5: Transformationen und Modellhierarchien

Lehrstuhl für Computergraphik und
Multimediasysteme

Universität Siegen

Struktur & Inhalt des Kapitels

- 1: Einführung
- 2: Affine Transformationen
- 3: Viewing Transformation
- 4: Projektionstransformation
- 5: Hierarchische Modelle
- 6: Umsetzung mit OpenGL und glm

Bislang: Alle Elemente in der Szene in einem Koordinatensystem
$\Rightarrow$ sehr unhandliche Modellierung

Modellierungsansätze können verschiedenartig sein

- *Top-Down:* Zerlegung der Szene in Objekte $\Rightarrow$ Verfeinerung
- *Bottom-Up:* Zusammenfügen von Primitiven zu komplexeren Objekten

Ziel: Flexible Form der Manipulation von Objekten „als Ganzes“

Konkrete Aufgaben für die Szenenerstellung:

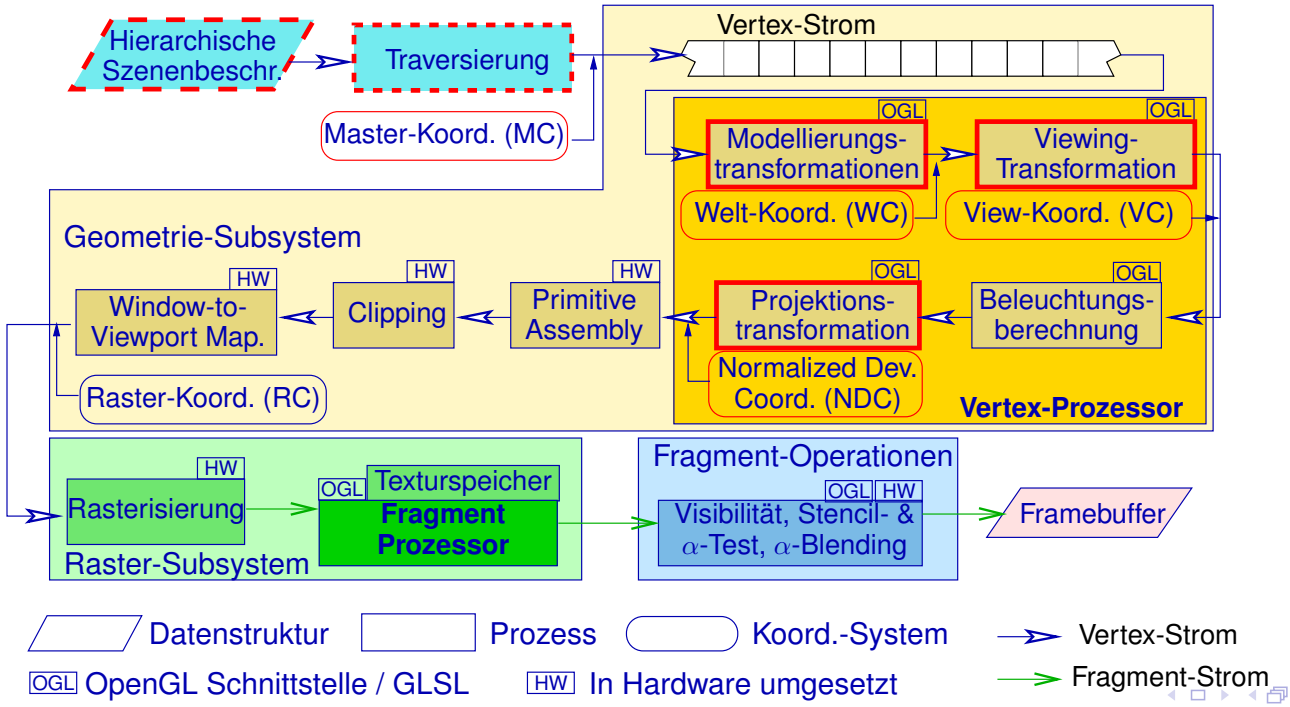
- Bewegung von Objekten (Verschieben, Drehen, Skalieren)
- Bewegung von zusammengesetzten Objekten (z.B. Charaktere)

hierarchische Modellstruktur

- flexible Arten der Modellerstellung ($\rightarrow$ CG-II)
- anwendungsspezifische Modellierungstechniken ($\rightarrow$ CG-II)

Bezug zur Graphik-Pipeline

Transformationen: *Nicht Teil* von OpenGL; wir nutzen `glm`
Hierarchie & Traversierung: *Nicht Teil* von OpenGL; wir nutzen eine eigene Stack-Klasse & der Programmcode legt Traversierung implizit fest



Bezeichnung (Affine Kombination, konvexe Hülle)

Affine Kombination: *Für Punkte* $\mathbf{P}_1, \ldots, \mathbf{P}_k \in A$ *und skalare Werte* $s_1, \ldots, s_k$ *mit* $\sum_{i=1}^{k} s_i = 1$ *ist* $\sum_{i=1}^{k} s_i \mathbf{P}_i \in A$ *ebenfalls ein Punkt,*

$$\text{denn } \sum_{i=1}^{k} s_i \mathbf{P}_i = s_1\mathbf{P}_1 + s_2\mathbf{P}_2 + \ldots + s_k\mathbf{P}_k$$
$$= (\underbrace{s_1 + \ldots + s_k}_{=1})\mathbf{P}_1 + s_2 \underbrace{(\mathbf{P}_2 - \mathbf{P}_1)}_{\vec{\mathbf{v}}_2} + \ldots s_k \underbrace{(\mathbf{P}_k - \mathbf{P}_1)}_{\vec{\mathbf{v}}_k}$$

