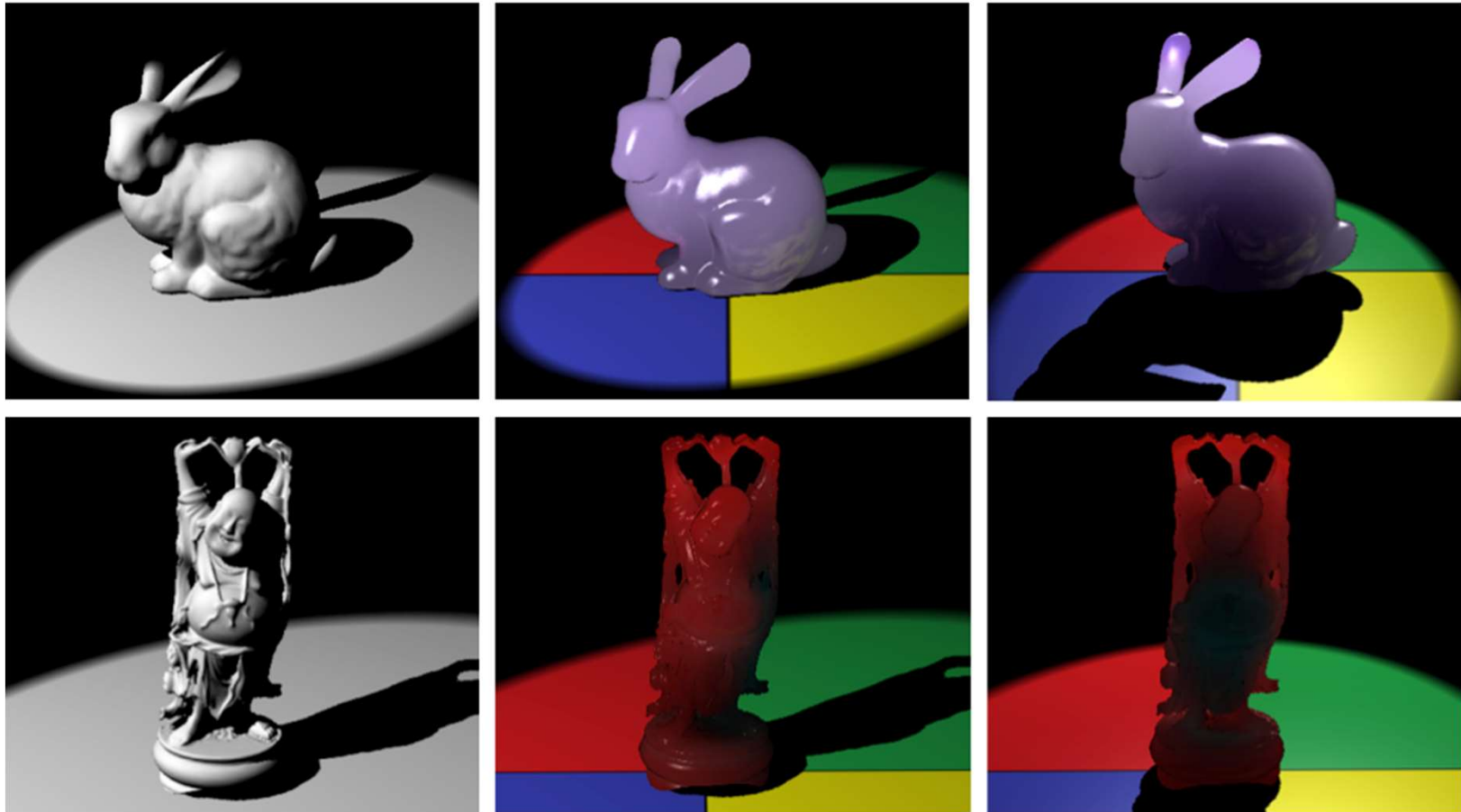
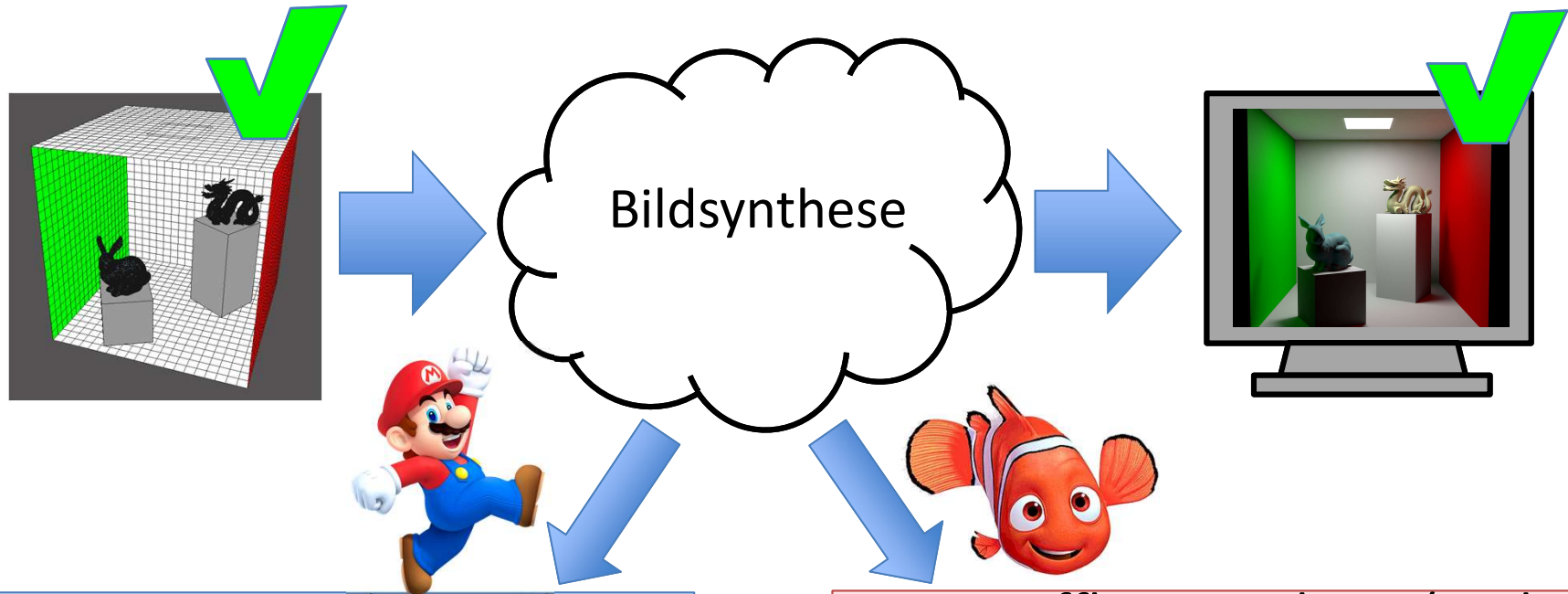


Computergrafik



Vorlesung

Philipp Lensing



Echtzeitsynthese (schnell)

Rasterzeilen-
Verfahren



OpenGL/Direct3D
Bibliothek

Hardware
Renderpipeline

Shader



Proj
Transformationen



Beleuchtungsmodelle

Szen
ment



Postprocessing



Offline-Synthese (exakt)

Raytracing

Photon Map

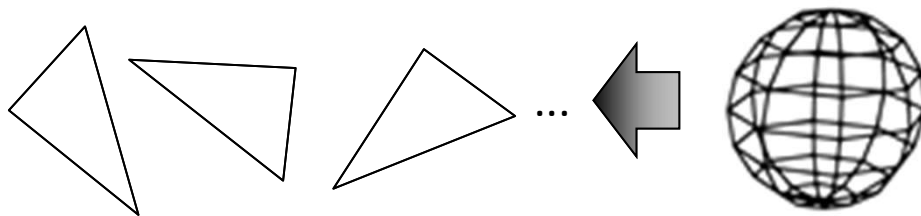


Radiosi

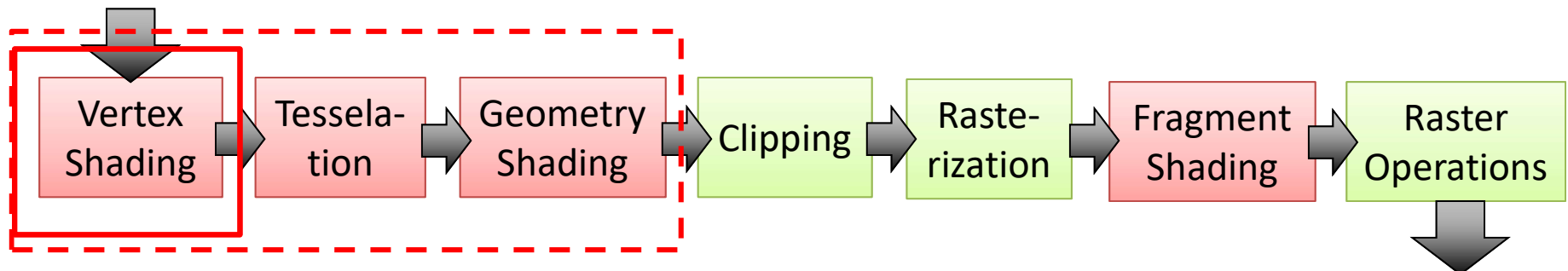


OpenGL-Rendering-Pipeline

Überblick der einzelnen Stufen der OpenGL-Rendering-Pipeline:



Der Pipeline werden die Geometriedaten etc. übergeben.

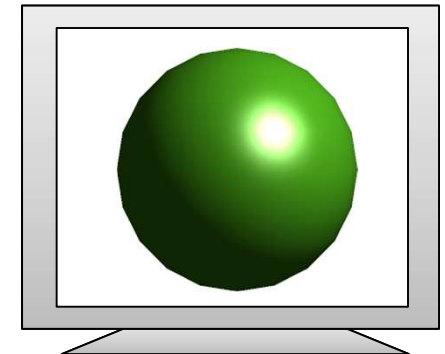


Affine & projektive Transformation von Vertices.

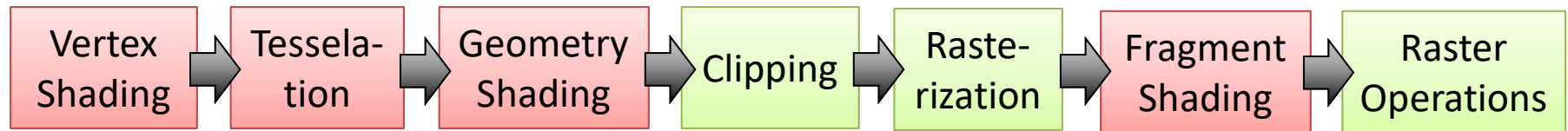
 Programmierbare Stufen

 Parametrisierbare Stufen

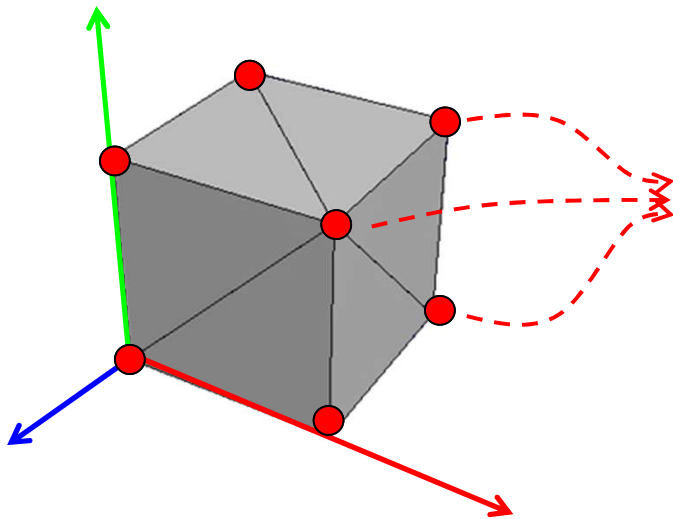
Ausgang der Pipeline ist das gerasterte Bild.



Rendering-Pipeline: Vertex Shading

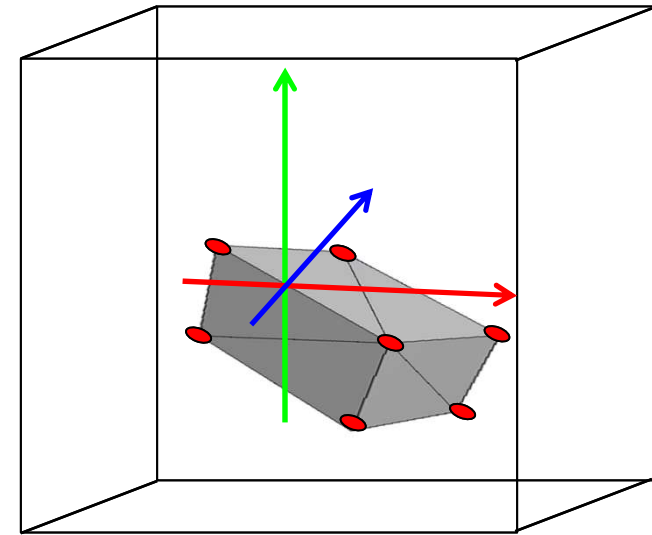


Eingang: Vertex-Attribute im Modell-Raum z. B. Position, Normale, etc.



Vertex Shading

Ausgang: Vertex-Attribute im Bildraum.

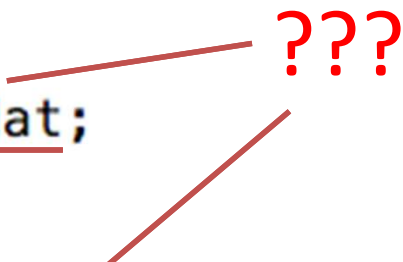


Beim Vertex-Shading werden die Eckpunkte des Modells (Vertizes) affin und projektiv transformiert. Dabei durchlaufen die Vertizes eine **Transformations-Pipeline**.

Vertex Shader

- Vertex-ConstantShader aus Praktikum (ConstantShader.cpp):

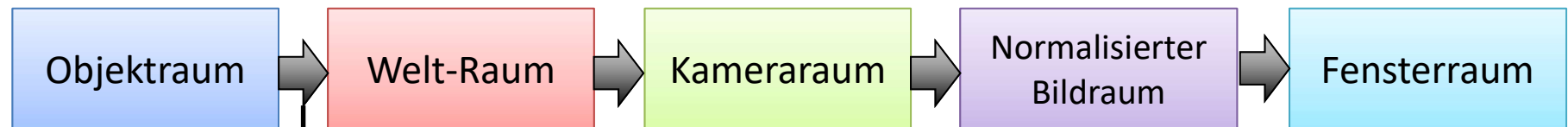
```
#version 400
in vec4 VertexPos;
uniform mat4 ModelViewProjMat;
void main()
{
    gl_Position = ModelViewProjMat * VertexPos;
}
```



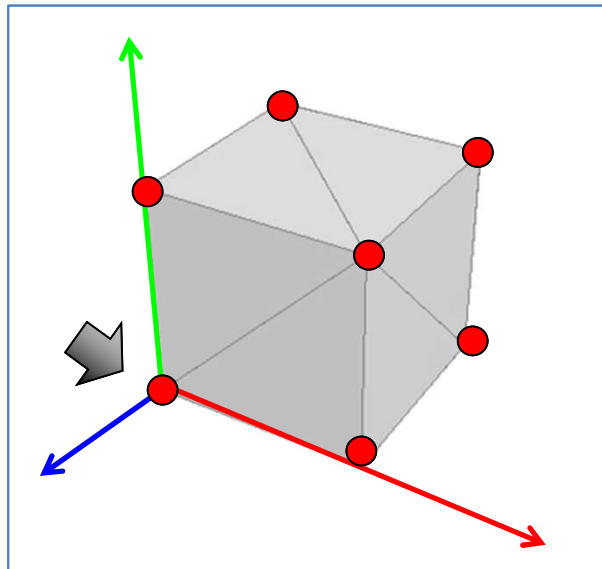
- Eine (kombinierte) Matrix transformiert die Modell-Vertizes (aus Vertexbuffer) in den korrekten Bildraum!
 - Berücksichtigt werden:
 - Modell-Position und -Ausrichtung (*Model*)
 - Kamera-Position und -Ausrichtung (*View*)
 - Kamera-Perspektive (*Proj*)

Transformations-Pipeline

Ein Vertex durchläuft nacheinander unterschiedliche Transformationsräume:



Objektraum



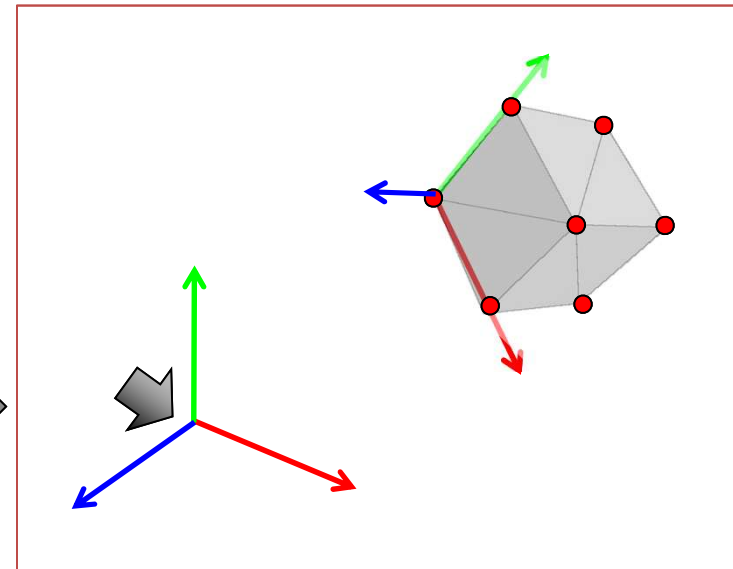
Raum, in dem das Objekt definiert wurde (mit 3D-Modellierungs-programm).

