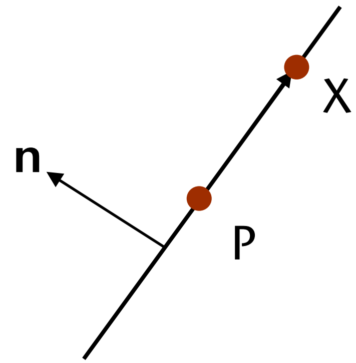
- Behauptung:
  Wenn das Objekt um M transformiert wird, dann müssen die Normalen der Oberfläche um $N = (M^{-1})^T$ transformiert werden
- Bei starren (euklidischen) Transformationen:
  - Translation beeinflusst die Normalen der Oberfläche nicht
  - Im Fall der Rotation ist $M^{-1} = M^T$ und somit $N = M$
- Bei nicht-uniformer Skalierung und Scherung ist $N = (M^{-1})^T \neq M$!
  - Beispiel:



Beweis

- Wir wissen:

$$(X - P)^{\mathsf{T}}\mathbf{n} = 0$$

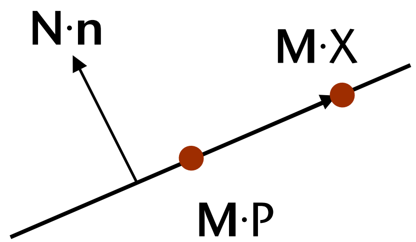


- Gesucht ist *N*, so daß:

$$(M\cdot X - M\cdot P)^{\mathsf{T}}\cdot(N\cdot\mathbf{n}) = (X - P)^{\mathsf{T}}\cdot M^{\mathsf{T}}\cdot N\cdot\mathbf{n} = 0$$

- Setze also

$$N = \left(M^{\mathsf{T}}\right)^{-1}$$

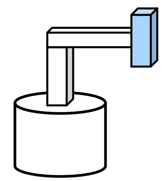


- Damit ist

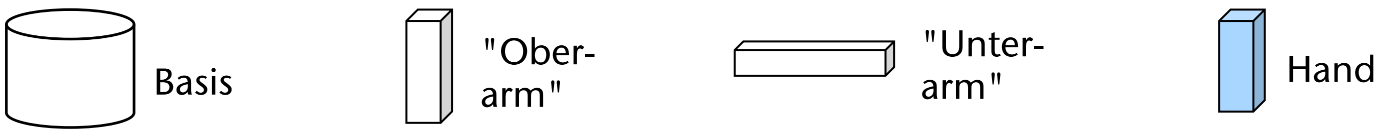
$$(X - P)^{\mathsf{T}}\cdot M^{\mathsf{T}}(M^{\mathsf{T}})^{-1}\cdot\mathbf{n} = (X - P)^{\mathsf{T}}\cdot I\cdot\mathbf{n} = 0$$

Relative Transformationen

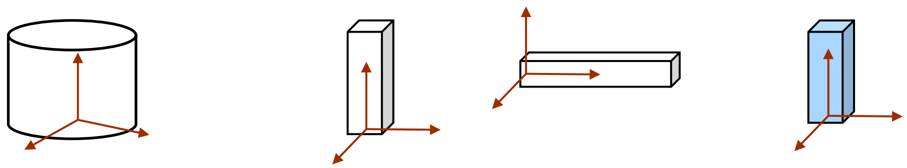
- Eine Konkatenierung von Transformationen kann man auch als eine Folge von (voneinander abhängigen) Koordinatensystemen ansehen
- Beispiel: Roboter



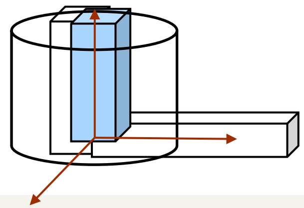
  - Besteht aus diesen Einzelteilen



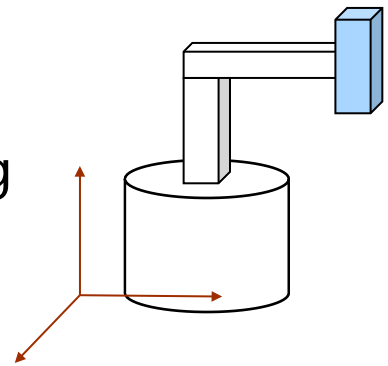
  - Jedes Teil wurde in seinem eigenen Koordinatensystem spezifiziert (als Array von Punkten) → heißt Objektkoordinatensystem



