h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Visual Computing

Grafische Objekte und deren Programmierung

Hochschule Darmstadt

Björn Frömmer

Prof. Dr. Benjamin Meyer

KAPITEL 7

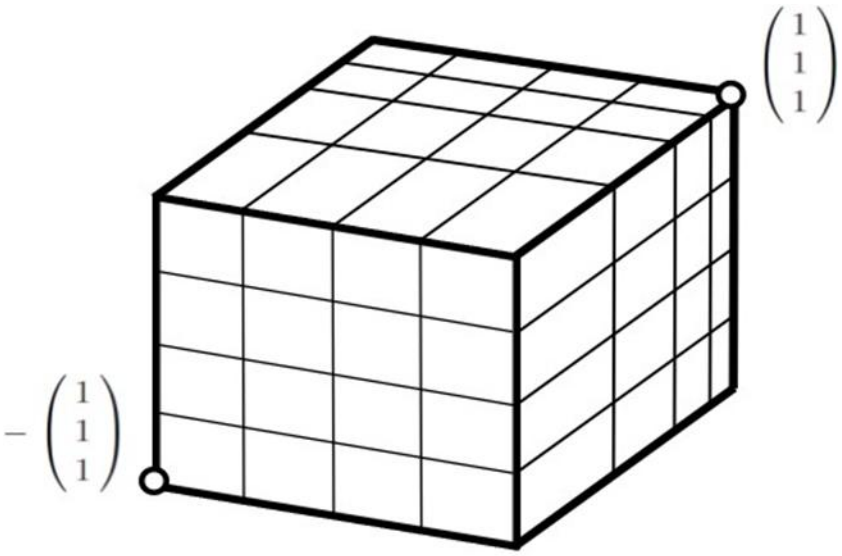
Kameras

NDC == Normalized Device Coordinates

Der Normalized Device Space (NDC)

- Bis jetzt haben wir intuitiv unsere Geometrie immer im Wertebereich $[-1, 1]^3$ definiert und angezeigt.
  - Dieser (normierte) Bereich wird auch **Normalized Device Space** genannt.
  - Nur Vertices innerhalb dieses Würfels werden letztlich gerendert.

- Doch die Frage, welche Geometrie gerendert wird, sollte eigentlich von der Betrachter-/Kameraposition abhängen!

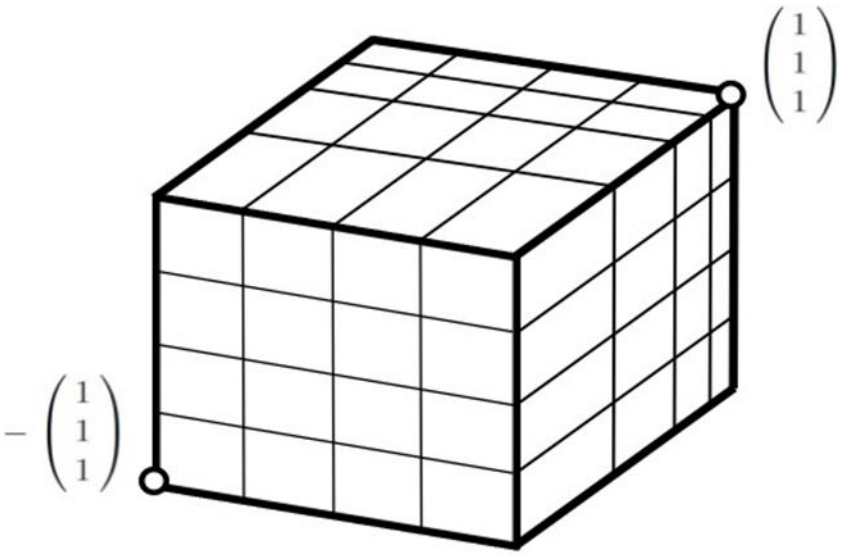


Normalized Device Space:
Würfel von (-1, -1, -1) bis (1, 1, 1)

Kameraausrichtung

Frage:

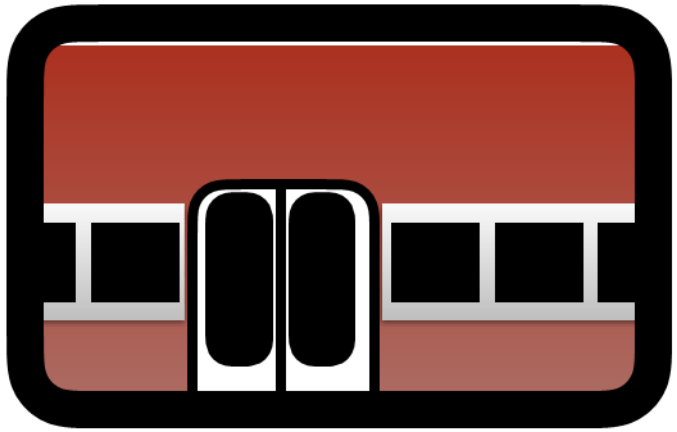
- Wie würden Sie eine Kamera realisieren, die sich an einer beliebigen Position und mit beliebiger Ausrichtung befindet, indem Sie ausschließlich Transformationen verwenden?



Normalized Device Space:
Würfel von (-1, -1, -1) bis (1, 1, 1)

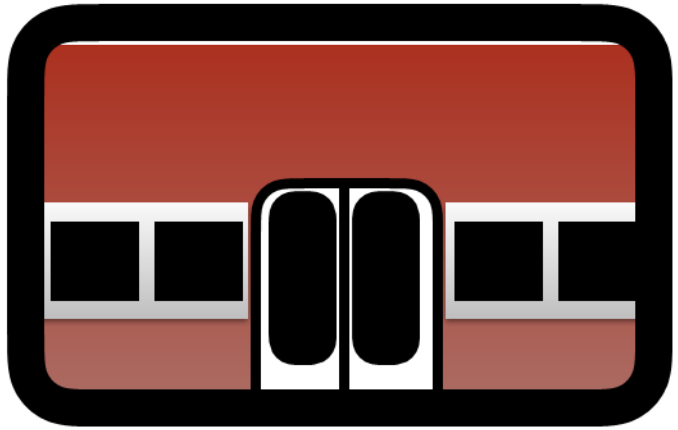
Blendenproblem

Situation: Sie sitzen in einem Zug und schauen aus dem Fenster.



Blendenproblem

Situation: Sie sitzen in einem Zug und schauen aus dem Fenster.

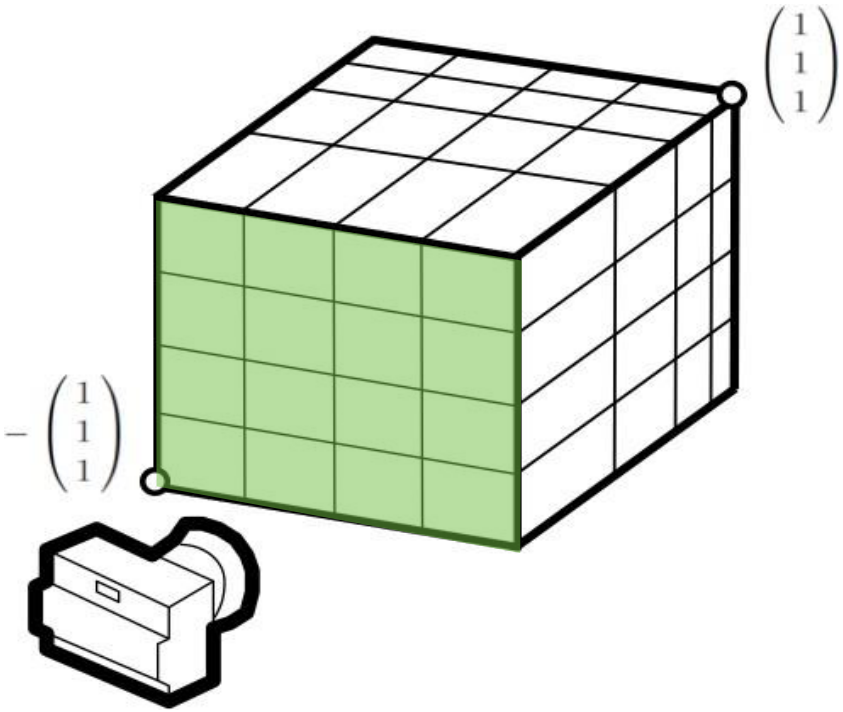


- Fährt Ihr Zug oder der andere?
- Mit einem begrenzten Blick auf die Szene unmöglich zu entscheiden

Kameraausrichtung

Frage:

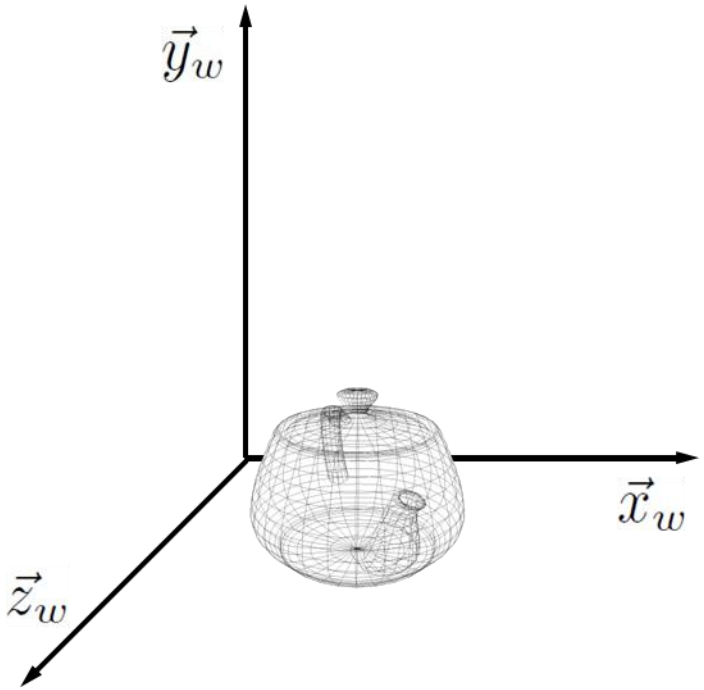
- Wie würden Sie eine Kamera realisieren, die sich an einer beliebigen Position und mit beliebiger Ausrichtung befindet, indem Sie ausschließlich Transformationen verwenden?



