Zur Anatomie einer Matrix

- Erst Rotation, dann Translation:

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

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

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

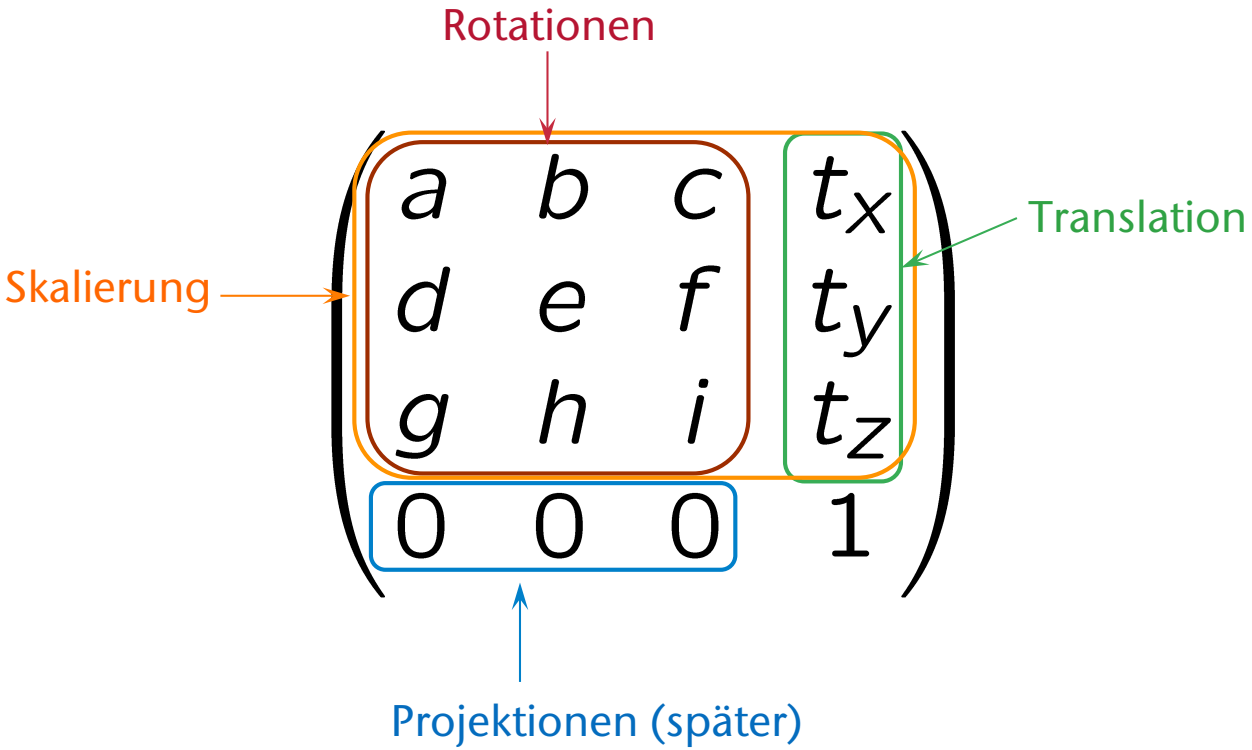
- Erst Translation, dann Rotation:

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

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

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

- Allgemeiner Aufbau (vereinfacht!):



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:

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

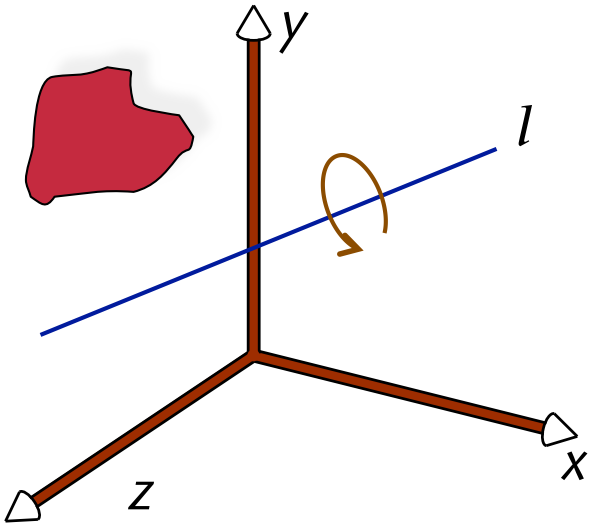
- Inverse Rigid-Body Transformation:

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

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

Alternative Rotation um eine beliebige Achse in 3D

- Man möchte mit θ um die Gerade l rotieren

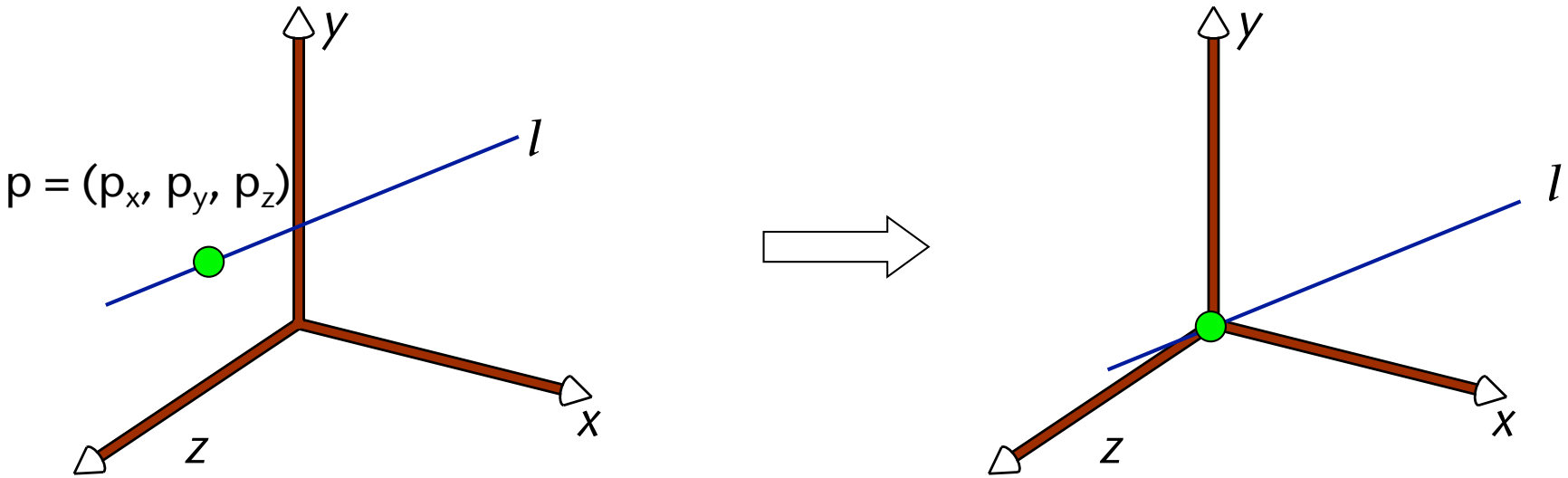


- Gesucht: eine Matrix *M*, die diese Transformation enthält
- Wir wissen, wie man um eine Koordinatenachse rotiert
- Somit müssen wir die Szene in eine Situation transformieren, mit der wir umgehen können

- Grundidee:
  1. Verschiebe einen Punkt der Geraden in den Ursprung
  2. Rotiere um eine Achse, so daß l in einer Koordinatenebene liegt
  3. Rotiere um eine weiter Achse, so daß l auf einer Koordinatenachse liegt
  4. Rotiere um diese Achse mit θ
  5. Invertierte Rotation um die Koordinatenachse aus Schritt 3
  6. Invertierte Rotation um die Koordinatenachse aus Schritt 2
  7. Invertiere Verschiebung aus Schritt 1, so daß l wieder in Ausgangsposition