Konvexe Hülle *der Punkte* $\mathbf{P}_1, \ldots, \mathbf{P}_k \in A$*: Die kleinste, sie umfassende, konvexe Menge bestehend aus*

$$\mathbf{Q} = \sum_{i=1}^{k} s_i \mathbf{P}_i, \text{ mit } \sum_{i=1}^{k} s_i = 1 \text{ und } s_i \geq 0 \; \forall i = 1, \ldots, k$$

Charakterisierung affiner Transformationen

Affine Transformationen sind die Grundlage für

- Positionierung von Objekten
- Wechsel des Koordinatensystems

Definition: Sei A ein n-dimensionaler affiner Raum. Eine Transformation $T : A \longrightarrow A$ heißt *affin*, falls sie affine Kombinationen invariant läßt, d.h.

$$\forall \mathbf{P}_1, \ldots, \mathbf{P}_k \in A \text{ und } s_1, \ldots, s_k \in \mathbb{R} : \sum s_i = 1 \text{ gilt:}$$
$$T(s_1\mathbf{P}_1 + \ldots + s_k\mathbf{P}_k) = s_1 T(\mathbf{P}_1) + \ldots + s_k T(\mathbf{P}_k)$$

Charakterisierung: T ist genau dann affin, wenn es eindeutig aus einer linearen Abbildung und einer Translation besteht, d.h. $\exists$ Matrix $M \in \mathbb{R}^{n \times n}$ und Vektor $\vec{\mathbf{t}} \in A$ mit:

$$T(\mathbf{P}) = M \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix} + \begin{pmatrix} t_1 \\ \vdots \\ t_n \end{pmatrix} \text{ falls } \mathbf{P} = \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix}, \; \vec{\mathbf{t}} = \begin{pmatrix} t_1 \\ \vdots \\ t_n \end{pmatrix}$$

Homogene Koordinaten

Ziel: Kompakte Darstellung affiner Transformationen

Ansatz: Hinzunahme einer vierten Koordinate (*homogene* oder w-Koordinate):

$$\mathbf{P} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \rightarrow \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} \text{ und } M \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \rightarrow \begin{bmatrix} & M & & \begin{matrix} t_x \\ t_y \\ t_z \end{matrix} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

Homogene Koordinatenschreibweise für Punkte und Vektoren:

$$\text{Punkt: } \mathbf{P} = p_1\vec{\mathbf{u}}_1 + p_2\vec{\mathbf{u}}_2 + p_3\vec{\mathbf{u}}_3 + 1\mathcal{O} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix}$$

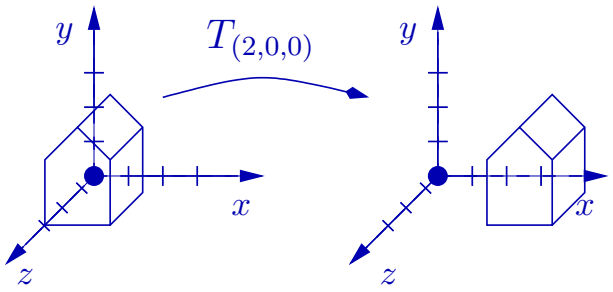
$$\text{Vektor: } \vec{\mathbf{v}} = v_1\vec{\mathbf{u}}_1 + v_2\vec{\mathbf{u}}_2 + v_3\vec{\mathbf{u}}_3 + 0\mathcal{O} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{bmatrix}$$

Beachte: Hier sind Punkt und Vektor unterscheidbar!

Basistransformationen

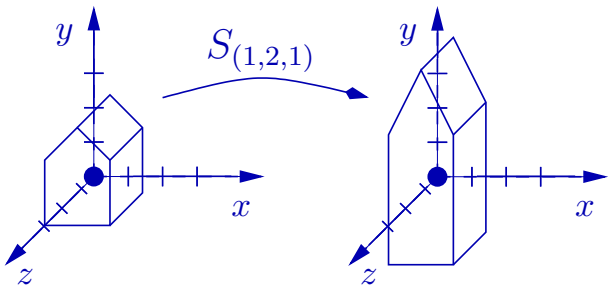
Translation: $\begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \rightarrow \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$

$$T_{\vec{\mathbf{t}}} \cdot \mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$



Skalierung: $\begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \rightarrow \begin{pmatrix} s_x \cdot p_x \\ s_y \cdot p_y \\ s_z \cdot p_z \end{pmatrix}$

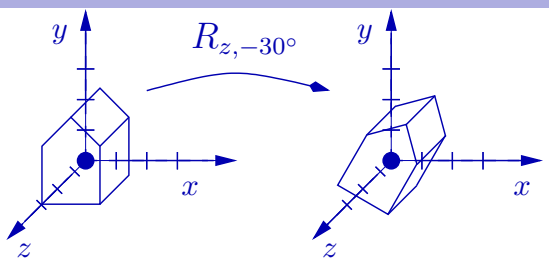
$$S_{s_x,s_y,s_z} \cdot \mathbf{P} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$



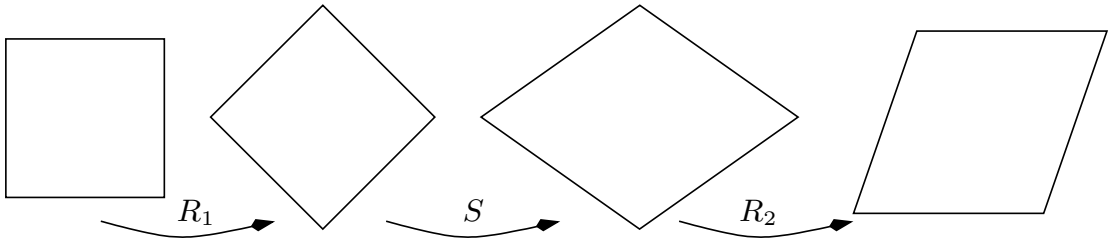
Basistransformationen

Rotation: Um z-Achse mit Winkel ϕ

$$R_{z,\phi} \cdot \mathbf{P} = \begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