  - Rendert man alle Teile ohne jede Transformation, entsteht folgendes:



- Würde man jedes Teil, ausgehend vom Ursprung des Weltkoordinatensystems, an seinen Platz transformieren, sähe das ungefähr so aus:



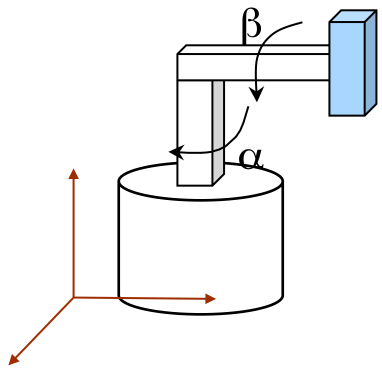
```
// set up camera
[...]
// render robot
glLoadIdentity();
glTranslatef( robot_pos_x, robot_pos_y , … );
render base …

glLoadIdentity();
glTranslatef( robot_pos_x, robot_pos_y + 10, … );
render upper arm …

glLoadIdentity();
glTranslatef( robot_pos_x, robot_pos_y + 10 + 5, … );
render lower arm …
. . .
```

Ann.: Höhe der Basis ist 10

Ann.: Höhe des Oberarms ist 5

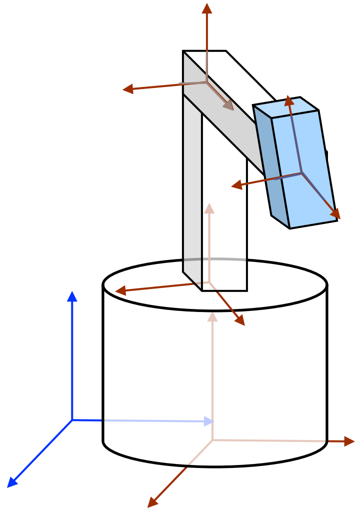
- Natürlich macht man es ungefähr so:

```
glLoadIdentity();

glTranslatef( robot_pos_x, robot_pos_y , … );
render base …

glTranslatef( 0, HEIGHT_BASE, 0 );
glRotatef( alpha, 0, 1, 0 );
render upper arm …

glTranslatef( 0, LEN_UPPER_ARM, 0 );
glRotatef( beta, 1, 0, 0 );
render lower arm …

glTranslatef( X_LEN_LOWER_ARM, 0, 0 );
render hand …
```



Solche Parameter würde man natürlich in einer Klasse 'Roboter' als Instanzvariablen speichern

- Alternative Betrachtungsweise ist, daß bei jeder Transformation ein neues lokales Koordinatensystem entsteht, das bezüglich seines Vater-Koordinatensystems um genau diese Transf. transformiert ist



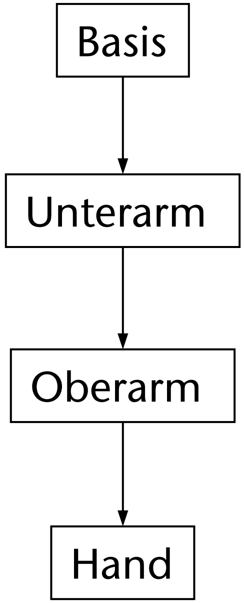
In dieser Reihenfolge entstehen die lokalen Koordinatensysteme aus dem Weltkoordinatensystem

