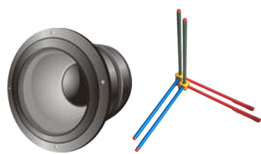
Bremen

CG
VR

Computer-Graphik I
Transformationen

$$\begin{bmatrix} \cos 90^\circ & \sin 90^\circ \\ -\sin 90^\circ & \cos 90^\circ \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} a_2 & -a_1 \end{bmatrix}$$

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



Motivation

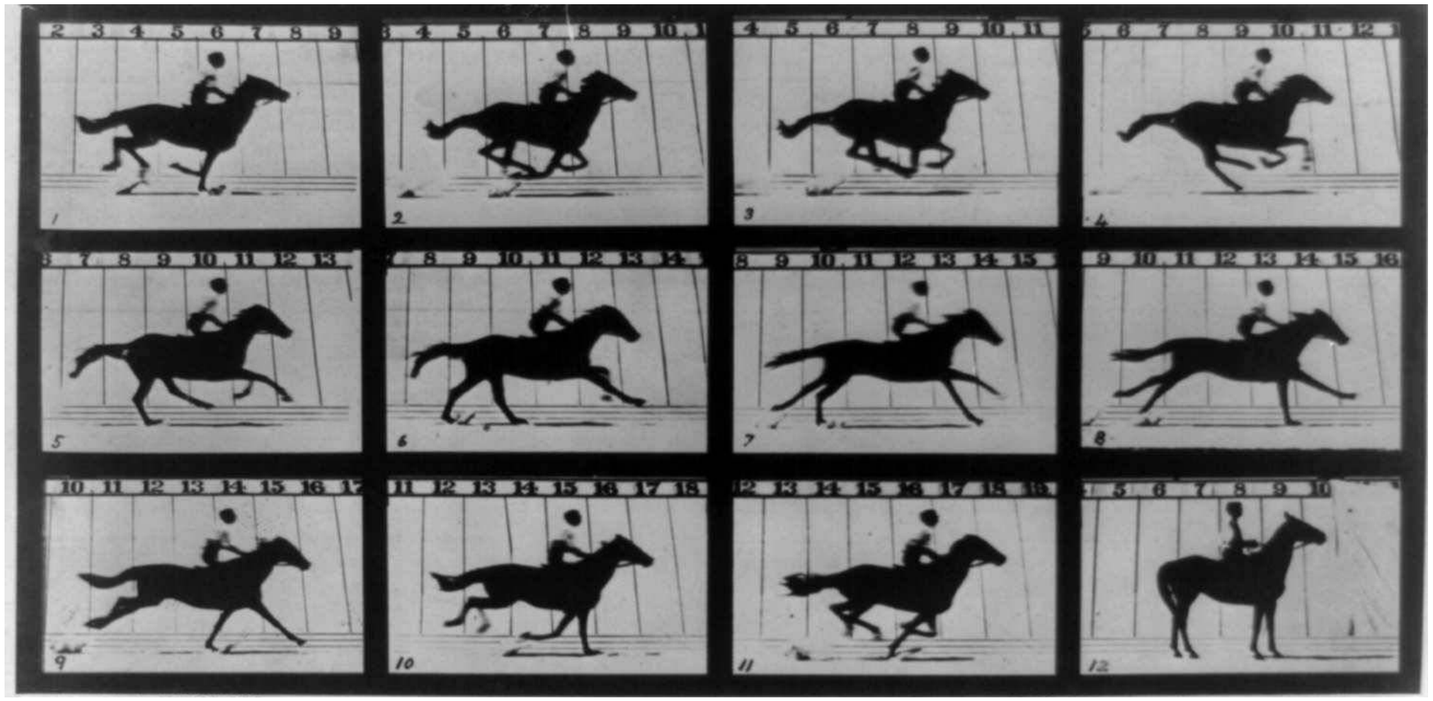
- Transformationen werden benötigt, um ...
- Objekte, Lichtquellen und Kamera zu positionieren
- Alle Berechnungen zur Beleuchtung im selben Koordinatensystem durchzuführen
- Objekte auf den 2D-Bildschirm zu projizieren
- Die Szene zu animieren

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 object
   set new position & orientation for camera
   render scene
```

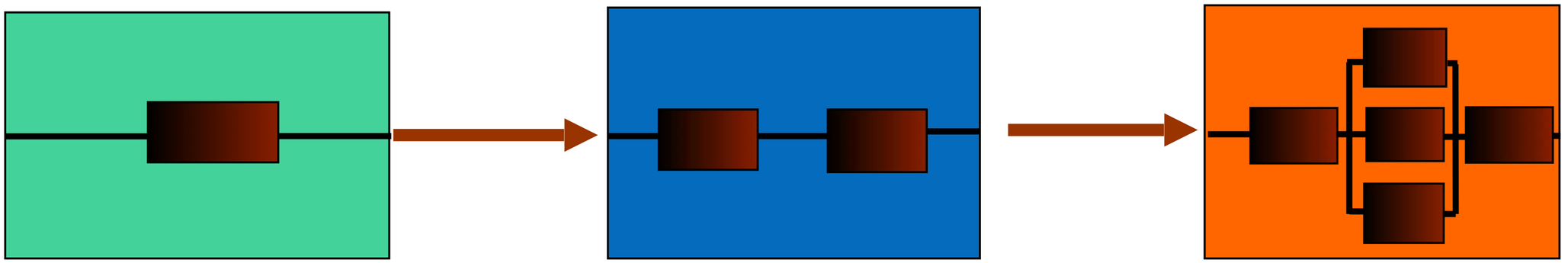
Edward Muybridge, 1878



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)

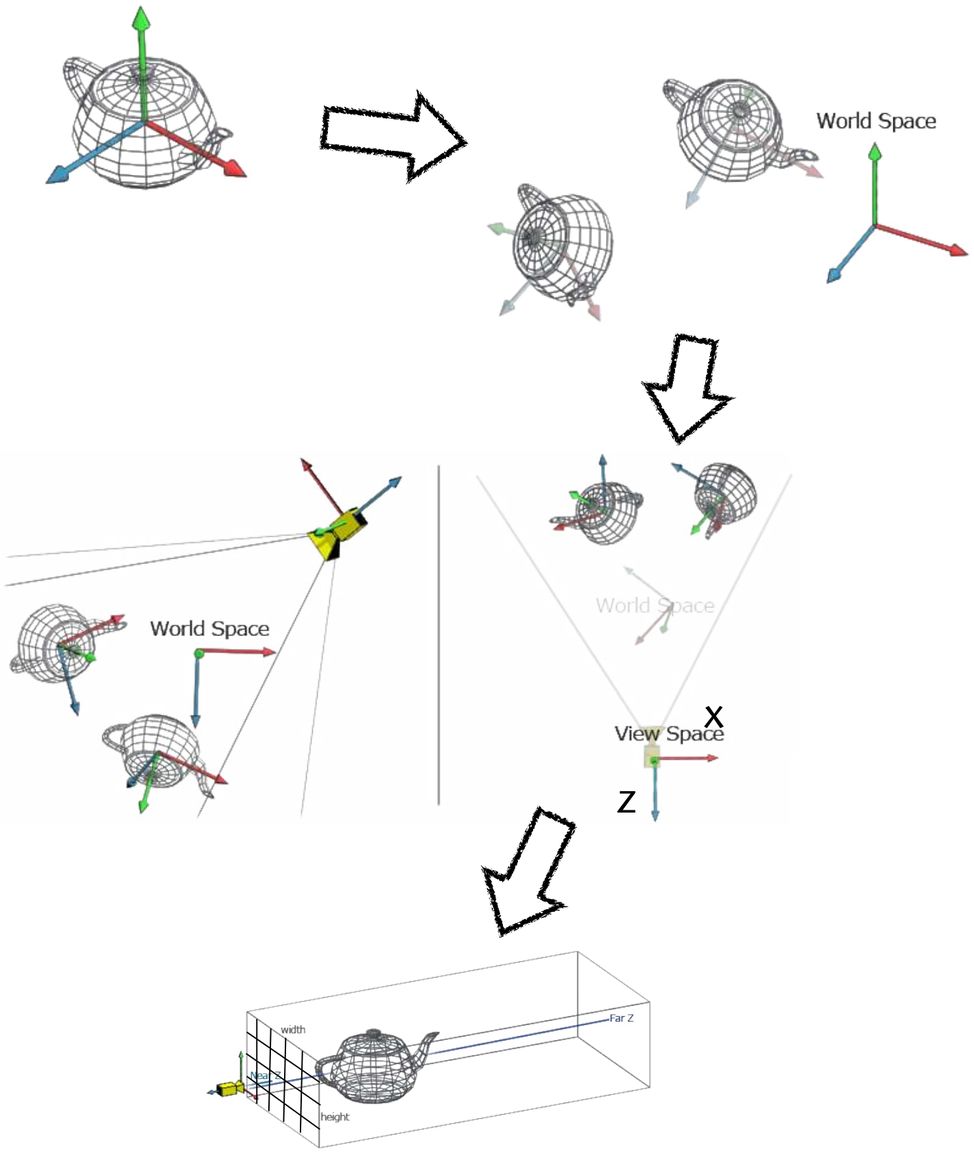
Fragment Processing Fragment Tests

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

World Space
Für alle Objekte gleich

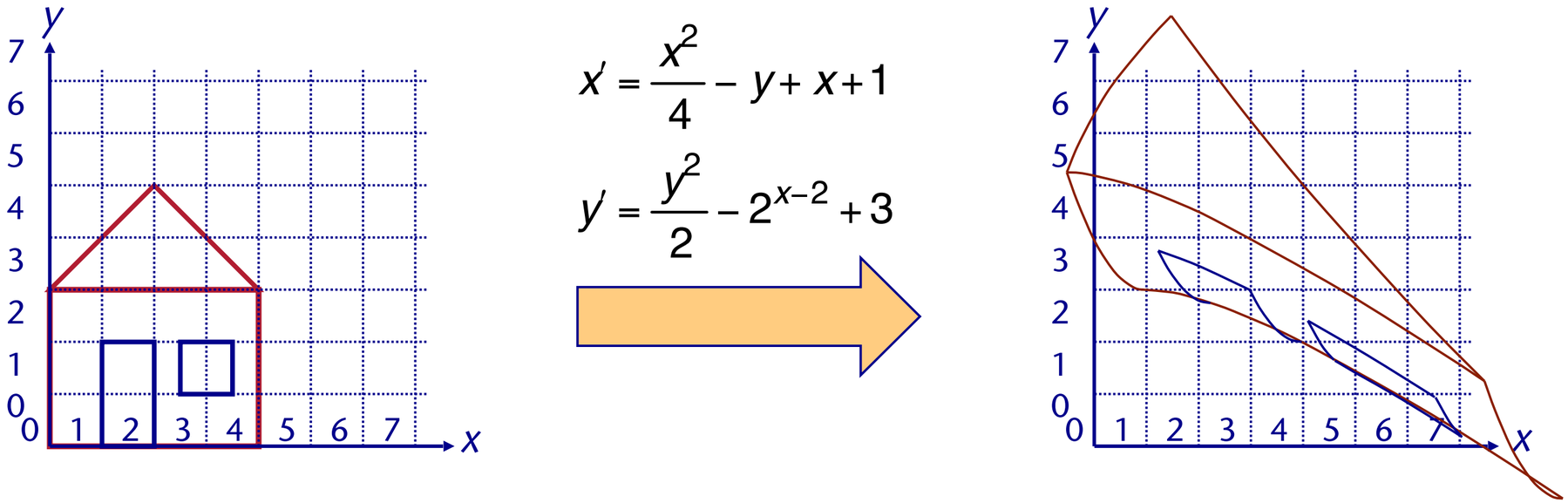
Eye Space / Camera Space

Screen Space
adressiert die Hardware



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

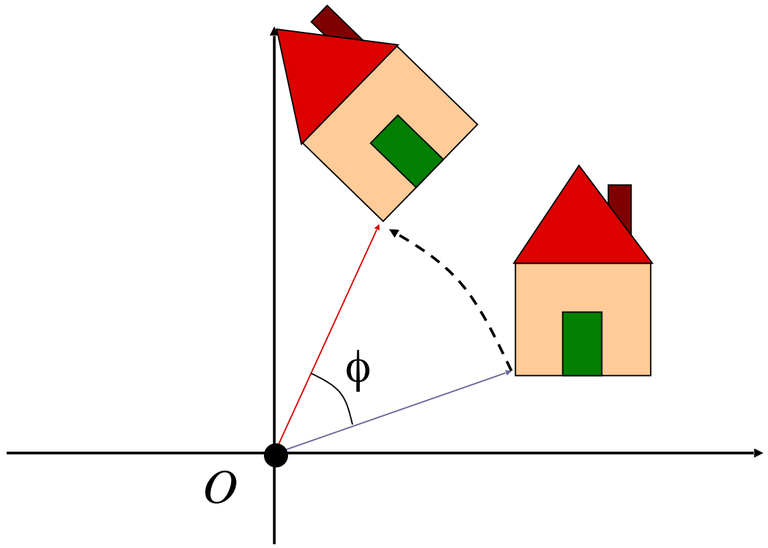
$$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & ?\cos\phi & ?\sin\phi \\ 0 & ?\sin\phi & ?\cos\phi \end{pmatrix}$$

X-Koord. bleibt unverändert

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

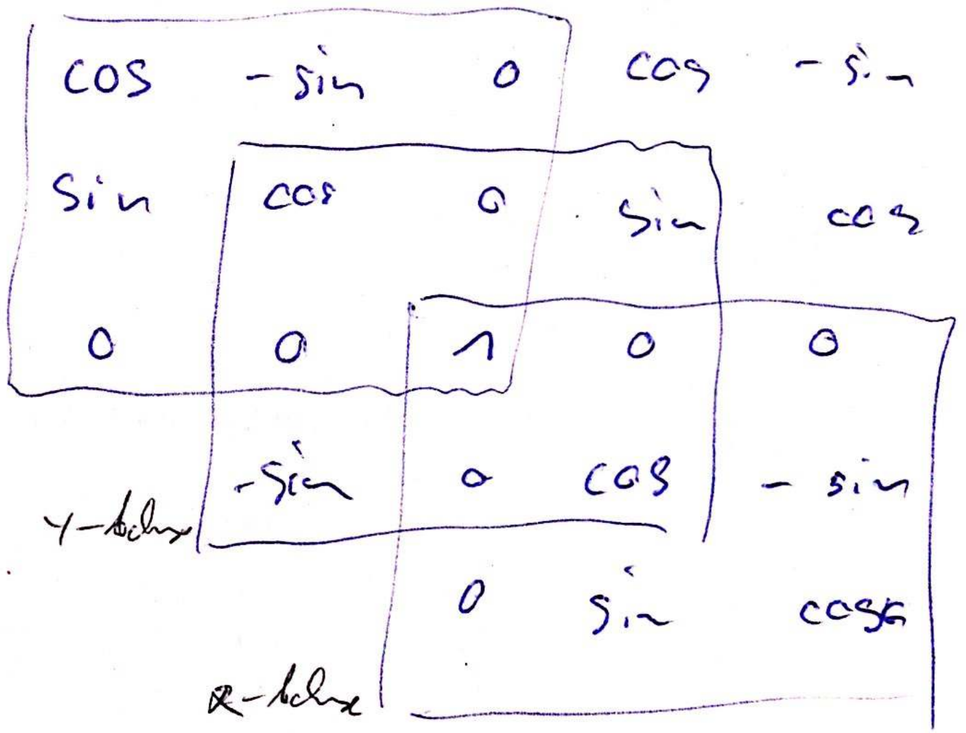
$$R_y(\phi) = \begin{pmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{pmatrix}$$

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



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

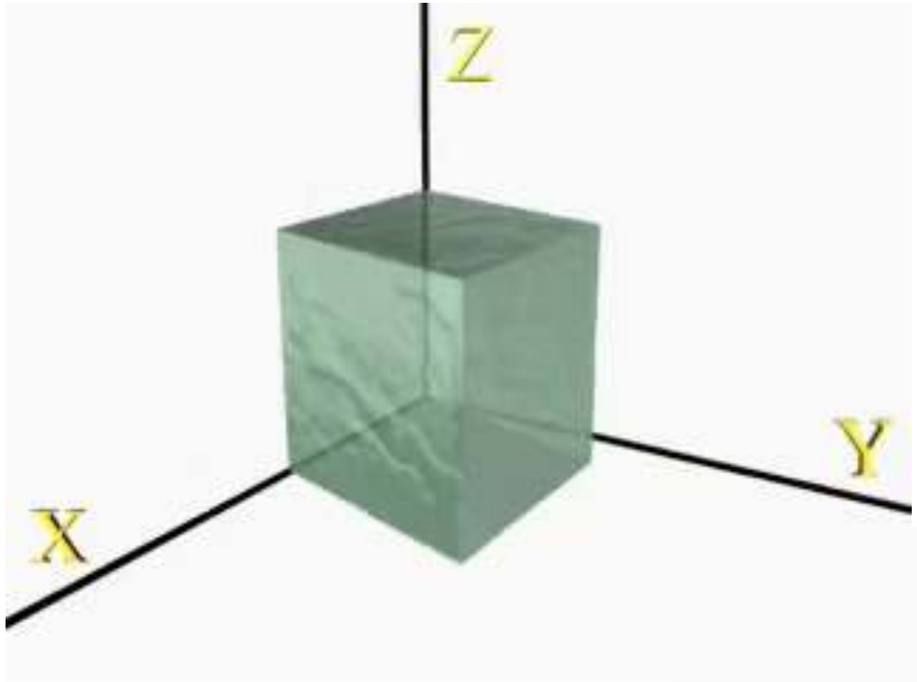
$$H_{xz}(s)\cdot\mathbf{p} = \begin{pmatrix} 1 & 0 & s \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} p_x + sp_z \\ p_y \\ p_z \end{pmatrix}$$

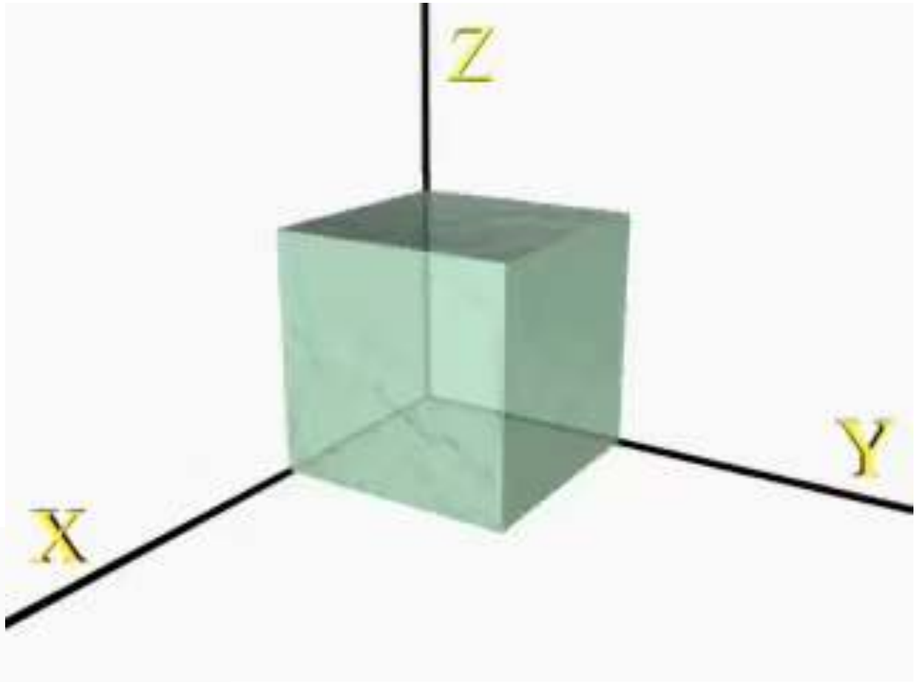
- Insgesamt: 6 mögliche Scherungen
- Inverse:

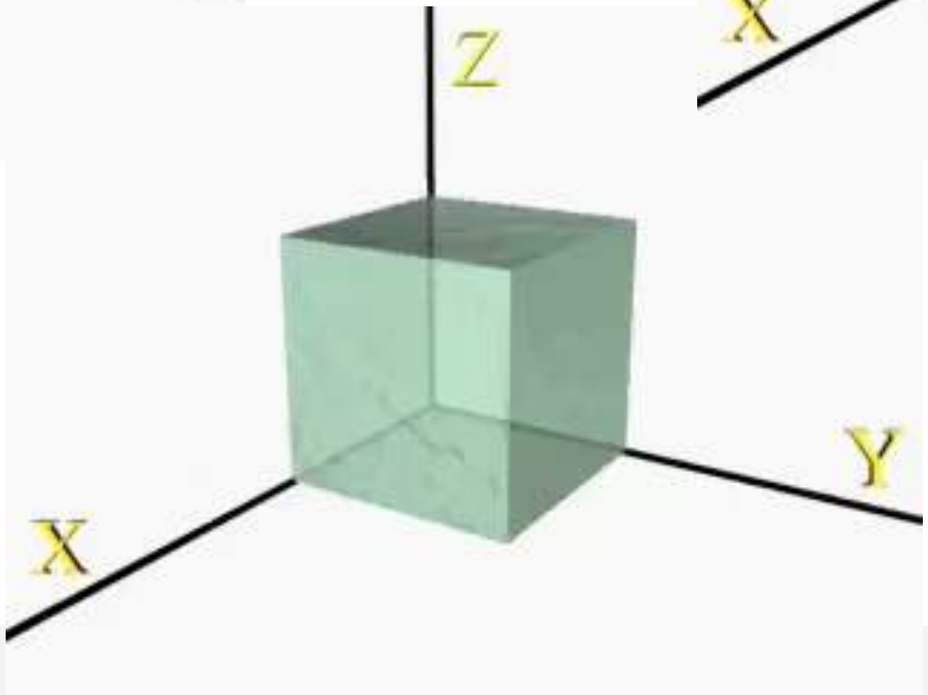
$$H_{xz}^{-1}(s) = H_{xz}(-s)$$

- 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

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!