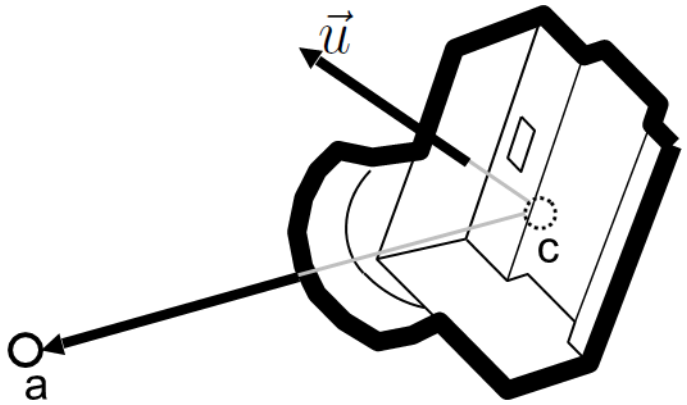
Antwort:

- Anstatt die Kamera zu bewegen/drehen, bewegen/drehen Sie die Szene!
  - Gleiches Ergebnis, effizienter in der Umsetzung.
  - Außerdem wissen wir bereits, wie man das macht!
- Ziel: Kamera liegt im Ursprung und schaut entlang der negativen Z-Achse

Äußere (extrinsische) Kameraparameter



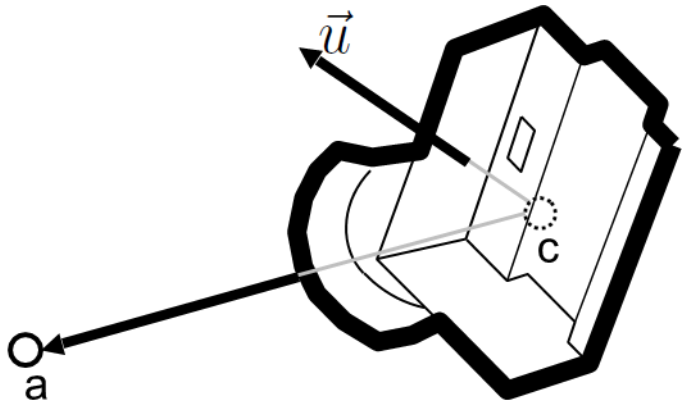
- Weltkoordinatensystem
- Platzierung von Objekten



- Kamerazentrum $\vec{c}$
- Betrachtungspunkt $\vec{a}$
- up-Vektor $\vec{u}$

View Transformation

- Die Transformation der Szene kann wiederum als Matrixmultiplikation ausgedrückt werden!
  - Erinnerung: Rechtshändiges Koordinatensystem
  - Standardmäßig guckt die Kamera vom Nullpunkt entlang der negativen Z-Achse.

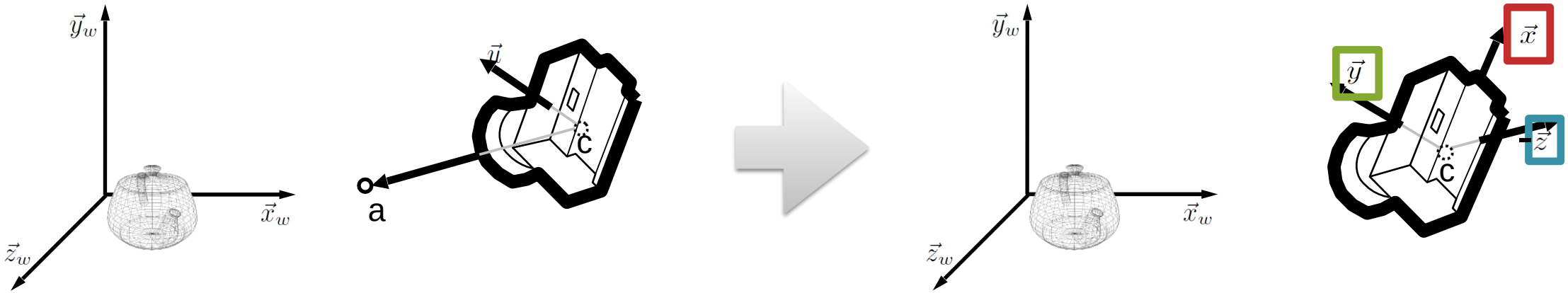


- Diese sog. **View Matrix** kann automatisch berechnet werden:

```
lookAt( cx, cy, cz,        // Kamerazentrum c
        ax, ay, az,        // Betrachtungspunkt a
        ux, uy, uz );      // up-Vektor u
```

- Das resultierende Koordinatensystem wird als **View Space** bezeichnet.

View Transformation Matrix

- 

$$V = \begin{pmatrix} x_{x_w} & x_{y_w} & x_{z_w} & 0 \\ y_{x_w} & y_{y_w} & y_{z_w} & 0 \\ z_{x_w} & z_{y_w} & z_{z_w} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -c_{x_w} \\ 0 & 1 & 0 & -c_{y_w} \\ 0 & 0 & 1 & -c_{z_w} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Vertex Transformationen bis jetzt

Model Transformation M_{Model}: Akkumulierte Transformationen, platziert Objekte in der Szene (Welt)

View Transformation V: Rotiert und transliert die gesamte Szene, so dass die Kamera im Ursprung liegt und entlang der negativen Z-Achse schaut.

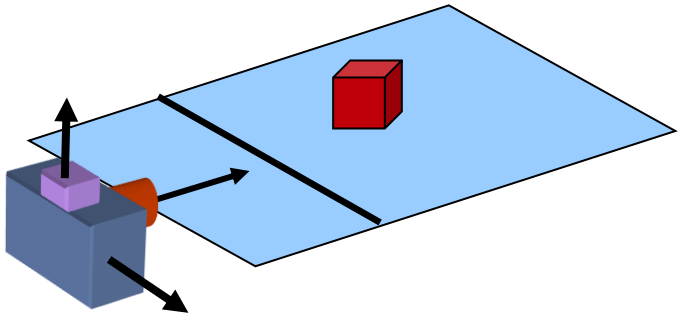
$$\mathbf{p}' = V \cdot \underbrace{T \cdot \ldots \cdot T \cdot S \cdot R}_{M_{Model}} \cdot \mathbf{p}$$

logische Reihenfolge der Ausführung

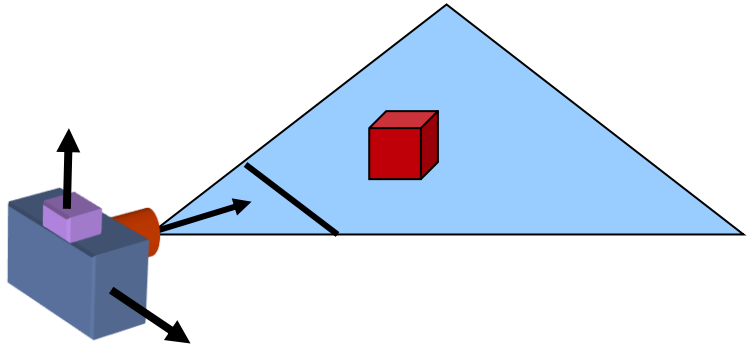
Innere (intrinsische) Kameraparameter

- Die extrinsischen Kameraparameter (Position & Ausrichtung) sind durch die View-Matrix bereits integriert, es fehlen noch die intrinsischen Parameter:
  - Vereinfacht: Welcher Bereich der Szene soll gerendert werden
  - Bei einer echten Kamera entspräche das dem Öffnungswinkel und der Brennweite

- Projektionsmöglichkeiten:



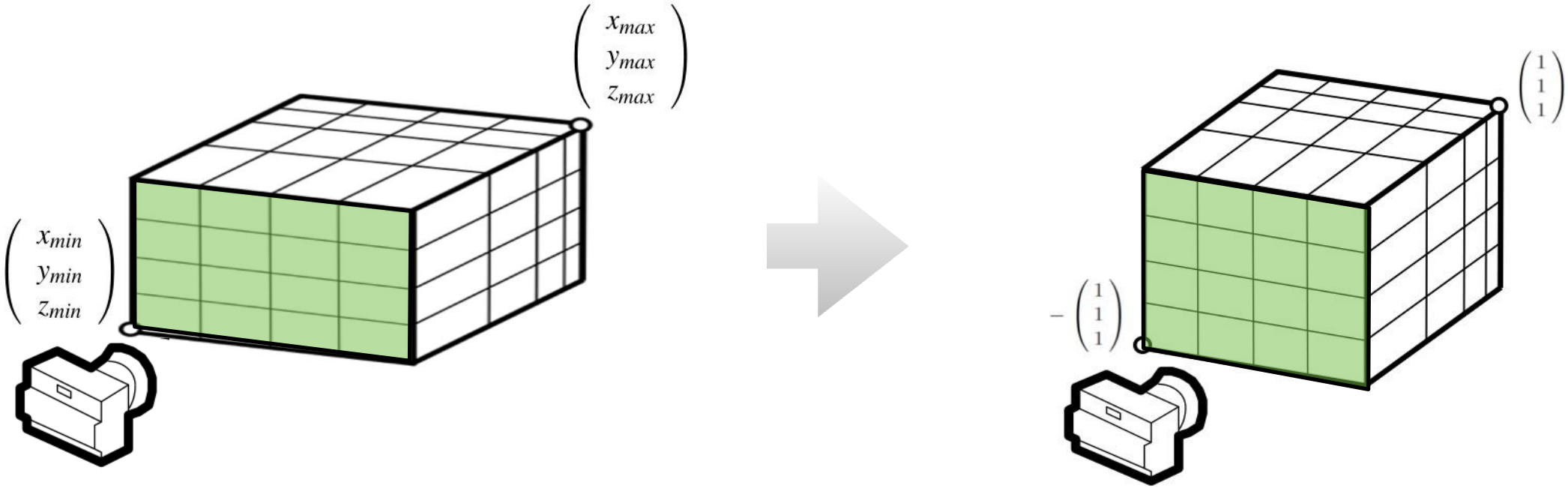
Parallel- bzw. orthografische Projektion



Perspektivische Projektion

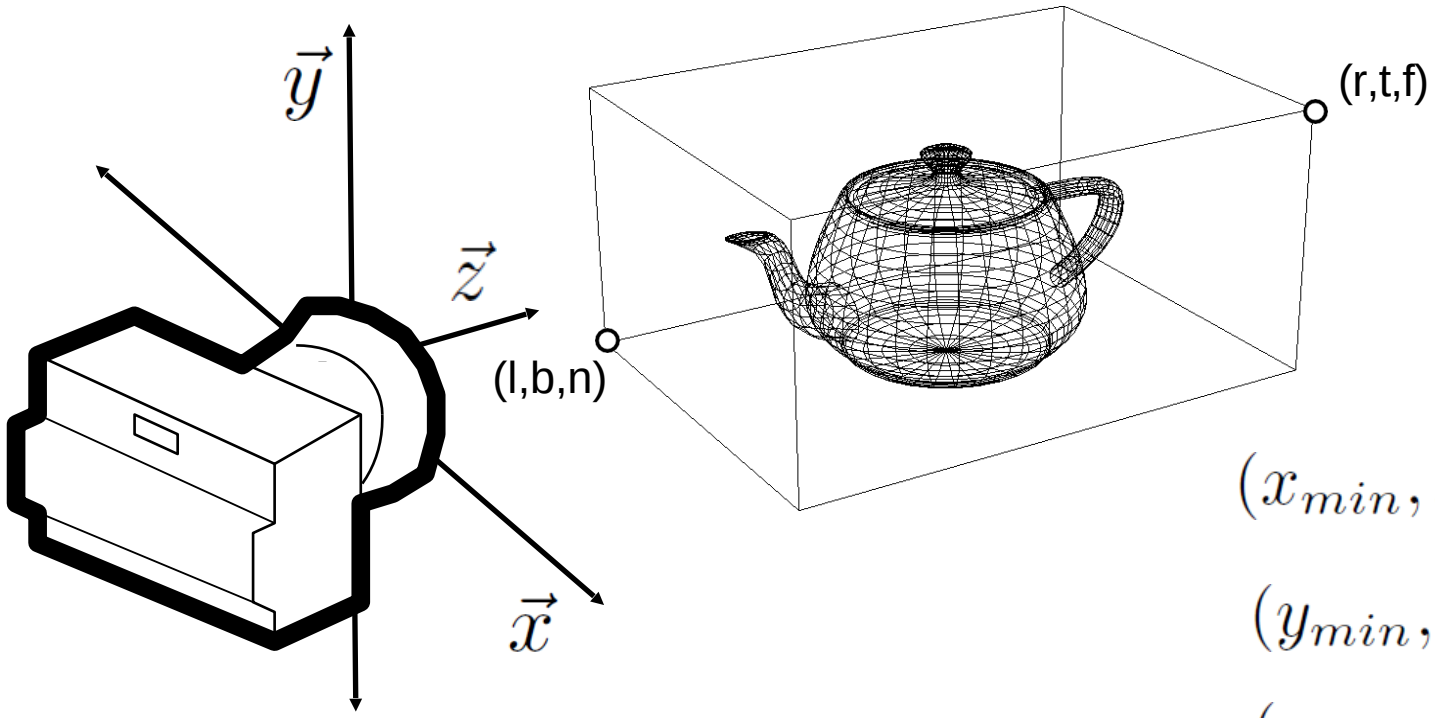
Orthografische Projektion

- **Unterschiedliche Seitenverhältnisse** können simuliert werden, indem die gesamte Szene transformiert wird



Orthografische Projektion II

- Definition des zu rendernden Volumens und Transformation in ein kanonisches Volumen (Normalized Device Space)



$$(x_{min}, x_{max}) = (left, right)$$

$$(y_{min}, y_{max}) = (bottom, top)$$

$$(z_{min}, z_{max}) = (near, far)$$

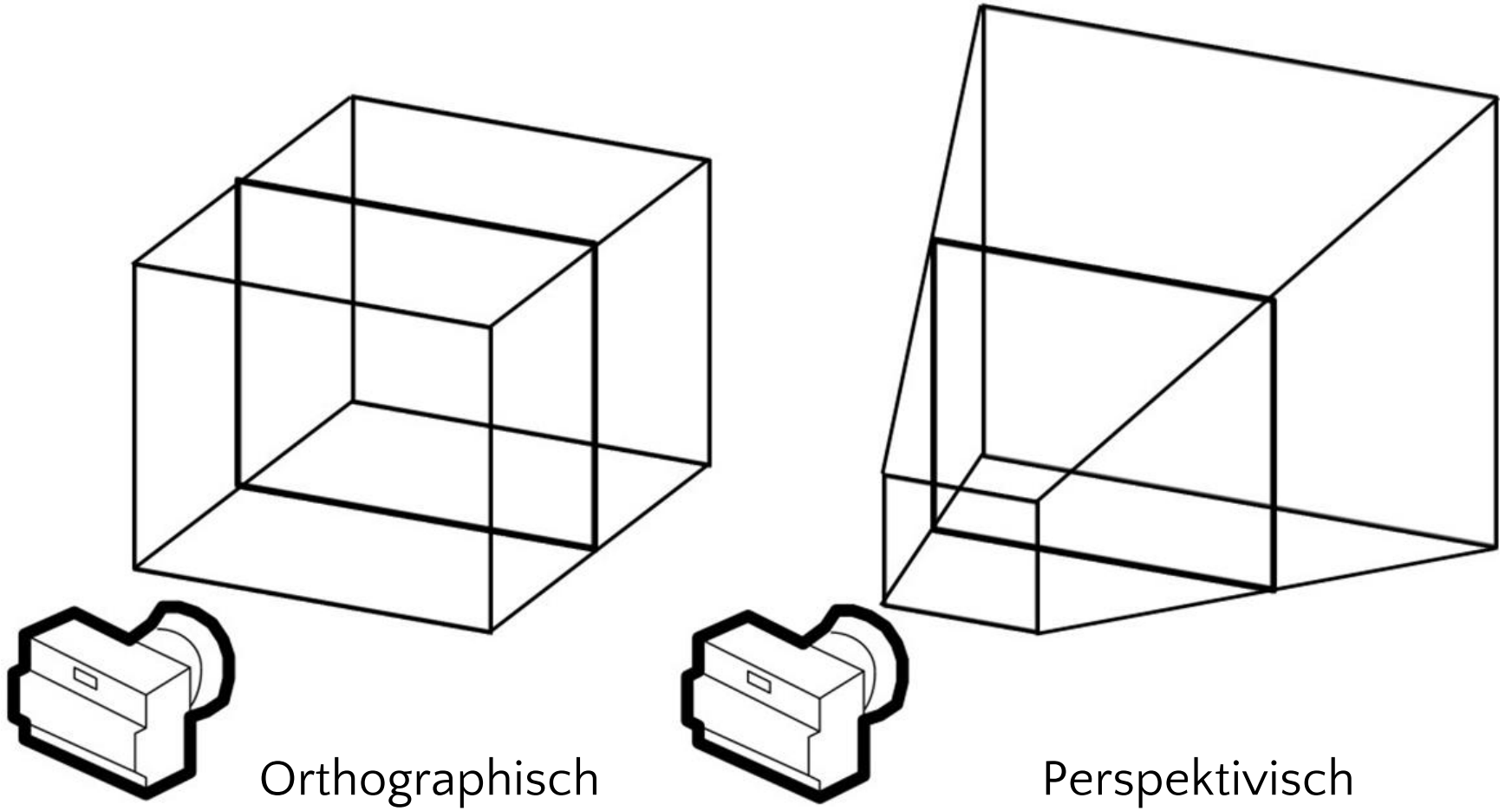
Orthografische Projektion III

- Setzen des Volumenraums und Berechnung der Projektionsmatrix geschieht durch:

```
glOrtho(  GLdouble left, GLdouble right,
          GLdouble bottom, GLdouble top,
          GLdouble near, GLdouble far);
```

- Vergessen Sie nicht, auch Ihre Fensterauflösung anzupassen
  - Das Seitenverhältnis der Kamera und des Fensters sollten gleich sein, um Verzerrungen zu vermeiden

Unterschiedliche Projektionen



Perspektivische Wahrnehmung

- Perspektivische Wahrnehmung bedeutet:
- Je weiter hinten das Objekt ist, desto kleiner erscheint es uns.
- Wird diese Regel gebrochen erhalten wir bspw. eine fehlerhafte Größeninformation.