Affine Transform.:

- Verschiebung
- Rotation
- Skalierung

Modell-Transform.
(Welt-Transform.)

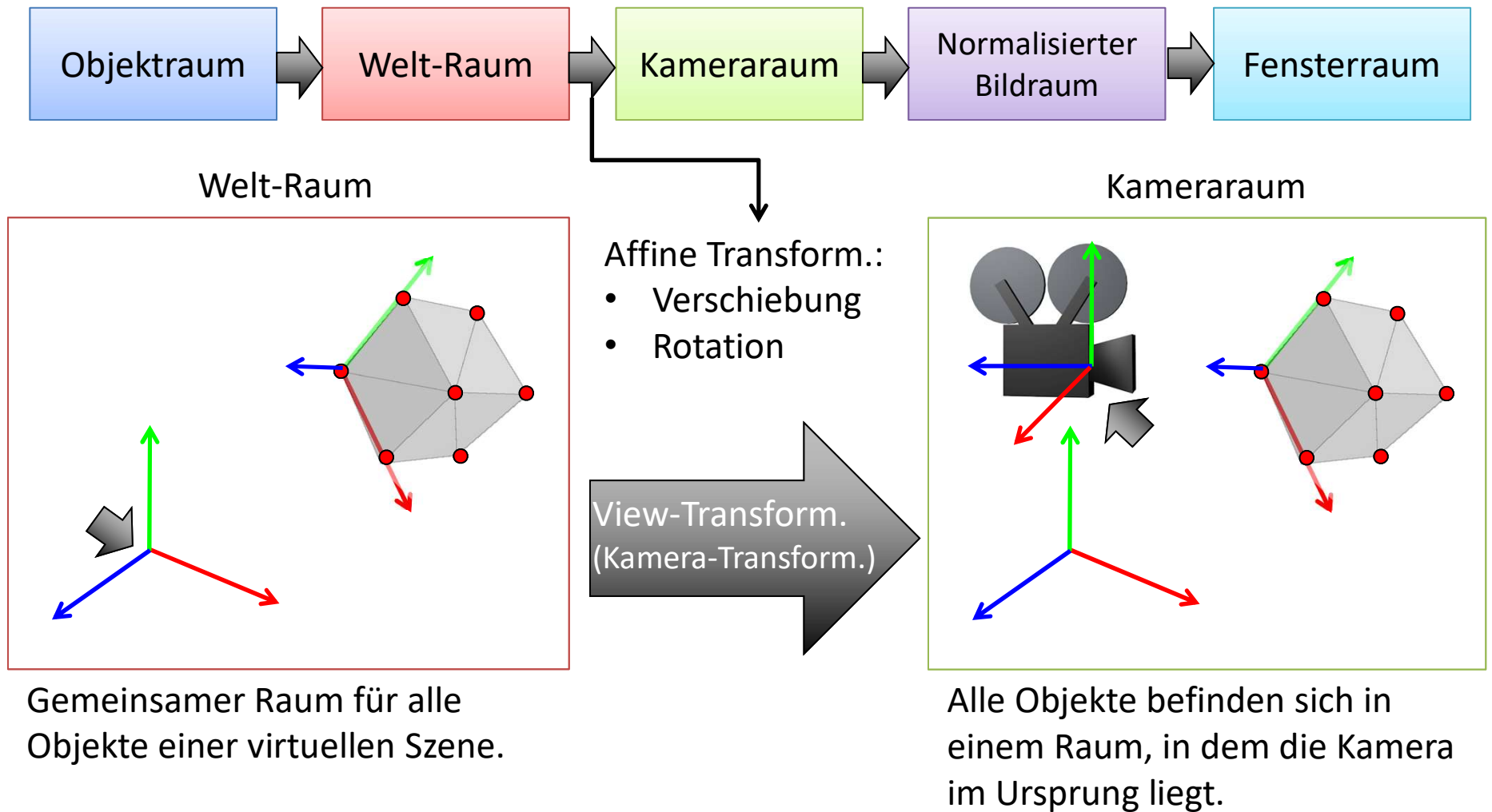
Welt-Raum



Gemeinsamer Raum für alle Objekte einer virtuellen Szene.

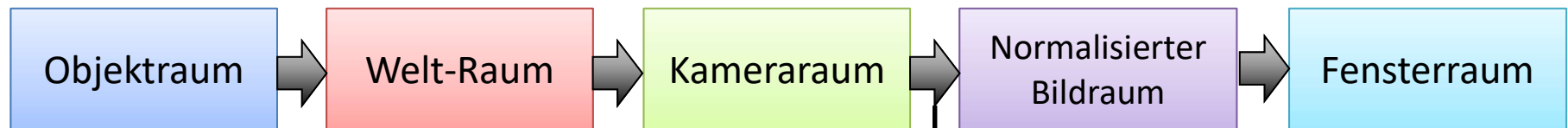
Transformations-Pipeline

Ein Vertex durchläuft nacheinander unterschiedliche Transformationsräume:

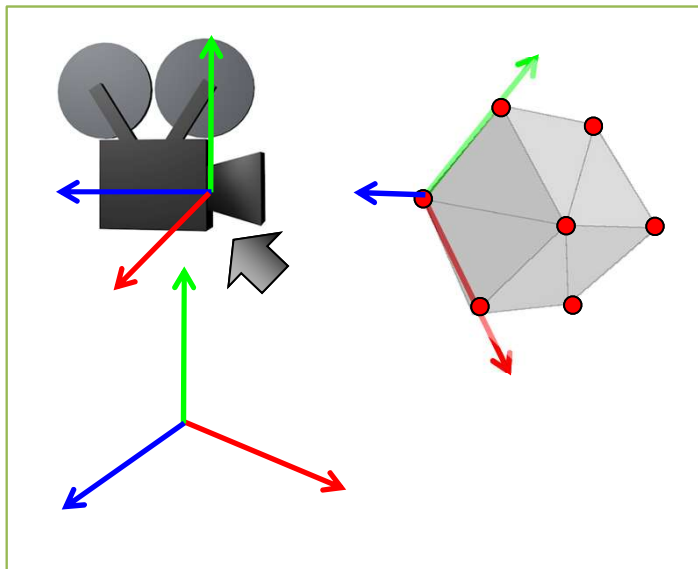


Transformations-Pipeline

Ein Vertex durchläuft nacheinander unterschiedliche Transformationsräume:



Kameraraum

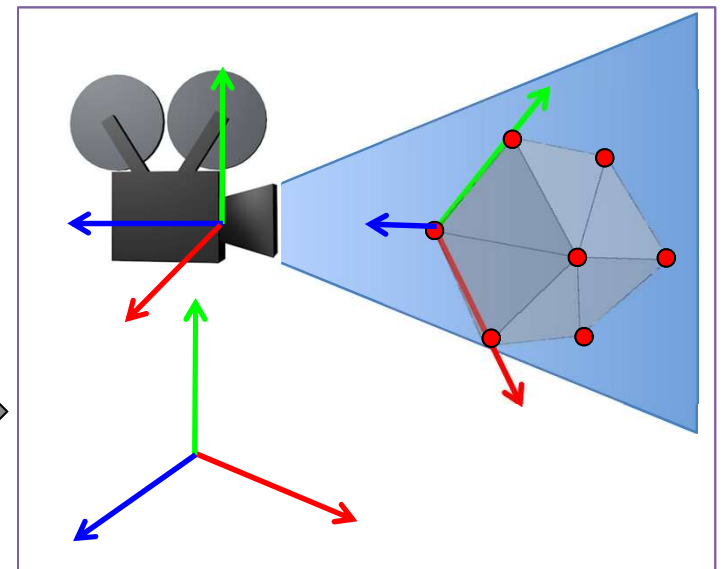


Alle Objekte befinden sich in einem Raum, in dem die Kamera im Ursprung liegt.

Projektive Transform.:
• Orthogonal
• Perspektivisch

Projektionstranf.

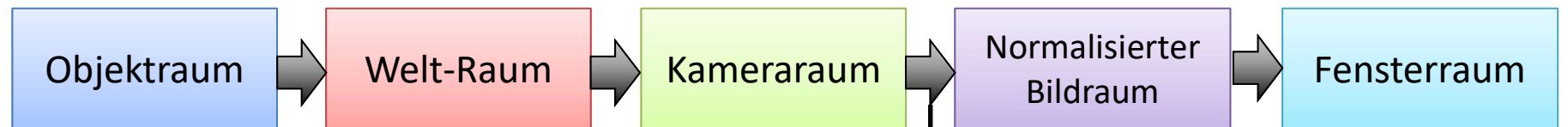
Clipping-Raum



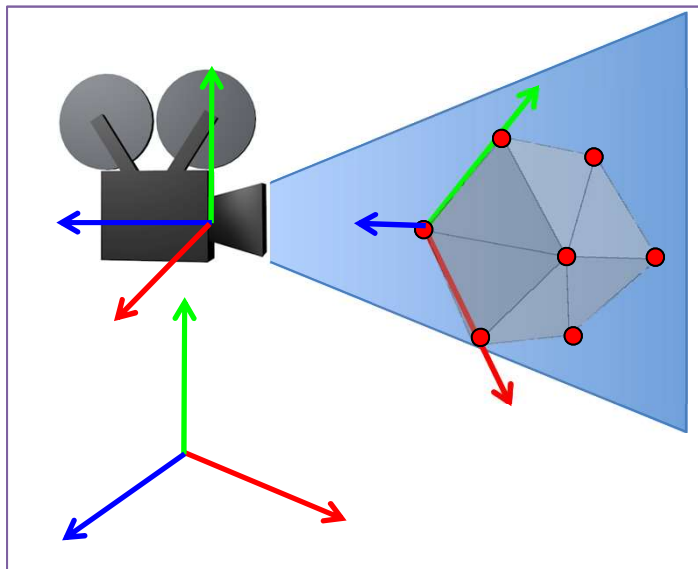
Der Clipping-Raum ist eine Vorstufe zum normalisierten Bildraum (ohne persp. Teilung)

Transformations-Pipeline

Ein Vertex durchläuft nacheinander unterschiedliche Transformationsräume:



Clipping-Raum



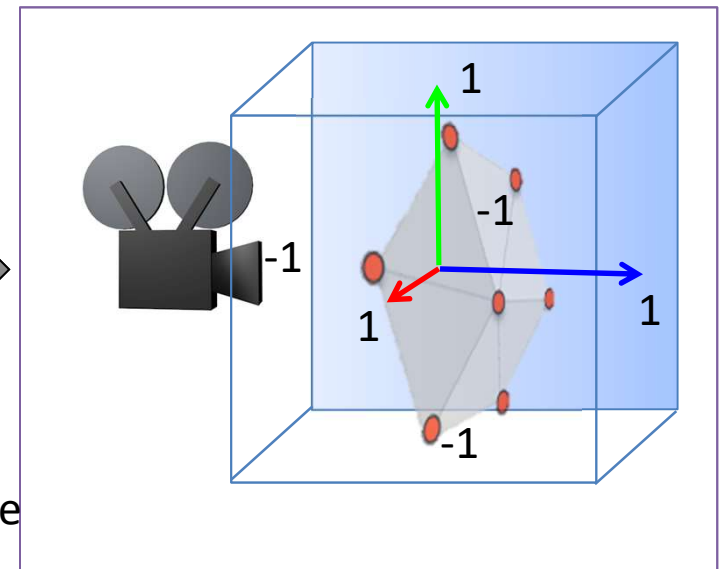
Der Clipping-Raum ist eine Vorstufe zum normalisierten Bildraum (ohne persp. Teilung).

Perspektivische
Teilung

Projektionstranf.

(Nahe Objekte
werden größer, ferne
kleiner)

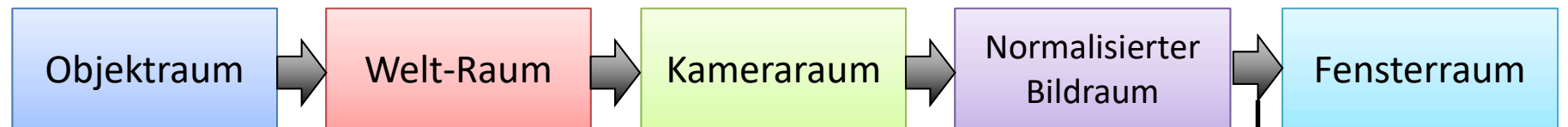
Normalisierter Bildraum



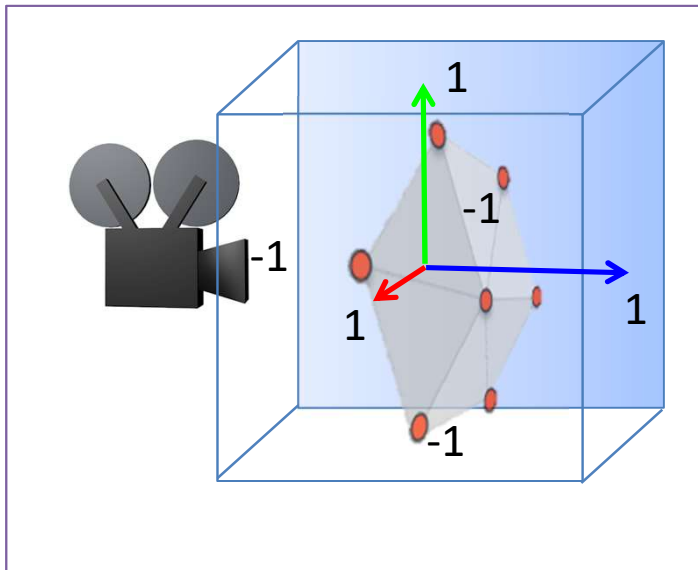
Nach der persp. Teilung befinden sich alle Vertizes in einem Würfel mit den Kantenlängen 2,2,2.

Transformations-Pipeline

Ein Vertex durchläuft nacheinander unterschiedliche Transformationsräume:

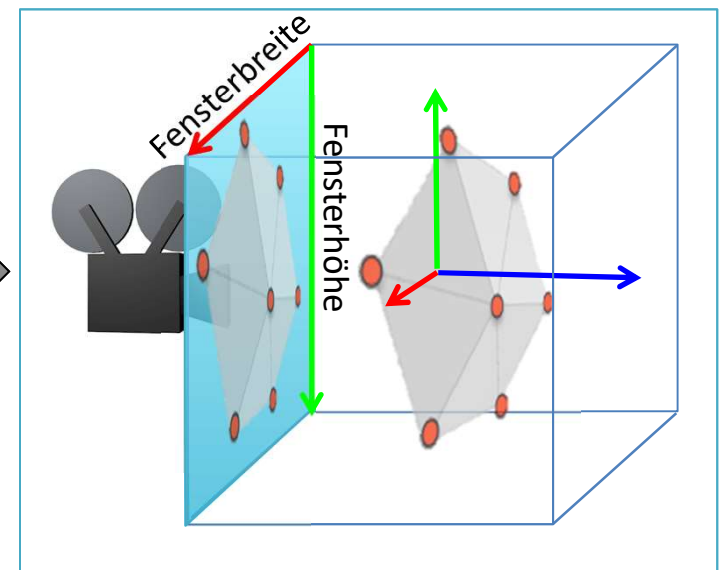


Normalisierter Bildraum



Nach der persp. Teilung befinden sich alle Vertizes in einem Würfel mit den Kantenlängen 2,2,2.