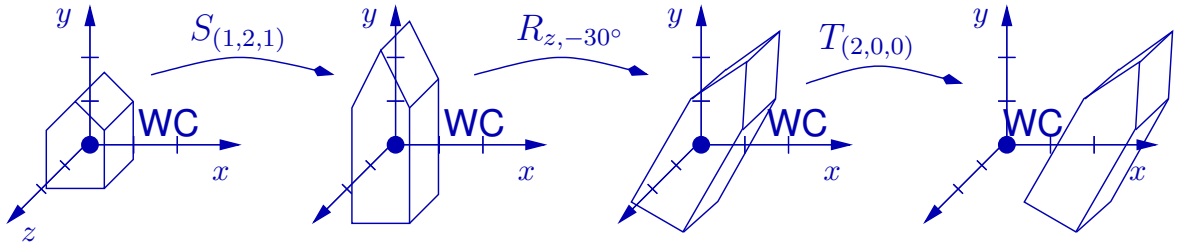


Beliebige affine Transformation: Jede affine Transformation lässt sich durch Hintereinanderausführung von Basistransformationen erzeugen.
Beispiel: Scherung als Kombination von Rotationen und Skalierung

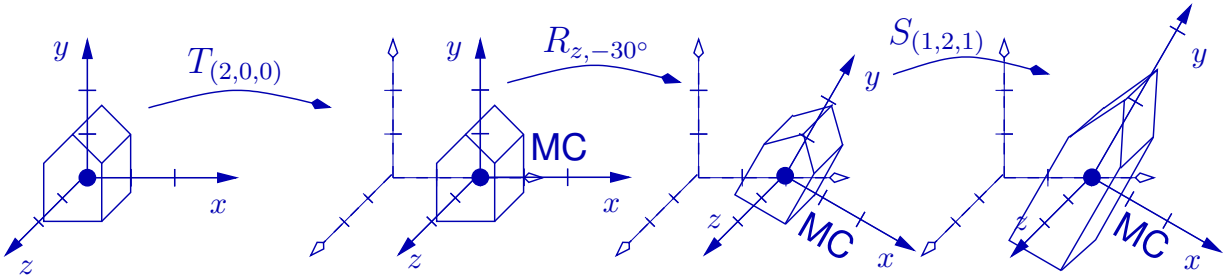


Kumulierte Transformationen

Sichtweise 1: Transformation bzgl. eines *festen (Welt-)Koordinatensystems (WC)*



Sichtweise 2: Transformation bzgl. des *lokalen (Modell-)Koordinatensystems (MC)*



Diese Sichtweise ist für die Modellierung wesentlich intuitiver!

Ziel: Transformation der Objekte in gemeinsames Koordinatensystem (WC)

Transformation MC $\rightarrow$ WC (Sichtweise 1) des Objektpunktes $\mathbf{P}^{MC}$

$$\mathbf{P}^{WC} = \left(T_{(2,0,0)} \left(R_{z,-30^\circ} \left(S_{(1,2,1)} \mathbf{P}^{MC}\right)\right)\right) = \left(T_{(2,0,0)} R_{z,-30^\circ} S_{(1,2,1)}\right) \mathbf{P}^{MC}$$

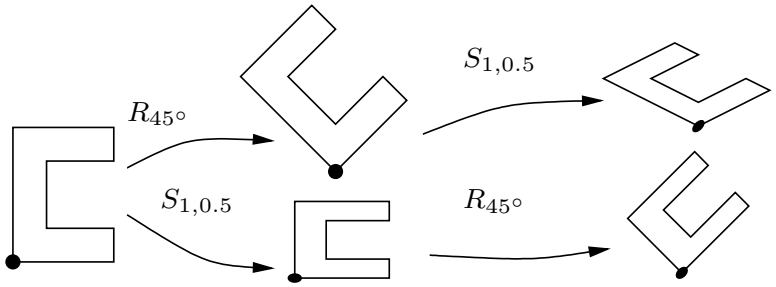
Zum vorherigen Beispiel ergibt sich $T^{MC\rightarrow WC} = T_{(2,0,0)} R_{z,-30^\circ} S_{(1,2,1)}$ zu

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \frac{1}{2} \begin{bmatrix} \sqrt{3} & 1 & 0 & 0 \\ -1 & \sqrt{3} & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{3}}{2} & 1 & 0 & 2 \\ -\frac{1}{2} & \sqrt{3} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Beispielsweise wird $(1, 0, 0)$ abgebildet auf

$$T^{MC\rightarrow WC} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \sqrt{3}/2 + 2 \\ -1/2 \\ 0 \end{pmatrix}$$

- Die Reihenfolge der Ausführung ist wichtig, z.B.



