

Computergrafik

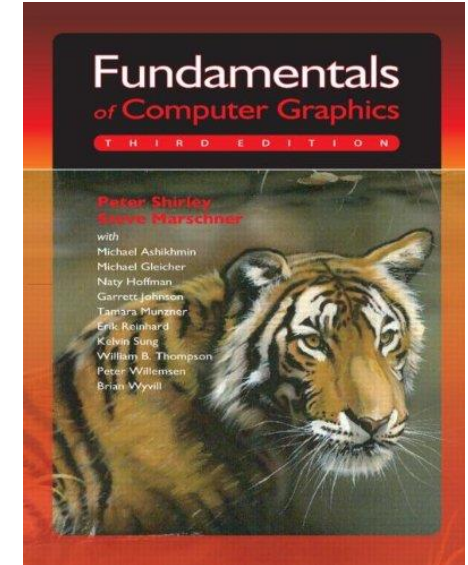
Vorlesung im Wintersemester 2020/21

Kapitel 6: Rasterisierung, Clipping und Projektionstransformationen

Prof. Dr.-Ing. Carsten Dachsbacher
Lehrstuhl für Computergrafik
Karlsruher Institut für Technologie



- ▶ **Fundamentals of Computer Graphics,**
P. Shirley, S. Marschner, 3rd Edition, AK Peters
→ Kapitel 7 und 8



▶ Bildsynthese (engl. Rendering)

- ▶ erzeugt ein Rasterbild aus einer Szenenbeschreibung (Objekte)
- ▶ also: bestimme, welche Objekte die Farbe jedes Pixels beeinflussen

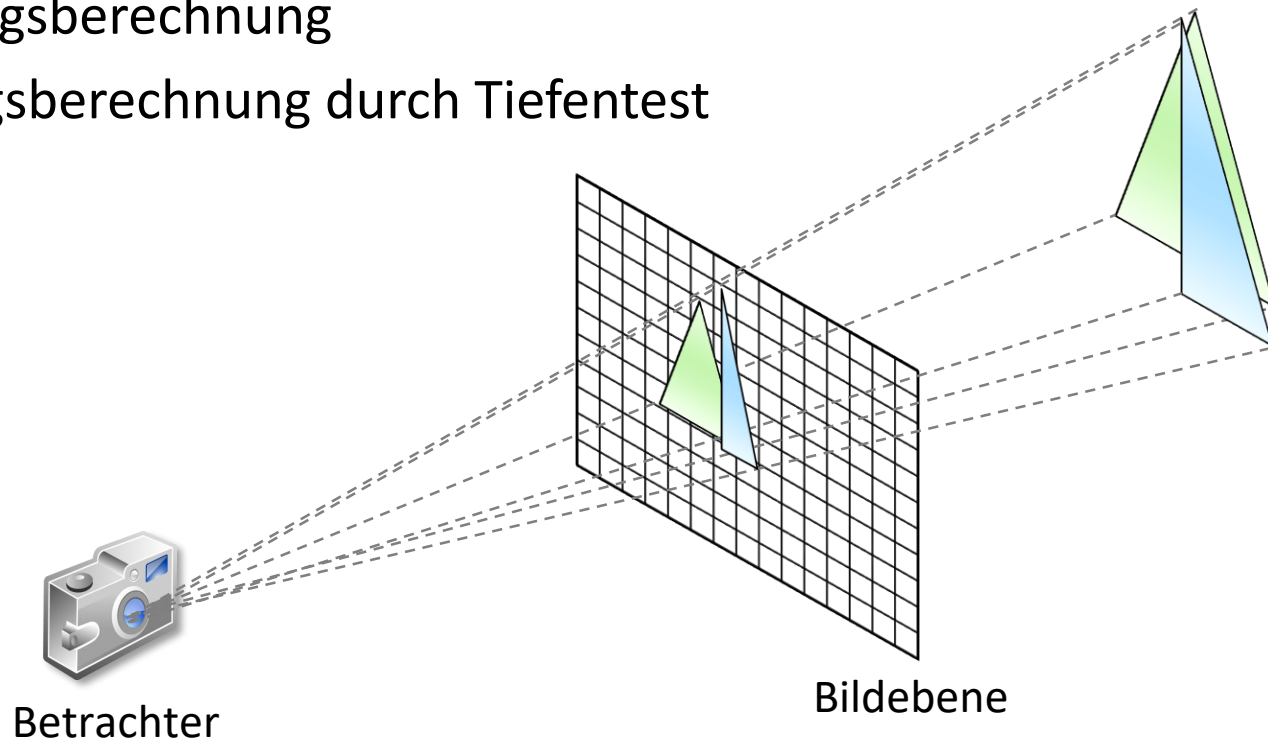
▶ **Bildbasiert (Image-Order Rendering) → Raytracing**

- ▶ betrachte einen Pixel nach dem anderen
- ▶ finde heraus, welches Primitiv an dieser Stelle sichtbar ist
- ▶ bestimme die Pixelfarbe

▶ **Objektbasiert (Object-Order Rendering) → Rasterisierung**

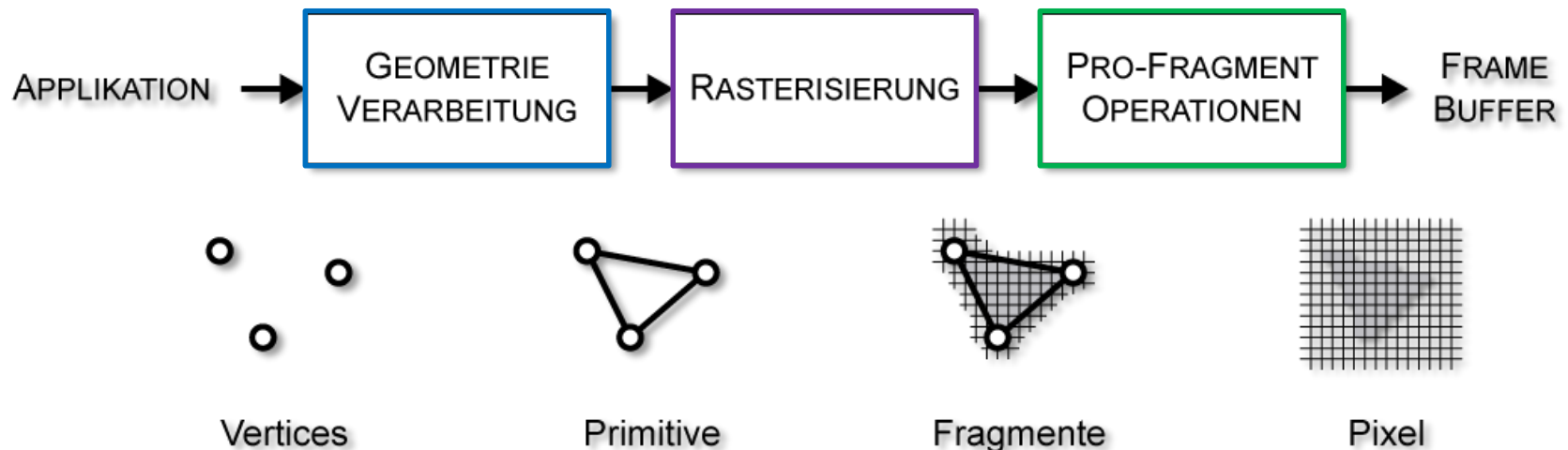
- ▶ betrachte ein Objekt/ein Primitiv/eine Fläche nach der anderen
- ▶ finde heraus, welche Pixel das Primitiv bedeckt
- ▶ bestimme die Pixelfarbe

- ▶ Rasterisierung ist Grundlage fast aller interaktiven CG-Anwendungen
 - ▶ Repräsentation von Oberflächen meist durch Dreiecke
 - ▶ Abbilden von Dreiecken auf 2D Bildschirmkoordinaten + Clipping
 - ▶ Rasterisierung der Dreiecke
 - ▶ Interpolation von Farben, Texturkoordinaten, Tiefenwert, etc. für jeden Pixel (an den Eckpunkten gegeben)
 - ▶ Schattierungsberechnung
 - ▶ Verdeckungsberechnung durch Tiefentest



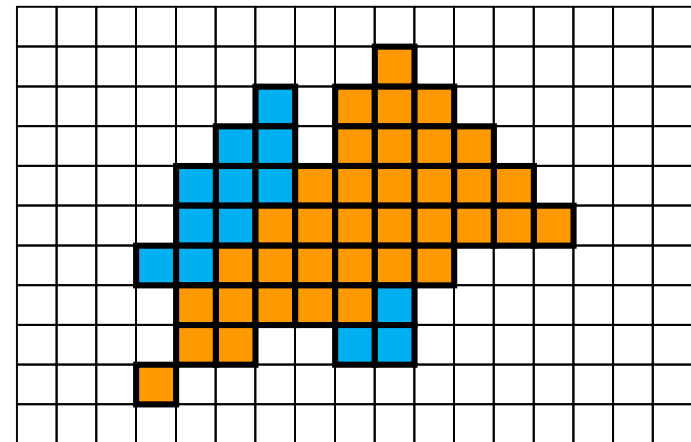
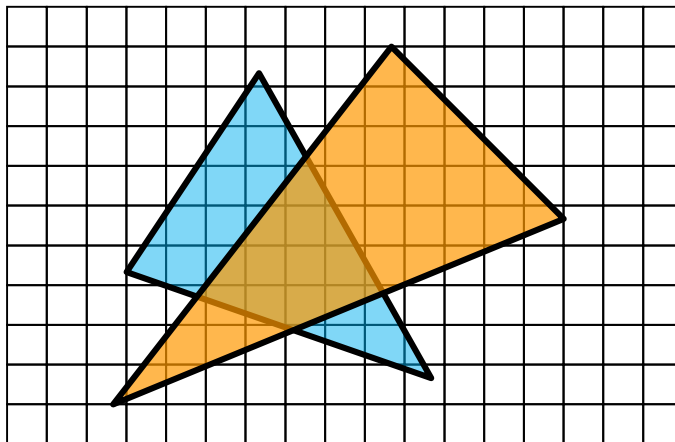
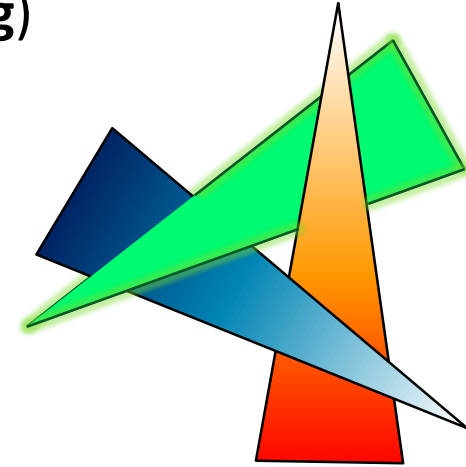
- ▶ Rasterisierung ist Grundlage fast aller interaktiven CG-Anwendungen
 - ▶ Repräsentation von Oberflächen meist durch Dreiecke
 - ▶ Abbilden von Dreiecken auf 2D Bildschirmkoordinaten + Clipping
 - ▶ Rasterisierung der Dreiecke
 - ▶ Interpolation von Farben, Texturkoordinaten, Tiefenwert, etc. für jeden Pixel (an den Eckpunkten gegeben)
 - ▶ Schattierungsberechnung
 - ▶ Verdeckungsberechnung durch Tiefentest

Grafik-Pipeline



Grundlagen für Rasterisierungsverfahren (Basis von OpenGL, Direct3D, ...)

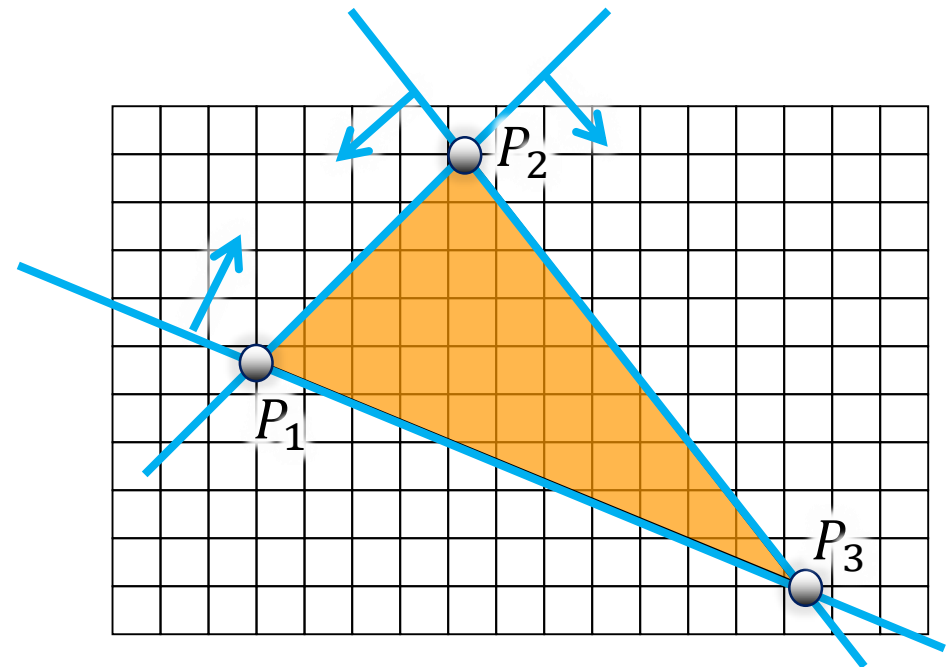
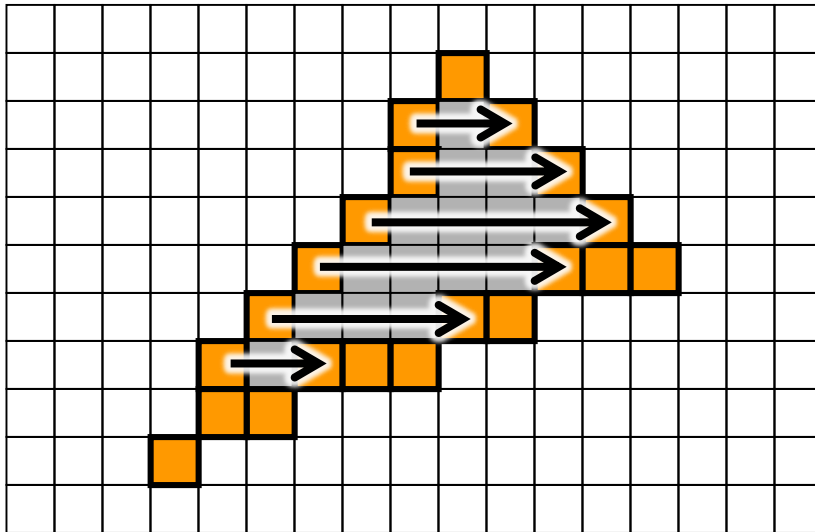
- ▶ Grundlegende Rasterisierungsansätze (in der **Übung**)
- ▶ Clipping von Linien und Polygonen (in der **Übung**)
 - ▶ Cohen-Sutherland und α -Clipping
 - ▶ Sutherland-Hodgeman
- ▶ Sichtbarkeit und Tiefenpuffer (Z-Buffer)
- ▶ Projektive Abbildungen
 - ▶ Clipping in homogenen Koordinaten
 - ▶ perspektivische Interpolation