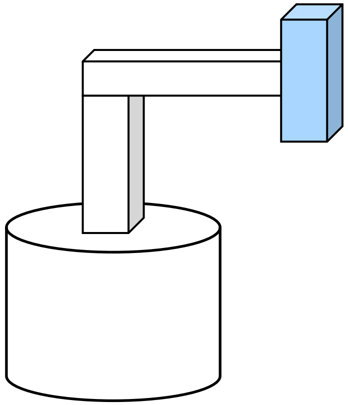
```
glLoadIdentity();
glTranslatef( robot_pos_x, robot_pos_y ,
… );
render base …
glTranslatef( 0, HEIGHT_BASE, 0 );
glRotatef( alpha, 0, 1, 0 );
render upper arm …
glTranslatef( 0, HEIGHT_UPPER_ARM, 0 );
glRotatef( beta, 1, 0, 0 );
render lower arm …
glTranslatef( X_SIZE_LOWER_ARM, 0, 0 );
render hand …
```

In dieser Reihenfolge werden die Transformationen auf die Geometrie (d.h., die Punkte) angewendet

Objekthierarchien

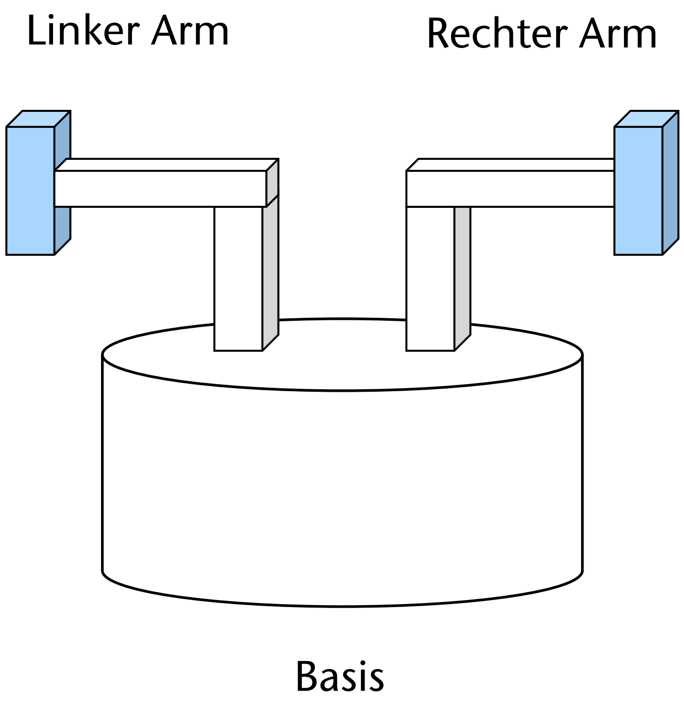
- Dadurch ergibt sich eine Abhängigkeit der Objekte
  - Sie betrifft vor allem deren Transformationen
  - Betrifft später auch andere Attribute (z.B. Farbe)
- Der so definierte Baum heißt Szenengraph

