h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

# Visual Computing

## Grafische Objekte und deren Programmierung


Hochschule Darmstadt

Prof. Dr. Elke Hergenröther
Björn Frömmer
Prof. Dr. Benjamin Meyer

# KAPITEL 5

## Transformationen

# Mathematische Grundlagen: Transformationen

## Punkte:

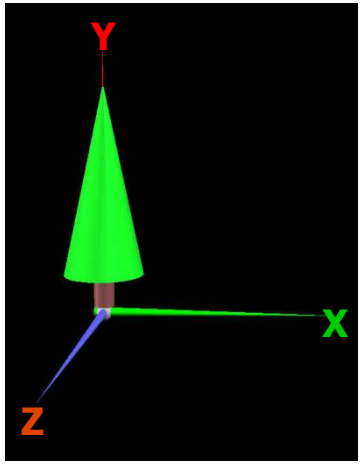
- Punkte (bzw. Vertices) in der Ebene werden durch ihre x- und y-Koordinaten festgelegt.
  - Im dreidimensionalen Raum entsprechend durch ihre x-, y- und z-Koordinaten

- Wir schreiben Punkte als Spaltenvektoren auf:

$$\vec{p} = \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{bzw.} \quad \vec{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

# Anordnung der Objekte im Raum

1.

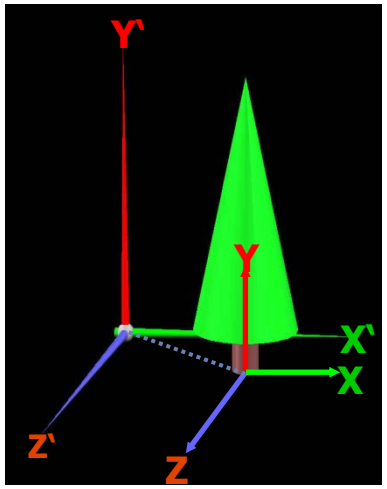


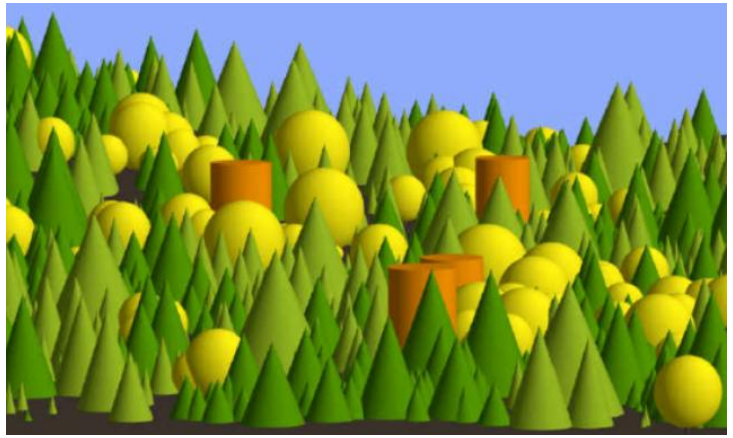

Modellierung des Objektes im lokalen Koordinatensystem

(Modellierungs-koordinatensystem oder körpereigenes Koordinatensystem)

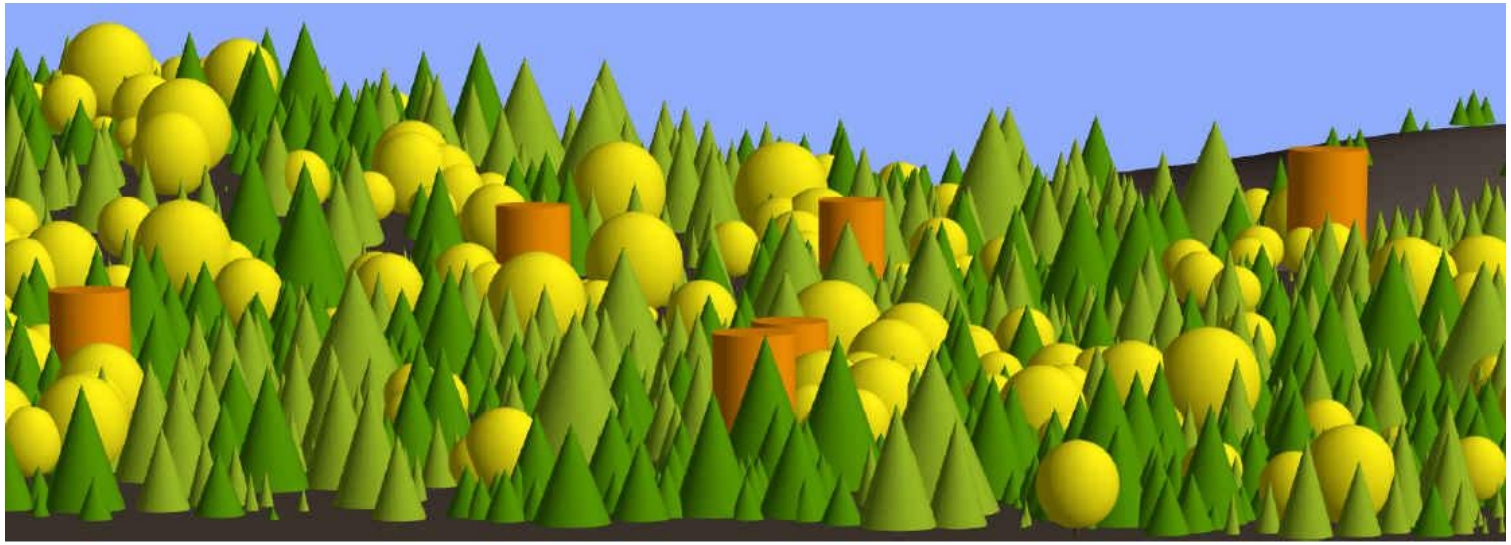
Transformation des Objektes ins Weltkoordinaten-system

2.



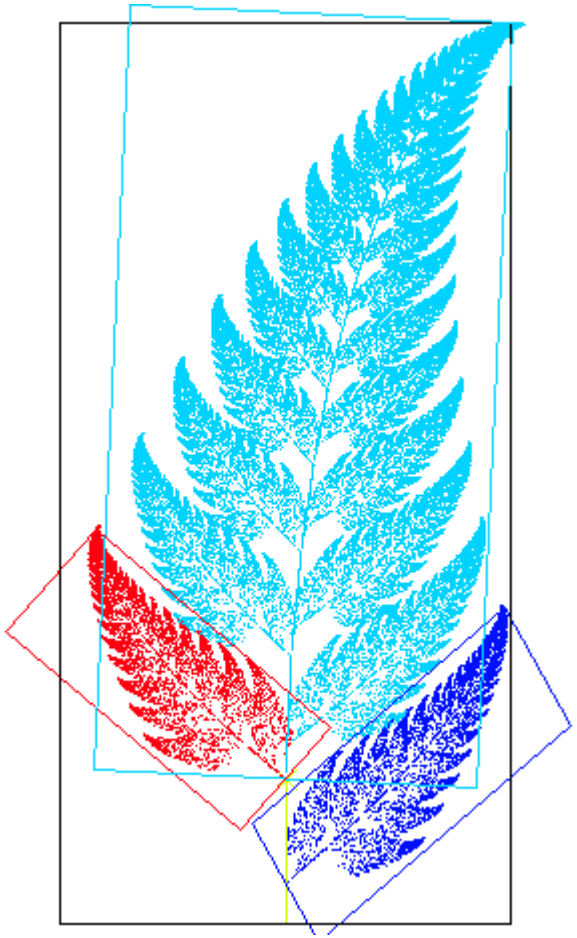
# Anordnung der Objekte im Raum II



# Affine Transformationen

- Die hier verwendeten Transformationen
  - Translation (Verschiebung)
  - Rotation (Drehung)
  - Skalierung (Größenänderung)

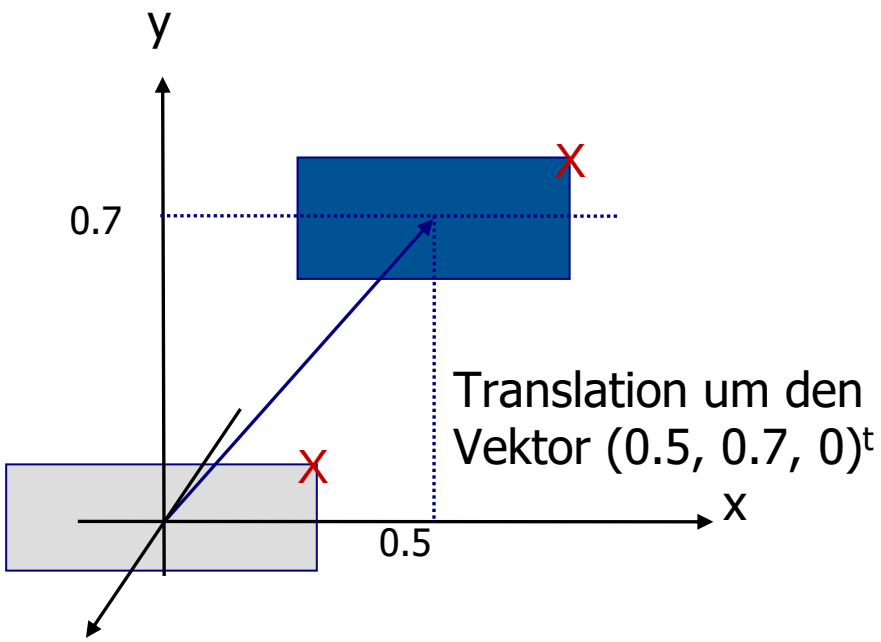
- … werden auch **affine Transformationen** genannt



**Affin:**

- Was parallel ist, bleibt parallel
- Das Verhältnis von Länge, Fläche und Volumen bleibt konstant
  Lässt sich aus dem konstanten Teilverhältnis ableiten.

