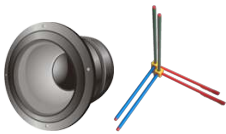
Computer-Graphik I
Transformationen

G. Zachmann
University of Bremen, Germany
cgvr.informatik.uni-bremen.de



Motivation

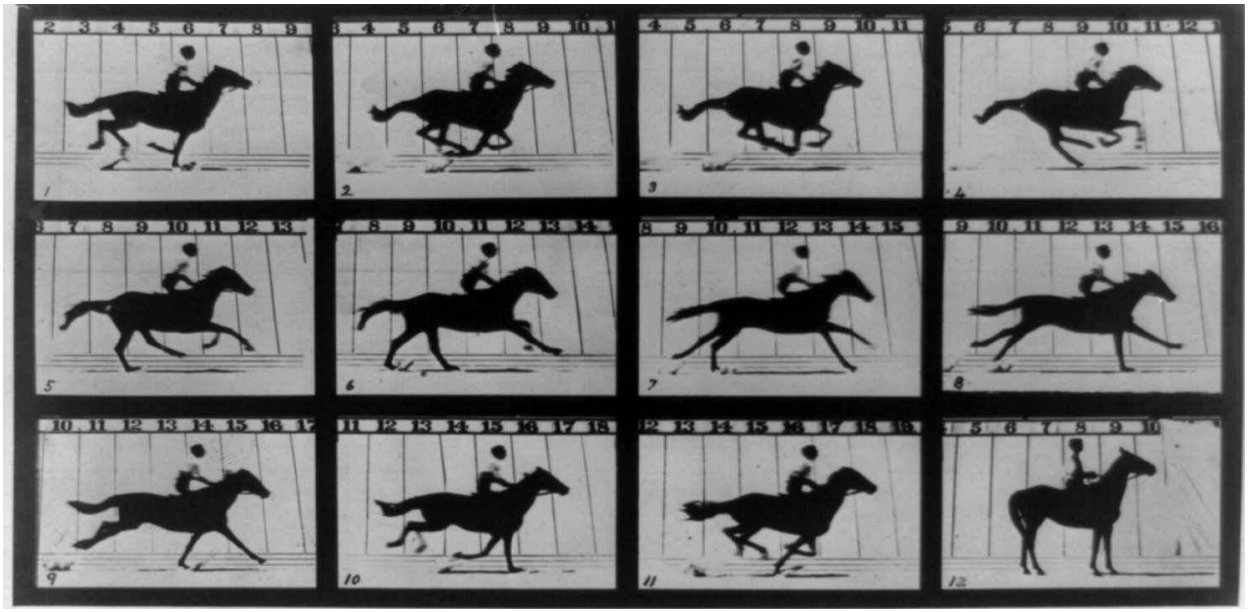
- Transformationen werden benötigt, um ...
- Objekte, Lichtquellen und Kamera zu positionieren
- Alle Berechnungen zur Beleuchtung im selben Koordinatensystem durchzuführen
- Objekte auf den 2D-Bildschirm zu projizieren
- Die Szene zu animieren

Das Prinzip der Animation

- In einer interaktiven Computergraphik-Applikation: setze für jedes animierte Objekt in jedem neuen Frame eine neue Transformation (z.B. Rotation), die sich nur wenig von der vorherigen unterscheidet

```
loop forever:                                                    // main loop
    ask system for current time
    for every animated obj:
       calc new position & orientation based on current time
       store this as translation & rotation with obj
   set new position & orientation for camera
   render scene
```

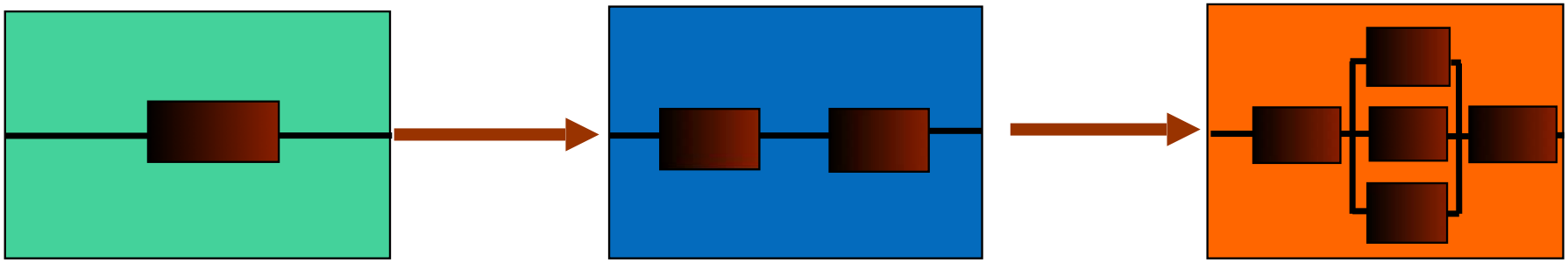
Edward Muybridge, 1878



Film roll

Marcel Duchamp, 1912

Verortung in der Graphik-Pipeline (stark vereinfacht)



Anwendung

Geometrie-Stufe

Raster-Stufe

Im folgenden diese Tasks

Alle Berechnungen, die 1x pro Polygon oder pro Vertex durchgeführt werden.
Arbeitet im 3D.
Z.B.:

- Modell- und Viewing-Transformation
- Projektion
- Beleuchtung
- Clipping

Alle Berechnungen, die pro Fragment durchgeführt werden.
Arbeitet im 2D.
Kennen wir (teilweise) schon
Z.B.:

- Scan Conversion
- Z-Test

Koordinatensysteme in der Pipeline

Model-to-World Transformations

Illumination

Viewing Transformation (Perspective / Orthographic)

Clipping

Projection (to Screen Space)

Scan Conversion (Rasterization)

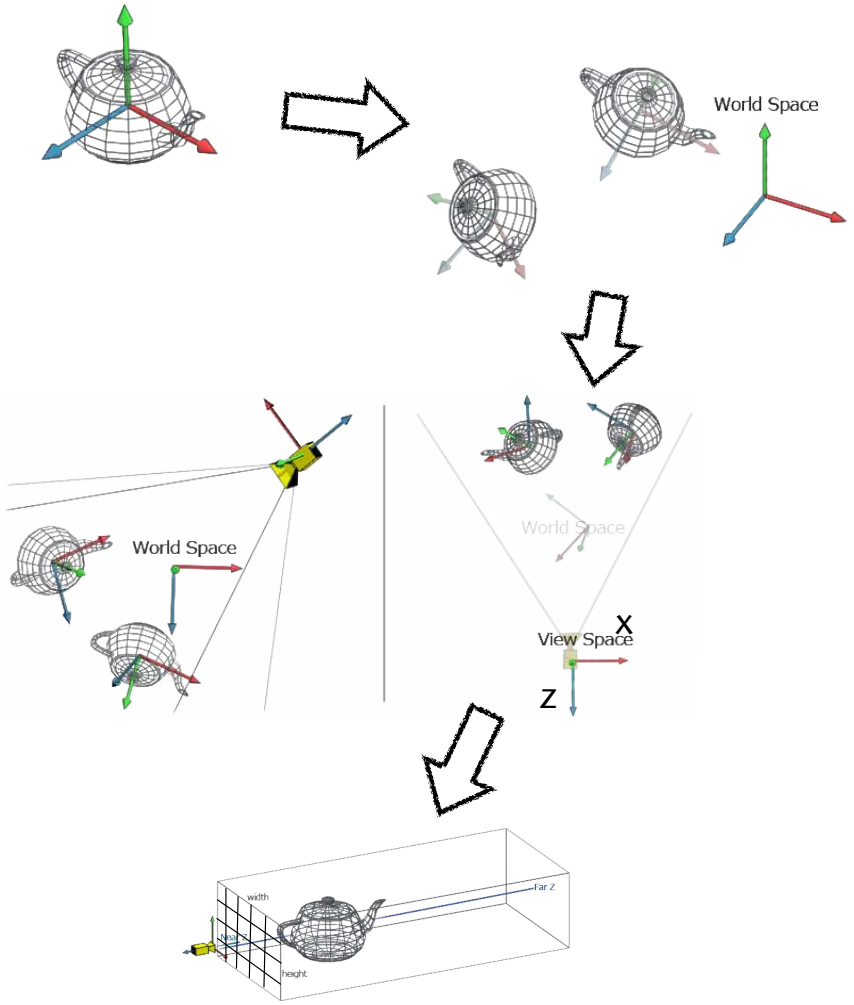
Fragment Processing Fragment Tests

Object Space (a.k.a. local space)

World Space
Für alle Objekte gleich

Eye Space / Camera Space

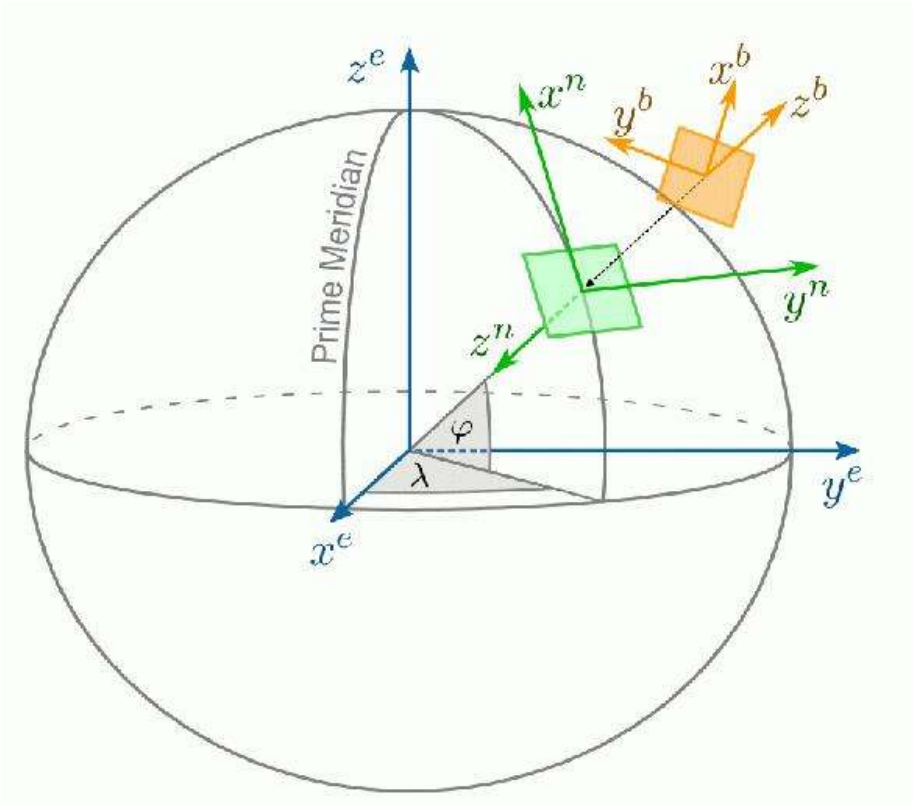
Screen Space
adressiert die Hardware

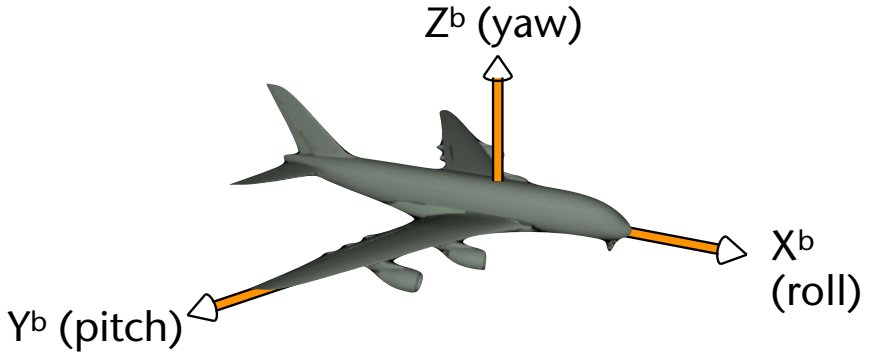


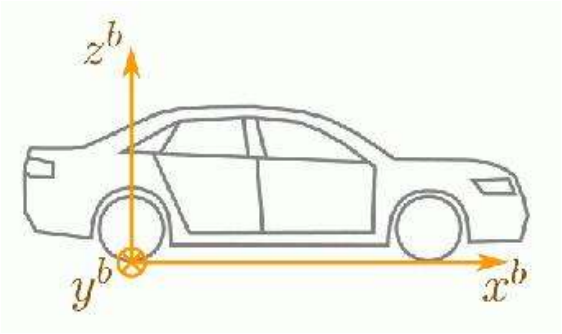
Exkurs: Koordinatensysteme in der Luftfahrt/Raumfahrt