Fensterraum (Viewport)



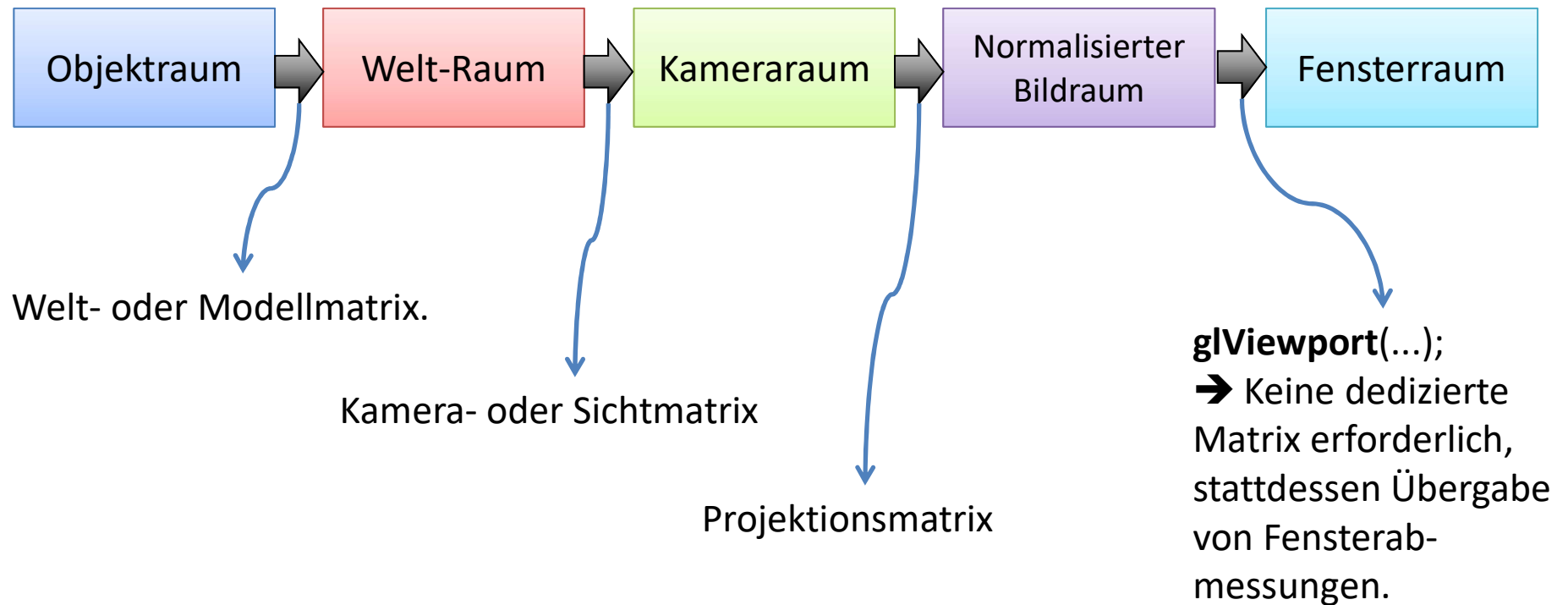
Im Fensterraum werden die normalisierten Modelldaten in Pixelkoordinaten des Fensters transformiert.

Umrechnung in
Fenster-Pixelkoord.

Viewport-Transf.

Transformations-Pipeline mit OpenGL

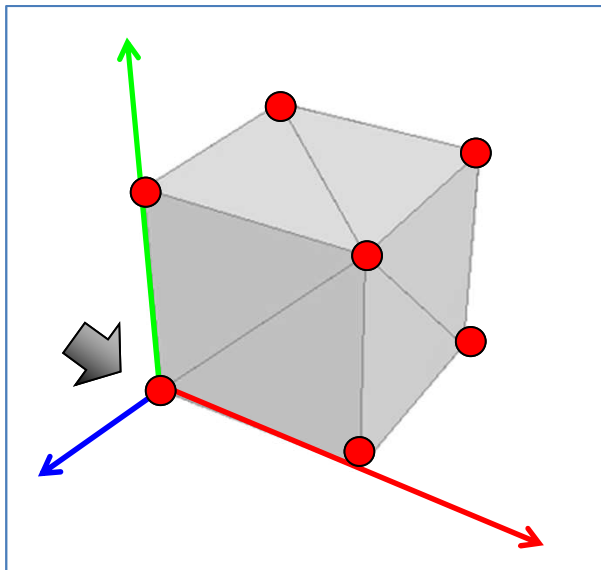
Ein Vertex durchläuft nacheinander unterschiedliche Transformationsräume.
Bei OpenGL werden die Transformationen für die Räume wie folgt angegeben:



Modell-Transformation (Welt-Transformation)

Nachfolgend soll erläutert werden, wie ein Objekt durch Verschiebung, Rotation und Skalierung im virtuellen Welt-Raum beschrieben wird.

Objektraum

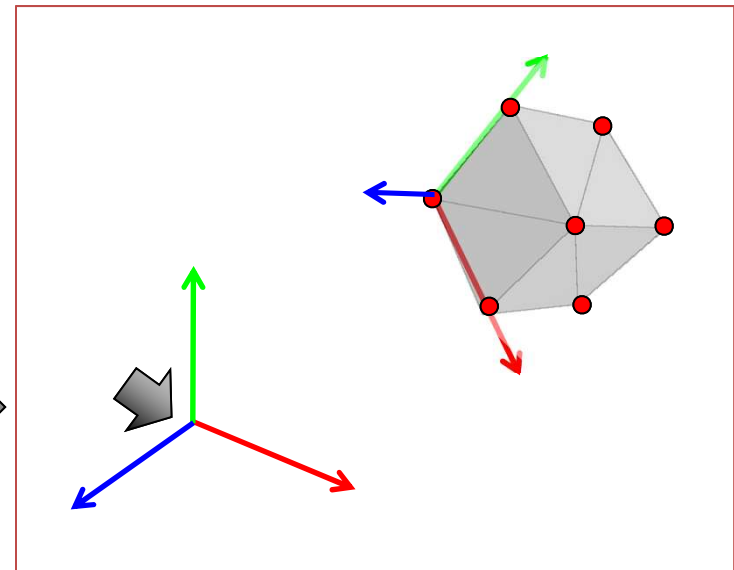


Affine Transform.:

- Verschiebung
- Rotation
- Skalierung

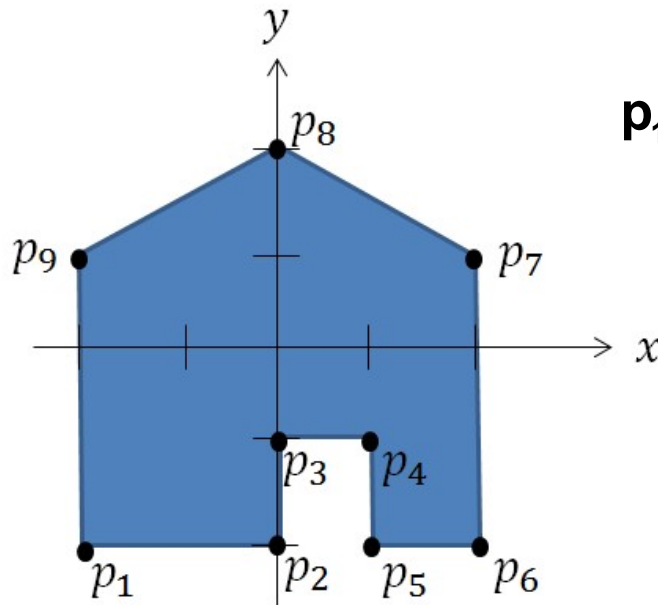
Modell-Transform.
(Welt-Transform.)

Welt-Raum



Affine Transformationen

- Zur Verringerung der Komplexität wollen wir vorerst alle Transformationen im Zweidimensionalen betrachten
 - Verschiebung (Translation)
 - Skalierung
 - Rotation
- Als Grundlage dient ein einfaches Modell eines „Hauses“
- Die Form des Hauses wird durch seine 9 Vertices in der XY-Ebene beschrieben.

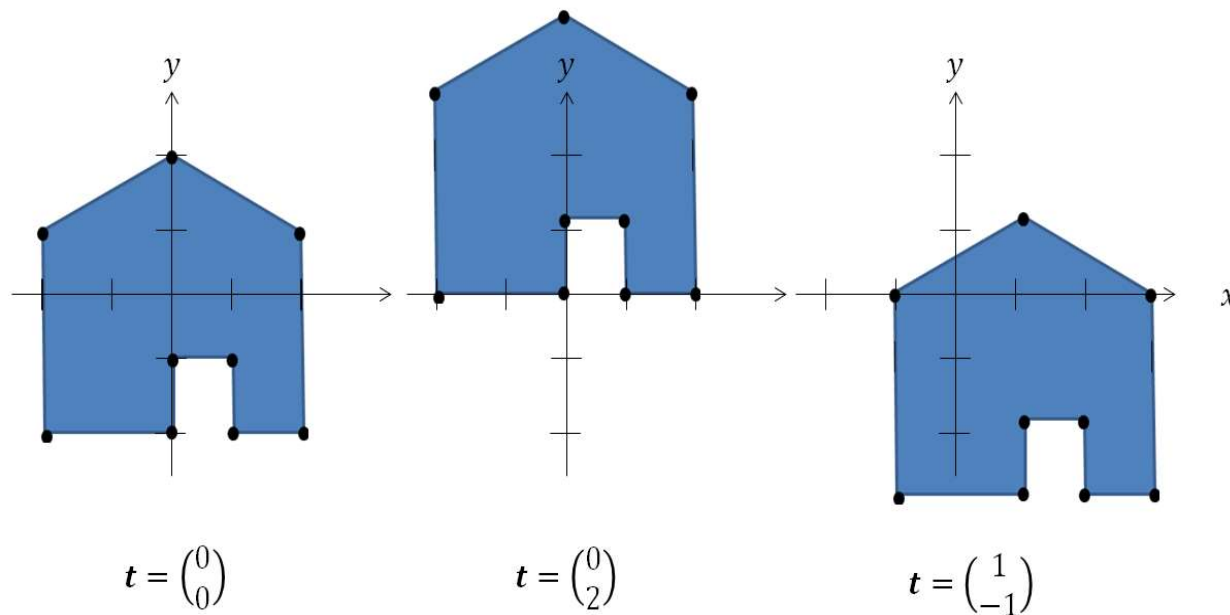


$$\mathbf{p}_1 = \begin{pmatrix} -2 \\ -2 \end{pmatrix}, \mathbf{p}_2 = \begin{pmatrix} 0 \\ -2 \end{pmatrix}, \mathbf{p}_3 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \dots$$

$$\mathbf{p}_i \in \mathbb{R}^2$$

Verschiebung (Translation)

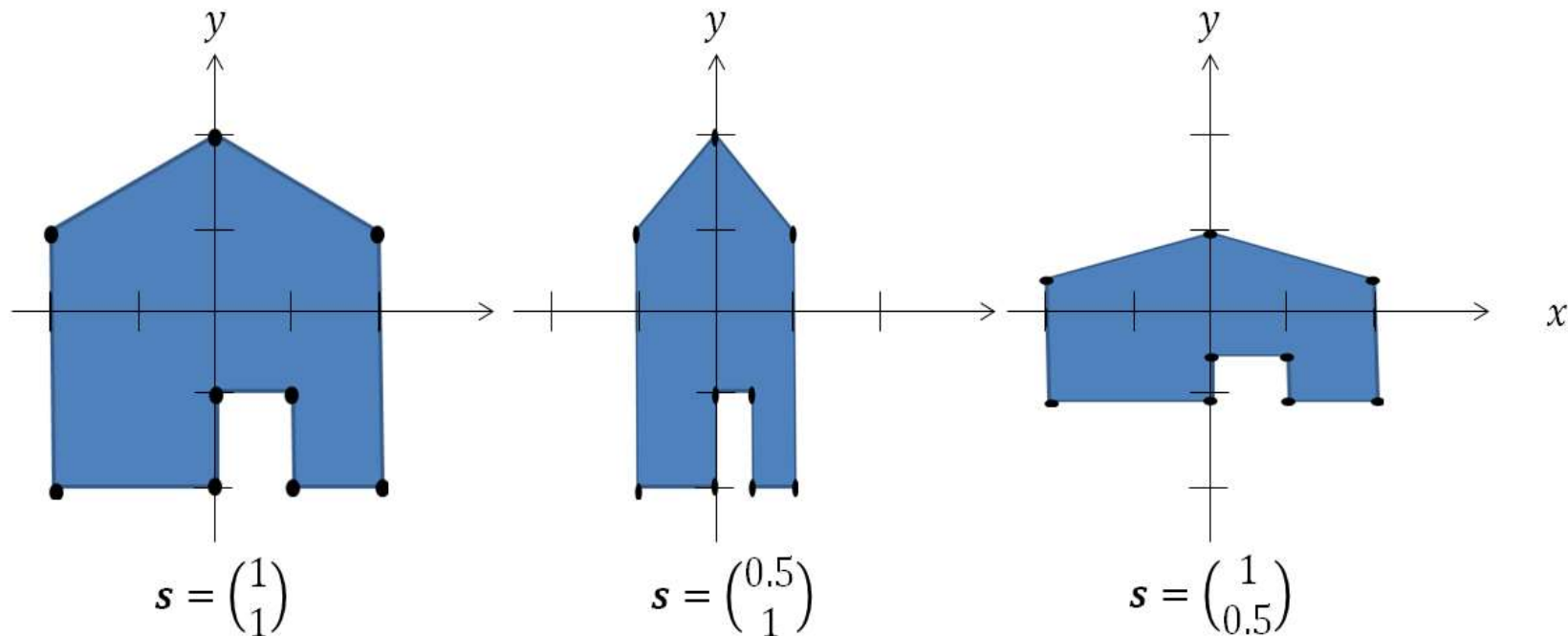
- Das Haus soll verschoben werden. Welche Operation muss auf die Eckpunkte des Hauses angewendet werden, damit die folgenden Verschiebungen resultieren?



$$T(\mathbf{p}_i, \mathbf{t}) = \mathbf{p}_i + \mathbf{t} = \begin{pmatrix} x_{pi} \\ y_{pi} \end{pmatrix} + \begin{pmatrix} x_t \\ y_t \end{pmatrix} = \begin{pmatrix} x_{pi} + x_t \\ y_{pi} + y_t \end{pmatrix}$$

Skalierung

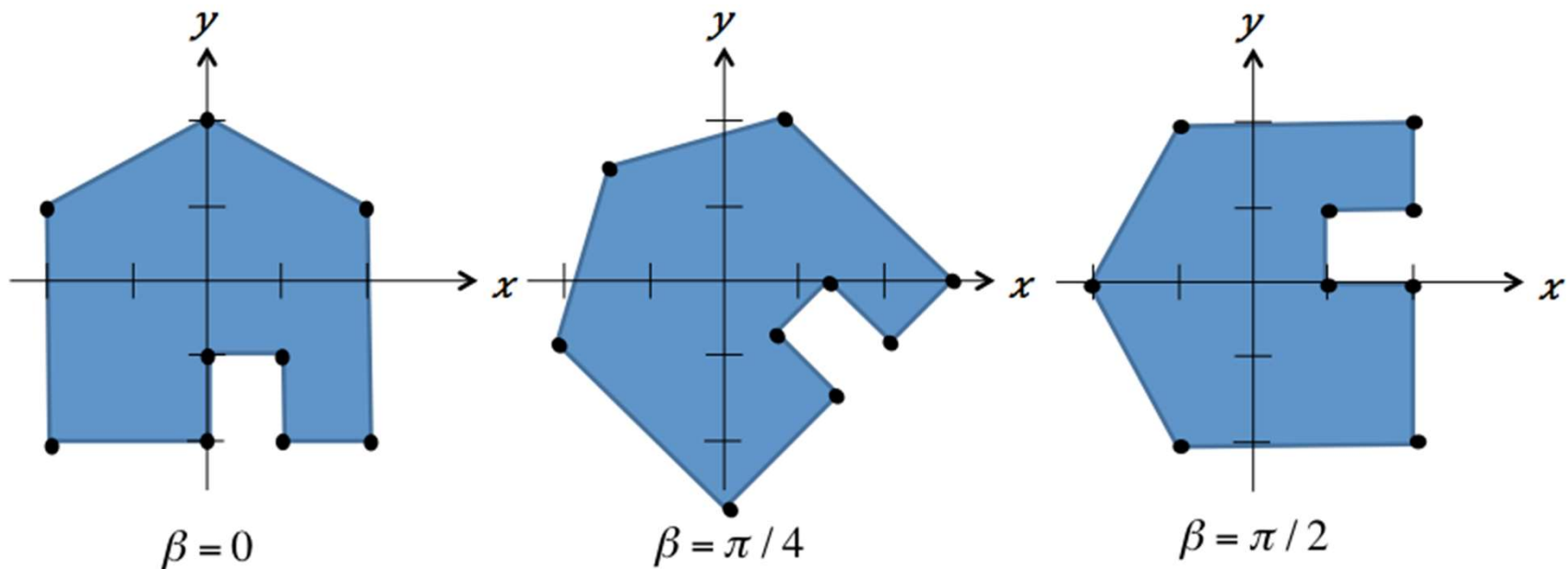
- Das Haus soll skaliert werden. Welche Operation muss auf die Eckpunkte des Hauses angewendet werden, damit die folgenden Skalierungen resultieren?