Rasterisierung von konvexen Polygonen/Dreiecken



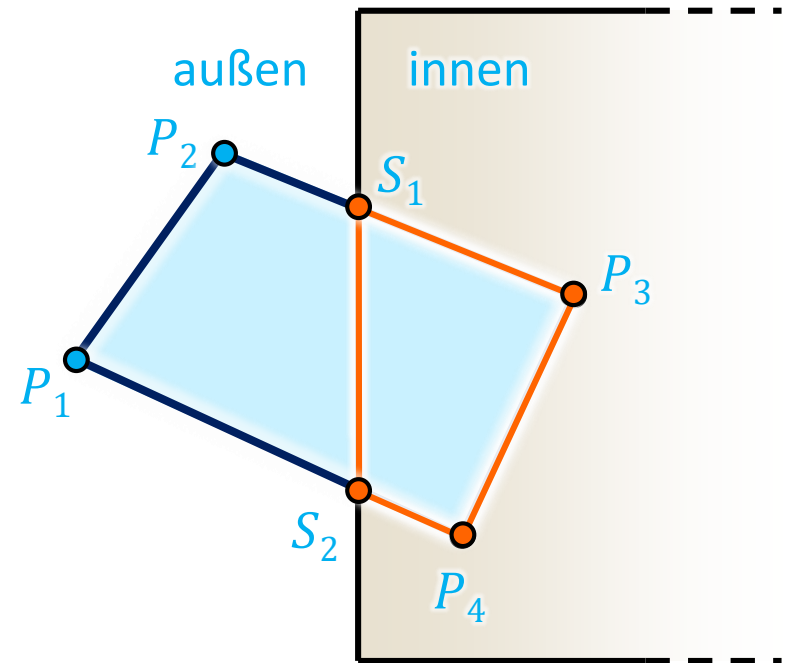
- ▶ Scanline-Verfahren (links)
 - ▶ interpoliere links und rechte Kante(n)
 - ▶ und dazwischen zeilenweise
- ▶ Edge-based/Kanten-basiertes Verfahren (rechts)
 - ▶ Dreieckskanten aufstellen
 - ▶ Pixel im Dreieck liegen in der positiven Halbebene aller Kanten



- ▶ Clipping: Abschneiden von Linien/Polygon-Teilen die außerhalb des Bildschirms liegen (oder allg. außerhalb eines gewünschten Bereichs)
- ▶ Clipping kann wichtig sein für die Effizienz: rasterisiere nichts außerhalb des sichtbaren Bereichs, keine Objekte hinter der Kamera (in 3D)
- ▶ Clipping ist unbedingt notwendig zur **Vermeidung problematischer Fälle bei Projektionen** (gleich mehr)

Beispiel: Clipping gegen eine Kante

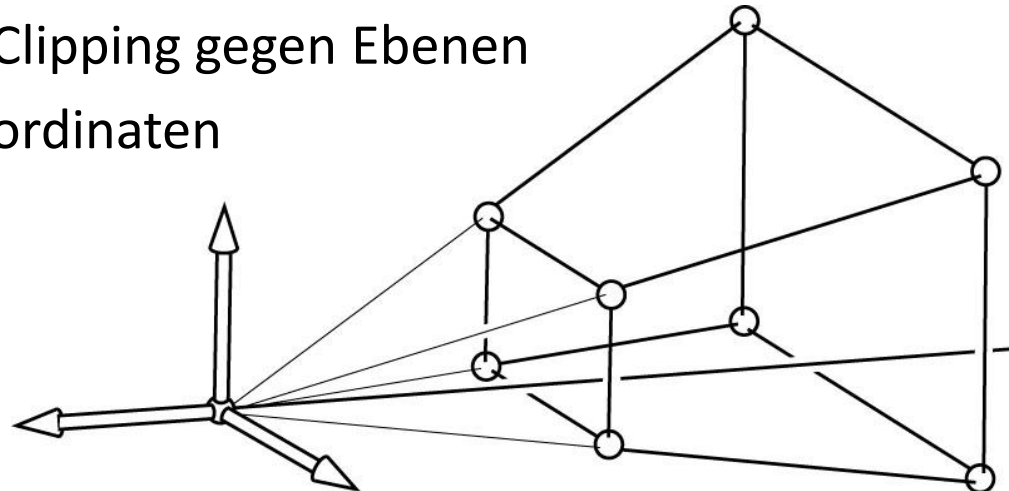
- ▶ Eingabe: Liste der Vertices des Polygons
- ▶ Ausgabe: Liste von Vertices des resultierenden Polygons
- ▶ hier:
 - ▶ Eingabe: P_1, P_2, P_3, P_4
 - ▶ Ausgabe: S_1, P_3, P_4, S_2



- ▶ Clipping: Abschneiden von Linien/Polygon-Teilen die außerhalb des Bildschirms liegen (oder allg. außerhalb eines gewünschten Bereichs)
- ▶ Clipping kann wichtig sein für die Effizienz: rasterisiere nichts außerhalb des sichtbaren Bereichs, keine Objekte hinter der Kamera (in 3D)
- ▶ Clipping ist unbedingt notwendig zur **Vermeidung problematischer Fälle bei Projektionen** (gleich mehr)

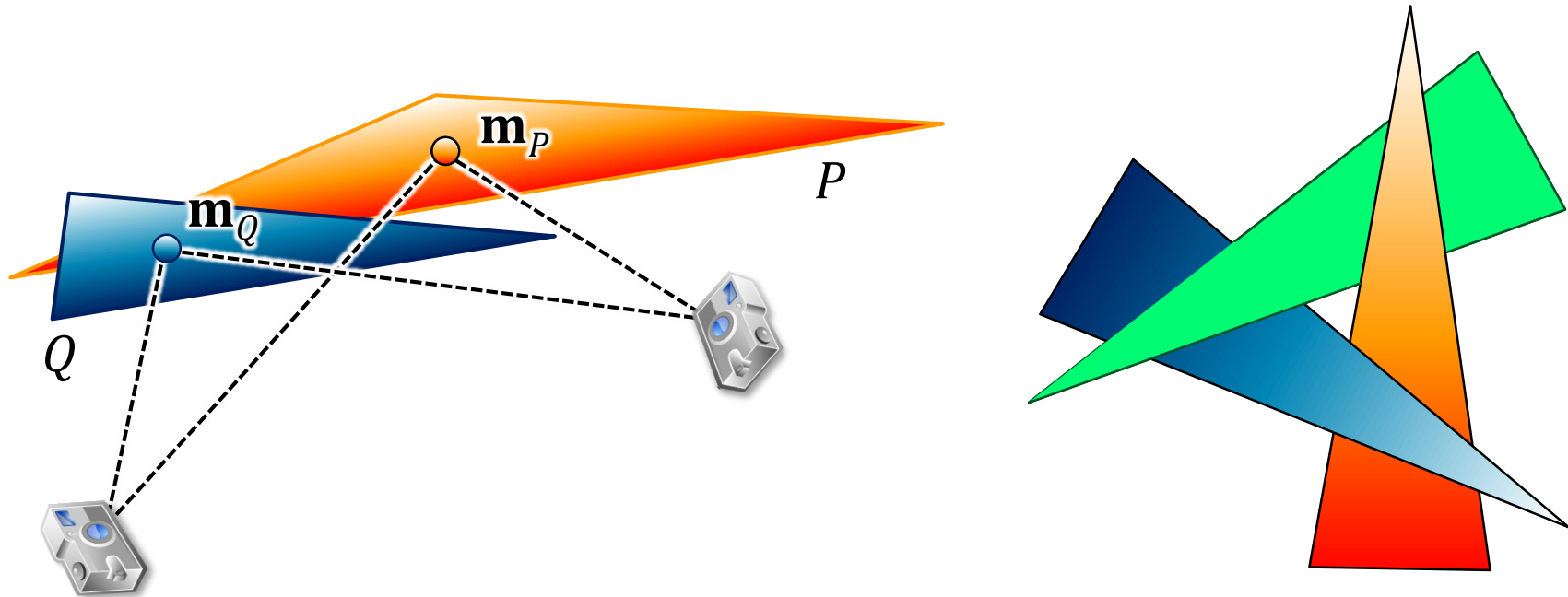
Clipping in 3D

- ▶ Clipping gegen ein konvexes Sichtvolumen
 - ▶ statt Clipping gegen Kanten: Clipping gegen Ebenen
 - ▶ α -Clipping in homogenen Koordinaten



Rasterisierung und Sichtbarkeit

- ▶ Maler-Algorithmus – Sortieren der Polygone – funktioniert nicht immer
- ▶ außerdem benötigt die Sortierung Zugriff auf alle Polygone

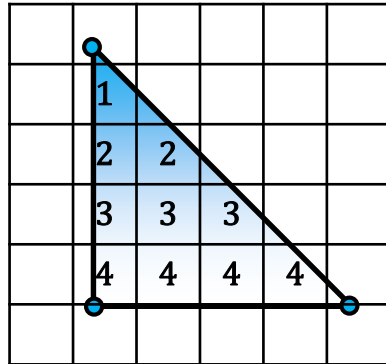


- ▶ wichtig für effiziente Verarbeitung (v.a. in Hardware): Pipelining
 - ▶ Bearbeitung eines Dreiecks nach dem anderen
 - ▶ unabhängig von allen anderen Dreiecken

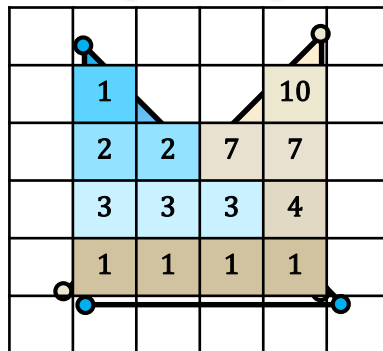
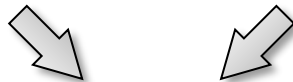
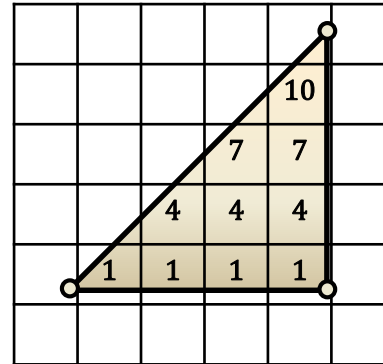
Idee, Prinzip

- ▶ bildbasierter Ansatz: speichere für jeden Pixel Distanz zur nächsten Fläche
- ▶ Entfernung-/Tiefen-Wert wird pro Vertex berechnet und interpoliert
- ▶ zusätzlich zum Farbpuffer gibt es dazu den Z-Buffer (16 bis 32 Bit/Pixel)

Tiefenwerte Dreieck 1

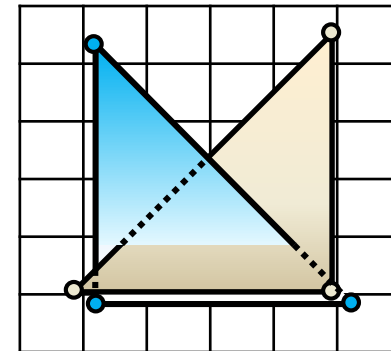


Tiefenwerte Dreieck 2



Sichtbarkeit mit Tiefenpuffer

zum Vergleich:
Sichtbarkeit exakt



Tiefenpuffer, Z-Buffering



Rasterisierung mit Tiefenpuffer

► Initialisierung

- ▶ fülle Farbpuffer/Framebuffer mit Hintergrundfarbe
- ▶ fülle Tiefenpuffer mit maximalem Tiefenwert („Far-Plane“, gleich mehr)

► Rasterisierung

```
// in beliebiger Reihenfolge
foreach Dreieck {
    foreach Pixel(x,y) im projizierten Dreieck {
        // Tiefe durch Interpolation
        z = z(x,y)

        // Tiefentest
        if ( z < z_buffer[x,y] ) {
            z_buffer[x,y] = z;
            framebuffer[x,y] = color(x,y);
        }
    }
}
```



Tiefenpuffer als
Graustufenbild dargestellt

Anm. wir können auch $1/z$ im Tiefenpuffer speichern und den Test umdrehen

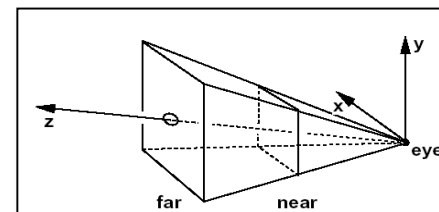
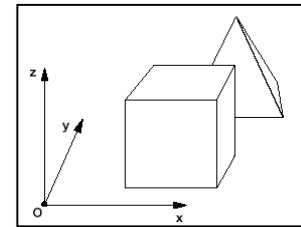
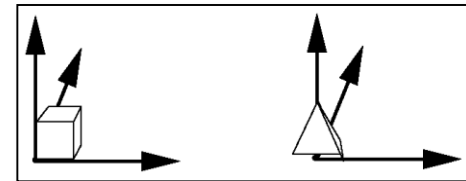
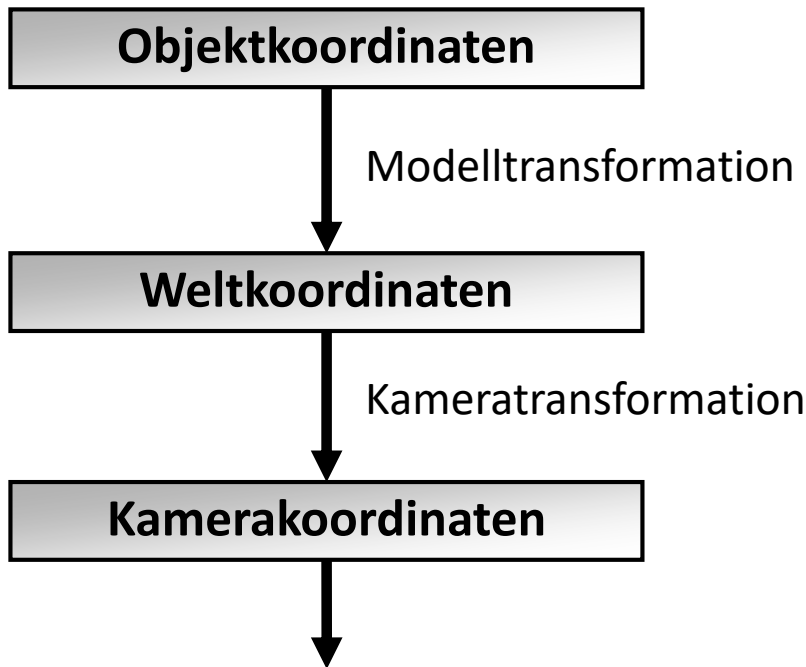
Nachteile (oder besser: was man einfach bedenken sollte)

- ▶ zusätzlicher Speicherbedarf und -bandbreite (heute nicht mehr kritisch)
- ▶ begrenzte Genauigkeit und z-Aliasing (wichtig!)
- ▶ transparente Flächen können nicht ohne Weiteres behandelt werden
- ▶ unnötiger Aufwand in Szenen mit hoher Tiefenkomplexität
 - ▶ Tiefenkomplexität = Anzahl der Schnitte entlang eines Primärstrahls mit den Oberflächen der Szene
 - ▶ unnötiger Aufwand, weil verdeckte Flächen rasterisiert werden (gilt gleichermaßen für Maler-Algorithmus)

Vorteile

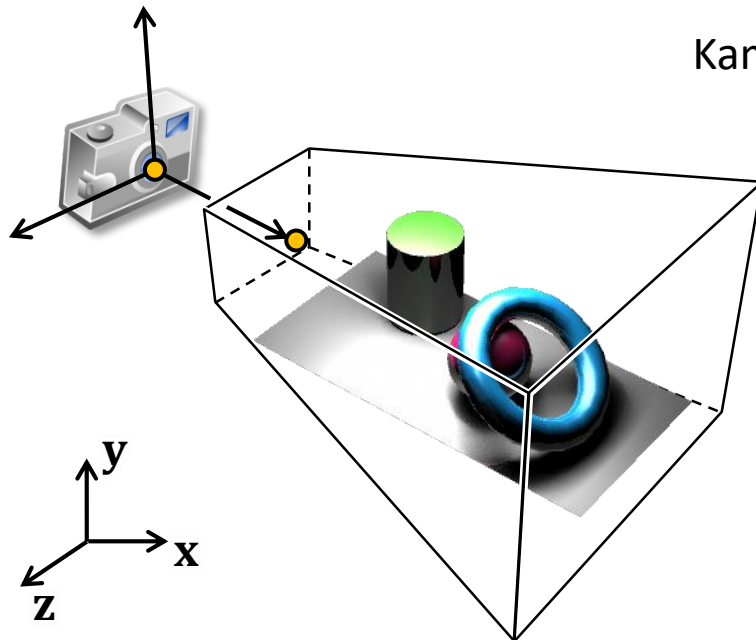
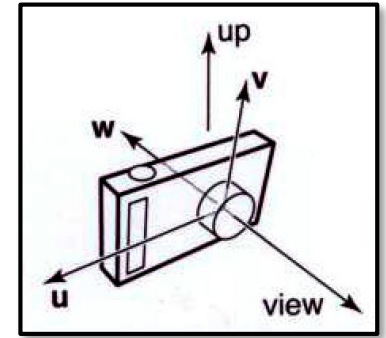
- ▶ Dreiecke können in beliebiger Reihenfolge verarbeitet werden
- ▶ Z-Buffering ist Standard in allen Rasterisierern (inkl. Grafik-Hardware)
- ▶ für die meisten der obigen Probleme existieren heute spezielle Lösungen oder Rendering-Techniken