Schritt 1

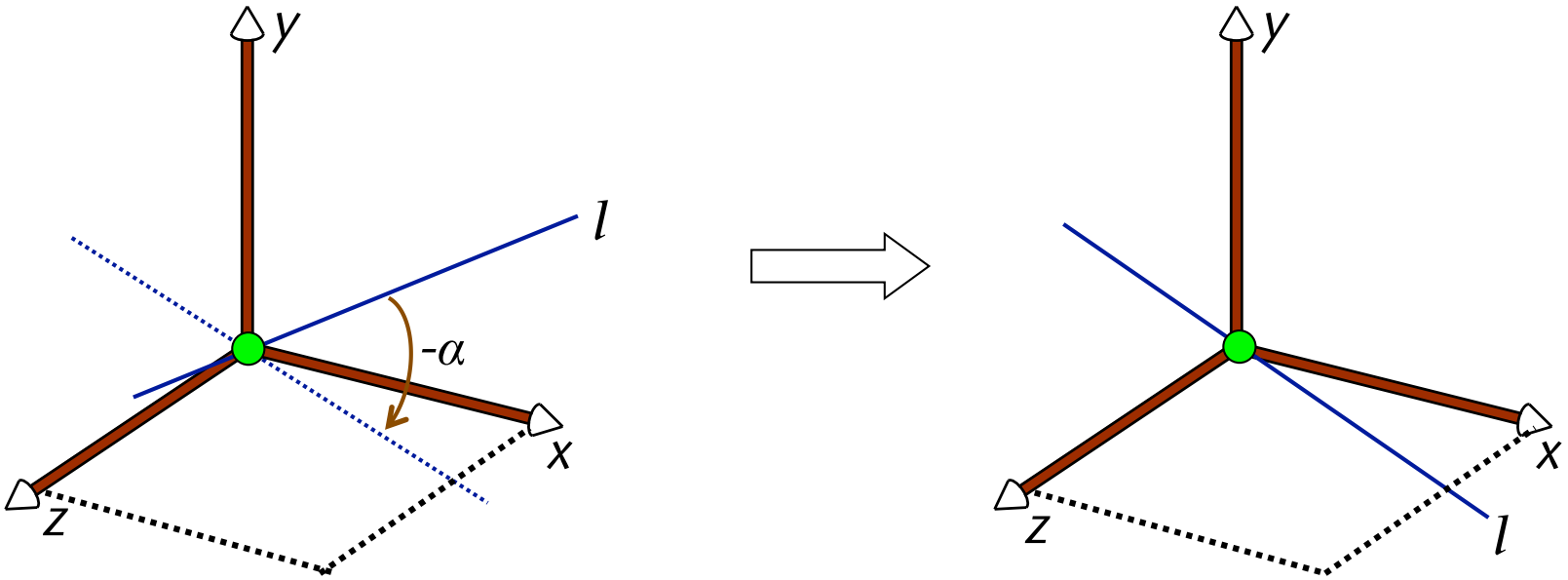
- Verschiebe Gerade, so daß ein Punkt im Ursprung liegt:



$$T_1 = \begin{pmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Schritt 2

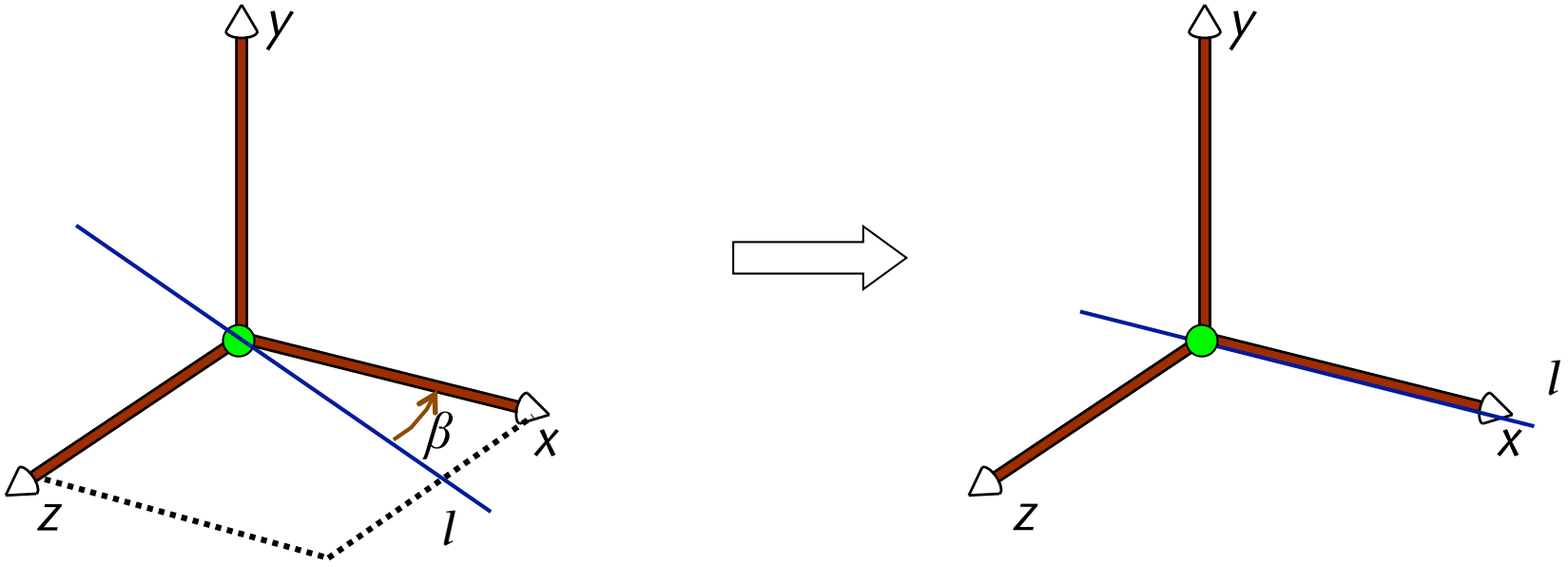
- Rotiere, so daß *l* in einer Koordinatenebene liegt
- Z.B.: rotiere mit -α um die z-Achse, so daß *l* in der xz-Ebene liegt



$$R_z(-\alpha) = \begin{pmatrix} \cos(-\alpha) & -\sin(-\alpha) & 0 & 0 \\ \sin(-\alpha) & \cos(-\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) & 0 & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Schritt 3

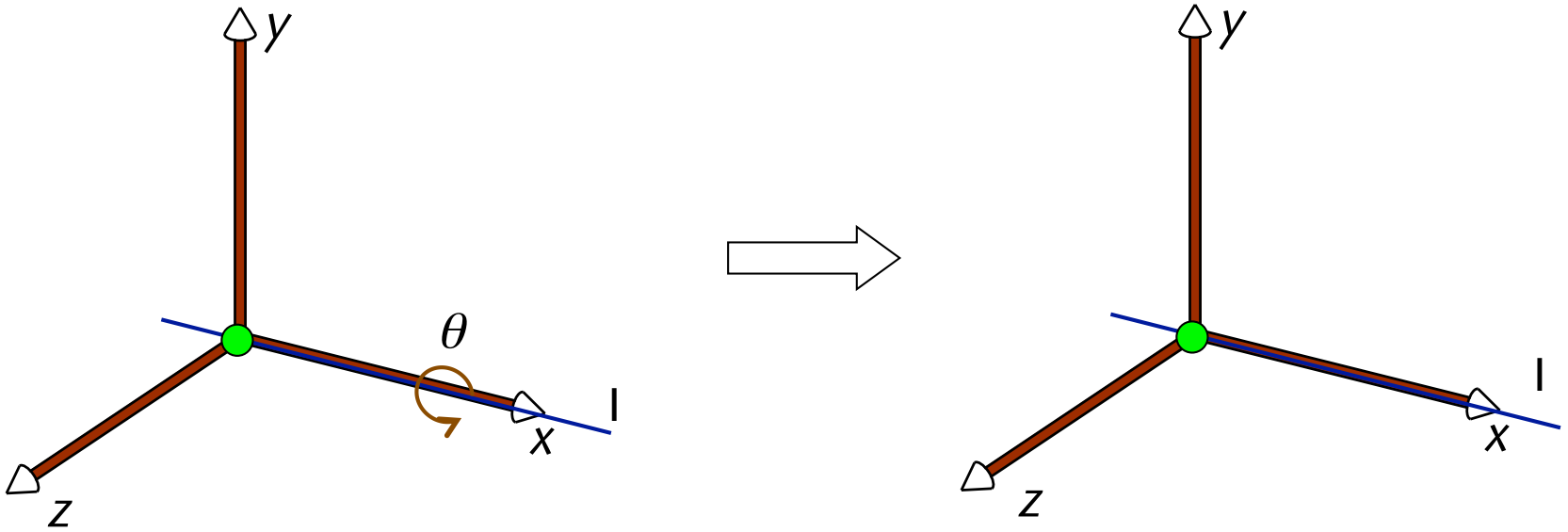
- Rotiere, so daß *l* auf einer Koordinatenachse liegt
- Hier: rotiere mit β um y-Achse damit Gerade auf der x-Achse liegt