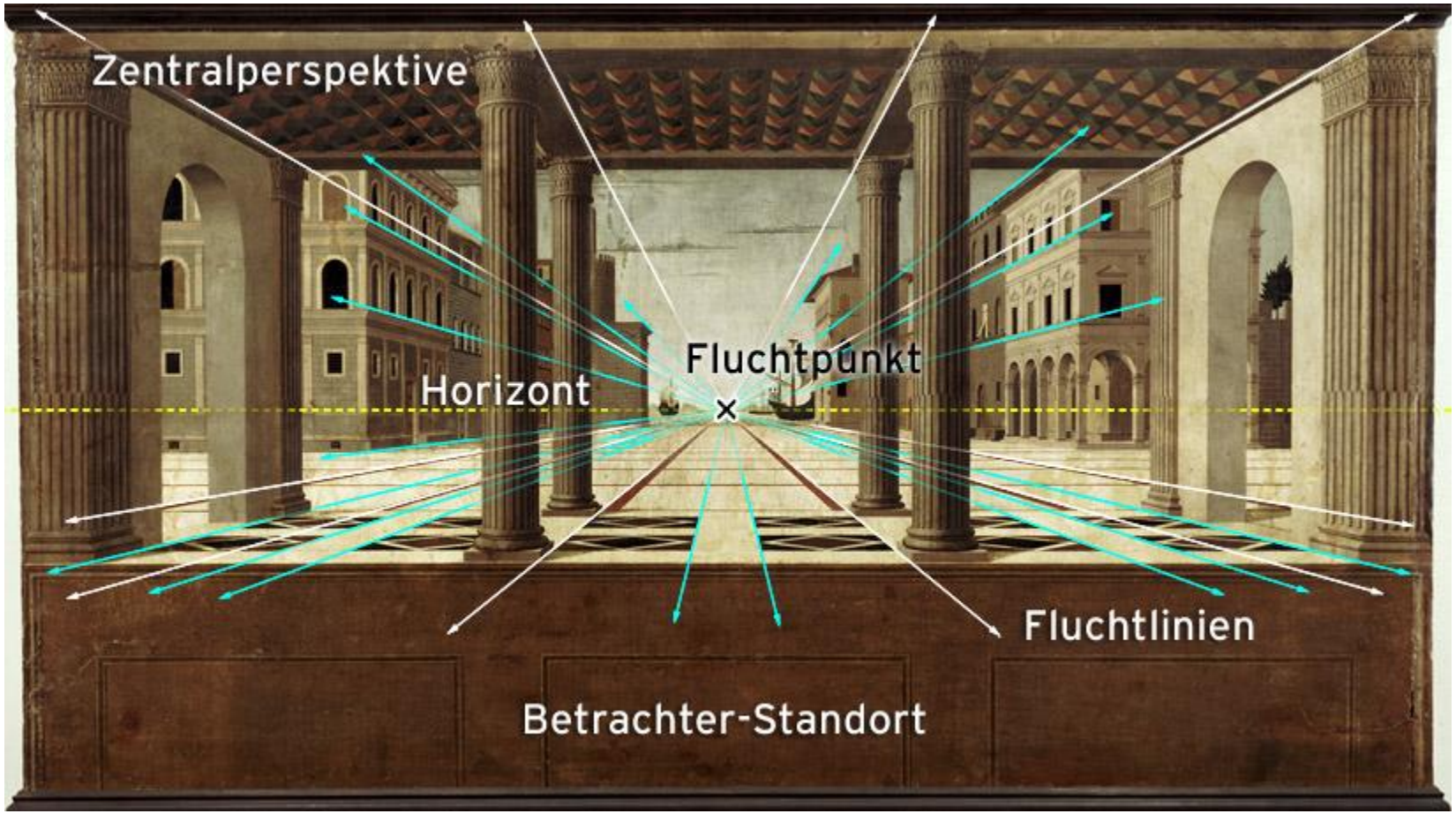
Quelle: Wikipedia – Optische Täuschungen

Perspektivische Wahrnehmung II

Perspektivische Projektion

- „Natürliche Darstellung“
- Am meisten verwendete Projektionsart
- Sehstrahlen treffen sich in einem **Fluchtpunkt**

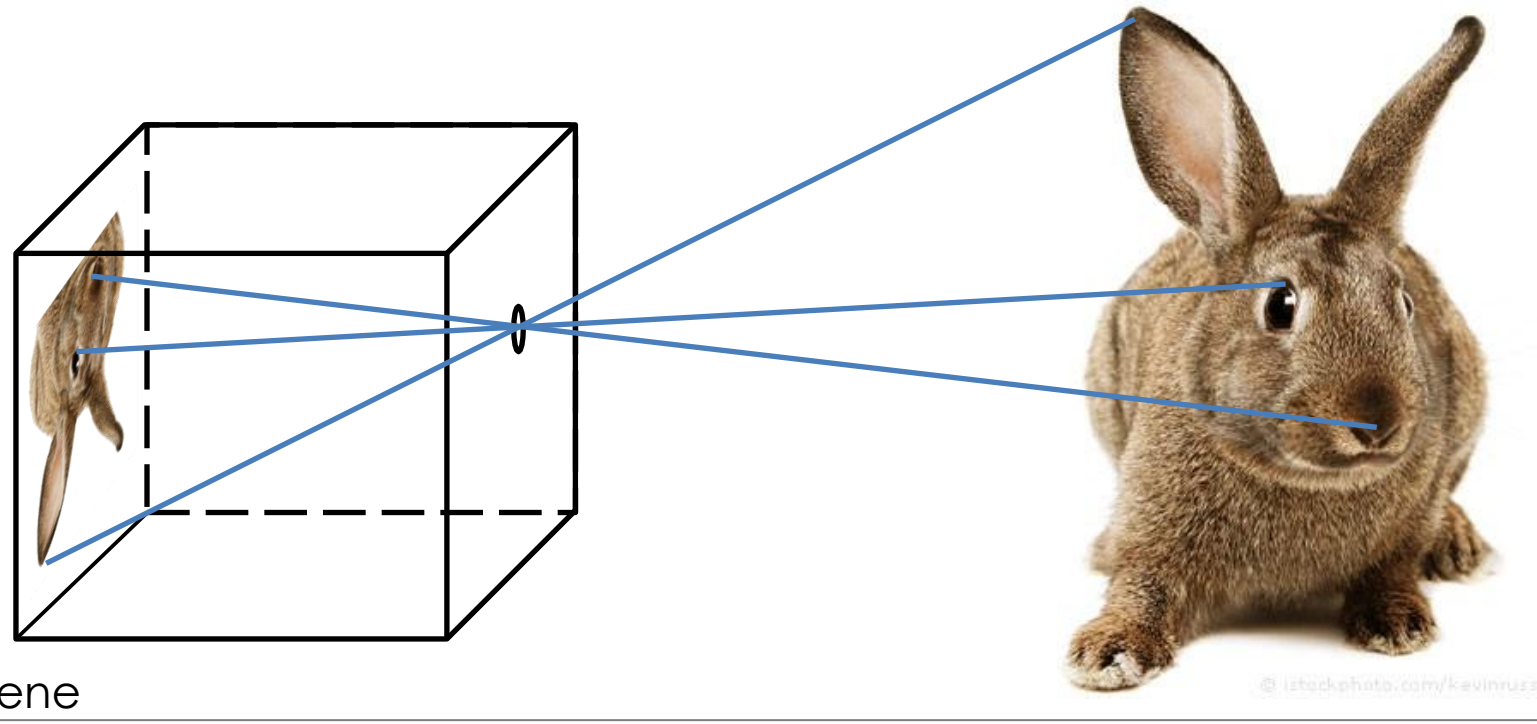
Perspektivische Wahrnehmung III



Bildquelle: https://www.martin-missfeldt.de/perspektive-zeichnen-tutorial/fluchtpunktperspektive.php