- ▶ ...jetzt aber zurück zu den Transformationen...
- ▶ Objekte in einer Szene werden zur Modellierung (Beschreibung) in ihrem eigenen **Objekt- oder Modell-Koordinatensystem** angegeben
- ▶ die Platzierung der Objekte im **Weltkoordinatensystem** erfolgt dann durch Translation, Rotation, Skalierung etc.
- ▶ Transformation in das **Kamerakoordinatensystem** ist notwendig für Rasterisierung



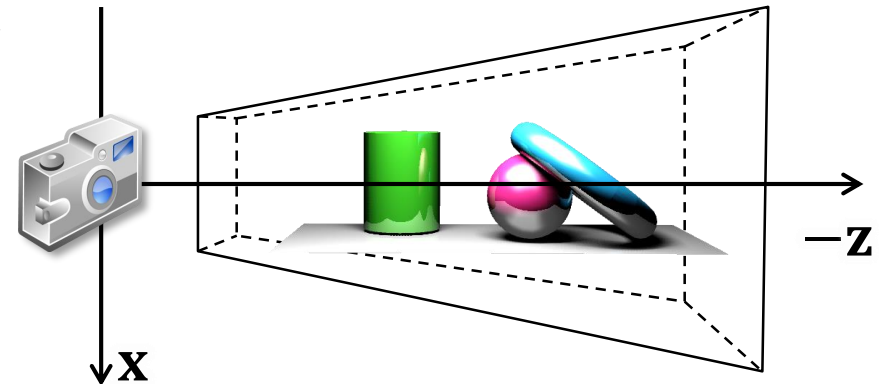
Kameratransformation

- ▶ virtuelle Kamera definiert durch
 - ▶ Position \mathbf{e} und (negative) Blickrichtung \mathbf{w}
 - ▶ „Up-Vektor“ \mathbf{up}
 $\Rightarrow \mathbf{u} = \mathbf{up} \times \mathbf{w}$ und $\mathbf{v} = \mathbf{w} \times \mathbf{u}$
 - ▶ zuerst Translation um $-\mathbf{e}$, dann Transformation in das Kamera-Koordinatensystem dessen Basis von $\mathbf{u}, \mathbf{v}, \mathbf{w}$ gebildet wird (als Zeilenvektoren)



Kamera und Sichtpyramide

Kameratransformation

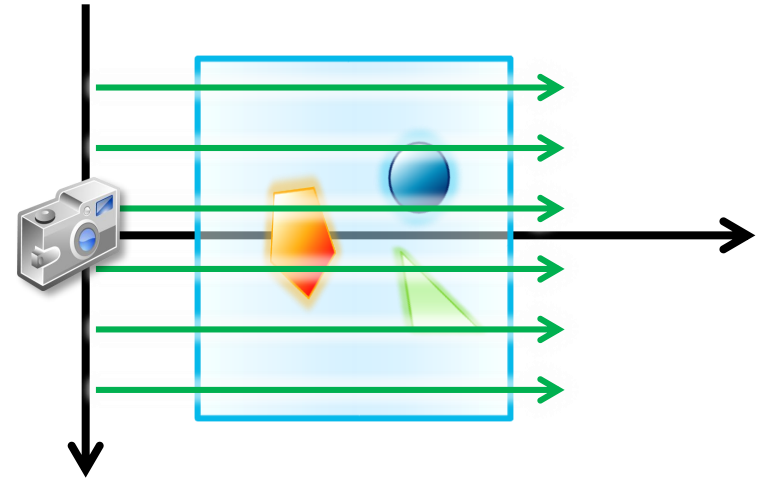


nach der Kameratransformation

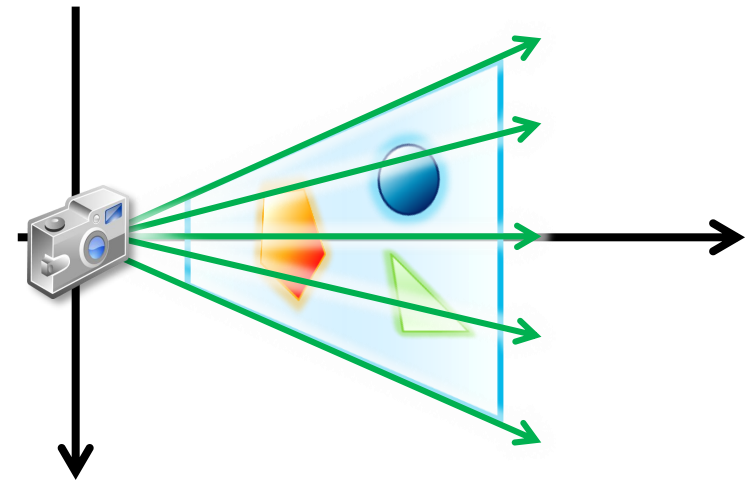
Projektion

- ▶ wir betrachten nur zwei Arten von Projektionen

- ▶ orthographische Kamera:
parallele **Sichtstrahlen**
(Anm. wir betrachten nur senkrechte Projektionen auf die Bildebene, es gibt auch schiefe Projektionen mit parallelen Sichtstrahlen)

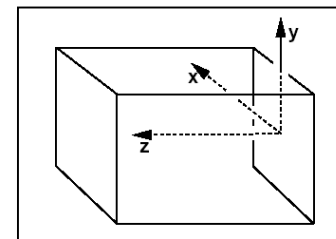
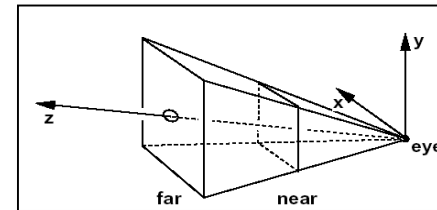
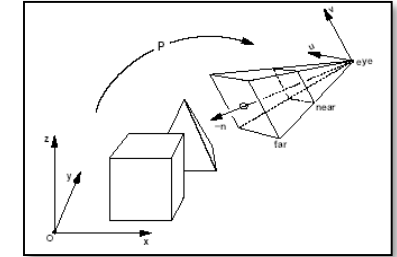
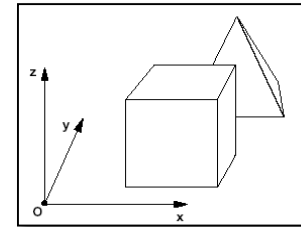
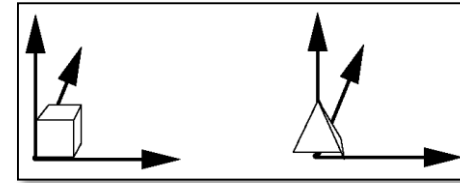
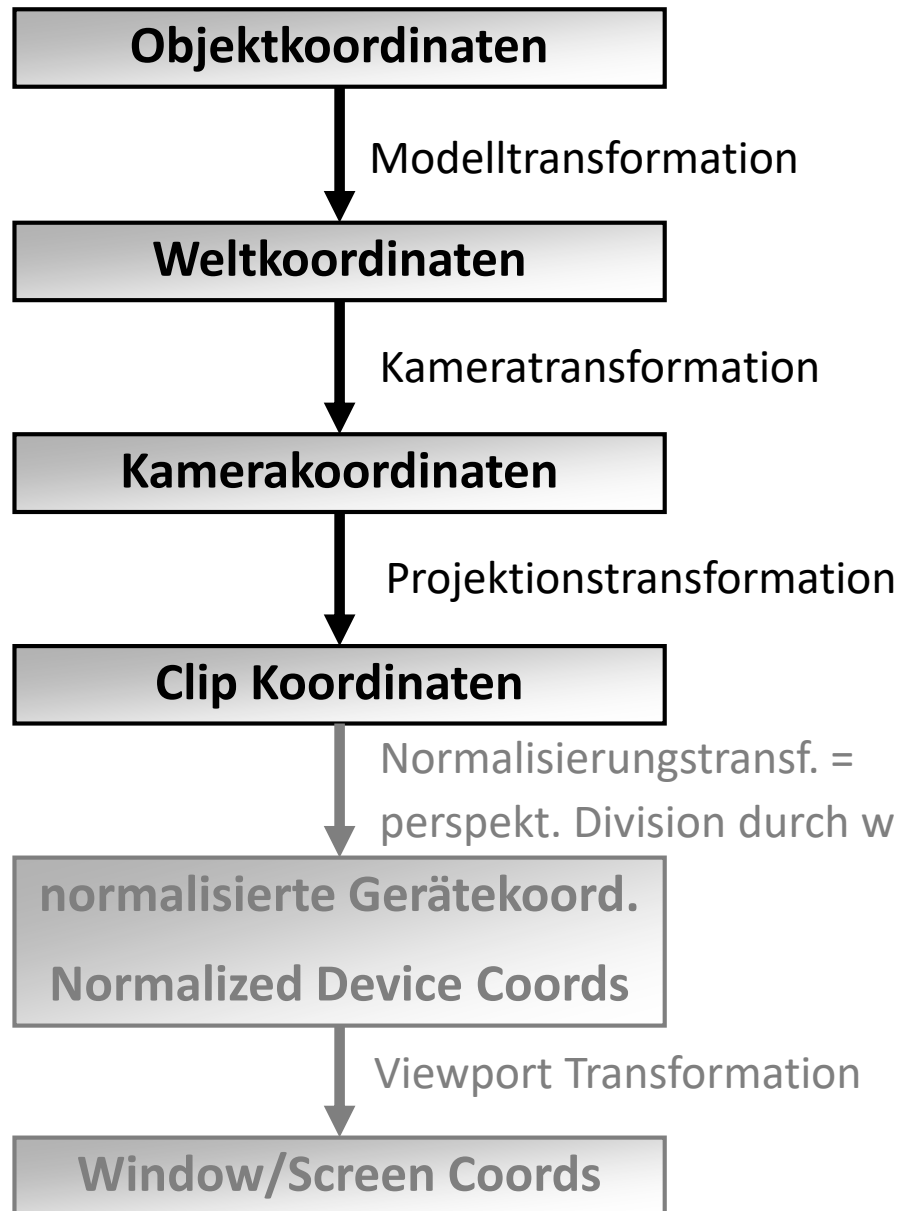


- ▶ perspektivische Kamera:
Sichtstrahlen ausgehend von einem Projektionszentrum

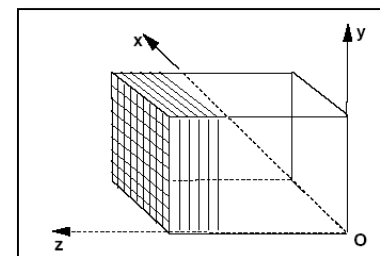


- ▶ es geht um die Frage, wie unterschiedliche Projektionen in der Grafik-Pipeline einheitlich durchgeführt werden

(Noch mehr) Koordinatensysteme in der CG



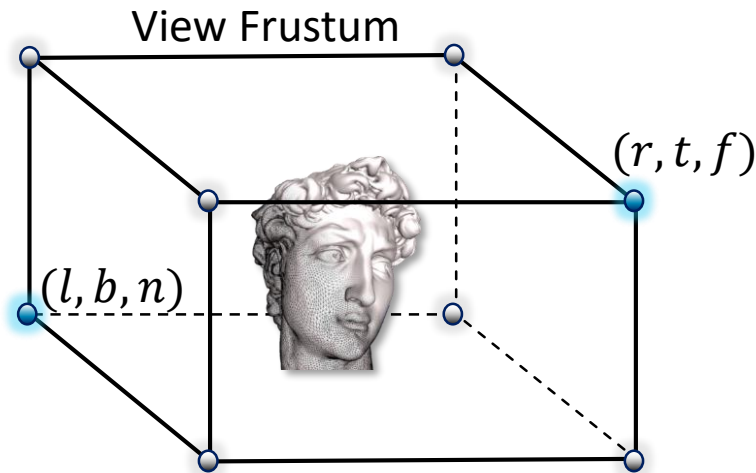
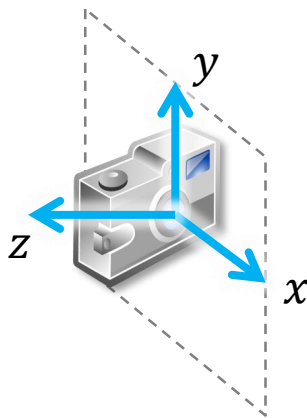
Canonical
Viewing Volume



Orthographische Kamera/Projektion



- ▶ übliche Konvention bei Rasterisierungs-APIs (in Kamerakoordinaten):
 - ▶ x -Achse zeigt nach rechts, y -Achse nach oben, Blick in neg. z -Richtung
- ▶ das Sichtvolumen (View Frustum) ist ein Quader $[l; r] \times [b; t] \times [n; f]$
 - ▶ left l , right r , bottom b , top t , near n und far f
 - ▶ Achtung: $n > f$ (beide negativ)
- ▶ die Bildebene ist parallel zur xy -Ebene

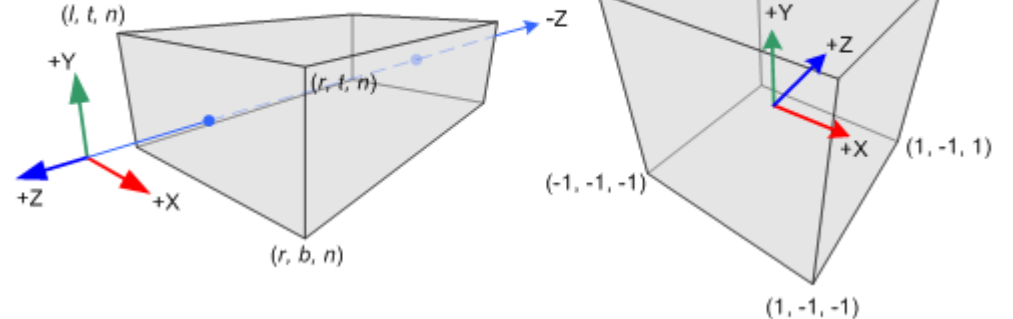


Orthographische Projektion



- ▶ Konvention: wir suchen nun die Abbildung, die das View Frustum $[l; r] \times [b; t] \times [n; f]$ auf den Einheitswürfel $[-1; 1]^3$ abbildet
 - ▶ l auf $x = -1$, r auf $x = 1$, b auf $y = -1$, n auf $z = -1$, ...
 - ▶ **Ziel: z-Wert („Tiefe“) zur Sichtbarkeitsbestimmung verwenden**

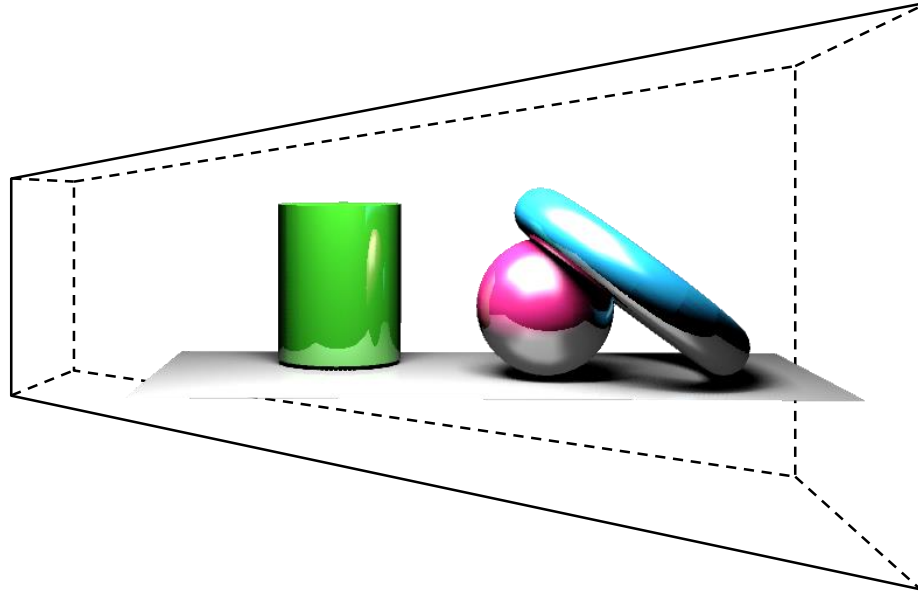
$$\mathbf{M}_{orth} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{n-f} & \frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