$$S(p_i, s) = sp_i = \begin{pmatrix} x_s x_{pi} \\ y_s y_{pi} \end{pmatrix}$$

Rotation

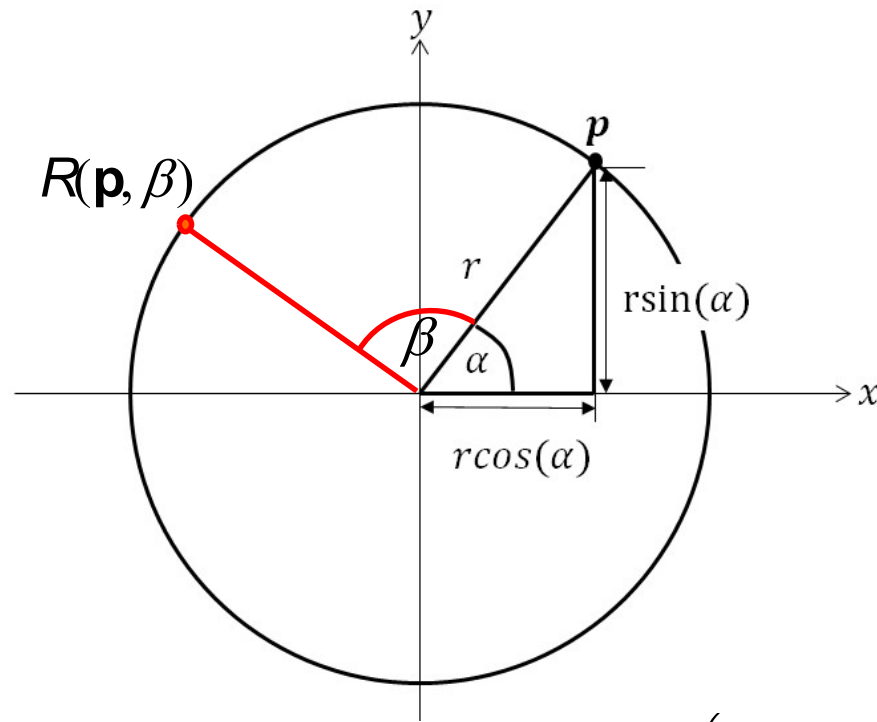
- Das Haus soll rotiert werden. Welche Operation muss auf die Eckpunkte des Hauses angewendet werden, damit die folgenden Rotationen resultieren?



$$R(\mathbf{p}_i, \beta) = ?$$

Rotation

- Um ein Objekt zu rotieren, macht es Sinn, die kartesischen Koordinaten als Funktion eines Winkels und eines Radius zu interpretieren:



$$\mathbf{p}_i = \begin{pmatrix} x_{pi} \\ y_{pi} \end{pmatrix} = \begin{pmatrix} r_i \cos(\alpha_i) \\ r_i \sin(\alpha_i) \end{pmatrix}$$

$$R(\mathbf{p}_i, \beta) = \begin{pmatrix} r_i \cos(\alpha_i + \beta) \\ r_i \sin(\alpha_i + \beta) \end{pmatrix}$$

Noch etwas unschöne
Form ...

Rotation

1. Ersetzung mit Hilfe des Additionstheorems für Sinus und Cosinus

$$\sin(\alpha + \beta) = \cos(\alpha)\sin(\beta) + \sin(\alpha)\cos(\beta)$$

$$\cos(\alpha + \beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)$$

2. Wiedereinsetzung der kartesischen Koordinaten

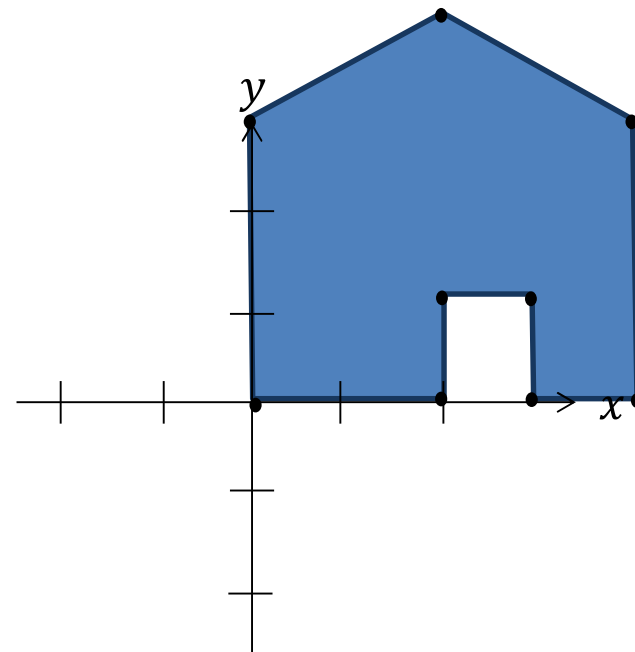
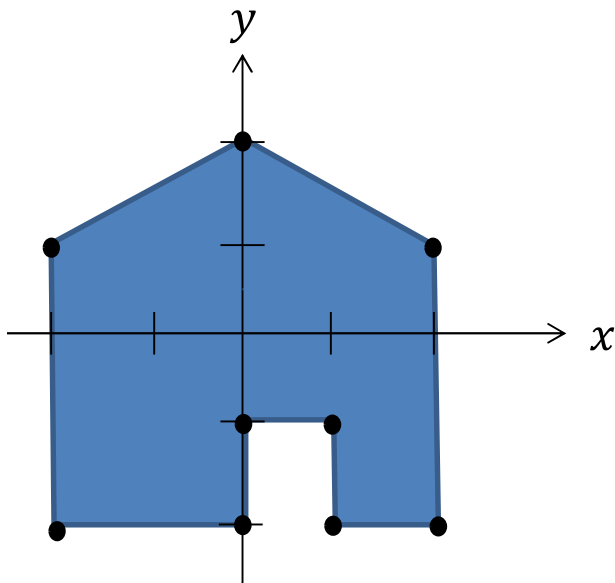
$$\mathbf{p}_i = \begin{pmatrix} r_i \cos(\alpha_i) \\ r_i \sin(\alpha_i) \end{pmatrix} = \begin{pmatrix} x_{pi} \\ y_{pi} \end{pmatrix}$$

$$R(\mathbf{p}_i, \beta) = \begin{pmatrix} r_i \cos(\alpha_i + \beta) \\ r_i \sin(\alpha_i + \beta) \end{pmatrix} = \begin{pmatrix} r_i \cos(\alpha_i) \cos(\beta) - r_i \sin(\alpha_i) \sin(\beta) \\ r_i \cos(\alpha_i) \sin(\beta) + r_i \sin(\alpha_i) \cos(\beta) \end{pmatrix}$$

$$R(\mathbf{p}_i, \beta) = \begin{pmatrix} x_{pi} \cos(\beta) - y_{pi} \sin(\beta) \\ x_{pi} \sin(\beta) + y_{pi} \cos(\beta) \end{pmatrix}$$

Rotation

WICHTIG: Rotation immer im Bezug zum Ursprung!

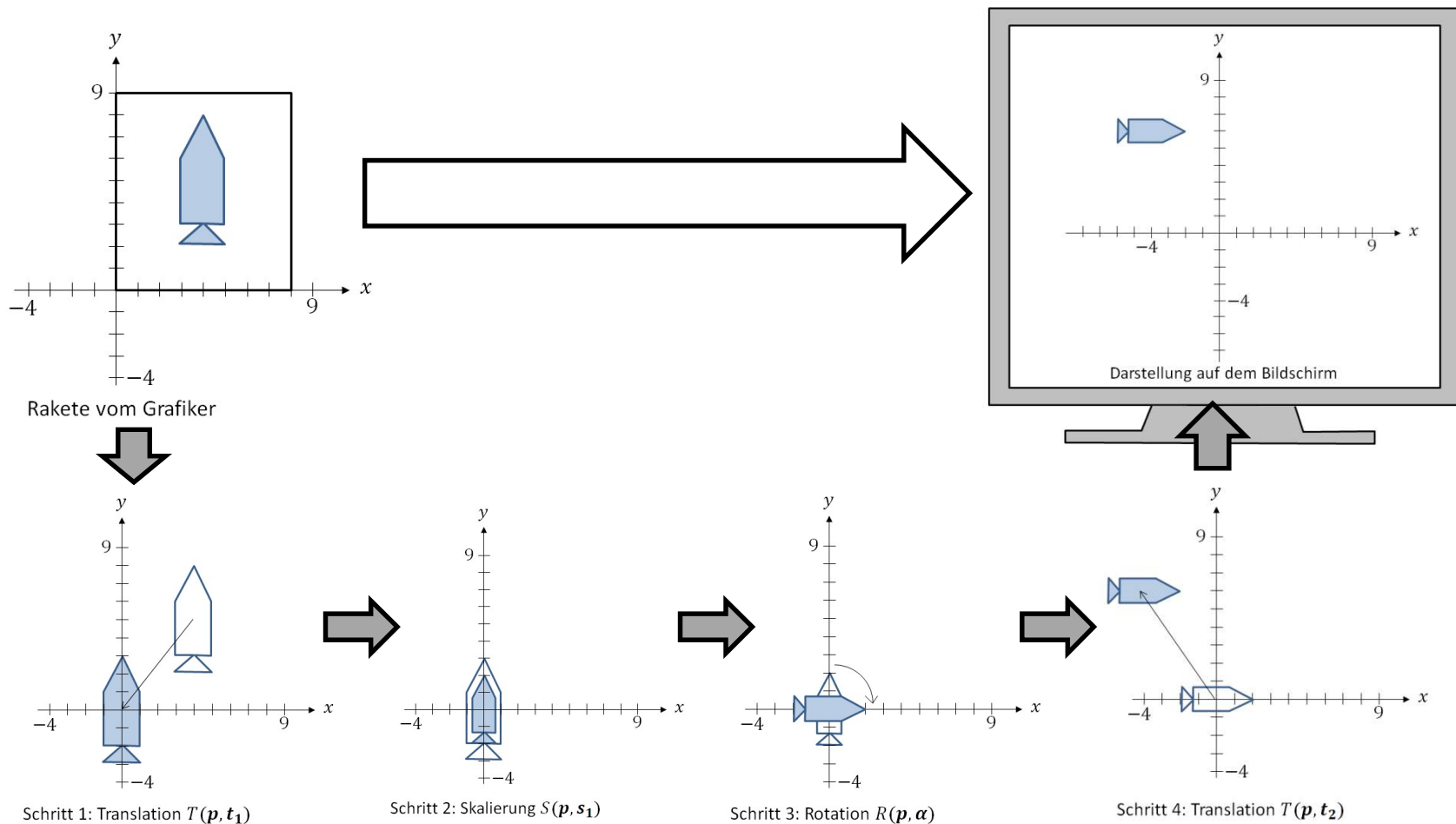


Verschiebung, Skalierung & Rotation in der Ebene

- Verschiebung: $T(\mathbf{p}_i, \mathbf{t}) = \begin{pmatrix} x_{pi} + x_t \\ y_{pi} + y_t \end{pmatrix}$
- Skalierung: $S(\mathbf{p}_i, \mathbf{s}) = \begin{pmatrix} x_s x_{pi} \\ y_s y_{pi} \end{pmatrix}$
- Rotation: $R(\mathbf{p}_i, \beta) = \begin{pmatrix} x_{pi} \cos(\beta) - y_{pi} \sin(\beta) \\ x_{pi} \sin(\beta) + y_{pi} \cos(\beta) \end{pmatrix}$

Praktisches Beispiel

- 2D-Grafik passend auf Bildschirm anzeigen



$$T(R(S(T(\mathbf{p}_i, \mathbf{t}_1), \mathbf{s}), \alpha), \mathbf{t}_2)$$

Praktisches Beispiel

$$G(\mathbf{p}) = T(R(S(T(\mathbf{p}, \mathbf{t}_1), \mathbf{s}), \alpha), \mathbf{t}_2)$$

- Jeder Datenpunkt durchläuft nacheinander jede Transformation
- Laufzeitkomplexität bei N Datenpunkten und M Transformationen:

$$O(MN)$$

- Praktische Objekte/Grafiken bestehen aus sehr vielen Datenpunkten (z. B. Pixel oder Vertices bei 3D-Modellen)
 - Eine Grafik/Objekt durchläuft u. U. sehr viele Transformationen
- Diese Transformationsrechnung ist teuer!
- Diese Transformationsrechnung ist uneinheitlich!

→ Die Verwendung von Matrizen löst Probleme:

$$\mathbf{T}_2(\mathbf{R}(\mathbf{S}(\mathbf{T}_1 \mathbf{p}))) = (\mathbf{T}_2 \mathbf{R} \mathbf{S} \mathbf{T}_1) \mathbf{p} = \mathbf{G} \mathbf{p} \rightarrow O(M + N)$$

Matrizen

- Matrix beschreibt Koeffizienten für lineares Gleichungssystem in kompakter Form:

$$\mathbf{A}\mathbf{p} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x_p \\ y_p \end{pmatrix} = \begin{pmatrix} a_{11}x_p + a_{12}y_p \\ a_{21}x_p + a_{22}y_p \end{pmatrix}$$

- Matrix-Multiplikation (Zeile x Spalte):

$$AB = C$$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ \boxed{a_{21}} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix} \begin{pmatrix} b_{11} & \boxed{b_{12}} & b_{13} \\ b_{21} & \boxed{b_{22}} & b_{23} \\ b_{31} & \boxed{b_{32}} & b_{33} \\ b_{41} & \boxed{b_{42}} & b_{43} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & \boxed{c_{22}} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}$$

Zeile
Spalte

$$a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} + a_{24}b_{42} = c_{22}$$

- Identitätsmatrix (neutrale Matrix):

$$\mathbf{I}\mathbf{p} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_p \\ y_p \end{pmatrix} = \begin{pmatrix} 1x_p + 0y_p \\ 0x_p + 1y_p \end{pmatrix} = \begin{pmatrix} x_p \\ y_p \end{pmatrix} = \mathbf{p}$$

- Nicht kommutativ:

$$\mathbf{AB} \neq \mathbf{BA}$$

Verschiebung, Skalierung & Rotation in der Ebene mit 2x2 Matrizen

- Skalierung: $S(\mathbf{p}_i, \mathbf{s}) = \begin{pmatrix} x_s x_{pi} \\ y_s y_{pi} \end{pmatrix} \rightarrow \mathbf{Sp}_i = \begin{pmatrix} \underline{x_s} & 0 \\ 0 & \underline{y_s} \end{pmatrix} \begin{pmatrix} x_{pi} \\ y_{pi} \end{pmatrix}$
- Rotation: $R(\mathbf{p}_i, \beta) = \begin{pmatrix} x_{pi} \cos(\beta) - y_{pi} \sin(\beta) \\ x_{pi} \sin(\beta) + y_{pi} \cos(\beta) \end{pmatrix}$
 $\rightarrow \mathbf{Rp}_i = \begin{pmatrix} \underline{\cos(\beta)} & \underline{-\sin(\beta)} \\ \underline{\sin(\beta)} & \underline{\cos(\beta)} \end{pmatrix} \begin{pmatrix} x_{pi} \\ y_{pi} \end{pmatrix}$
- Verschiebung: $T(\mathbf{p}_i, \mathbf{t}) = \begin{pmatrix} x_{pi} + x_t \\ y_{pi} + y_t \end{pmatrix} \rightarrow \mathbf{Tp}_i = \begin{pmatrix} \text{red circle with slash} & \text{red circle with slash} \\ \text{red circle with slash} & \text{red circle with slash} \end{pmatrix} \begin{pmatrix} x_{pi} \\ y_{pi} \end{pmatrix}$

Homogener Raum/Koordinaten

Problem mit Translation mit 2x2-Matrix

$$\mathbf{T}\mathbf{0} = \mathbf{T} \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \mathbf{0}, \quad \text{gewollt: } \mathbf{T}\mathbf{0} = \mathbf{t}$$


Lösung: Einbettung in (n+1)-dimensionalen projektiven Raum \rightarrow Homogener Raum (hier $H = P(R^3)$)

$$R^2 \rightarrow H, \begin{pmatrix} x_p \\ y_p \end{pmatrix} \rightarrow \begin{pmatrix} x_p \\ y_p \\ 1 \end{pmatrix} \quad H \rightarrow R^2, \begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix} \rightarrow \begin{pmatrix} x_p / z_p \\ y_p / z_p \end{pmatrix}$$