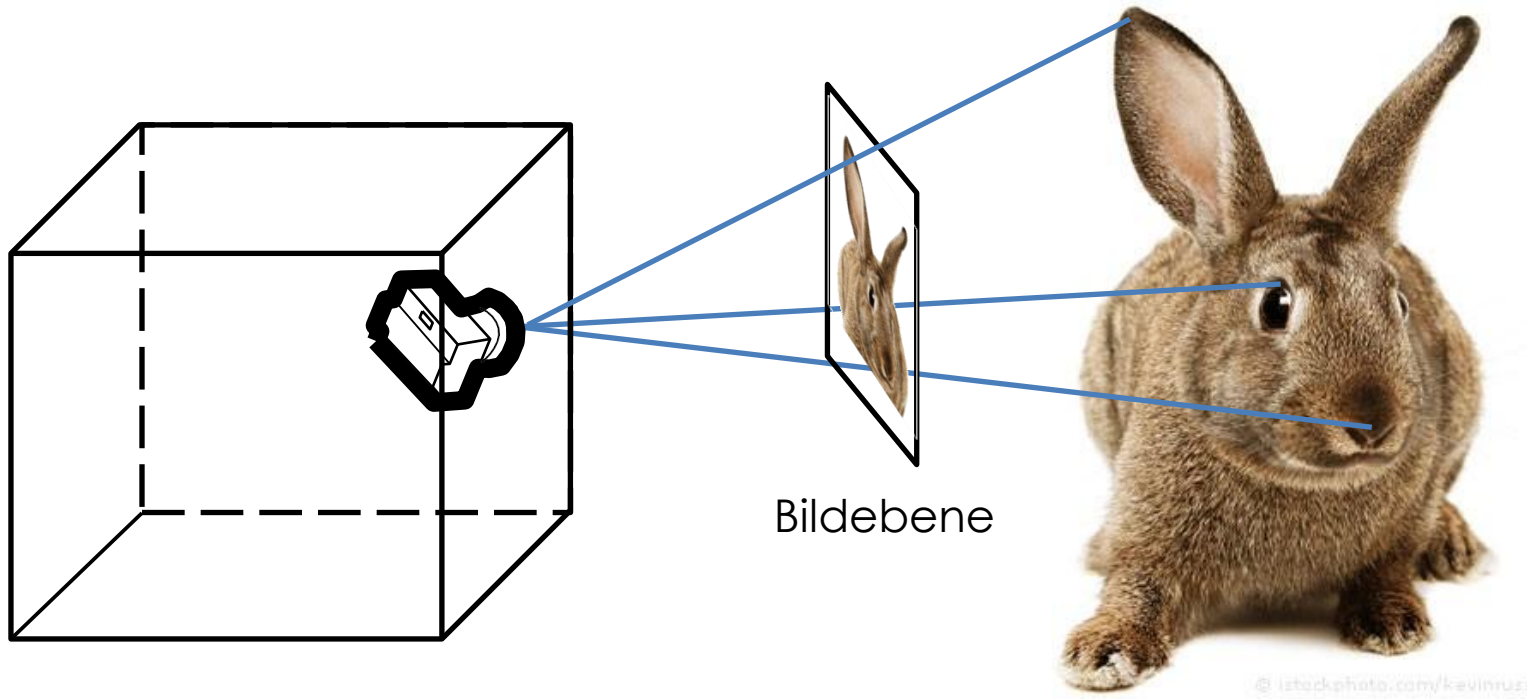
Perspektivische Projektion

- Lochkamera:
  - Schachtel mit Loch
  - Perfektes Bild für "punktgroßes" Loch



Perspektivische Projektion II

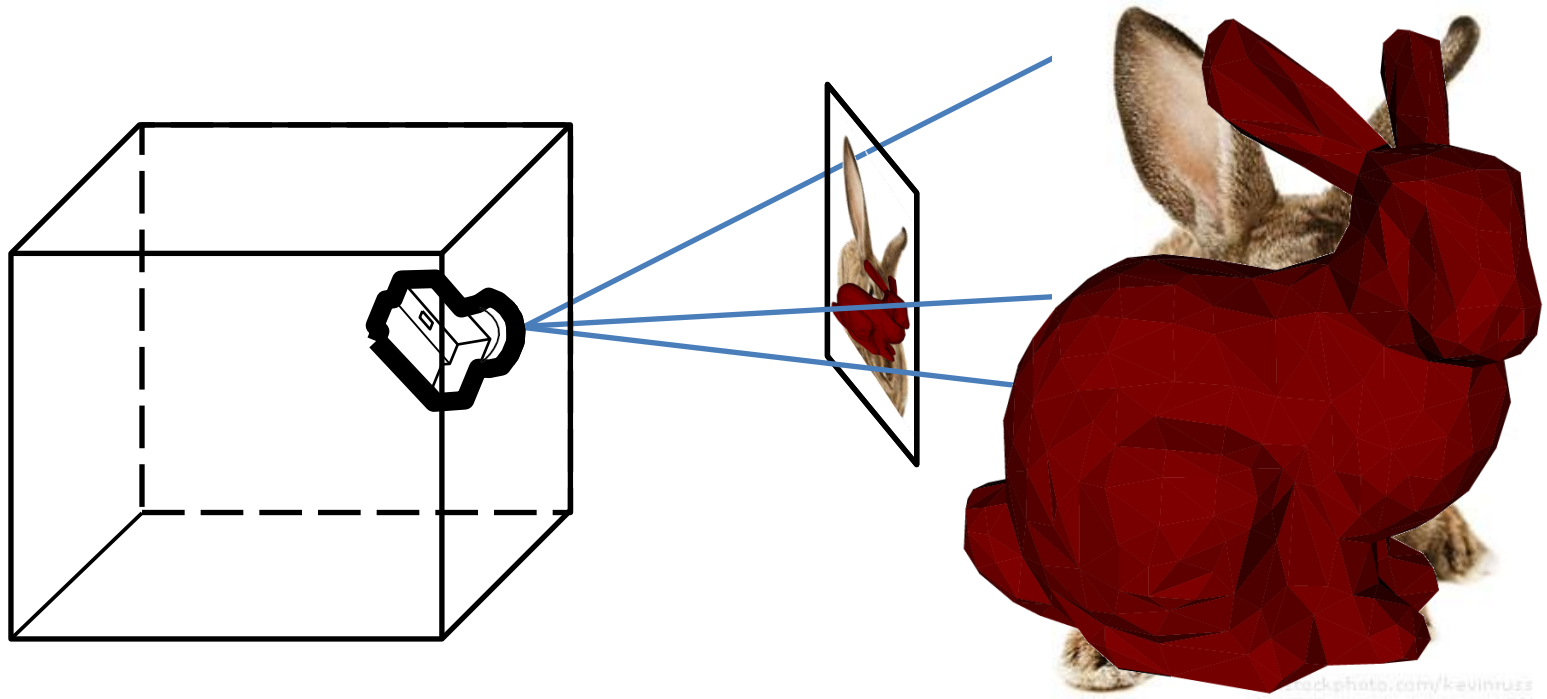
- Lochkamera:
  - Schachtel mit Loch
  - Perfektes Bild für "punktgroßes" Loch



Perspektivische Projektion III

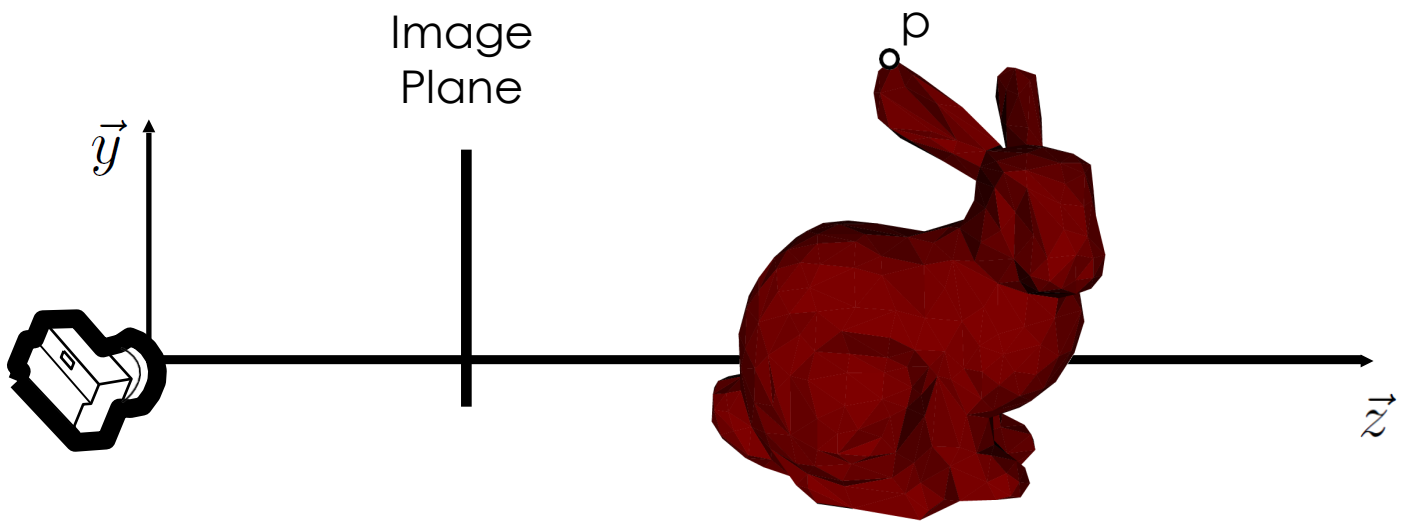
- Lochkamera:
  - Schachtel mit Loch
  - Perfektes Bild für "punktgroßes" Loch