# Beispiel für eine Translation




- Erinnerung: Es werden nur die Eckpunkte der Geometrie bewegt!
- Mathematisch ist dies eine Addition eines Translationsvektors $\vec{t}$ zum ursprünglichen Punkt $\vec{p}$ :

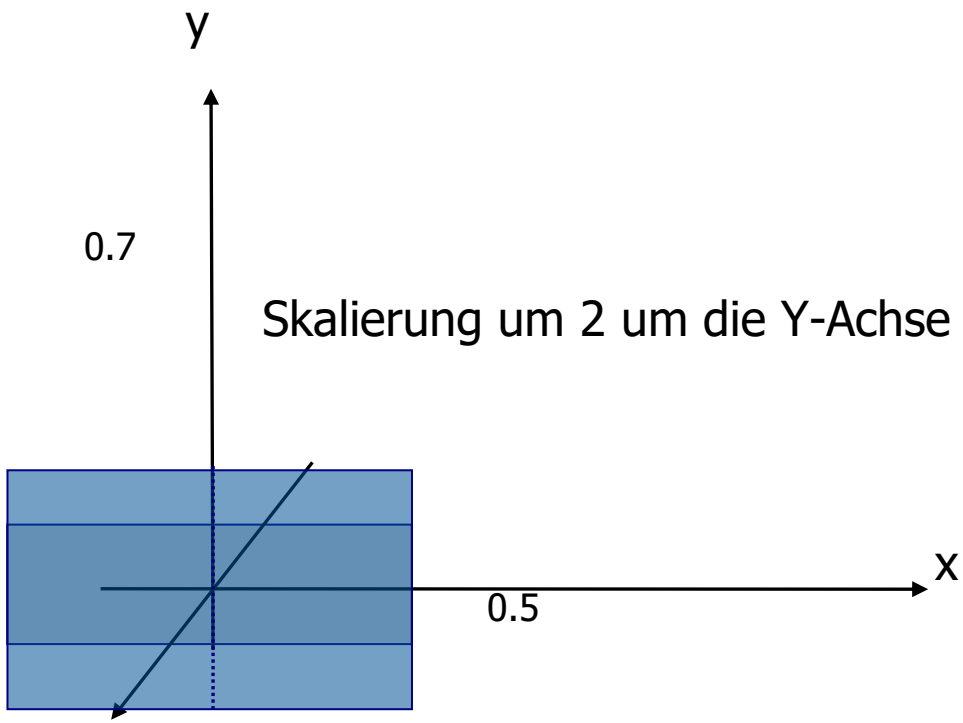
$$\overrightarrow{p'} = \vec{t} + \vec{p}$$

- Beispielhaft für die Ecke oben rechts an den Koordinaten $(0.3, 0.1, 0)^t$:

$$\vec{t} + \vec{p} = \begin{bmatrix} 0.5 \\ 0.7 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.3 \\ 0.1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.8 \\ 0.8 \\ 0 \end{bmatrix} = \overrightarrow{p'}$$

neutrales Element der Translation

# Beispiel für eine Skalierung




- Es gilt für jeden Punkt $\vec{p}$ bei einer Skalierung um $s_x$ entlang der X-, $s_y$ entlang der Y- und $s_z$ entlang der Z-Achse:

$$p'_x = s_x * p_x \;,\; p'_y = s_y * p_y \quad \text{und} \quad p'_z = s_z * p_z$$

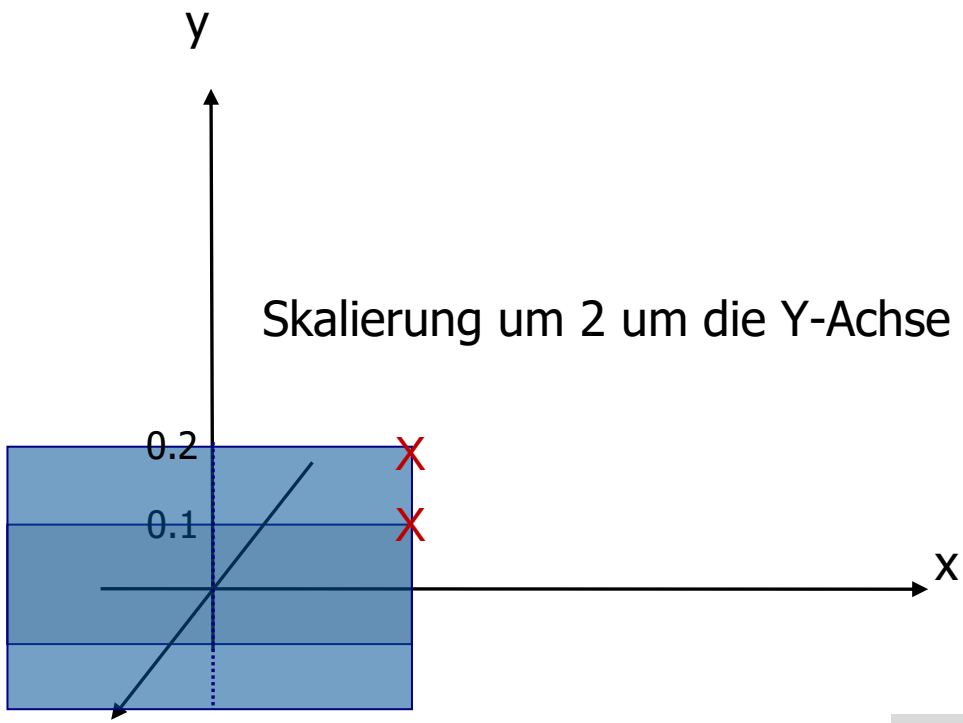
- Diese Gleichungen lassen sich durch eine Matrixmultiplikation zusammenfassen!

→

$$\begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} * \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

Achtung! Reihenfolge der Multiplikation beachten!

# Beispiel für eine Skalierung II



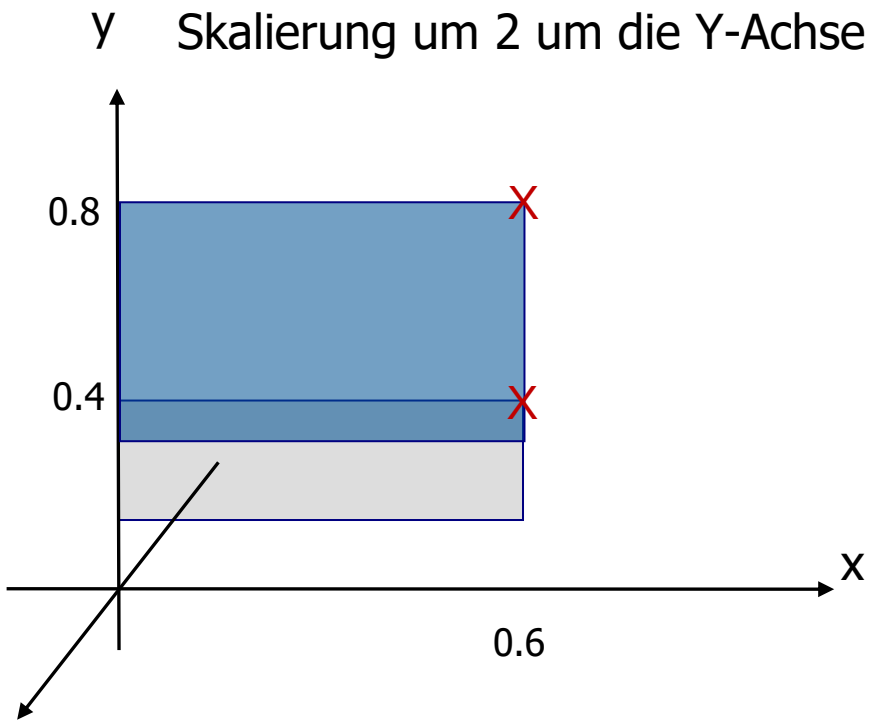

- Beispielhaft für die Koordinaten $(0.3, 0.1, 0)^t$:

$$\vec{s} * \vec{p} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} * \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 0.3 \\ 0.1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0 \end{bmatrix} = \overrightarrow{p'}$$

neutrales Element der Skalierung!