$$R_y(\beta) = \begin{pmatrix} \cos(\beta) & 0 & \sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Schritt 4

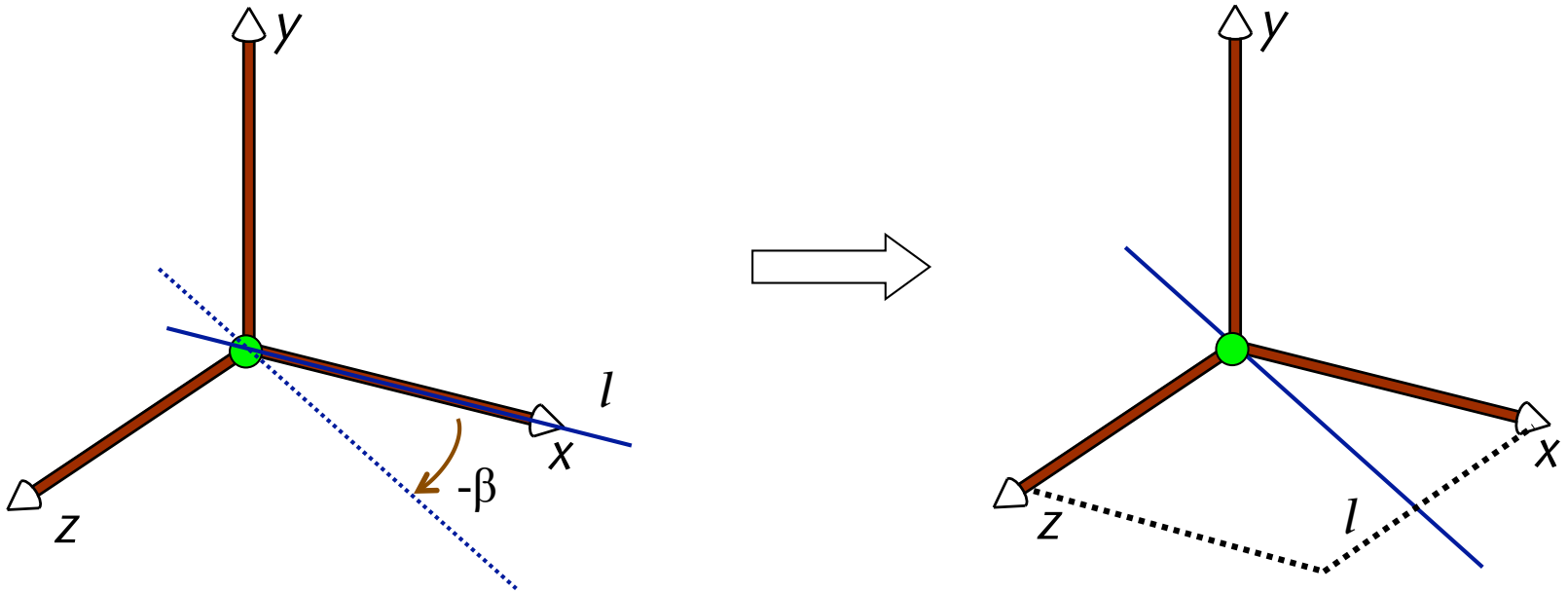
- Durchführen der gewünschten Rotation (rotiere mit θ um x-Achse)



$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Schritt 5

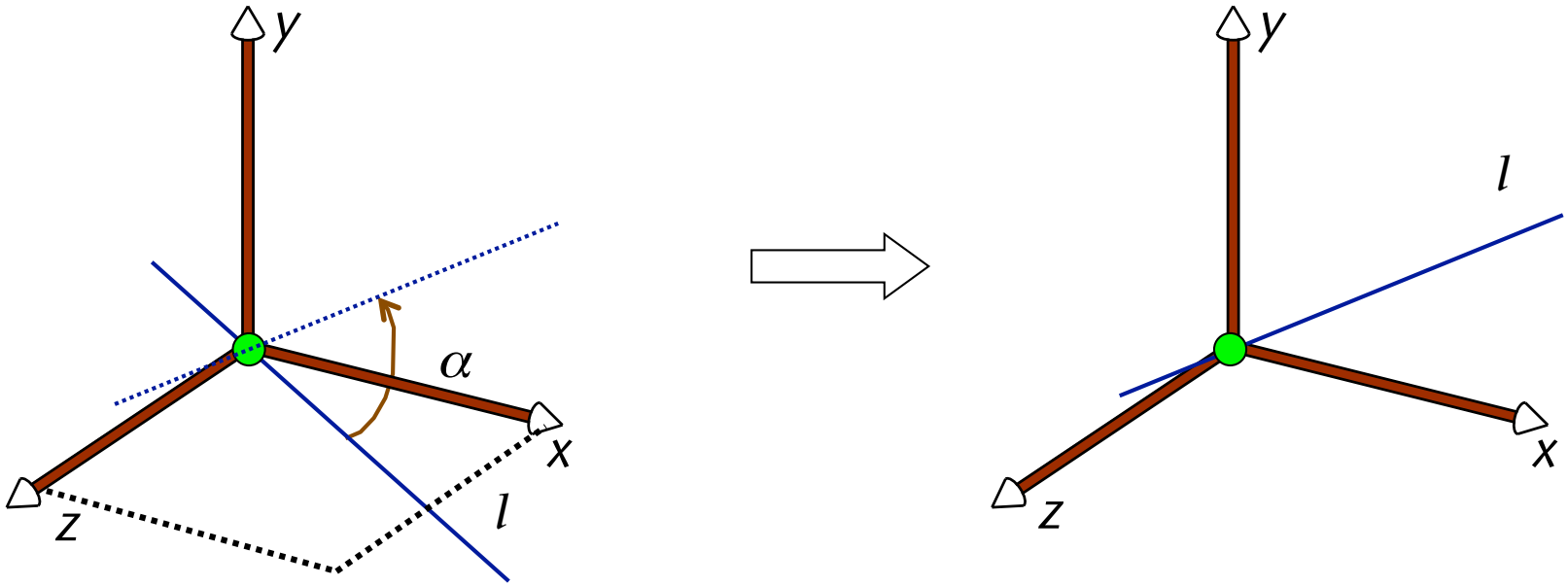
- Invertiere Rotation von l aus Schritt 3: rotiere mit -β um die y-Achse



$$R_y(-\beta) = \begin{pmatrix} \cos(-\beta) & 0 & \sin(-\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(-\beta) & 0 & \cos(-\beta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos(\beta) & 0 & -\sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\beta) & 0 & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Schritt 6