FYI

- Body Frame, Navigation Frame, Earth-Fixed Frame

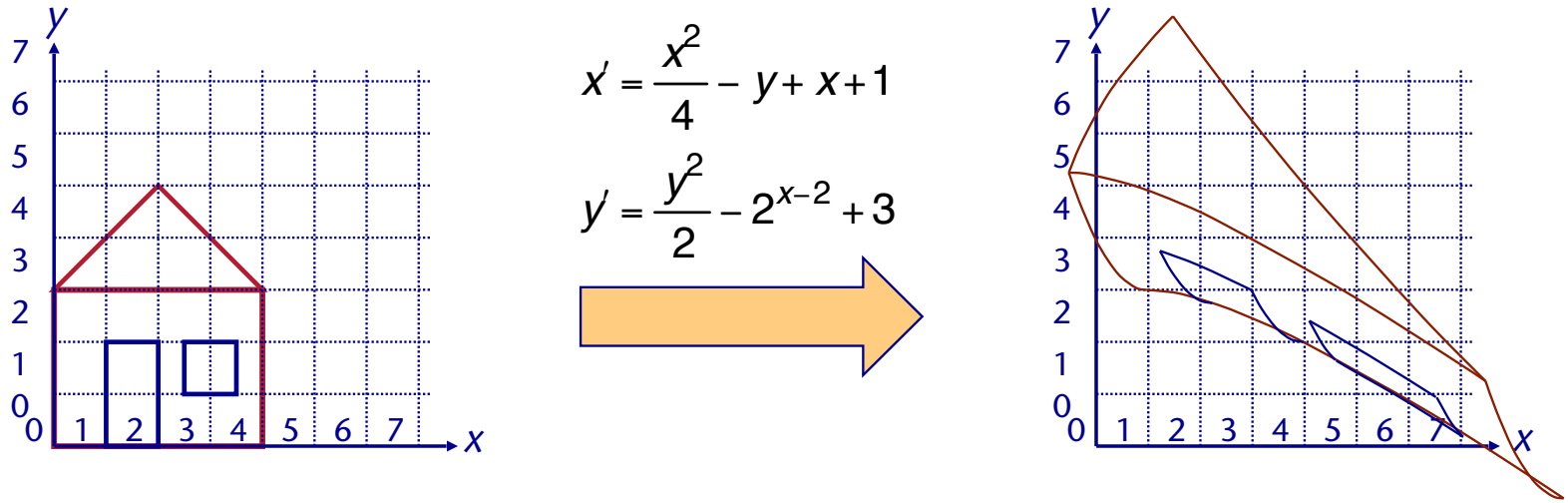






Allgemeine Transformationen

- Allgemeine Transformation ist evtl. nicht linear → Geraden werden i.A. nicht wieder auf Geraden abgebildet
- Beispiel:



- Folge: jeder Punkt (auf einer Linie, in einem Polygon, ...) muss transformiert werden ⟶ nicht effizient, nicht interessant für Echtzeit-Graphik

Vorteile der linearen Abbildungen

- Lineare Transformationen (z.B. Rotation, Skalierung und Scherung) heißen "linear", weil die Eigenschaft der Linearität der Abbildung gilt:

$$X' = A \cdot (\alpha X + \beta Y) = \alpha A \cdot X + \beta A \cdot Y$$

- Folge aus Linearität $\longrightarrow$ Geraden werden auf Geraden abgebildet
- Lineare Abbildungen kann man durch Matrizen darstellen
- Merke die Konvention:

"Matrix mal Vektor"

Die elementaren Transformationen im 3D

1. Rotation
2. Translation
3. Skalierung
4. Spiegelung (kommt sehr selten vor)
5. Scherung (kommt in der Praxis fast nie vor)

Elementare Rotation

- Rotation um x-, y-, z-Achse um Winkel ϕ

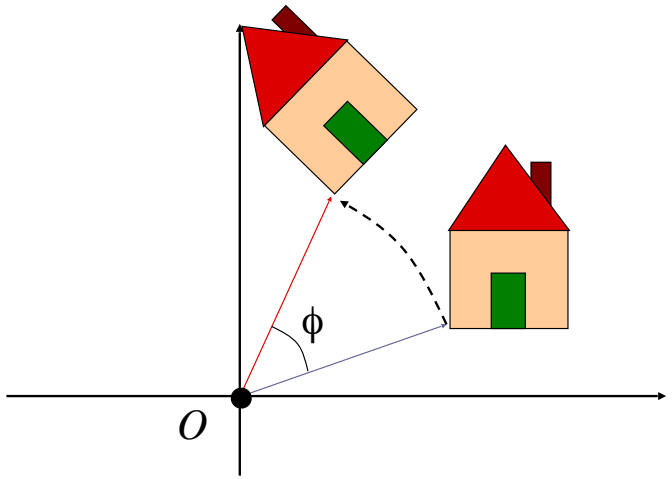
$$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & ?\cos\phi & ?\sin\phi \\ 0 & ?\sin\phi & ?\cos\phi \end{pmatrix}$$

X-Koord. bleibt unverändert

Vorzeichentest: ϕ=90 ⟶
y geht nach z, z geht nach -y.

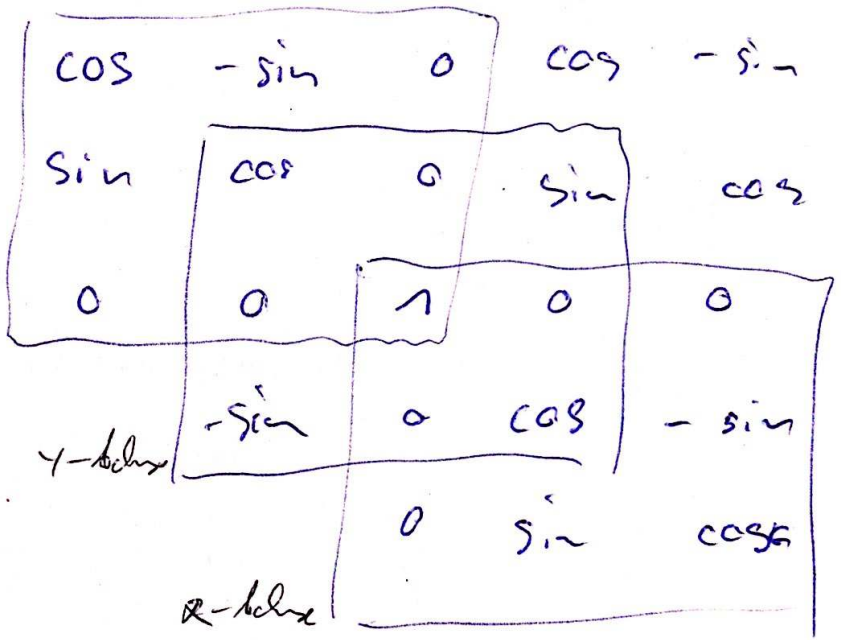
$$R_y(\phi) = \begin{pmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{pmatrix}$$

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



Eselsbrücken zum Merken der elementaren Rotationsmatrizen

- Man merkt sich, dass es im Wesentlichen immer ein "cos/sin-Kästchen" ist, und muss sich dann nur noch überlegen, welche Stelle eine 1 haben muss, damit die Koordinaten der gewünschten Achse unverändert bleiben
- Man merkt sich folgendes Schema [Laura Spillner, 2015]:



Skalierung

- Kann zum Vergrößern oder Verkleinern verwendet werden:

$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix}$$

- s_x, s_y, s_z beschreiben Längenänderung in x-, y-, z-Richtung
  → nicht-uniforme (anisotrope) Skalierung
- Uniforme (isotrope) Skalierung: $s_x = s_y = s_z$
- Inverse:

$$S^{-1}(s_x, s_y, s_z) = S\left(\frac{1}{s_x}, \frac{1}{s_y}, \frac{1}{s_z}\right)$$

- Ein alternative Skalierungs-Matrix:

$$S(s,s,s) = \begin{pmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cong \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{s} \end{pmatrix}$$

- Aber besser die "normale" Skalierungsmatrix verwenden

Scherung (Shearing)

- Verschiebt z.B. die *x*-Koordinate abhängig von der Entfernung zur Ebene *z*=0 (d.h., der *xy*-Ebene), also abhängig von der z-Koord.
- Zum Beispiel: $H_{XZ}(s)$ schert den *x*-Wert gemäß dem *z*-Wert

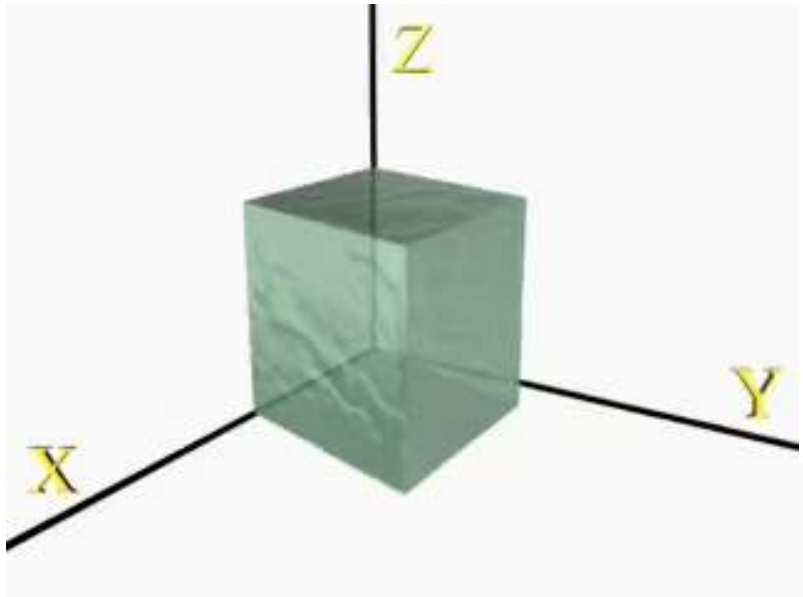
$$H_{xz}(s)\cdot\mathbf{p} = \begin{pmatrix} 1 & 0 & s \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} p_x + sp_z \\ p_y \\ p_z \end{pmatrix}$$

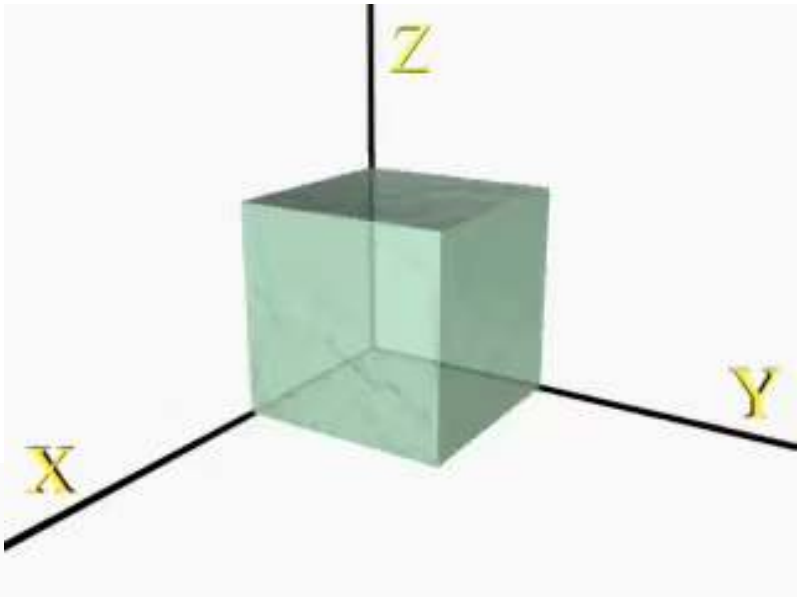
- Insgesamt: 6 mögliche Scherungen
- Inverse:

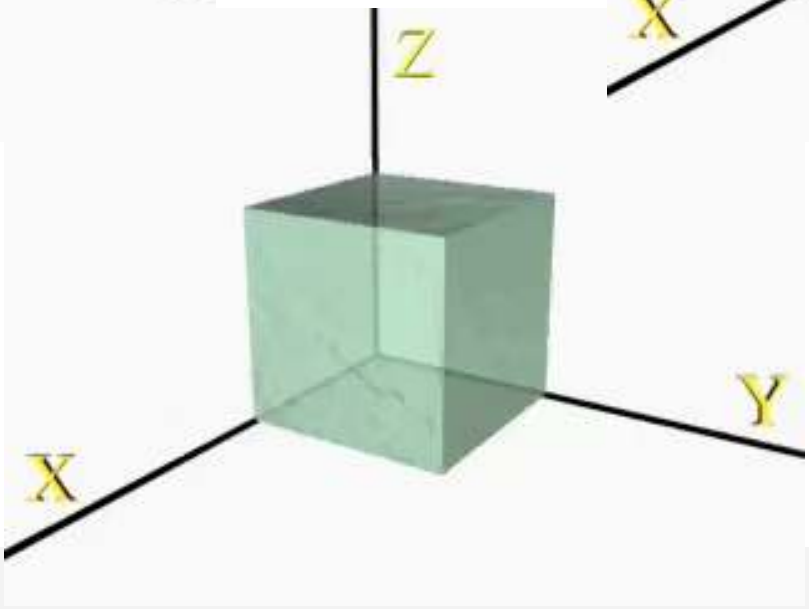
$$H_{xz}^{-1}(s) = H_{xz}(-s)$$

- Bemerkung: Determinante = 1 → Volumen bleibt erhalten
  - Aber Winkel werden hier nicht erhalten!

Visualisierung mit Animation des Parameters *s*







Shearing for Sculpture

Museum of Fine Arts, Boston

Spiegelung

- Spiegelung entlang der x-Achse, m.a.W., Spiegelung bzgl. der yz-Ebene:

$$M_x = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

  - Analog die anderen beiden Spiegelungen
- Achtung: $\det(M_x) < 0$!
  - Bei allen anderen Transformationen T bisher war $\det(T) > 0$
- Spiegelungen sind in der CG eigtl. immer ausgeschlossen
  - U.a., weil der Umlaufsinn der Polygone umgedreht wird, und weil ein positives Volumen zu einem negativen Volumen wird!

Denksport-Aufgabe

- Wie kommt es, dass ein Spiegel links/rechts vertauscht, aber nicht oben/unten?
- Antwort:
  - In Wahrheit vertauscht er auch links/rechts nicht, sondern vorne/hinten!
    - Die Nase im Spiegel zeigt nach Süden,
      wenn unsere reale Nase nach Norden zeigt,
      aber wenn wir mit einer Hand nach Westen zeigen,
      zeigt auch die Hand im Spiegel nach Westen!
  - Der Fehler ist, dass wir uns *vorstellen*, dass eine Person hinter dem Spiegel steht
  - Alternative Betrachtung: spannen wir mit der rechten Hand ein *rechtshändiges* Koordinatensystem auf, so spannt die gegenüberliegende Hand im Spiegel ein *linkshändiges* Koordinatensystem auf!