- Inverse der Basis-Transformationen sind einfach ermittelbar:

$$\left(T_{\vec{\mathbf{t}}}\right)^{-1} = T_{-\vec{\mathbf{t}}} \qquad \left(S_{s_x,s_y,s_z}\right)^{-1} = S_{\frac{1}{s_x},\frac{1}{s_y},\frac{1}{s_z}} \qquad \left(R_{z,\phi}\right)^{-1} = R_{z,-\phi}$$

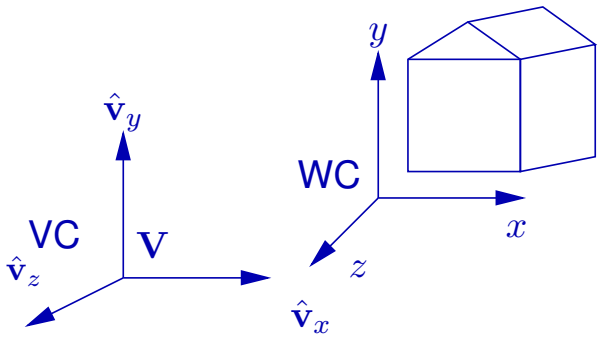
- Inverse Transformationsmatrix für kummulierte Basis-Transformationen

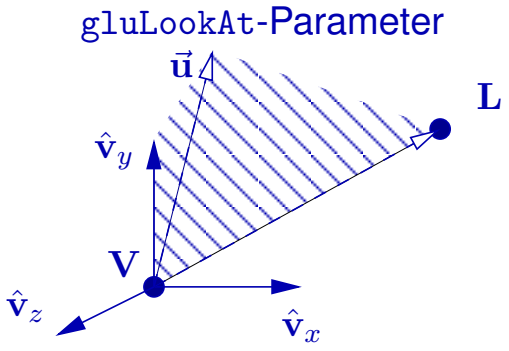
$$(A \cdot B)^{-1} = B^{-1} \cdot A^{-1}$$

5.3: Viewing Transformation

Ziel: Einbetten des Beobachters in die Szene durch eigenes Beobachterkoordinatensystem.

Ausgangssituation:
1. alle Objekte liegen in Weltkoordinaten vor
2. Rechtshändiges, orthonormales Beobachter-Koordinatensystem $\{\mathbf{V}, \hat{\mathbf{v}}_x, \hat{\mathbf{v}}_y, \hat{\mathbf{v}}_z\}$ in WC mit:
   - $\hat{\mathbf{v}}_x$: *Beobachter-Horizont*
   - $\hat{\mathbf{v}}_y$: Vertikale, d.h. $\hat{\mathbf{v}}_x - \hat{\mathbf{v}}_y$-Ebene ist die *Projektions-Ebene* für den Beobachter.
   - $\hat{\mathbf{v}}_z$: Blickrichtung (entlang -$\hat{\mathbf{v}}_z$)





5.3: Viewing Transformation

Die `glm::lookAt`-Funktion

`glm::lookAt` verwendet andere Kontrollparameter (ebenfalls in WC!):

- Beobachter-Position $\mathbf{V}$
- *Lookat-Punkt* $\mathbf{L}$: Vektor $\mathbf{L} - \mathbf{V}$ gibt Blickrichtung an ($-\hat{\mathbf{v}}_z$ Richtung)
- *up-Vektor* $\vec{\mathbf{u}}$ spannt zusammen mit $\mathbf{L} - \mathbf{V}$ die $\hat{\mathbf{v}}_y - \hat{\mathbf{v}}_z$-Ebene auf

Satz (Viewing-Transformation)

Gegeben: *Rechtshändiges, orthonormales Beobachter-Koordinatensystem* $\mathbf{V}, \hat{\mathbf{v}}_x, \hat{\mathbf{v}}_y, \hat{\mathbf{v}}_z$

Gesucht: *Viewing-Transformation* T_V *zum Wechsel von Weltkoordinaten (WC) in Viewkoordinaten (VC)*

Lösung:

$$T_V = \begin{bmatrix} & A^T & & -A^T\mathbf{V} \\ 0 & 0 & 0 & 1 \end{bmatrix} \in \mathbb{R}^{4\times 4} \quad \textit{mit} \quad A = (\hat{\mathbf{v}}_x, \hat{\mathbf{v}}_y, \hat{\mathbf{v}}_z)$$

Beispiel Viewing-Transformation

Gegeben: $\mathbf{V} = (1,0,1)^T, \quad \mathbf{L} = (0,0,0)^T, \quad \vec{\mathbf{u}} = (2,1,2)^T$

Berechnung: Beobachter-Koordinatensystems und Viewing-Transformation

$$\hat{\mathbf{v}}_z = \frac{\mathbf{V}-\mathbf{L}}{\|\mathbf{V}-\mathbf{L}\|} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix},$$

$$\vec{\mathbf{v}}_y = \vec{\mathbf{u}} - (\vec{\mathbf{u}} \cdot \hat{\mathbf{v}}_z)\hat{\mathbf{v}}_z = \begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix} - \tfrac{1}{2}\left(\begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}\right)\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

$$\hat{\mathbf{v}}_x = \hat{\mathbf{v}}_y \times \hat{\mathbf{v}}_z = \tfrac{1}{\sqrt{2}}\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \tfrac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}, \; A = (\hat{\mathbf{v}}_x, \hat{\mathbf{v}}_y, \hat{\mathbf{v}}_z) = \tfrac{1}{\sqrt{2}}\begin{pmatrix} 1 & 0 & 1 \\ 0 & \sqrt{2} & 0 \\ -1 & 0 & 1 \end{pmatrix}$$

$$A^T\mathbf{V} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 0 & -1 \\ 0 & \sqrt{2} & 0 \\ 1 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \sqrt{2} \end{pmatrix}, \quad T_V = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & -\sqrt{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Aufgaben der Projektionstransformation

Projektionstransformationen legen den *Sichtbereich des Beobachters* und die *Projektionsart* fest:

Sichtbereich: Teilmenge des $3D$ in VC, die im Bild dargestellt wird

Projektionsart: Wie wird die $3D$-Geometrie später auf das $2D$-Bild projiziert?