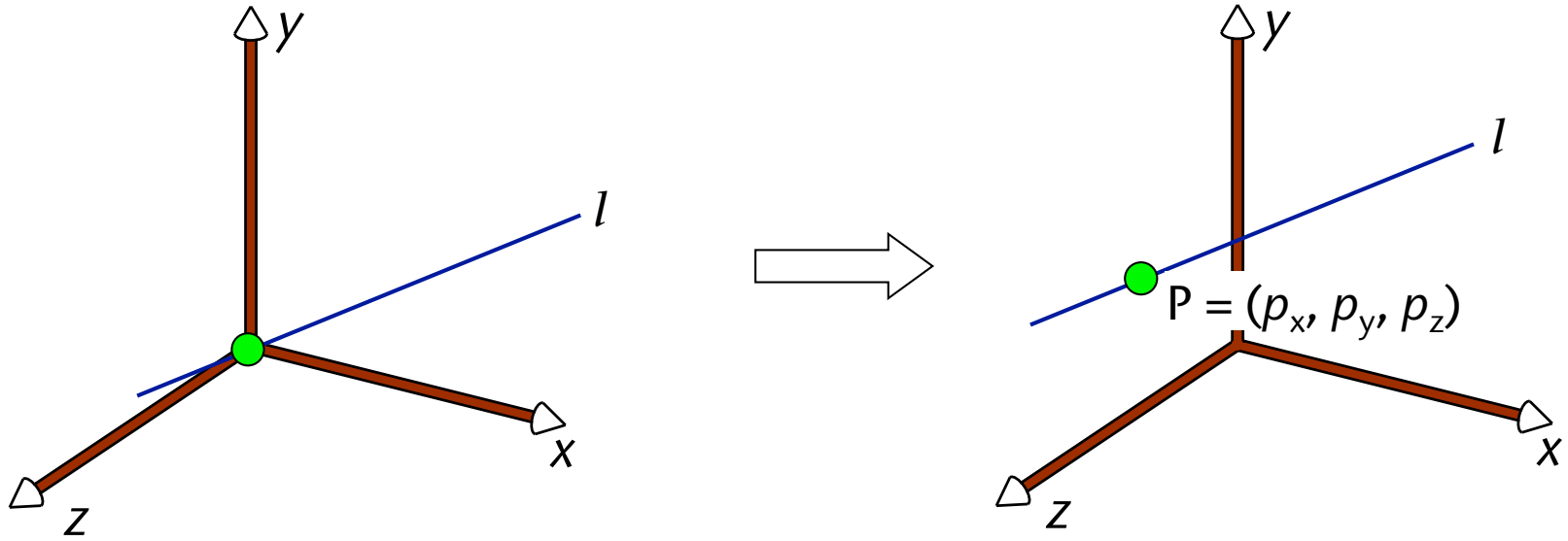
- Invertiere Rotation aus Schritt 2: rotiere mit α um z-Achse



$$R_z(\alpha) = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Schritt 7

- Invertiere die Translation aus Schritt 1



$$T_2 = \begin{pmatrix} 1 & 0 & 0 & p_x \\ 0 & 1 & 0 & p_y \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

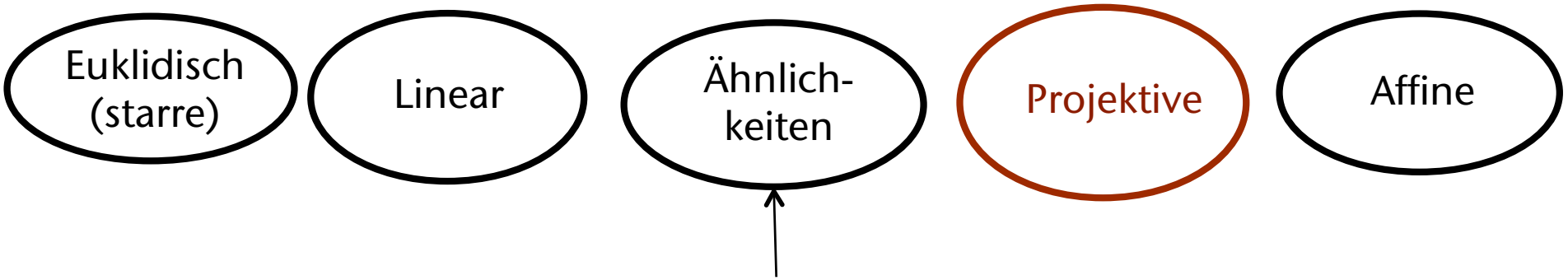
Zusammenfassung

- Die vollständige Transformation zum Rotieren um eine beliebige Achse ist:

$$R_{arb} = T_2\left(p_x, p_y, p_z\right) R_z\left(\alpha\right) R_y\left(-\beta\right) R_x\left(\theta\right) \cdot$$
$$R_y\left(\beta\right) R_z\left(-\alpha\right) T_1\left(-p_x, -p_y, -p_z\right)$$

  - Es gibt auch andere Varianten
- Hat man diese Matrix, so wendet man diese auf jeden Punkt des Objektes an, was den Effekt der Rotation dieses Objektes um die vorgegebene Achse hat
  - Das überläßt man natürlich OpenGL