Translation mit homogener Matrix

- Homogene Translationsmatrix mit Translationsvektor $\mathbf{t} = \begin{pmatrix} x_t \\ y_t \end{pmatrix}$

$$\mathbf{T}\mathbf{0} = \mathbf{T} \begin{pmatrix} x_p \\ y_p \\ 1 \end{pmatrix} = \begin{pmatrix} x_p + x_t \\ y_p + y_t \\ 1 \end{pmatrix}$$


$$\mathbf{T} = \begin{pmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{pmatrix}$$

Verschiebung, Skalierung & Rotation in der Ebene mit homogenen Matrizen

- Verschiebung: $\mathbf{T_p} = \begin{pmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_p \\ y_p \\ 1 \end{pmatrix}$

- Skalierung: $\mathbf{S_{p_i}} = \begin{pmatrix} \underline{x_s} & 0 & 0 \\ 0 & \underline{y_s} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_p \\ y_p \\ 1 \end{pmatrix}$

- Rotation: $\mathbf{R_{p_i}} = \begin{pmatrix} \underline{\cos(\beta)} & \underline{-\sin(\beta)} & 0 \\ \underline{\sin(\beta)} & \underline{\cos(\beta)} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{pi} \\ y_{pi} \\ 1 \end{pmatrix}$

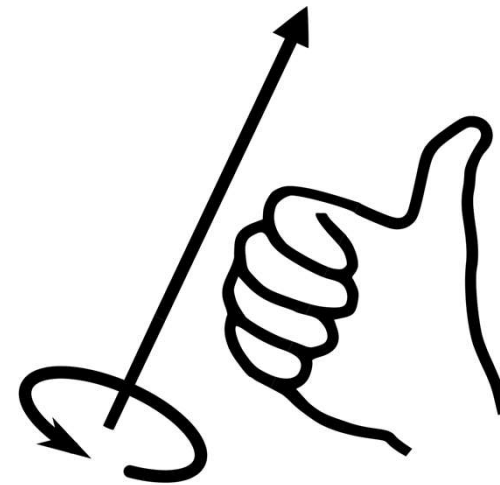
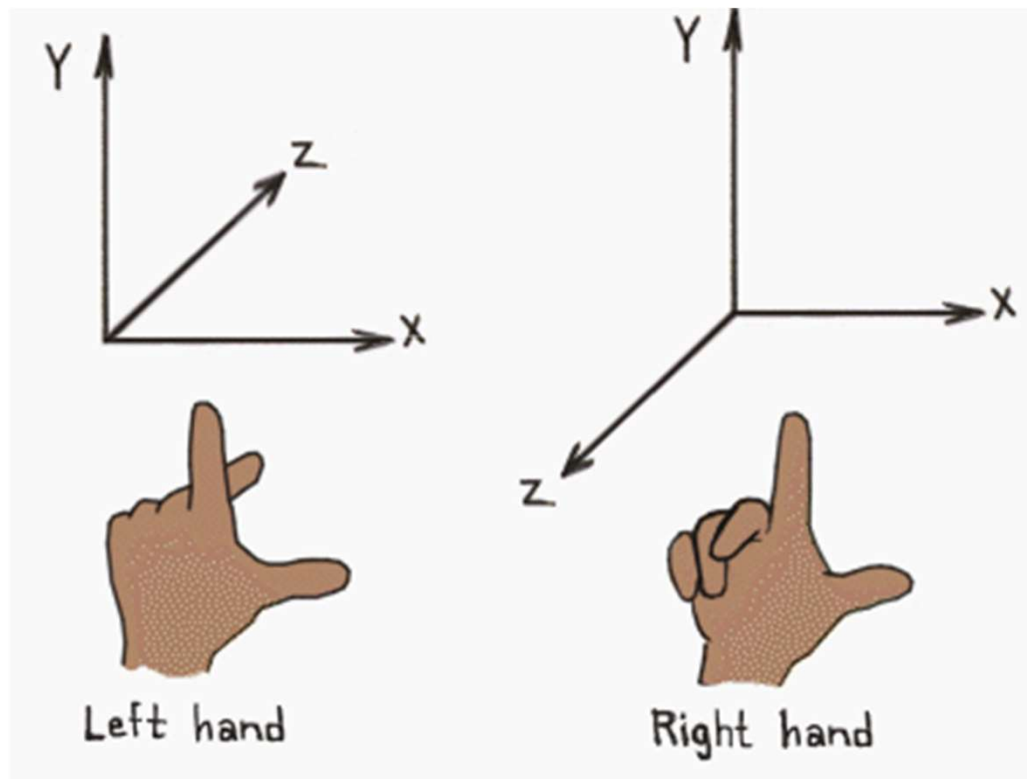
Skalierung & Translation im dreidimensionalen Raum

Translation & Skalierung können einfach um dritte (Z-)Koordinate erweitert werden (wieder homogene Koordinaten):

$$\begin{array}{ccc} \text{2D} & & \text{3D} \\ \mathbf{Tp} = \begin{pmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_p \\ y_p \\ 1 \end{pmatrix} & \xrightarrow{\quad} & \mathbf{Tp} = \begin{pmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_p \\ y_p \\ z_p \\ 1 \end{pmatrix} \\ \mathbf{Sp} = \begin{pmatrix} x_s & 0 & 0 \\ 0 & y_s & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_p \\ y_p \\ 1 \end{pmatrix} & \xrightarrow{\quad} & \mathbf{Sp} = \begin{pmatrix} x_s & 0 & 0 & 0 \\ 0 & y_s & 0 & 0 \\ 0 & 0 & z_s & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_p \\ y_p \\ z_p \\ 1 \end{pmatrix} \end{array}$$

Rotation im dreidimensionalen Raum

- Koordinatensystem-Wahl wirkt sich auf Rotation aus
 - Cinema 4D, Direct3D, ... → linkshändisches Koordinatensystem
 - Maya, OpenGL, ... → rechtshändisches Koordinatensystem
 - Kein einheitlicher Standard



Rotation im dreidimensionalen Raum

- Rotation kann um die 3 Hauptachsen (X,Y,Z) erfolgen
- Prinzip der 2D-Rotation in XY-Ebene, um gedachte Z-Achse, lässt sich direkt in den dreidimensionalen Raum überführen:

$$\begin{array}{c} \text{2D} \\ \mathbf{R} = \begin{pmatrix} \cos(\gamma) & -\sin(\gamma) & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \quad \Rightarrow \quad \begin{array}{c} \text{3D} \\ \mathbf{R}_z \mathbf{p} = \begin{pmatrix} \cos(\gamma) & -\sin(\gamma) & 0 & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_p \\ y_p \\ z_p \\ 1 \end{pmatrix} \end{array}$$

- Praktische Konstruktionsregeln:
 - Die Cosinus-Terme müssen immer auf der Hauptdiagonalen liegen und die Sinus-Terme nicht, denn $\gamma = 0 \rightarrow \mathbf{R} = \mathbf{I}$ (Identitätsmatrix)
 - Die Koordinaten der Achse, um die gedreht wird, bleiben unverändert (die dazugehörige Spalte ist also neutral)
 - Da die Rotationsmatrix eine orthogonale Matrix bildet, gilt $\mathbf{R}^{-1} = \mathbf{R}^T$. Hierdurch ergibt sich, dass ebenfalls eine neutrale Zeile resultiert

Rotation im dreidimensionalen Raum

- Rotation für die drei einzelnen Hauptachsen

$$\mathbf{R}_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{R}_y = \begin{pmatrix} \cos(\gamma) & -\sin(\gamma) & 0 & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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

Anstatt mit 3 Euler-Winkeln zu arbeiten, kann eine Rotationsmatrix auch durch eine beliebig gewählte Drehachse und einen Drehwinkel berechnet werden.

- Rotation um alle Achsen $\mathbf{R}(\alpha, \beta, \gamma)$ (Reihenfolge muss einheitlich gewählt werden)

$$\mathbf{R} = \mathbf{R}_z \mathbf{R}_y \mathbf{R}_x$$

Translation, Skalierung & Rotation im dreidimensionalen Raum

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{S} = \begin{pmatrix} x_s & 0 & 0 & 0 \\ 0 & y_s & 0 & 0 \\ 0 & 0 & z_s & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{R}_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

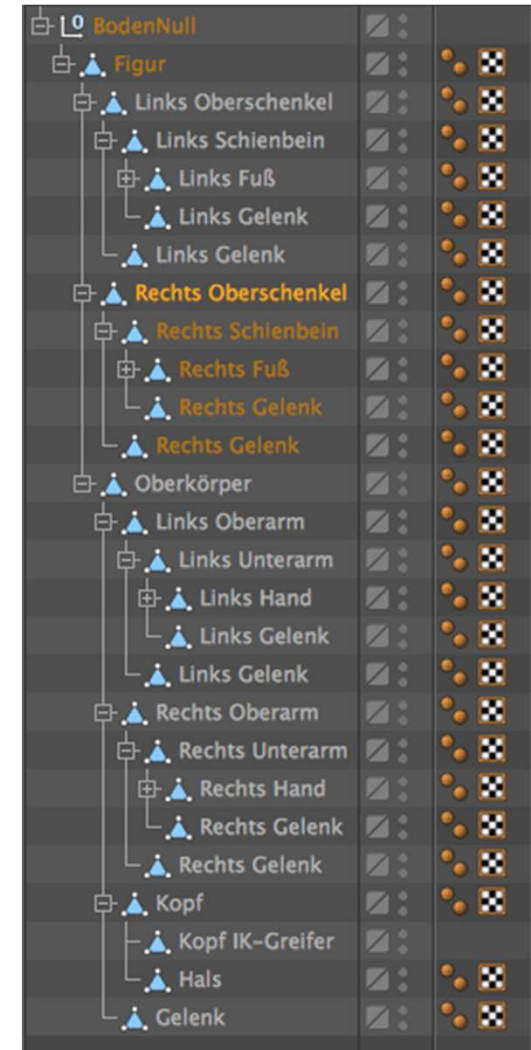
$$\mathbf{R}_y = \begin{pmatrix} \cos(\beta) & 0 & \sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{R}_z = \begin{pmatrix} \cos(\gamma) & -\sin(\gamma) & 0 & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Objekt-Gesamttransformation:

$$\mathbf{M}_o = \mathbf{T} \mathbf{R}_z \mathbf{R}_y \mathbf{R}_x \mathbf{S}$$

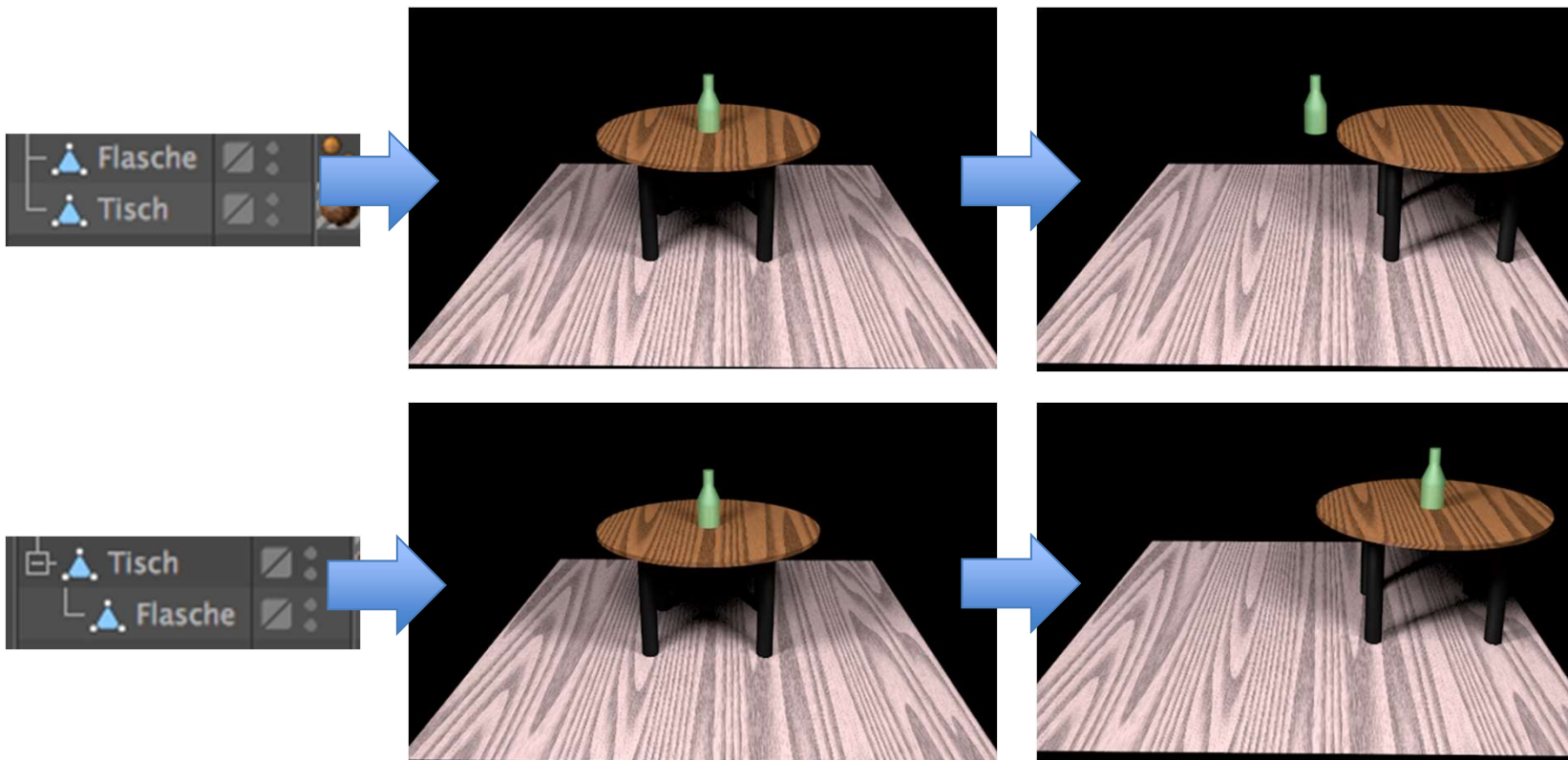
Szenengraph/Szenen-Management

- Hierarchische Anordnung von Objekten, um geometrische und semantische Zusammenhänge abzubilden



Szenengraph

- Hierarchische Anordnung von Objekten, um geometrische und semantische Zusammenhänge abzubilden



- Transformationshierarchie wird durch Matrix-Multiplikation abgebildet:

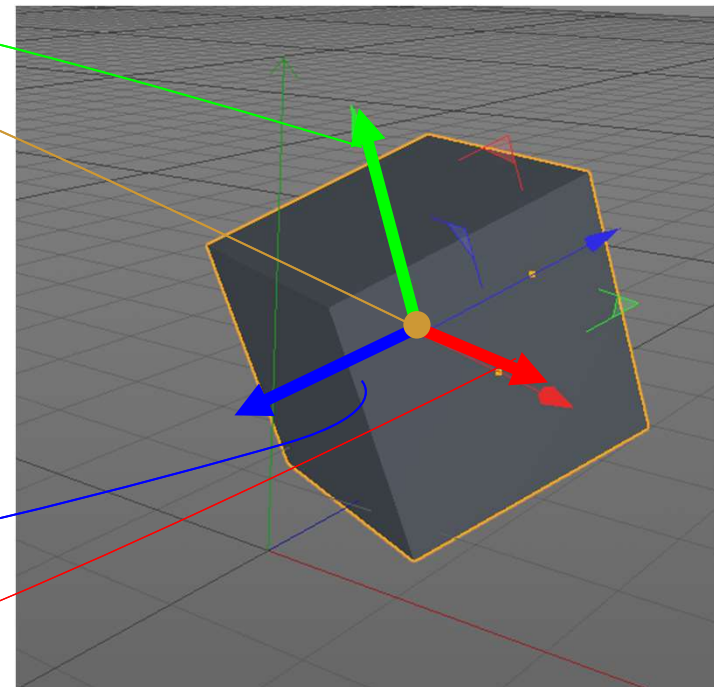
$$M_{\text{Tisch,Welt}} = M_{\text{Tisch,Objekt}}$$

$$M_{\text{Flasche,Welt}} = M_{\text{Tisch,Objekt}} M_{\text{Flasche,Objekt}}$$

Matrizen als Koordinatensysteme interpretieren

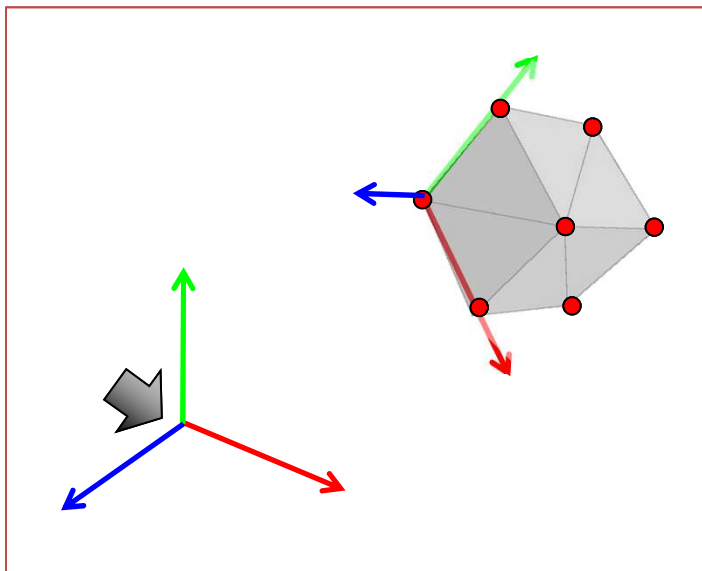
- Gegenwärtig verstehen wir eine Matrix, die eine affine Transformation abbildet, als Operand, der auf die Punkte eines Modells angewendet wird.
- Manchmal ist es aber sinnvoll, eine Matrix mit affiner Transformation als Koordinatensystem zu interpretieren (**Right-**, **Up-**, **Forward-** & **Position-** Vektor):

$$M = \begin{pmatrix} x_r & x_u & x_f & x_t \\ y_r & y_u & y_f & y_t \\ z_r & z_u & z_f & z_t \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