Unterscheide folgende Projektionstransformationen

- *Orthographisch:* Sichtbereich ist ein *Quader* (Parallelprojektion)
- *Perspektivisch:* Sichtbereich ist ein *Pyramidenstumpf* mit Spitze im Beobachterpunkt

Ergebnis der Projektionstransformation:

- Der Sichtbereichs liegt in *Normalized Device Coordinates (NDC)* als $[-1,1]^3$ vor.
- Das $2D$-Bild entsteht anschließend durch *Parallelprojektion* in NDC entlang z-Achse
- NDC ist *linkshändiges* Koordinatensystem, damit

$+z \propto$ „Entfernung zum Beobachter“

Orthographische Transformation

Sichtbereich: Achsenparalleles Rechteck in VC:

$$[x_{min}, x_{max}] \times [y_{min}, y_{max}] \times [z_{min}, z_{max}]$$

Transformation: Mit
$\Delta x = x_{max} - x_{min}, \Delta y = y_{max} - y_{min}, \Delta z = z_{max} - z_{min}$:

$$T_O \mathbf{P} = \begin{bmatrix} 2/\Delta x & 0 & 0 & -2\frac{x_{min}}{\Delta x} - 1 \\ 0 & 2/\Delta y & 0 & -2\frac{y_{min}}{\Delta y} - 1 \\ 0 & 0 & -2/\Delta z & 2\frac{z_{min}}{\Delta z} + 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{2(p_x - x_{min})}{\Delta x} - 1 \\ \frac{2(p_y - y_{min})}{\Delta y} - 1 \\ \frac{2(z_{min} - p_z)}{\Delta z} + 1 \\ 1 \end{bmatrix} \in [-1, 1]^3$$

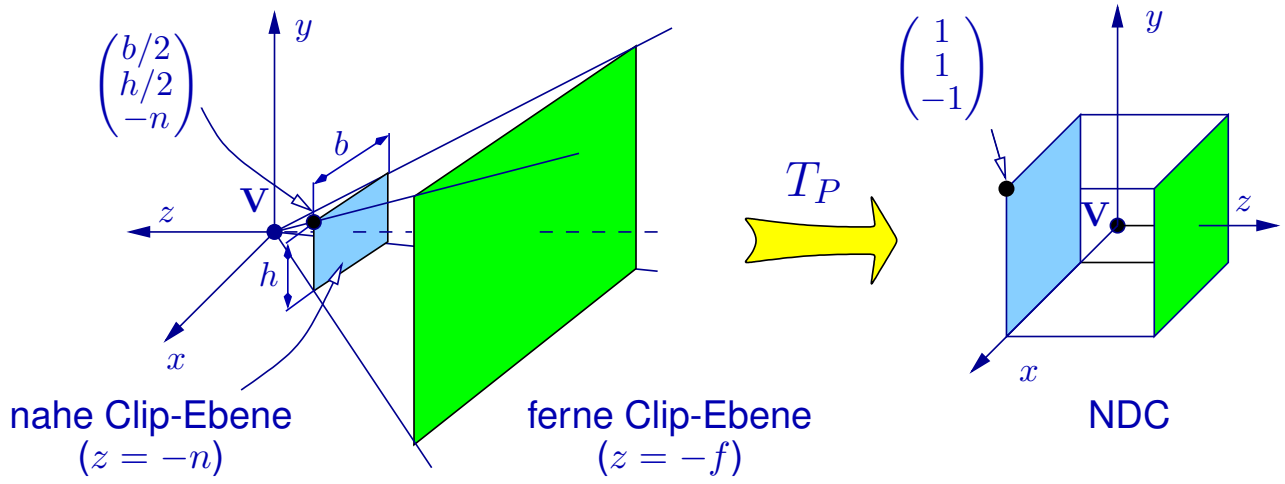
entspricht einer kombinierten Translation und Skalierung

glm-Funktion: `ortho(...)`

Perspektivische Transformation

Sichtbereich: *Viewing Frustum* (Pyramidenstumpf) mit Spitze im Beobachterpunkt $\mathbf{V}$ im Ursprung:

Abstand nahe Clip-Ebene:	$z = -n$	Abstand ferne Clip-Ebene:	$z = -f$
Breite nahe Clip-Ebene:	b	Höhe nahe Clip-Ebene:	h



Satz (Perspektivische Transformation)

Die perspektivische Transformation T_P wird durch folgende 4×4-Matrix beschrieben:

$$T_P = \begin{bmatrix} \frac{2}{b} & 0 & 0 & 0 \\ 0 & \frac{2}{h} & 0 & 0 \\ 0 & 0 & \frac{n+f}{n(n-f)} & \frac{2f}{n-f} \\ 0 & 0 & -\frac{1}{n} & 0 \end{bmatrix}$$

glm-Funktionen: `frustum(...)`, `perspective(...)`

Beachte:

- T_P ist **keine** affine Transformation, da nach Anwendung i.a. $w = -z/n \neq 1$
- Perspektivische Transformation nicht winkel-erhaltend, damit ungeeignet für Raytracing und Beleuchtungsberechnung (diese muss vorher stattfinden)

1. Geraden werden auf Geraden abgebildet, d.h.

   Für $\mathbf{P}_i = (x_i, y_i, z_i)$ und $\mathbf{P}(\alpha) = (1-\alpha)\mathbf{P}_1 + \alpha\mathbf{P}_2$ in VC

   gilt: $T_P(\mathbf{P}(\alpha)) = (1-\beta)T_P(\mathbf{P}_1) + \beta T_P(\mathbf{P}_2)$, mit $\beta = \beta(\alpha) = \dfrac{\alpha z_2}{(1-\alpha)z_1 + \alpha z_2}$

2. Ordnung von Punkten auf Geraden $\mathbf{P}(\alpha)$ bleibt erhalten, d.h.

   $$\alpha_1 < \alpha_2 \Leftrightarrow \beta(\alpha_1) < \beta(\alpha_2)$$

3. Geraden durch den Beobachterpunkt (in VC $\mathbf{V} = \mathcal{O}$) werden zu z-achsenparalleln Geraden, denn es gilt