Stanford bunny - Wikipedia



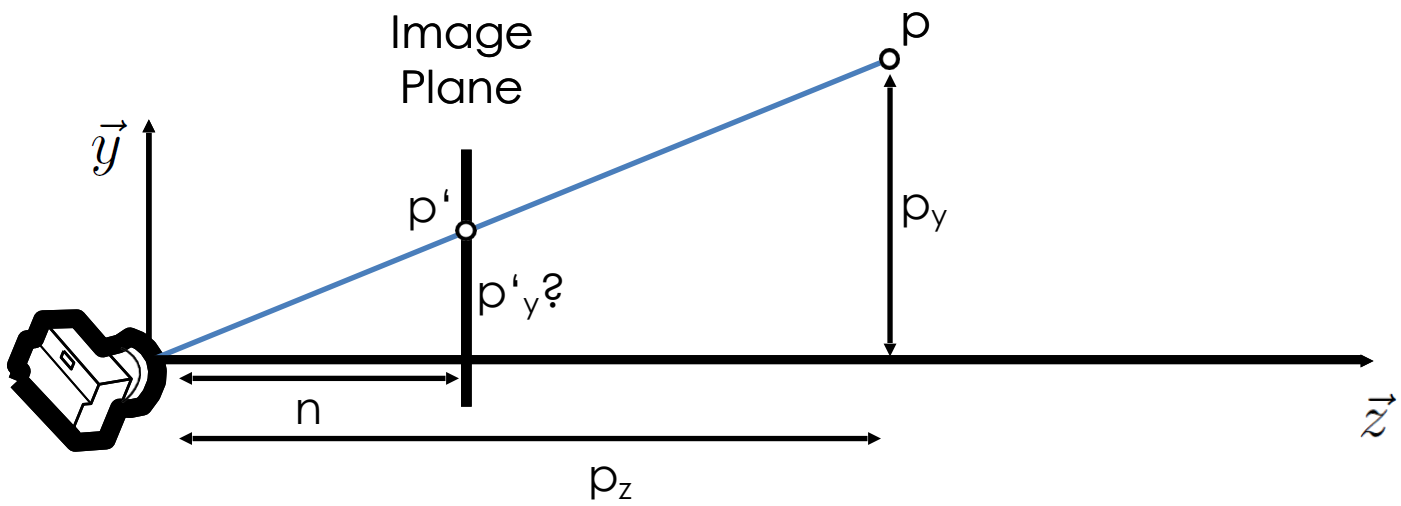
Perspektivische Projektion IV

- Projektion auf die Bildebene:
  - (Hier beispielhaft in 2D)



Perspektivische Projektion V

- Projektion auf die Bildebene:
  - (Hier beispielhaft in 2D)

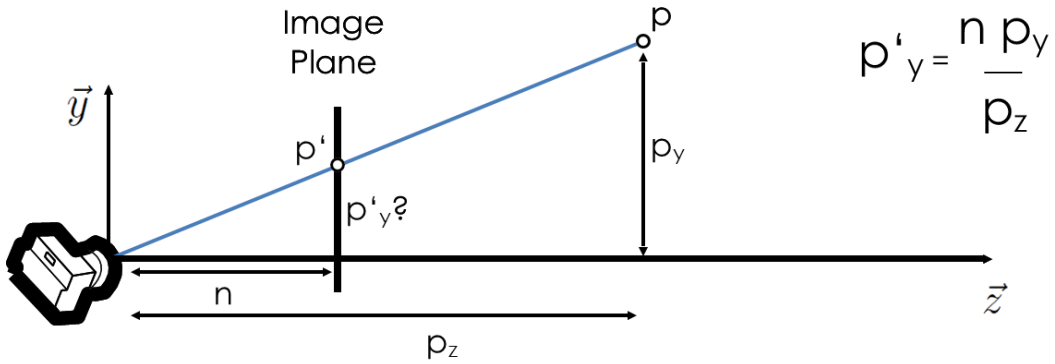


- Strahlensatz:

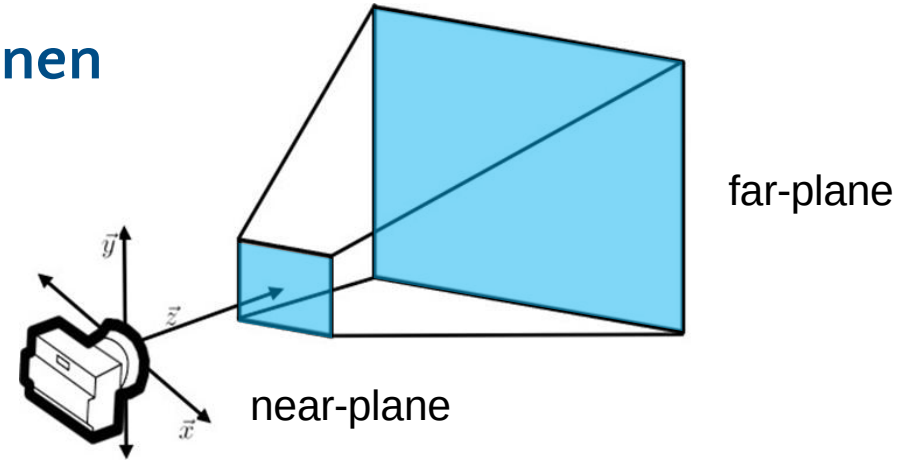
$$\frac{p'_y}{n} = \frac{p_y}{p_z} \quad \Rightarrow \quad p'_y = \frac{n\, p_y}{p_z}$$

Perspektivische Projektion VI

- Wie lässt sich diese Projektion als Matrixmultiplikation realisieren?

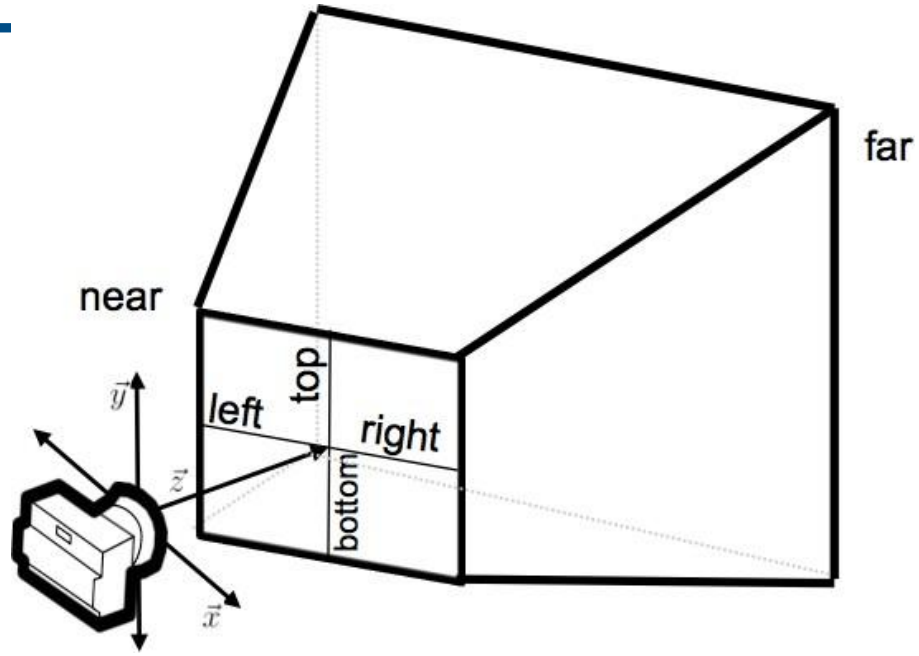


- Lösung: **Nah- und Fern-Clipping Ebenen**



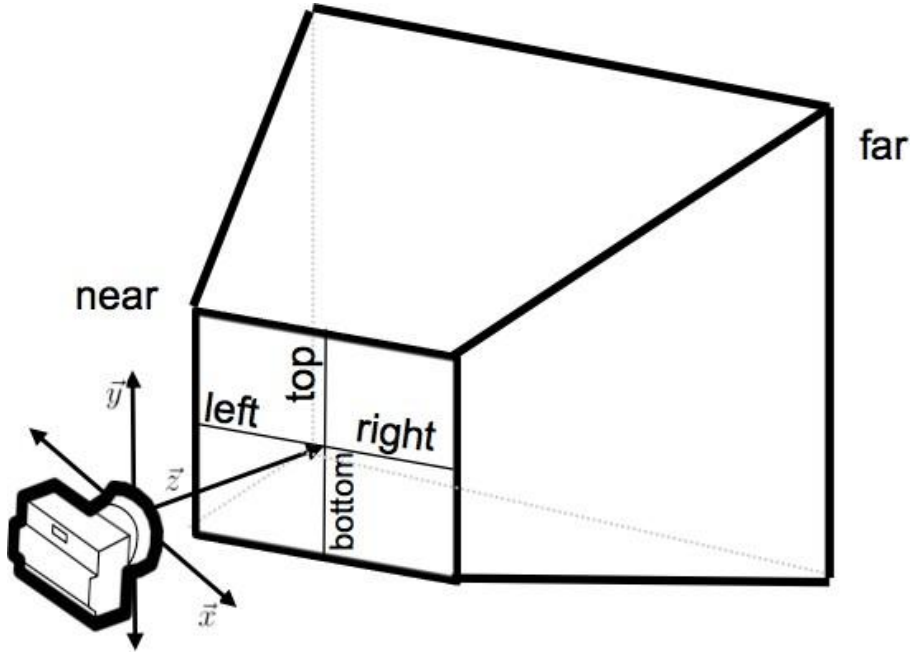
View-Frustum

Definition des View Frustums



- `Near, far` in Bezug auf die Kamera
- `Top/bottom/left/right` ist der Abstand zum Mittelpunkt auf der nahen Ebene
- `glm::frustum(float left, float right, float top, float bottom, float near, float far);`
- `Der Öffnungswinkel ergibt sich aus dem Abstand der near-Plane zur Kamera`

Definition des View Frustums



theta: Vertikaler Öffnungswinkel
aspect: Seitenverhältnis Höhe : Breite

Alternative:

- `glm::perspective(float theta, float aspect, float near, float far);`
- Einschränkung: Nur symmetrisches Frustum