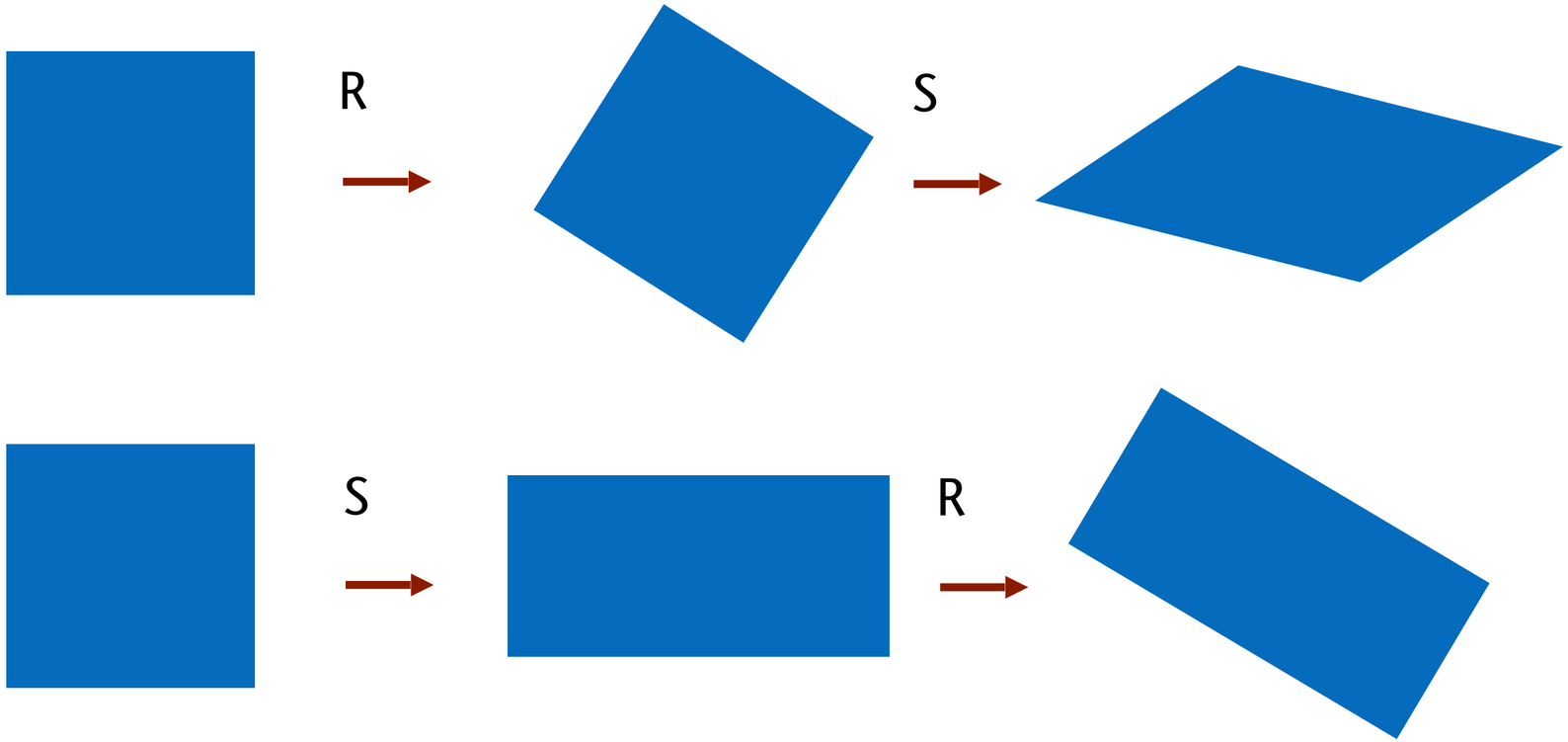
Translation (a.k.a. Verschiebung)

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

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

Verknüpfung / *Concatenation*

- Beispiel:



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

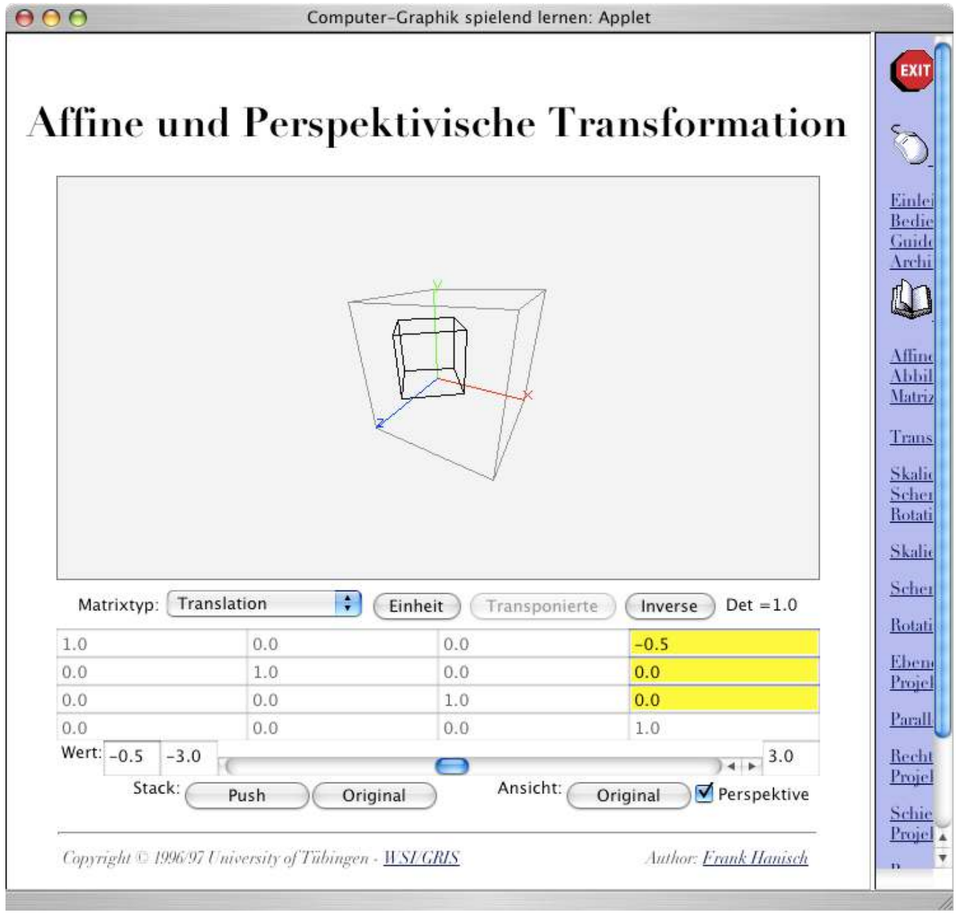
- Reihenfolge in einer Matrixkette:

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

Reihenfolge der Ausführung

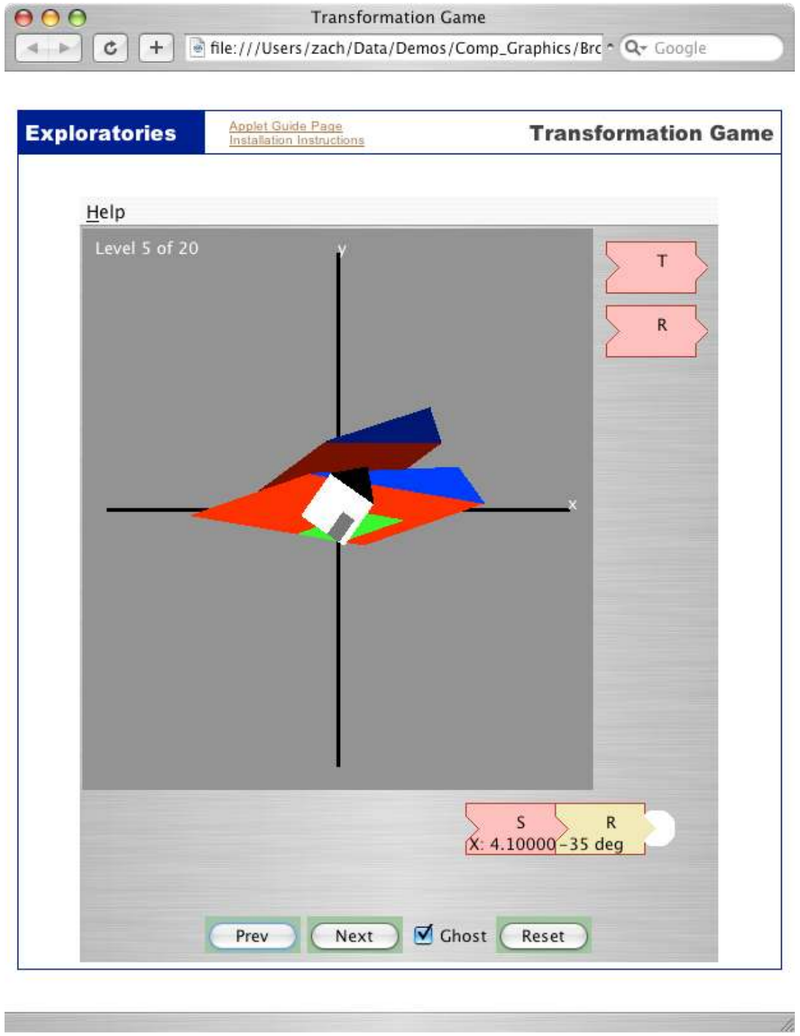
- Nützlich zur Steigerung der Effizienz

Demo zur Konkatenation von Transformationen



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

Demo

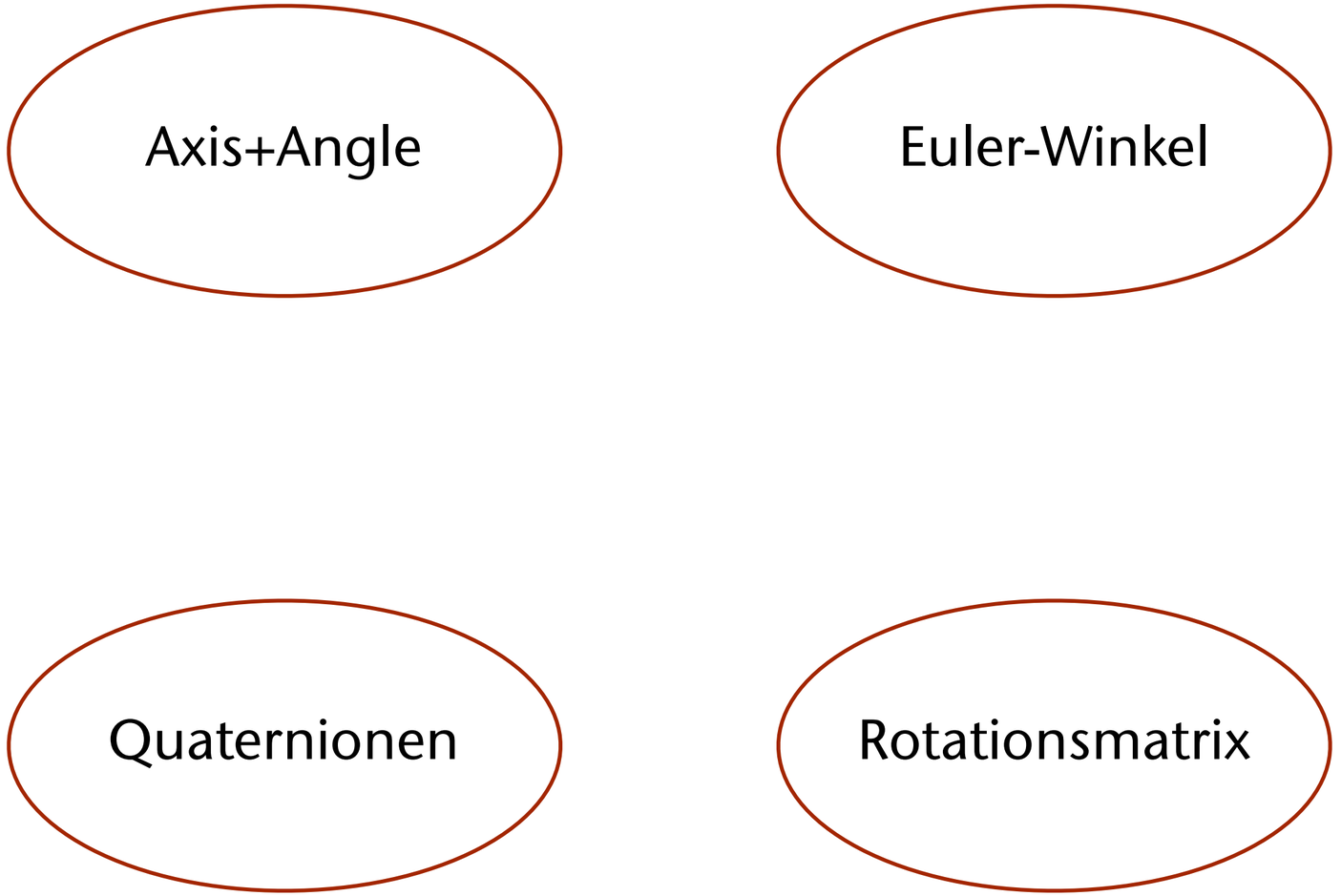


http://graphics.cs.brown.edu/research/exploratory/freeSoftware
⟶ Complete Catalog ⟶ Transformation Game

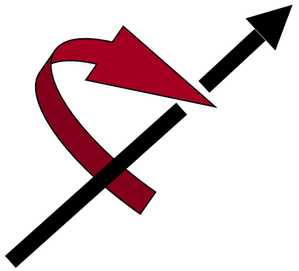
http://cgvr.cs.uni-bremen.de
Teaching ⟶ CG1
⟶ "Transformation Game" suchen, ZIP-File entpacken, dann

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

Vier Darstellungen von allgemeinen Rotation



Darstellung: Achse + Winkel



- Intuitive Darstellung: $(\varphi, \mathbf{r})$
  - Meist mit der Nebenbedingung $\|\mathbf{r}\| = 1$
- Anzahl Freiheitsgrade allgemeiner Rotationen: ^{3} DOFs (*degrees of freedom*)
  - 2 DOFs für die Achse + 1 DOF für den Winkel
- Anmerkung: in der Robotik wird gerne die Variante verwendet, wo

$$\varphi = \|\mathbf{r}\|$$

(damit benötigt man also wieder nur einen 3D-Vektor)

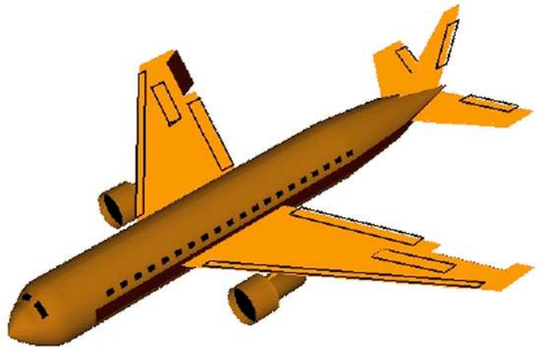
  - Bezeichnungen: Euler-Vektor, oder Rotationsvektor

Die Euler-Winkel

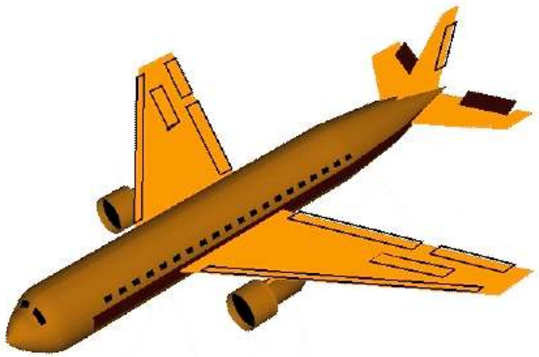
- Intuitive (?) Konstruktion einer beliebigen Rotation: Konkatenation aus 3 elementaren Rotationen — erst Rotation um X-, dann um Y-, dann um Z-Achse
- Diese Winkel heißen Euler-Winkel und bezeichnen meistens Rotationen um eine der drei Welt-Koordinaten-Achsen
- Häufige Variante im Maschinenbau:

$$R(r,p,y) = R_z(y)R_x(p)R_y(r)$$

- Bezeichnung:
  roll, pitch, yaw (Schiff)
  roll, pitch, heading (Flugzeug)



Roll



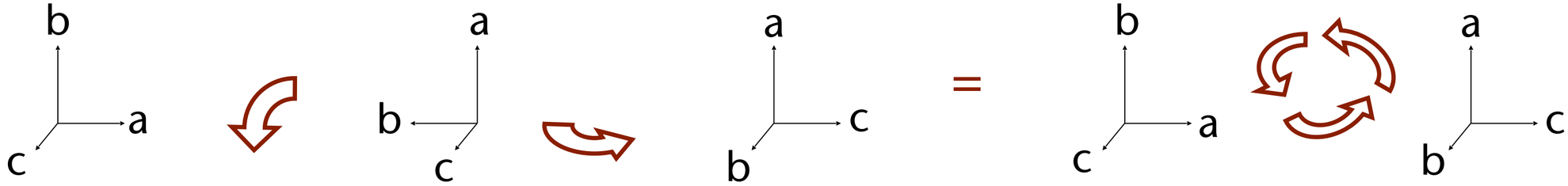
Pitch

Yaw
(Heading)

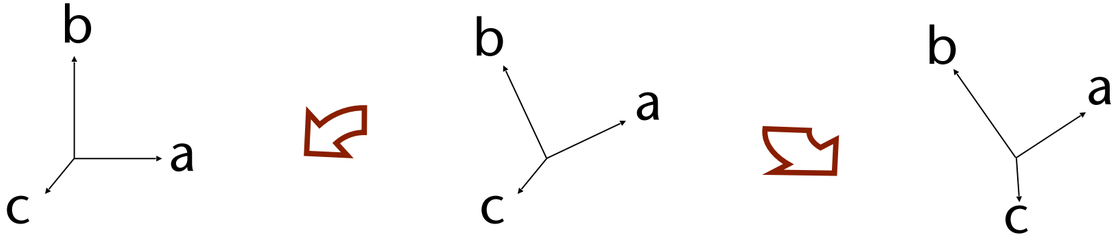
Spezifikation einer Rotation mittels Euler-Winkeln macht viele Probleme!!

- Reihenfolge der Rotationen ist nicht festgelegt!
  - Oft *"roll, pitch, yaw"* — aber Zuordnung zu den Achsen nicht definiert!
  - Manchmal auch: z, x, z ! Oder ...
- Führt manchmal zu Gimbal Lock!
- Werte-Bereiche: für einen der Winkel ist der Bereich nur [-90, +90]
  - Sonst: Mehrfachüberdeckung von SO(3)
- Rechnen (z.B. interpolieren oder mitteln) ist i.A. sinnlos und/oder unmöglich!

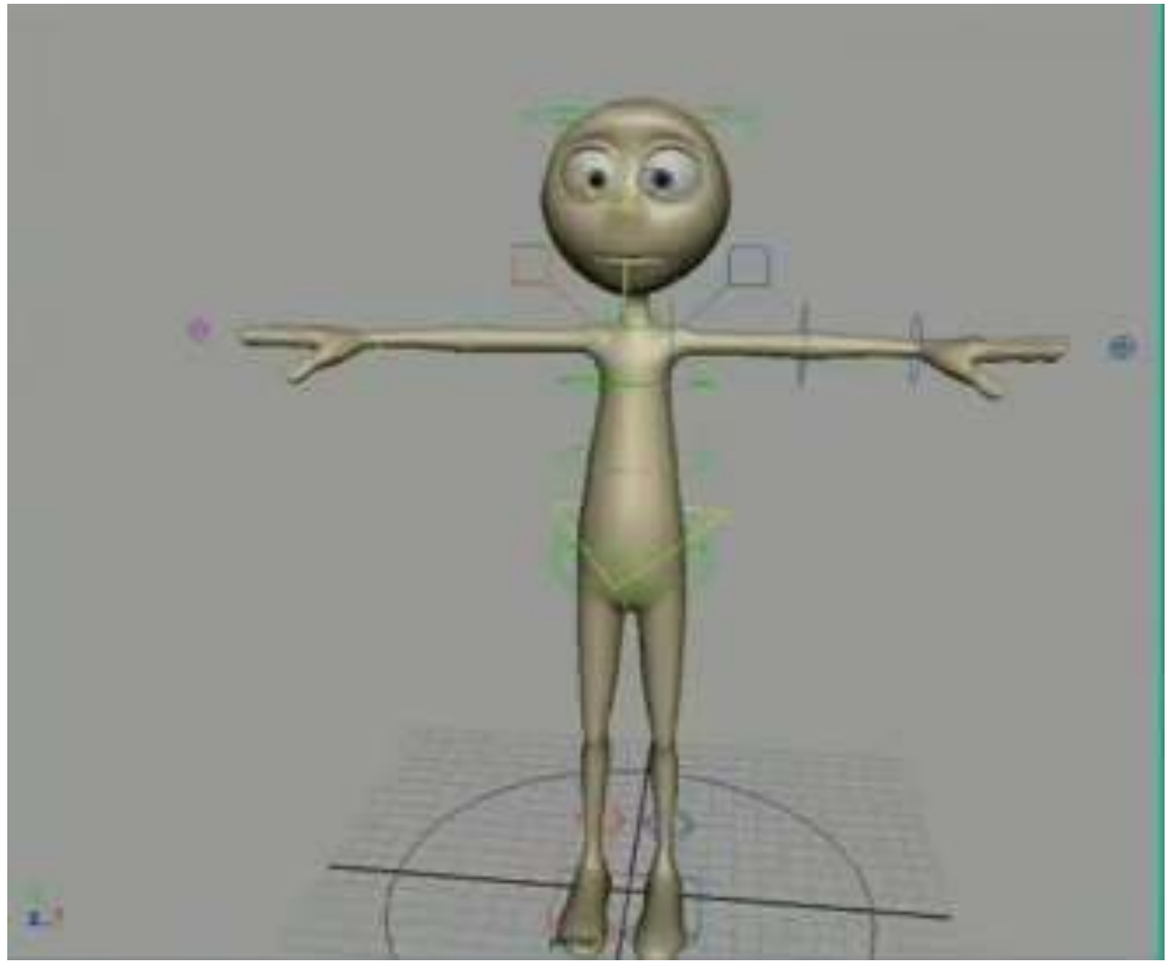
- Interpolation zwischen zwei Rotationen (Orientierungen) mittels Euler-Winkel liefert unerwünschte Resultate
- Beispiel:
  - Rotation von 90° um Z, dann 90° um Y = 120° um (1, 1, 1)

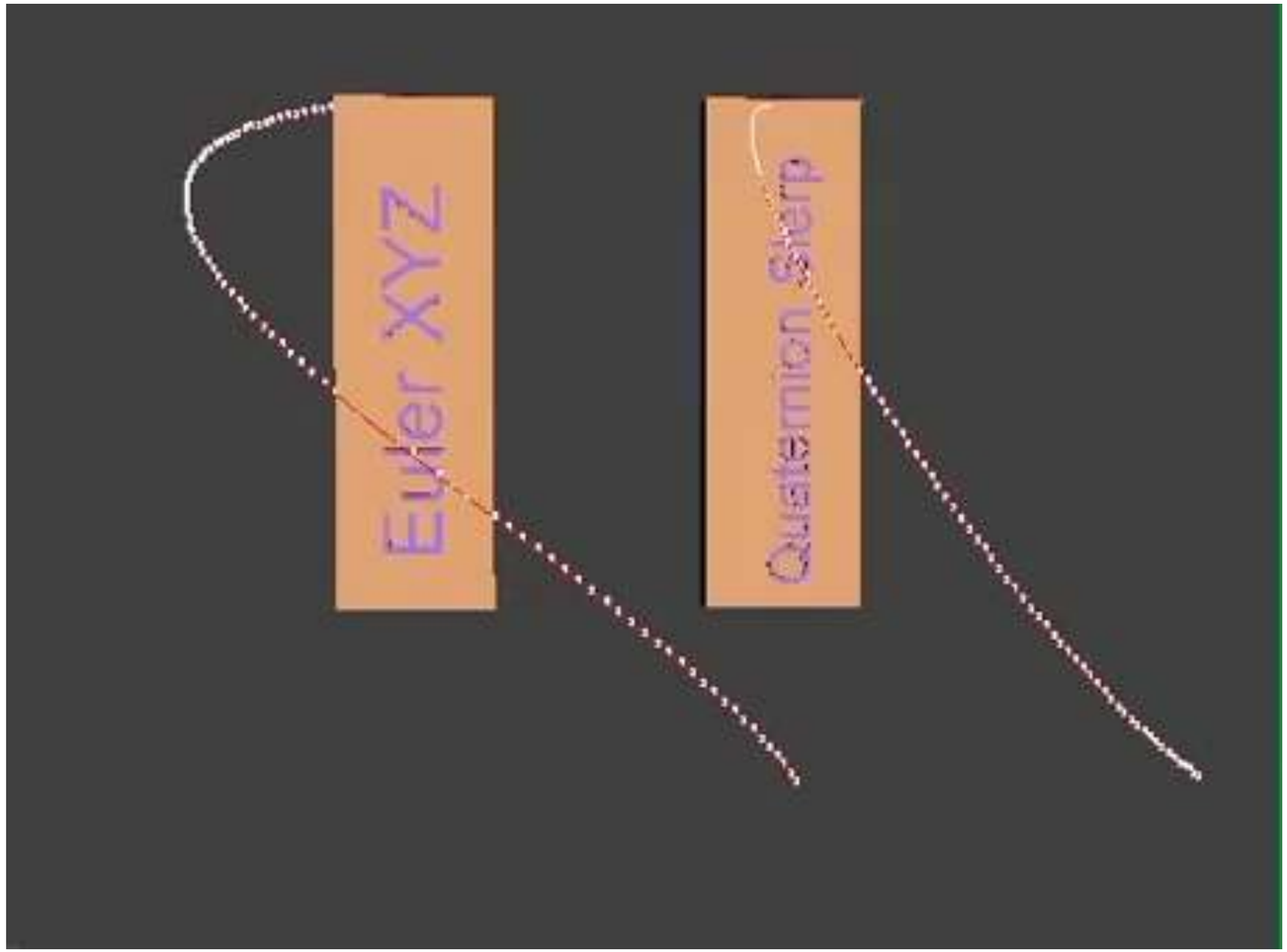


  - Rotation von 30° um Z, dann 30° um Y ≠ 40° um (1, 1, 1) !