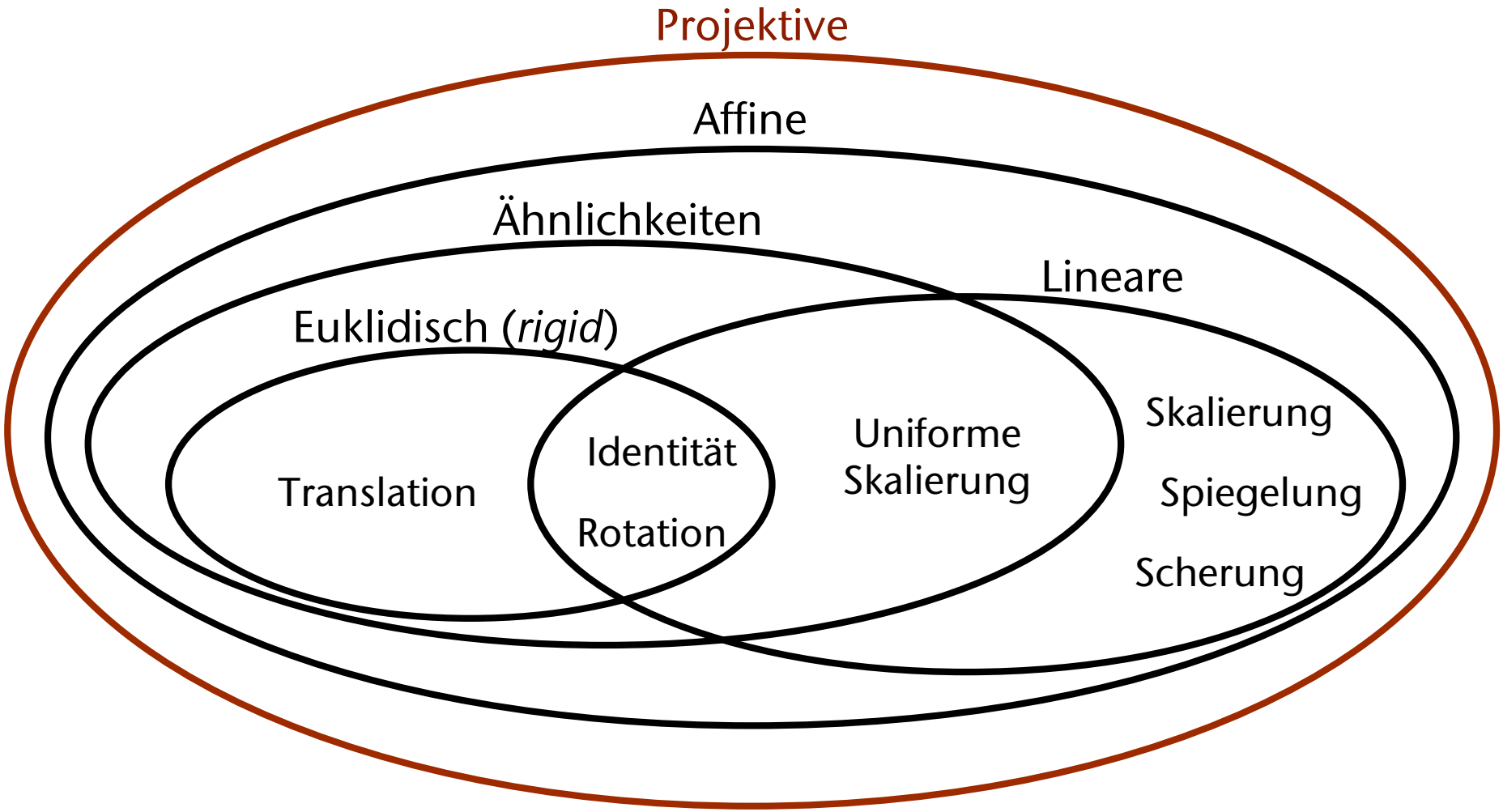
Klassifikation aller Transformationen



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

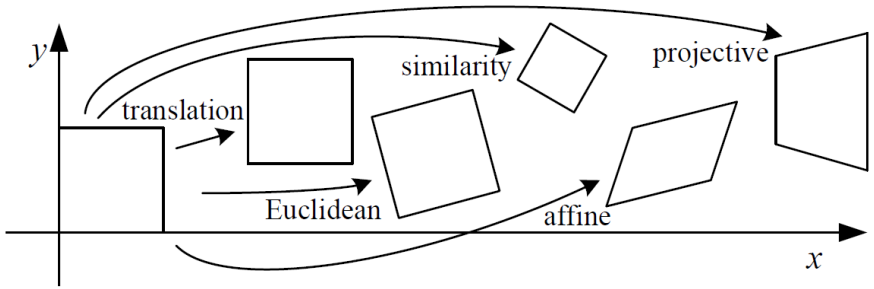
Translation Identität Rotation Uniforme Skalierung Skalierung Spiegelung Scherung

Klassifikation aller Transformationen



Eine Hierarchie von Transformationen (hier in 2D)

Transformation	Matrix	# DoF	Preserves	Icon
translation	$\begin{bmatrix} \boldsymbol{I} \mid \boldsymbol{t} \end{bmatrix}_{2\times3}$	2	orientation	
rigid (Euclidean)	$\begin{bmatrix} \boldsymbol{R} \mid \boldsymbol{t} \end{bmatrix}_{2\times3}$	3	lengths	
similarity	$\begin{bmatrix} s\boldsymbol{R} \mid \boldsymbol{t} \end{bmatrix}_{2\times3}$	4	angles	
affine	$\begin{bmatrix} \boldsymbol{A} \end{bmatrix}_{2\times3}$	6	parallelism	
projective	$\begin{bmatrix} \tilde{\boldsymbol{H}} \end{bmatrix}_{3\times3}$	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 )
```

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

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

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

- 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):

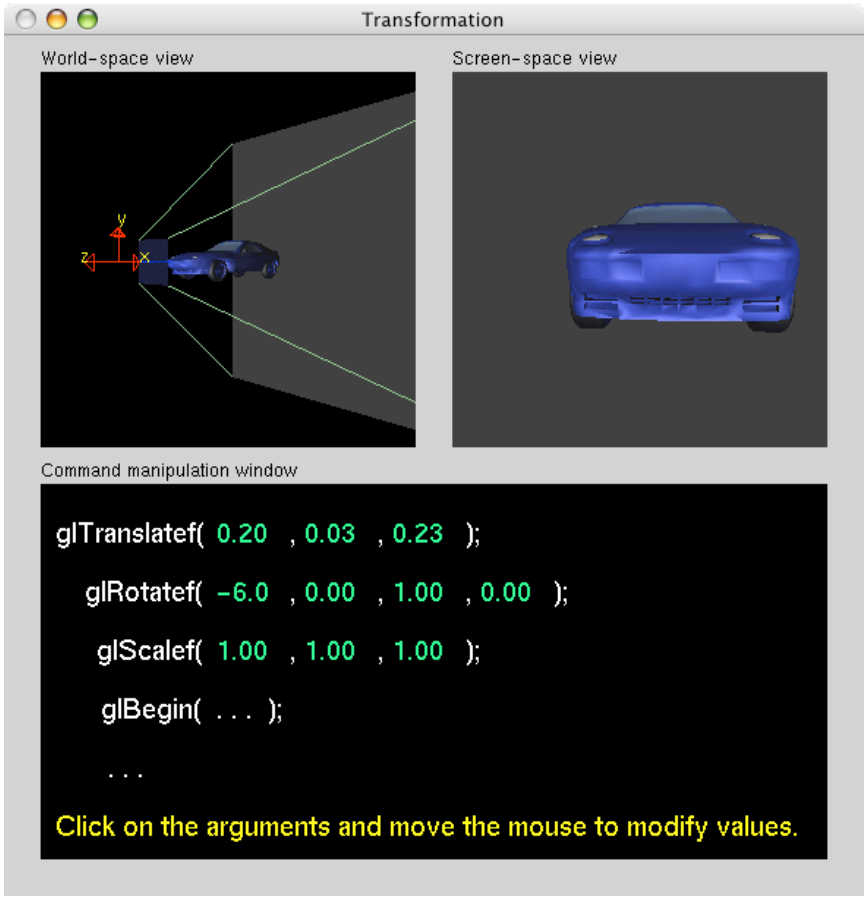
```
glGetFloatv( GL_MODELVIEW_MATRIX, float * m );
```

- 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*)

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

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

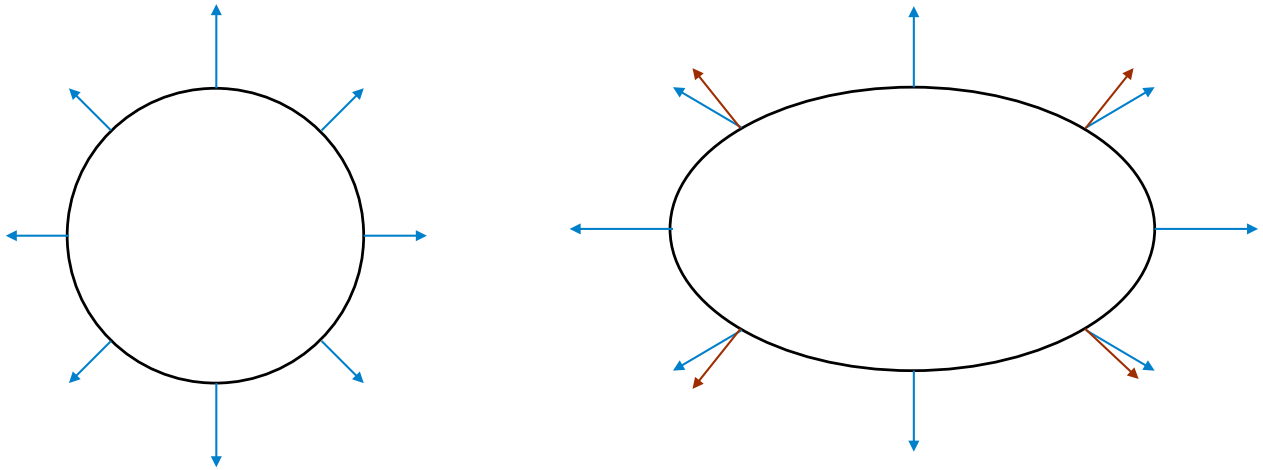
Demo ...



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

Transformation von Normalen

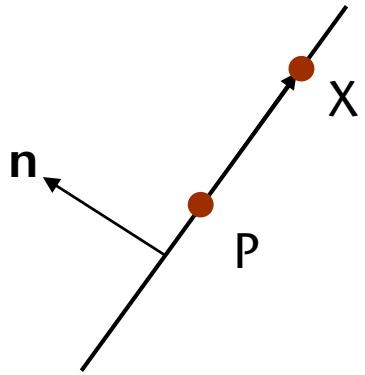