- ▶ diese Matrix bereitet nur vor (ist eine „**Projektionstransformation**“), die Projektion im eigentlichen Sinn ist das Weglassen der z-Komponente!
- ▶ die orthographische Projektionstransformation ist eine affine Abbildung

Perspektivische Projektion

- ▶ auch hier suchen wir eine Projektionstransformation, die das Sichtvolumen auf den Einheitswürfel $[-1; 1]^3$ abbildet
→ einheitliche Behandlung unterschiedlicher Projektionen



Perspektivische Projektion in 2D



- ▶ Konvention in diesem Beispiel: Projektionszentrum liegt bei $(0, -D)$ auf der negativen z -Achse, Blick in Richtung der positiven z -Achse

- ▶ Projektion auf die Bildebene mit $z = 0$

$$\frac{x'}{D} = \frac{x}{z + D} \Rightarrow x' = \frac{x}{z/D + 1} = \begin{pmatrix} x_h \\ w_h \end{pmatrix}$$

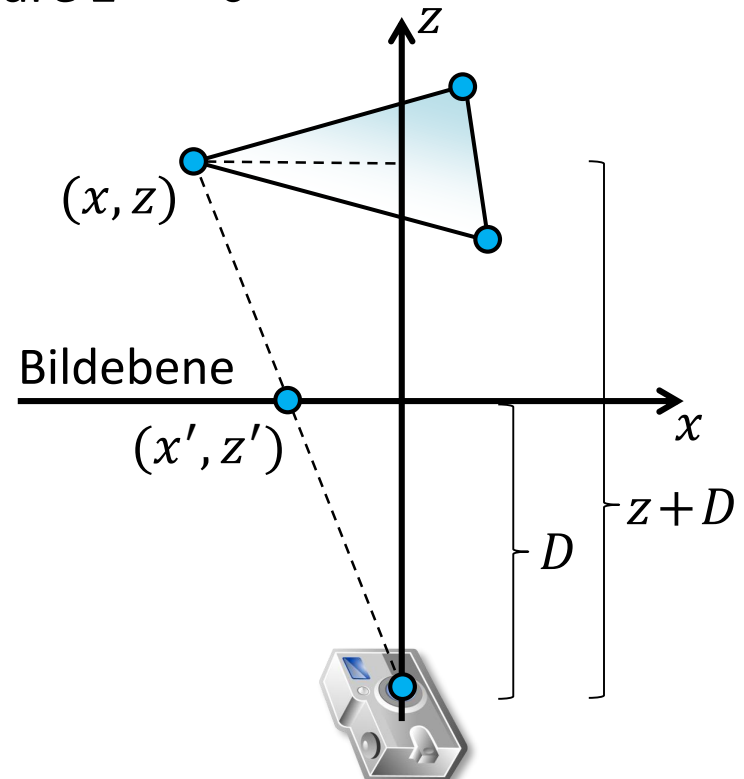
- ▶ nach einer „herkömmlichen“ Projektion wäre $z' = 0$

- ▶ wir berechnen z' analog zu x'

$$\text{mit } z' = \frac{z}{z/D + 1} = \begin{pmatrix} z_h \\ w_h \end{pmatrix}$$

- ▶ ... und heben z' als Tiefe für den späteren Z-Buffer-Test auf

$$\begin{pmatrix} x_h \\ z_h \\ w_h \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/D & 1 \end{pmatrix} \begin{pmatrix} x \\ z \\ 1 \end{pmatrix}$$



Perspektivische Projektion in 2D



- ▶ Konvention in diesem Beispiel: Projektionszentrum liegt bei $(0, -D)$ auf der negativen z -Achse, Blick in Richtung der positiven z -Achse

- ▶ Projektion auf die Bildebene mit $z = 0$

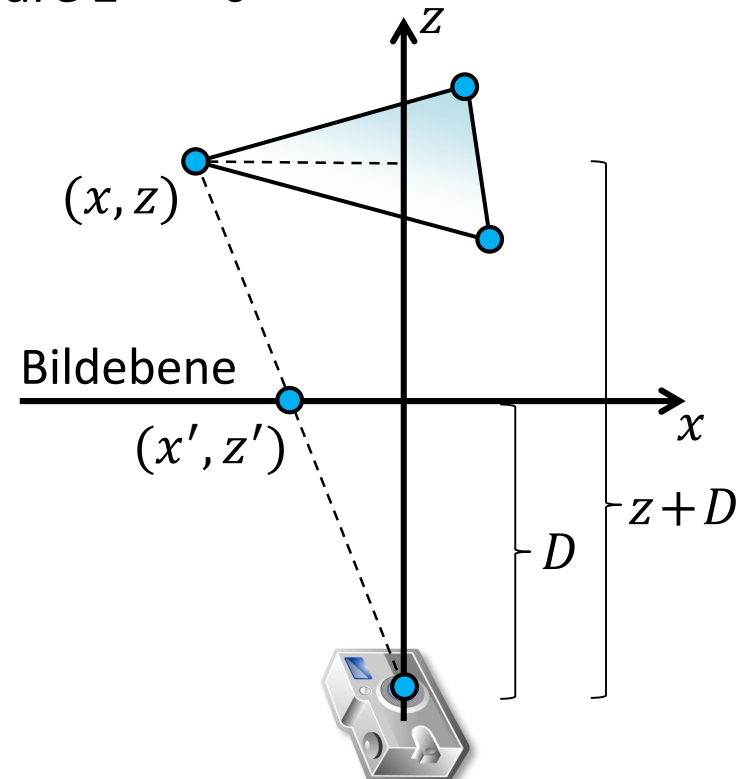
$$\frac{x'}{D} = \frac{x}{z + D} \Rightarrow x' = \frac{x}{z/D + 1} = \begin{pmatrix} x_h \\ w_h \end{pmatrix}$$

- ▶ nach einer „herkömmlichen“ Projektion wäre $z' = 0$

- ▶ warum Tiefe $z' = \frac{z}{z/D + 1}$ und nicht einfach z für Z-Buffering?

- ▶ Genauigkeit bei der Repräsentation der Tiefenwerte

- ▶ perspektivisch-korrekte Attributinterpolation



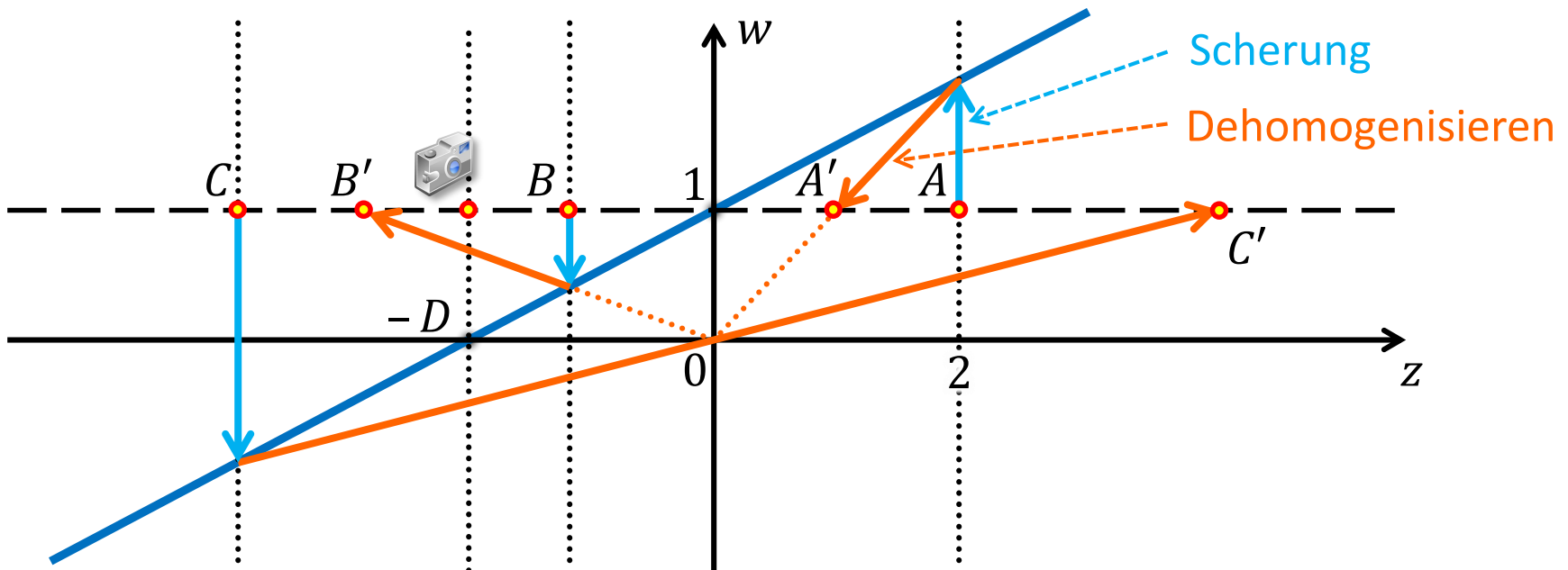
Perspektivische Projektion in 2D

Betrachtung der Abbildung des Tiefenwerts

▶ wie hängt z und $z' = \frac{z}{z/D+1}$ zusammen?

▶ Scherung in w -Richtung (hier $D = 2$)
und anschließende Dehomogenisierung

$$\begin{pmatrix} x_h \\ z_h \\ w_h \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/D & 1 \end{pmatrix} \begin{pmatrix} x \\ z \\ 1 \end{pmatrix}$$



A : vor der Kamera, $z > 0$

→ kleinerer, positiver Tiefenwert

B : vor der Kamera, aber $z < 0$

→ Abbildung auf negative Werte

C : hinter der Kamera, $z < -D$

→ Abbildung auf positive Werte

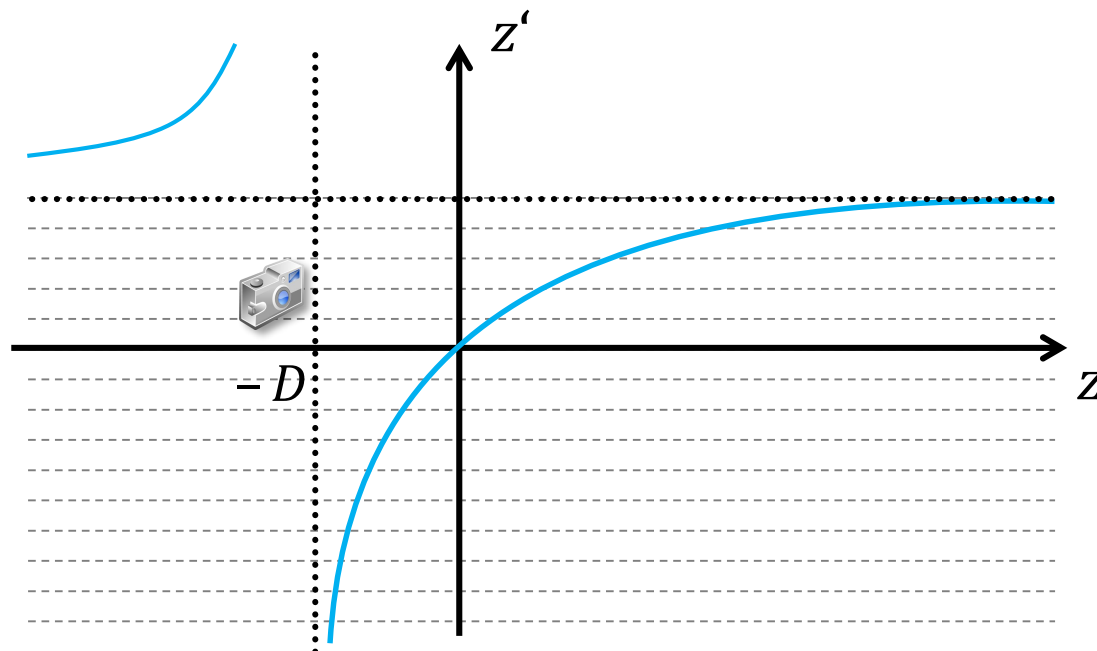
Kameraposition

→ $-\infty$

Perspektivische Projektion in 2D

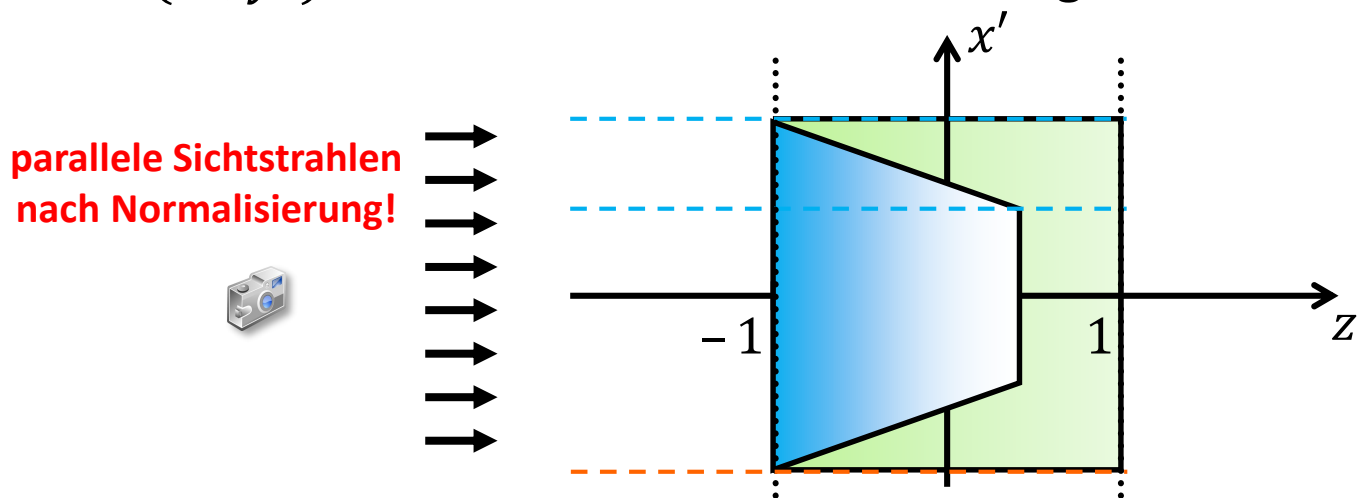
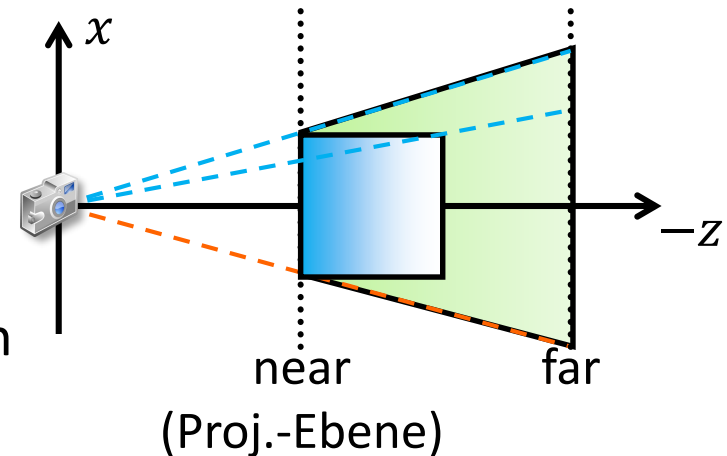
Betrachtung der Abbildung des Tiefenwerts