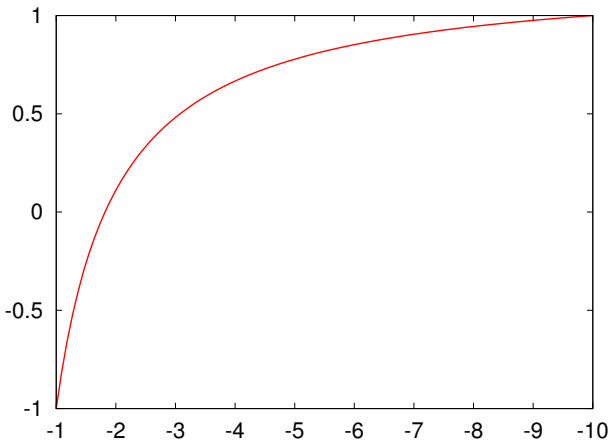
   $$\mathbf{P}(\alpha) = (1-\alpha)\mathcal{O} + \alpha\mathbf{Q} = \begin{pmatrix} \alpha q_x \\ \alpha q_y \\ \alpha q_z \end{pmatrix} \Longrightarrow T_P(\mathbf{P}(\alpha)) = \begin{pmatrix} -\frac{2nq_x}{bq_z} \\ -\frac{2nq_y}{hq_z} \\ * \end{pmatrix}$$

4. Tiefenabstände werden verzerrt, genauer:

> Abstände nahe am bzw. fern vom Beobachter werden vergrößert bzw. verkleinert

Konkret: z-Wert von $T_P(x, y, z)$:

$$-\frac{n}{z}\left(\frac{z(n+f)}{n(n-f)}+\frac{2f}{n-f}\right)=-\left(\frac{2nf}{z(n-f)}+\frac{n+f}{n-f}\right)$$



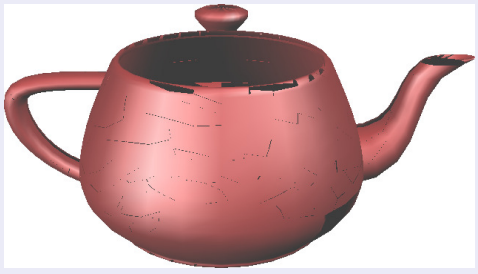
z-Wertes nach perspektivischer Transformation ($n = 1, f = 10$)

Beispiel (Verzerrung Tiefenabstände - Quantisierung)

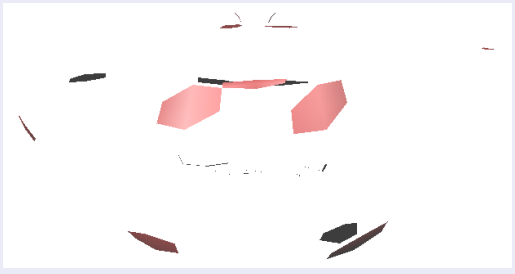
Beispielszene *mit verschiedenen Angaben der nahen Clippingebene*

$n = 10^{-1}$

$n = 1.25 \cdot 10^{-6}$



$n = 1.6 \cdot 10^{-7}$



$n = 10^{-8}$

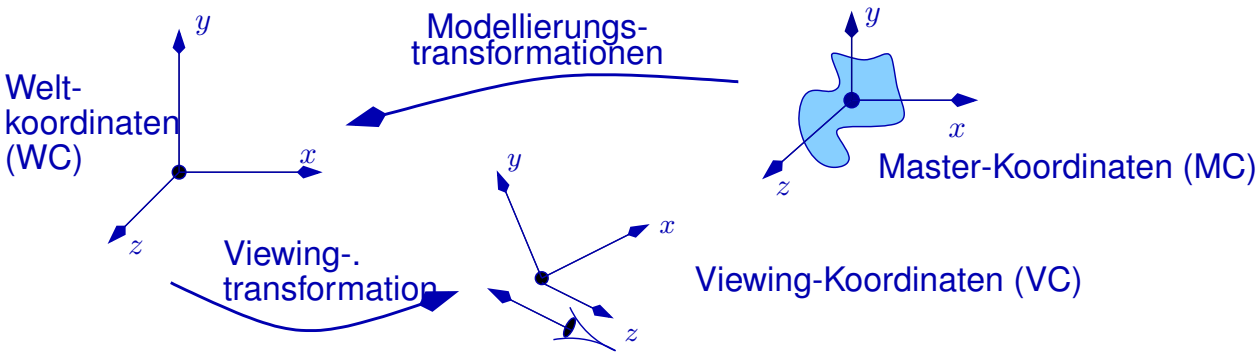
Ziel: Datenstruktur zur Beschreibung von Objekten und Transformationen in einer *Hierarchie*

Koordinatensysteme, zwischen denen zu unterscheiden ist:

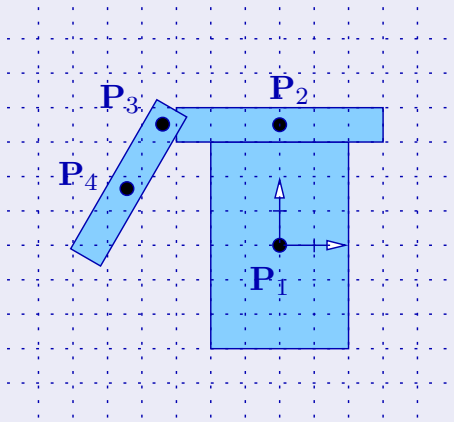
Master- bzw. Modellkoordinaten (MC): System zur Beschreibung **eines** Objektes/Primitives.