# Beispiel für eine Skalierung III



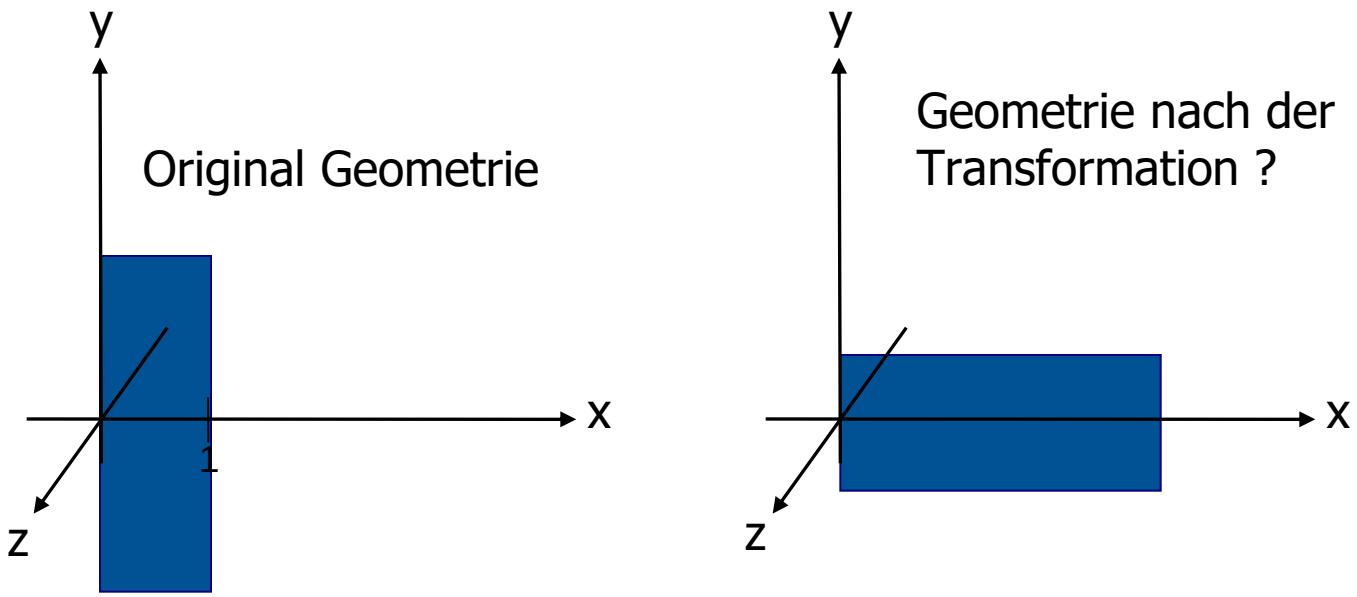

- Erneut für die obere rechte Ecke, jetzt bei $(0.6, 0.4, 0)^t$

$$\vec{s} * \vec{p} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 0.6 \\ 0.4 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.6 \\ 0.8 \\ 0 \end{bmatrix} = \vec{p'}$$

- **Beobachtung:** Das Objekt wird nicht nur in Y-Richtung skaliert, sondern verändert auch die Position
  → der Abstand zum Nullpunkt wird mit skaliert!

- **Daher gilt:** Die Skalierung ist nur gegenüber dem Nullpunkt invariant!

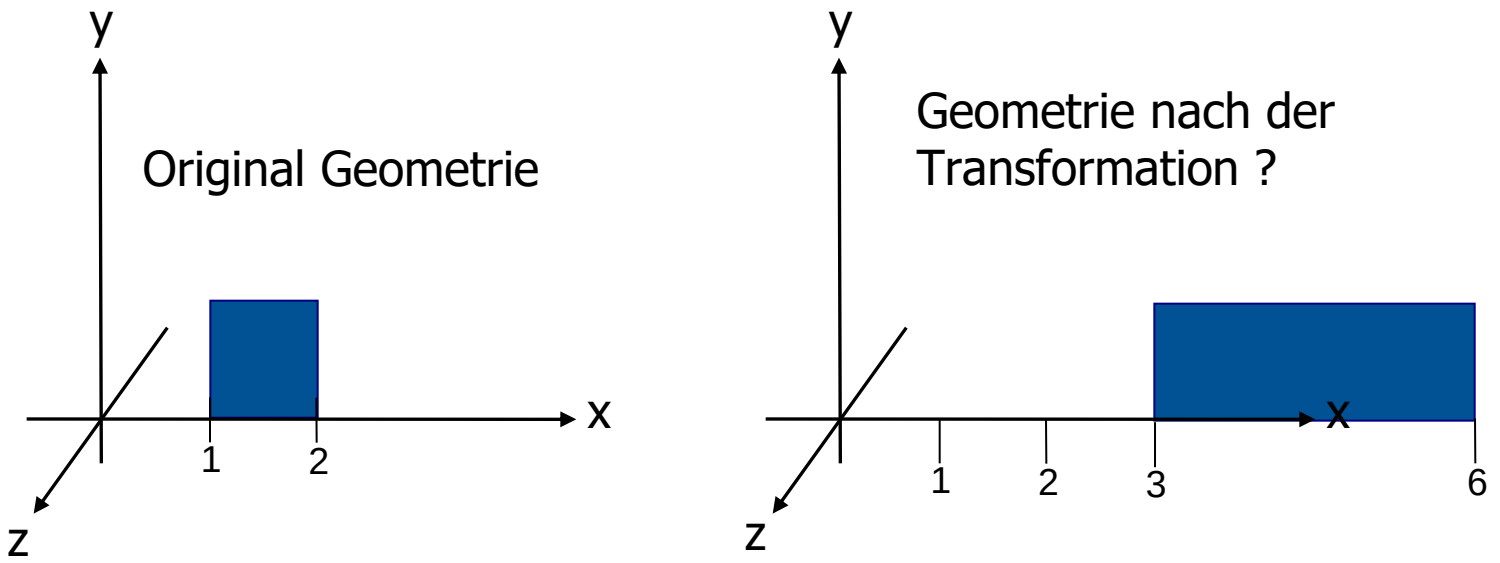
# Beispiel für eine Skalierung III




Wie sieht die Geometrie nach der Skalierung mit diesen Faktoren aus?

x = 3 , y = 0.3 und z = 1

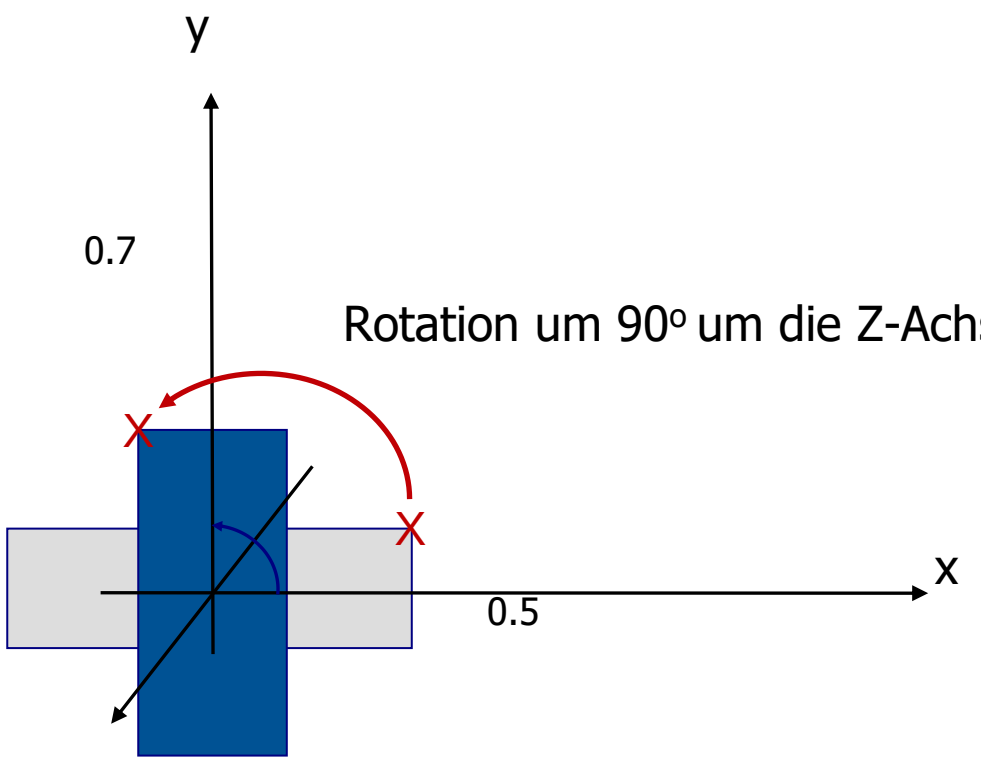
# Beispiel für eine Skalierung III




Wie sieht die Geometrie nach der Skalierung mit diesen Faktoren aus?

x = 3 , y = 1 und z = 1

# Beispiel für eine Rotation




- Auch die Rotation kann als Matrixmultiplikation der einzelnen Eckpunkte formuliert werden
- Rotation um die Z-Achse um den Winkel α:

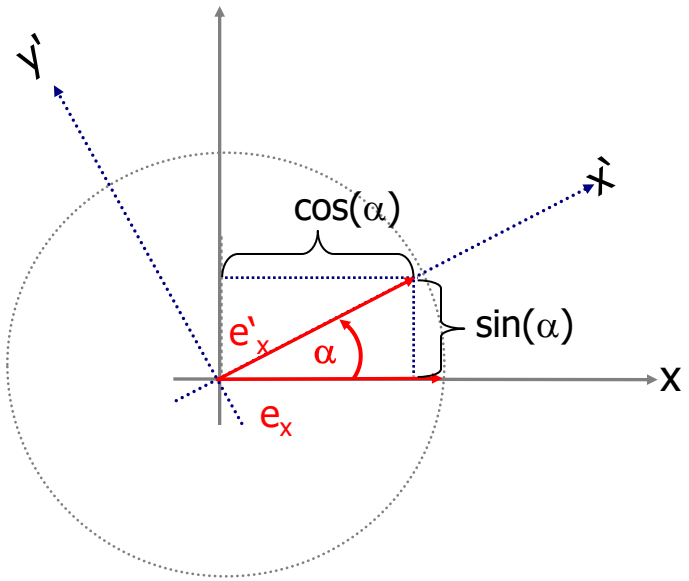
$$\vec{r} * \vec{p} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = \vec{p'}$$

- Beispielhaft für die Koordinaten $(0.3, 0.1, 0)^t$ und 90°:

$$\vec{r} * \vec{p} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 0.3 \\ 0.1 \\ 0 \end{bmatrix} = \begin{bmatrix} -0.1 \\ 0.3 \\ 0 \end{bmatrix}$$

# Herleitung der Rotationsmatrix (in 2D)

Nicht das Objekt wird rotiert, sondern das Koordinatensystem:



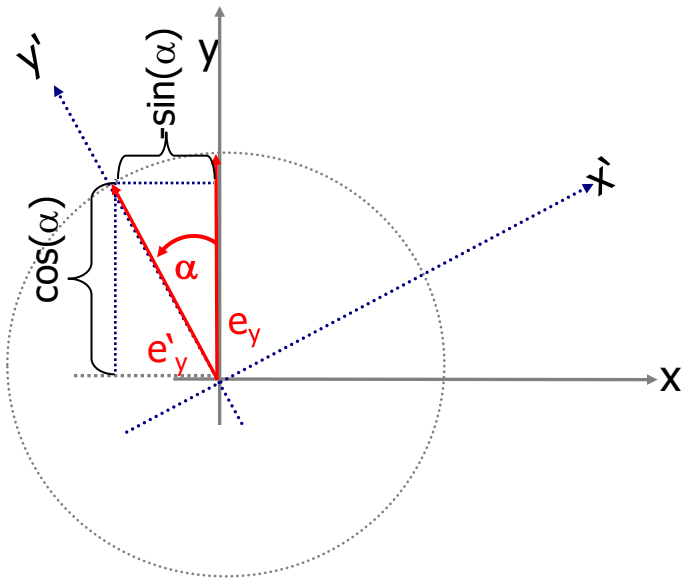

Bei einer Rotation um den Winkel α werden die Einheitsvektoren $e_x$ und $e_y$ des kartesischen Koordinatensystems auf die Basisvektoren $e'_x$ und $e'_y$ des affinen Koordinatensystems abgebildet:

$$e'_x = T_R * e_x$$

$$\begin{pmatrix} \cos(\alpha) \\ \sin(\alpha) \end{pmatrix} = \begin{pmatrix} \cos(\alpha) & n.d. \\ \sin(\alpha) & n.d. \end{pmatrix} * \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

# Herleitung der Rotationsmatrix (in 2D)

Nicht das Objekt wird rotiert, sondern das Koordinatensystem:



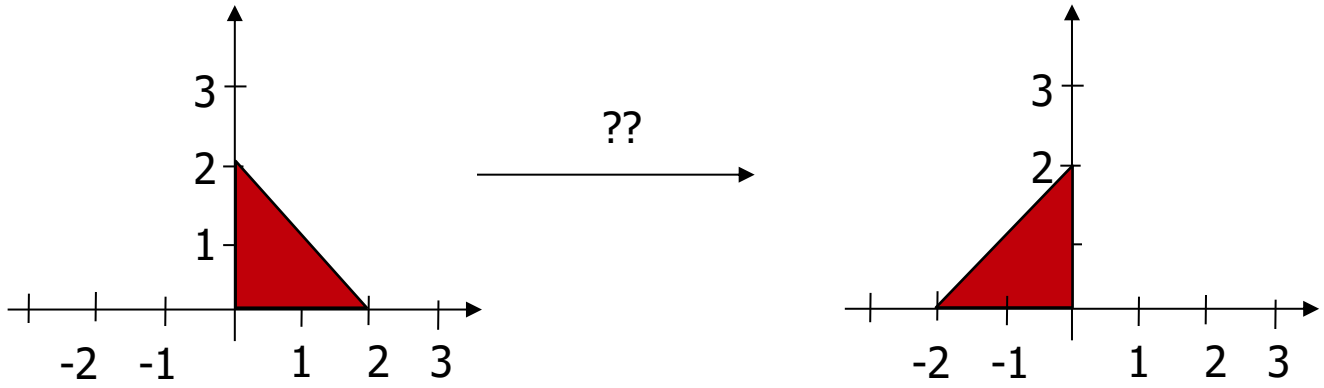

Bei einer Rotation um den Winkel α werden die Einheitsvektoren $e_x$ und $e_y$ des kartesischen Koordinatensystems auf die Basisvektoren $e'_x$ und $e'_y$ des affinen Koordinatensystems abgebildet:

$$e'_y = T_R * e_y$$

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

# Beispiel in 2D

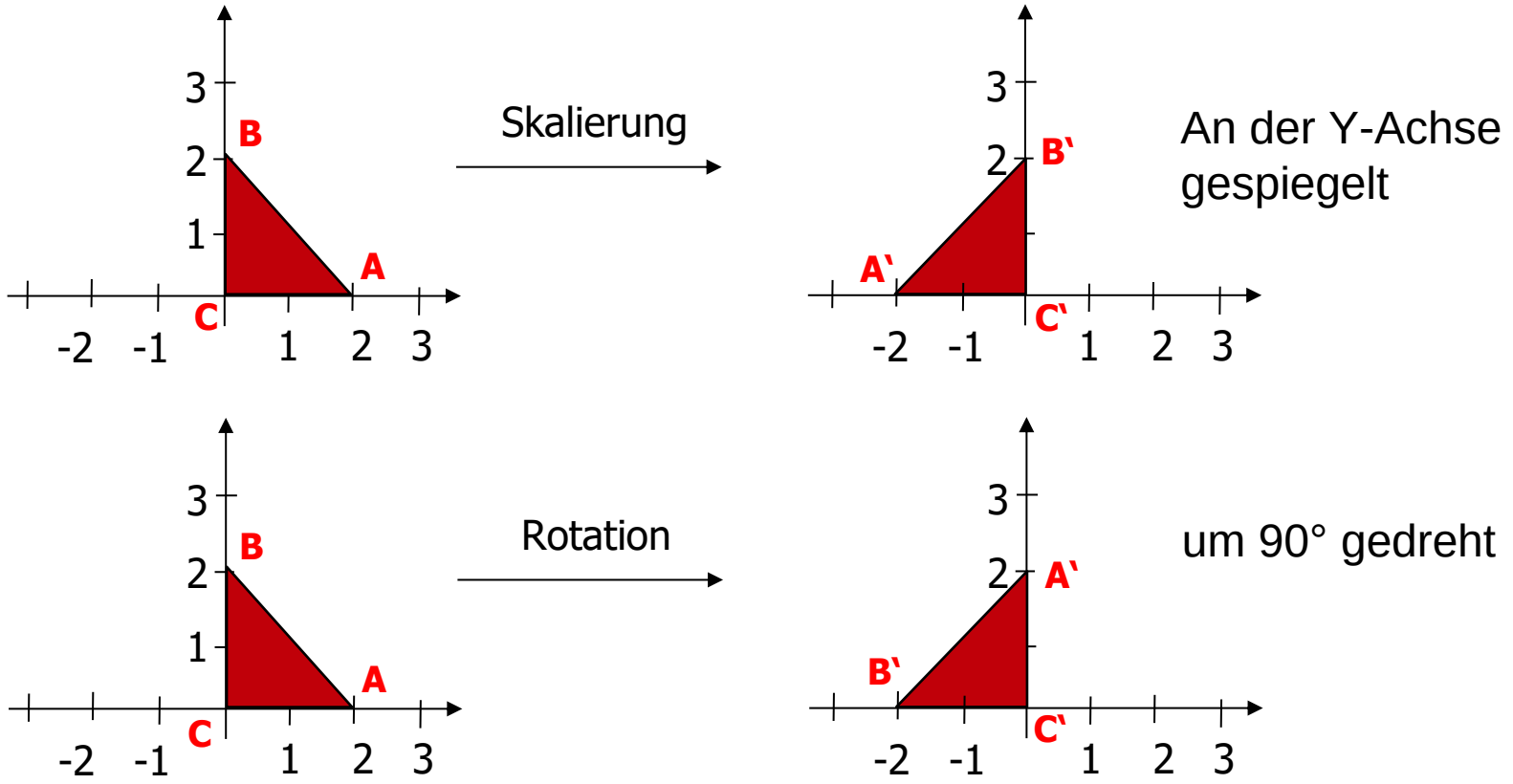
Welche Transformation wird gesucht?




Ohne Beschriftung der Eckpunkte gibt es mehrere Lösungen!

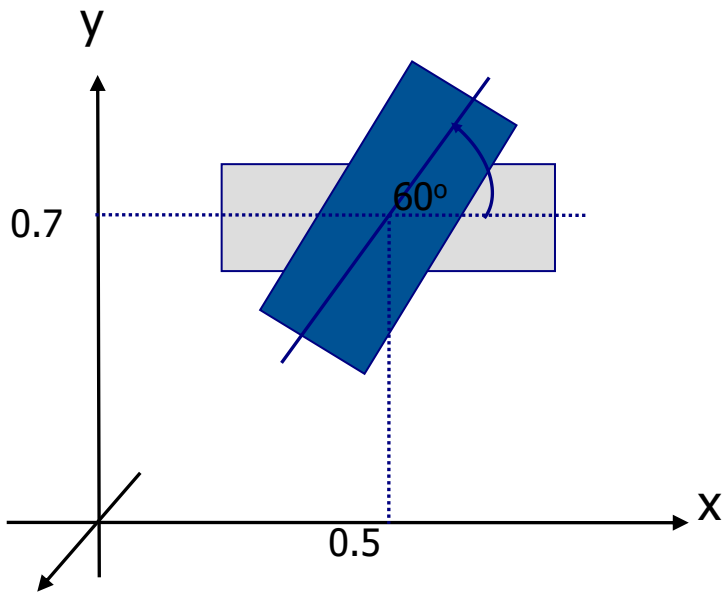
Welche?

# Beispiel in 2D



# Komplexe Transformationen

- Welche Transformationen sind notwendig um das Rechteck, wie vorgegeben, zu drehen?