Translation (a.k.a. Verschiebung)

- Definition: $X' = X + \mathbf{t}$

- Problem: affine Transformationen können **nicht** in Form einer 3x3-Matrix dargestellt werden

Affine Abbildungen

- Pragmatische Definition:
  Affine Abbildungen wirken auf Punkte, und bilden Geraden wieder auf Geraden, Ebenen wieder auf Ebenen ab.
  Alle affinen Abbildungen sind von der Form

$$\mathbf{p}' = M\mathbf{p} + \mathbf{t}$$

  wobei $\mathbf{p}$ der Ortsvektor zum Punkt P ist.

- Problem: affine Transformationen können **nicht** in Form einer 3x3-Matrix dargestellt werden (wegen der Translation)

Lösung: Homogene Koordinaten im 3D

- "Trick" (Homogenisierung):
  - Bette die Räume der 3D-Punkte und 3D-Vektoren in einen gemeinsamen 4D-Raum ein
  - Homogener Punkt $\mathbf{p} = (p_x, p_y, p_z, 1)$
  - Homogener Vektor $\mathbf{v} = (v_x, v_y, v_z, 0)$
- Vorteil: wir können mit beiden weiterrechnen, als wären es Vektoren

Addition von Punkten und Vektoren in homogenen Koordinaten

- Punkt + Vektor = Punkt

$$\begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} + \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix} = \begin{pmatrix} p_x + v_x \\ p_y + v_y \\ p_z + v_z \\ 1 \end{pmatrix}$$

- Vektor + Vektor = Vektor

$$\begin{pmatrix} u_x \\ u_y \\ u_z \\ 0 \end{pmatrix} + \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix} = \begin{pmatrix} u_x + v_x \\ u_y + v_y \\ u_z + v_z \\ 0 \end{pmatrix}$$

- Punkt – Punkt = Vektor

$$\begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} - \begin{pmatrix} q_x \\ q_y \\ q_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x - q_x \\ p_y - q_y \\ p_z - q_z \\ 0 \end{pmatrix}$$

Lineare Abbildungen in homogenen Koordinaten

- 3x3-Form:

$$M \cdot \mathbf{v} = \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

- Homogene Form:

$$M_{4x4} \cdot \mathbf{v}' = \begin{pmatrix} m_{00} & m_{01} & m_{02} & 0 \\ m_{10} & m_{11} & m_{12} & 0 \\ m_{20} & m_{21} & m_{22} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix}$$

Translation

- Translation eines Punktes:

$$T_t \cdot P = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{pmatrix}$$

- "Translation" eines Vektors:

$$T_t \cdot \mathbf{v} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix}$$

- Inverse: $(T_t)^{-1} = T_{-t}$

Allgemeine affine Abbildungen im 3D

- 3x3-Form:

$$M \cdot \mathbf{p} + t = \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$$

- Homogene Form:

$$M_{4x4} \cdot \mathbf{p}_4 = \begin{pmatrix} m_{00} & m_{01} & m_{02} & t_x \\ m_{10} & m_{11} & m_{12} & t_y \\ m_{20} & m_{21} & m_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

➢ In homogenen Koordinaten lassen sich sogar alle affinen Abbildungen als einfache Matrix-Vektor-Multiplikation darstellen!

Recap: die elem. linearen Transformationen in homogenen Koordinaten

- Rotation:

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

- Skalierung:

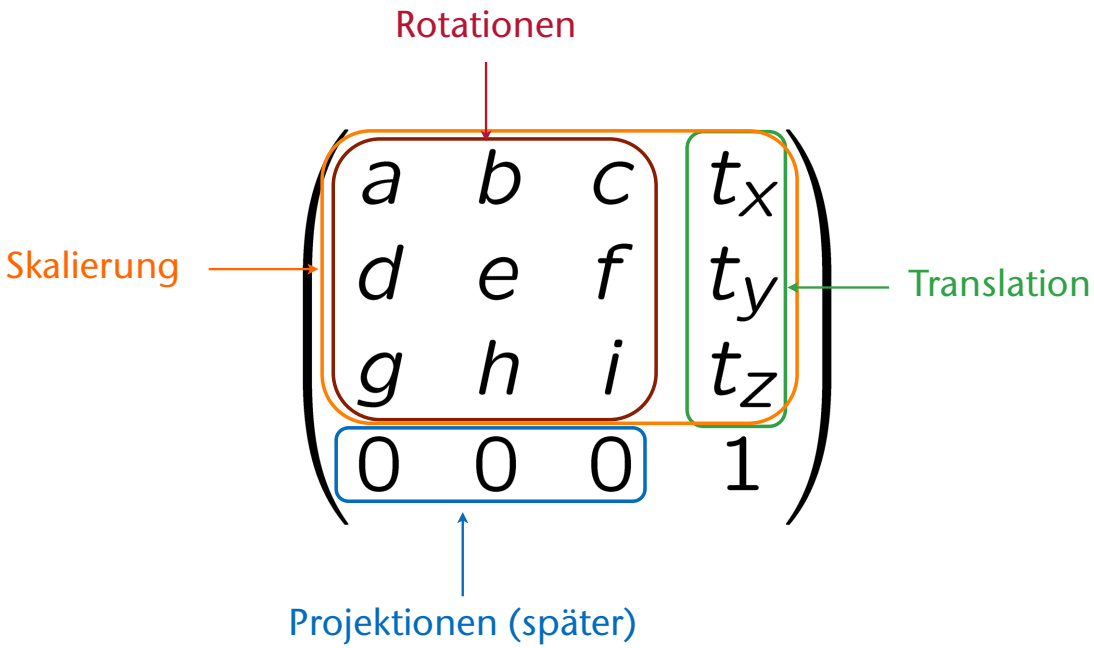
$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Scherung:

$$H_{xz}(s)\cdot\mathbf{p} = \begin{pmatrix} 1 & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + sp_z \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

Zur Anatomie einer Matrix

- Allgemeiner Aufbau (vereinfacht!):

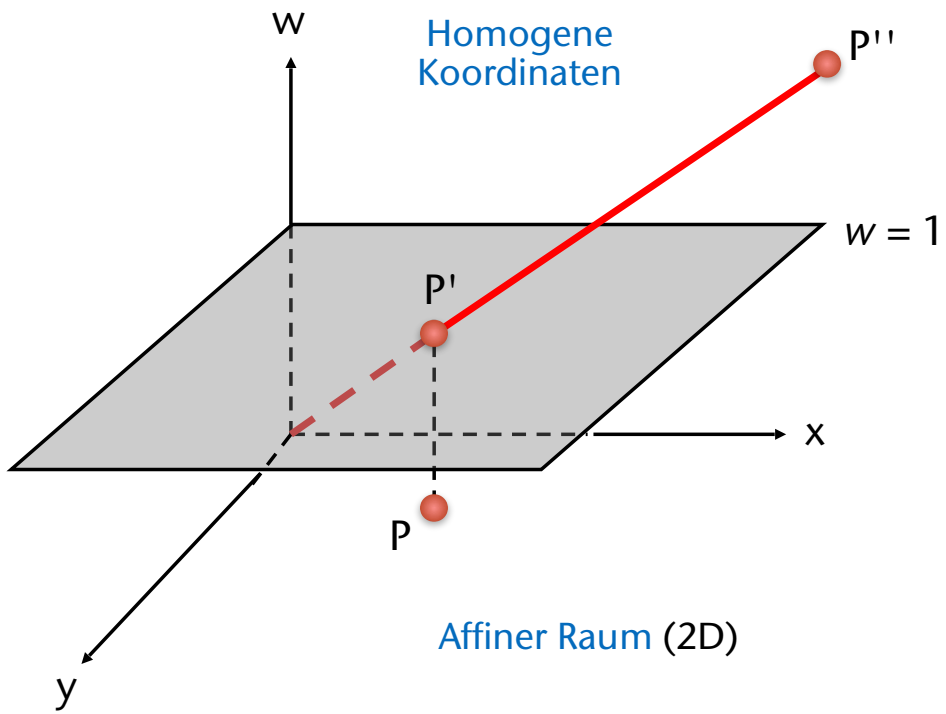


Mathematischer Hintergrund zu homogenen Koordinaten

- Allgemeine Form der Einbettung des R^3 in den R^4:
  - Identifiziere Vektoren $(v_x, v_y, v_z) \in \mathbb{R}^3$ mit Vektoren in $(v_x, v_y, v_z, 0) \in \mathbb{R}^4$
  - Identifiziere Punkte $(p_x, p_y, p_z) \in \mathbb{R}^3$ mit der Geraden $w \cdot (p_x, p_y, p_z, 1) \in \mathbb{R}^4$
    - Der Vektor $(p_x, p_y, p_z, 1) \in \mathbb{R}^4$ ist nur ein Repräsentant dieser Geraden!
- Vorteil: wir können mit Punkten und Vektoren aus dem 3D im $\mathbb{R}^4$ weiterrechnen, als wären es Vektoren!
  - Achtung: dabei ist w≠0 und w≠1 erlaubt und kann vorkommen!
  - Nicht erlaubt: Wechsel von w=0 zu w≠0 und umgekehrt!
- Rückprojektion: gegeben $\hat{\mathbf{v}} = (x, y, z, w)$
  - Falls $w = 0$: $\hat{\mathbf{v}} \longmapsto \mathbf{v} = (x, y, z)$ = Vektor
  - Falls $w \neq 0$: $\hat{\mathbf{v}} \longmapsto P = (\frac{x}{w}, \frac{y}{w}, \frac{z}{w})$ = Punkt

Veranschaulichung durch Analogie im 2D

- Erweitere Punkt P = (*x*,*y*) zu P' = (*x*,*y*,1)
- Assoziiere Linie *w*·(*x*,*y*,1) = (*wx*, *wy*, *w*) mit P'
- M.a.W.: irgend ein 3D-Vektor (*x*, *y*, *w*) beschreibt ...
  - ... den 2D-Punkt (*x*/*w*, *y*/*w*) für *w* ≠ 0
  - ... den 2D-Vektor (*x*, *y*) für *w* = 0



Caveat bei Subtraktion homogener Punkte im 4D

- Achtung: im Allgemeinen darf man homogene *Punkte* nicht einfach so im 4D voneinander subtrahieren!
- Subtraktion in 4D *vor* der Rückprojektion ergäbe:

$$\begin{pmatrix} p_x \\ p_y \\ p_z \\ p_w \end{pmatrix} - \begin{pmatrix} q_x \\ q_y \\ q_z \\ q_w \end{pmatrix} = \begin{pmatrix} p_x - q_x \\ p_y - q_y \\ p_z - q_z \\ p_w - q_w \end{pmatrix} \widehat{=} \frac{\mathbf{p}_{xyz} - \mathbf{q}_{xyz}}{p_w - q_w}$$

Kurzschreibweise für den Vektor (q_x, q_y, q_z)

- Subtraktion in 3D *nach* der Rückprojektion ergibt:

$$\frac{\mathbf{p}_{xyz}}{p_w} - \frac{\mathbf{q}_{xyz}}{q_w} = \frac{q_w \cdot \mathbf{p}_{xyz} - p_w \cdot \mathbf{q}_{xyz}}{p_w \cdot q_w}$$

- Wann darf man "einfach so" im 4D subtrahieren? $\longrightarrow$ wenn die w's gleich sind!

Column-/Row-Major Order (Matrizen in OpenGL)

- Achtung: Matrizen werden **spaltenweise** im Speicher abgelegt, *nicht* — wie in C üblich — zeilenweise!
  - Das nennt sich "*column-major order*" (der Standard, auch in C, ist *row-major order*)
- Eine mathematische(!) Matrix in column-major order in C/C++ und als lineares Array:

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

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

⇕

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

- Matrizen als 2D-Arrays in C/C++:

```
GLfloat matrix[4][4] =
{
   1, 0, 0, tx,
   0, 1, 0, ty,
   0, 0, 1, tz,
   0, 0, 0, 1
};
```

```
{ 1, 0, 0, tx, 0, 1, 0, ty, 0, 0, 1, tz, 0, 0, 0, 1 }
```

```
GLfloat matrix[4][4] =
{
   1, 0, 0, 0,
   0, 1, 0, 0,
   0, 0, 1, 0,
   tx, ty, tz, 1
};
```

```
{ 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, tx, ty, tz, 1 }
```

- Ein 2D-Array (Matrix) in C/C++ als row major order *interpretiert*:

```
GLfloat matrix[4][4] =
{
   1, 0, 0, tx,
   0, 1, 0, ty,
   0, 0, 1, tz,
   0, 0, 0, 1
};
```