- ▶ Nicht-Linearität der z -Transformation
 - ▶ wir möchten z' diskretisieren und für den Tiefentest verwenden, um herauszufinden welches Primitiv in einem Pixel vor welchem liegt
 - ▶ **gut: höhere Genauigkeit für nahe Oberflächen,**
dafür wenig Präzision für entfernte Oberflächen (große Werte)
- ▶ positive Werte für Objekte hinter der Kamera, gegen $-\infty$ für $z \rightarrow -D$
- ▶ Lösung: beschränke Tiefenbereich des View Frustum mit **Near und Far (Clipping) Planes**



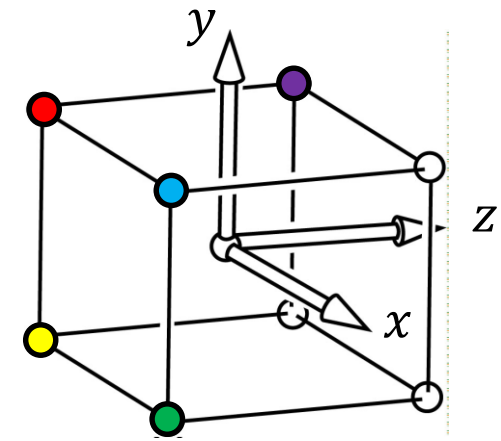
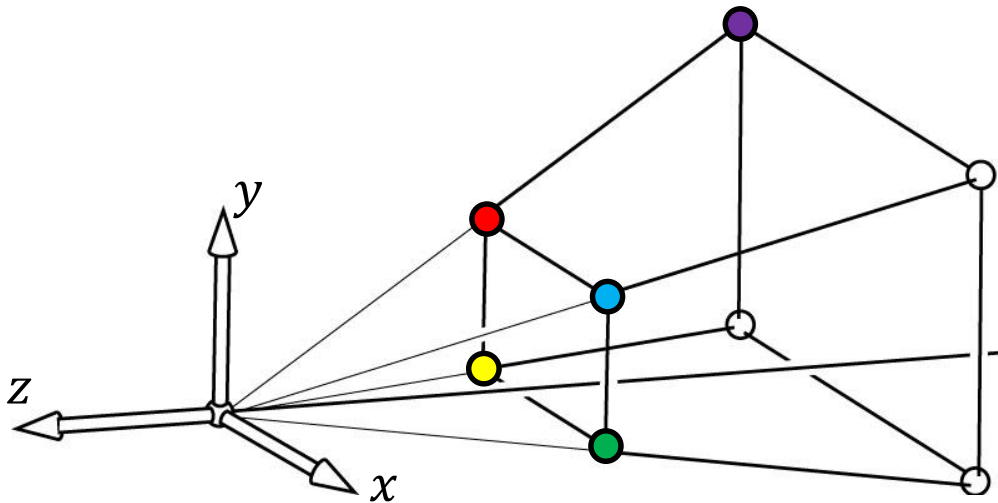
Perspektivische Projektion in 2D

- ▶ wir suchen eine spezielle Projektion so, dass die **Sichtpyramide** auf den Einheitswürfel $[-1; 1]^3$ abgebildet wird, d.h. auch „near“ auf -1 und „far“ auf $+1$
- ▶ nach der Projektionstransformation haben wir noch homogene Koordinaten!
- ▶ Dehomogenisieren danach nennt man „Normalisierungstransformation“
 - ▶ ergibt sog. **Normalized Device Coordinates**, Bildschirmkoordinaten (x', y') erhalten wir danach durch Weglassen der Tiefe



Perspektivische Projektion in 3D

- ▶ Projektionstransformation und anschließendes Dehomogenisieren (die Normalisierungstransformation) bilden zusammen das View Frustum auf den Einheitswürfel $[-1; 1]^3$ (in kartesischen Koordinaten) ab

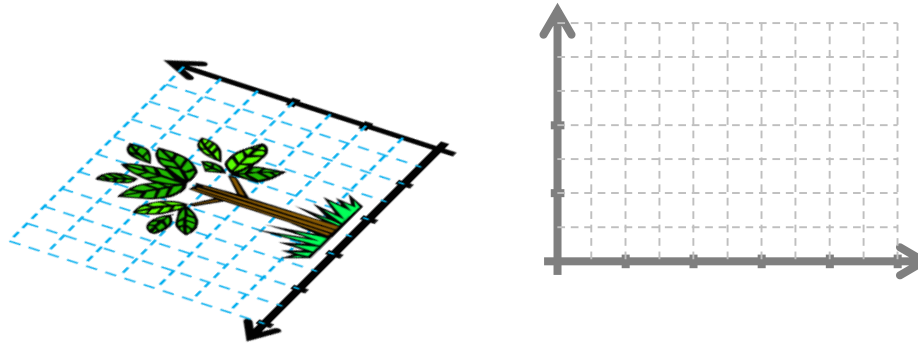


- ▶ eine oft gebrauchte Konvention (z.B. auch bei OpenGL) ist
 - ▶ Kamera befindet sich im Ursprung
 - ▶ Blickrichtung entlang der negativen z-Achse
- ▶ wie viele Punkte bestimmen eindeutig die Projektionstransformation?

Bestimmung der Projektionsmatrix



- ▶ wie viele Punkte bestimmen eindeutig eine affine 3D-Transformation?
 - ▶ 3×4 Einträge einer 4×4 Matrix (linearer Teil und Translation)
 - ▶ 12 Unbekannte, also 4 Punkte in 3D



- ▶ eine Projektion in 3D ist definiert durch die Abb. von 5 Punkten im \mathbb{R}^3
 - ▶ 4×4 Matrix, aber homogene Koordinaten sind skalierungsinvariant
 - ▶ $5 \times 3 = 4 \times 4 - 1$ Elemente
- ▶ Projektion in 2D: $3 \times 3 - 1 = 8$ Elemente, also 4 Punkte im \mathbb{R}^2

Bestimmung der Projektionsmatrix (in 2D)

2D Projektionstransformation: 3×3 – 1 Unbekannte

- ▶ betrachte Abbildung von 4 Punkten in 2D
(Faktoren k_i wg. homogenen Koordinaten)

- ▶ Kameraposition nach $-\infty$

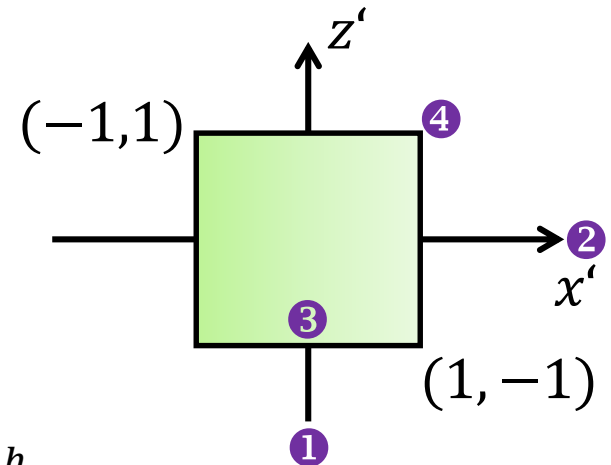
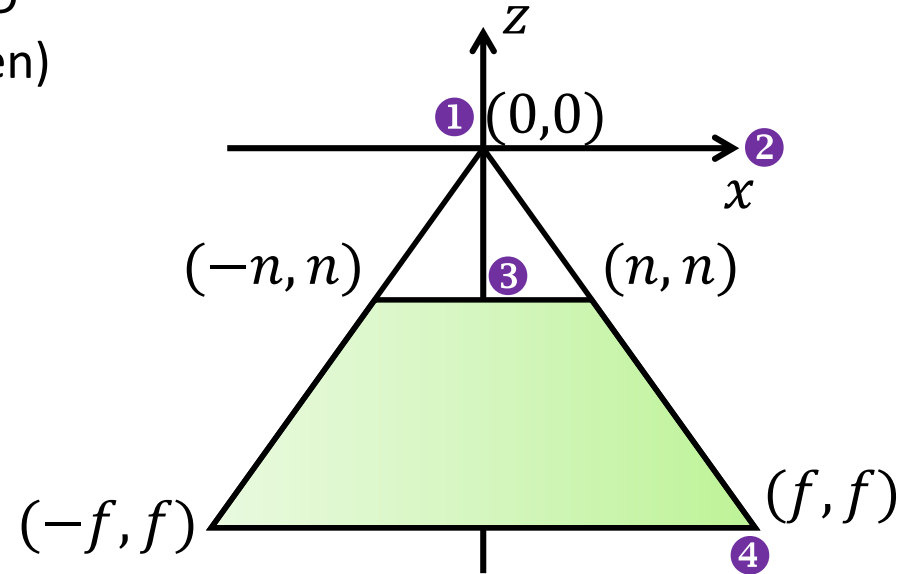
1
$$\mathbf{M} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}_h = k_1 \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix}_h$$

- ▶ x -Richtung beibehalten

2
$$\mathbf{M} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}_h = k_2 \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}_h$$

- ▶ zwei weitere Punkte (Achtung: nicht 3 Punkte auf einer Geraden wählen)

3
$$\mathbf{M} \begin{pmatrix} 0 \\ n \\ 1 \end{pmatrix}_h = k_3 \begin{pmatrix} 0 \\ -1 \\ 1 \end{pmatrix}_h$$
 4
$$\mathbf{M} \begin{pmatrix} f \\ f \\ 1 \end{pmatrix}_h = k_4 \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}_h$$



Bestimmung der Projektionsmatrix (in 2D)



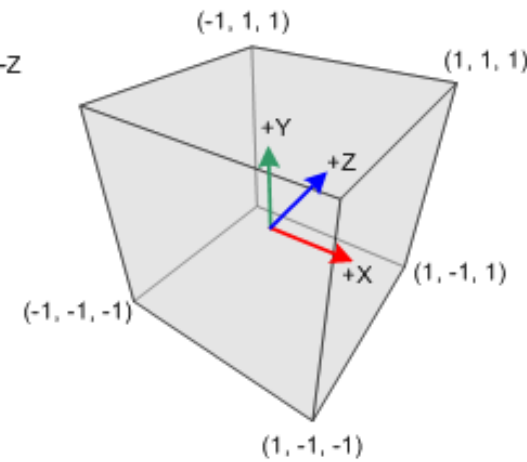
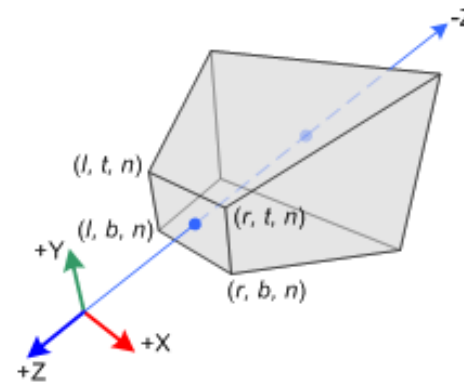
▶ ergibt 12 Gleichungen für 12 Unbekannte: $3 \times 3 - 1 + 4$ (für k_1, \dots, k_4)

▶ Ergebnis in diesem Beispiel:

$$\mathbf{M}_{2D} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 1 & 0 \end{pmatrix}$$

▶ Projektionstransformation in 3D (mit beliebig asymmetrischem Frustum)

$$\mathbf{M}_{3D} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{-2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$



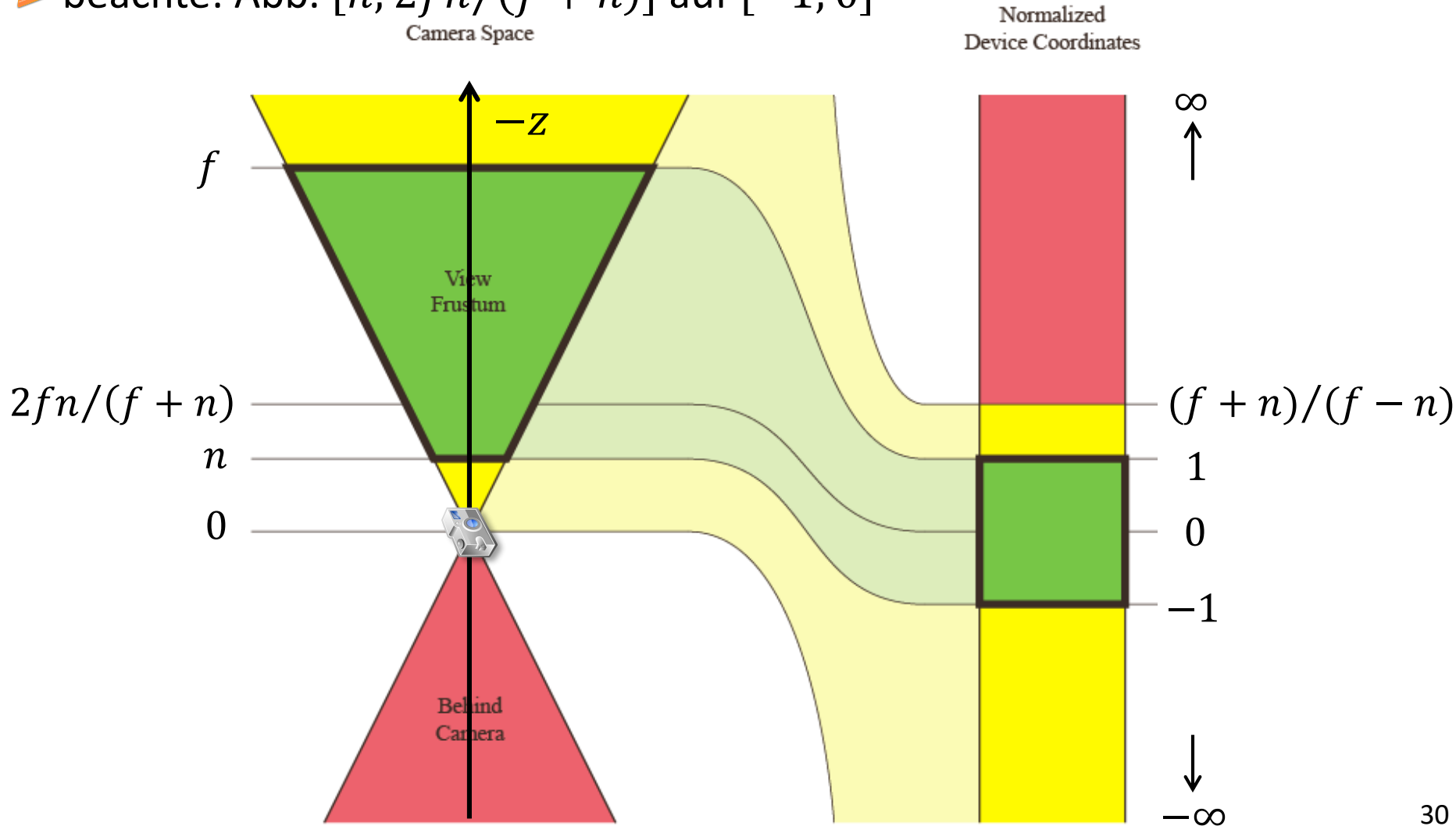
http://www.songho.ca/opengl/gl_projectionmatrix.html

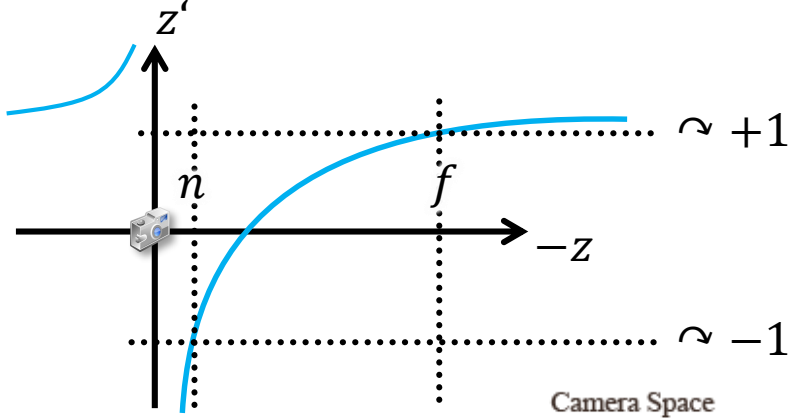
<http://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/>