View-Transformation (Kamera-Transformation)

Welt-Raum



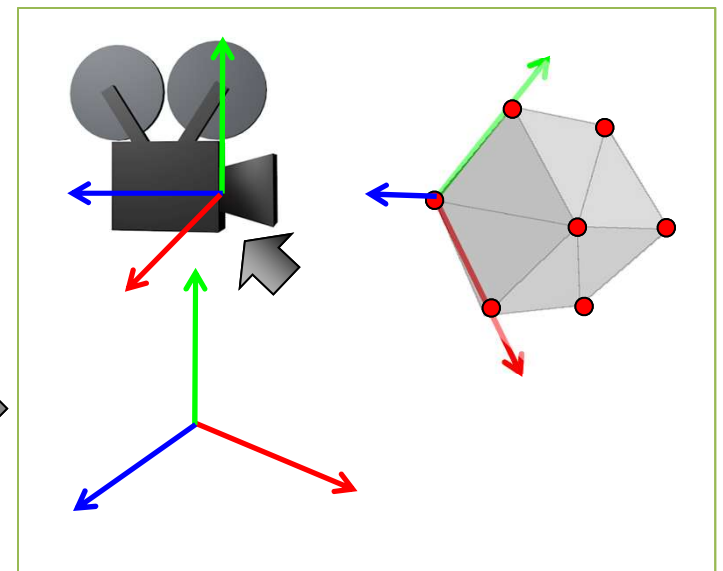
Gemeinsamer Raum für alle
Objekte einer virtuellen Szene.

Affine Transform.:

- Verschiebung
- Rotation

View-Transform.
(Kamera-Transform.)

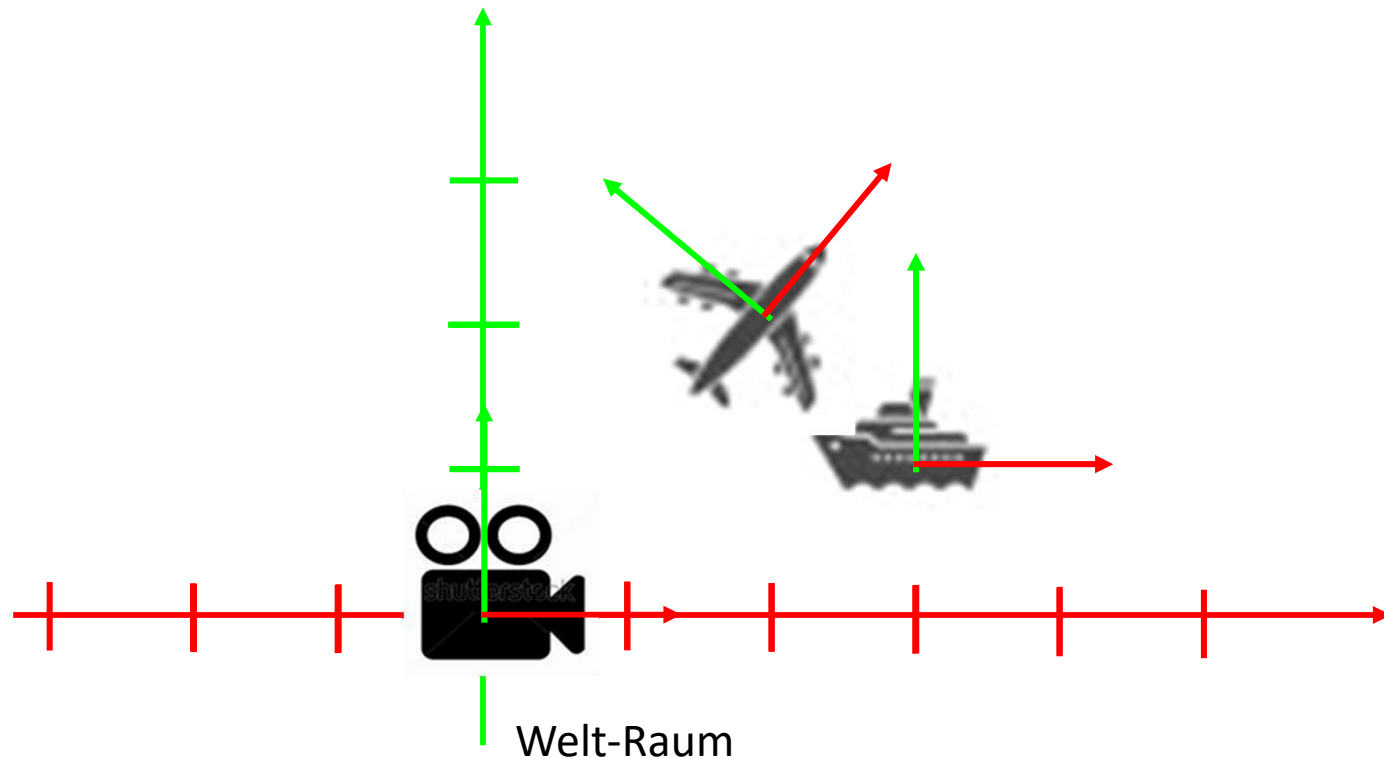
Kameraraum



Alle Objekte befinden sich in
einem Raum, in dem die Kamera
im Ursprung liegt.

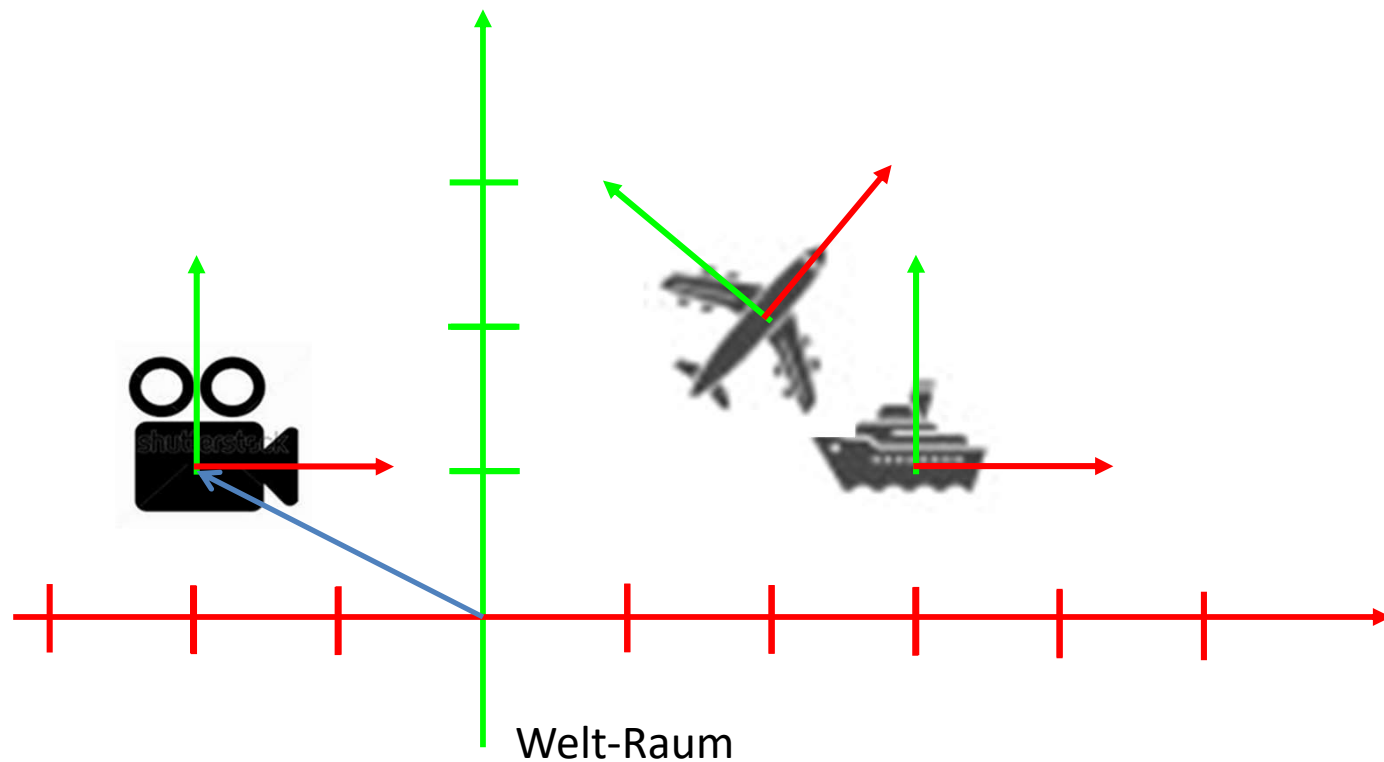
View-Transformation

- Transformationen werden auf Modellpunkte angewendet, dies gilt auch für die Sicht-Transformation.
- Frage: Wie können die Modelle transformiert werden, damit sie in Kamera-Koordinaten liegen?



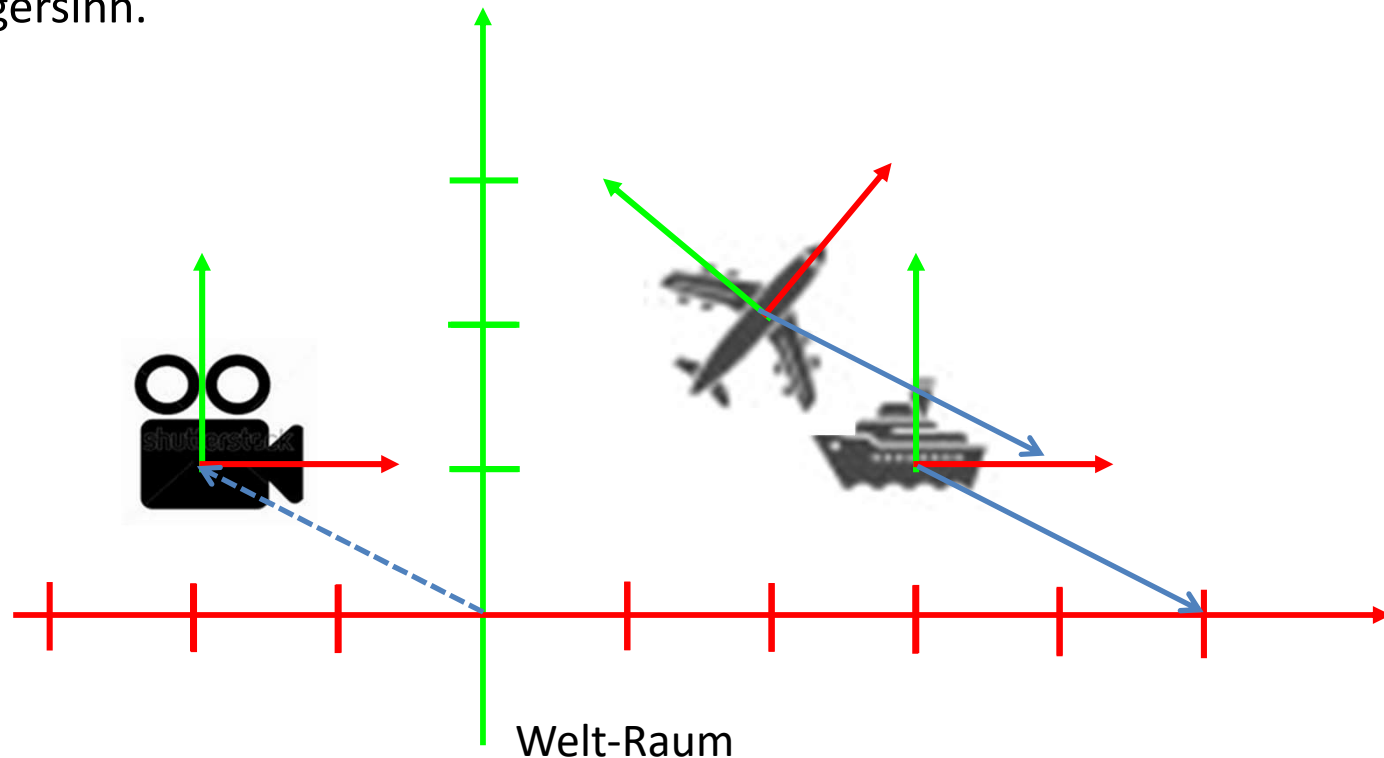
View-Transformation

- Bewegung der Kamera auf passende Position $(-2,1)$
- Schiff liegt auf $(5,0)$ und Flugzeug auf $(4,1)$ (im Kameraraum)



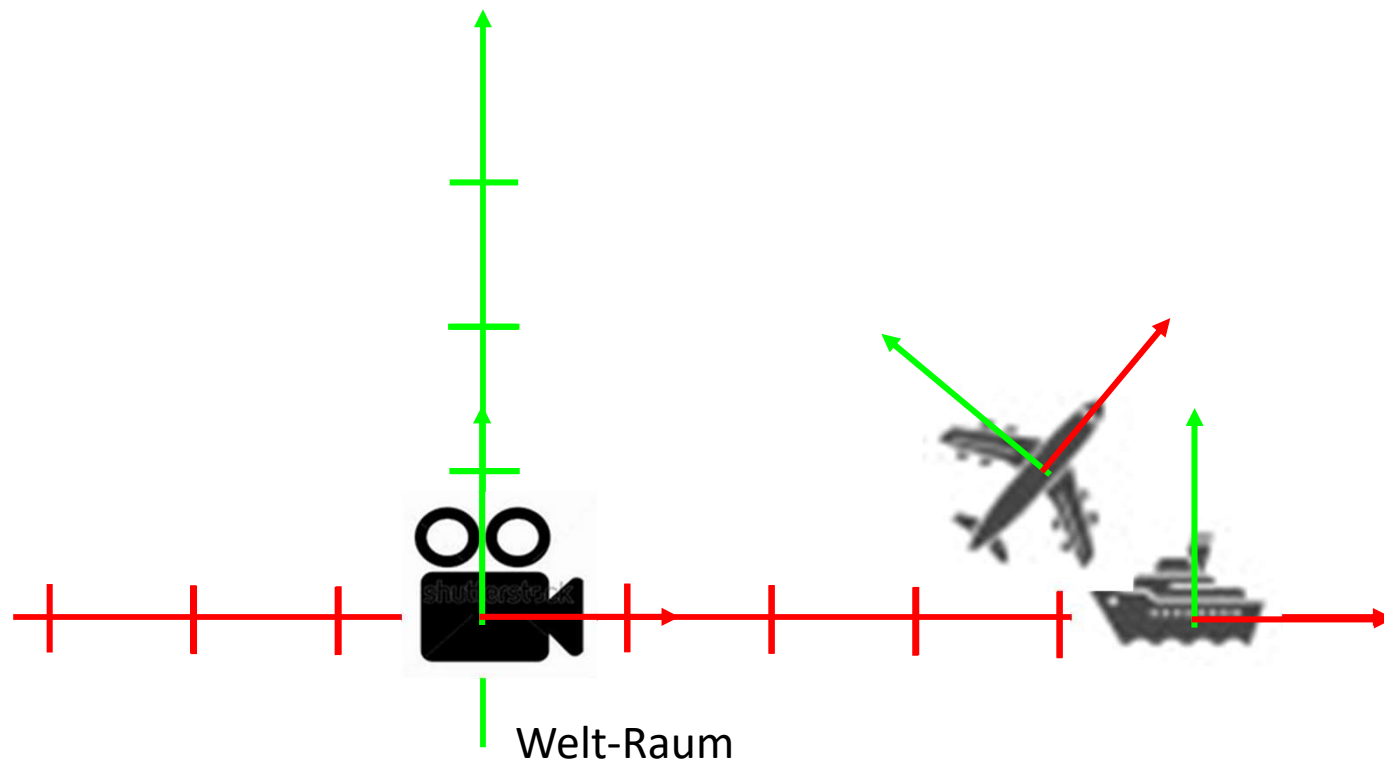
View-Transformation

- Idee: Anstatt die Kamera zu bewegen, werden alle Objekte mit der inversen Sicht-Transformation bewegt.
- ➔ Bewegt sich die Kamera nach rechts, wandern alle Objekte nach links.
- ➔ Bewegt sich die Kamera nach oben, wandern alle Objekte nach unten.
- ➔ Rotiert die Kamera im Uhrzeigersinn, rotieren die Objekte gegen den Uhrzeigersinn.