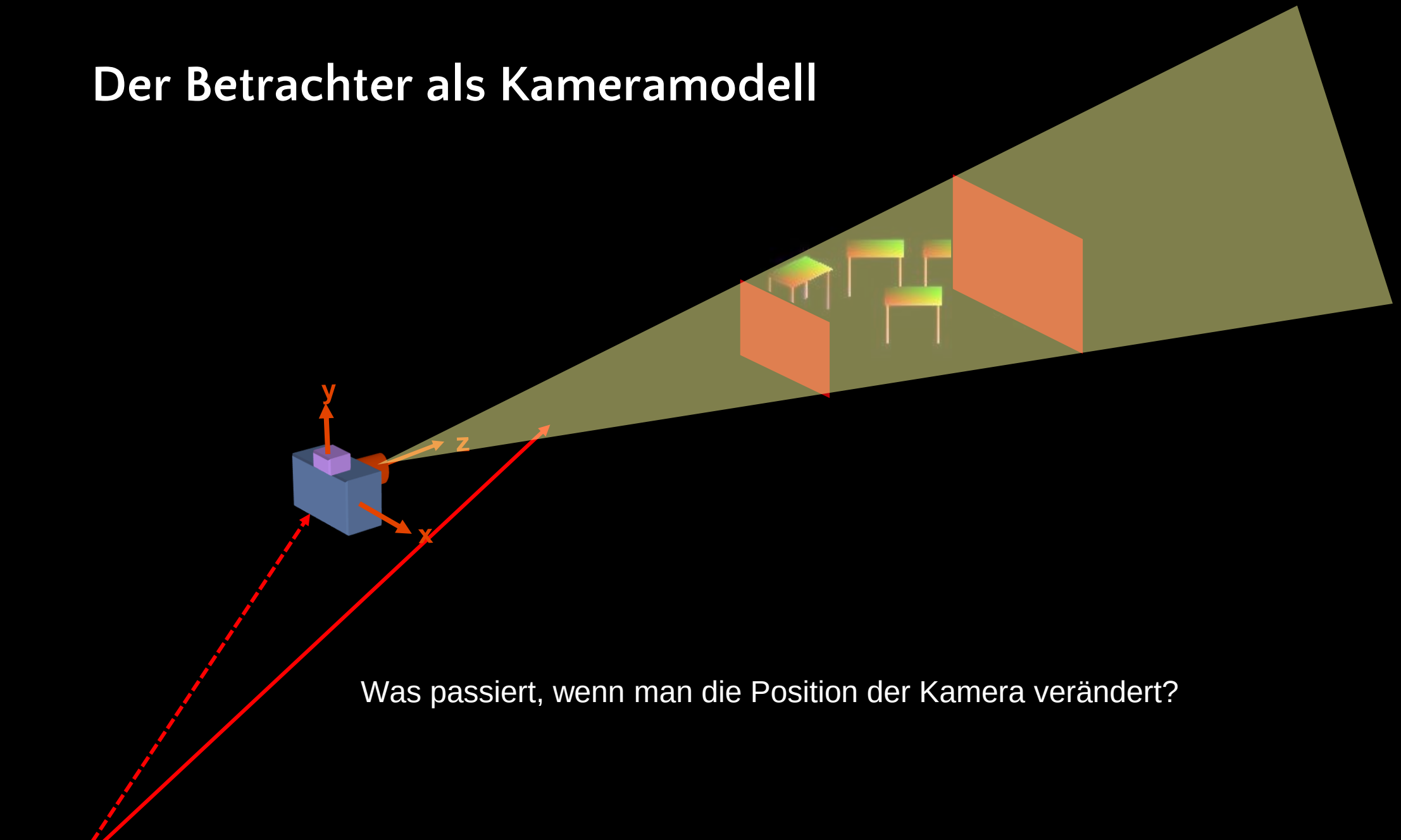
Nicht klausurrelevant!

Perspektivische Projektionsmatrix

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ w \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{n+f}{n} & -f \\ 0 & 0 & 1/n & 0 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

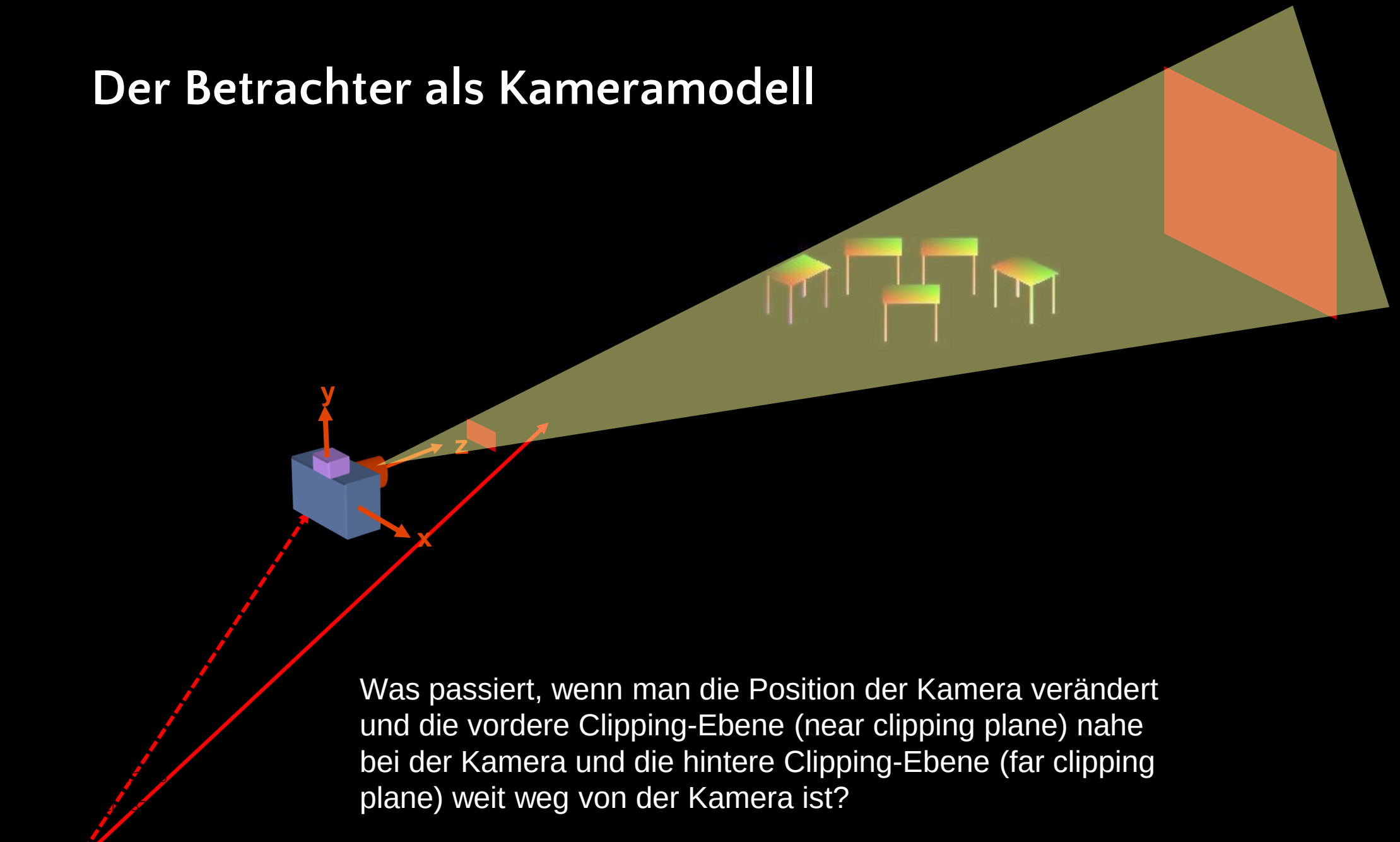
$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ w \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \cdot \frac{n+f}{n} - f \\ p_z/n \end{pmatrix} = \begin{pmatrix} np_x/p_z \\ np_y/p_z \\ n + f - nf/p_z \\ 1 \end{pmatrix}$$

Der Betrachter als Kameramodell



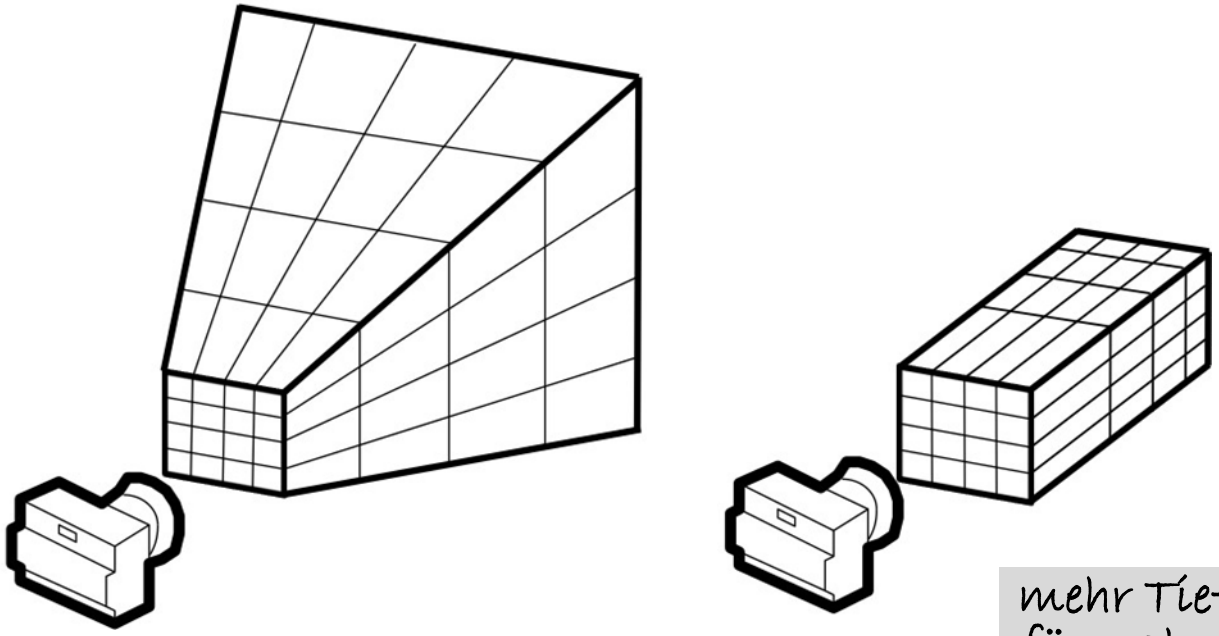
Was passiert, wenn man die Position der Kamera verändert?