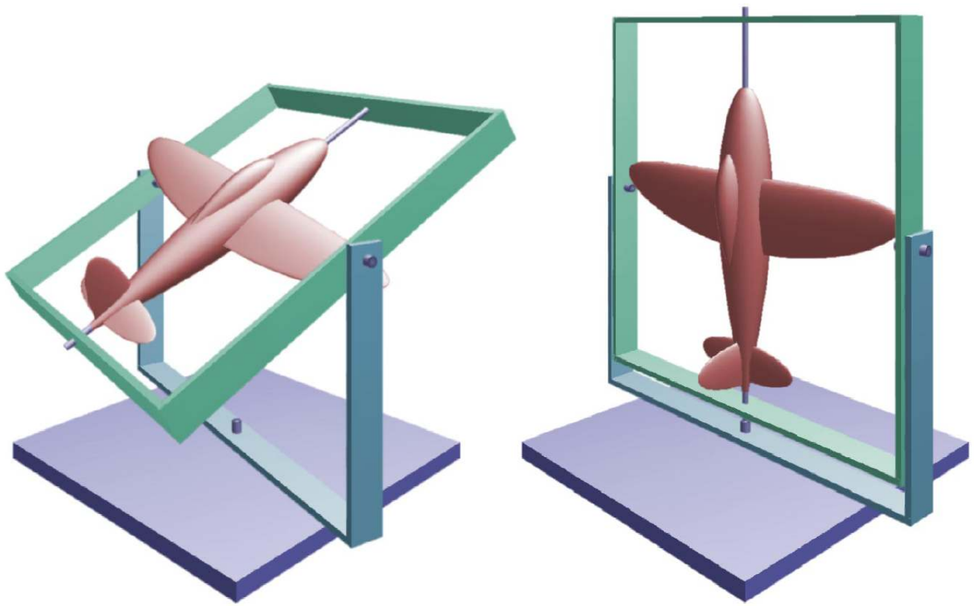
Beispiel für die unerwünschten Effekte bei Interpolation von Euler-Winkeln





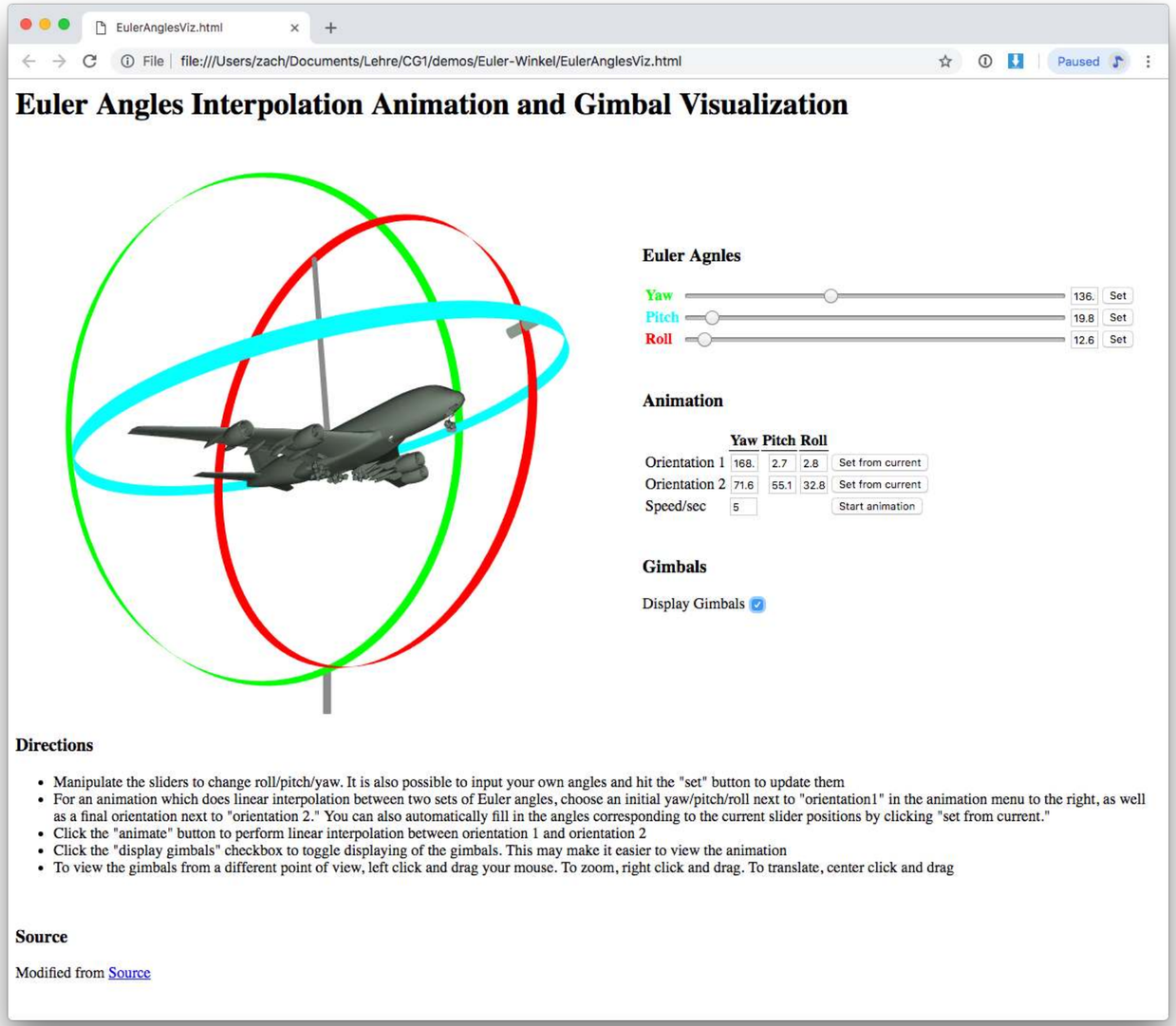
Der *Gimbal Lock*

- Immer dann, wenn 2 Achsen (fast) gleich sind
- Folgen: nur noch 2 Freiheitsgrade! (obwohl es immer noch 3 Parameter sind)
  - Rotation um eine der (lokalen) Flugzeugachsen geht nicht mehr!



4 Gimbals

Demo zu Interpolation und Gimbal Lock



Anekdote: Euler Angles in Spaceflight

- In den Apollo-Raketen wurde ein Kreiselkompass verwendet
  - (Basiert auf der Trägheit einer sich sehr schnell drehenden Masse)
- Kleine Anekdote dazu:

  About two hours after the Apollo 11 landing, Command Module Pilot Mike Collins had the following conversation with CapCom Owen Garriott:
  104:59:35 *Garriott*: Columbia, Houston. We noticed you are maneuvering very close to gimbal lock. I suggest you move back away. Over.
  104:59:43 *Collins*: Yeah. I am going around it, doing a CMC Auto maneuver to the Pad values of roll 270, pitch 101, yaw 45.
  104:59:52 *Garriott*: Roger, Columbia. (Long Pause)
  105:00:30 *Collins*: (Faint, joking) How about sending me a fourth gimbal for Christmas.

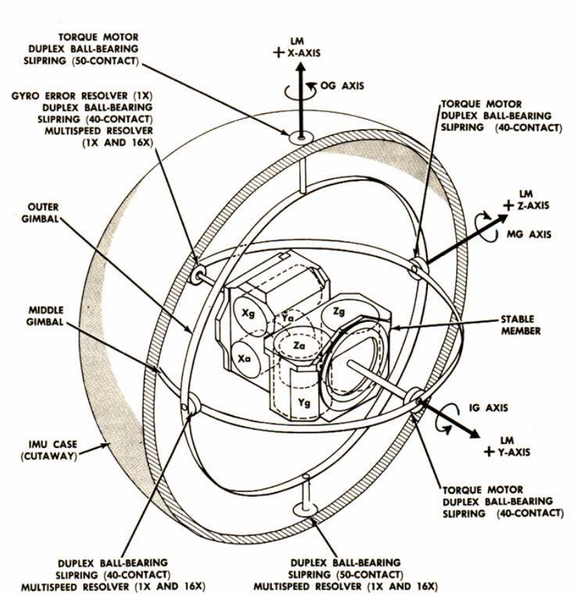


Figure 2.1-24. IMU Gimbal Assembly

- Um Gimbal-Lock zu vermeiden, wurde tatsächlich ein 4-ter Gimbal eingeführt!

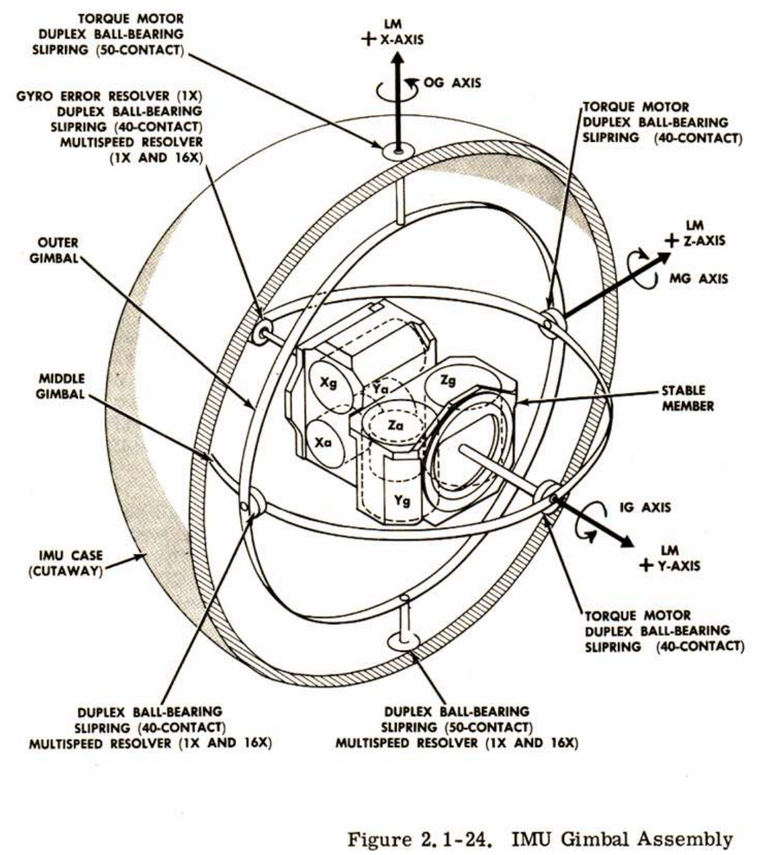
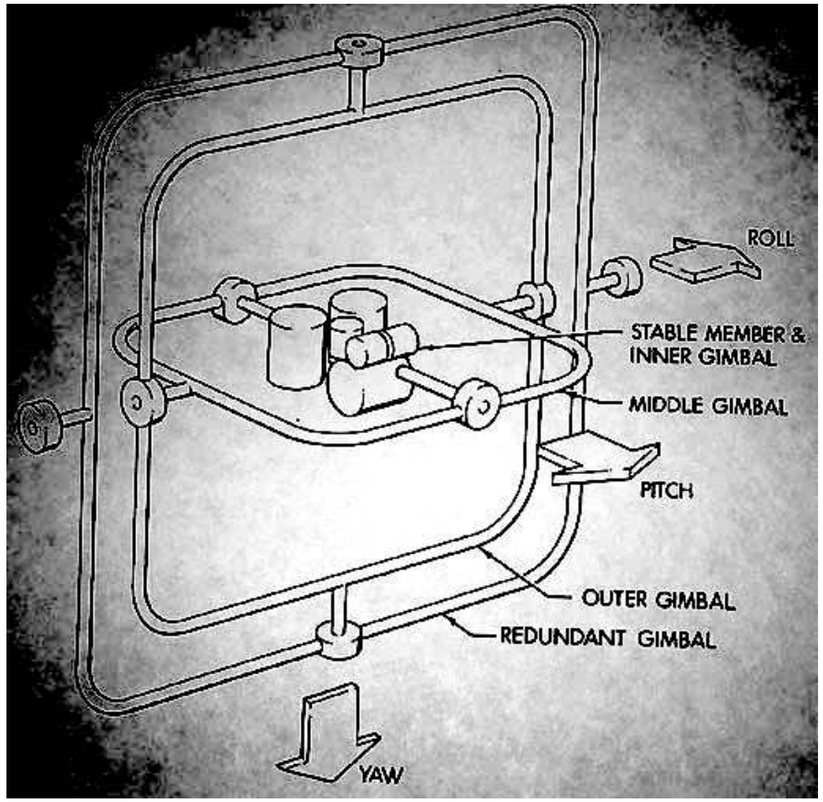


Figure 2.1-24. IMU Gimbal Assembly



Quelle: http://www.hq.nasa.gov/office/pao/History/alsj/gimbals.html

Matrix-Darstellung von Rotationen

- Gesucht: Rotationsmatrix zu gegebener Rotationsachse **r**
  (oBdA geht **r** durch den Ursprung)

1. Erzeuge neue Basis (**r**, **s**, **t**) (bestimme **s** und **t**, orthogonal zu **r**; siehe Kapitel "Kurze Wiederholung in Geometrie")
2. Transformiere alles, so daß die Basis (**r**, **s**, **t**) in die Standardbasis (**x**, **y**, **z**) übergeht
3. Rotiere mit Winkel θ um **x**-Achse
4. Transformiere zurück in die ursprüngliche Basis

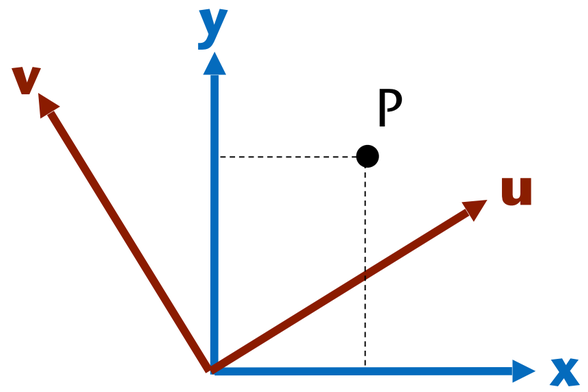
- Zusammen:

$$M = BR_x(\theta)B^{\mathsf{T}} \quad \text{mit} \quad B = \begin{pmatrix} | & | & | \\ \mathbf{r} & \mathbf{s} & \mathbf{t} \\ | & | & | \end{pmatrix}$$

Konstruktion einer Rotation aus Koordinatenachsen

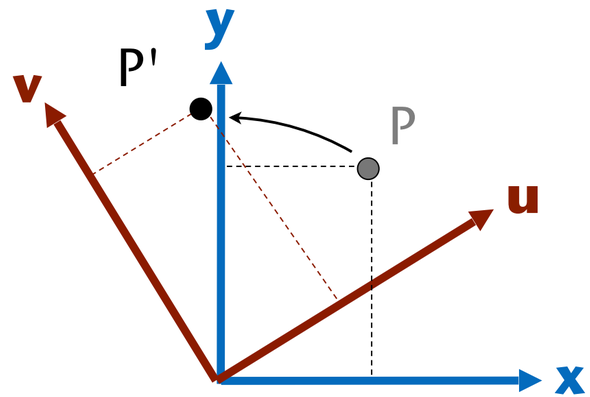
- Man kann eine Rotationsmatrix direkt aus den 3 Koordinatenachsen eines (neuen) Koordinatensystems konstruieren
- Gegeben: Einheitsvektoren **u**, **v**, **w**
- Setze:

$$R = \begin{pmatrix} | & | & | \\ \mathbf{u} & \mathbf{v} & \mathbf{w} \\ | & | & | \end{pmatrix}$$



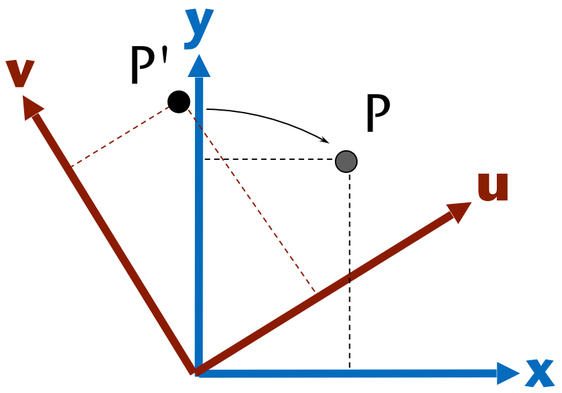
- Damit ist $R \cdot R^T = I$ und $\det(R) = 1$
- Also: R ist eine Rotation
- Und zwar von xyz-Koordinaten $\longrightarrow$ uvw, denn:

$$R \cdot \mathbf{e}_x = \mathbf{u}, \quad R \cdot \mathbf{e}_y = \mathbf{v}, \quad R \cdot \mathbf{e}_z = \mathbf{w}$$



- Was bedeutet die Matrix R^{-1} ?

➢ Rotation von uvw-Koord. → xyz-Koord.



- Beachte: bei dieser Betrachtungsweise sind die Koordinaten der Punkte (bzw. Ortsvektoren) **IMMER** im xyz-Koord.system gegeben!
  - Denn auch die Vektoren **u**, **v**, **w**, haben wir immer in xyz-Koord. definiert!

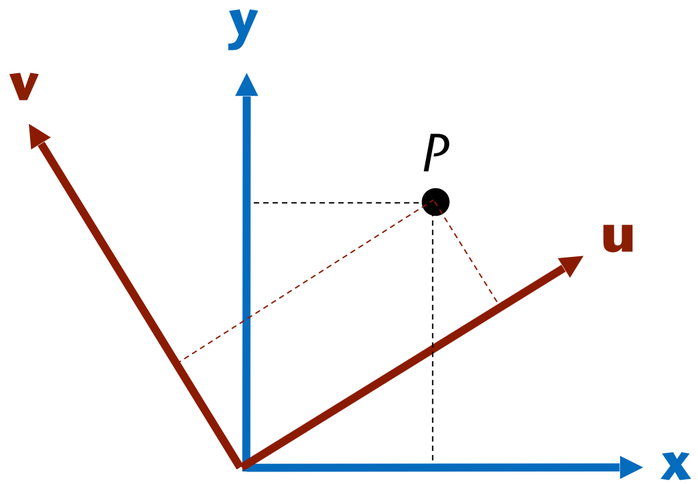
Zusammenhang zwischen Rotation und Basiswechsel

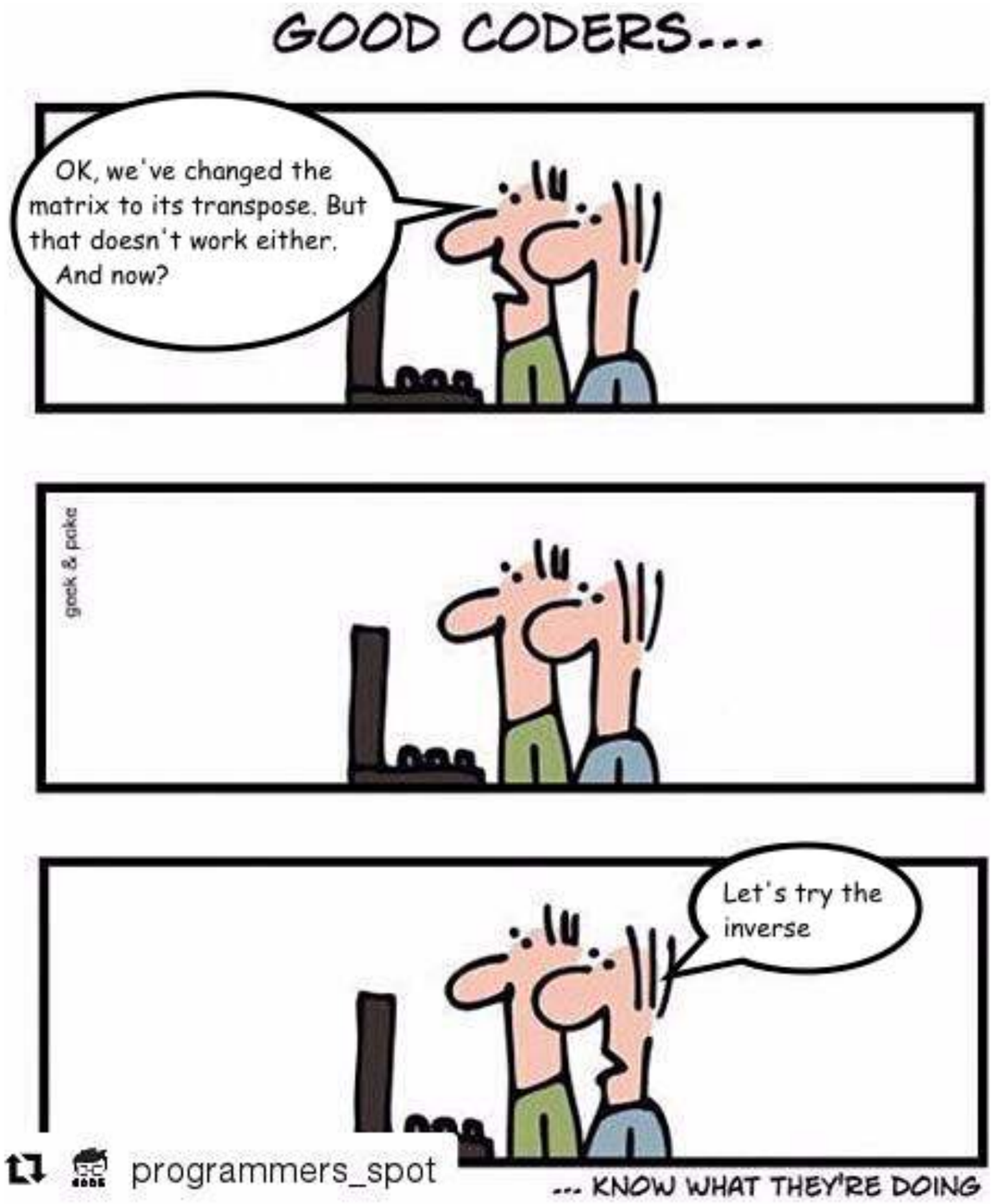
- Betrachte einen Punkt $P = \mathbf{p}_{xyz}$ im xyz-Koord.system und multipliziere mit R^T:

$$R^T \cdot P_{xyz} = R^T \cdot \mathbf{p}_{xyz} = \begin{pmatrix} - & \mathbf{u} & - \\ - & \mathbf{v} & - \\ - & \mathbf{w} & - \end{pmatrix} \cdot \mathbf{p}_{xyz} = \begin{pmatrix} \mathbf{u} \cdot \mathbf{p}_{xyz} \\ \mathbf{v} \cdot \mathbf{p}_{xyz} \\ \mathbf{w} \cdot \mathbf{p}_{xyz} \end{pmatrix} = P_{uvw}$$

➢ P_{uvw} stellt den **selben** Punkt P im Raum wie P_{xyz} dar!

- puvw stellt ihn in uvw-Koordinaten dar, pxyz stellt ihn in xyz-Koordinaten dar!




GOOD CODERS...
OK, we've changed the matrix to its transpose. But that doesn't work either. And now?
geek & poke
Let's try the inverse
programmers_spot
... KNOW WHAT THEY'RE DOING

Orthogonale Matrizen und der Zusammenhang zu Rotationen

- Definition (Wdhg aus Mathe):

  eine Matrix R heißt orthogonal $\Leftrightarrow$ $RR^T = R^T R = I$

- Folgen:

  Die Spalten von R sind zueinander *orthonormal* (nicht nur orthogonal)

  $\det(R) = \pm 1$

  $R^{-1} = R^{\mathsf{T}}$

  R^{T} ist orthogonal

  $\|Rv\| = \|v\|$ (Längenerhaltung)

  $(Ru)\cdot(Rv) = u\cdot v$ (Winkelerhaltung)

  R_1, R_2 sind orthogonal $\Rightarrow$ $R_1 R_2$ ist orthogonal

Charakterisierung von (reinen) Rotationsmatrizen

- R ist orthogonal ⇔ R ist Rotationsmatrix und/oder Spiegelung
- R ist eine ordentliche Rotation ⇔ $RR^{\mathsf{T}} = I \ \wedge \ \det(R) = +1$
- Die Menge der orthogonalen Matrizen mit Determinante =1 bezeichnet man oft mit SO(3) (aka. die *Gruppe der 3D Rotationen*, mit der Multiplikation als Operator)

Zerlegung einer Rotationsmatrix (Konvertierung von Matrix in Achse+Winkel)

- Gegeben: Rotationsmatrix R

1. Aufgabe: den Rotationswinkel θ bestimmen

- Lösung: $1 + 2\cos\theta = \text{spur}(R)$
- Beweis:
  - Zu R gibt es eine Basiswechselmatrix U, so daß

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

  - Es gilt: $\text{spur}(R) = \text{spur}\left(URU^{-1}\right) = \text{spur}\left(R_x(\theta)\right) = 1 + 2\cos\theta$

    (Spezielle Eigenschaft der Spur)

2. Aufgabe: Rotationsachse **r** bestimmen

- Lösung: **r** ist der Eigenvektor zum Eigenwert 1 der Matrix R
- Beweis: alle Vektoren auf der Rotationsachse bleiben fest, d.h. $R\mathbf{r} = 1 \cdot \mathbf{r}$
- Berechnung der Eigenvektoren einer 3x3-Matrix:
  - Zur Erinnerung: zu jeder Matrix A gibt es eine adjungierte Matrix $A^{\#}$
  - Die Elemente $a_{ij}^{\#}$ der adjungierten Matrix sind definiert als

$$a_{ij}^{\#} = (-1)^{i+j} \det \begin{pmatrix} a_{11} & \cdots & a_{1i} & \cdots \\ \cdots & & & \\ a_{j1} & \cdots & a_{ji} & \cdots \\ \cdots & & & \end{pmatrix}$$

  - Es gilt: $AA^{\#} = \det(A) I$

- Falls $\det(A) = 0$, dann ist $AA^{\#} = 0 \cdot I = 0$
- Also gilt für jeden Spaltenvektor **v** aus $A^{\#}$, daß $A \cdot \mathbf{v} = 0$
- Gesucht: Eigenvektor **r** zum Eigenwert 1 von R, also ein **r**, so daß $(R - I) \cdot \mathbf{r} = 0$
- Das sind gerade die Spalten von $(R - I)^{\#}$

- Bemerkung: alle Spalten sind Vielfache voneinander
- Bemerkung: das funktioniert auch für größere Matrizen, ist aber nicht mehr effizient

Euler's Satz über Rotationen

Jede Rotation kann mit Hilfe 3-er Winkel um (fast) beliebige Achsen beschrieben werden.

Jede beliebige Rotation im Raum lässt sich als Rotation um eine bestimmte Achse mit einem bestimmten Winkel darstellen.

(Teil zwei haben wir gerade bewiesen. Teil eins führt zu Euler-Winkeln.)