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

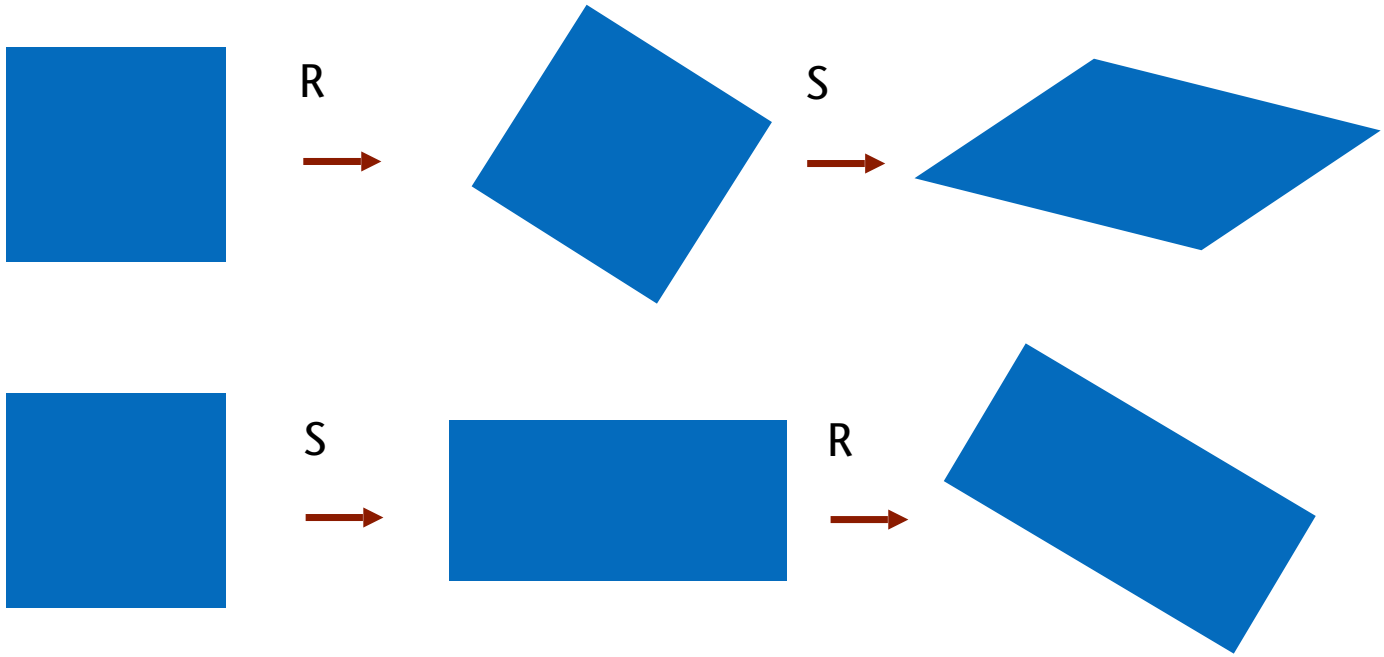
- Das selbe 2D-Array als column major order *interpretiert*:

```
GLfloat matrix[4][4] =
{
   1, 0, 0, tx,
   0, 1, 0, ty,
   0, 0, 1, tz,
   0, 0, 0, 1
};
```

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

Verknüpfung / *Concatenation*

- Beispiel:



- Stimmt mit der Mathematik überein: Multiplikation von Matrizen ist nicht kommutativ ⟶ Reihenfolge der Transformation spielt eine Rolle!

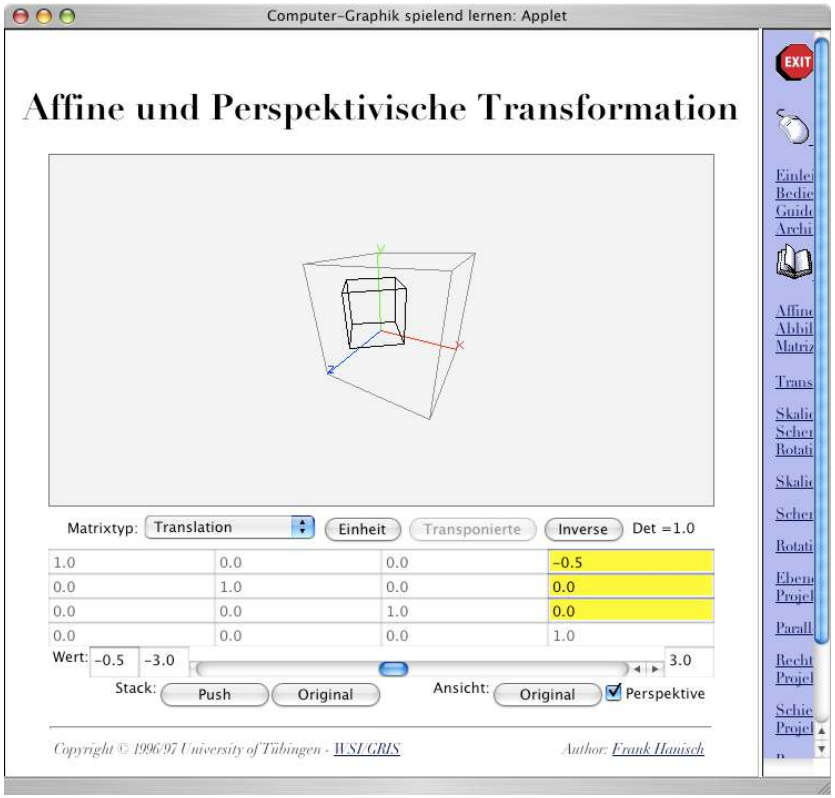
- Reihenfolge in einer Matrixkette:

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

Reihenfolge der Ausführung

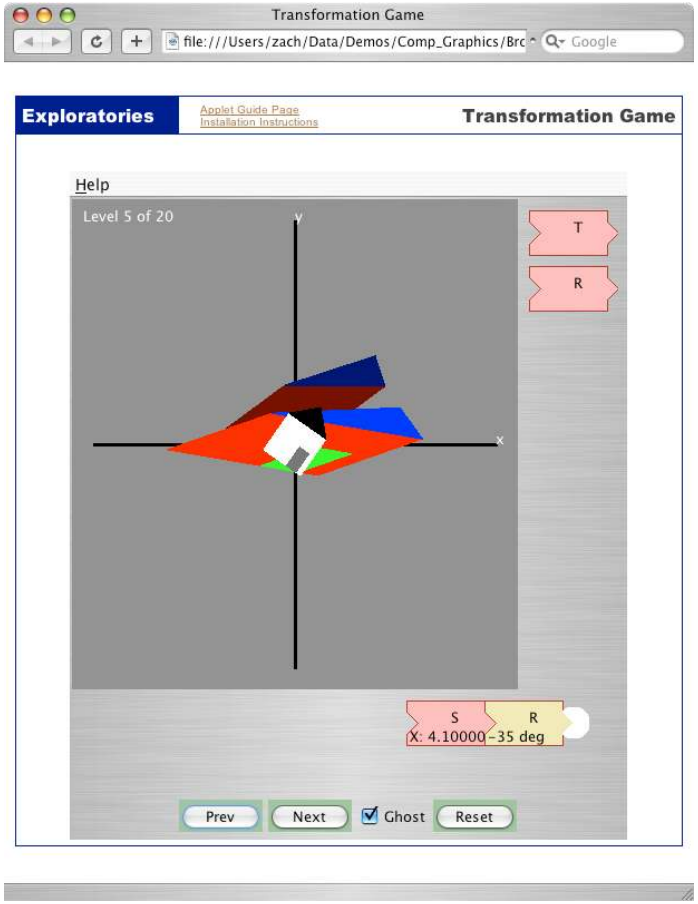
- Nützlich zur Steigerung der Effizienz

Demo zur Konkatenation von Transformationen



(Urspr. Quelle: http://www.gris.uni-tuebingen.de/gris/GDV/java/doc/html/etc/AppletIndex.html)

Demo

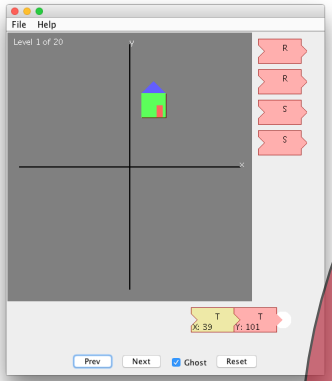


http://graphics.cs.brown.edu/research/exploratory/freeSoftware
⟶ Complete Catalog ⟶ Transformation Game

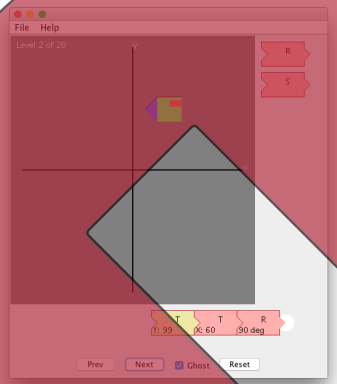
http://cgvr.cs.uni-bremen.de
Teaching ⟶ CG1
⟶ "Transformation Game" suchen, ZIP-File entpacken, dann

```
cd freeSoftware/repository/edu/brown/cs/exploratories/-
   applets/transformationGame
java -jar transformation_game.jar
```

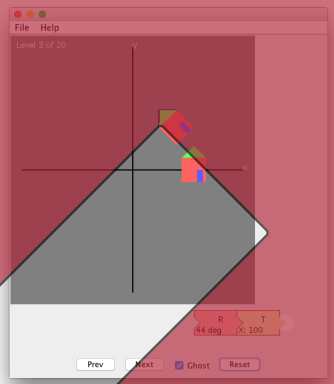
In-Class Aufgaben



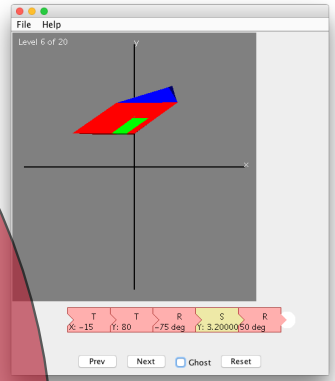
Level 1



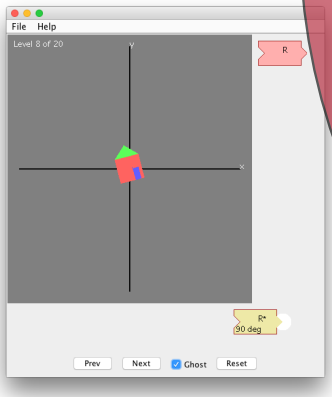
Level 2



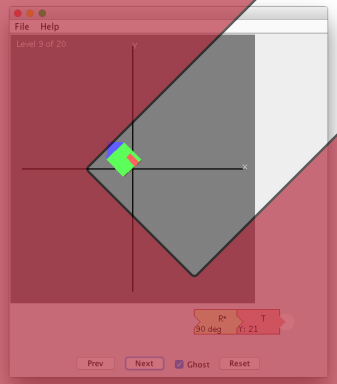
Level 3



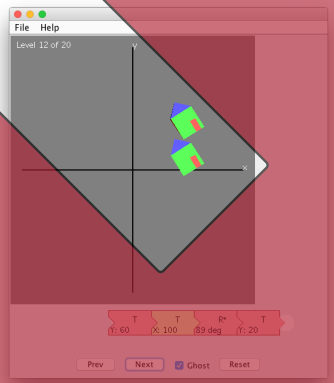
Level 6



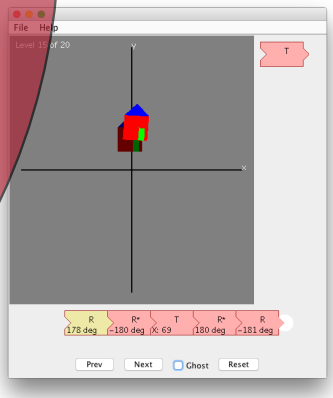
Level 8



Level 9



Level 12



Level 15

(es reicht, aufrecht stehendes Häuschen auf Halbkreis rotieren zu lassen

Starre Transformationen (*Rigid-Body Transform*)

- Beliebige Folge von Translationen und Rotationen
  - Aka. Euklidische Transformation
- 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:

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

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

Zur Anatomie der Matrix

- Betrachte "erst Rotation, dann Translation":

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

$$M = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{21} & R_{22} & R_{23} & 0 \\ R_{31} & R_{32} & R_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \left(\begin{array}{ccc|c} R_{11} & R_{12} & R_{13} & t_x \\ R_{21} & R_{22} & R_{23} & t_y \\ R_{31} & R_{32} & R_{33} & t_z \\ \hline 0 & 0 & 0 & 1 \end{array}\right) = \begin{pmatrix} R & \mathbf{t} \\ 0 & 1 \end{pmatrix}$$

- Betrachte "erst Translation, dann Rotation":

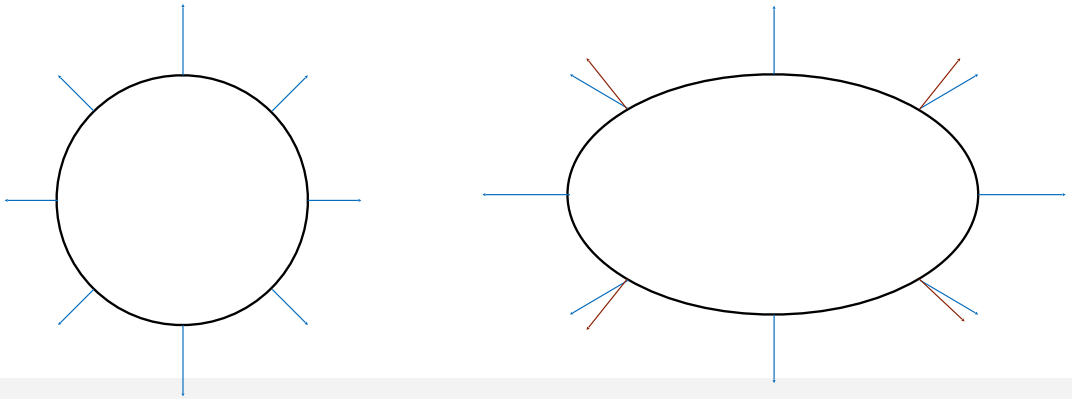
$$P' = (RT)P = MP \cong R(P + \mathbf{t}) = R_{3x3}P + R_{3x3}\mathbf{t}$$

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

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

Transformation von Normalen

- Behauptung: wenn ein Objekt um M transformiert wird, dann müssen die Normalen der Oberfläche um $N = (M^T)^{-1}$ 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 $N = (M^T)^{-1} \neq M$!
  - Beispiel:



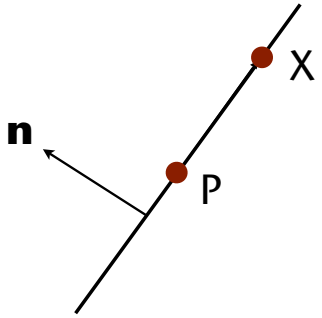
Beweis

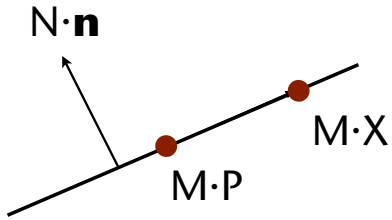
- Wir wissen: $(X - P)^{\mathsf{T}}\mathbf{n} = 0$
- Gesucht ist N, so daß:

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

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

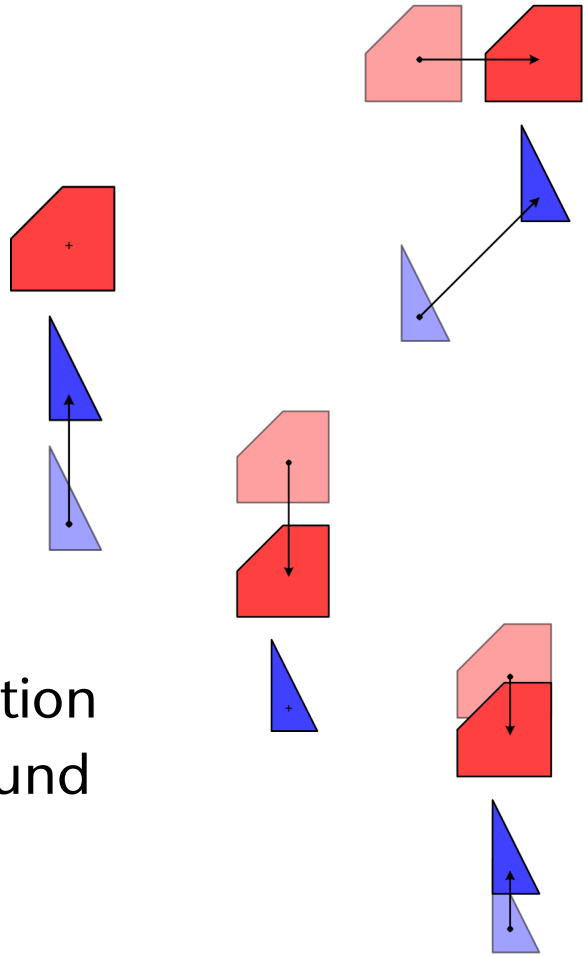
$$(X - P)^{\mathsf{T}} \cdot M^{\mathsf{T}} (M^{\mathsf{T}})^{-1} \cdot \mathbf{n} = (X - P)^{\mathsf{T}} \cdot I \cdot \mathbf{n} = 0$$





Relative Bewegung

- Betrachte Objekte A und B, die sich bzgl. des Weltkoordinatensystems bewegen
- Betrachte die Bewegung von A's Koordinaten-system (reference frame) aus
- Und von B's Koordinatensystem aus
- Vom Intertialsystem aus
  (Schwerpunkt zwischen beiden)
- Fazit: man kann zu jedem Zeitpunkt eine Transformation *M* finden, so dass *M*(A) immer im Ursprung bleibt – und *M*(B) sich relativ dazu bewegt. (Oder umgekehrt)

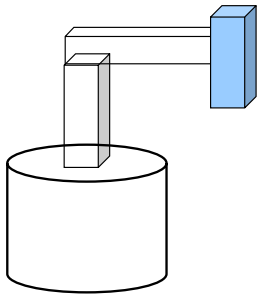


Eine Anwendung

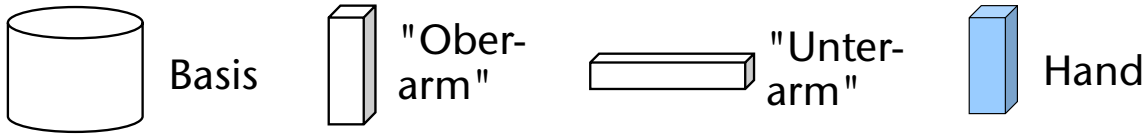
- It is always possible to reduce a collision check between two moving objects to a collision check between a moving object and a stationary object (by reframing)
- Verfahren:
  - Seien $M_A(t)$ und $M_B(t)$ die Model-to-World-Trf.en zur Zeit t für Objekt A bzw. Objekt B
  - Transfomiere A nicht (bleibt in A's Objekt-Koord.-System)
  - Transformiere B mit der Matrix $M_A^{-1}(t) \cdot M_B(t)$

Hierarchische Transformationen

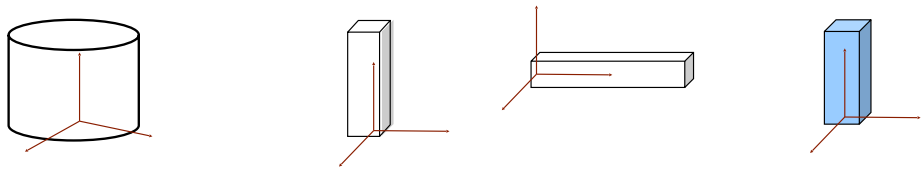
- Eine Konkatenierung von Transformationen kann man auch als eine Folge von (voneinander abhängigen) Koordinatensystemen ansehen



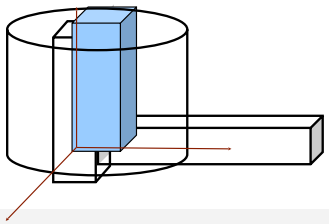
- Beispiel: Roboter
  - Besteht aus diesen Einzelteilen



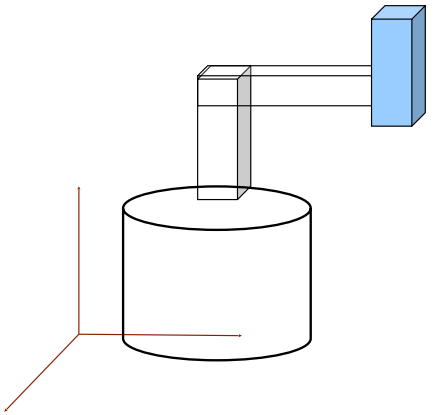
  - Jedes Teil wurde in seinem eigenen Koordinatensystem spezifiziert (als Array von Punkten) → heißt Objektkoordinatensystem



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



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



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

setTranslation( robot_pos_x, robot_pos_y + 10, … )
render upper arm …

setTranslation( robot_pos_x, robot_pos_y + 10 + 5, … )
render lower arm …
. . .
```

Hypothetische Funktion

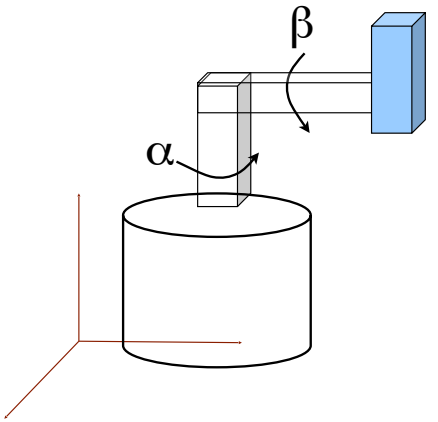
Ann.: Höhe der Basis ist 10

Ann.: Höhe des Oberarms ist 5

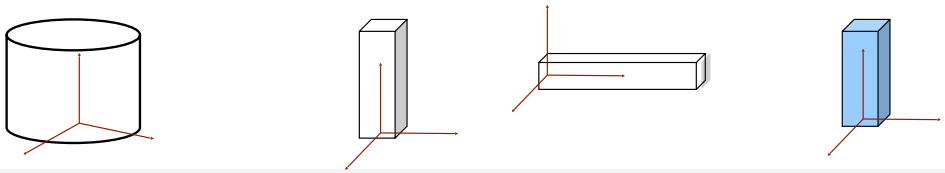
- Natürlich macht man es ungefähr so:

```
clearAllTransforms

translate( robot_pos_x, robot_pos_y , … )
render base …

translate( 0, HEIGHT_BASE, 0 )
rotate( alpha, 0, 1, 0 );
render upper arm …

translate( 0, LEN_UPPER_ARM, 0 )
rotate( beta, 1, 0, 0 );
render lower arm …

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

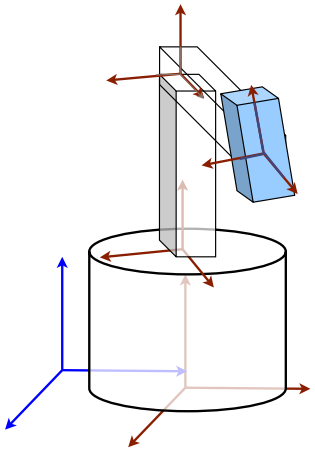
Hypothetische Funktionen, die eine Transformation *RELATIV* zur bis dahin gültigen Transformation setzen



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



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



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

```
clearAllTransforms

translate( robot_pos_x, robot_pos_y , … )
render base …

translate( 0, HEIGHT_BASE, 0 )
rotate( alpha, 0, 1, 0 );
render upper arm …

translate( 0, HEIGHT_UPPER_ARM, 0 )
rotate( beta, 1, 0, 0 )
render lower arm …

translate( X_SIZE_LOWER_ARM, 0, 0 )
render hand …
```

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

- Mathematische Realisierung: Konkatenation der einzelnen (= relativen) Transformationsmatrizen

Reihenfolge der Transformations-Befehle im Programmablauf

Vertex in Welt-Koordinaten ("hinteres" Ende der Pipeline)

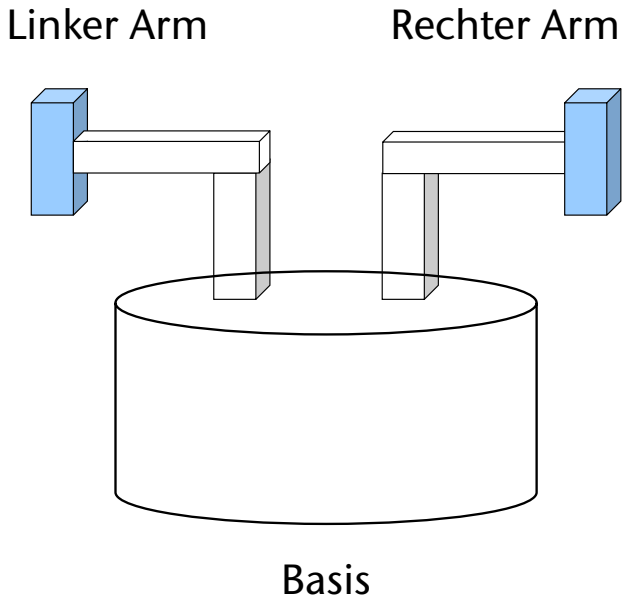
$$p' = M_n \cdot \ldots \cdot M_2 \cdot M_1 \cdot p$$

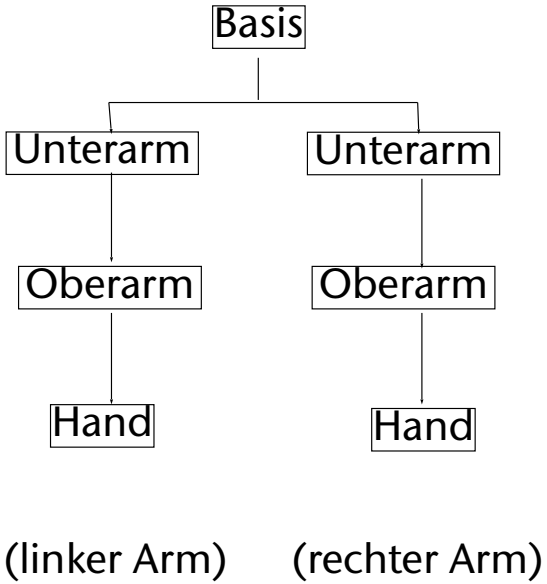
Vertex in Modell-Koordinaten (vorderes Ende der Pipeline)

Reihenfolge der Ausführung

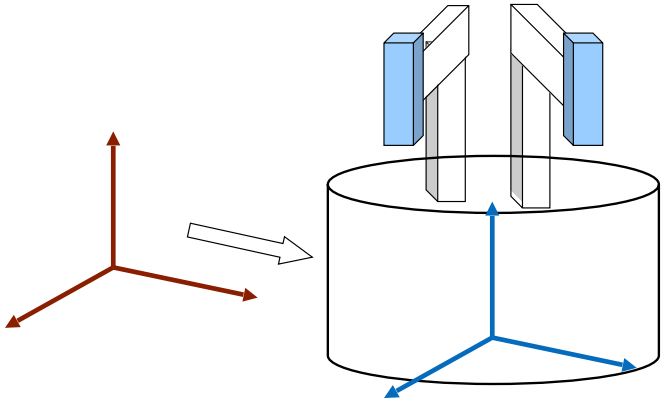
- Bemerkung: da Vertices von rechts an die Matrix (Matrizen) multipliziert werden, *muss* M_n die zuerst im Programmablauf gesetzte Transformation sein
- Definition:
  - Model-to-World-Matrix = $M_n \cdot \ldots \cdot M_2 \cdot M_1$
    = Konkatenation aller aktuell gültigen, relativen Transf.en