3. Rücktransformation in die Originalposition
2. Rotation um die Z-Achse
1. Transformation in den Ursprung

- → **Reihenfolge, in der die Transformationen angewendet werden, ist wichtig!**

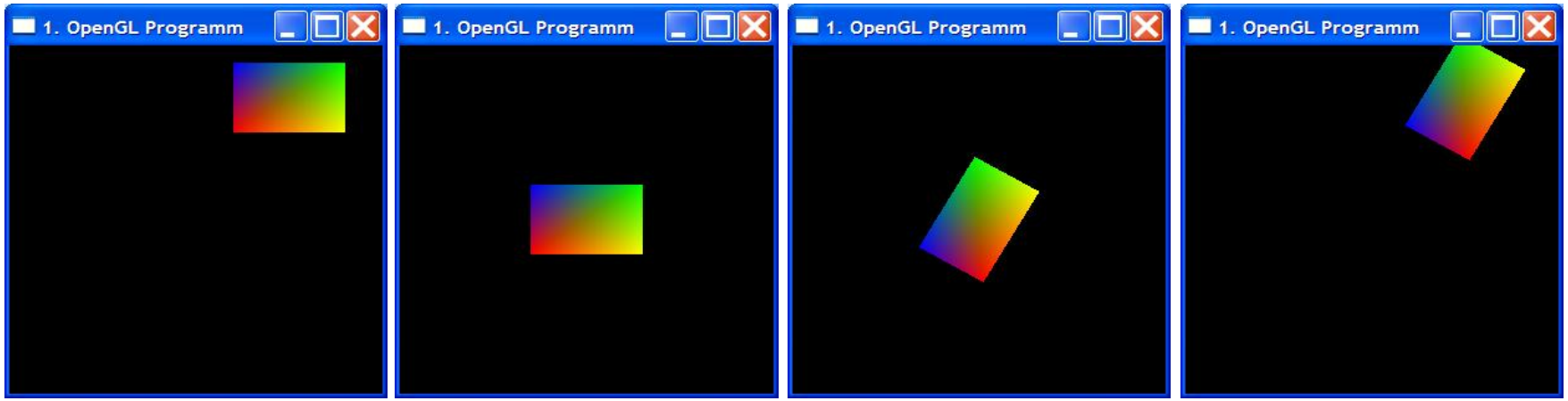
# Reihenfolge der Transformationen II

**Beispiel:** Drehen eines Quadrats um 60°

- Transformationen werden in **umgekehrter** Reihenfolge angegeben:

Reihenfolge in der, die Transformationen auf das Quadrat angewendet werden:

```
Matrix4f m = new Matrix4f();    // neue Mattix anlegen
m.translate( 0.5, 0.7, 0.);     // Rücktransformation
m.rotate( 60., 0., 0., 1.);     // Rotation
m.translate( -0.5, -0.7, 0.);   // In den Ursprung
```



# Reihenfolge der Transformationen V

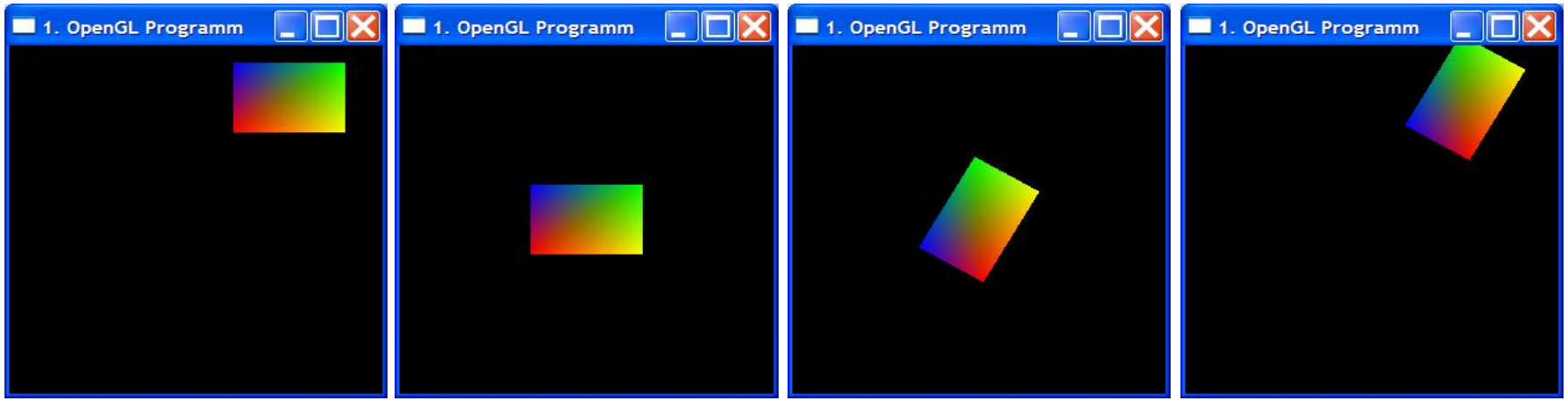
**Beispiel:** Drehen eines Quadrats um 60°

- Transformationen werden in **umgekehrter** Reihenfolge angegeben:

Reihenfolge in der, die Transformationen auf die akkumulierte Matrix M multipliziert werden:

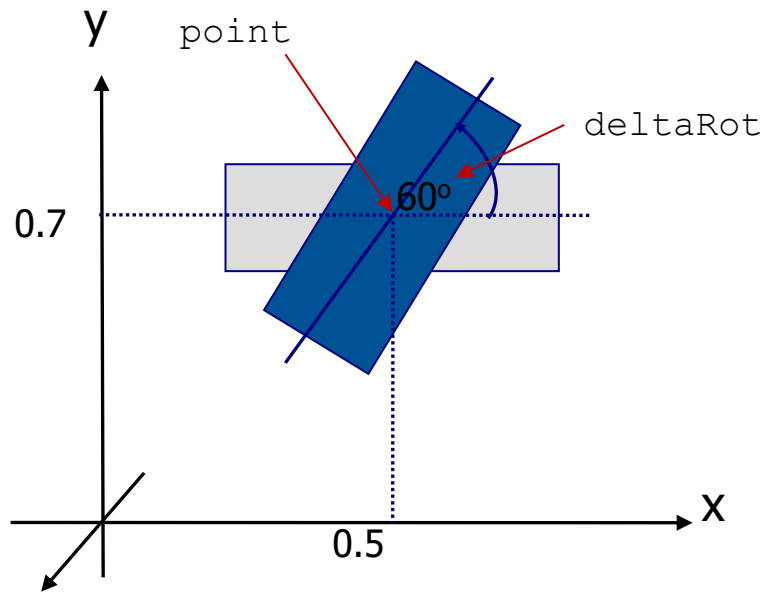
$M = M_{T2} * M_R * M_{T1}$

```
Matrix4f m = new Matrix4f();    // neue Mattix anlegen
m.translate( 0.5, 0.7, 0.);     // Rücktransformation
m.rotate( 60., 0., 0., 1.);     // Rotation
m.translate( -0.5, -0.7, 0.);   // In den Ursprung
```



# Komplexe Transformationen

- Welche Transformationen sind notwendig um das Rechteck, wie vorgegeben, zu drehen?
- Im Praktikumsframework finden sie die Funktion `Transform::rotateAroundPoint`, die diese komplexe Transformation durchführt.




3. Rücktransformation in die Originalposition
2. Rotation um die Z-Achse
1. Transformation in den Ursprung

```
void Transform::rotateAroundPoint(
        const glm::vec3 point,
        const glm::quat &deltaRot)
{
        glm::mat4x4 mm = getMatrix();
        mm = glm::translate(point)*mm;
        mm = glm::toMat4(deltaRot)*mm;
        mm = glm::translate(-point) *mm;
        setMatrix(mm);
}
```

Aus: `OpenGLPraktikum\src\Framework\SceneElements\Transform.cpp`

OpenGL Mathematics (GLM)

GLM is a C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specification.

# Zusammenfassung bis hier hin..

- Die Drehung und Skalierung erfolgt **immer relativ zum Ursprung.**

- Drehen/Skalieren mit Referenzpunkt ist ein dreistufiger Algorithmus:
  - Verschieben des Referenzpunktes zum Ursprung
  - Transformation anwenden
  - Verschieben des Referenzpunktes zurück in die ursprüngliche Position

- Verkettung von Transformationen notwendig.....
- aber die Anwendung einer Transformation nach der anderen auf jeden Eckpunkt **dauert lange** und **führt zu Rundungsfehlern**

- Besseres Konzept: **Transformationen zusammenfassen** und auf die ursprünglichen Eckpunkte anwenden → **akkumulierte Transformationsmatrix**

# Reihenfolge der Transformationen III

Warum werden die Transformationen in umgekehrter Reihenfolge angegeben?

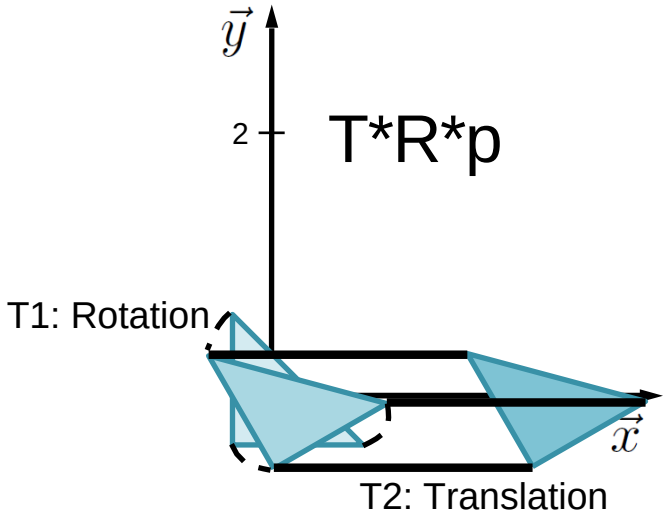
- Matrixmultiplikationen sind nicht kommutativ (d.h. sie sind nicht in ihrer Reihenfolge vertauschbar)
- Beispiel: a) liefert nicht dasselbe Ergebnis, wie b)

a)
$$\begin{bmatrix}3\\2\\3\\1\end{bmatrix} = \begin{bmatrix}1&0&0&2\\0&1&0&3\\0&0&1&0\\0&0&0&1\end{bmatrix} \cdot \begin{bmatrix}0&1&0&0\\-1&0&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} \cdot \begin{bmatrix}1\\2\\3\\1\end{bmatrix}$$