Der Betrachter als Kameramodell



Was passiert, wenn man die Position der Kamera verändert und die vordere Clipping-Ebene (near clipping plane) nahe bei der Kamera und die hintere Clipping-Ebene (far clipping plane) weit weg von der Kamera ist?

Ergebnis der Projektionsmatrix



mehr Tiefenpräzision für nahe Objekte

3D-Transformation:

- Punkte auf der Nahbereichsebene werden nicht verändert
- x,y-Koordinaten werden perspektivisch skaliert
- Nichtlineare Skalierung der z-Koordinaten

Vertex Transformationen bis jetzt

Model Transformation M_{Model}: Akkumulierte Transformationen, platziert Objekte in der Szene (Welt)

View Transformation V: Rotiert und translatiert die gesamte Szene, sodass die Kamera im Ursprung liegt und entlang der negative Z-Achse schaut.

Projection Transformation M_{proj}: Berechnet Projektion, bildet das Kamera-Frustum auf den Würfel $[-w, w]^3$ ab; Blickrichtung entlang der z-Achse

$$\mathbf{p}' = \underbrace{M_{Projection}}_{M_{ortho}\ oder\ M_{pers}} \cdot V \cdot \underbrace{T \cdot \ldots \cdot T \cdot S \cdot R}_{M_{Model}} \cdot \mathbf{p}$$

Negation der z-Koordinate

Eine Kleinigkeit fehlt noch:

- Designer mögen es nicht, in einem linkshändigen Koordinatensystem zu arbeiten, aber der **Normalized Device Space benutzt ein linkshändiges Koordinatensystem** (um positive z-Werte für die Entfernung zu haben)
  - Zur Erinnerung: Im Camera-Space soll unsere Kamera entlang der negativen Z-Achse schauen

- Wie kommen wir von einem linken Koordinatensystem zu einem rechten Koordinatensystem? Und andersherum?

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{M_{R \to L}} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

Vertex Transformationen bis jetzt

Model Transformation M_{Model}: Akkumulierte Transformationen, platziert Objekte in der Szene (Welt)

View Transformation V: Rotiert und transliert die gesamte Szene, sodass die Kamera im Ursprung liegt und entlang der negative Z-Achse schaut.

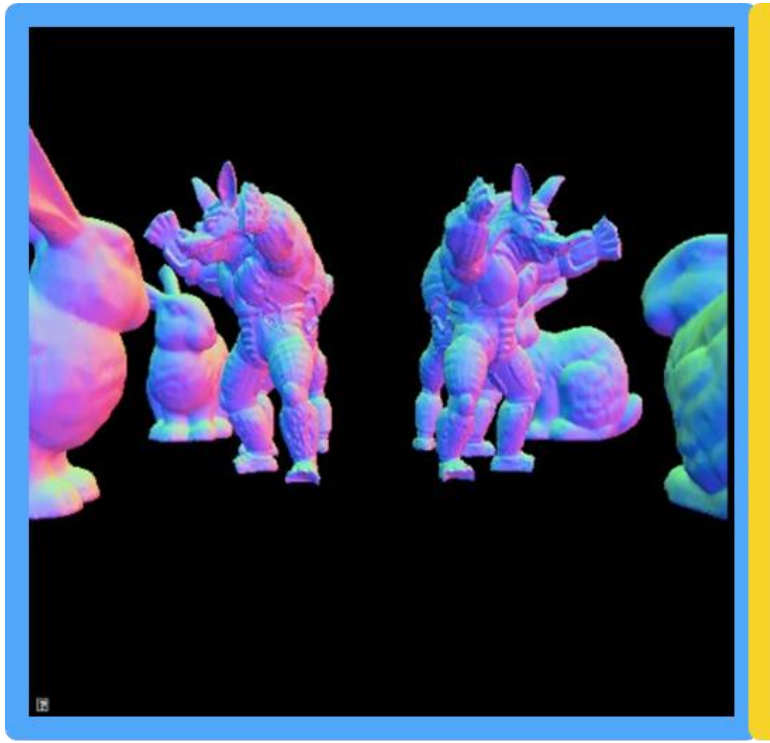
Projection Transformation M_{proj}: Berechnet Projektion, bildet das Kamera-Frustum auf den Würfel $[-w, w]^3$ ab; Blickrichtung entlang der z-Achse

Negation der Z-Achse: Multiplikation der Z-Werte mit -1

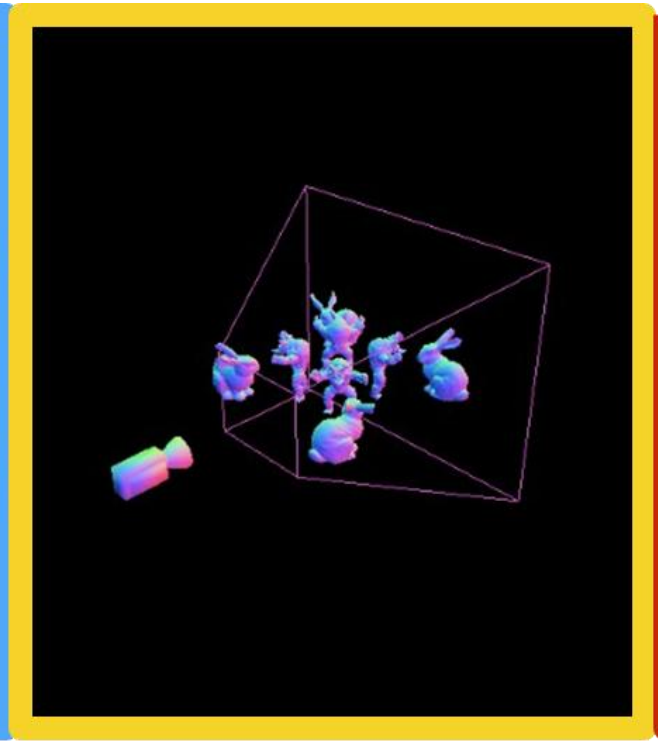
Invertierung der Z-Achse bereits integriert in den gl-Methoden

$$\mathbf{p}' = \underbrace{M_{Ortho} \cdot M_{Persp} \cdot M_{R \to L}}_{M_{ortho} \text{ oder } M_{pers}} \cdot V \cdot \underbrace{T \cdot \ldots \cdot T \cdot S \cdot R}_{M_{Model}} \cdot \mathbf{p}$$

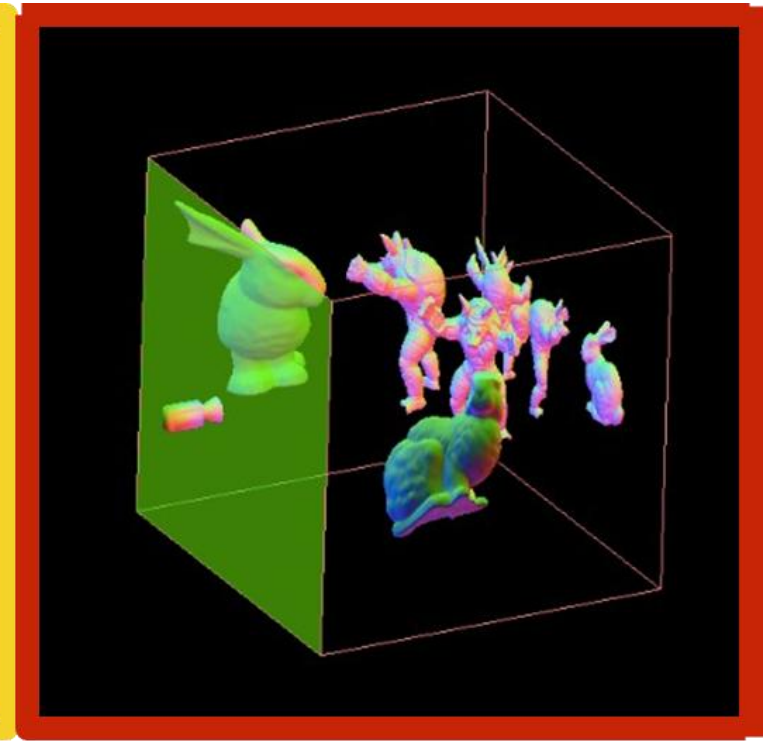
Beispielprojektion



Camera View



Camera Space



Normalized Device Space

Implementierung der Vertex-Transformationen im Shader

Vertex Shader

```
#version 330
layout(location = 0) in vec3 vertex;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
	gl_Position = projection * view *  model * vec4(vertex, 1.0);
}
```

- Benötigt nur 4 Zeilen Code im Vertexshader
- Output ist die Vertex-Position in homogenen Koordinaten, vor der **perspektivischen Division** (Homogenisierung)