Intermezzo: ein Film, in dem viele Interpolationen von Rotationen / Orientierungen vorkommen

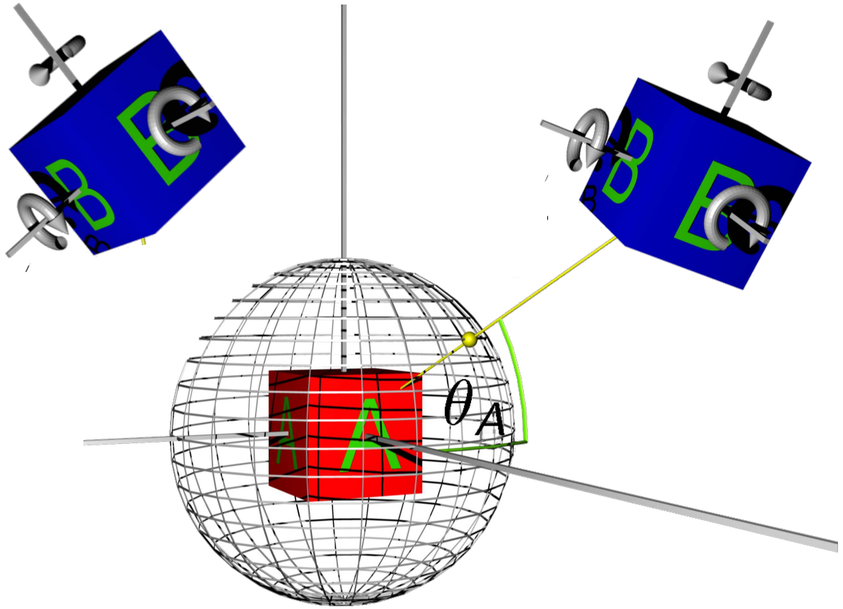
George Francis, Louis Kauffman,
Dan Sandin, Chris Hartman,
John Hart

http://www.evl.uic.edu/hypercomplex/

Interpolation von Orientierungen

- Definition: Orientierung
  = "Lage" (*attitude/pose*) eines Obj.s im Raum
  = Rotation aus dessen Null-Lage

Punkt	Orientierung
Vektor	Rotation
Null-Element = (0,0,0)	Null-Element = Einheitsmatrix



- Häufige Aufgabe:
  - Gegeben: zwei Orientierungen O_1 und O_2 für dasselbe Objekt
  - Gesucht: eine Funktion $O(t)$ zur Interpolation zwischen O_1 und O_2 , die ästhetisch ansprechend und effizient ist
- Frage: welche Repräsentation ist dafür gut geeignet?

Vorbetrachtungen

- Komplexe Zahlen kann man als Punkte in der Ebene betrachten
- Und als Rotation in der Ebene um den Ursprung !

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

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

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

Visualisierung: siehe Video auf der VL-Homepage unter *Videos*

- Wir können reelle Zahlen leicht invertieren: $x \cdot x^{-1} = 1$
- Auch für komplexe Zahlen können wir ein Inverses finden: $\frac{z \cdot z^*}{|z|^2} = 1$
- Gibt es etwas Analoges auch in "höheren Dimensionen"? $\longrightarrow$ Nein!
  - Z.B. eine "dreidimensionale" Verallgemeinerung von $\mathbb{C}$?
- Aber: im 4D klappt es wieder ... (fast)

Quaternionen

[Hamilton, 1843]

- Erweiterung der komplexen Zahlen:

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

- 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:

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

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.

- Bemerkung: manchmal ist es zweckmäßig, die Multiplikation zweier Quaternionen auch mit Hilfe einer Matrix-Multiplikation darzustellen

$$q_1 \cdot q_2 = \begin{pmatrix} w_1 & -a_1 & -b_1 & -c_1 \\ a_1 & w_1 & -c_1 & b_1 \\ b_1 & c_1 & w_1 & -a_1 \\ c_1 & -b_1 & a_1 & w_1 \end{pmatrix} q_2 = \begin{pmatrix} w_2 & -a_2 & -b_2 & -c_2 \\ a_2 & w_2 & c_2 & -b_2 \\ b_2 & -c_2 & w_2 & a_2 \\ c_2 & b_2 & -a_2 & w_2 \end{pmatrix} q_1$$

Als Spaltenvektor geschrieben!

Als Spaltenvektor!

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

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

$$\mathbf{v} \in \mathbb{R}^3 \;\mapsto\; q_v = (0, \mathbf{v}) \in \mathbb{H}$$

- 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

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

- Beobachtung: $|q| = 1$

- Zurückrechnen: $q = (w, a, b, c)$ sei gegeben, mit $|q| = 1$
  Dann ist

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

- Satz:

  Jedes Einheitsquaternion kann man in der Form $\left(\cos\frac{\varphi}{2}, \sin\frac{\varphi}{2}\mathbf{r}\right)$ 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 $\mathbf{v}$, wobei Winkel und Achse durch q bestimmt sind.

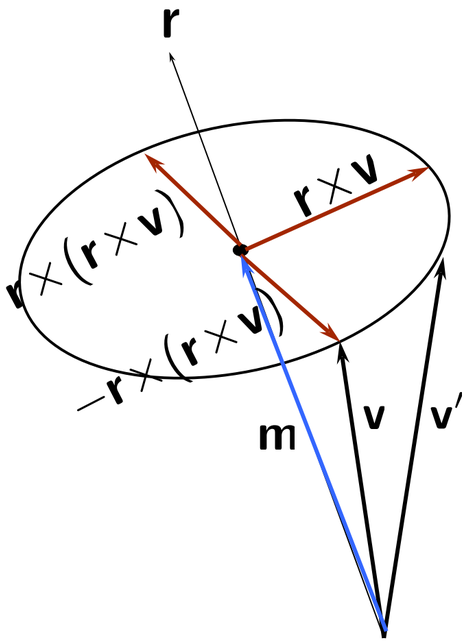
FYI

- Beweisskizze:

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

$$= \ldots \text{ (*)}$$

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



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

*) 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

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

- Caveat: die beiden Einheits-Quaternionen $q = (w, x, y, z)$ und $-q = (-w, -x, -y, -z)$ stellen die selbe Rotation dar!
  - M.a.W.: die 4-dim. Einheitskugel in $\mathbb{H}$ deckt die Gruppe aller Rotationen, SO(3), doppelt ab!

Lineare Interpolation von Quaternionen

- Gegeben: zwei Orientierungen q_1 , q_2
- Aufgabe: Orientierungen dazwischen interpolieren
- Einfachste Lösung: linear interpolieren

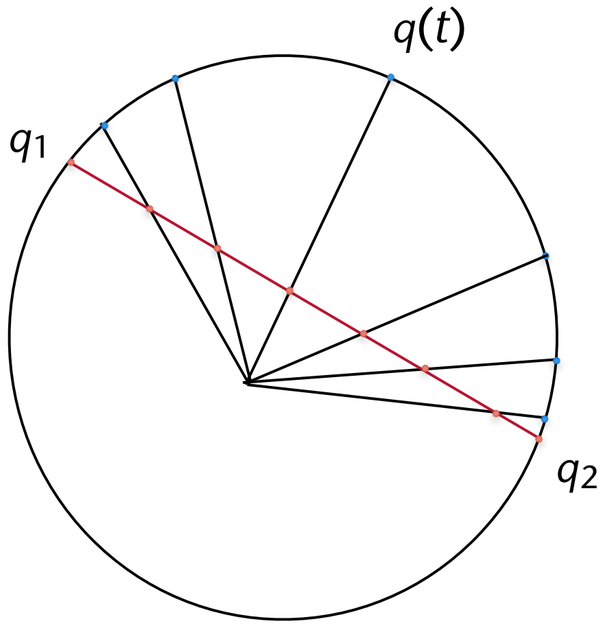
$$q(t) = (1-t)q_1 + tq_2 =: \text{lerp}(t; q_1, q_2)$$

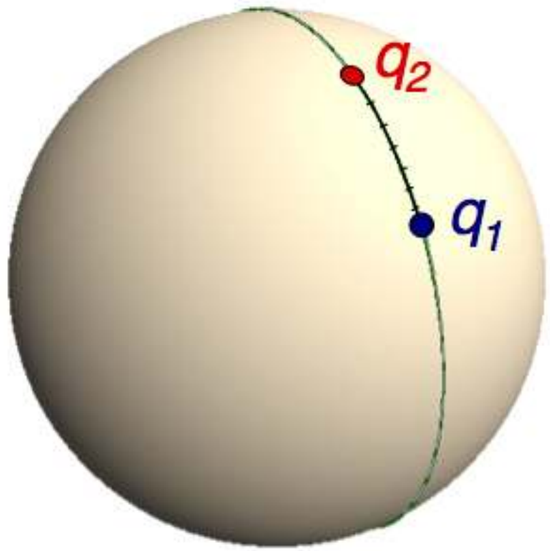
  - Wichtig: $q(t)$ hinterher immer normieren!

- Vorteil: Kein Gimbal Lock!

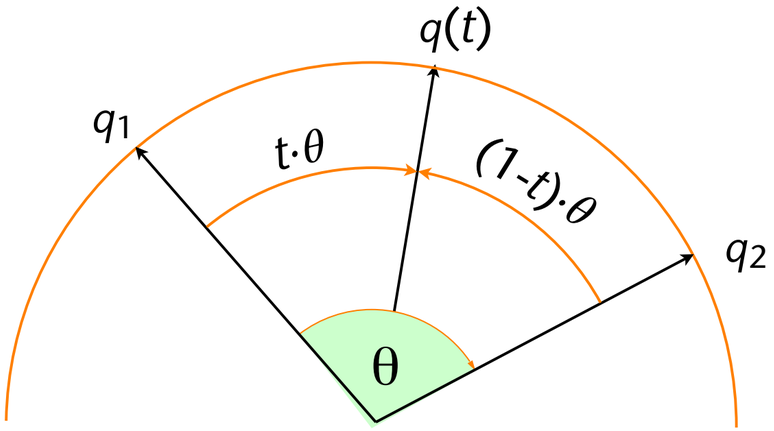
Sphärische lineare Interpolation

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





- Besser ist die sphärische lineare Interpolation *“slerp”*
  - Ansatz: interpoliere nicht die *Distanz* zwischen q_1 und q_2, sondern den *Winkel* dazwischen



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

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

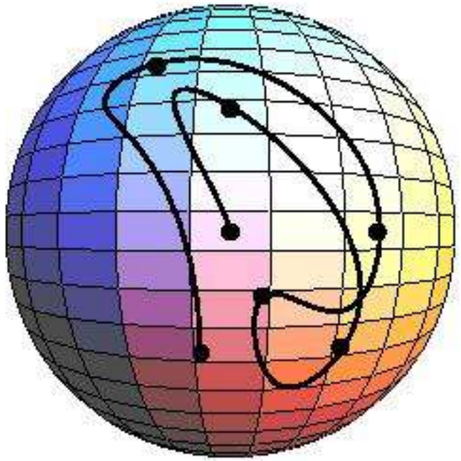
Nebenbemerkung (FYI)

- In der Praxis interpoliert man die Quaternionen mit höherem Grad, mit irgend einem Standard-Interpolationsverfahren, das parametrische Kurven liefert
  - Z.B. Bezier-Kurven (oder B-Splines)
  - Mit De Casteljau geht das sehr einfach:
    - Statt

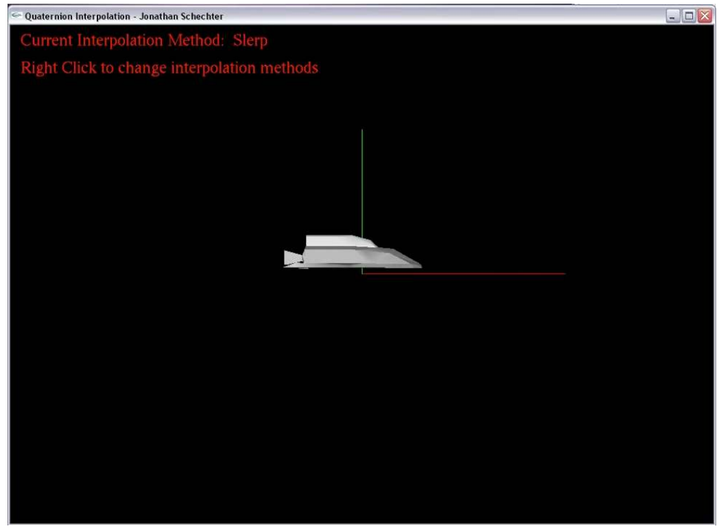
      $$Q_i = (1-t)P_i + tP_{i+1}$$

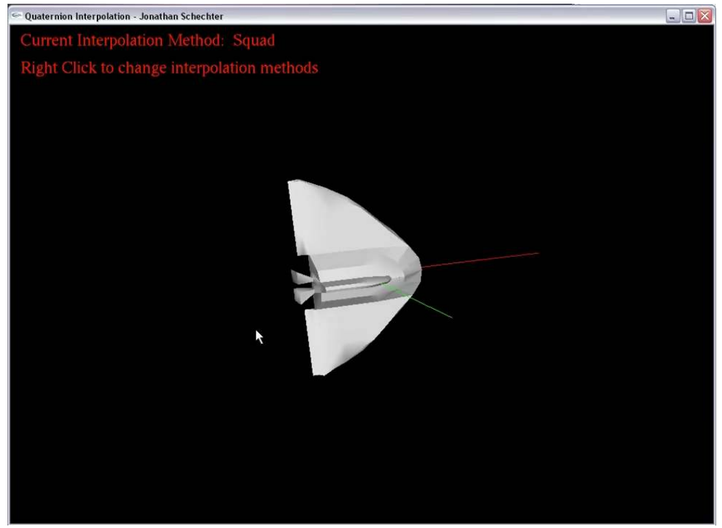
      berechne fortgesetzt

      $$Q_i = \text{slerp}(t; P_i, P_{i+1})$$



Lineare Interpolation mit slerp



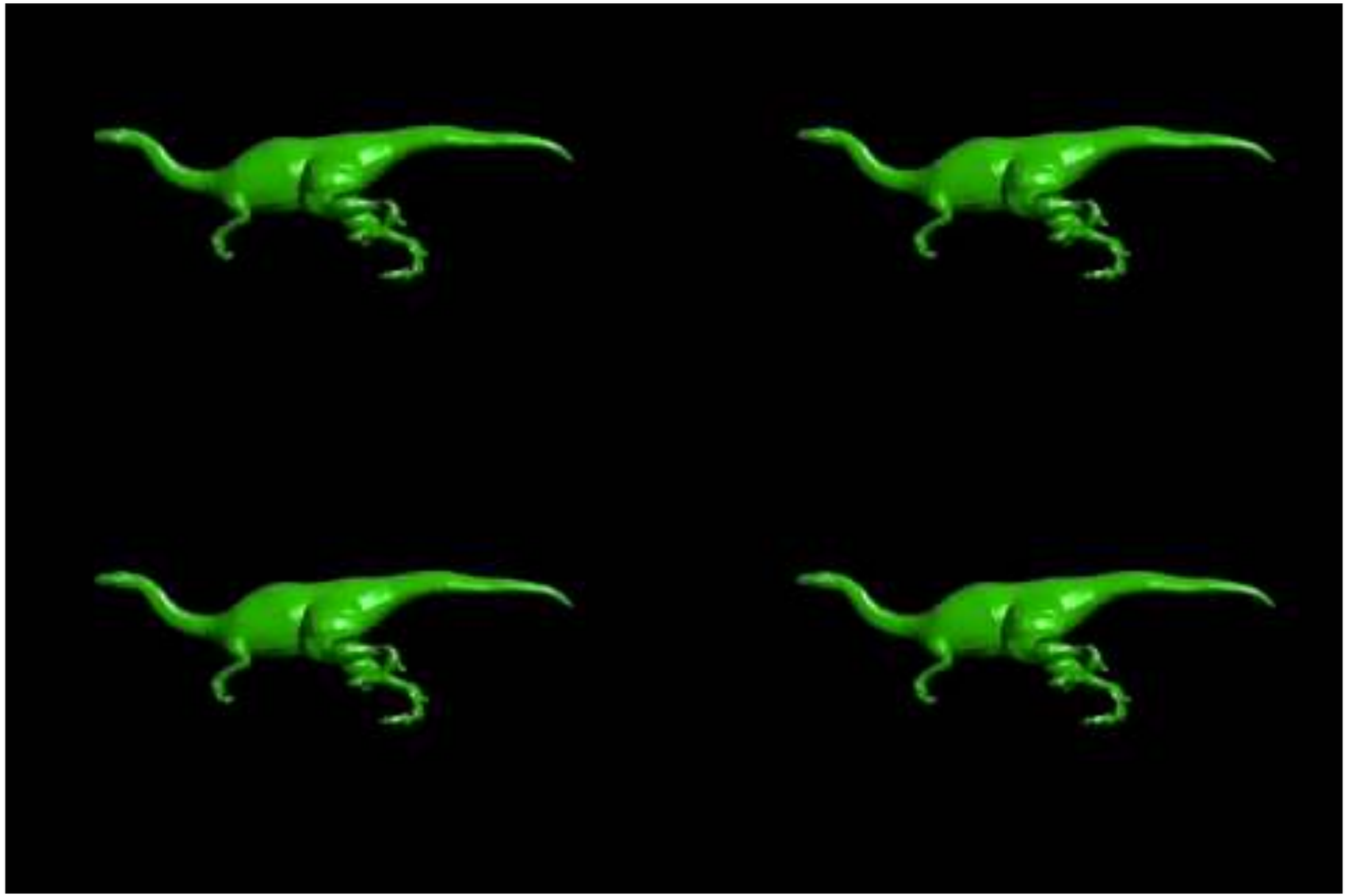


Kubische Interpolation mit slerp

Vergleich der verschiedenen linearen Interpolationsarten

Interpolation of Euler angles

Naïve interpolation of matrices

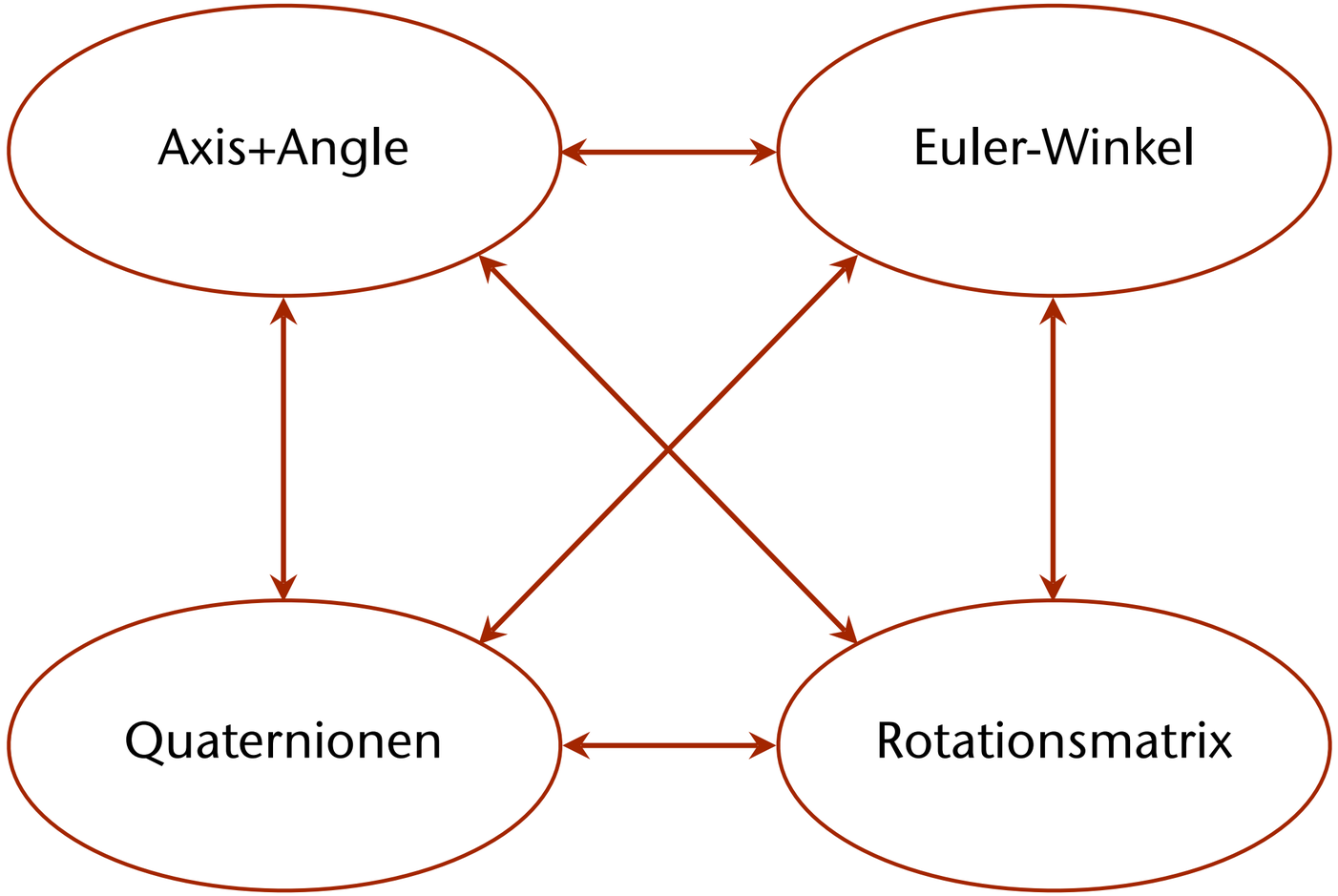


Slerp of quaternions

Lerp of quaternions (with normalization)

Gianluca Vatinno, Trinity College Dublin

Alle Darstellungen von Rotation lassen sich ineinander umrechnen



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

Vergleich der verschiedenen Arten zur Darstellung von Rotationen

Matrix	**Euler-Winkel**
Quaternionen	**Achse+Winkel**

Vergleich der verschiedenen Arten zur Darstellung von Rotationen

Matrix	**Euler-Winkel**
o Ubiquitär in OpenGL / Game Engines - 16 Floats (evtl. 12) + Vektor rotieren = 15 FLOPs + Einheitliche Repräsentation für alle affinen Transformationen - Rundungsfehler nach mehrfacher Konkatenation lassen sich nicht leicht korrigieren - Konkat. von Rotationen = 45 FLOPs	+ Intuitiv (zunächst) + 3 Floats - Gimbal lock - keine Algebra (Rechenop.) darauf ⟶ Konkatenation fkt. nicht direkt
Quaternionen	**Achse+Winkel**
+ Intuitiv + 4 Floats + Rundungsfehler nach mehrfacher Konkatenation lassen sich leicht korrigieren (einfach normieren) + Konkat. von Rotationen = 28 FLOPs - Vektor rotieren = 30 FLOPs	+ Sehr intuitiv + 3 Floats - keine Algebra (Rechenop.) darauf - Vektor rotieren = 40 FLOPs

Anwendung: virtueller Trackball

- Interaktionsaufgabe: ein Objekt um eine beliebige Achse rotieren
- Mit echtem Trackball ist es trivial

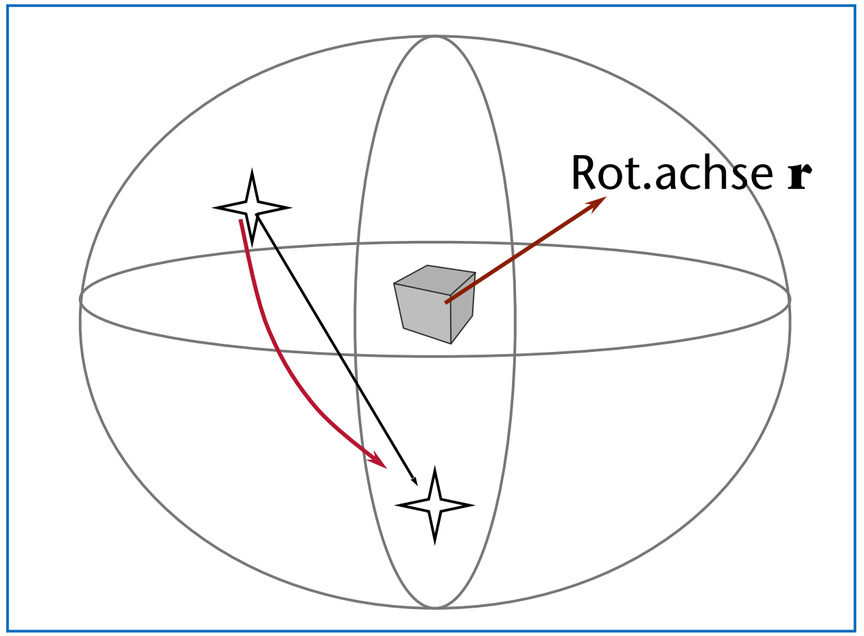
- Wie gibt man beliebige Rotationen mit der 2D-Maus ein?

Die Interaktionsmetapher

- Idee:
  - Lege gedachte (virtuelle) Kugel um das Objekt
  - Kugel kann um ihr Zentrum rotieren
  - Maus pickt Punkt auf Oberfläche, den man zieht
- Geg.: Startpunkt = (x_1,y_1), Endpunkt = (x_2,y_2) (in 2D!)