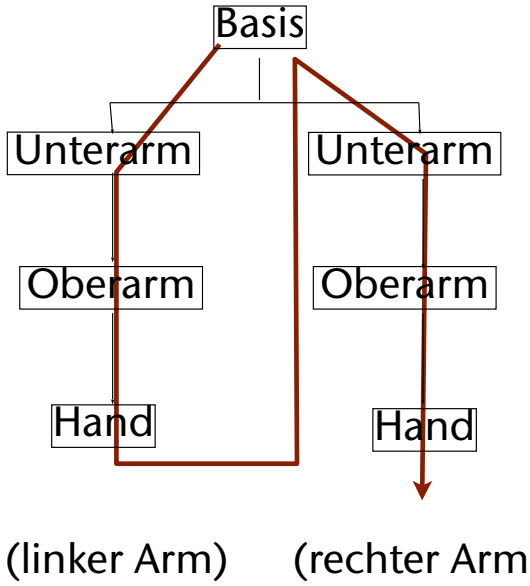
- Ein etwas komplizierteres Beispiel:





- Aufgabe: folgende Konfiguration darstellen
- Natürliche Vorgehensweise ist Depth-First-Traversal durch den Szenengraph





(linker Arm) (rechter Arm)

```
Do transformation(s)

Draw base

Do transformation(s)

Draw left arm

Do transformation(s)

Draw right arm
```

Lösung: ein Matrix-Stack

```
Init ModelToWorld M = M0 = I

Translate(5, 0, 0) → M := M0·T

Draw base

Rotate(75, 0, 1, 0) → M := M0·T·R(a)

Draw left arm

Rotate(-75, 0, 1, 0)

Draw right arm
```

Speichere die aktuelle Model-to-World Matrix an dieser Stelle in einem Zwischenspeicher

Restauriere diese gemerkte Model-To-World an dieser Stelle aus dem Zwischenspeicher ⟶ M

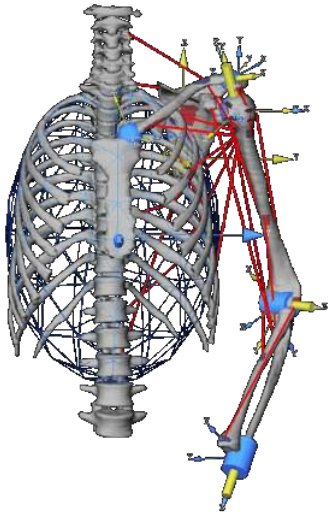
Lösung: ein Matrix-Stack

```
Init ModelToWorld M = M0 = I

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 Model-to-World auf den Stack pushen
Z.B.: matrixStack.push(M);

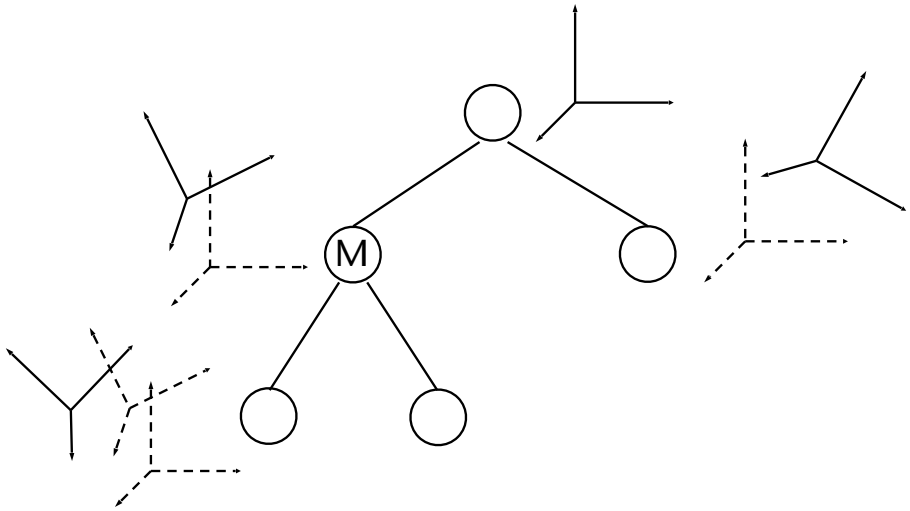
An dieser Stelle die oberste Matrix vom Stack pop-en und in die Model-to-World schreiben
Z.B.: M = matrixStack.pop();

Der Szenengraph

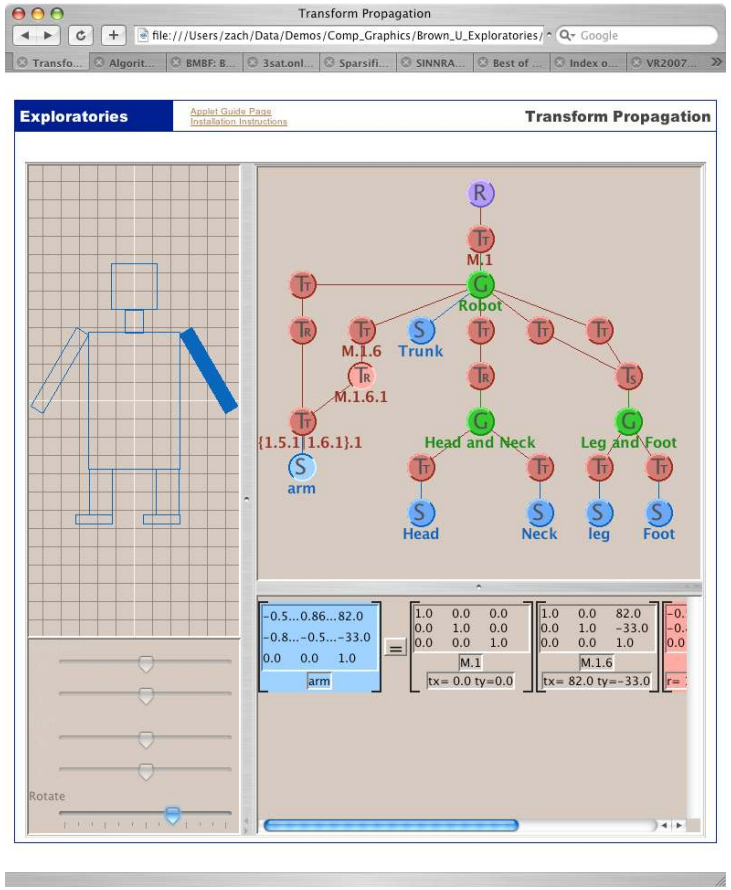


- Durch die relativen Transformationen ergibt sich eine Abhängigkeit der Objekte
- Der so definierte Baum heißt Szenengraph
- Knoten =
  - Transformationsknoten (speichern relative Traf.)
  - Geometrieknoten (i.A. Blätter)
- Aktion am Traf.-Knoten während Scenegraph-Traversal:

```
pushMatrix()
multMatrix( M )
    traverse sub-tree
popMatrix()
```



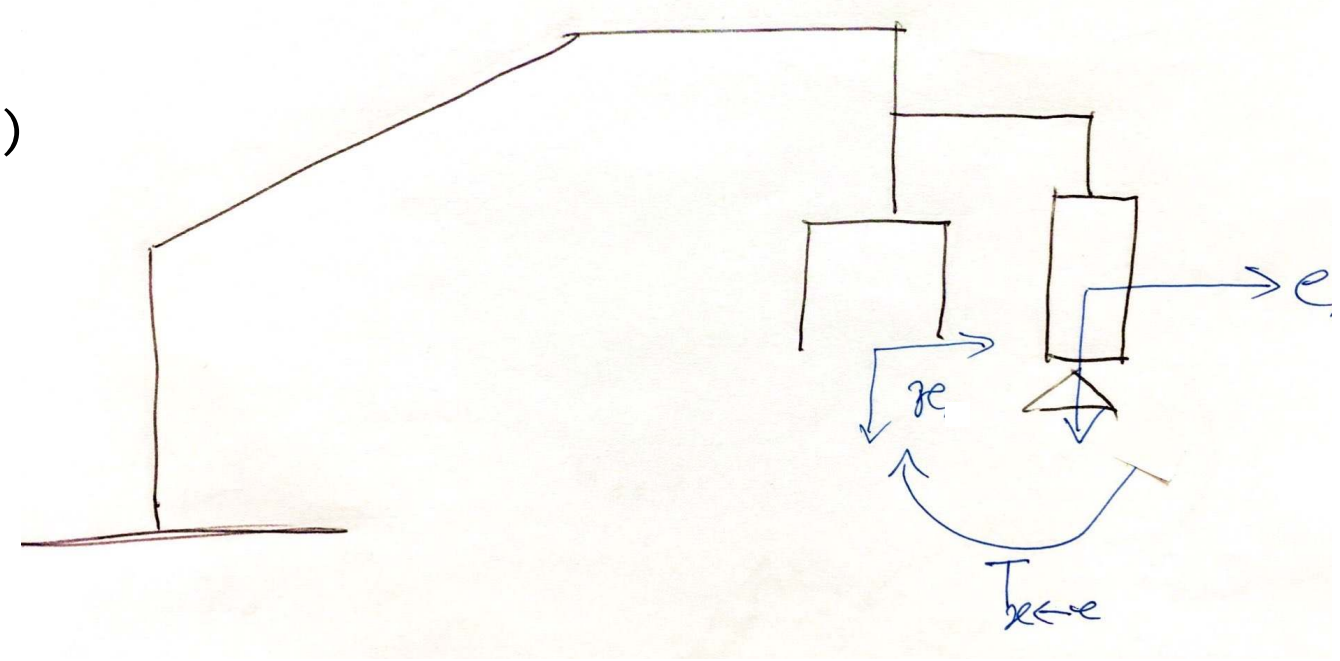
Demo zum Szenengraph



http://graphics.cs.brown.edu/research/exploratory/freeSoftware → Complete Catalog → Transformation Propagation

Beispiel-Anwendung.: Hand-Eye Calibration

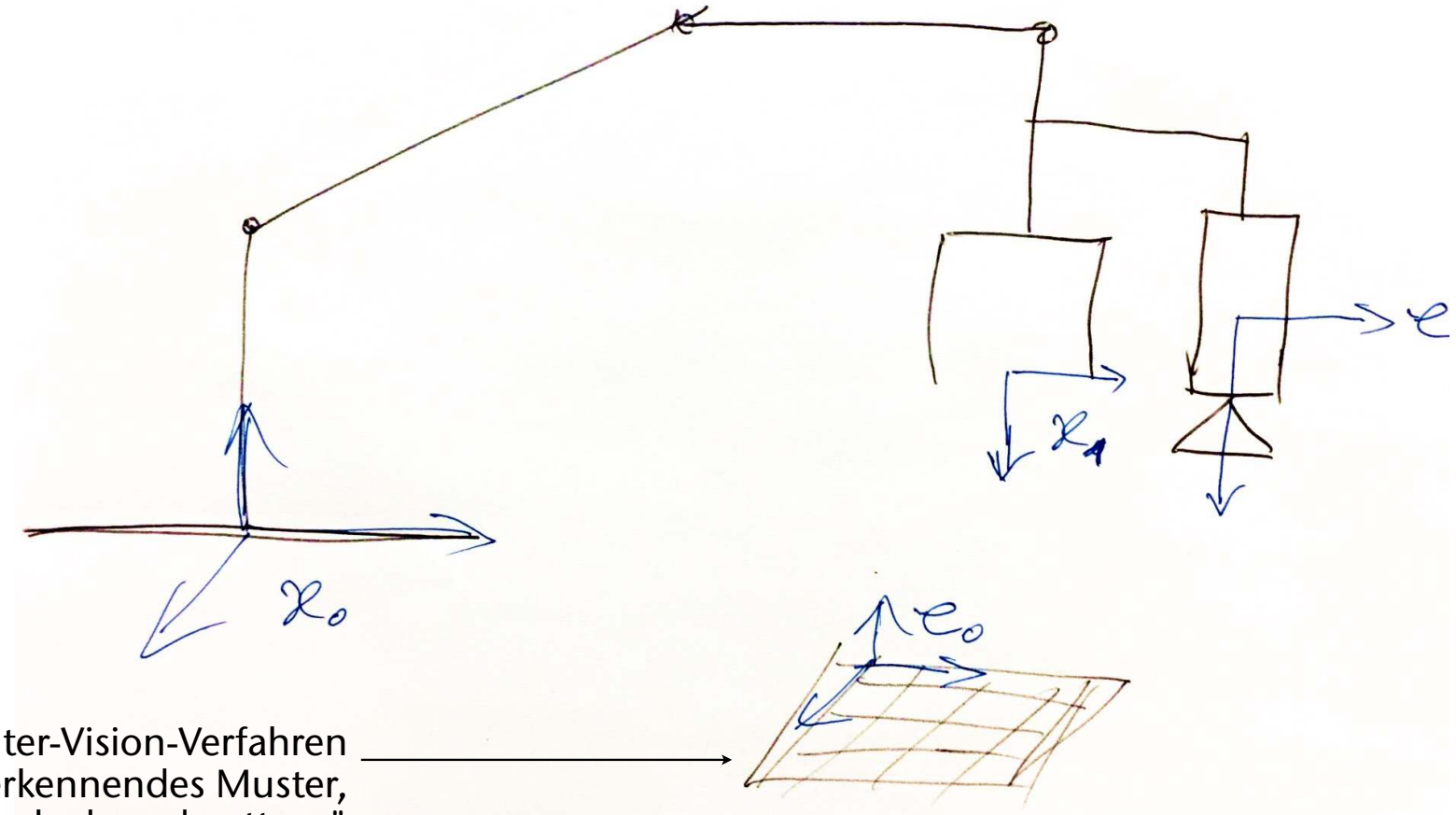
- A.k.a.: Sensor-manipulator calibration, Tracker-HMD-calibration, extrinsische Kamerakalibrierung, ...
- Gegeben:
  Roboter mit Kamera ("eye")
  und Endeffektor (“hand")



- Gesucht: $T_{\mathcal{H}\leftarrow\mathcal{C}}$
- Problem: auch C_1 kann man (oft) nicht bestimmen (relativ zu H? relativ zur Roboter-Basis?)

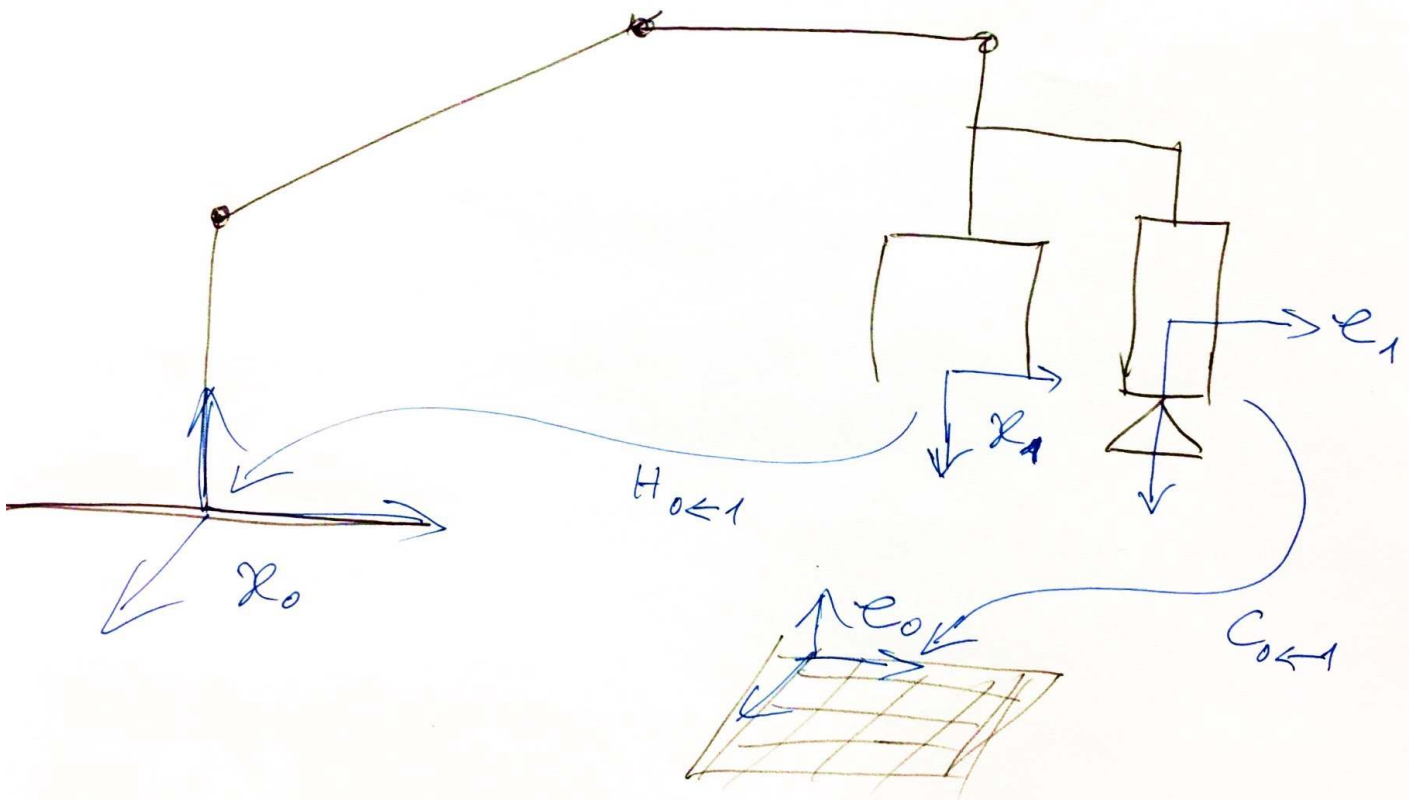
Ansatz

- Führe (zunächst) weitere Koordinaten-systeme ein

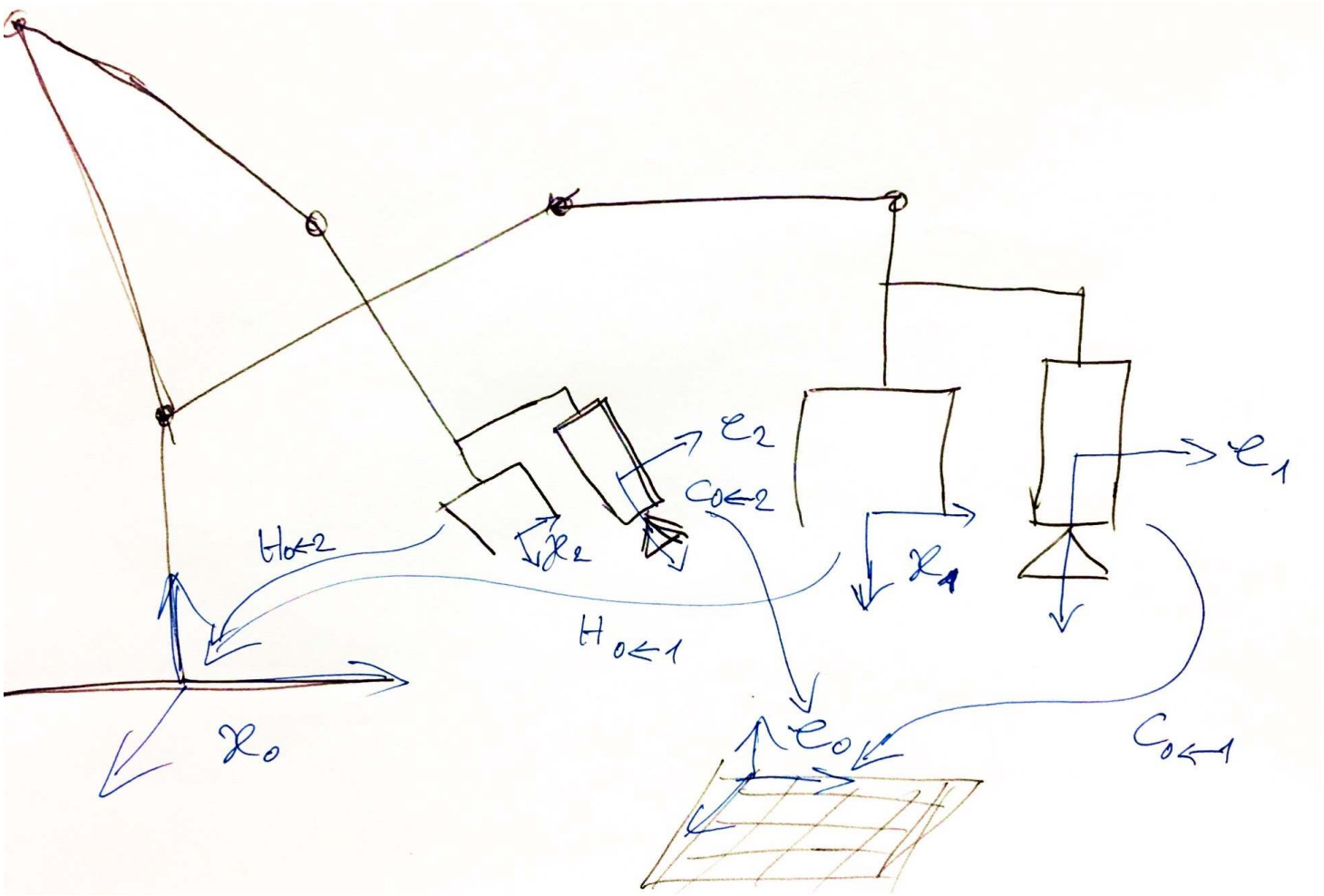


Ein per Computer-Vision-Verfahren “einfach" zu erkennendes Muster, typischerweise ein "checkerboard pattern"

- Bestimme mittels sog. "Vorwärts-Kinematik" $H_{0\leftarrow 1}$ (leicht und präzise)
- Mit Hilfe von Computer-Vision-Verfahren kann man $C_{0\leftarrow 1}$ "relativ leicht" bestimmen (Präzision ist allerdings nicht einfach zu erzielen)



- Fahre nun die Roboter-Hand in Position 2 (fast beliebige)



- Nun gilt:

Punkte in C_1 : P_{e_1}