Perspektivische Projektion in 2D

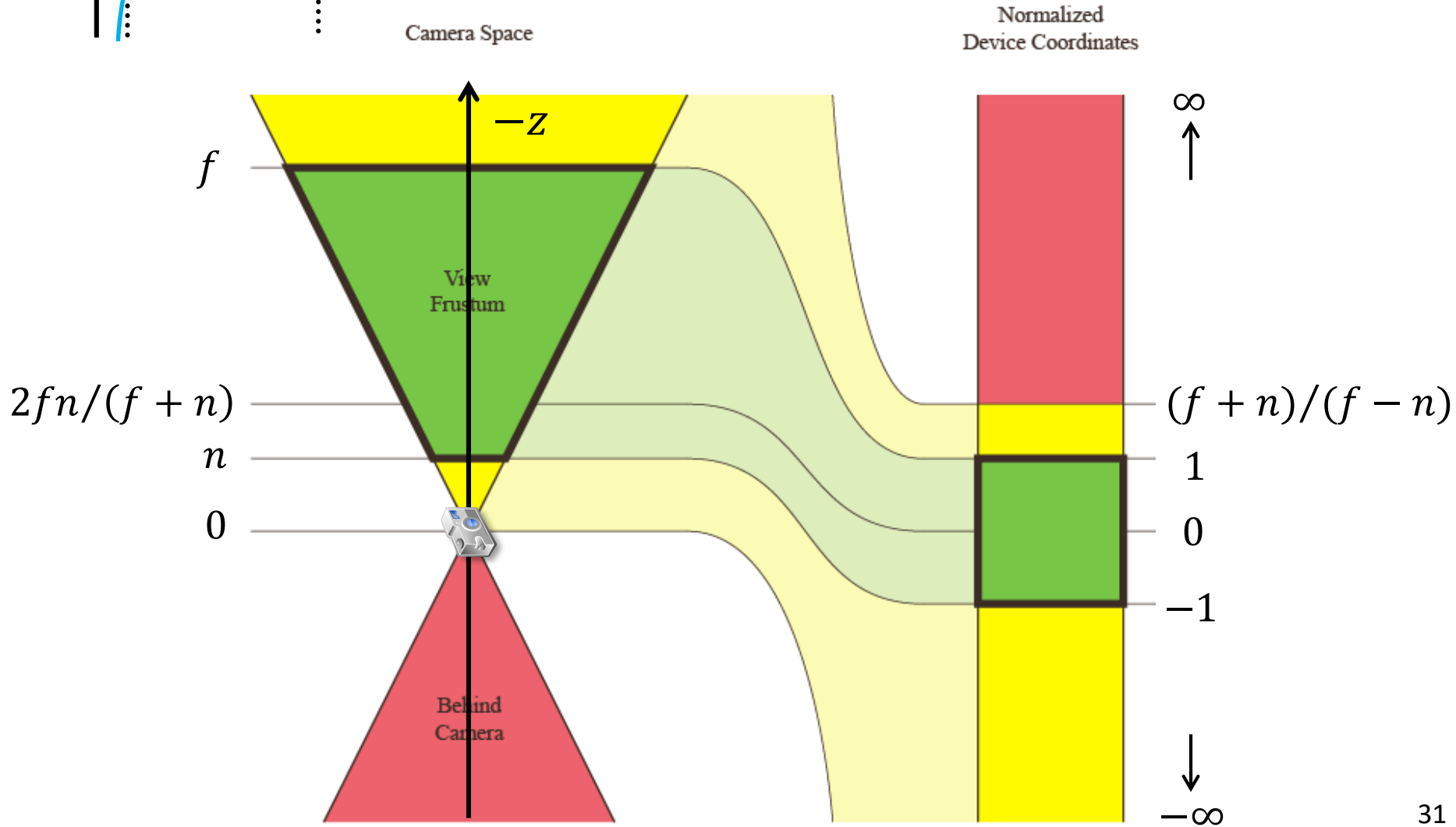
Normalisierungstransformation und Clipping

- ▶ betrachten wir nochmals die Abbildung der Tiefe...
- ▶ beachte: Abb. $[n; 2fn/(f + n)]$ auf $[-1; 0]$





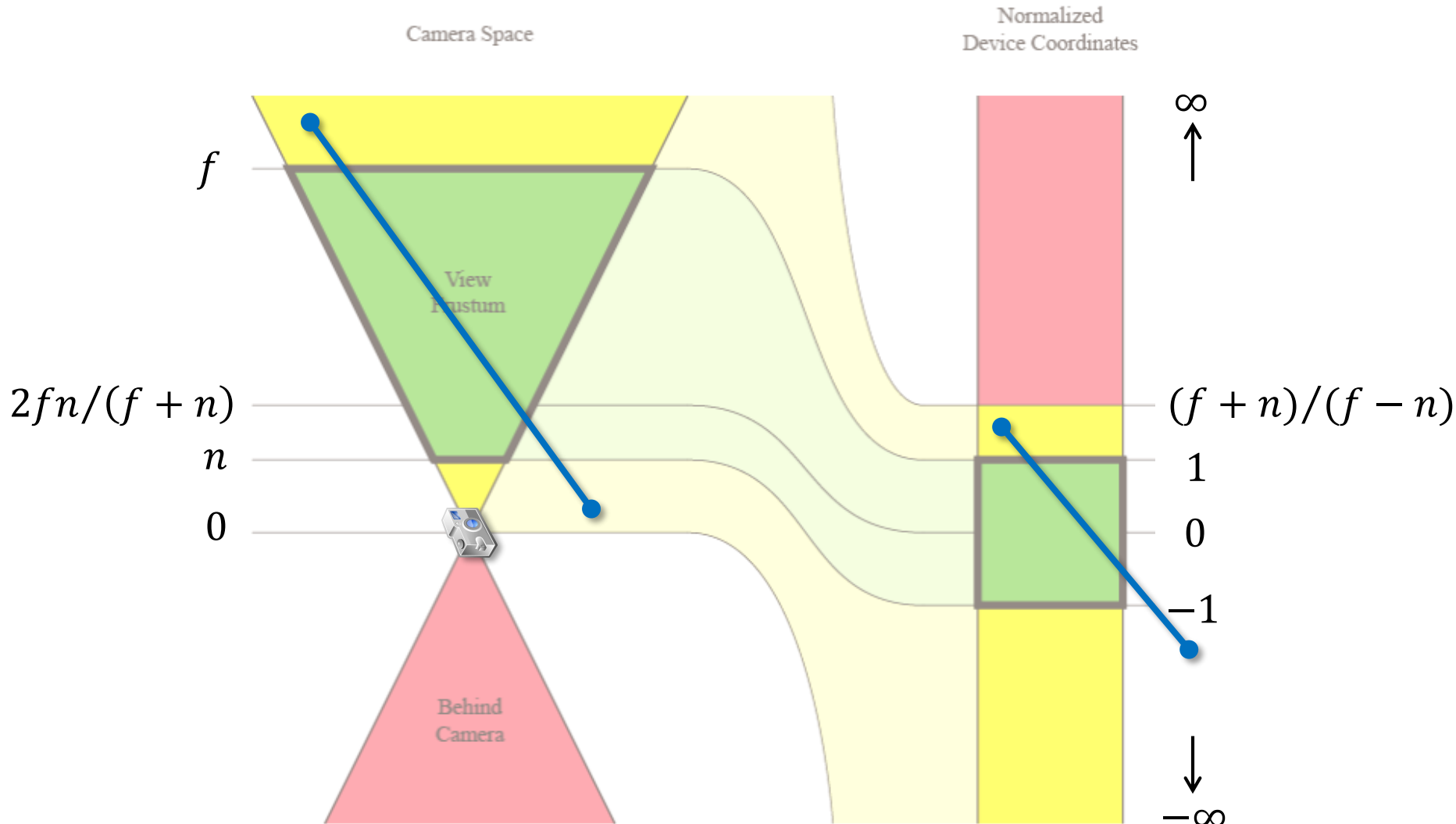
wähle $|n|$ immer so groß wie möglich!



Perspektivische Projektion in 2D

Normalisierungstransformation und Clipping

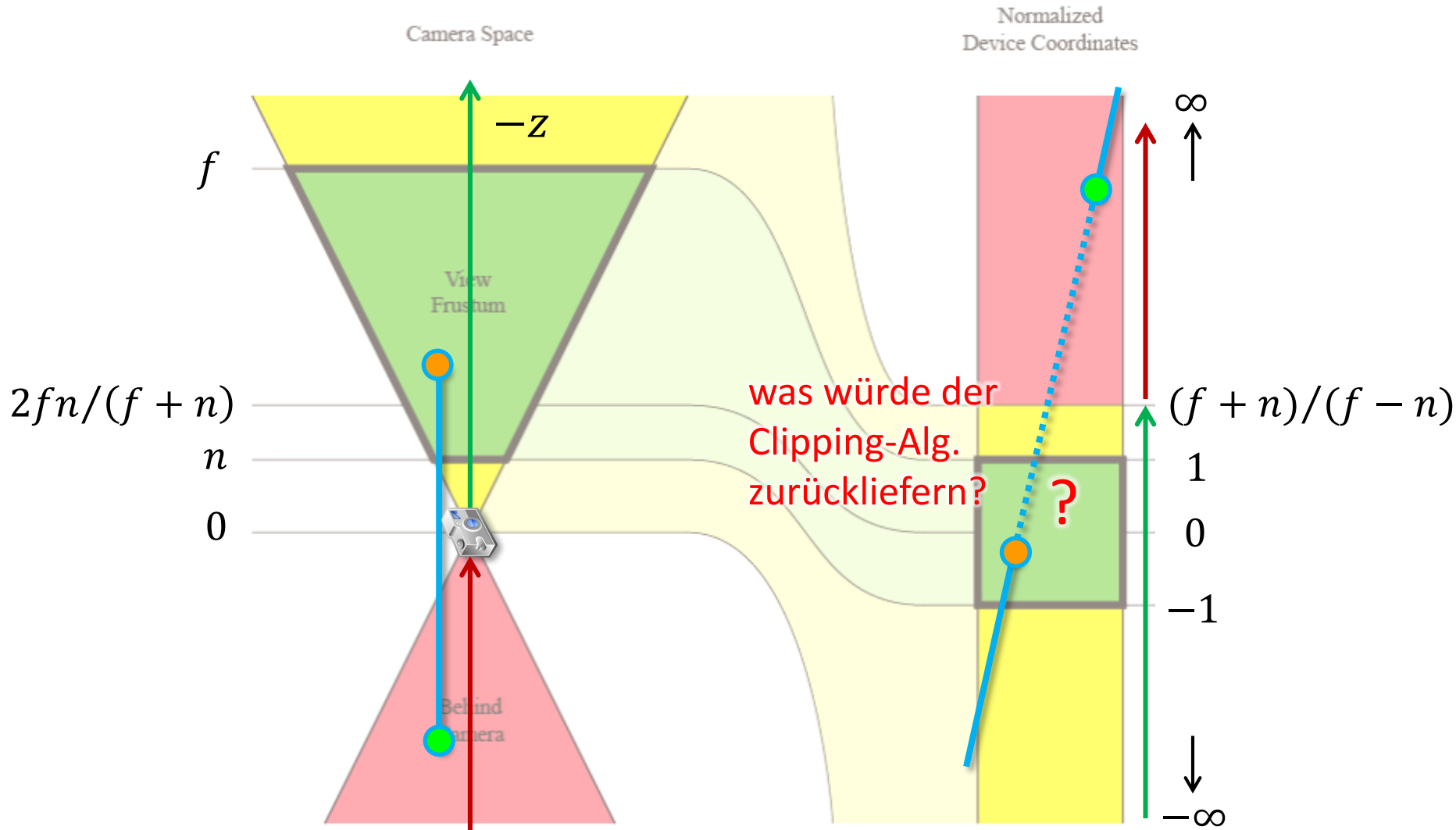
► Abbildung der Tiefe bereitet beim Clipping Schwierigkeiten



Perspektivische Projektion in 2D

Normalisierungstransformation und Clipping

► Abbildung der Tiefe bereitet beim Clipping Schwierigkeiten

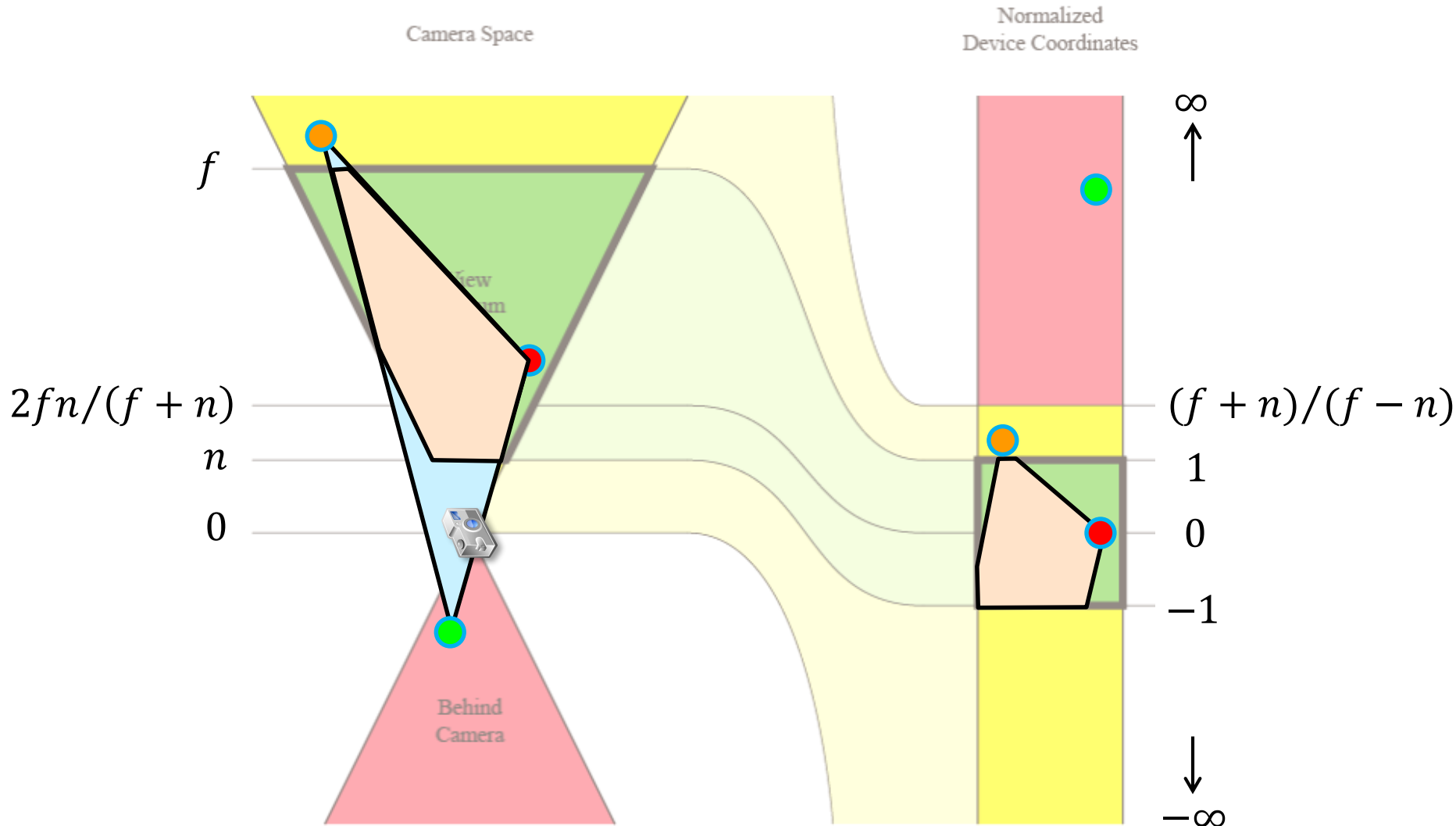


- ▶ **Clipping** in kartesischen Koordinaten **nach der Normalisierungstransformation ist falsch:**
Ambiguitäten und falsche Liniensegmente
- ▶ nach der Dehomogenisierung ($x' = x/w$) müssten Tiefenwerte von Liniensegmenten betrachtet werden, um problematische Fälle zu erkennen
- ▶ es kann auch passieren, dass $w = -z = 0$ ist → Division durch 0?
- ▶ aber wie würde Clipping für ein Polygon funktionieren?