- Ges.: Rotationsachse **r** (in 3D), Rotationswinkel
- Berechnung:
  1. Bestimme 3D Punkte

$$\mathbf{p}_i = (x_i, y_i, z_i) \quad z_i = \sqrt{1-(x_i^2 + y_i^2)}$$

  2. Rotationsachse $\mathbf{r} = \mathbf{p}_1 \times \mathbf{p}_2$

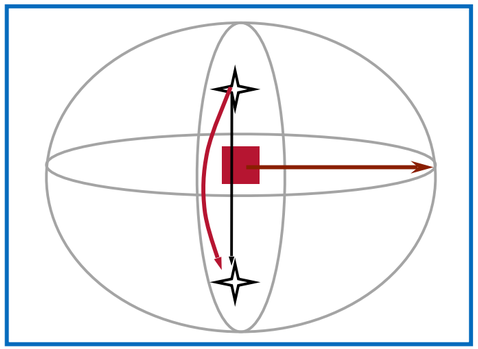


Gedachter Weg auf der Kugel = Segment des Großkreises

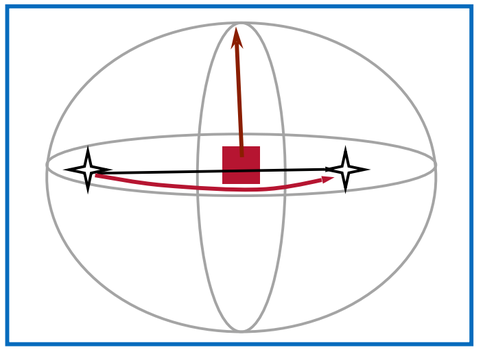
Weg der Maus im Fenster

Bemerkungen

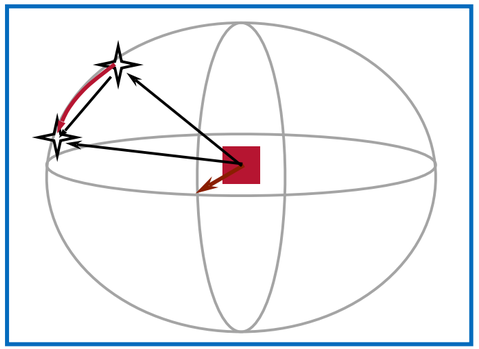
- Variante 1: $\mathbf{p}_1$ = erster Mausklick, $\mathbf{p}_2$ = aktuelle Mausposition: unintuitiv
- Variante 2: $\mathbf{p}_1$ = Mausposition vom vorigen Frame, $\mathbf{p}_2$ = aktuelle Mausposition: intuitiv, aber Rotation exakt um Z-Achse unmöglich



X



Y



Z

- Verbesserungen:
  - "Spinning trackball" vermeidet häufiges Nachfassen
  - "Snapping" für exaktes Rotieren um eine Koord.achse
  - Was macht man, wenn p2 die Ellipse verlässt? → andere 3D-Fläche verwenden, die an der Silhouette stetig anschließt

Bemerkungen

- Die Rotationsachse **r** ist zunächst im Kamera-Koordinatensystem definiert!
  - Sie muss aber nach Weltkoordinaten oder Obj.koordinaten zurückgerechnet werden (je nach dem, ob diese Rotation als letztes oder als erstes auf das Obj angewendet werden soll)
- Achtung: bei Variante 2 (inkrementeller Trackball) werden sehr viele Rotationen mit sehr kleinen Winkeln akkumuliert (pro Frame eine kleine Rotation) $\longrightarrow$ achte auf numerische Robustheit und Drift

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 **p** der Ortsvektor zum Punkt P ist.

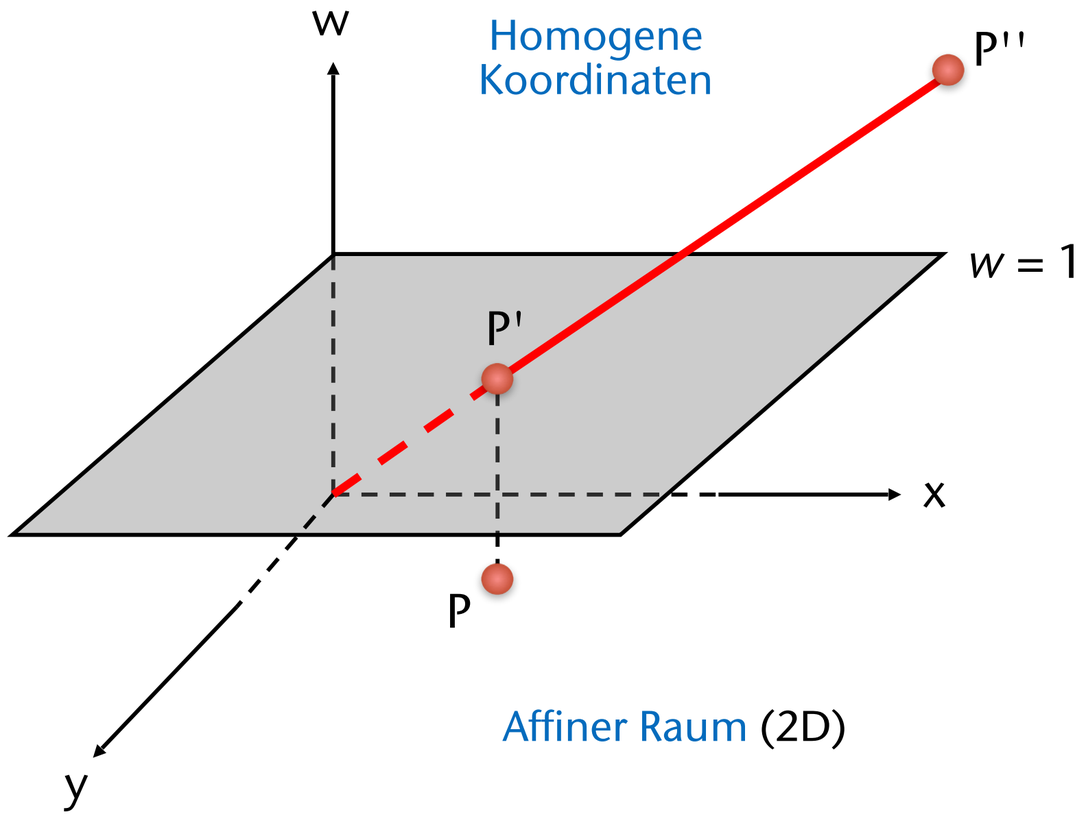
- Problem: affine Transformationen können **nicht** in Form einer 3x3-Matrix dargestellt werden (wegen der Translation)

Lösung: Homogene Koordinaten im 3D

- "Trick" (Homogenisierung):
  - Bette die Räume der 3D-Punkte und 3D-Vektoren in den 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
  - Achtung: dabei ist $w \neq 0$ und $w \neq 1$ *erlaubt* und kann vorkommen!
  - Nicht erlaubt: Wechsel von w=0 zu w≠0 und umgekehrt!
- Rück-Projektion:
  - Der homogene 4D-Vektor $\mathbf{p} = (x, y, z, w) \quad w \neq 0$
    beschreibt den 3D-*Punkt* an der Stelle $P = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$
  - Der 4D-Vektor $\mathbf{v} = (v_x, v_y, v_z, 0)$
    beschreibt den 3D-*Vektor* $\mathbf{v} = (v_x, v_y, v_z)$

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.: ein 3D-Vektor (*x*, *y*, *w*) beschreibt ...
  - ... den 2D-Punkt (*x*/*w*, *y*/*w*) für $w \neq 0$
  - ... den 2D-Vektor (*x*, *y*) für $w = 0$



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

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

Caveat

- Achtung: im Allgemeinen darf man homogene *Punkte* nicht einfach 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:

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

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:

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

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":

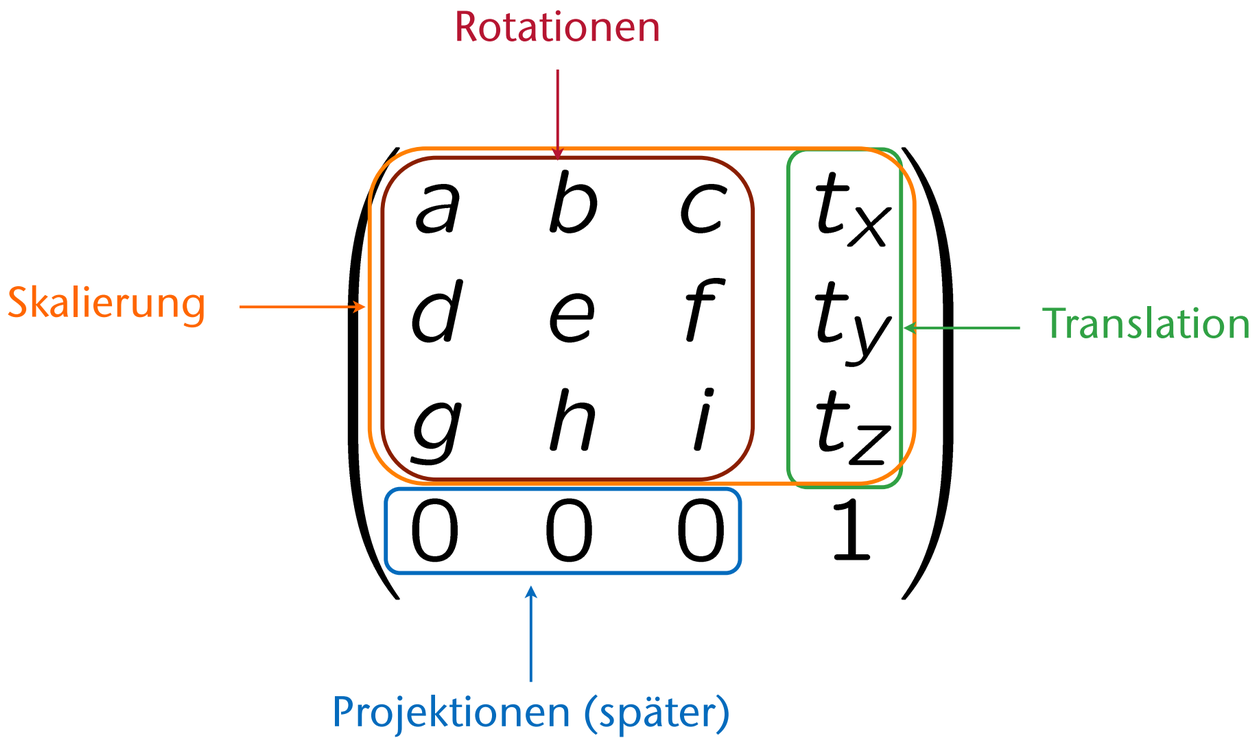
$$P' = (RT)P = MP \cong R(P + \mathbf{t}) = R_{3x3}P + R_{3x3}\mathbf{t}$$

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

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