- Behauptung:
  Wenn ein Objekt um M transformiert wird, dann müssen die Normalen der Oberfläche um $N = (M^{-1})^T$ transformiert werden
- Bei starren (euklidischen) Transformationen:
  - Translation beeinflusst die Normalen der Oberfläche nicht
  - Im Fall der Rotation ist $M^{-1} = M^T$ und somit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



Beweis

- Wir wissen:

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

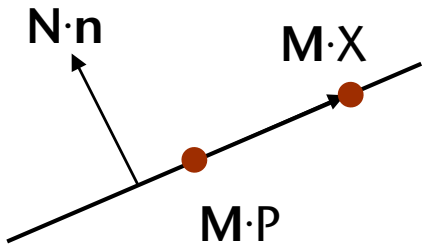


- Gesucht ist N, so daß:

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

- Setze also

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

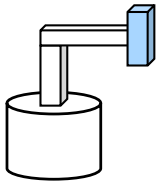


- Damit ist

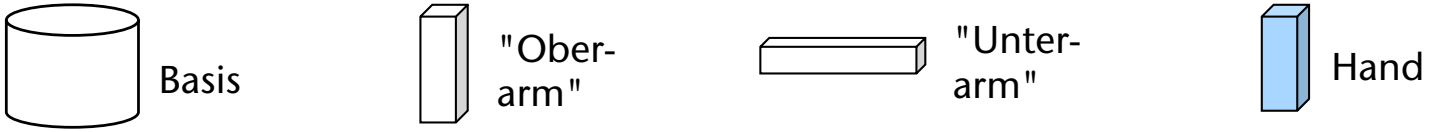
$$(X - P)^{\mathsf{T}} \cdot M^{\mathsf{T}} \left(M^{\mathsf{T}}\right)^{-1} \cdot \mathbf{n} = (X - P)^{\mathsf{T}} \cdot I \cdot \mathbf{n} = 0$$

Relative Transformationen

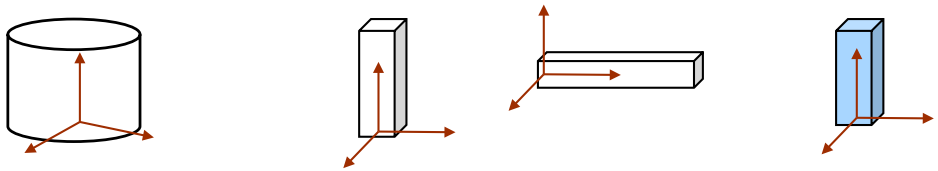
- Eine Konkatenierung von Transformationen kann man auch als eine Folge von (voneinander abhängigen) Koordinatensystemen ansehen
- Beispiel: Roboter



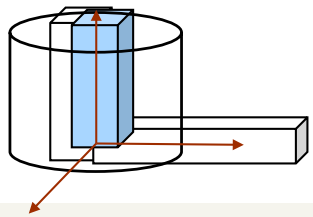
  - Besteht aus diesen Einzelteilen



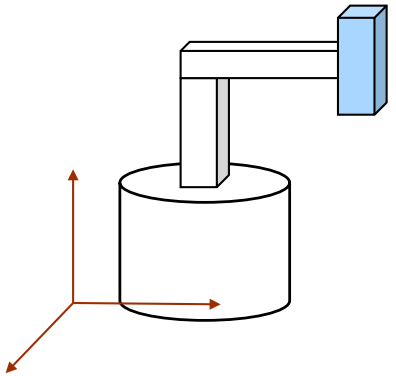
  - Jedes Teil wurde in seinem eigenen Koordinatensystem spezifiziert (als Array von Punkten) → heißt Objektkoordinatensystem



  - 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:



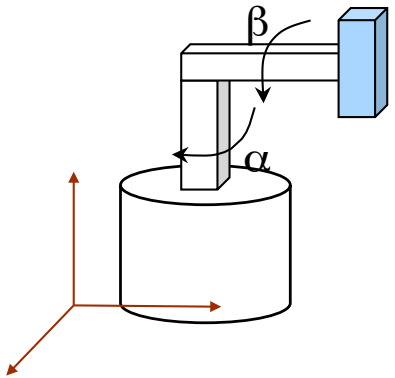
```
// set up camera
[...]
// render robot
glLoadIdentity();
glTranslatef( robot_pos_x, robot_pos_y , … );
render base …