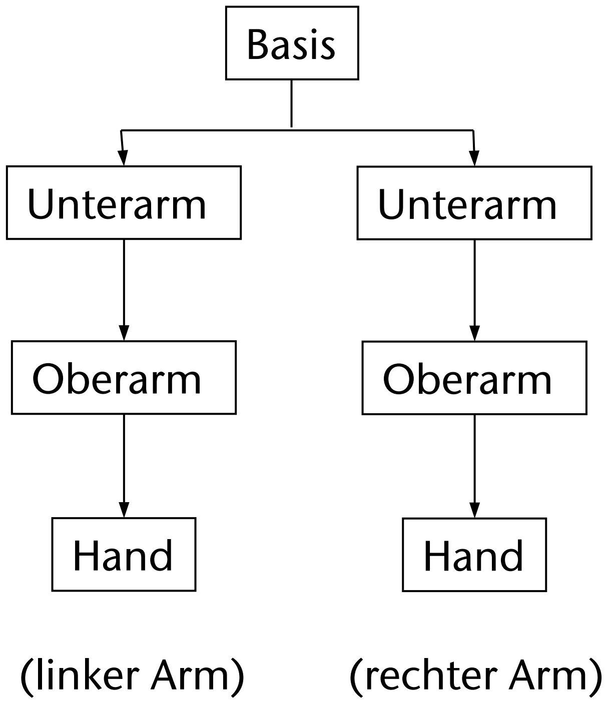




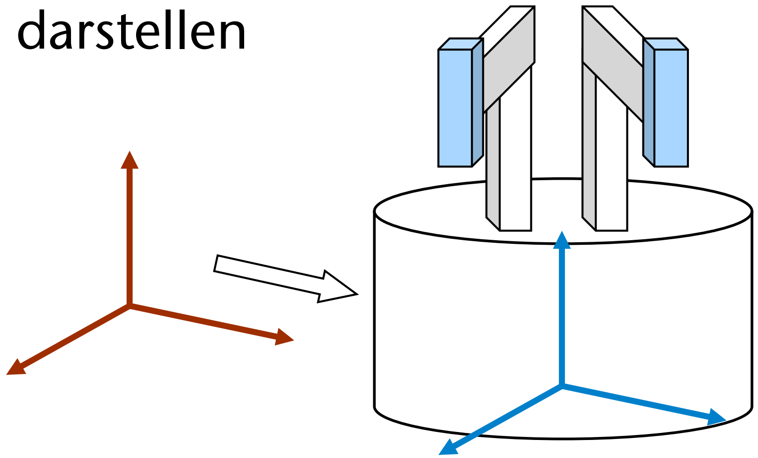
- Bemerkung: wir werden in "Computergraphik 1" Szenengraphen noch nicht explizit darstellen

- Ein etwas komplizierteres Beispiel:

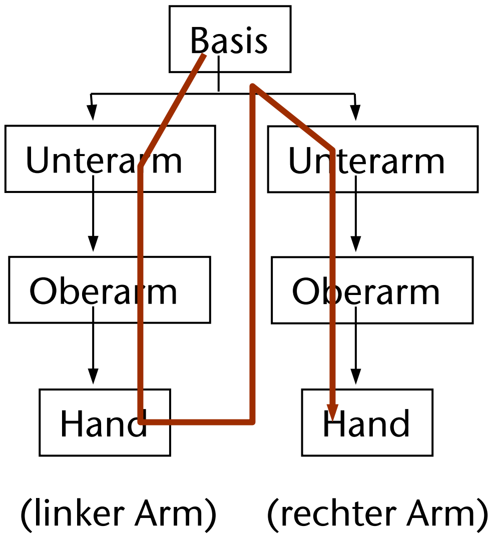




- Aufgabe: folgende Konfiguration darstellen



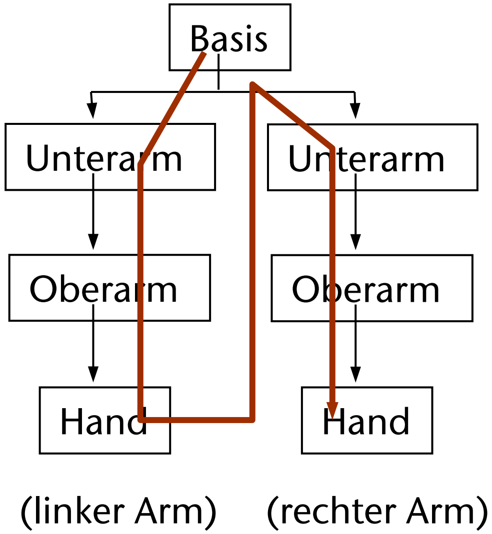
- Natürliche Vorgehensweise ist Depth-First-Traversal durch den Szenengraph:

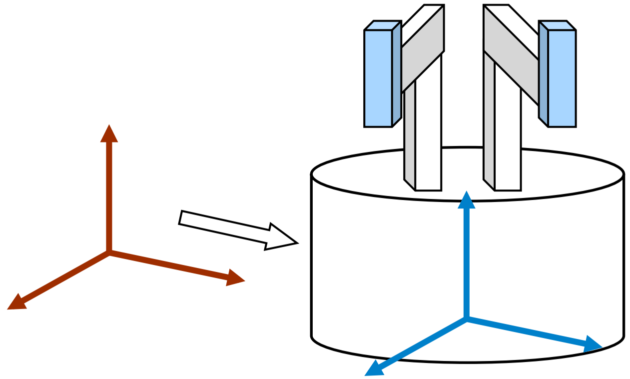



```
Do transformation(s)

Draw base

Do transformation(s)

Draw left arm

Do transformation(s)

Draw right arm
```

Welche sind das ??