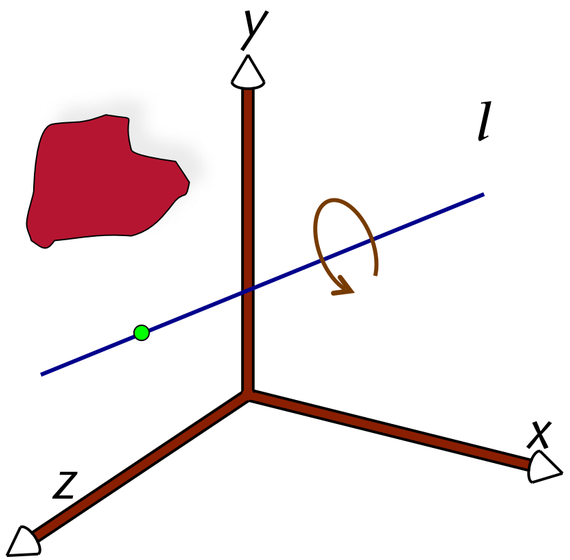
Zur Anatomie einer Matrix

- Allgemeiner Aufbau (vereinfacht!):



Rotation um eine *beliebige* Achse im Raum

- Man möchte mit θ um die Gerade l rotieren, Gerade geht durch den *beliebigen* Punkt p

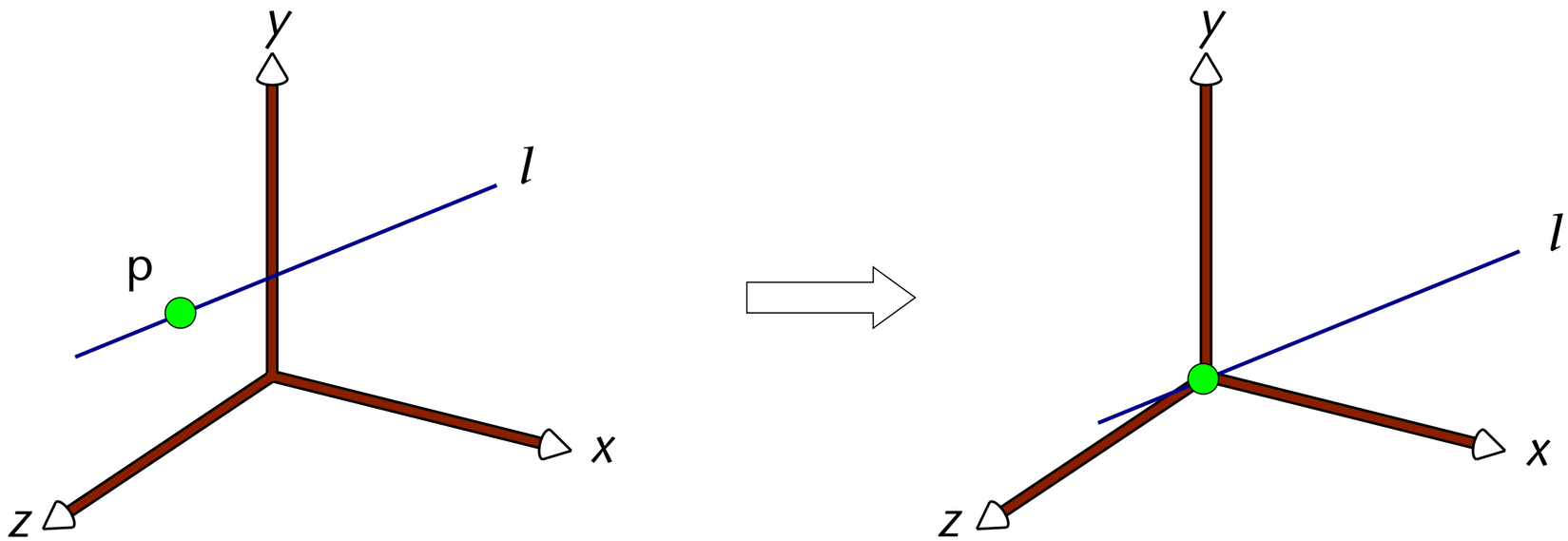


- Gesucht: eine Matrix *M*, die diese Transformation enthält
- Wir wissen, wie man um eine Koordinatenachse rotiert
- Somit müssen wir die Szene in eine Situation transformieren, mit der wir umgehen können

Grundidee

1. Verschiebe einen Punkt der Gerade 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 Winkel 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

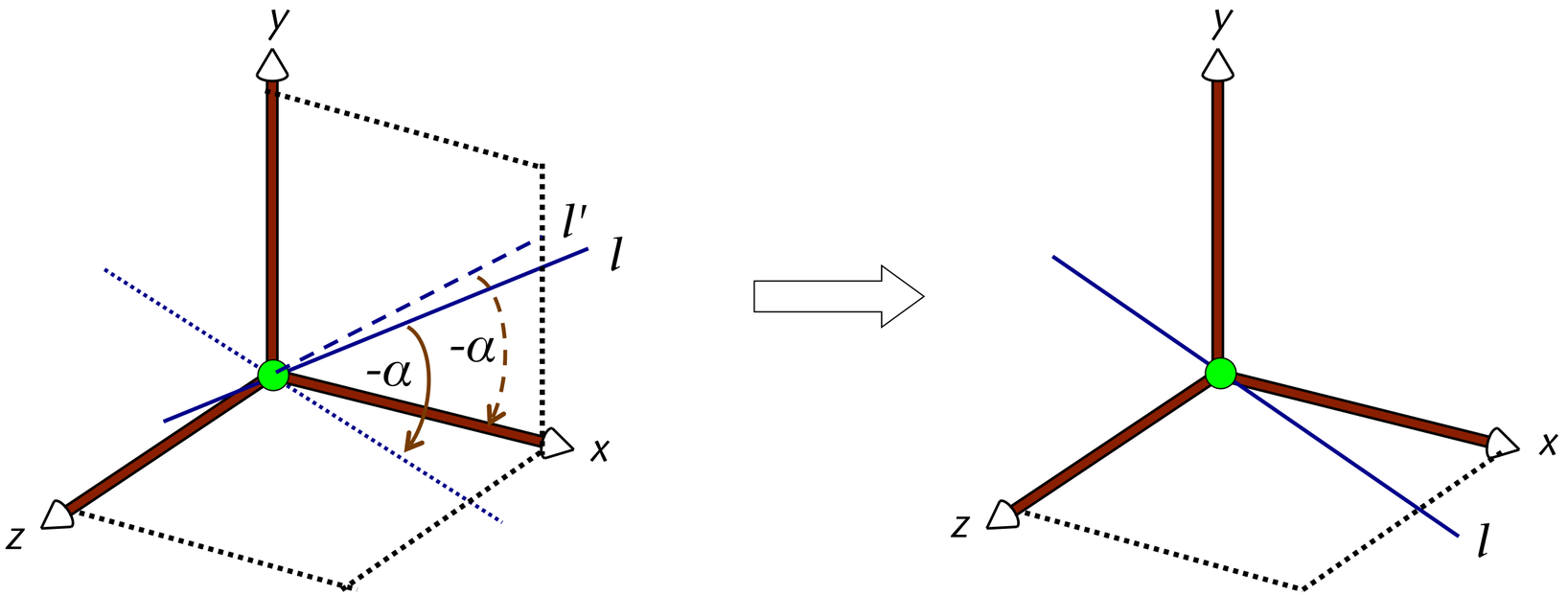
- Verschiebe Gerade, so daß ein Punkt von l im Ursprung liegt:



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

Schritt 2

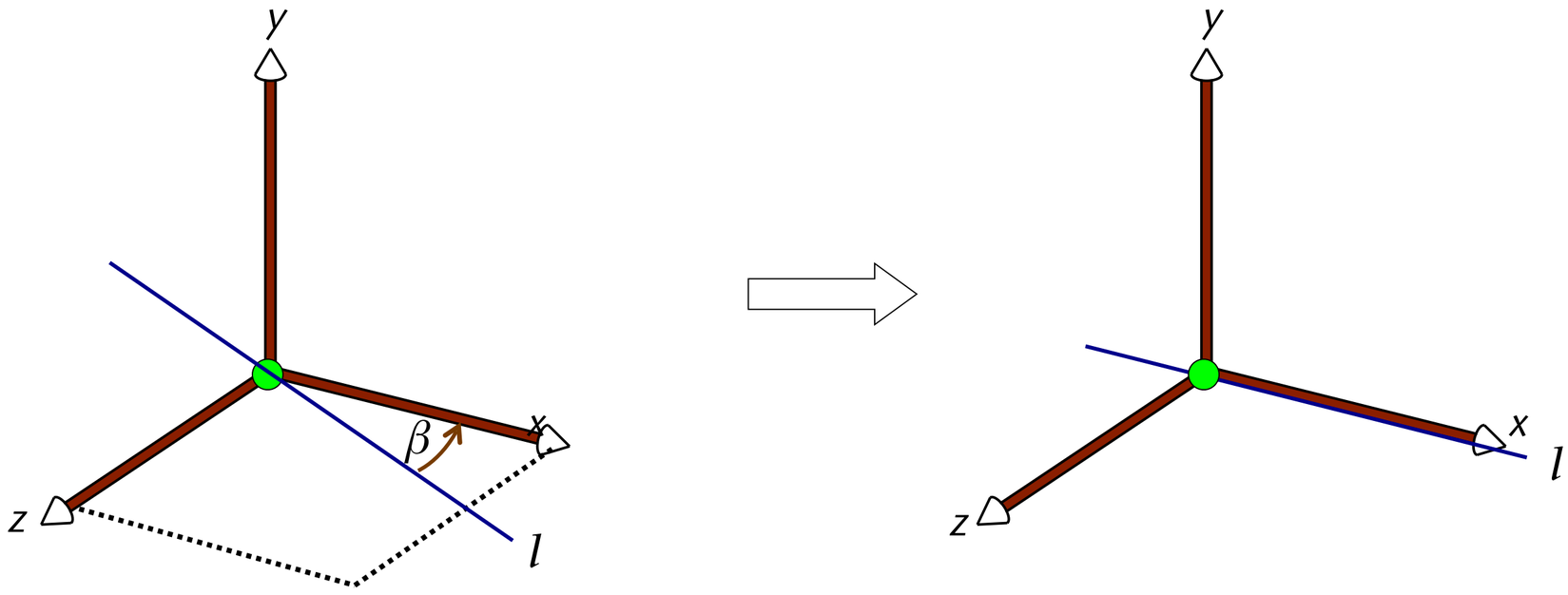
- Rotiere mit -α um die z-Achse, so daß l in der xz-Ebene liegt



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

Schritt 3

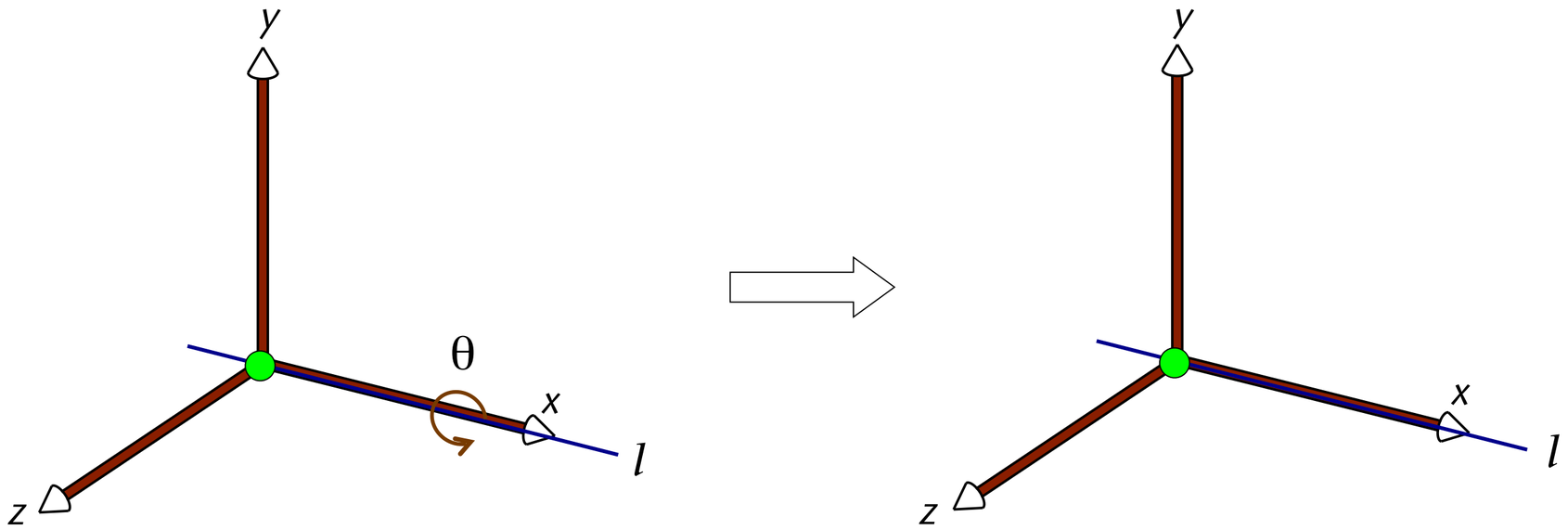
- Rotiere mit β um die *y*-Achse damit Gerade auf der *x*-Achse liegt



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

Schritt 4

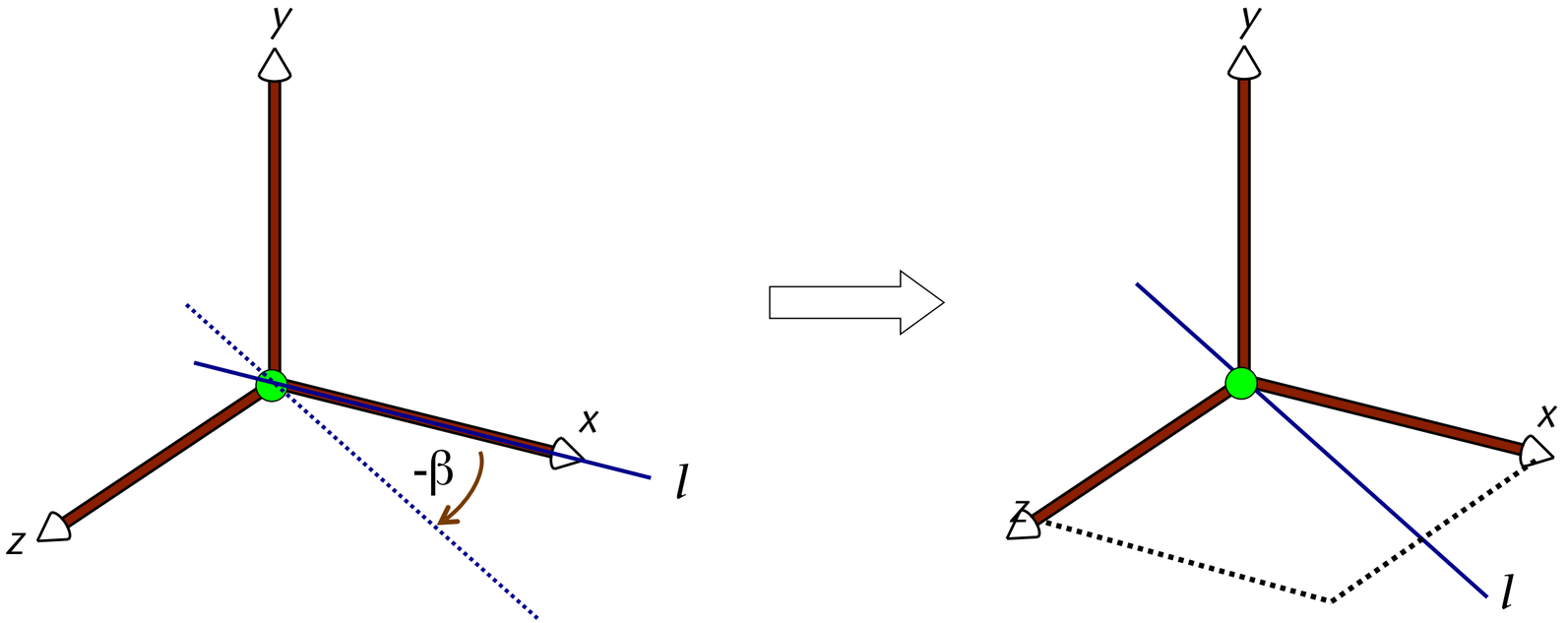
- Durchführen der gewünschten Rotation (rotiere mit θ um x-Achse)



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

Schritt 5

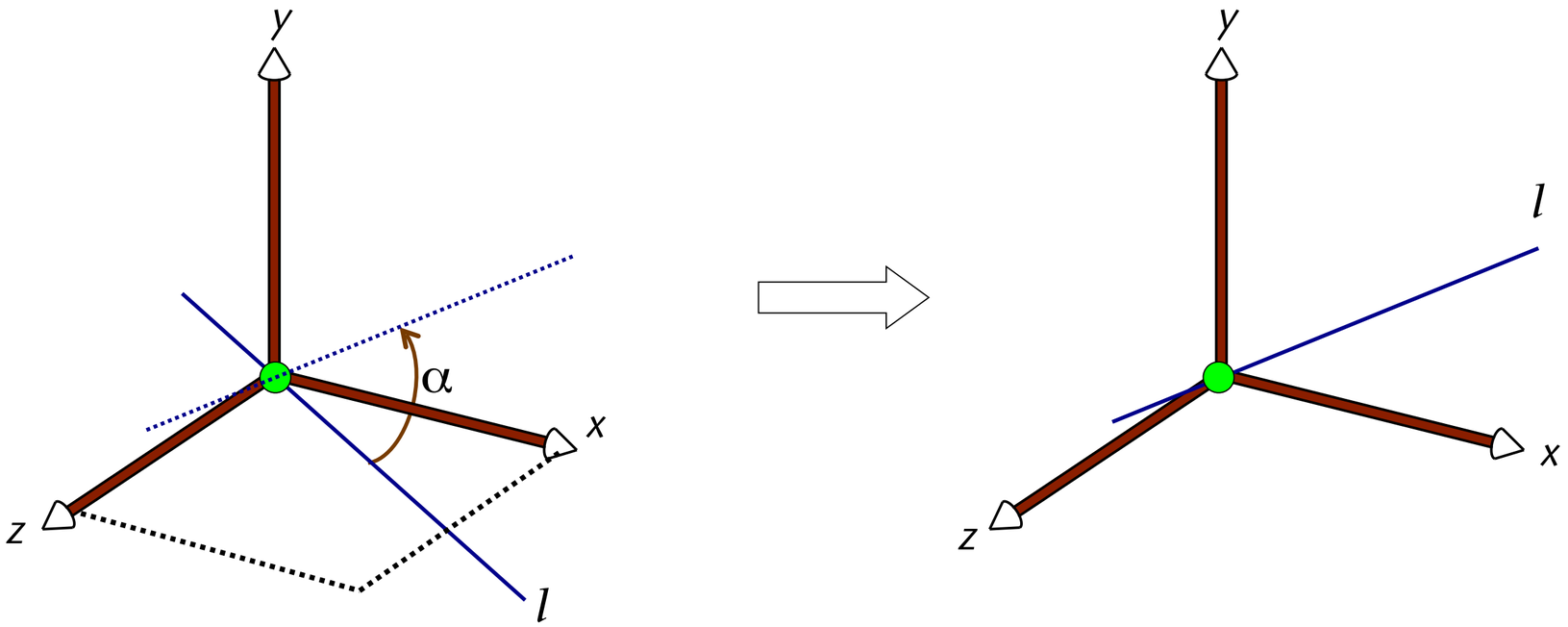
- Invertiere Rotation von l aus Schritt 3: rotiere mit -β um die y-Achse



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

Schritt 6

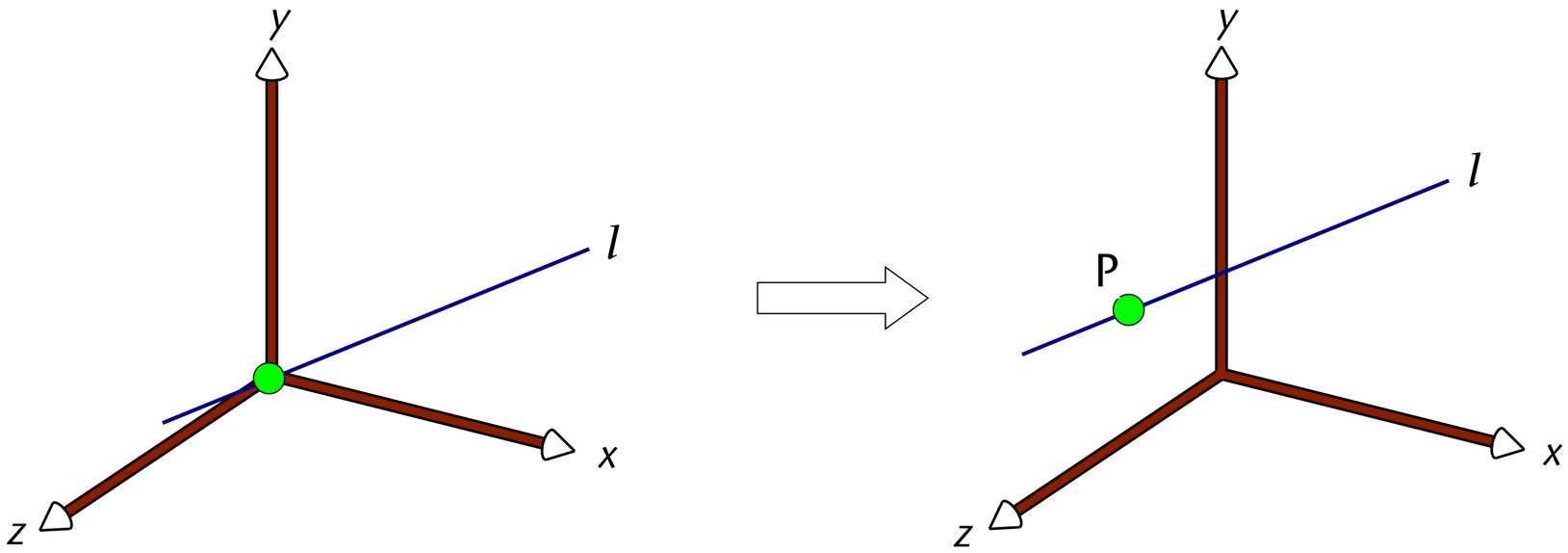
- Invertiere Rotation aus Schritt 2: rotiere mit α um z-Achse



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

Schritt 7

- Invertiere die Translation aus Schritt 1



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

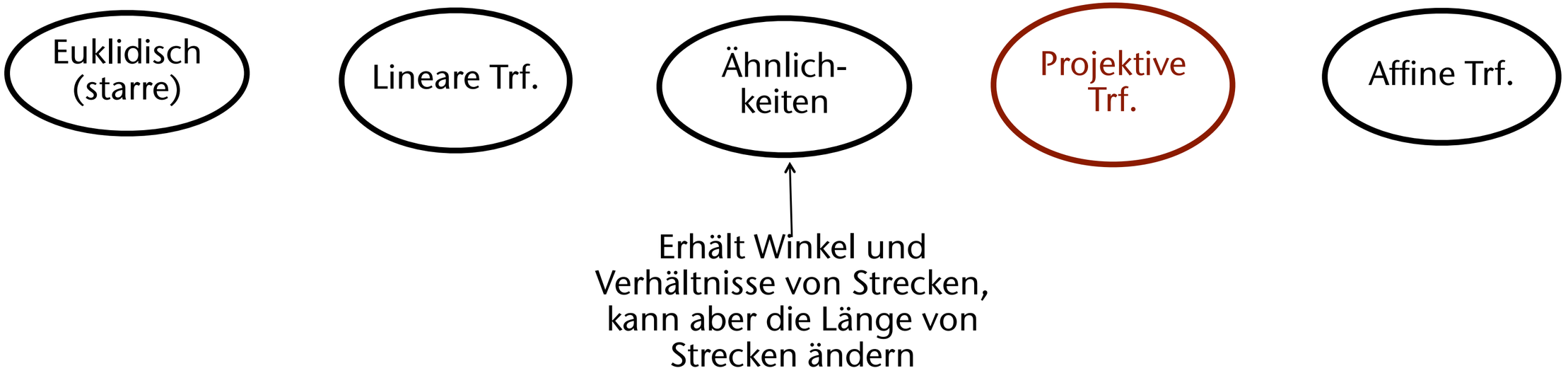
Zusammenfassung

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

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

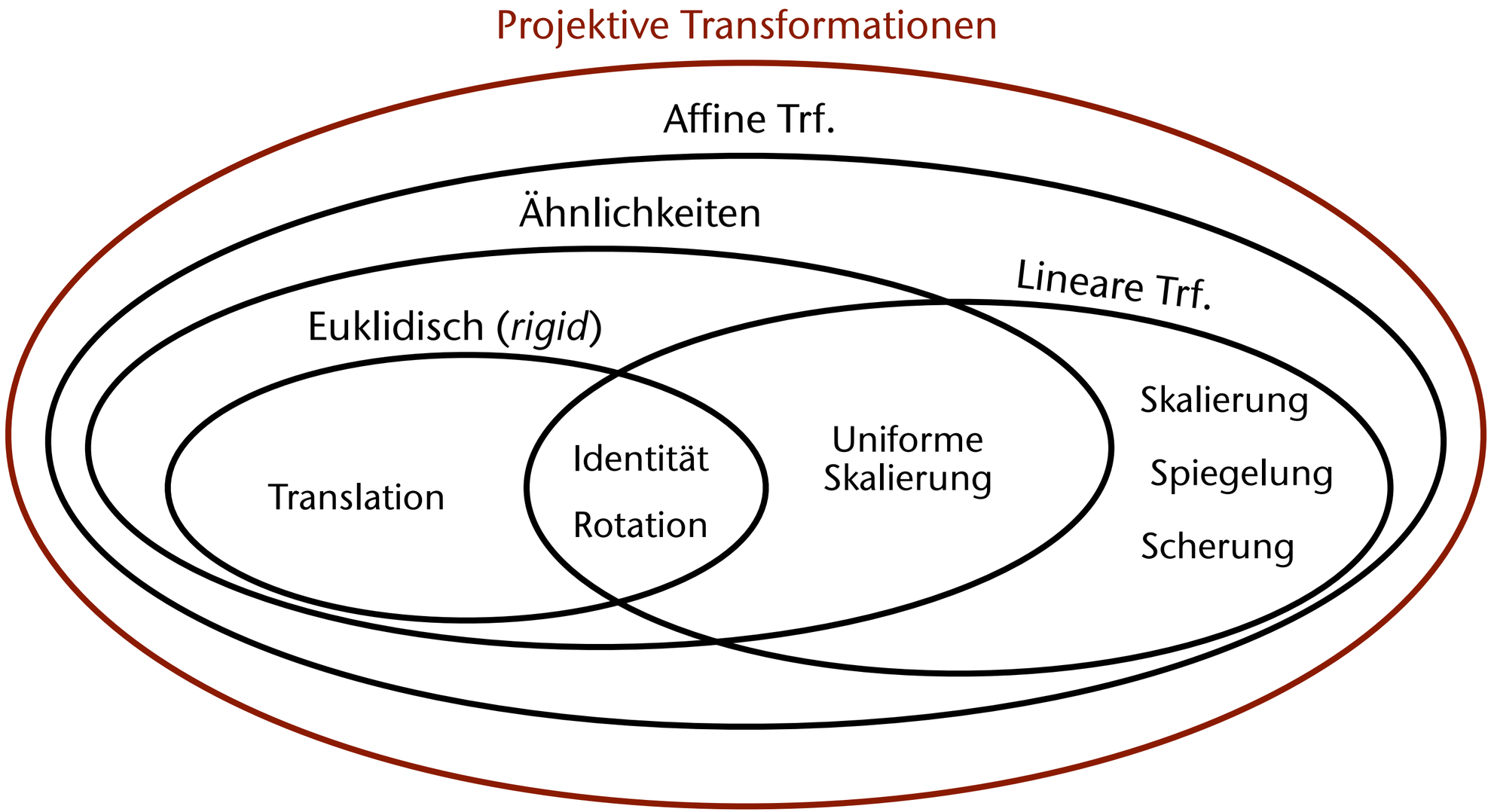
  - (Es gibt auch andere Varianten)
- Hat man diese Matrix, so wendet man diese auf jeden Vertex des Objektes an, was den Effekt der Rotation dieses Objektes um die vorgegebene Achse hat
  - Das überläßt man natürlich dem Vertex-Shader

Klassifikation aller Transformationen



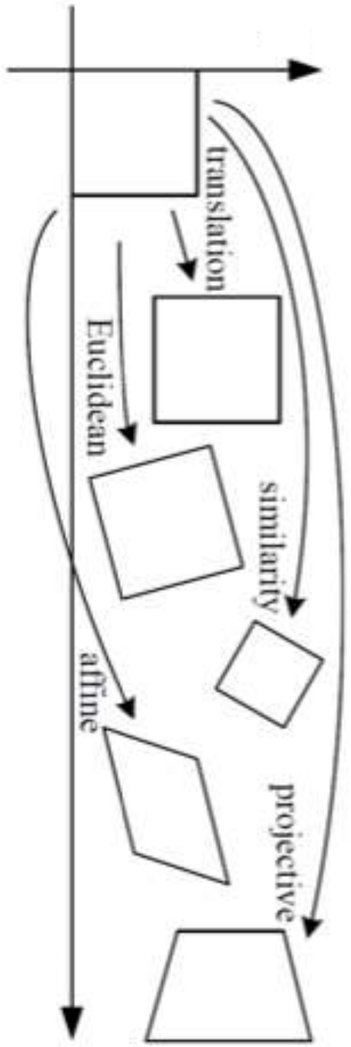
Translation | Identität | Rotation | Uniforme Skalierung | Skalierung | Spiegelung | Scherung

Klassifikation aller Transformationen



Eine Hierarchie von Transformationen (hier in 2D)

Transformation	Matrix	# DoF	Preserves	Icon
translation	$\left[\begin{array}{c\|c} \boldsymbol{I} & \boldsymbol{t} \end{array}\right]_{2\times3}$	2	orientation	
rigid (Euclidean)	$\left[\begin{array}{c\|c} \boldsymbol{R} & \boldsymbol{t} \end{array}\right]_{2\times3}$	3	lengths	
similarity	$\left[\begin{array}{c\|c} s\boldsymbol{R} & \boldsymbol{t} \end{array}\right]_{2\times3}$	4	angles	
affine	$\left[\boldsymbol{A}\right]_{2\times3}$	6	parallelism	
projective	$\left[\tilde{\boldsymbol{H}}\right]_{3\times3}$	8	straight lines	



Matrizen in OpenGL

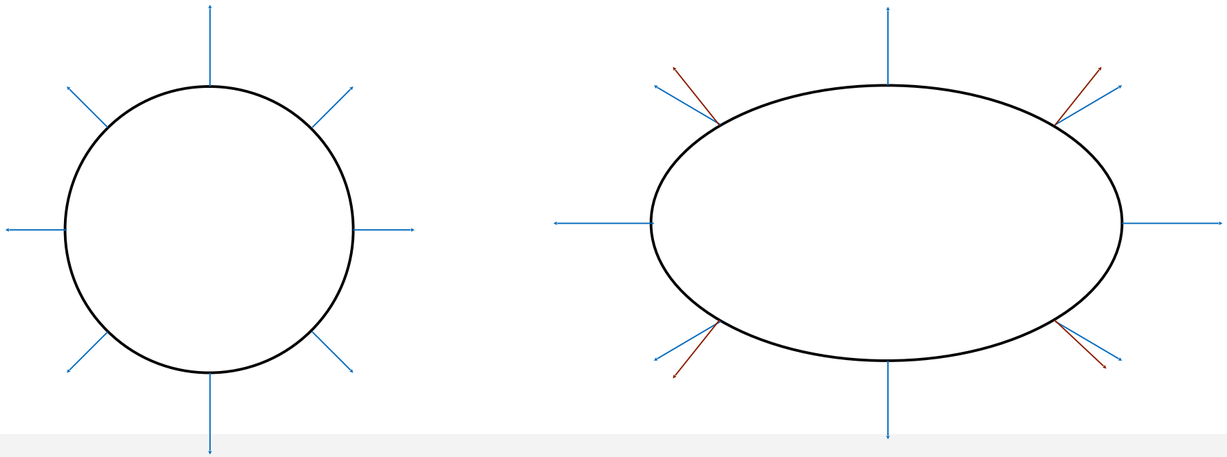
- Achtung: Matrizen werden **spaltenweise** im Speicher abgelegt, *nicht* — wie in C üblich — zeilenweise!