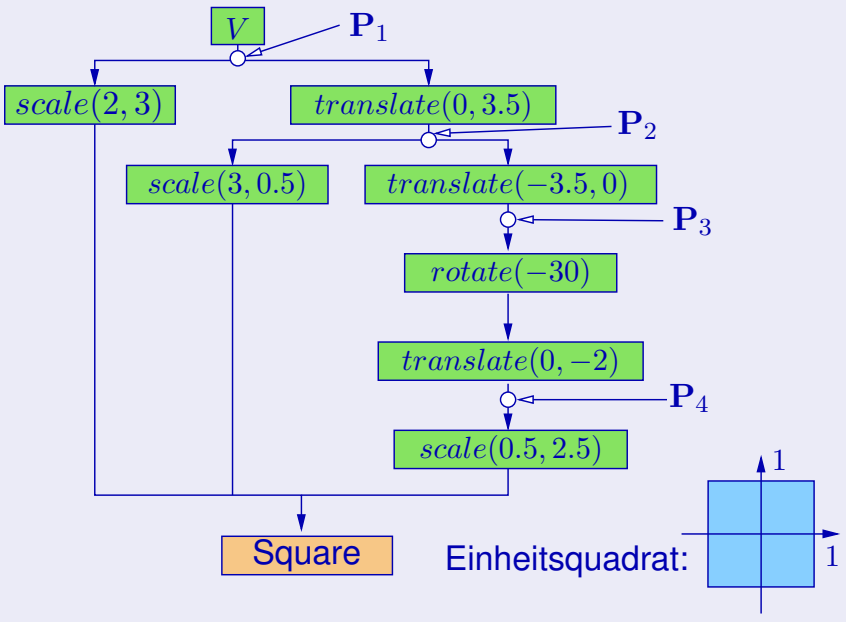
Weltkoordinaten (WC): Zusammenhang zwischen allen Objekten und Beobachter.

View-Koordinaten (VC): Beobachterposition und Blickrichtung



Beispiel ($2D$-Situation)





$\mathbf{P}_1$: Zentrum Weltkoordinaten
$\mathbf{P}_2$: Zentrum Schulterpartie
$\mathbf{P}_3$: Rotationszentrum Schulter
$\mathbf{P}_4$: Zentrum Oberarm

Beachte: *Mehrfachverwendung des Quadrates, von V und einer Translation*

Gerichteter azyklischer Graph (*DAG*)

DAG wird auch *Szenengraph* genannt

Ziel: Hierarchische Beschreibung der Transformationen und Geometrien.

Elemente des Graphen sind

- *Knoten:* Transformations- und Geometrieknoten
- *Wurzel:* Viewingtransformation V
- *Blätter:* Geometriebeschreibungen der Primitive
- *Kanten:* Stellen gerichtete Verbindungen zwischen Knoten her

Regeln zur Interpretation und Umsetzung des DAG in eine Szene

- Ein Pfad von der Wurzel zu einem Blatt (Geometrie) entspricht einem gezeichneten Primitiv
- Mehrfachverwendung: Knoten können mehrere Vorgänger haben
- Zyklen: Der Graph darf keine Zyklen enthalten

Matrixstack

Matrix-Stack: Wird weder von OpenGL, noch von glm zur Verfügung gestellt. Wir nutzen die eigene Klasse `MatrixStack`

Matrixstack: Wichtige Speicherstruktur zur Umsetzung eines DAG

Current Transformation Matrix (CTM): Oberstes Element des Stack

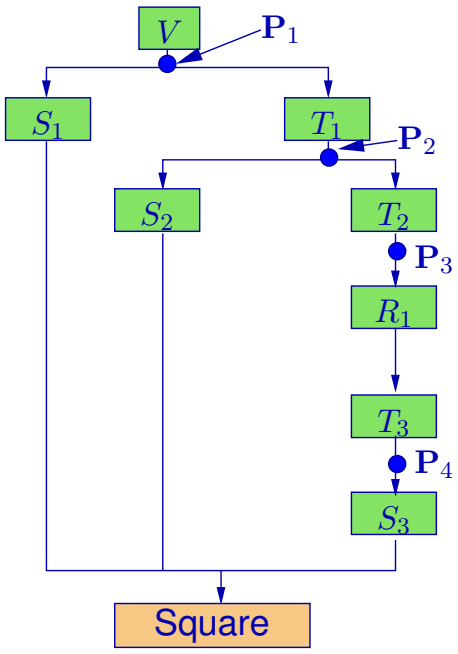
- Modell-View-Matrix: Transformation von Modell- in View-Koordinaten;wird auf *Modell-View-Stack* verwaltet
- Projection-Matrix: Transformation von View-Koordinaten in NDC.
- Die CTMs werden an Vertex-Shader übergeben und dort verwendet
- Umsetzung der Sichtweise 2: Transformation[`translate()`,`rotate()`, `scale()`, `ortho()`, `perspective()` *von rechts* an CTM multiplizieren.