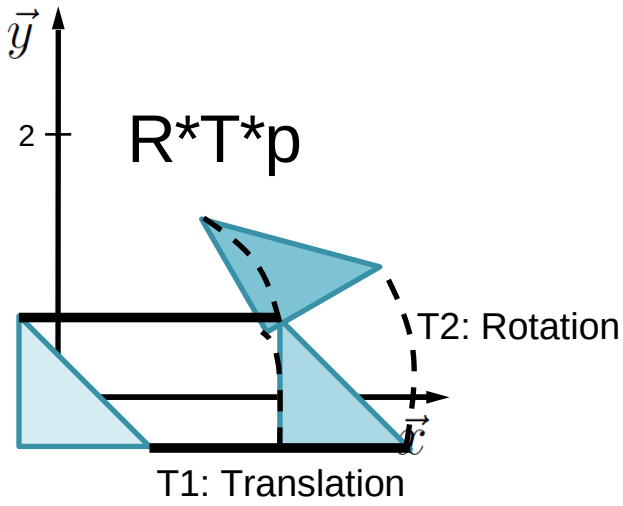
b)
$$\begin{bmatrix}5\\-3\\3\\1\end{bmatrix} = \begin{bmatrix}0&1&0&0\\-1&0&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} \cdot \begin{bmatrix}1&0&0&2\\0&1&0&3\\0&0&1&0\\0&0&0&1\end{bmatrix} \cdot \begin{bmatrix}1\\2\\3\\1\end{bmatrix}$$

# Reihenfolge der Transformationen

```
Matrix4f m = new Matrix4f();
m.translate(2, 0, 0); //T2
m.rotate(30, 0, 0, 1); //T1
```




```
Matrix4f m = new Matrix4f();
m.rotate(30, 0, 0, 1); //T2
m.translate(2, 0, 0); //T1
```




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

# Reihenfolge der Transformationen IV

Warum werden die Transformationen in umgekehrter Reihenfolge angegeben?

- Matrixmultiplikationen sind nicht kommutativ (d.h. sie sind nicht in ihrer Reihenfolge vertauschbar)
- Äquivalente Berechnungen von P‘:

$$\vec{P}' = M_2 \cdot (M_1 \cdot \vec{P}) \Leftrightarrow \vec{P}' = (M_2 \cdot M_1) \cdot \vec{P}$$

$$\vec{P}' = M_2 \cdot \vec{P}^{M_1} \Leftrightarrow \vec{P}' = M \cdot \vec{P}$$

Intuitive Vorgehensweise wenn man „von Hand“ transformiert:
P wird zunächst mit $M_1$ und dann wird der Ergebnisvektor mit $M_2$ multipliziert

Vorgehensweise von OpenGL:
Erstellen einer akkumulierten Matrix ($M_2$ wird mit $M_1$ multipliziert).

# Homogene Koordinaten

## Problem:

Translation = Vektoraddition
Skalierung & Rotation = Matrixmultiplikation

Da es sinnvoll ist die verschiedenen Transformationen miteinander zu einer akkumulierten Matrix zu „verrechnen“ und dann auf alle Eckpunkte anzuwenden (siehe OpenGL), muss es eine Möglichkeit geben, den Additivanteil (Translation) und den Multiplikativanteil (Rotation, Skalierung) miteinander zu verknüpfen.

# Homogene Koordinaten

## Problem:

- **Rotation** und **Skalierung** können durch **Matrixmultiplikation** ausgedrückt werden
- Zur Erstellung der akkumulierten Matrix benötigen wir eine entsprechende Darstellung der **Translation** (also einer **Vektoraddition**)
- Wie müsste eine solche Matrix aussehen, sodass gilt:

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \end{pmatrix} = \begin{pmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \end{pmatrix}$$

- Eine solche 3x3 Matrix existiert leider nicht.
- Lösung: **Homogene Koordinaten**

# Homogene Koordinaten II

- Mathematischer Trick: Hinzufügen einer 4. Koordinate!

$$\mathbb{R}^3 \ni \begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \in \mathbb{P}^3 \ ,$$

- Homogenisierung (zurück in den euklidischen Raum)

$$\begin{pmatrix} X \\ Y \\ Z \\ W \end{pmatrix} = \begin{pmatrix} X/W \\ Y/W \\ Z/W \end{pmatrix} \ , \forall W \neq 0$$

- Teilen durch die 4. Koordinate

# Homogene Koordinaten in 2D

- Die Vektoren $\overrightarrow{x'}$, $\overrightarrow{y'}$ und $\vec{t}$ werden zu einer Matrix T (Transformationsmatrix) zusammengefasst

$$\begin{pmatrix} p'_1 \\ p'_2 \end{pmatrix} = \begin{pmatrix} x_1' & y_1' \\ x_2' & y_2' \end{pmatrix} \cdot \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \quad \Leftrightarrow \quad \begin{pmatrix} p'_1 \\ p'_2 \\ 1 \end{pmatrix} = \begin{pmatrix} x_1' & y_1' & t_1 \\ x_2' & y_2' & t_2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}$$

Homogene Koordinaten

# Homogene Koordinaten in 2D

- Die Vektoren $\overrightarrow{x'}$, $\overrightarrow{y'}$ und $\vec{t}$ werden zu einer Matrix T (Transformationsmatrix) zusammengefasst

$$\begin{pmatrix} p'_1 \\ p'_2 \\ 1 \end{pmatrix} = \begin{pmatrix} x_1' & y_1' & t_1 \\ x_2' & y_2' & t_2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix} \Leftrightarrow \begin{pmatrix} p'_1 \\ p'_2 \\ 1 \end{pmatrix} = \begin{pmatrix} x_1' \cdot p_1 + y_1' \cdot p_2 + t_1 \cdot 1 \\ x_2' \cdot p_1 + y_2' \cdot p_2 + t_2 \cdot 1 \\ 0 + 0 + 1 \end{pmatrix}$$

Multiplikativer Teil

Additiver Teil

Homogene Koordinaten

# Übersicht aller Transformationen (in 4D)

- Rotation:

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

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

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

- Skalierung:

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

- Translation:

$$T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

# Homogene Koordinaten in 2D

- Die Vektoren $\overrightarrow{x'}$, $\overrightarrow{y'}$ und $\vec{t}$ werden zu einer Matrix T (Transformationsmatrix) zusammengefasst

$$\begin{pmatrix} p'_1 \\ p'_2 \\ 1 \end{pmatrix} = \begin{pmatrix} x_1' & y_1' & t_1 \\ x_2' & y_2' & t_2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix} \Leftrightarrow \begin{pmatrix} p'_1 \\ p'_2 \\ 1 \end{pmatrix} = \begin{pmatrix} x_1' \cdot p_1 + y_1' \cdot p_2 + t_1 \cdot 1 \\ x_2' \cdot p_1 + y_2' \cdot p_2 + t_2 \cdot 1 \\ 0 + 0 + 1 \end{pmatrix}$$

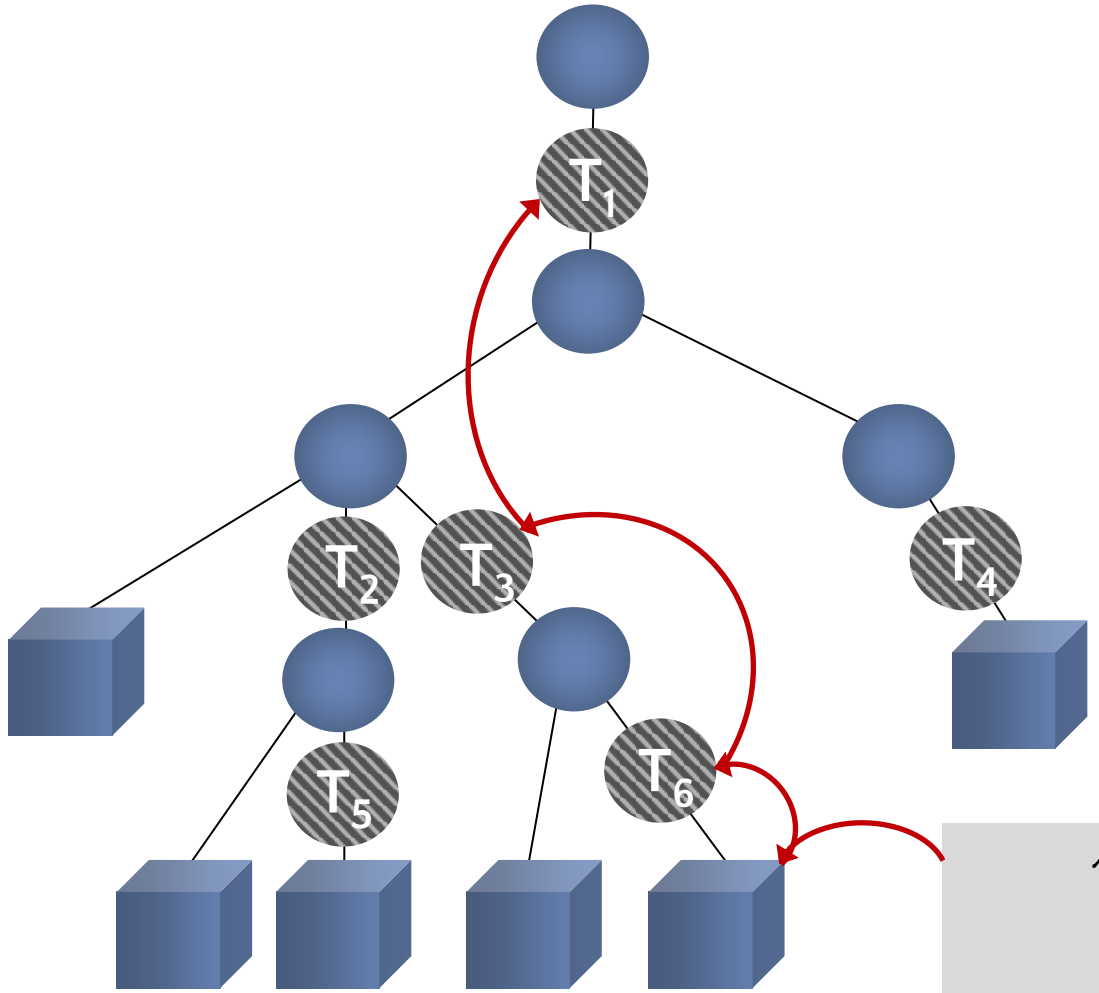
Multiplikativer Teil

Additiver Teil

Homogene Koordinaten

Nutzt man den multiplikativen und gleichzeitig den additiven Teil der Matrix, wird zuerst die multiplikative Transformation und danach die Additive Transformation ausgeführt.