$$P_{e_1} = \underbrace{T_{x\leftarrow e}^{-1} \cdot H_{0\leftarrow 1}^{-1} \cdot H_{0\leftarrow 2} \cdot T_{x\leftarrow e} \cdot C_{0\leftarrow 2}^{-1} \cdot C_{0\leftarrow 1}}_{\stackrel{!}{=} I} \cdot P_{e_1}$$

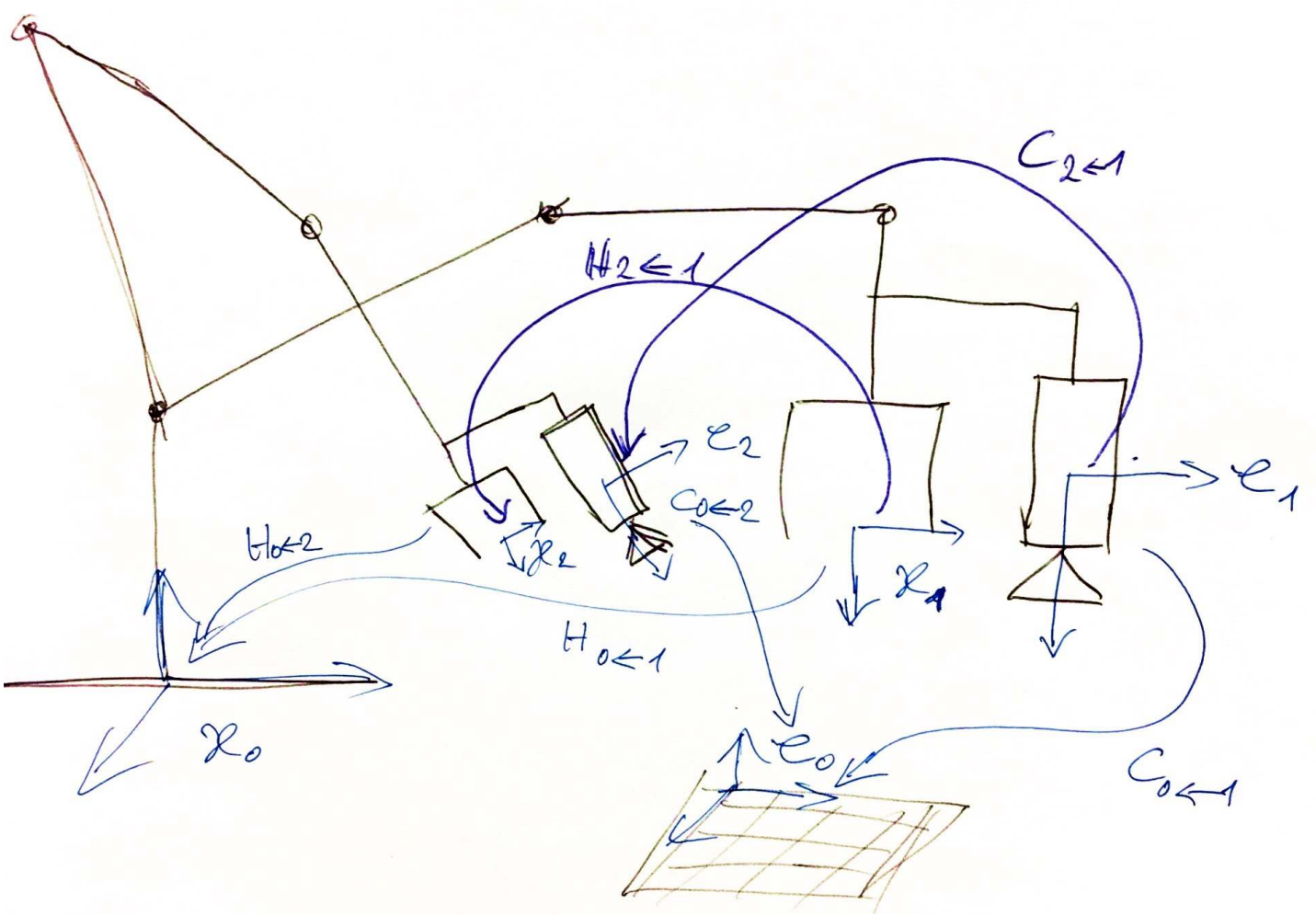
$$\Rightarrow \quad T_{x\leftarrow e} \cdot C_{0\leftarrow 2}^{-1} \cdot C_{0\leftarrow 1} \stackrel{!}{=} H_{0\leftarrow 2}^{-1} \cdot H_{0\leftarrow 1} \cdot T_{x\leftarrow e}$$

$C_{0\leftarrow 2}^{-1} = C_{2\leftarrow 0} = C_{2\leftarrow 1}$

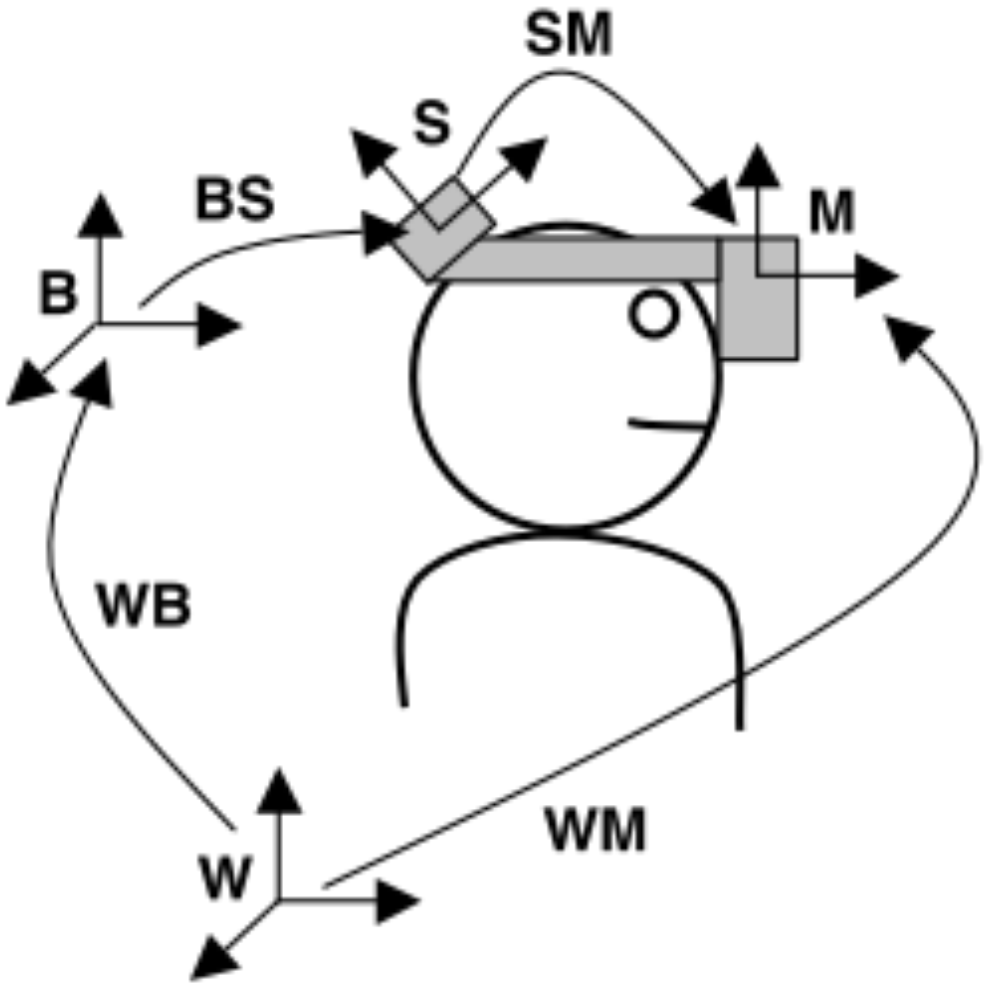
$H_{0\leftarrow 2}^{-1} = H_{2\leftarrow 0} = H_{2\leftarrow 1}$

- Für eine beliebige Position 2 für Hand+Auge gilt:

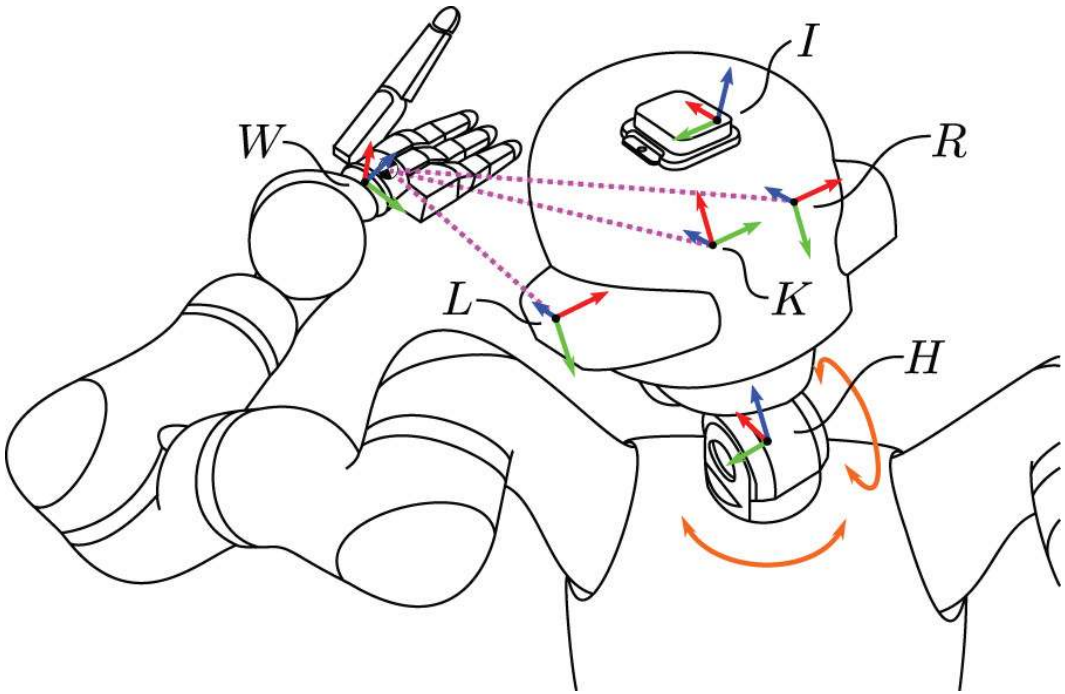
$$T_{\mathcal{H}\leftarrow\mathcal{C}} \cdot C_{2\leftarrow 1} = H_{2\leftarrow 1} \cdot T_{\mathcal{H}\leftarrow\mathcal{C}}$$



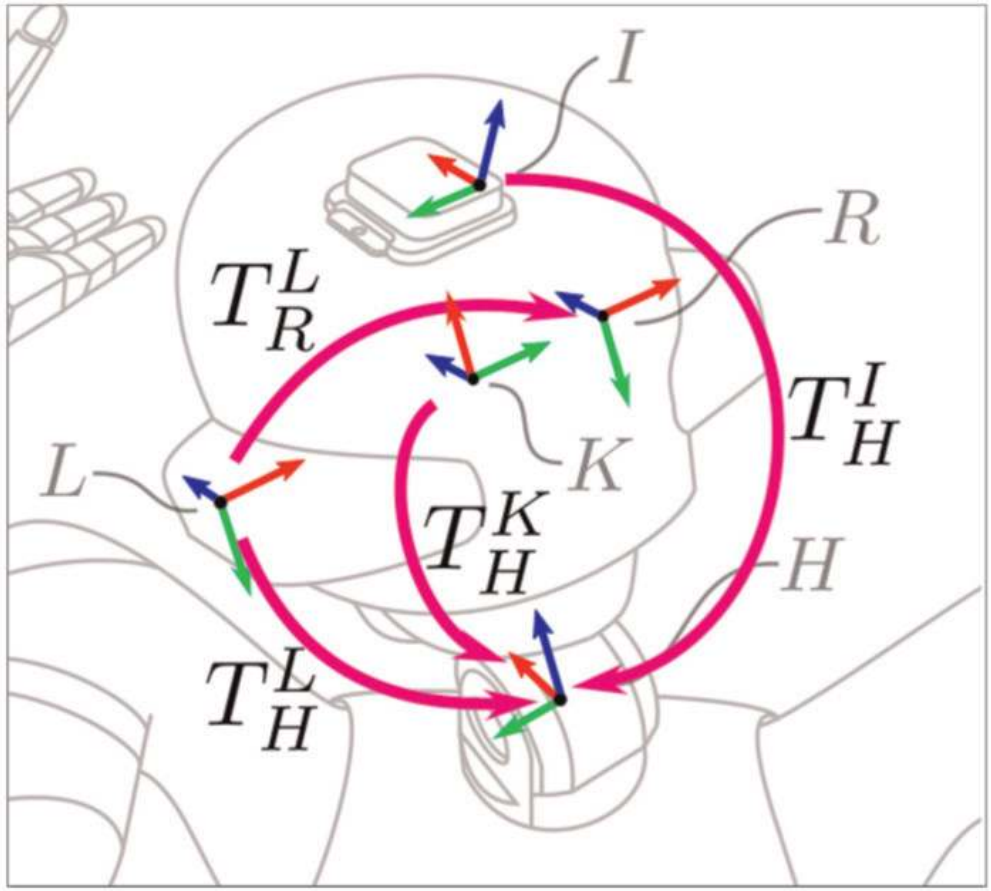
Weiteres Beispiel: Tracker-HMD-Calibration



Weiteres Beispiel: Robotik

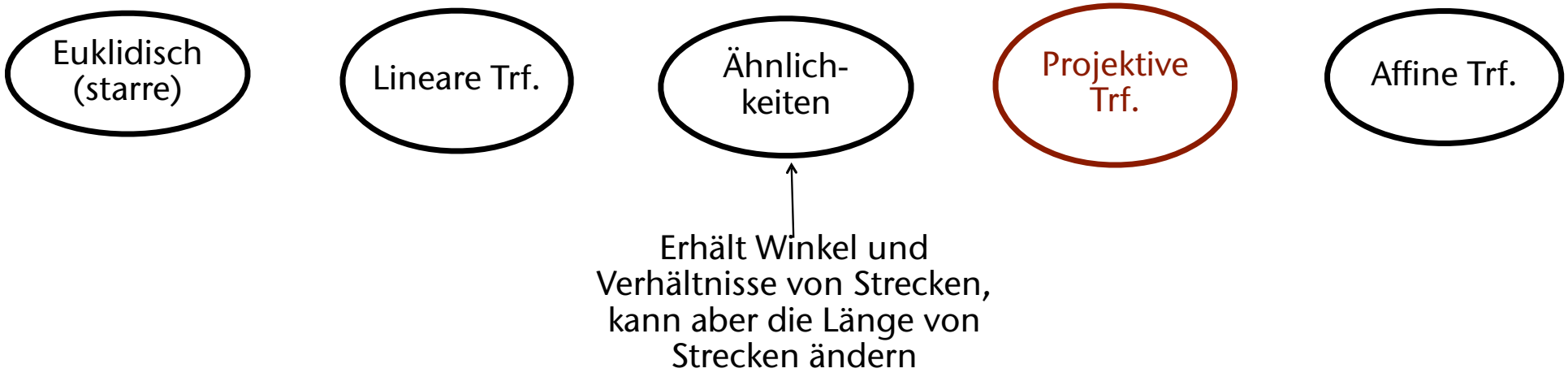


Sketch of the calibration process

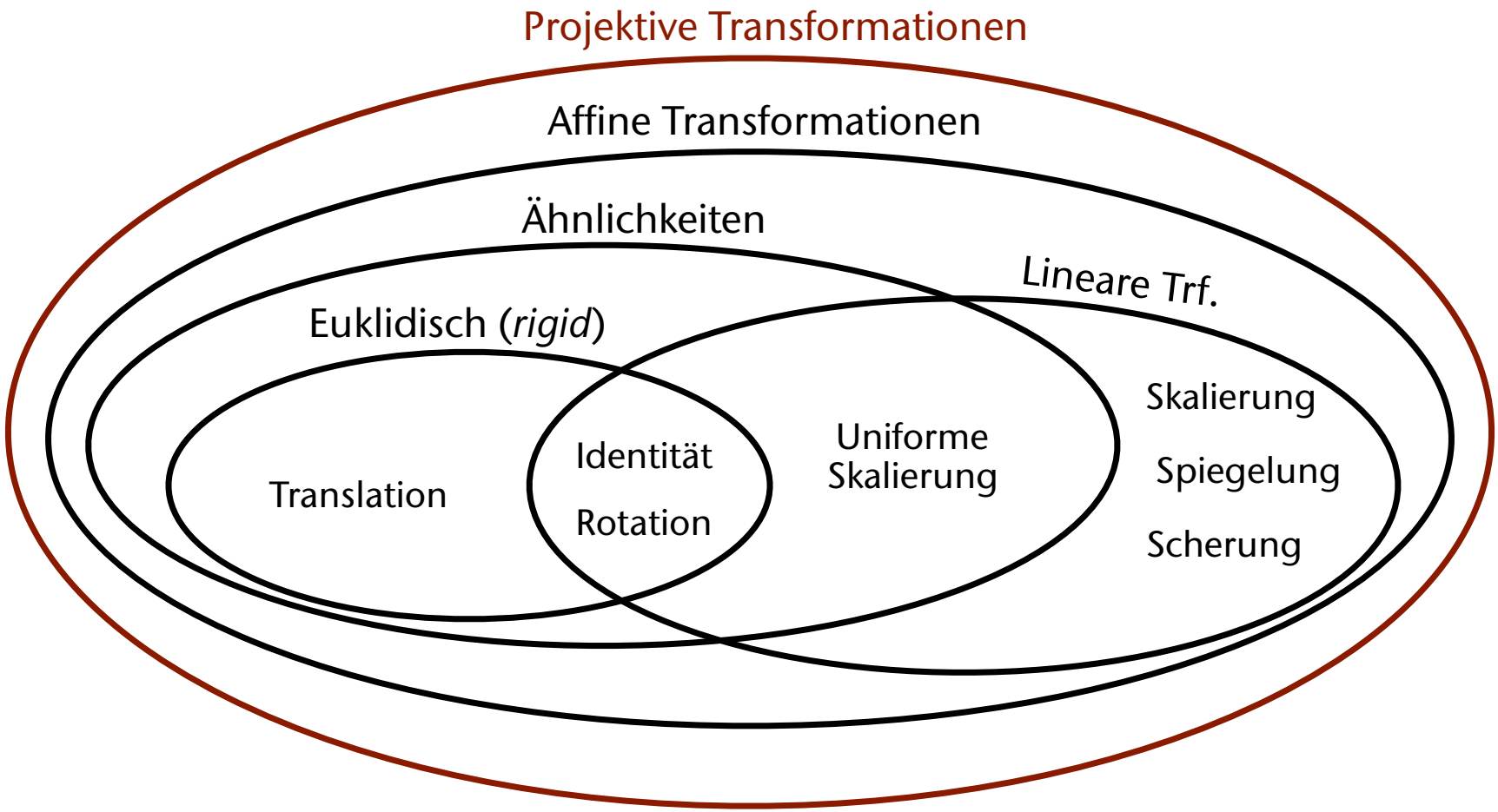


[Birbach et al., 2014]

Klassifikation aller Transformationen



Klassifikation aller Transformationen



Eine Hierarchie von Transformationen (hier in 2D)

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