  - Das nennt sich "*column-major order*" (der Standard, auch in C, ist *row-major order*)

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

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

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:



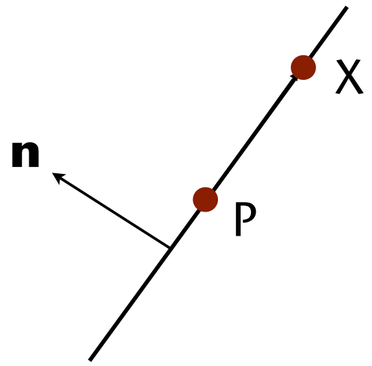
Beweis

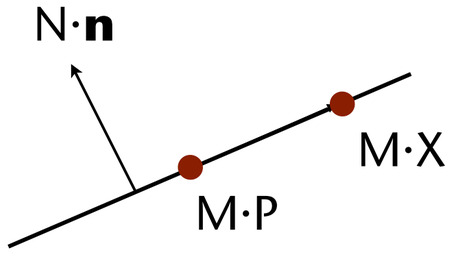
- Wir wissen: $(X - P)^{\mathsf{T}}\mathbf{n} = 0$
- Gesucht ist N, so daß:

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

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

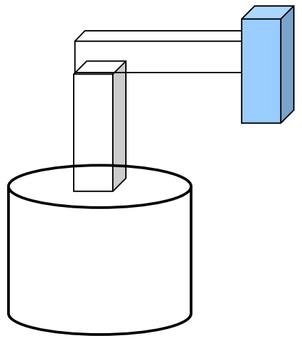
$$(X - P)^{\mathsf{T}} \cdot M^{\mathsf{T}} (M^{\mathsf{T}})^{-1} \cdot \mathbf{n} = (X - P)^{\mathsf{T}} \cdot I \cdot \mathbf{n} = 0$$



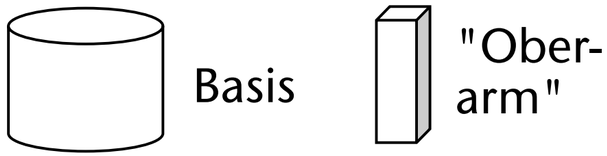


Relative und hierarchische Transformationen

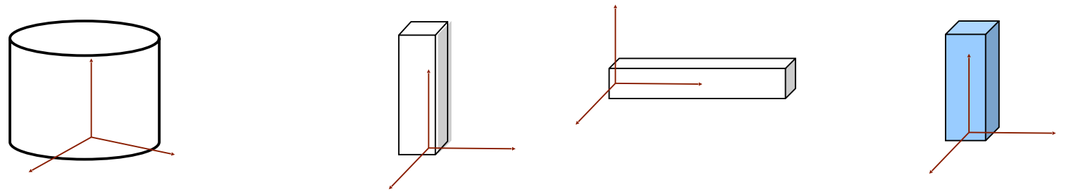
- Eine Konkatenierung von Transformationen kann man auch als eine Folge von (voneinander abhängigen) Koordinatensystemen ansehen



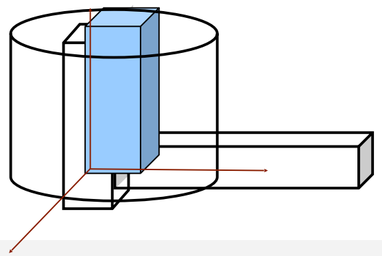
- Beispiel: Roboter
  - Besteht aus diesen Einzelteilen



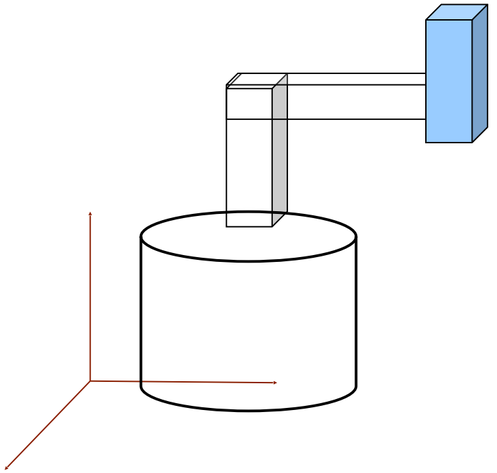
  - Jedes Teil wurde in seinem eigenen Koordinatensystem spezifiziert (als Array von Punkten) → heißt Objektkoordinatensystem



  - 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

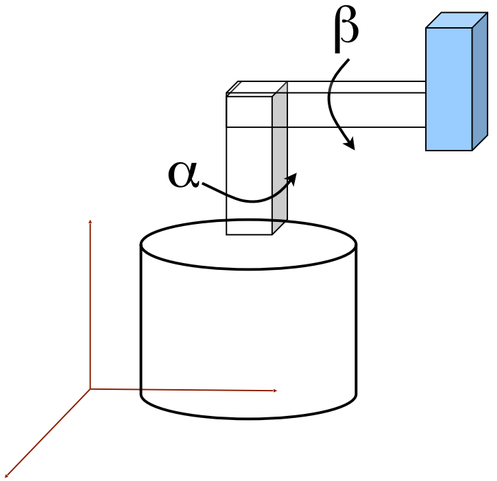
Ann.: Höhe der Basis ist 10

Ann.: Höhe des Oberarms ist 5

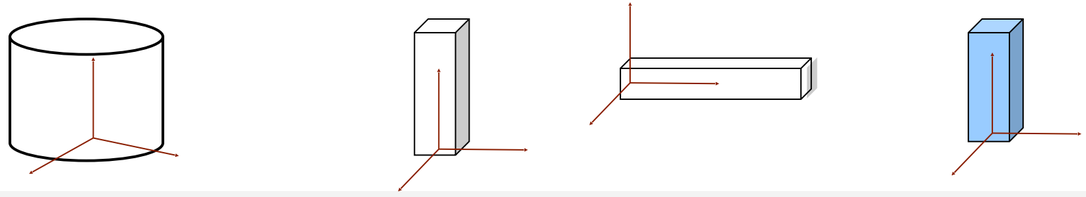
- Natürlich macht man es ungefähr so:

```
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 …
```

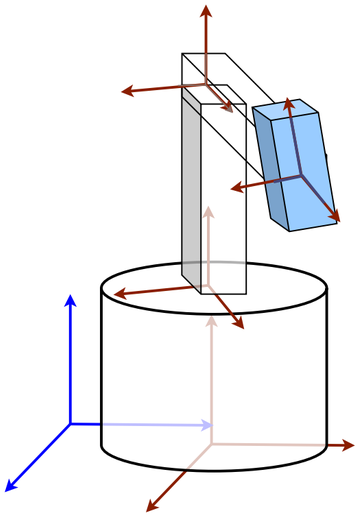
Hypothetische Funktionen, die eine Transformation *RELATIV* zur bis dahin gültigen Transformation setzen



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)

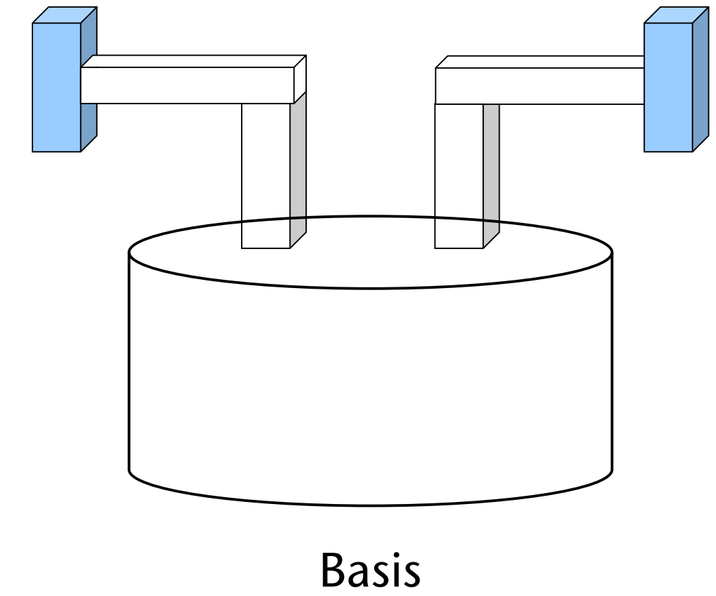
$$p' = M_n \cdot \ldots \cdot M_2 \cdot M_1 \cdot p$$

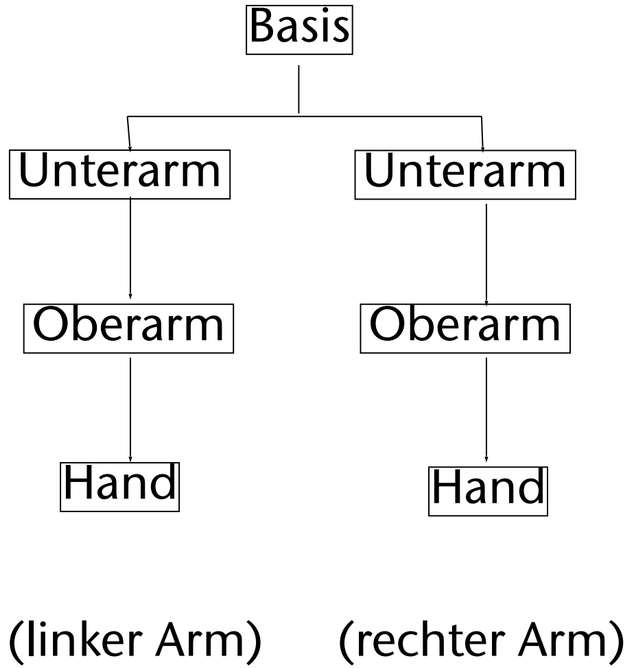
Vertex in Modell-Koordinaten (vorderes Ende der Pipeline)

Reihenfolge der Ausführung

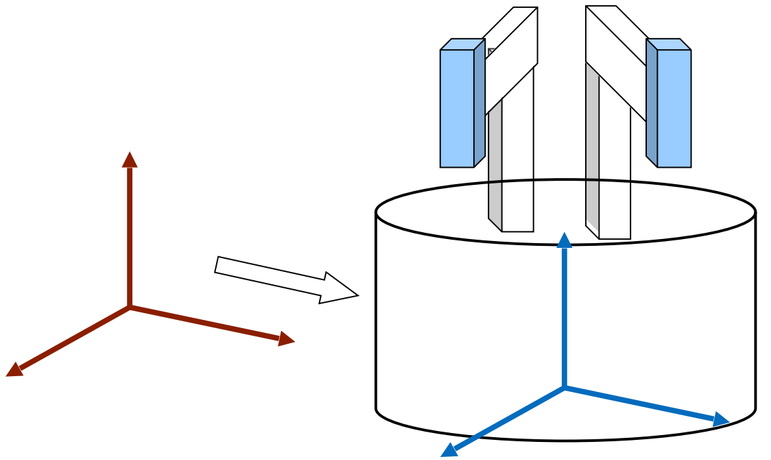
- Bemerkung: da Vertices von rechts an die Matrix (Matrizen) multipliziert werden, *muss* M_n die zuerst im Programmablauf gesetzte Transformation sein
- Definition:
  - Model-to-World-Matrix $= M_n \cdot \ldots \cdot M_2 \cdot M_1$
    $=$ Konkatenation aller aktuell gültigen, relativen Transf.en

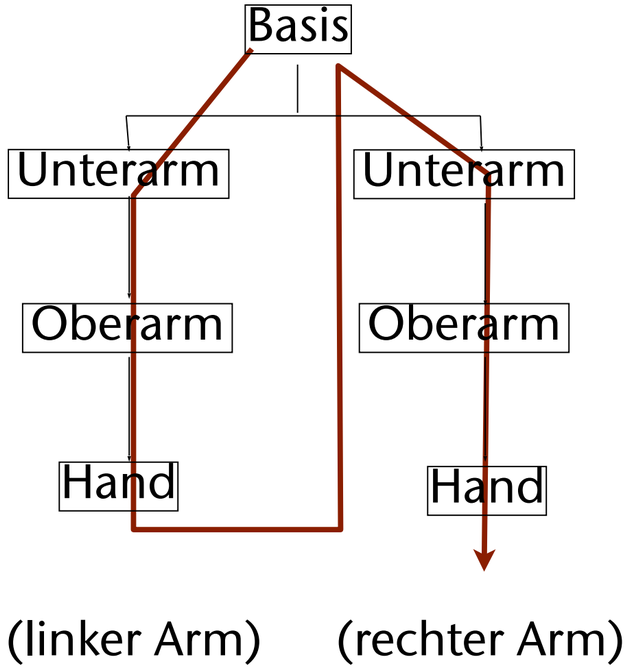
- Ein etwas komplizierteres Beispiel:





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






```
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
```

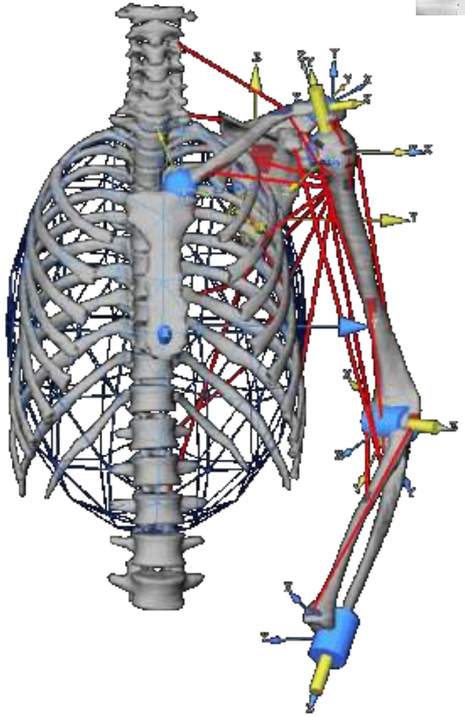
An dieser Stelle die aktuelle Model-to-World auf den Stack pushen
Z.B.: matrixStack.push(M);

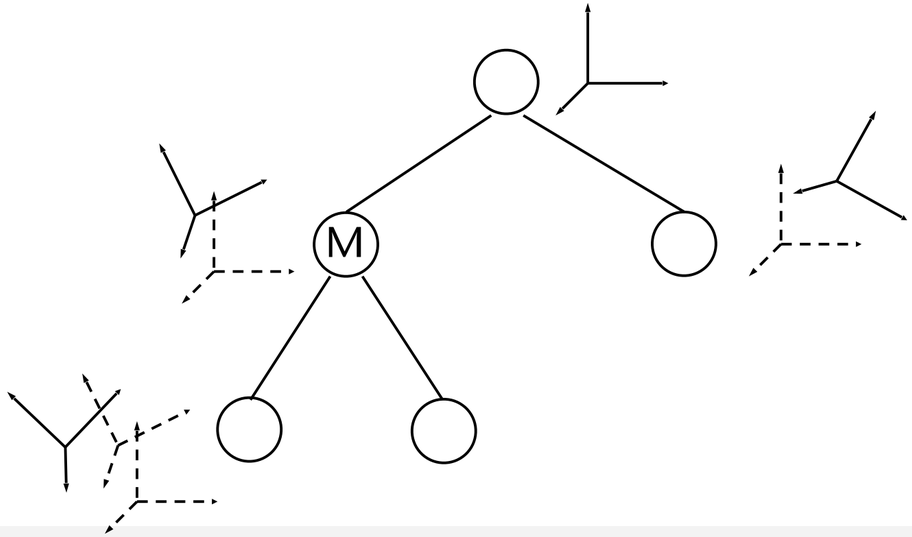
An dieser Stelle die oberste Matrix vom Stack pop-en und in die Model-to-World schreiben
Z.B.: M = matrixStack.pop();

Der Szenengraph

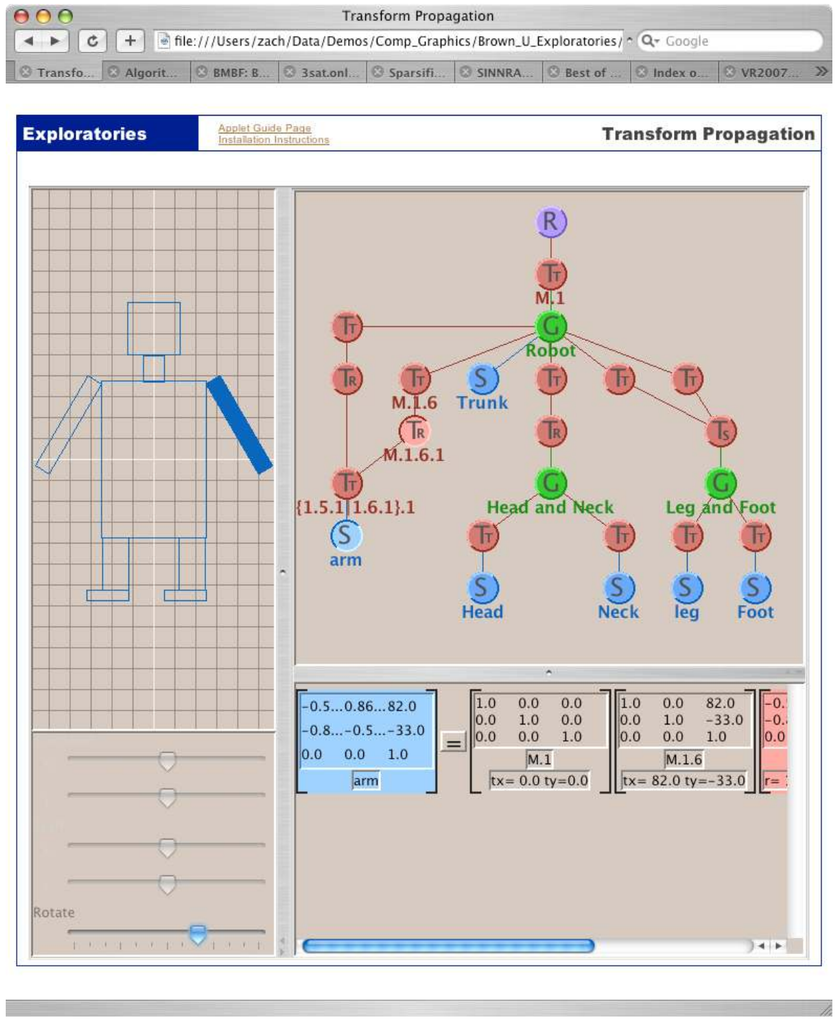
- Durch die relativen Transformationen ergibt sich eine Abhängigkeit der Objekte
- Der so definierte Baum heißt Szenengraph
- Knoten =
  - Transformationsknoten (speichern relative Traf.)
  - Geometrieknoten (i.A. Blätter)
- Aktion am Traf.-Knoten während Scenegraph-Traversal:

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





Demo zum Szenengraph

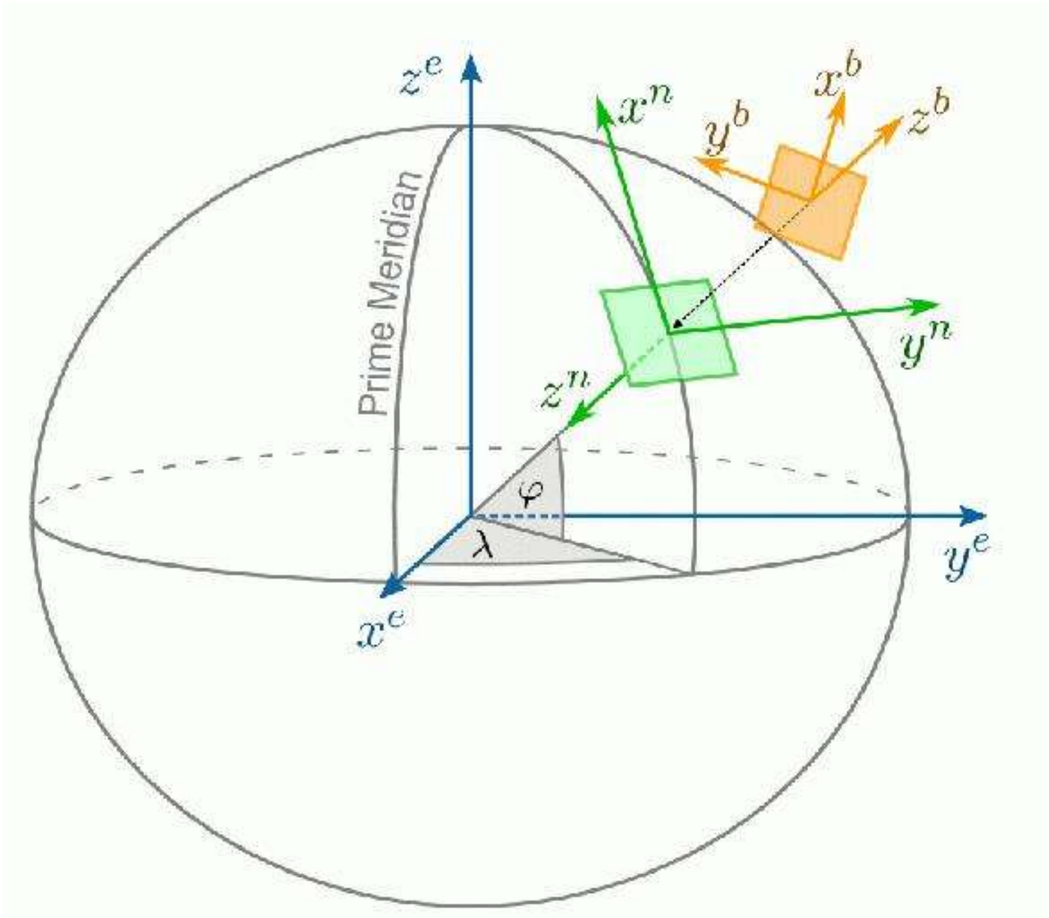


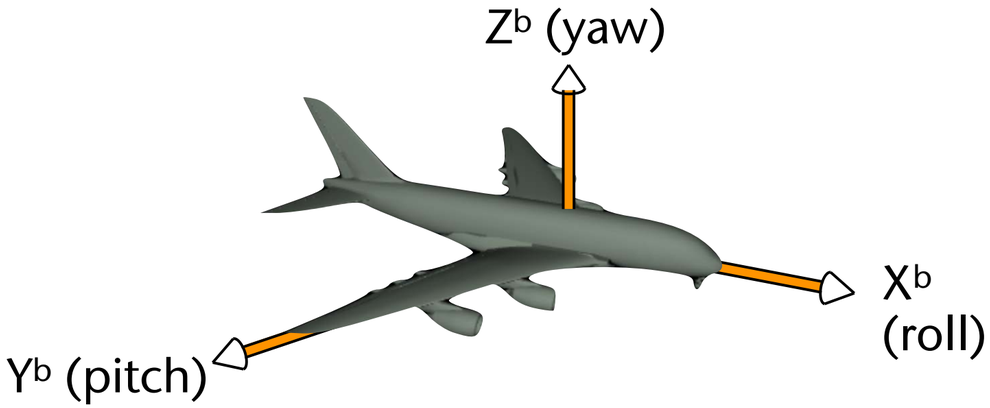
http://graphics.cs.brown.edu/research/exploratory/freeSoftware ⟶ Complete Catalog ⟶ Transformation Propagation

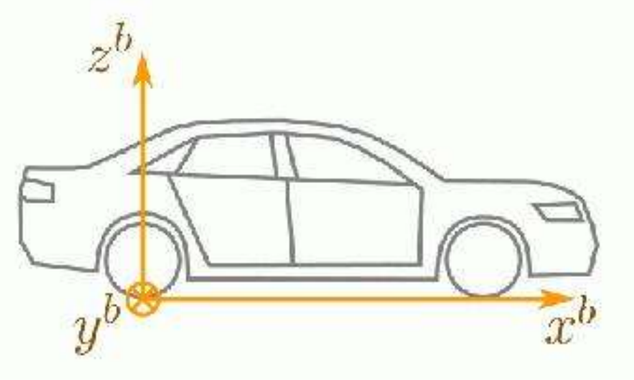
Exkurs: Koordinatensysteme in der Luftfahrt/Raumfahrt

FYI

- Body Frame, Navigation Frame, Earth-Fixed Frame

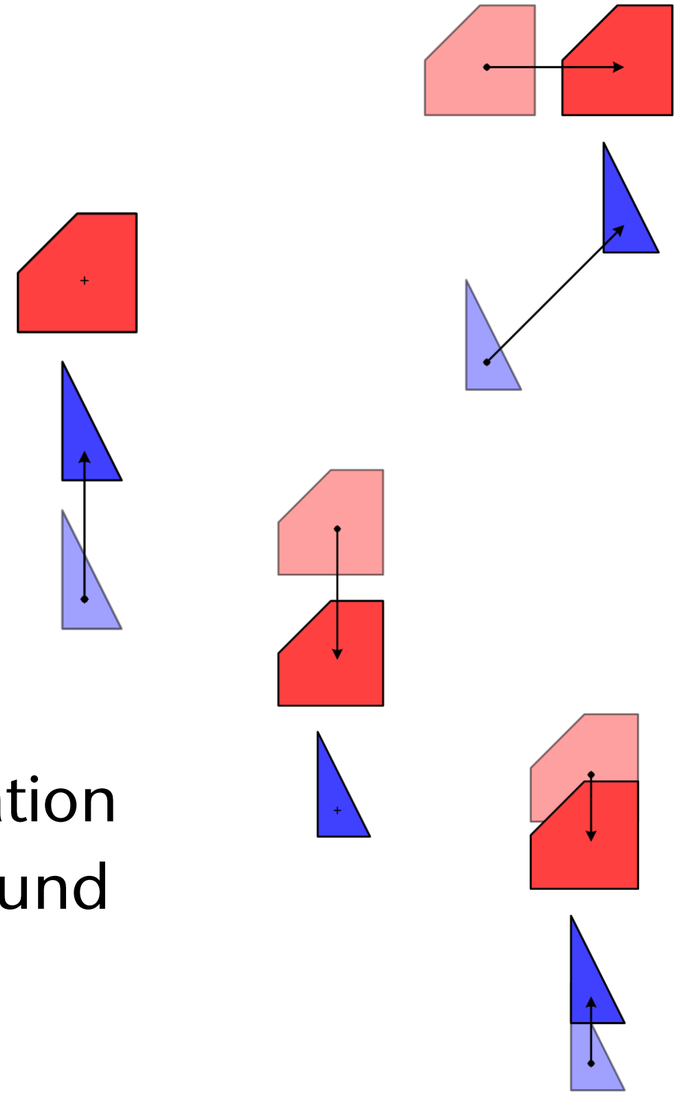






Relative Bewegung

- 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.

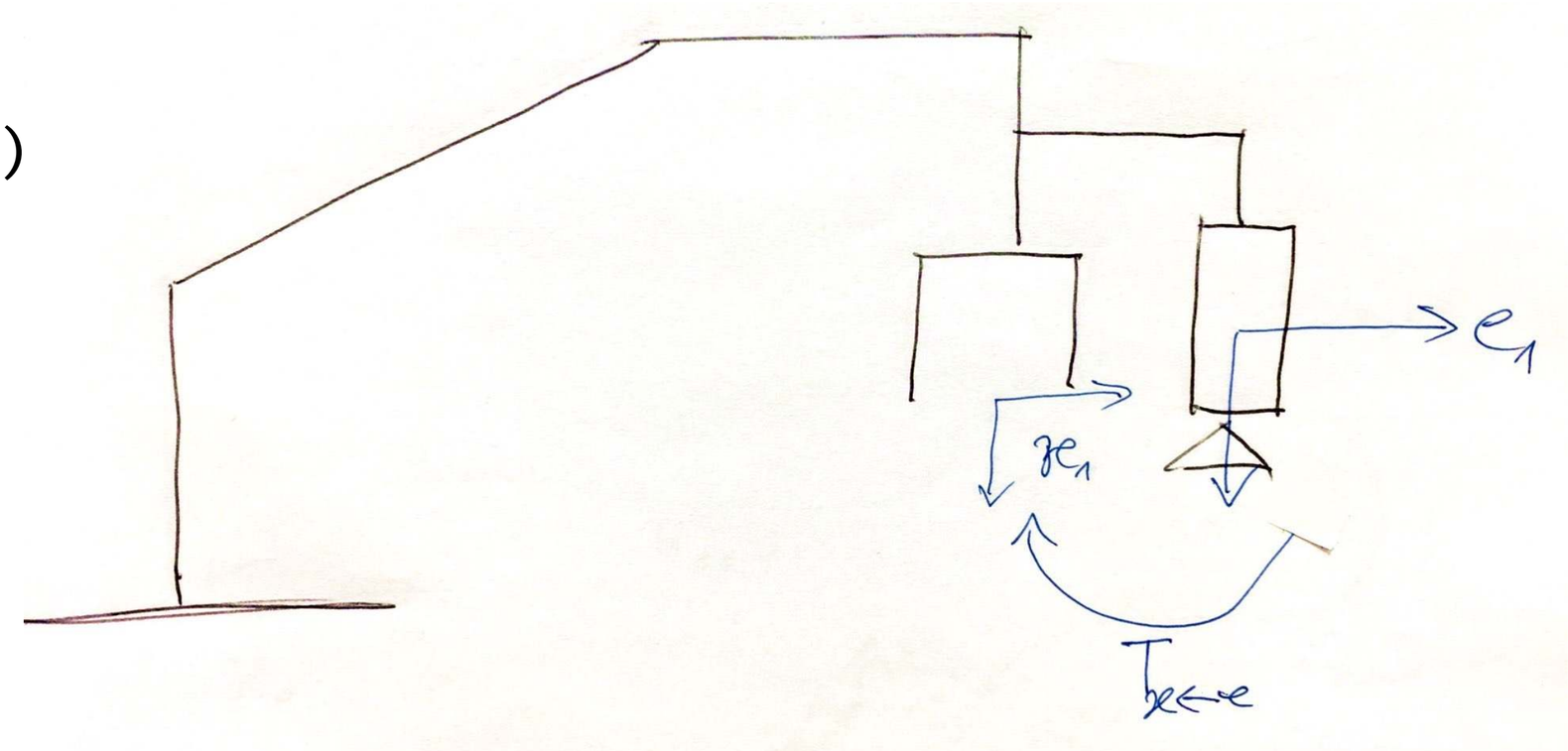


- Eine Anwendung:
  It is always possible to reduce a collision check between two moving objects to a collision check between a moving object and a stationary object (by reframing)

Hand-Eye Calibration

Optional

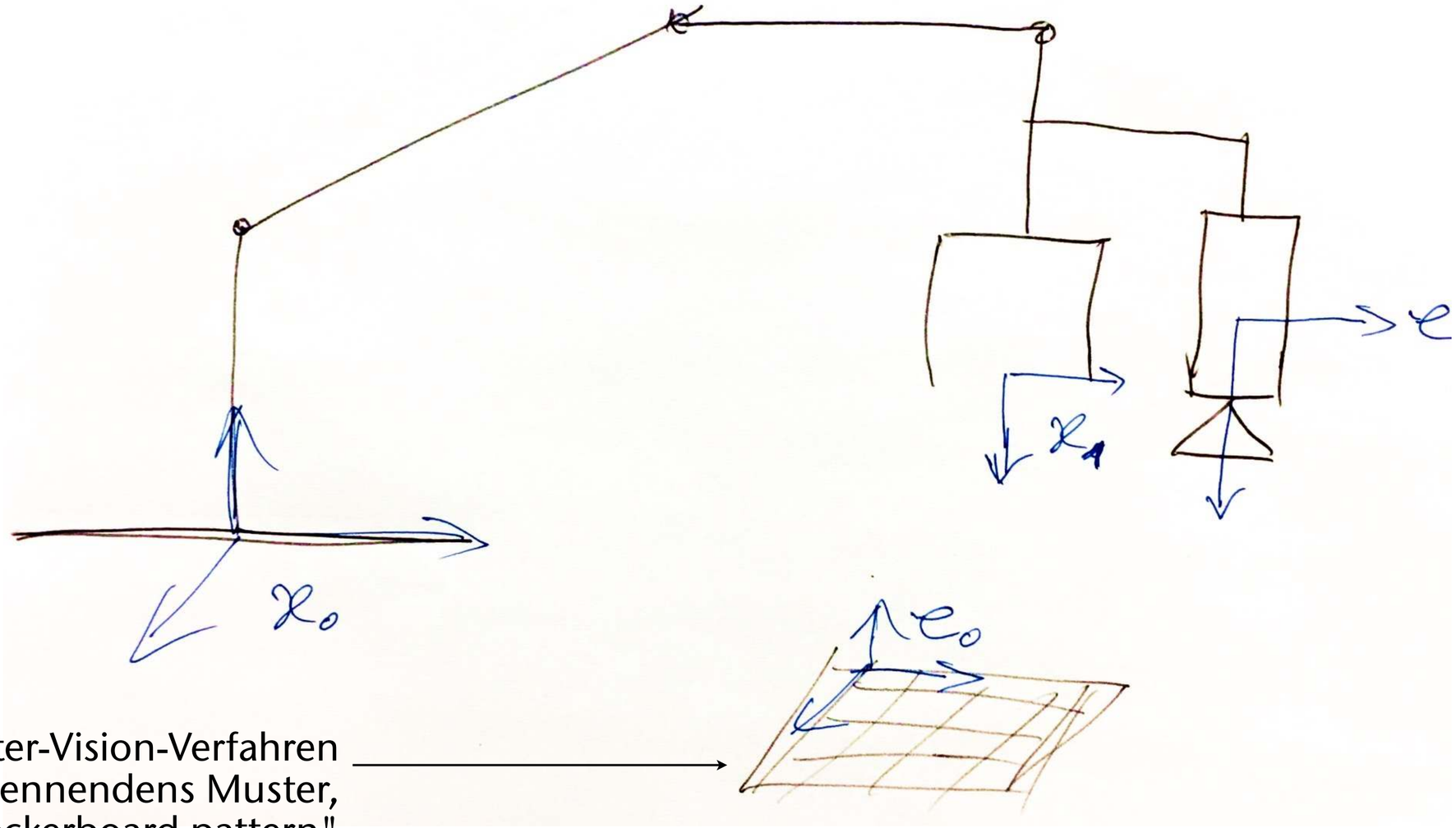
- A.k.a.: Sensor-manipulator calibration, Tracker-HMD-calibration, extrinsische Kamerakalibrierung, ...