# Implementierung des Szenegraphen




- In welcher Reihenfolge müssen die Transformationen des Szenegraphen berücksichtigt werden?
- Die Transformationen werden in der Reihenfolge, wie in der Abbildung links dargestellt, auf die jeweilige Geometrie angewendet.
- Die Kind-Objekte erben die Transformationen ihrer Eltern: $T_1$ wirkt also auf alle darunterliegende Zweige des Szenengraphen und zwar immer als letzte Transformation.

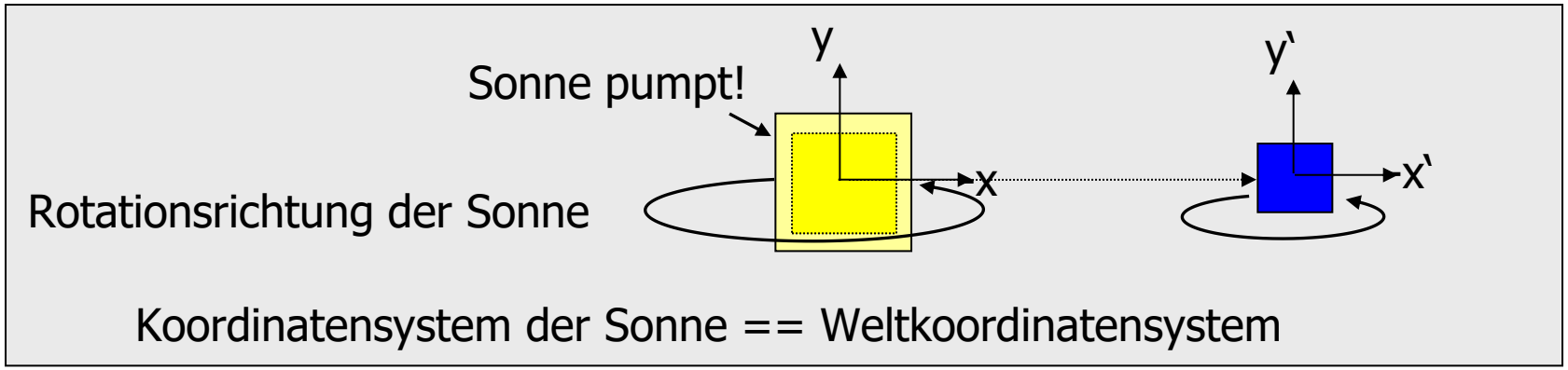
*Angewendete Transformationen:*

$$T_1 * T_3 * T_6 * P$$

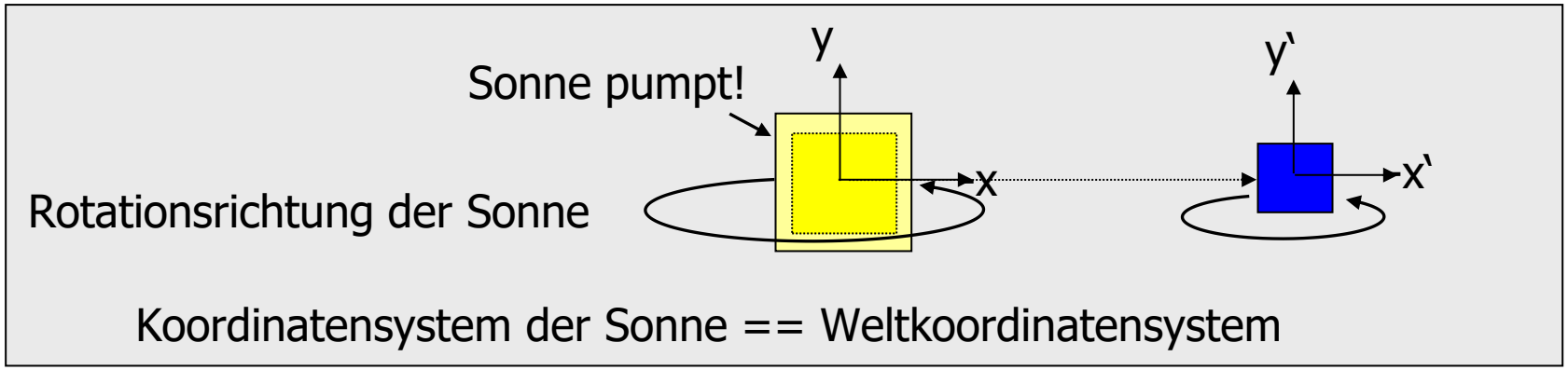
# Szenegraph – Beispiel

## Aufgabe: Miniatursonnensystem

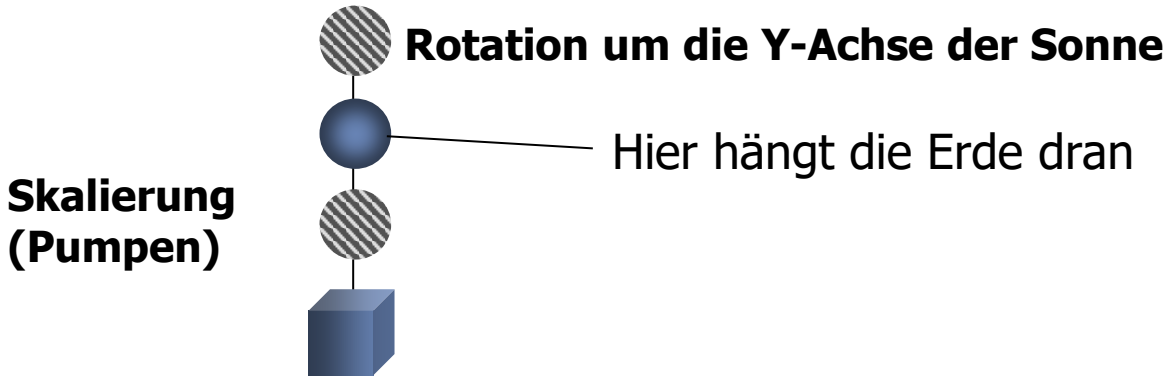
- Sonne rotiert um ihre Y-Achse.
- Während sie rotiert, pumpt sie sich auf und fällt wieder zusammen.
- Die Erde rotiert in einiger Entfernung mit gleicher Winkel-geschwindigkeit (d.h. überstrichener Winkel pro Zeiteinheit ist gleich) um die Y-Achse der Sonne.
- Zusätzlich rotiert die Erde um ihre eigene Y-Achse.
- Erde und Sonne sind eckig ;-)