Erster (falscher) Versuch





```
Translate(5,0,0)
Draw base
Rotate(75, 0, 1, 0)
Draw left arm
Rotate(-75, 0, 1, 0)
Draw right arm
```

Was ist hier falsch?!

Antwort: der rechte Arm soll relativ zur Basis um -75 Grad gedreht sein, in diesem Programm aber wird er relativ zum linken Arm gedreht!

Lösung

```
Initiale MODELVIEW Matrix M
```
Translate(5,0,0) → M = M·T
```
Draw base
```
Rotate(75, 0, 1, 0)
```
Draw left arm
```
Rotate(-75, 0, 1, 0)
```
Draw right arm
```

Speichere die MODELVIEW-Matrix an dieser Stelle in einem Zwischenspeicher

Restauriere diese gemerkte MODELVIEW-Matrix an dieser Stelle aus dem Zwischenspeicher

⇒ **Lösung: ein Matrix-Stack**

```
Initiale MODELVIEW Matrix M
```
Translate(5,0,0) → M = M·T
```
Draw base
```
Rotate(75, 0, 1, 0)
```
Draw left arm
```
Rotate(-75, 0, 1, 0)
```
Draw right arm
```

An dieser Stelle die aktuelle MODELVIEW-Matrix auf den Stack pushen

An dieser Stelle die oberste Matrix vom Stack pop-en und in die MODELVIEW-Matrix schreiben

Der Matrix-Stack in OpenGL

- In OpenGL gibt es einen MODELVIEW-Matrix-Stack
- Die oberste Matrix auf diesem Stack ist die aktuelle MODELVIEW-Matrix, die für die Geometrie-Transformation verwendet wird
- Alle Transformations-Kommandos (glLoadMatrix, glMultMatrix, glTranslate, ...) operieren auf dieser obersten Matrix!
- OpenGL-Befehle:

```
glPushMatrix();
```

dupliziert die oberste Matrix auf dem Stack und legt diese oben auf dem Stack ab;

```
glPopMatrix();
```

wirft die oberste Matrix vom Stack weg.