Gespeicherte Matrizen sind wiederverwendbare „Zwischenergebnisse“

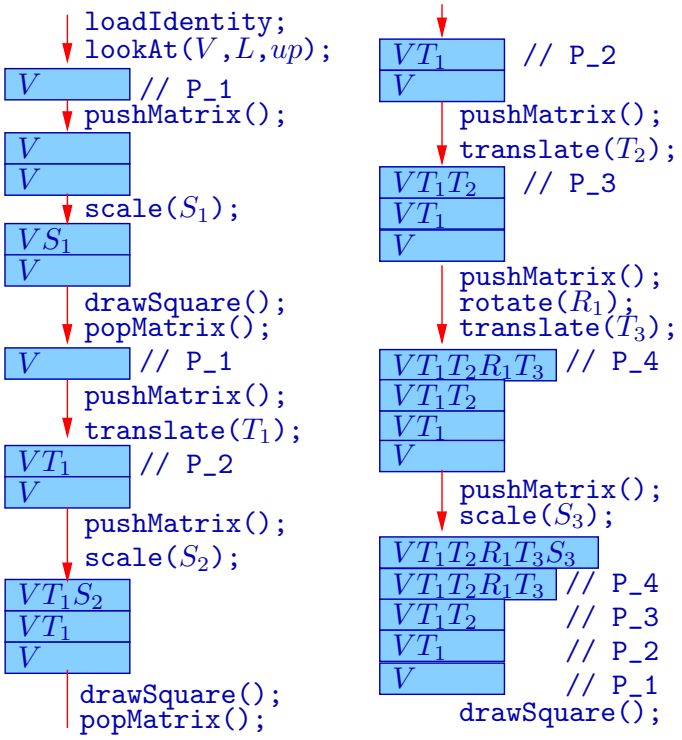
Operationen auf dem Stack:

`loadIdentity()`, `loadMatrix()`: Setzen der CTM
`push()`: Push des Stack und Duplizieren der CTM
`pop()`: pop vom Stack

Der DAG:



Abarbeitung mit Matrix-Stack:



glm Transformationen

OpenGL kennt nur Matrizen (z.B. `mat4`) und Vektoren (z.B. `vec3`)

Projektionsmatrizen mit glm: Funktionen zur Matrix-Erzeugung:

- `ortho(left, right, bottom, top, zNear, zFar)`
- `perspective(fovy, aspect, zNear, zFar)`, wobei `foyv` (field-of-view 'y') der Öffungswinkel in y-Richtung und `aspect` das Verhältnis von horizontalem zu vertikalem Öffungswinkel ist.

Basis-Transformationen mit glm: Funktionen zur Matrix-Erzeugung:

- `translate(mat4 M, vec3 v)`: Verschiebung um 3D-Vektor `v`
- `scale(mat4 M, vec3 s)`: Skalierung mit 3D Vektor `v`
- `rotate(mat4 M, float angle, vec3 v)`: Rotation um Achse `v` mit Winkel `angle`

Beachte: `translate, scale, rotate` bestimmen *die kummulierte Transformationsmatrix* aus `M * T` (`T` ist die entspr. Translations-, Skalierungs- bzw. Rotatationsmatrix) ⇒ *Sichtweise 2*, Transformation in lokalen Koordinaten.

Die Klasse `MatrixStack`

Grundlage ist die Standard Template Klasse, parametrisiert zur Verwaltung von 4×4-Matrizen: `std::stack<glm::mat4>`

Erweiterung der Stack-Klasse um Initialisierungs- und Transformations-Operatoren, z.B.

```
virtual void MatrixStack::loadMatrix(glm::mat4 const &m)
{ static_cast <std::stack<glm::mat4>*> (this)->top() = m; }

glm::mat4 MatrixStack::translate(glm::vec3 const &v)
{
  return static_cast <std::stack<glm::mat4>*> (this)->top() =
    glm::translate (static_cast <std::stack<glm::mat4>*> (this)->top(), v);
}

glm::mat4 MatrixStack::lookAt (glm::vec3 const &eye, glm::vec3 const &center,
                               glm::vec3 const &up)
{
  return static_cast <std::stack<glm::mat4>*> (this)->top() =
    static_cast <std::stack<glm::mat4>*> (this)->top() *
  glm::lookAt(eye, center, up);
}
```

Von-rechts Multiplikation an die CTM (oberstes Stack-Element `top`) realisiert die *Sichtweise 2* (Transformation in lokalen Koordinaten)

Hello-World OpenGL-Programm

```
void drawGeometry(unsigned int count) {
  // zeichne drei Dreiecke: Rot, grün und blau
  (...)

  // Hierarchie: Rekursion mit 3 skalierten Nachfolgern
  GLint loc = glGetUniformLocation(prg, "modviewMat");
  if ( count < maxCount ) {
    modviewStack.push();
      modviewStack.translate(vec3(0.4, 0.4, 0.4));
      modviewStack.scale(vec3(0.5, 0.5, 0.5));
      glUniformMatrix4fv(loc, "modviewMat", 1, GL_FALSE,
                         value_ptr(modviewStack.top()))
      modviewStack.pop();
    modviewStack.pop();
    modviewStack.push();
      modviewStack.translate(vec3(-0.6, 0.4, 0.4));
      modviewStack.scale(vec3(0.5, 0.5, 0.5));
      glUniformMatrix4fv(loc, "modviewMat", 1, GL_FALSE,
                         value_ptr(modviewStack.top()))
      drawGeometry(count+1);
    modviewStack.pop();
    modviewStack.push();
      modviewStack.translate(vec3(0.4, -0.6, 0.4));
      modviewStack.scale(vec3(0.5, 0.5, 0.5));
      glUniformMatrix4fv(loc, "modviewMat", 1, GL_FALSE,
                         value_ptr(modviewStack.top()))
      drawGeometry(count+1);
    modviewStack.pop();
  }
}
```

```
void GLWidget::paintGL() {
  (...)
  // Initialisiere projection Matrix & model-view Stack
  mat4 projMat = perspective(...);
  modviewStack.loadIdentity();
  modviewStack.lookAt(...);

  // Übergabe der uniform Parameterer
  glUseProgram(prg);
  (...)
  glUniformMatrix4fv(glGetUniformLocation(prg, "projMat"),
    1, GL_FALSE, value_ptr(projMat));
  glUniformMatrix4fv(glGetUniformLocation(prg, "modviewMat"),
    1, GL_FALSE, value_ptr(modviewStack.top()));

  drawGeometry(0);
```

Hinweis: *Rekursive* `drawGeometry` erhält Skalierung *bewusst*.