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

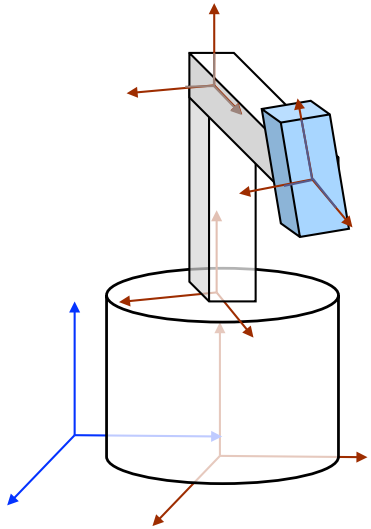
- Natürlich macht man es ungefähr so:

```
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( LEN_LOWER_ARM, 0, 0 );
render hand …
```

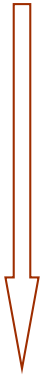


Solche Parameter würde man natürlich in einer Klasse 'Roboter' als Instanzvariablen speichern

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



In dieser Reihenfolge entstehen die lokalen Koordinaten-systeme aus dem Weltkoordinaten-system

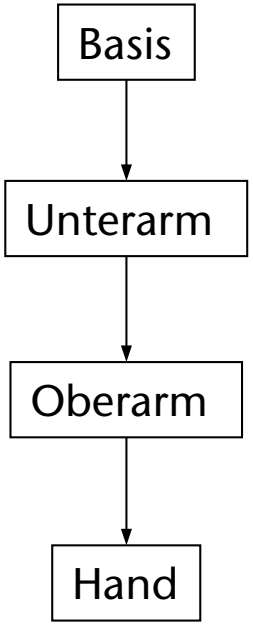


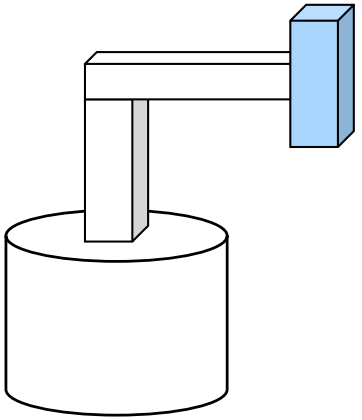
```
glLoadIdentity();
glTranslatef( robot_pos_x, robot_pos_y , … );
render base …
glTranslatef( 0, HEIGHT_BASE, 0 );
glRotatef( alpha, 0, 1, 0 );
render upper arm …
glTranslatef( 0, HEIGHT_UPPER_ARM, 0 );
glRotatef( beta, 1, 0, 0 );
render lower arm …
glTranslatef( X_SIZE_LOWER_ARM, 0, 0 );
render hand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

In dieser Reihenfolge werden die Transfor-mationen auf die Geometrie (d.h., die Punkte) angewendet

Objekthierarchien

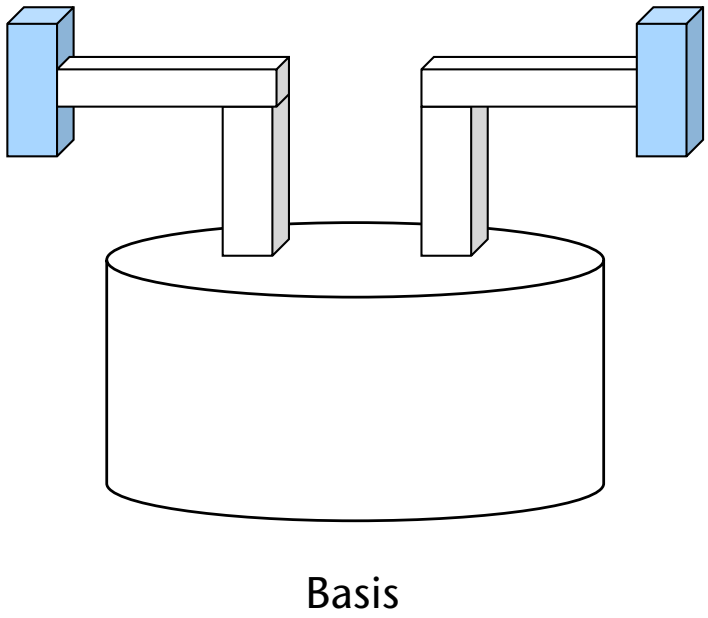
- Dadurch ergibt sich eine Abhängigkeit der Objekte
  - Sie betrifft vor allem deren Transformationen
  - Betrifft später auch andere Attribute (z.B. Farbe)
- Der so definierte Baum heißt Szenengraph

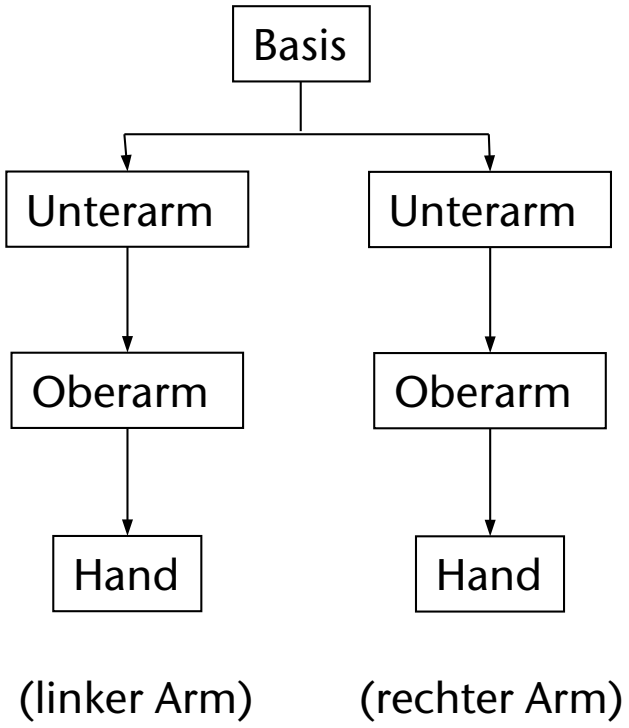




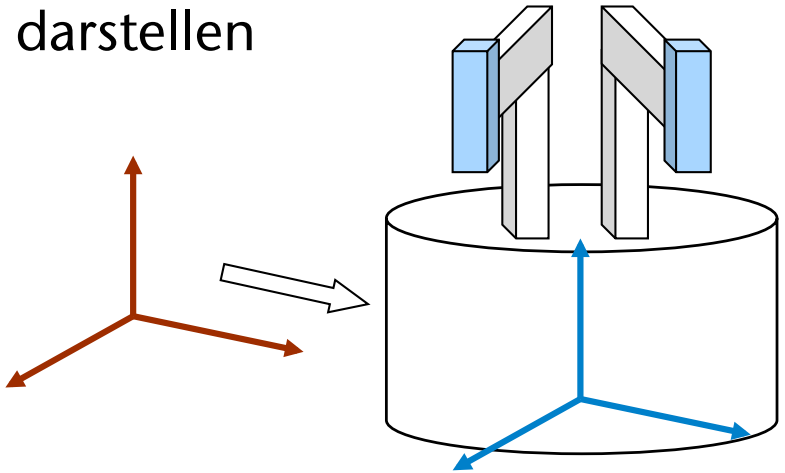
- Bemerkung: wir werden in "Computergraphik 1" Szenengraphen noch nicht explizit darstellen

- Ein etwas komplizierteres Beispiel:

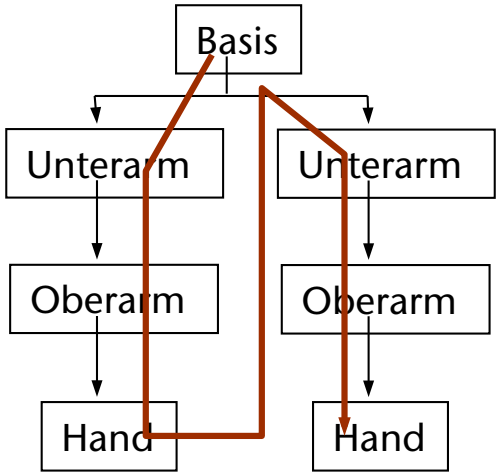




- Aufgabe: folgende Konfiguration darstellen



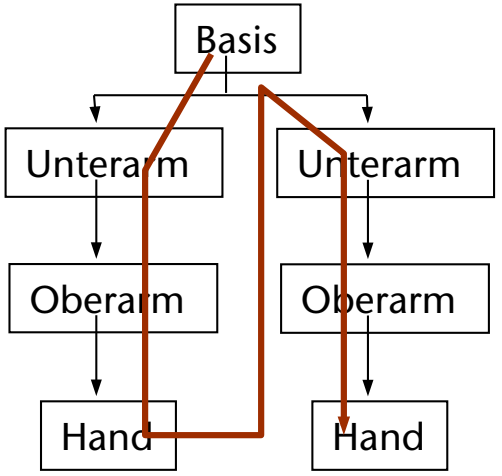
- Natürliche Vorgehensweise ist Depth-First-Traversal durch den Szenengraph:

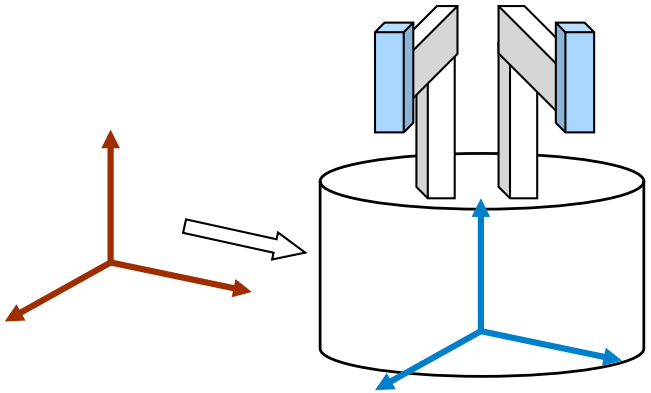



```
Do transformation(s)

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! (und würde außerdem noch an einer völlig falschen Position im Raum erscheinen!)