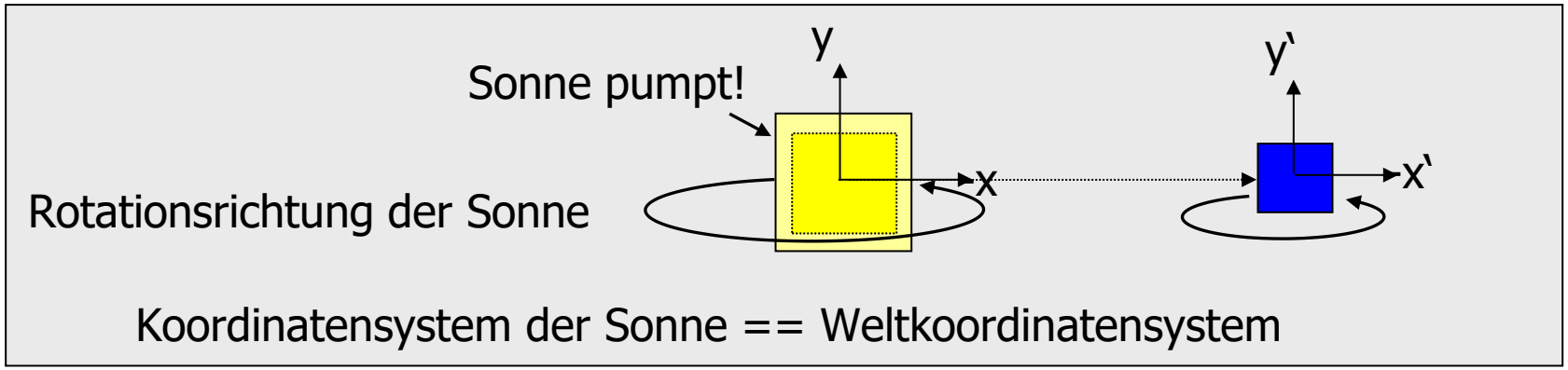
# Szenegraph – Beispiel



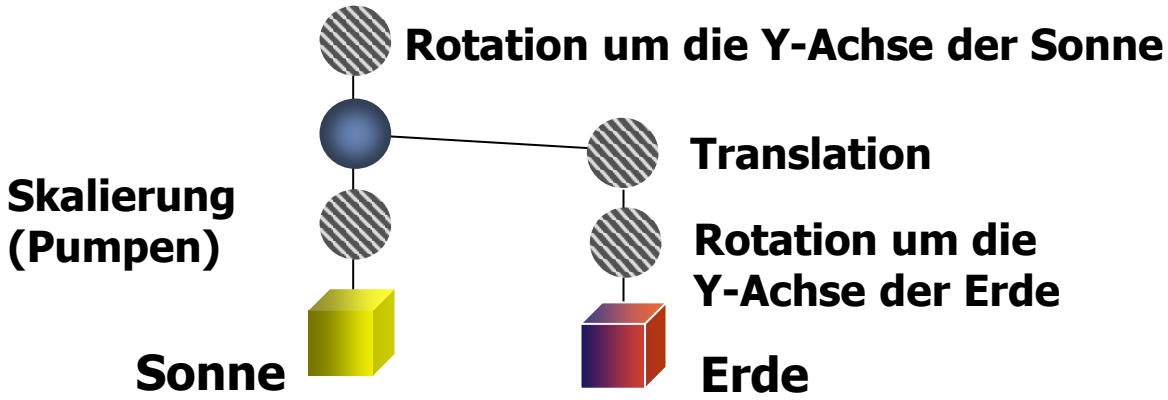

- Sonne und Erde rotieren mit gleicher Winkelgeschwindigkeit um die Y-Achse der Sonne
- Aber nur die Sonne pumpt sich auf und fällt zusammen



# Szenegraph – Beispiel



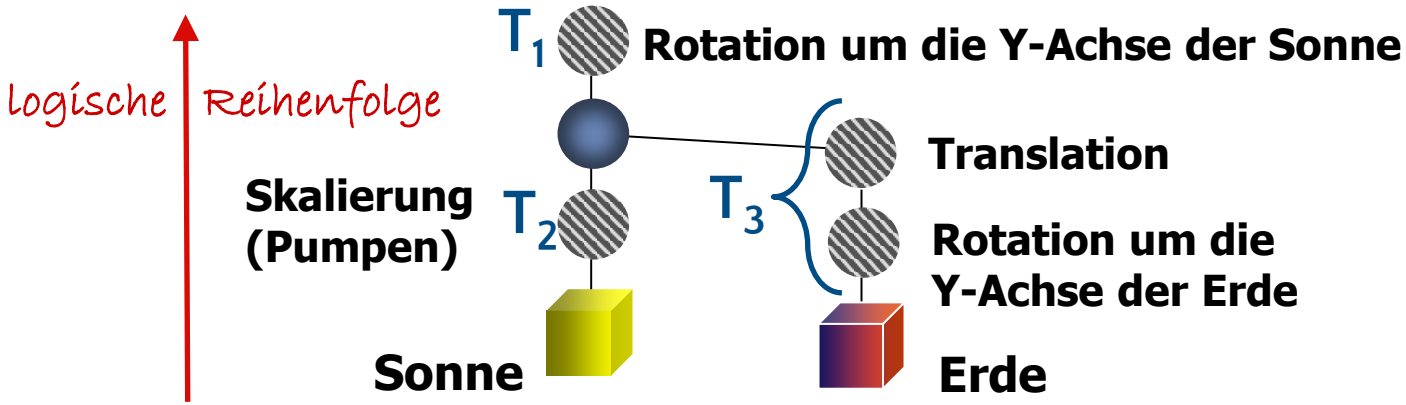

- Erde rotiert zusätzlich um ihre eigne Y-Achse



# Szenegraph – Beispielimplementierung

## Pseudocode:

```
Matrix4f T1 = new Matrix4f();          // neue Matrix T1 anlegen
T1.rotate( alpha, 0.0, 1.0, 0.0);      // Rotation mit Winkel alpha um den Vektor (0,1,0)^t

Matrix4f T2 = new Matrix4f();          // neue Matrix T2 anlegen
T2.scale( s, s, s);                    // gleichmäßige Skalierung um Faktor s

Matrix4f T3 = new Matrix4f();          // neue Matrix T3 anlegen
T3.translate(p1, p2, p3);              // Translation relativ zur Sonne, Position (p1, p2, p3)
T3.rotate( beta, 0.0, 1.0, 0.0);       // Rotation mit Winkel beta um den Vektor (0,1,0)^t
```



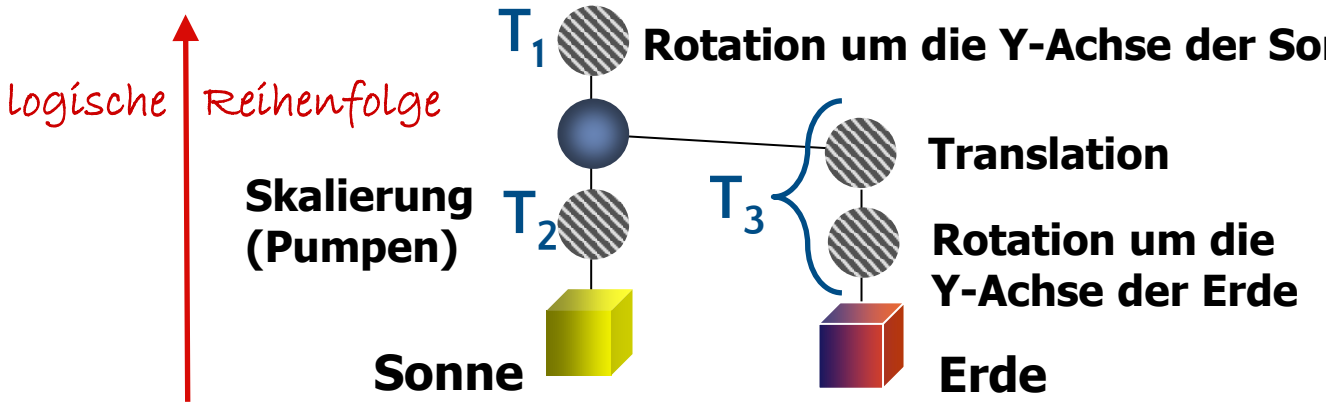

Akkumulierte Matrix für..

die Erde: $M_E$ = T1* T3;

die Sonne: $M_S$ = T1* T2;

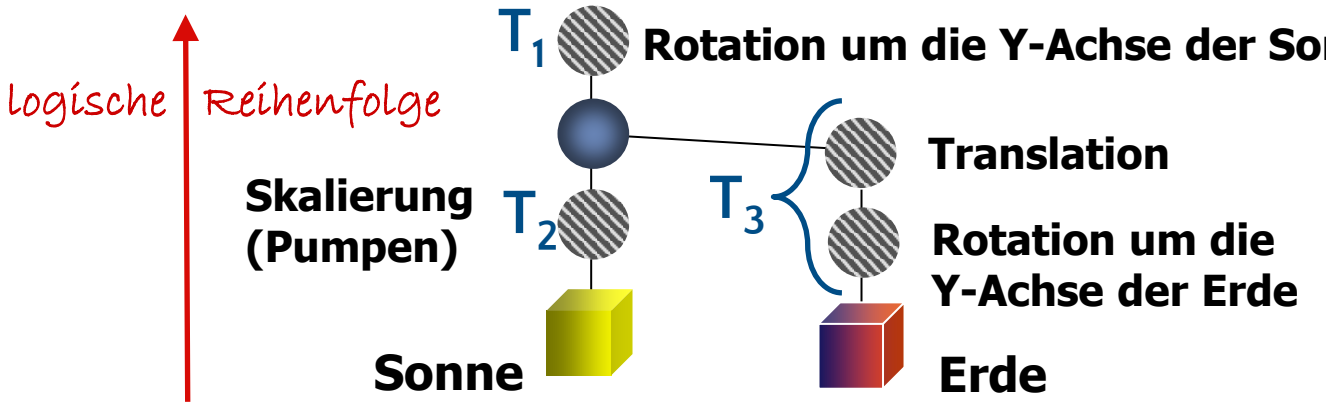
# Auszug aus `void Scene::render(float dt)` in `Scene.cpp`

```
// Zunächst werden die Matrizen T1, T2 und T3 mit den passenden Werten befüllt
T1 = new Transform; // Erzeugt Einheitsmatrix
T1->rotate(glm::vec3(0, 0.2 * dt, 0)); // Rotation im die Y-Achse.
                                       // Der Winkel erhöht sich mit durch dt

T2 = new Transform; // Erzeugt Einheitsmatrix
T2->scale(glm::vec3(scaling*dt, scaling*dt, scaling*dt)); // dt ändert Skalierung

T3 = new Transform; // Erzeugt Einheitsmatrix
T3->translate(glm::vec3(0.8f, 0, 0)); // einmalige Translation der Erde
T3->rotate(glm::vec3(0, 0.4f*dt, 0));
```



# Codeauszug aus Scene.cpp

```
//Laden der Shader in: bool Scene::init()
m_assets.addShaderProgram("shader",AssetManager::createShaderProgram(„…vertex.glsl",
                                                                      „…/fragment.glsl"));
m_shader = m_assets.getShaderProgram("shader");
m_shader->use();
// In: void Scene::render(float dt)
// Implementierung des linken Szenengraphzweigs - Sonne: T1*T2
m_shader->setUniform("mm", T1->getTransformMatrix() * T2->getTransformMatrix(), false);
// Implementierung des linken Szenengraphzweigs – Erde : T1*T3
m_shader->setUniform("mm", T1->getTransformMatrix() * T3->getTransformMatrix(), false),
```




Das Akkumulieren der Matrizen geschieht durch die Uniform-Parameter – siehe Kapitel 6

die Erde: $M_E$ = T1* T3;

die Sonne: $M_S$ = T1* T2;