Perspektivische Projektion in 2D

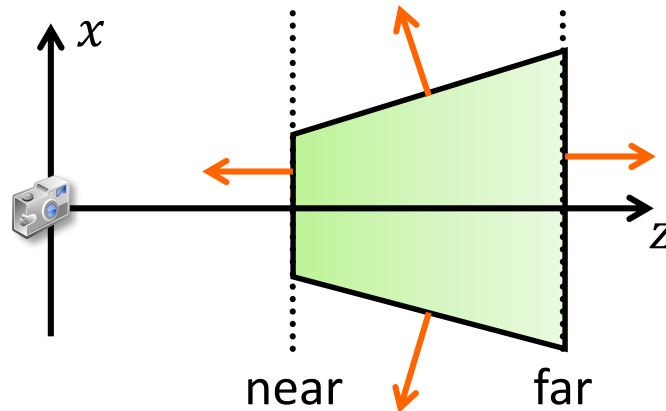
Normalisierungstransformation und Clipping

► Abbildung der Tiefe bereitet beim Clipping Schwierigkeiten



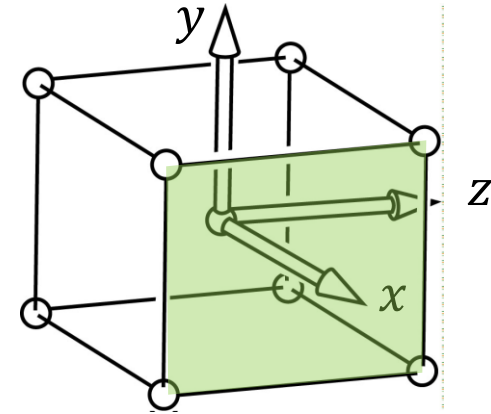
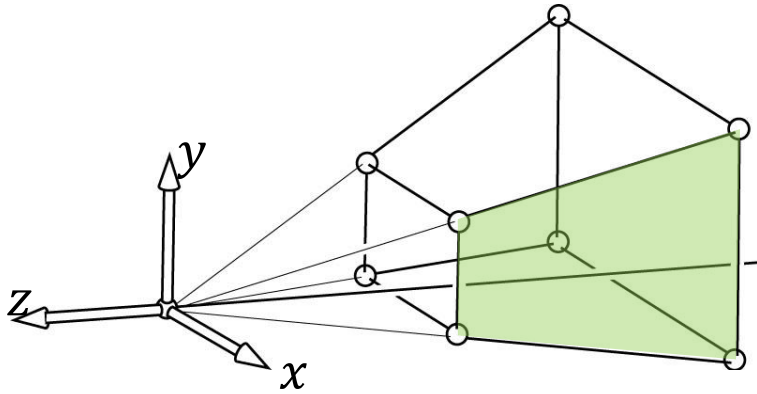
Clipping und Projektionen

- ▶ Option 1: Clipping gegen das Frustum in Welt- oder Kamerakoordinaten **vor** der Projektionstransformation
 - ▶ stelle 6 Ebenengleichungen auf (geht direkt aus Projektionsmatrix)
 - ▶ konvexer Körper → Clipping wie bisher/wie in der Übung
 - ▶ funktioniert, aber es geht effizienter!



Clipping in homogenen Koordinaten

- ▶ Option 2: Clipping in homogenen Clip-Koordinaten
 - ▶ nach der Projektionstransformation, aber vor dem Dehomogenisieren

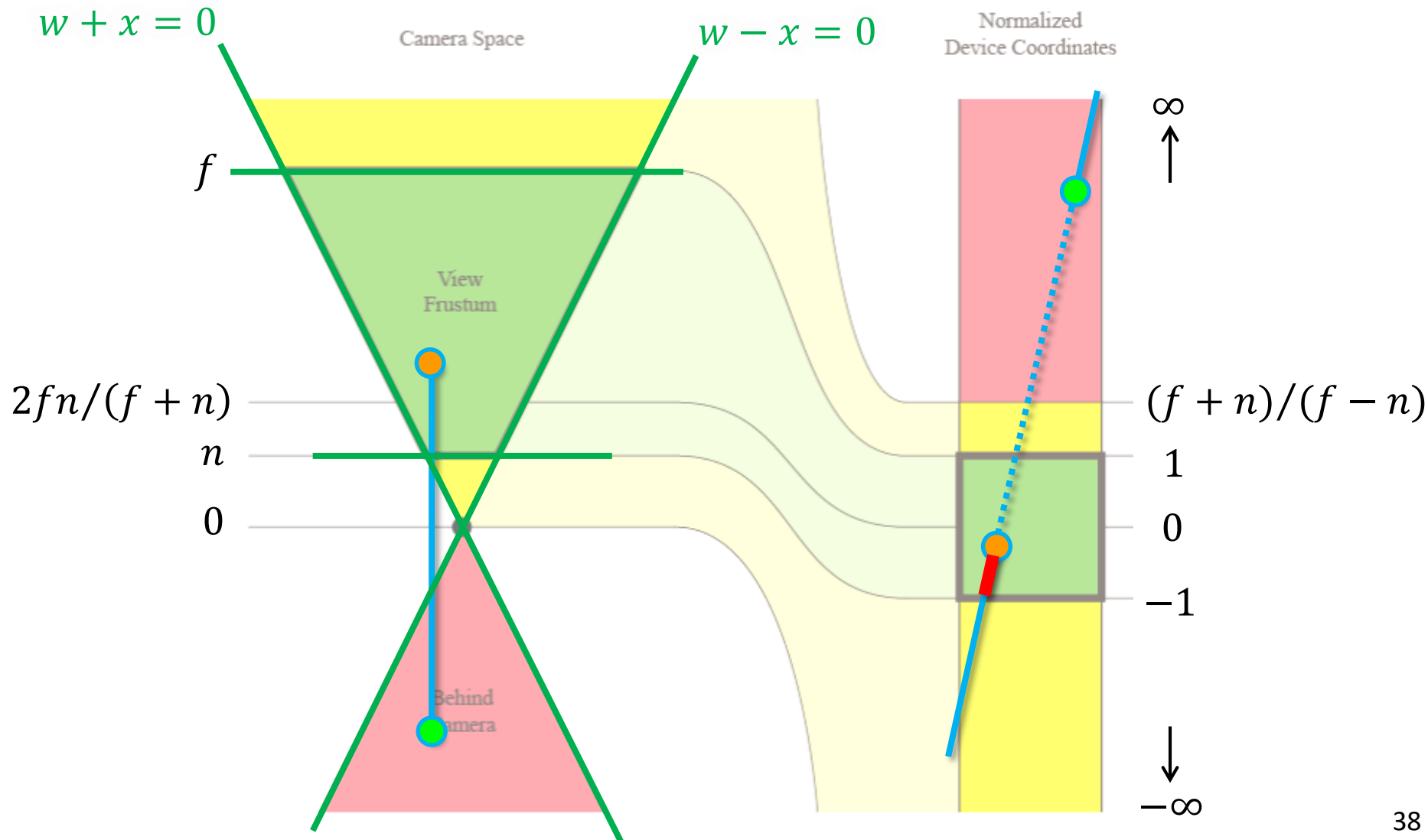


- ▶ für alle Punkte auf der grünen Seite gilt nach dem Dehomogenisieren (also in NDC): $x' = x/w = 1$
- ▶ vor dem Dehomogenisieren gilt: $x = w$
- ▶ erlaubt alle Transformationen (Modell, Kamera, Projektion) in einer Matrix zusammenzufassen

Perspektivische Projektion in 2D

Normalisierungstransformation und Clipping

► Clipping geg. alle Seiten vor Normalisierungstrafo löst Mehrdeutigkeiten



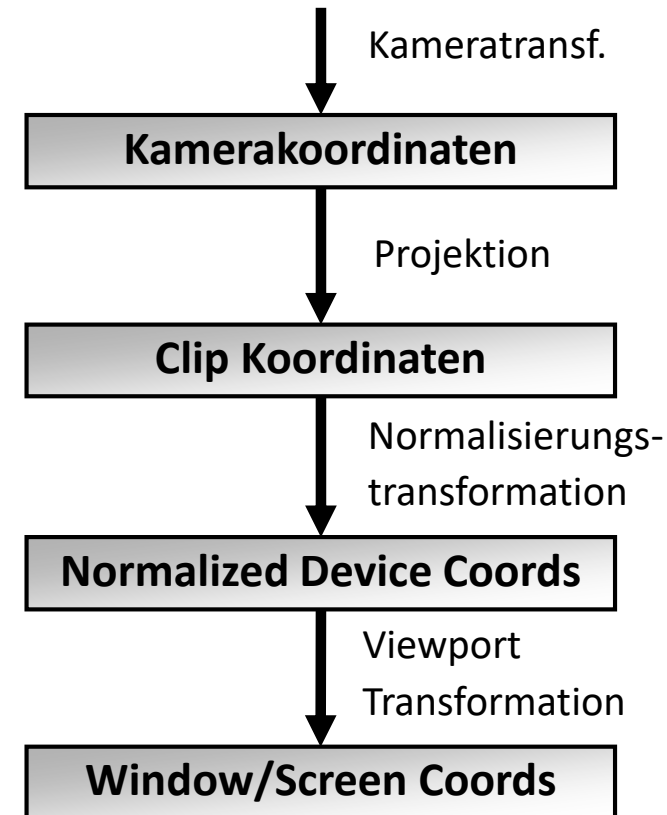
Clipping in homogenen Koordinaten



- ▶ Option 2: Clipping in homogenen Clip-Koordinaten
 - ▶ nach der Projektionstransformation, aber vor dem Dehomogenisieren

- ▶ Vorgehen (so macht man es!):

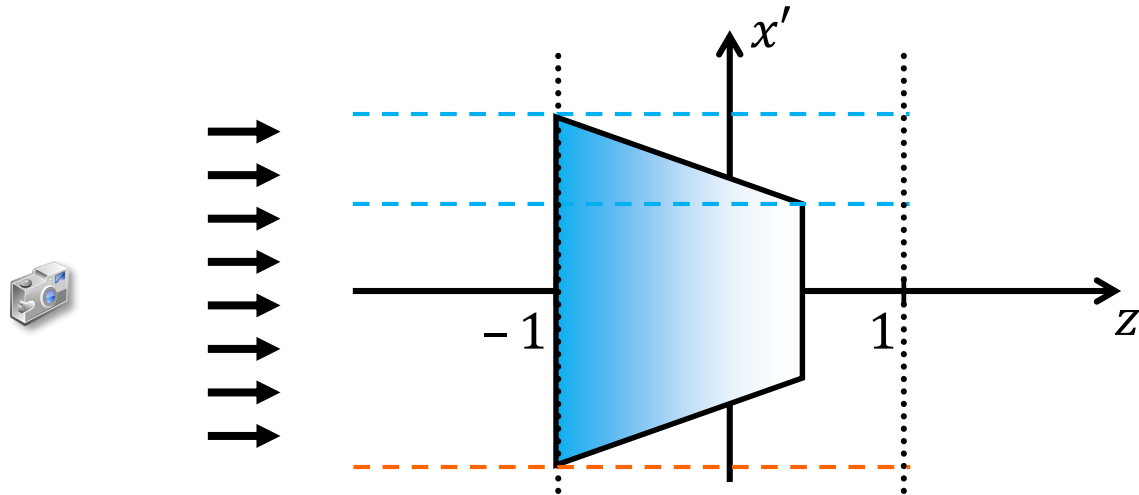
- ▶ Projektionstransformation
- ▶ Clipping in homogenen Koordinaten
- ▶ dann perspektivische Division durch w
→ ergibt Normalized Device Coords.
(Einheitswürfel)
- ▶ dann Projektion auf Bildschirmkoordinaten
und Viewport-Transformation
(Abb. von $x, y \in [-1; 1]$ auf
Bildschirmauflösung Breite \times Höhe)



NDC-To-Viewport Transformation



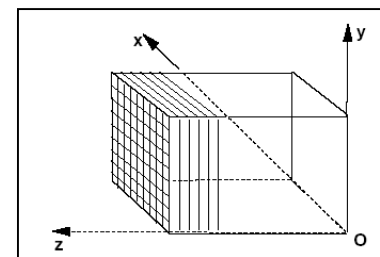
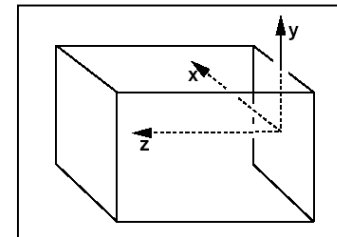
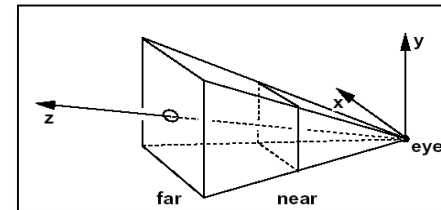
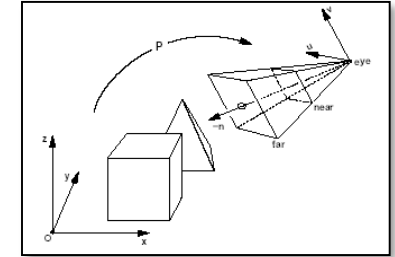
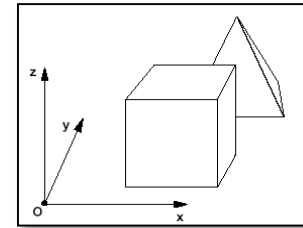
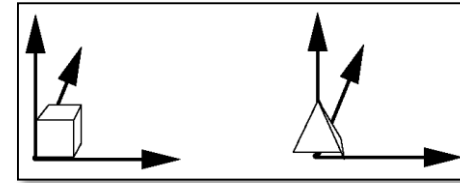
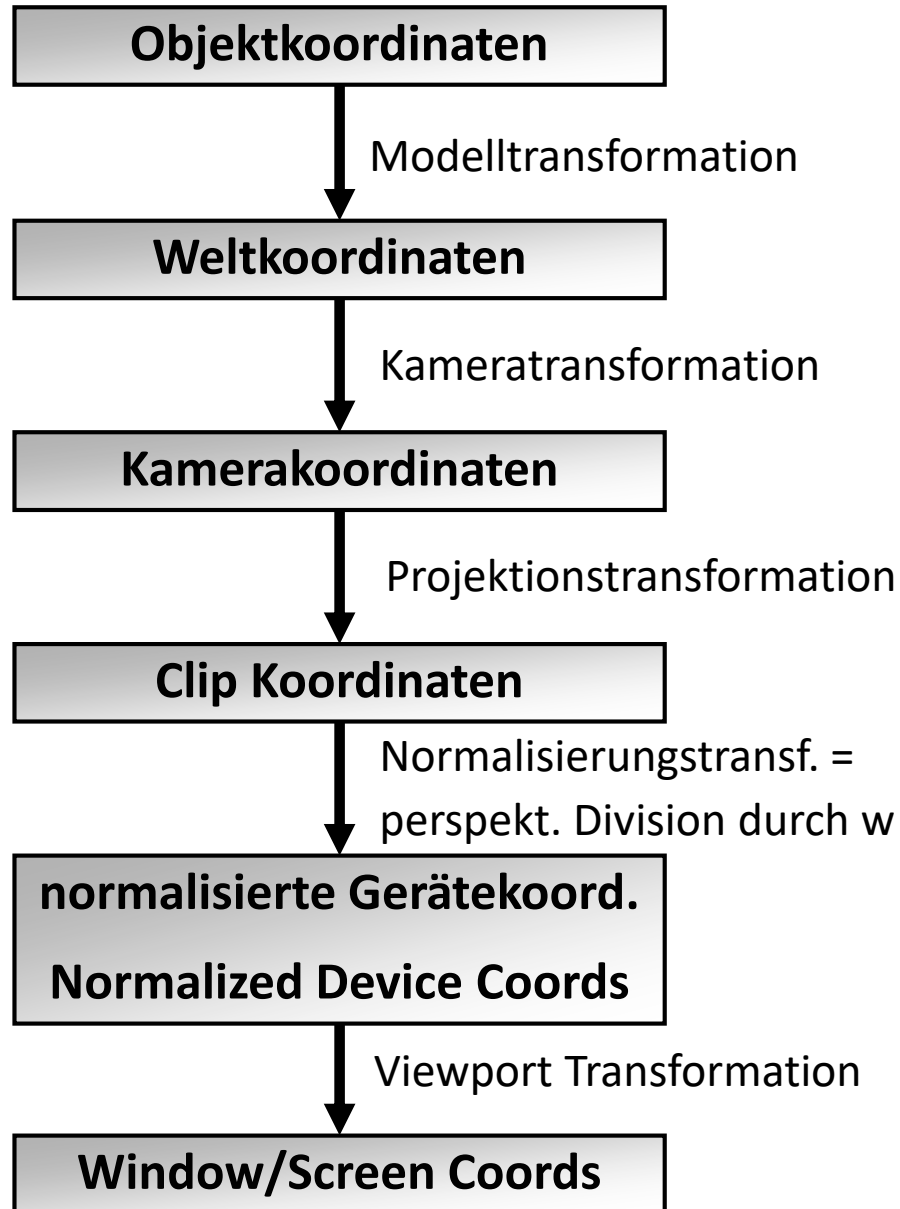
- ▶ der Viewport ist der rechteckige Bereich in dem das Bild dargestellt wird
- ▶ projizierte Koordinaten (x', y') durch Weglassen der Tiefe (= orthographische Projektion)