View-Transformation

- Schiff liegt wieder im Sicht-Raum auf (0,5) und Flugzeug auf (4,1)



View-Transformation

Aufbau der Kamera-Transformation:

1. Kamera wird wie Objekt im Raum transformiert (Rotations- und Translationsmatrix, Skalierung nicht nötig) $\rightarrow \mathbf{M}_{\infty}$.
2. Die sich ergebene Matrix wird invertiert $\rightarrow \mathbf{M}_c$.
3. Die invertierte Matrix wird mit den Objektmatrizen multipliziert.

$$\mathbf{M}_{\infty} = \mathbf{T}_{\infty} \mathbf{R}_{\infty,z} \mathbf{R}_{\infty,y} \mathbf{R}_{\infty,x}$$

$$\mathbf{M}_c = \mathbf{M}_{\infty}^{-1}$$

Schiff im Kameraraum: $\mathbf{M}_{c,\text{schiff}} = \mathbf{M}_c \mathbf{M}_{\text{schiff}}$

Flugzeug im Kameraraum: $\mathbf{M}_{c,\text{flugzeug}} = \mathbf{M}_c \mathbf{M}_{\text{flugzeug}}$

View-Transformation

Inverse für View-Transformation berechnen:

1. Der 3x3-Anteil der Matrix bildet orthogonales Koordinatensystem ab.
2. Für orthogonale Matrizen gilt: $\mathbf{M}\mathbf{M}^T = \mathbf{I}$.
3. Da für Matrizen $\mathbf{M}\mathbf{M}^{-1} = \mathbf{I}$ gilt, folgt aus 2, dass $\mathbf{M}^{-1} = \mathbf{M}^T$.

$$\mathbf{M}_{\infty} = \begin{pmatrix} \begin{array}{c} \text{r} \\ \boxed{x_r} \\ y_r \\ z_r \end{array} & \begin{array}{c} \text{u} \\ \boxed{x_u} \\ y_u \\ z_u \end{array} & \begin{array}{c} \text{f} \\ \boxed{x_f} \\ y_f \\ z_f \end{array} & \begin{array}{c} \text{t} \\ \boxed{x_t} \\ y_t \\ z_t \end{array} \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{M}_{\infty}^{-1} = \begin{pmatrix} \begin{array}{ccc} \boxed{x_r} & y_r & z_r \end{array} & \begin{array}{c} \boxed{-(\mathbf{t} \cdot \mathbf{r})} \end{array} \\ \begin{array}{ccc} \boxed{x_u} & y_u & z_u \end{array} & \begin{array}{c} \boxed{-(\mathbf{t} \cdot \mathbf{u})} \end{array} \\ \begin{array}{ccc} \boxed{x_f} & y_f & z_f \end{array} & \begin{array}{c} \boxed{-(\mathbf{t} \cdot \mathbf{f})} \end{array} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

4. Es gilt: $\mathbf{M}_{\infty} = \mathbf{T}_{\infty}\mathbf{R}_{\infty}$, daraus ergibt sich für die Inverse:

$$\mathbf{M}_{\infty}^{-1} = (\mathbf{T}_{\infty}\mathbf{R}_{\infty})^{-1} = \mathbf{R}_{\infty}^T \mathbf{T}_{\infty}^{-1} \quad (\text{die inverse Translationsmatrix besitzt lediglich den negativen Translationsvektor})$$

View-Transformation

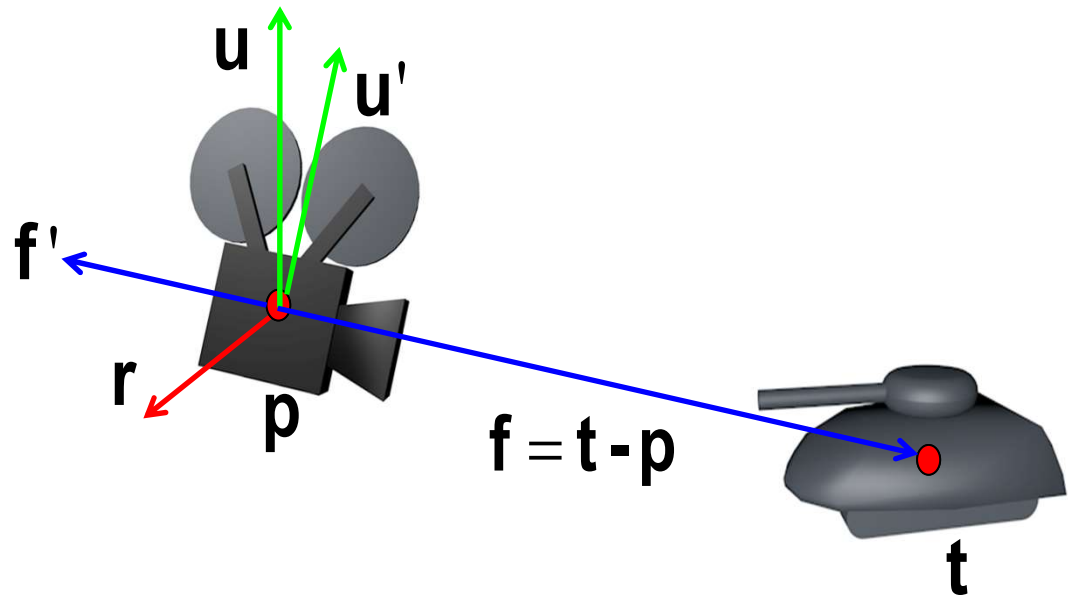
Konstruktion einer „LookAt“-View-Matrix mit Hilfe eines Target-, Position- & Up-Vektors ($\mathbf{t}, \mathbf{p}, \mathbf{u}$)

$$\mathbf{r} = \frac{\mathbf{f} \times \mathbf{u}}{|\mathbf{f} \times \mathbf{u}|}$$

$$\mathbf{u}' = \frac{\mathbf{r} \times \mathbf{f}}{|\mathbf{r} \times \mathbf{f}|}$$

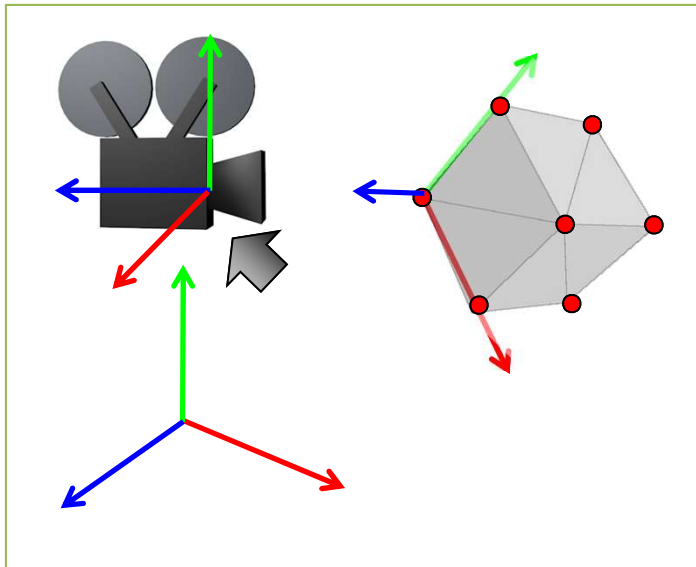
$$\mathbf{f}' = \frac{-\mathbf{f}}{|\mathbf{f}|}$$

$$\mathbf{M}_c = \begin{pmatrix} \mathbf{r} & \mathbf{u}' & \mathbf{f}' & \mathbf{p} \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} \mathbf{r}^T & -(\mathbf{r} \bullet \mathbf{p}) \\ \mathbf{u}'^T & -(\mathbf{u}' \bullet \mathbf{p}) \\ \mathbf{f}'^T & -(\mathbf{f}' \bullet \mathbf{p}) \\ \mathbf{0}^T & 1 \end{pmatrix}$$



Projektion

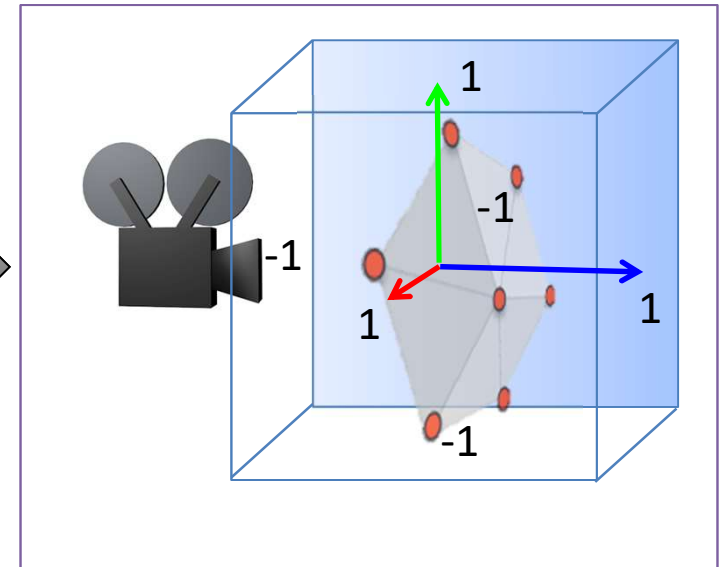
Kameraraum



Alle Objekte befinden sich in einem Raum, in dem die Kamera im Ursprung liegt.

Projektionstranf.

Normalisierter Bildraum

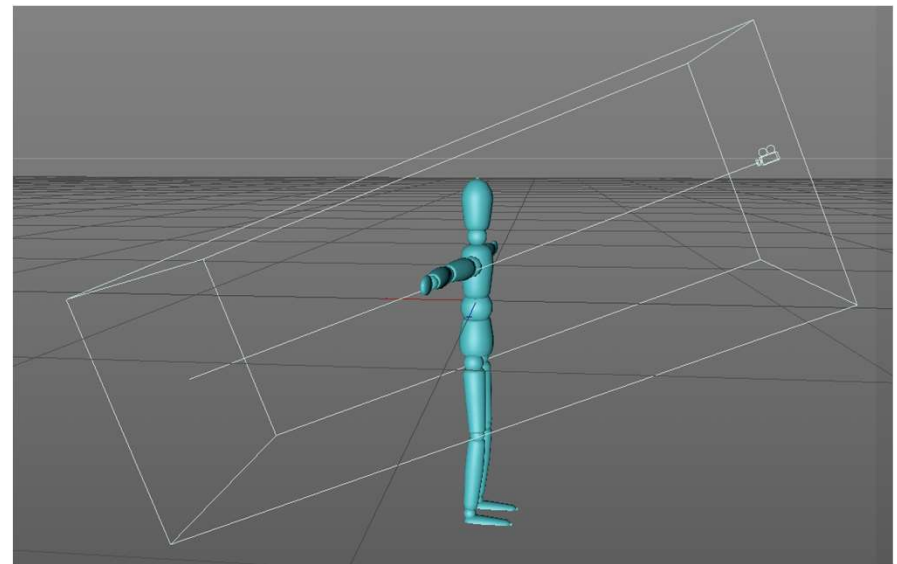


Nach der Projektion befinden sich alle Vertices in einem Würfel mit den Kantenlängen 2,2,2.

(Kamera-)Projektionen

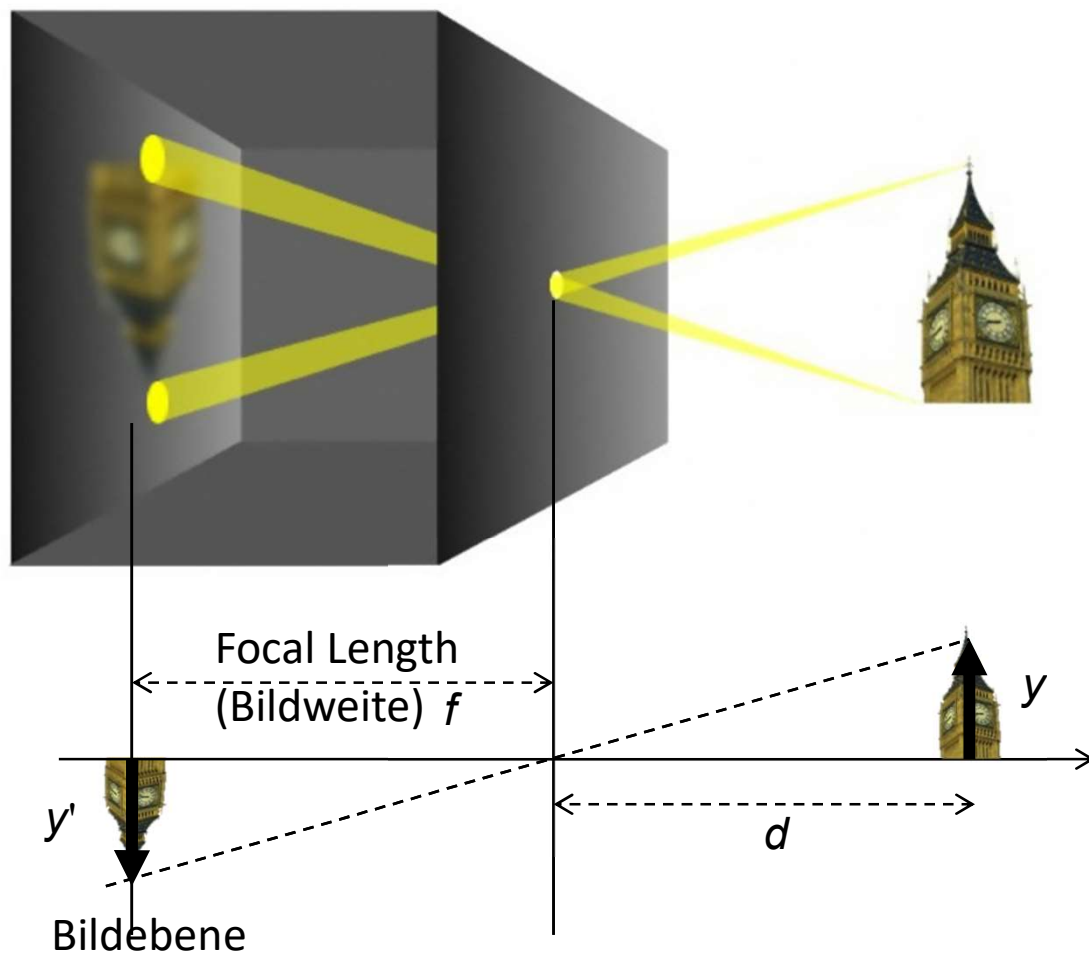
- Parallelprojektion im Kameraraum (Orthogonale Projektion):
 - w gibt Breite und h gibt Höhe des Sichtfensters an (in Raumeinheiten).
 - n gibt die Near- und f die Far-Clipping-Plane an (in Raumeinheiten)
 - Bei der Parallelprojektion kann die Entfernung zwischen Kamera und Objekt nicht abgeschätzt werden (warum?).
 - Nach der Projektion werden alle Koordinaten, die größer 1 und kleiner -1 sind, abgeschnitten (liegen nicht im Bild).

$$\mathbf{M}_{p,||} = \begin{pmatrix} 2/w & 0 & 0 & 0 \\ 0 & 2/h & 0 & 0 \\ 0 & 0 & \frac{-2}{f-n} & \frac{n-f}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Perspektivische Projektion