Beispiel

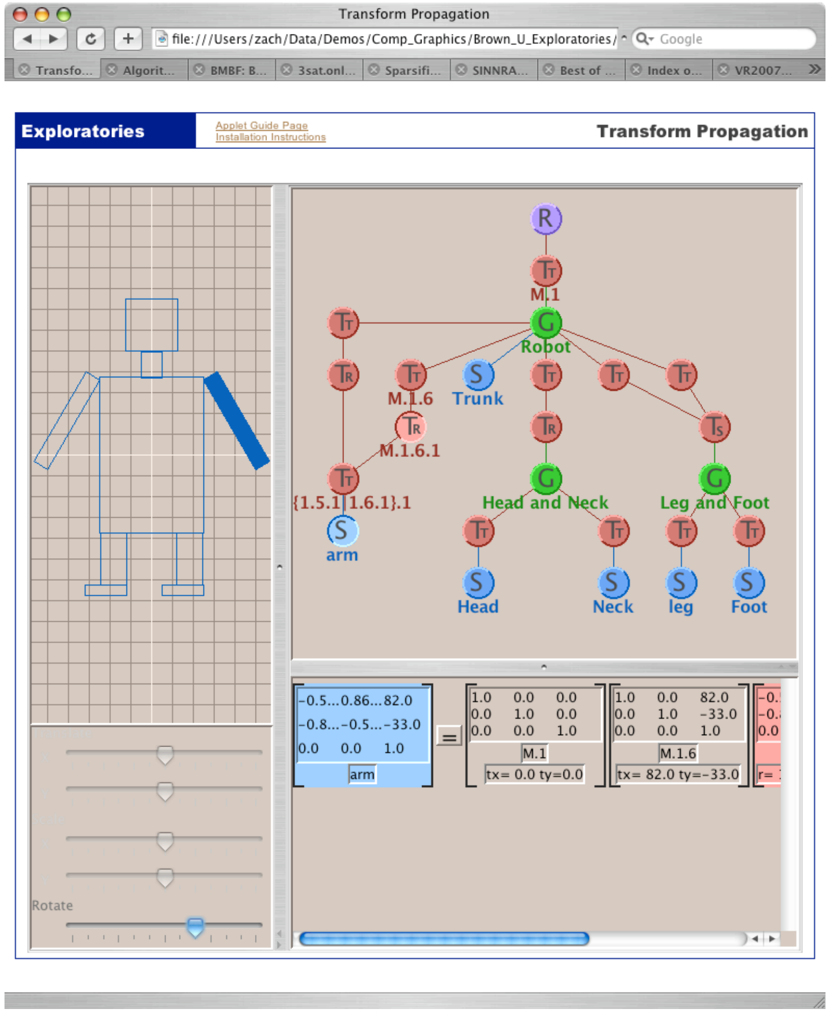
Code	Aktuelle MODELVIEW-Matrix:	Zustand des Matrix-Stacks:		
`glMatrixMode(GL_MODELVIEW);`				
`glLoadIdentity();`	I	I		
`glMultMatrix( M1 );`	M1	M1		
`glTranslate( T );`	M1·T	M1·T		
`glPushMatrix();`	M1·T	M1·T	M1·T	
`glRotate( R );`	M1·T·R	M1·T	M1·T·R	
`glPushMatrix();`	M1·T·R	M1·T	M1·T·R	M1·T·R
`glMultMatrix( M2 );`	M1·T·R·M2	M1·T	M1·T·R	M1·T·R·M2
`glPopMatrix();`	M1·T·R	M1·T	M1·T·R	
`glScale( S );`	M1·T·R·S	M1·T	M1·T·R·S	
`glPopMatrix();`	M1·T·R	M1·T		

Beispiel im Code

glwidget.cpp

```cpp
    m_lastPos = e->pos();
}


void GLWidget::mouseMoveEvent( QMouseEvent * e )
{
    int dx = e->x() - m_lastPos.x();
    int dy = e->y() - m_lastPos.y();

    bool ctrl_key = e->modifiers() & Qt::MetaModifier;      // only needed for Mac OS X, but doesn't hurt on other OSes

    if ( (e->buttons() & Qt::RightButton) ||
         ctrl_key )
    {
        // setXRotation(m_xRot + 8 * dy);
        // setZRotation(m_zRot + 8 * dx);
        m_zTrans += 0.5 * dy;
        m_xTrans += 0.5 * dx;
    }
    else if (e->buttons() & Qt::LeftButton)
    {
        setXRotation(m_xRot + 8 * dy);
        setYRotation(m_yRot + 8 * dx);
    }
    m_lastPos = e->pos();
    e->accept();
    updateGL();
}



/**  Render a "sphere flake"
*
* @param scaling    scaling to be applied with each recursion
* @param n_recursions   number of recursions for the flake
* @param lati,longi number of latitudes and longitudes
* @param radius     radius of the sphere
*
* @bug
*   This produces spheres that are inside the larger ones (two levels up in the recursions hierarchy)
**/


void GLWidget::renderSphereFlake( const float scaling, unsigned int n_recursions ) const
{
    glCallList(m_object);

    if ( n_recursions <= 1 )
        return;

    glPushMatrix();
    glRotatef( m_curr_rot, 1.0, 1.0, 1.0 );
    glTranslatef( 1.5, 0.0, 0.0 );
    glScalef( scaling, scaling, scaling );
    renderSphereFlake( scaling, n_recursions-1 );
    glPopMatrix();

    glPushMatrix();
    glRotatef( m_curr_rot, 1.0, 1.0, 1.0 );
    glTranslatef( -1.5, 0.0, 0.0 );
    glScalef( scaling, scaling, scaling );
```

Demo zum Szenengraph



http://www.cs.brown.edu/exploratories → Transformation Propagation