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

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrix( M1 );
glTranslate( T );
glPushMatrix();
glRotate( R );
glPushMatrix();
glMultMatrix( M2 );
glPopMatrix();
glScale( S );
glPopMatrix();
```

Aktuelle MODELVIEW-Matrix:

I

M1

M1·T

M1·T

M1·T·R

M1·T·R

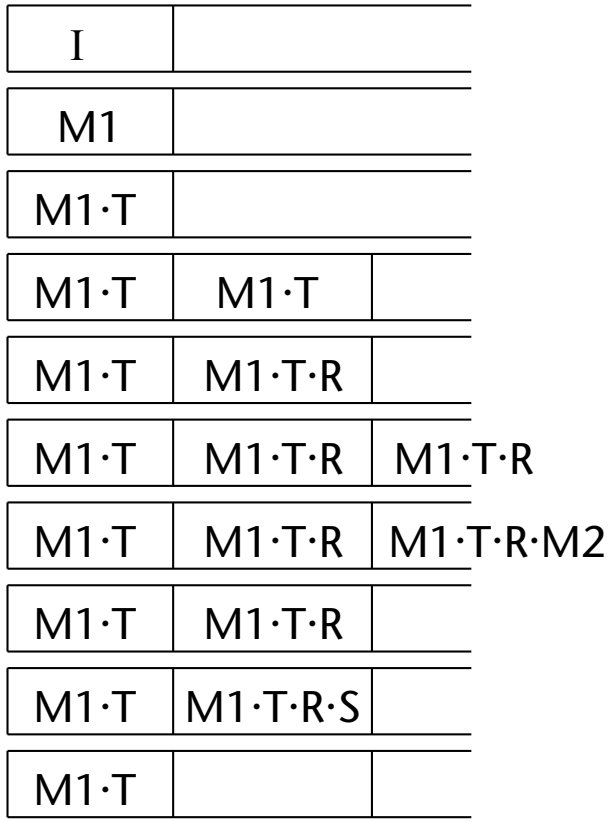
M1·T·R·M2

M1·T·R

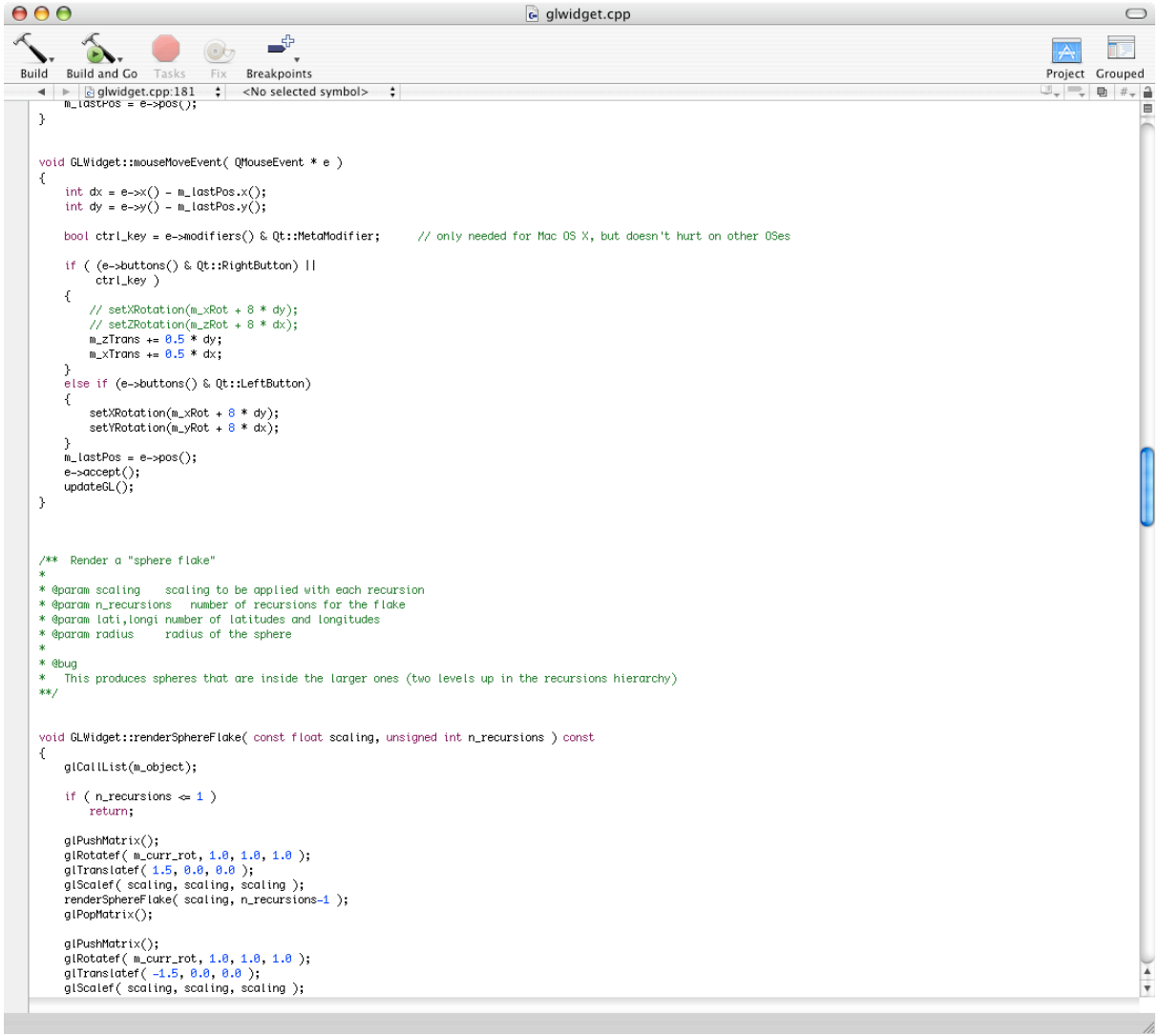
M1·T·R·S

M1·T·R

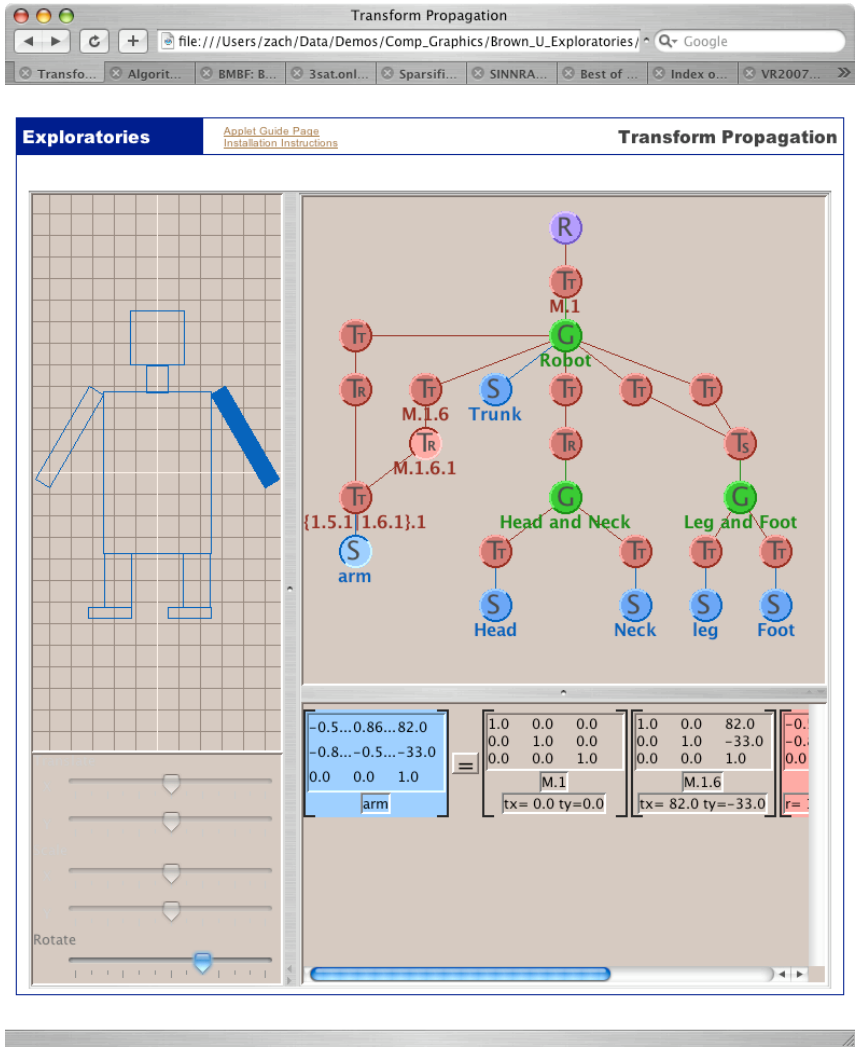
Zustand des Matrix-Stacks:



Beispiel im Code



Demo zum Szenengraph



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