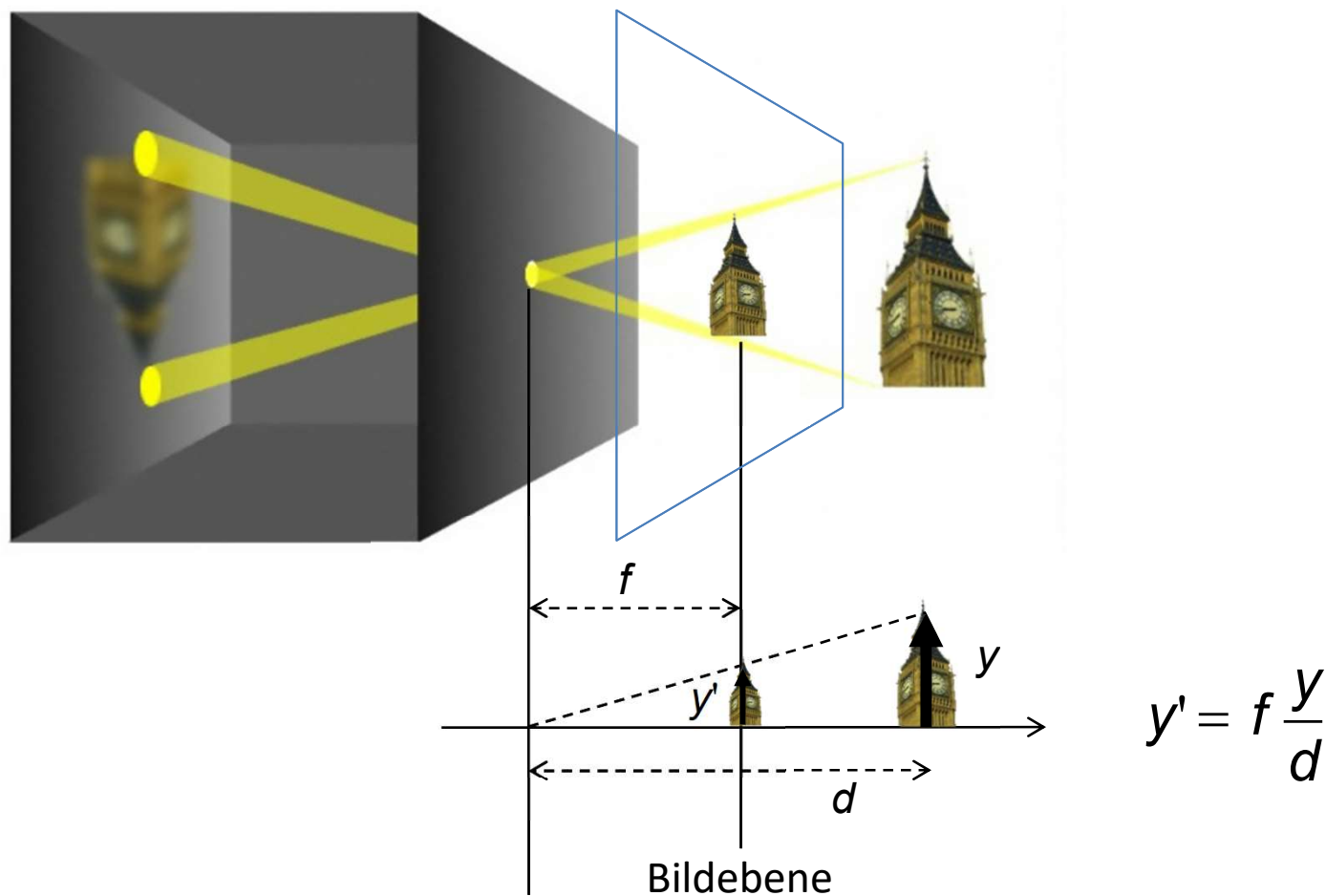
Es wird das physikalische Prinzip einer Lochkamera für die Abbildung der perspektivischen Projektion genutzt:



$$y' = f \frac{y}{d}$$

Perspektivische Projektion

Der Einfachheit halber wird die Bildebene vor dem Kameraloch angenommen, dies verändert nichts an den Berechnungen oder dem Prinzip der Lochkamera.



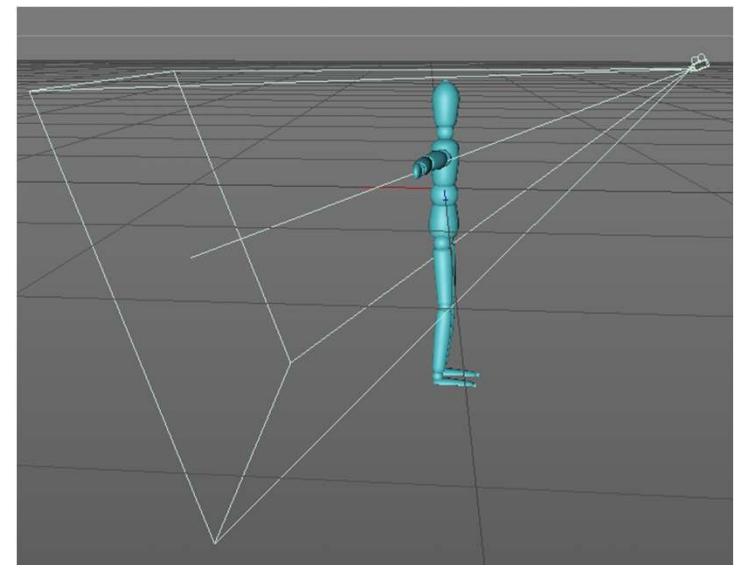
(Kamera-)Projektionen

Perspektivische (Zentral-)Projektion:

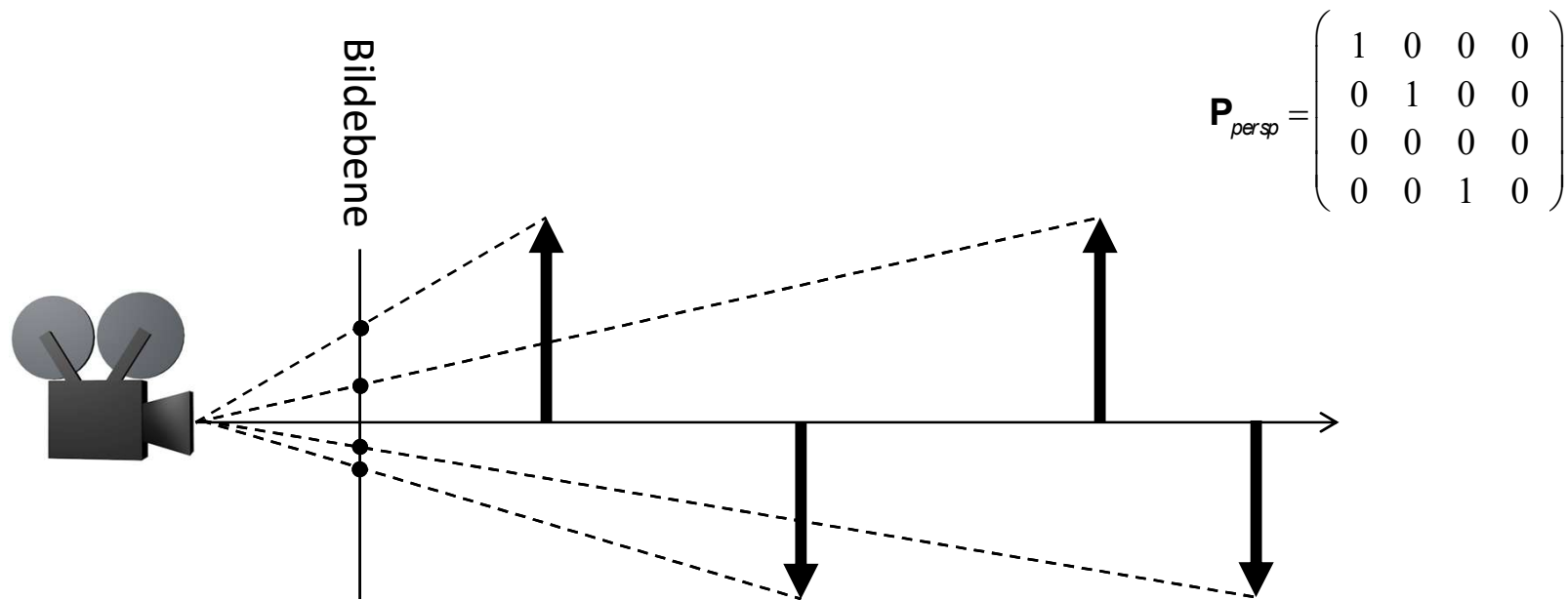
Einfaches Prinzip:

- Die X- und Y-Koordinate wird durch Z geteilt (perspektivische Teilung).
- Mit zunehmender Entfernung wird so die Abbildung eines Objekts kleiner und sie wird zum Zentrum der Projektion gezogen.

$$\mathbf{M}_{p,persp} \mathbf{V} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_v \\ y_v \\ z_v \\ 1 \end{pmatrix} = \begin{pmatrix} x_v \\ y_v \\ 0 \\ z_v \end{pmatrix} = \begin{pmatrix} x_v / z_v \\ y_v / z_v \\ 0 \\ 1 \end{pmatrix}$$



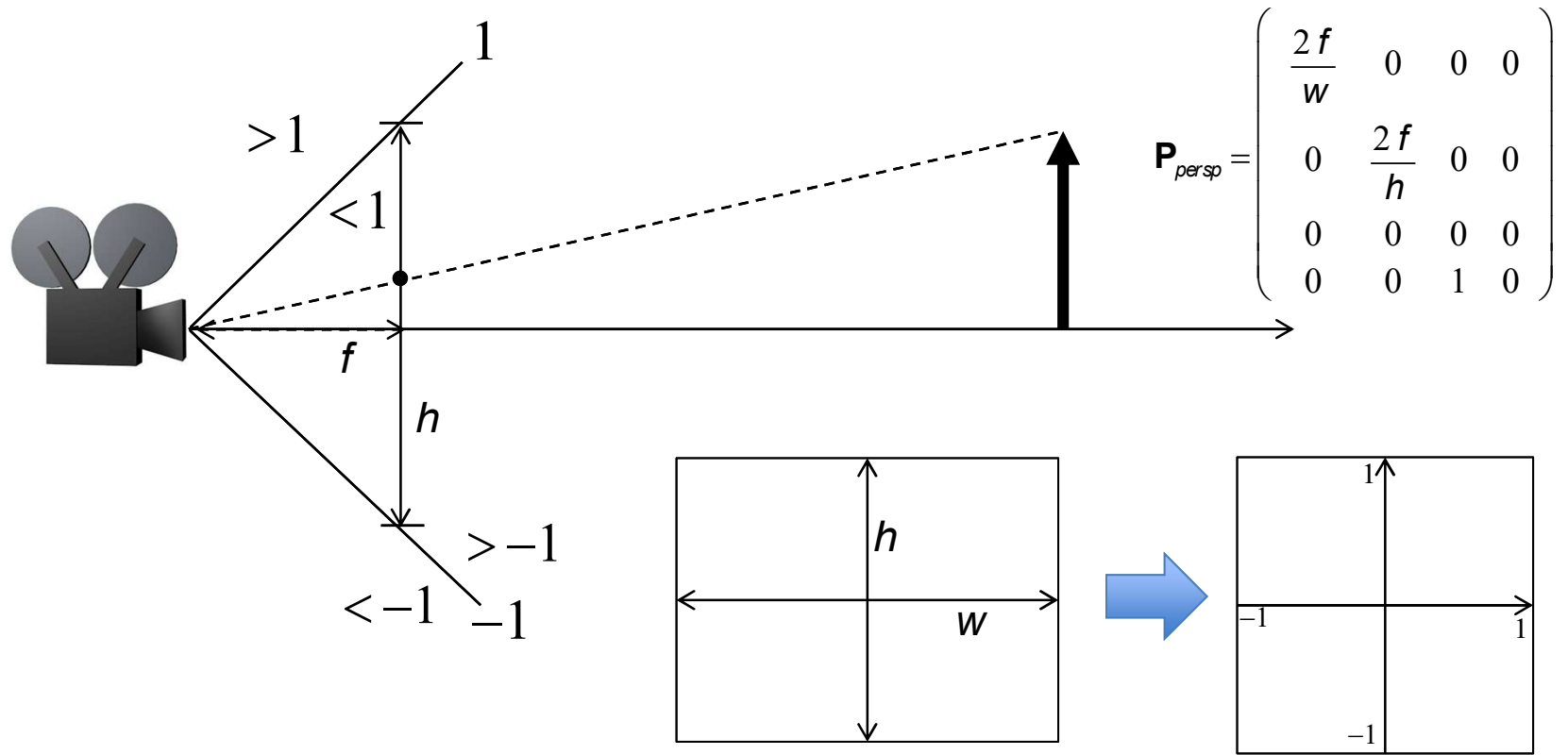
Perspektivische Projektion



Die Abbildung der Objekte wird mit zunehmender Entfernung kleiner. Ferner wandern entfernte Objekte zur Kameraachse.

Wann liefert die Matrix keine korrekten Ergebnisse?

Perspektivische Projektion



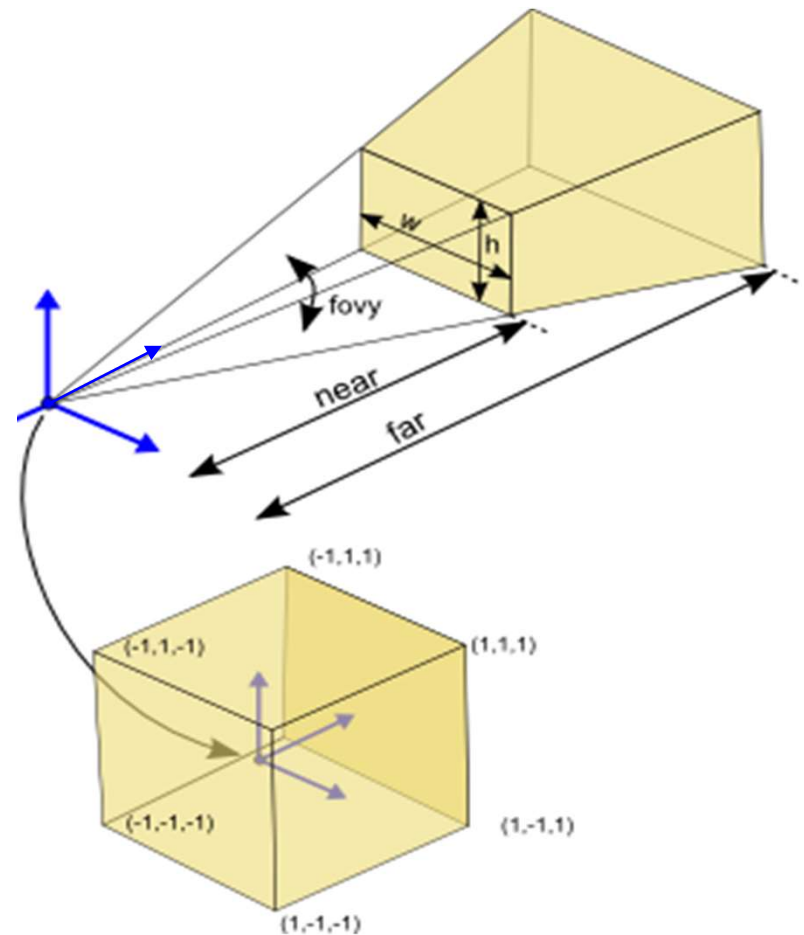
Entwickeln Sie eine perspektivische Projektionsmatrix, die die Bildweite f und die Breite (b) und Höhe (h) der Bildebene entgegen nimmt. Nach der Projektion soll jeder Punkt der auf der Bildebene abgebildet werden kann zwischen $[-1,1] \times [-1,1]$ liegen.

(Kamera-)Projektionen

Perspektivische Projektion:

- w gibt Breite und h gibt Höhe der nahen Clipping-Ebene an (in Raumeinheiten).
- n gibt den Abstand der nahen und f den Abstand der fernen Clipping-Ebene an (in Raumeinheiten).

$$\mathbf{M}_{p,persp} = \begin{pmatrix} \frac{2n}{w} & 0 & 0 & 0 \\ 0 & \frac{2n}{h} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$



Gesamt-Transformation bilden und dem Shader übergeben (Praktikum)

```
bool Shader::load(...)
{
    ...
    ShaderProgram = createShaderProgram(pVData, pFData);
    ModelViewProjLoc = glGetUniformLocation(ShaderProgram, "ModelViewProjMat");
}
```

Matrix m, v, p;

```
void Shader::activate(const BaseCamera& Cam) const
{
    ...
    Matrix ModelViewProj = Cam.getProjectionMatrix()
                          * Cam.getViewMatrix()
                          * modelTransform();
    glUniformMatrix4fv(ModelViewProjLoc, 1, GL_FALSE, ModelViewProj.m);
}
```

Zum Beispiel: `m = m1.translation(Pos) * m2.rotationY(RotY);` ←
`v.lookAt(Target, Up, Pos);` ←
`p.perspective(M_PI, WinWidth/WinHeight, 0.1f, 100.0f);` ←

Vielen Dank für Ihre Aufmerksamkeit!