- Gegeben:
  Roboter mit Kamera ("eye")
  und Endeffektor (“hand")



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

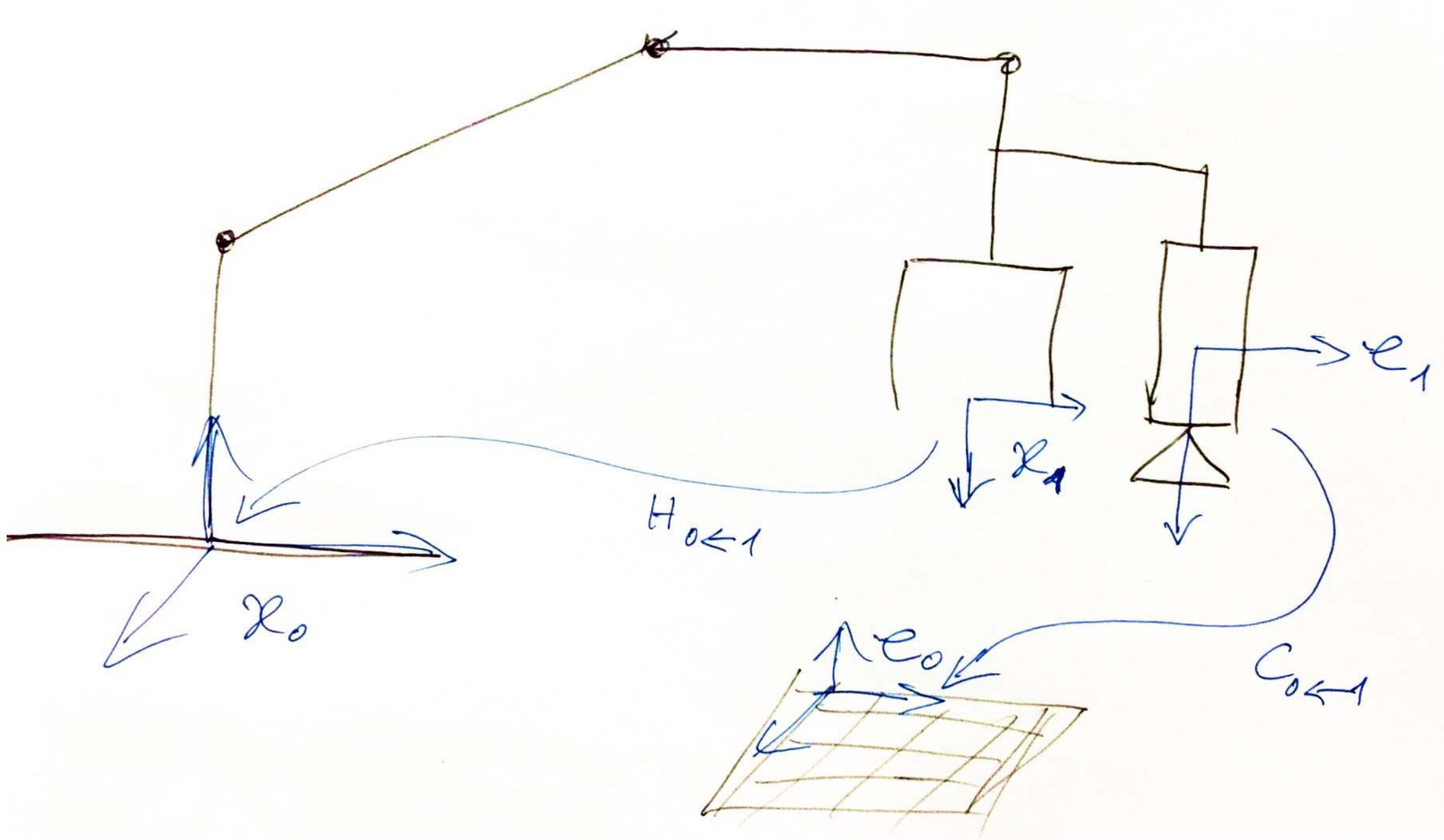
Ansatz

- Führe (zunächst) weitere Koordinaten-systeme ein

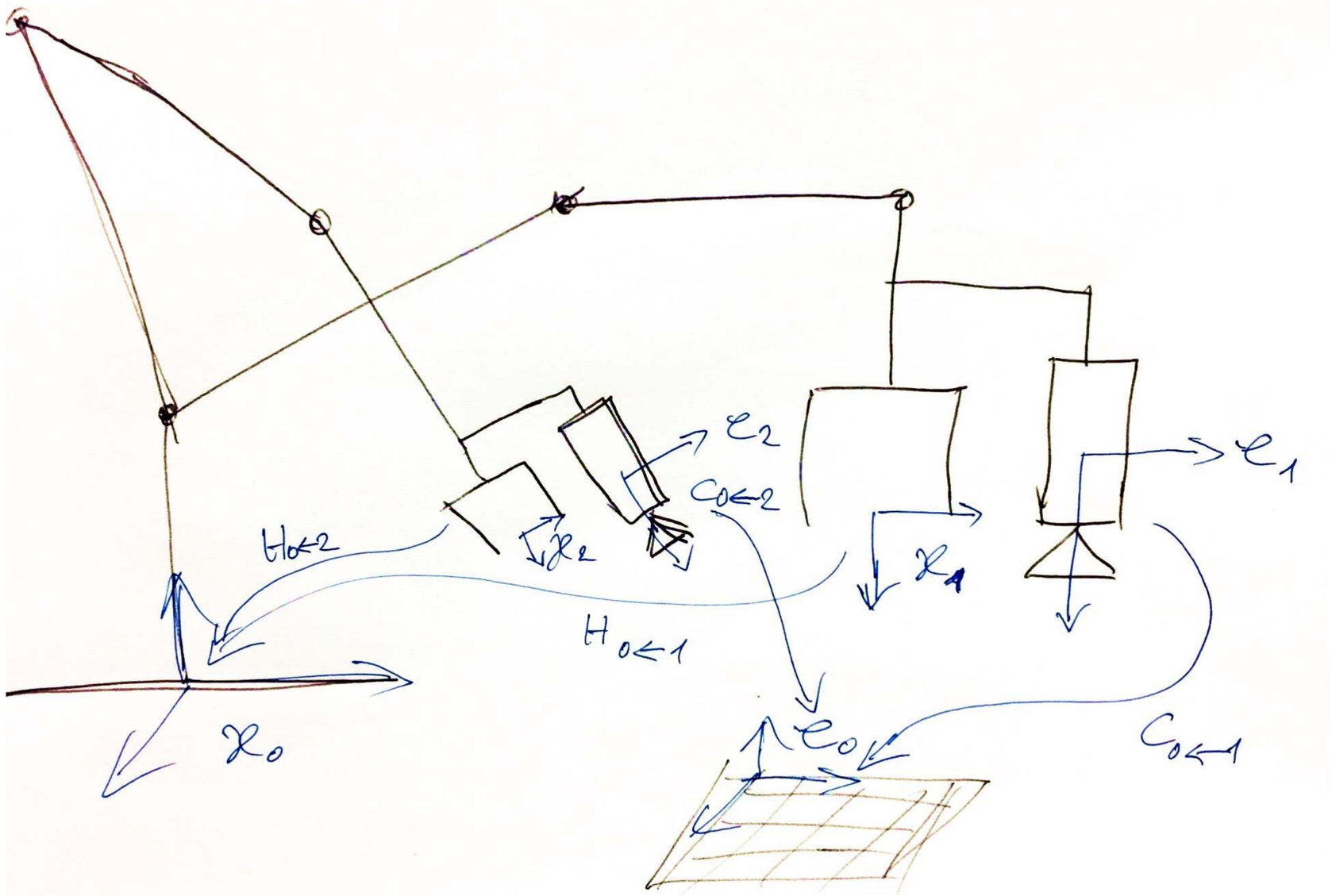


Ein per Computer-Vision-Verfahren “einfach" zu erkennendens Muster, typischerweise ein "checkerboard pattern"

- 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}

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

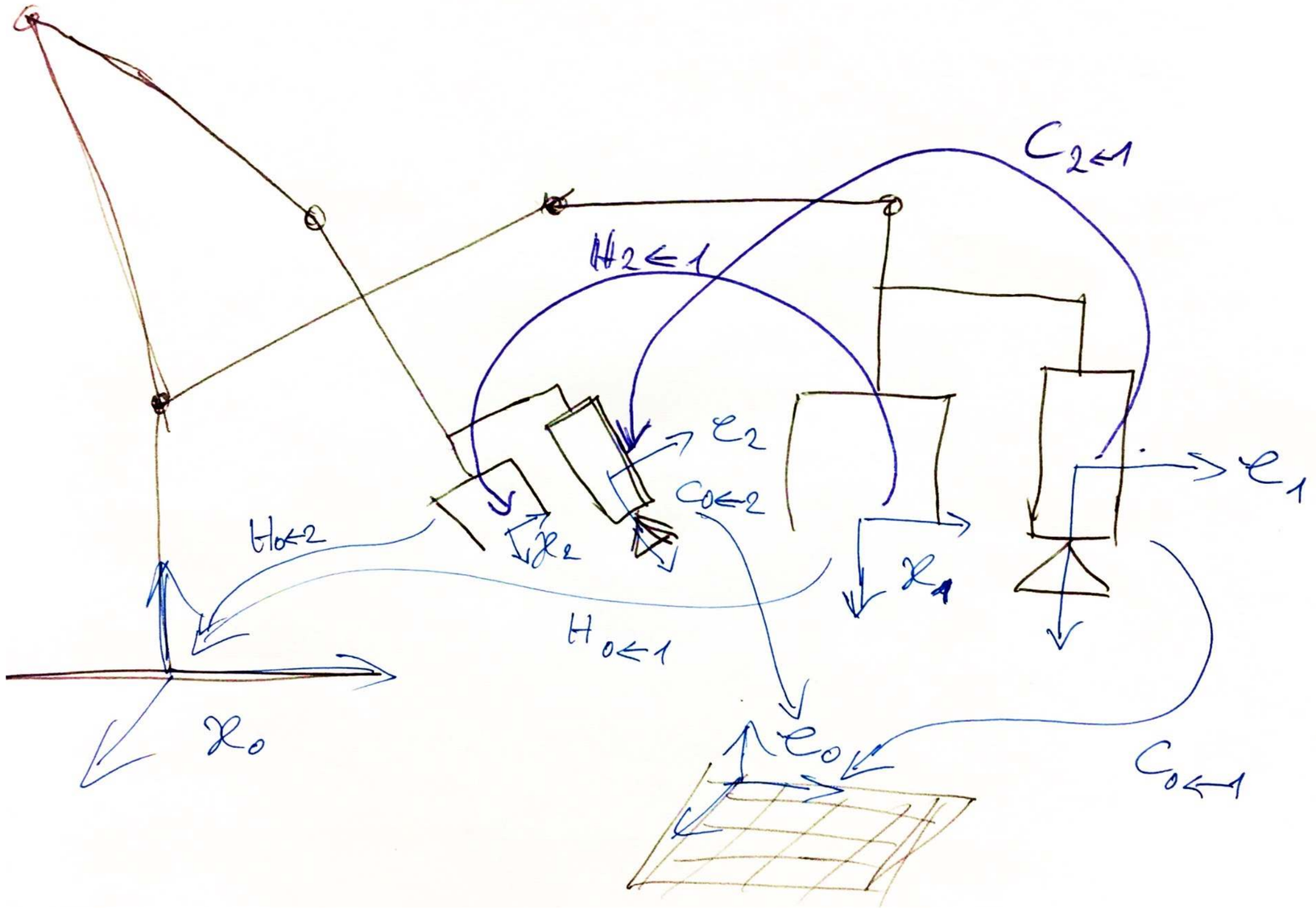
$$\Rightarrow \quad T_{x\leftarrow e} \cdot C_{0\leftarrow 2}^{-1} \cdot C_{0\leftarrow 1} \stackrel{!}{=} H_{0\leftarrow 2}^{-1} \cdot H_{0\leftarrow 1} \cdot T_{x\leftarrow e}$$

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

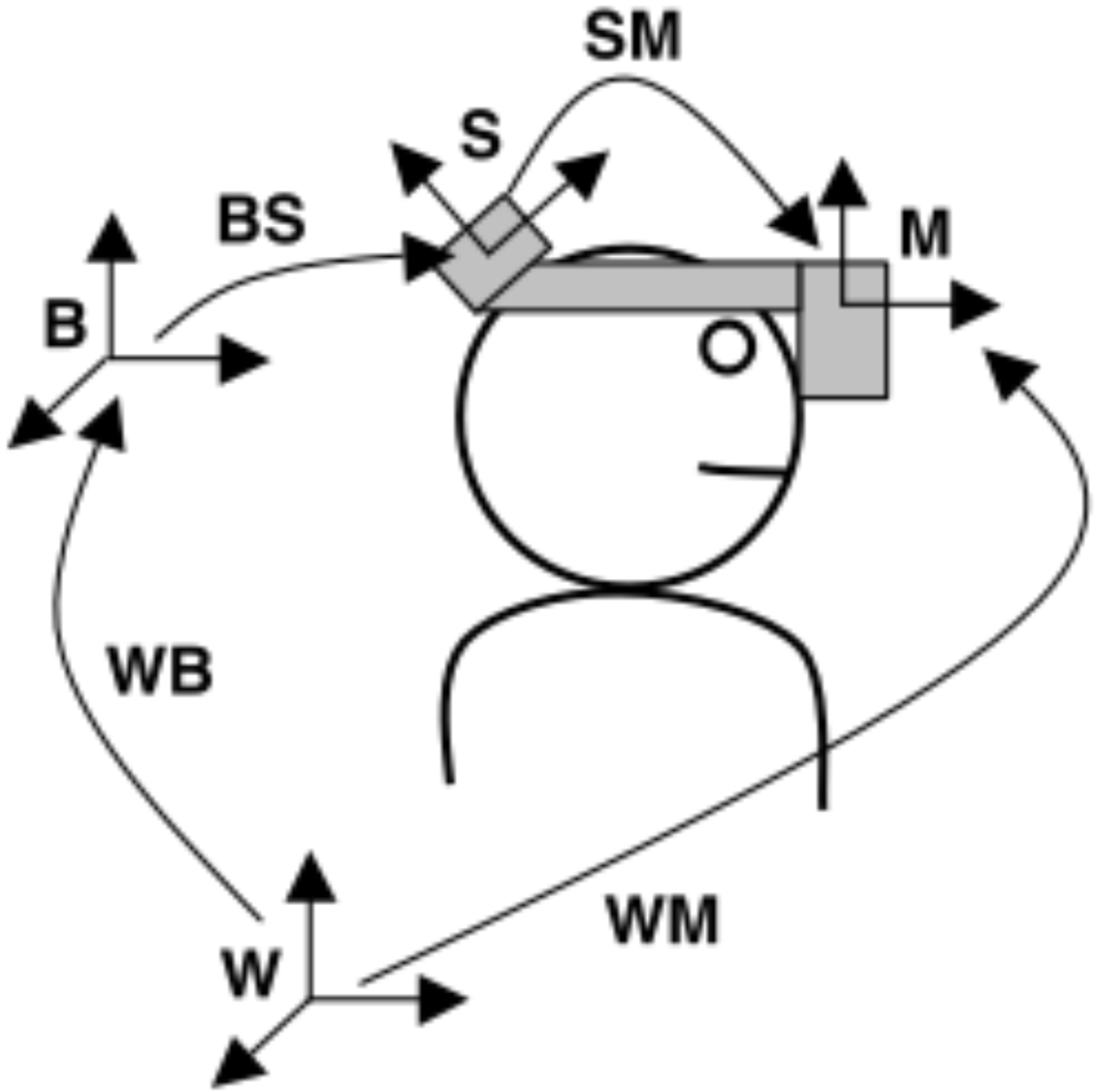
$H_{0\leftarrow 2}^{-1} = H_{2\leftarrow 0} = H_{2\leftarrow 1}$

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

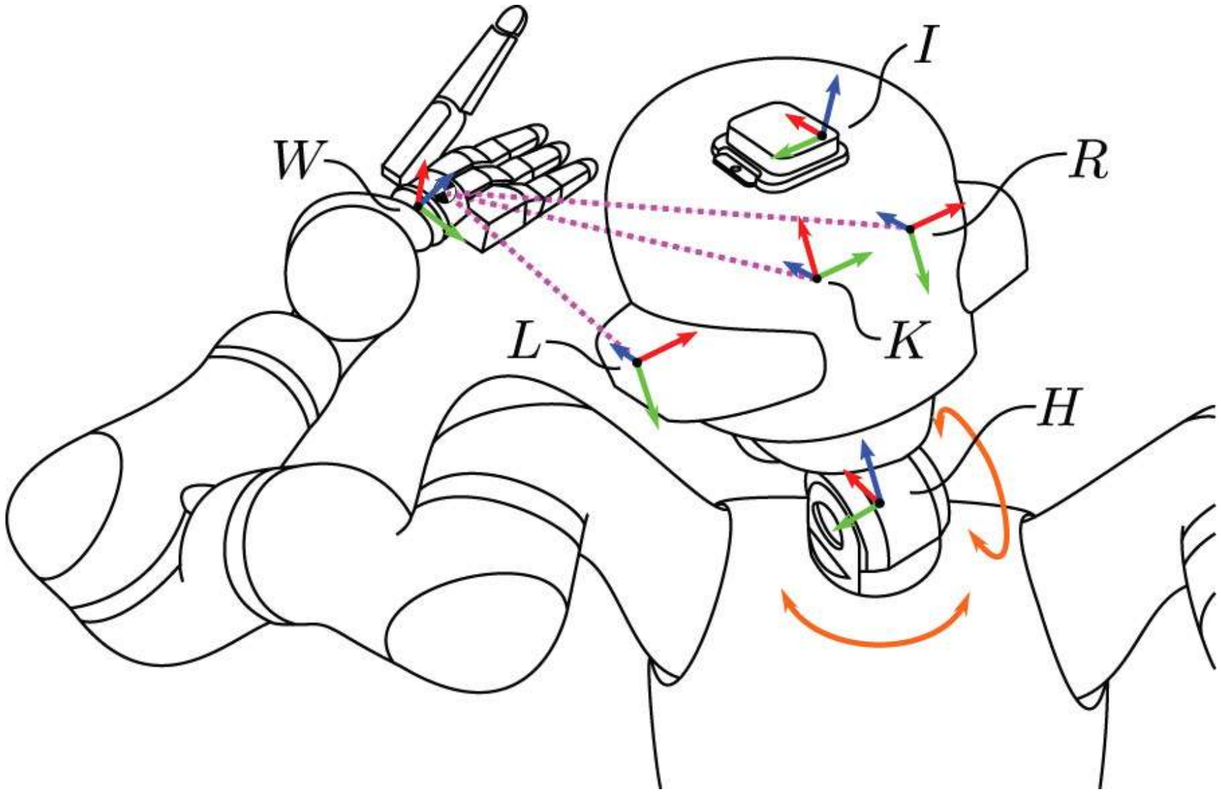
$$T_{\mathcal{H}\leftarrow\mathcal{C}} \cdot C_{2\leftarrow 1} = H_{2\leftarrow 1} \cdot T_{\mathcal{H}\leftarrow\mathcal{C}}$$



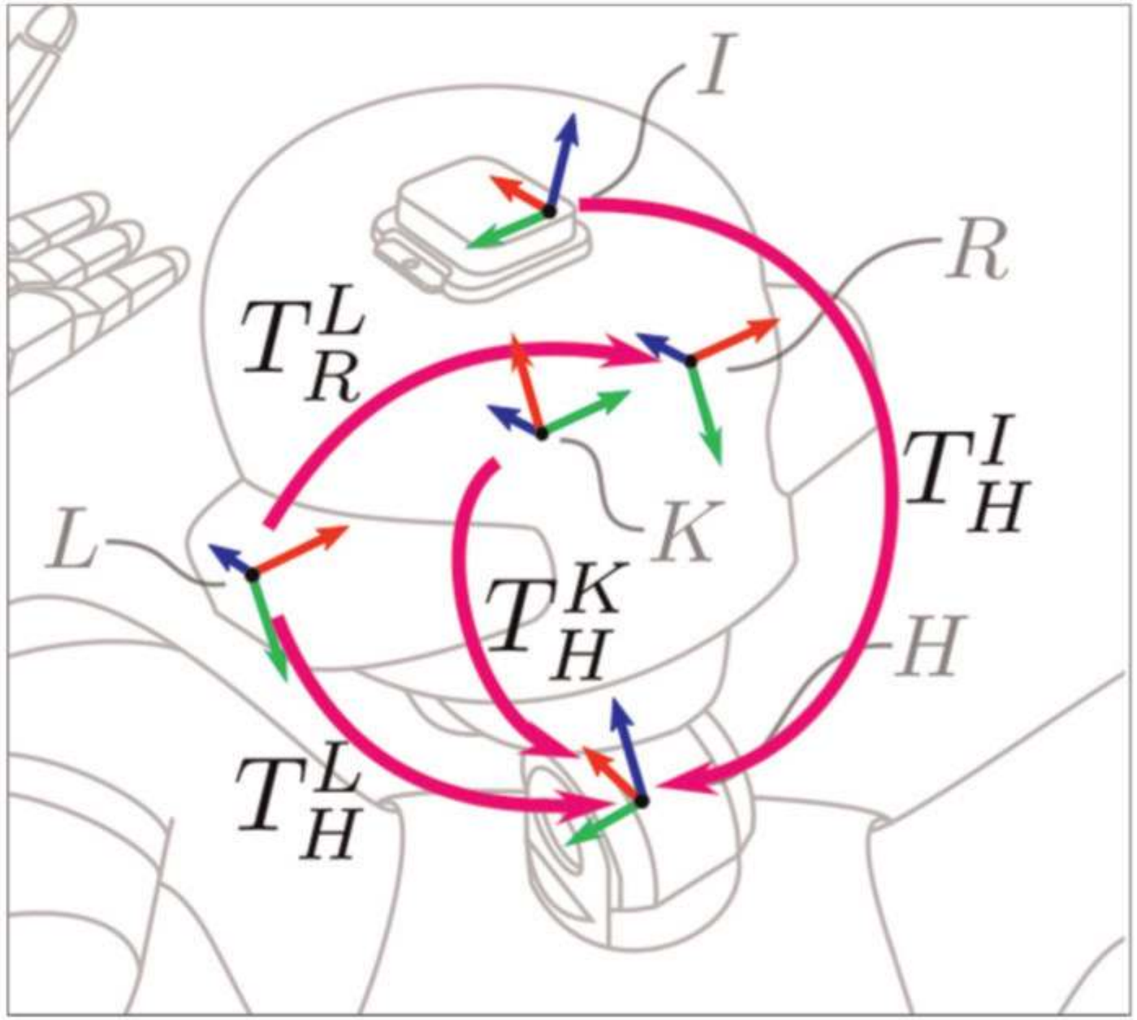
Weiteres Beispiel: Tracker-HMD-Calibration



Weiteres Beispiel: Robotik



Sketch of the calibration process



[Birbach et al., 2014]