- ▶ (x', y') liegen im Bereich $[-1; 1]^2$ und wir benötigen Pixelkoordinaten
- ▶ deshalb verwendet man die Viewport-Transformation, die NDC auf den Bildbereich $[x; x + width] \times [y; y + height]$ abbildet:

$$x_w = \frac{1}{2} (x' + 1) \cdot width + x \quad \text{bzw.} \quad y_w = \frac{1}{2} (y' + 1) \cdot height + y$$

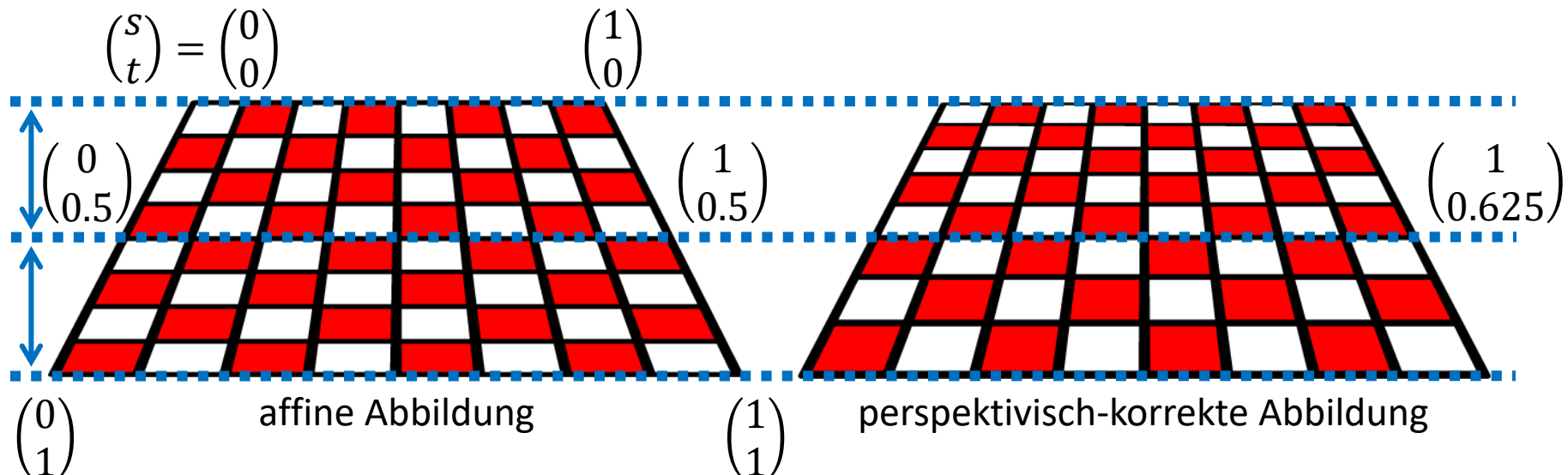
Koordinatensysteme in der CG



Perspektivisch-korrekte Attribut-Interpolation



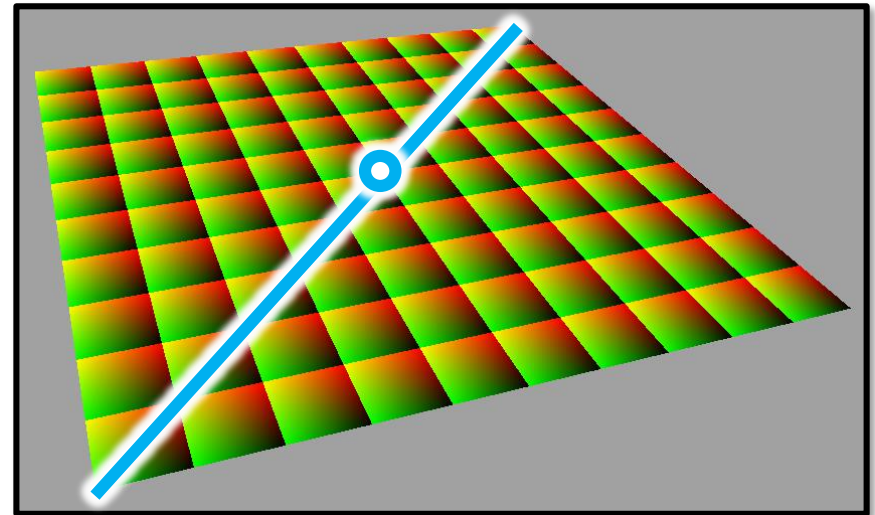
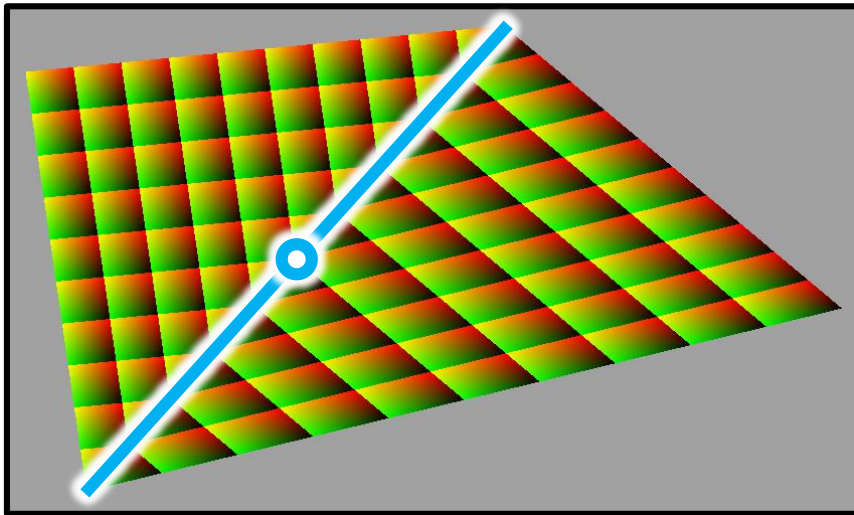
- ▶ lineare Interpolation im Bildraum bei der Interpolation von Texturkoordinaten, Farben, Normalen etc. führt zu Verzerrungen bei perspektivischen Abbildungen
- ▶ lineare Interpolation = baryzentrische Koordinaten im Bildraum
- ▶ lineare Interpolation (bzw. linearer Teil der affinen Interpol.) ändert Längenverhältnisse nicht, perspektivische Projektionen aber schon!



Perspektivisch-korrekte Attribut-Interpolation



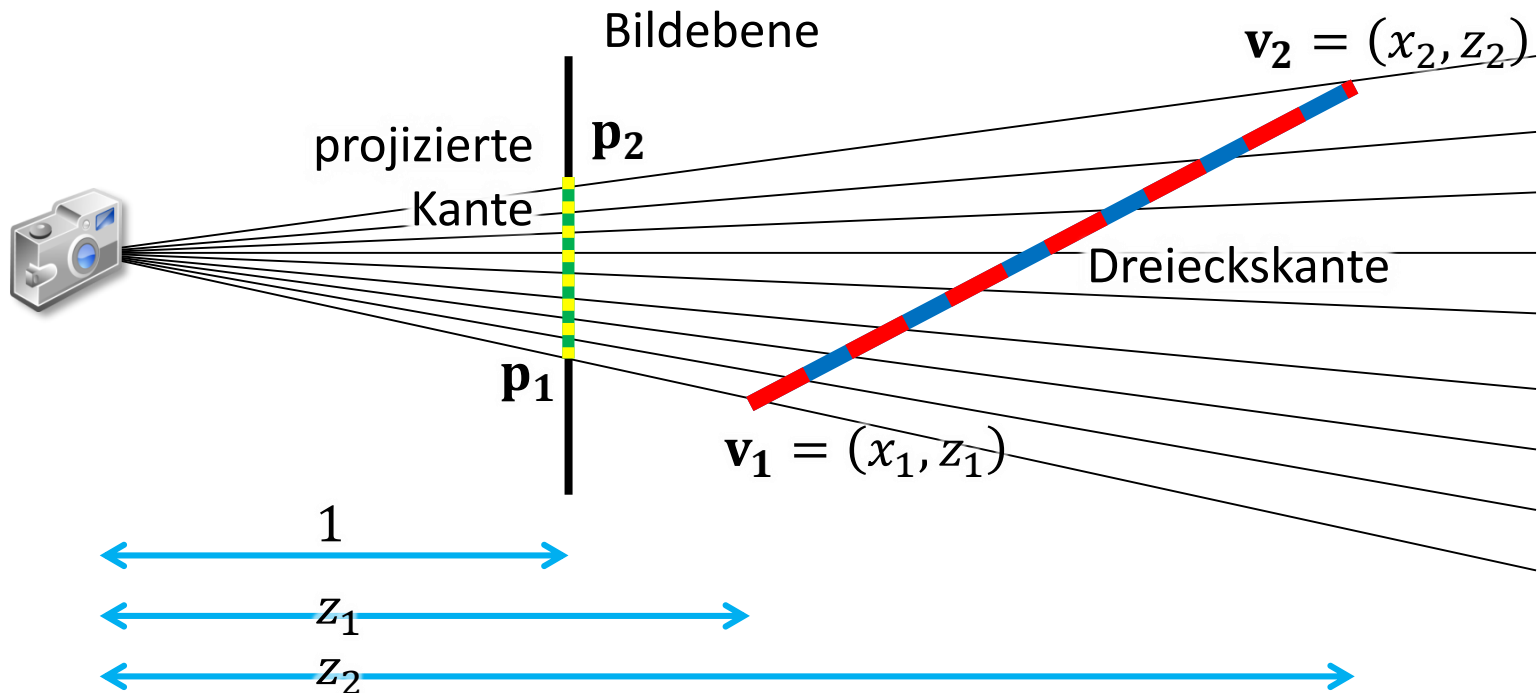
- ▶ lineare Interpolation im Bildraum führt zu Verzerrungen
 - ▶ lineare Interpolation (bzw. linearer Teil der affinen Interpol.) ändert Längenverhältnisse nicht, perspektivische Projektionen aber schon!
 - ▶ am deutlichsten sichtbar bei Textur(koordinaten)
- ▶ Bsp.: perspektivisch-korrekte Interpolation (rechts) und Interpolation im Bildraum (links), das Quadrat besteht aus 2 Dreiecken



Veranschaulichung in 2D:

- ▶ Bildebene bei $z = 1$ (obdA)
- ▶ lineare Interpolation entlang der Kante $\overline{\mathbf{v}_1\mathbf{v}_2}$ mit $\mathbf{v}_1 = (x_1, z_1)$ und $\mathbf{v}_2 = (x_2, z_2)$ **im Bildraum:**

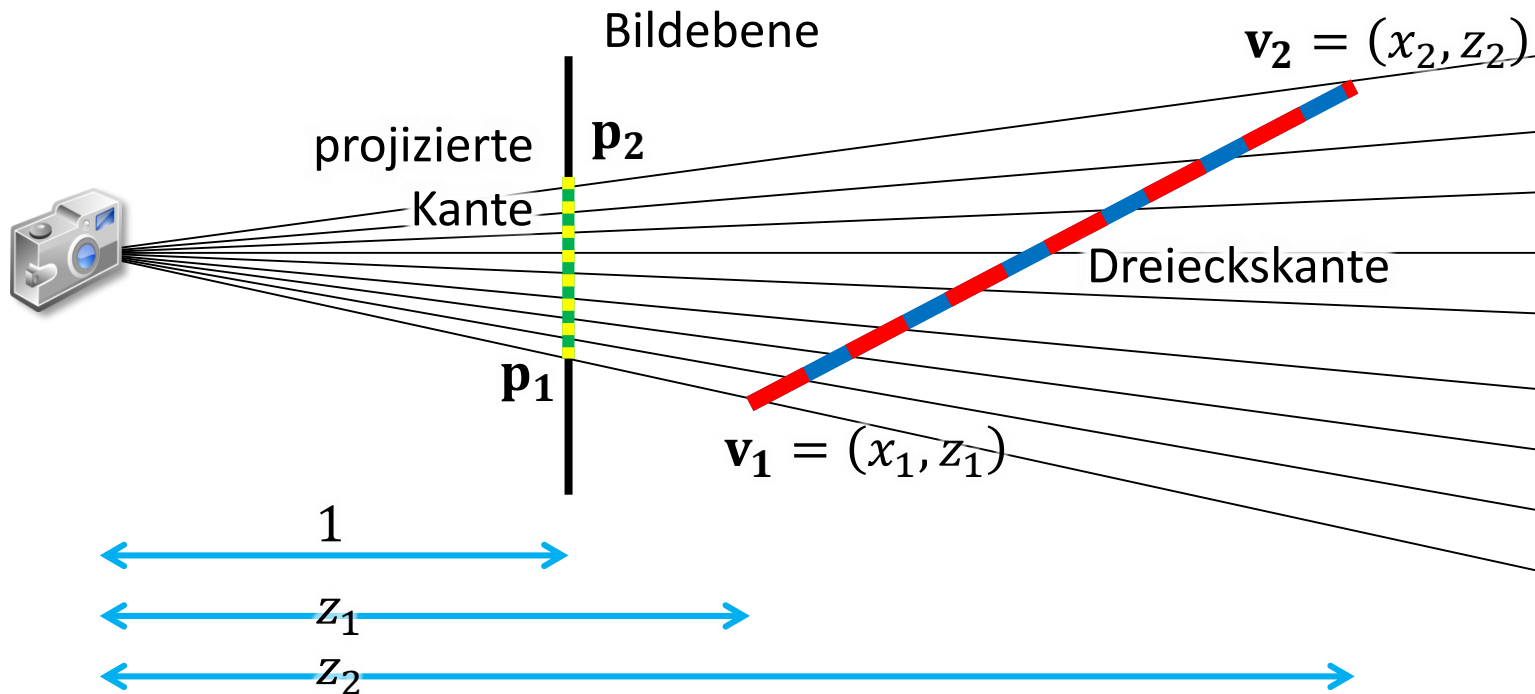
$$\mathbf{p}_{\text{Bild}}(t) = \mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1) = \frac{x_1}{z_1} + t \left(\frac{x_2}{z_2} - \frac{x_1}{z_1} \right)$$



Veranschaulichung in 2D:

► lineare Interpolation entlang der Kante $\overline{\mathbf{v}_1\mathbf{v}_2}$ **vor der Projektion**:

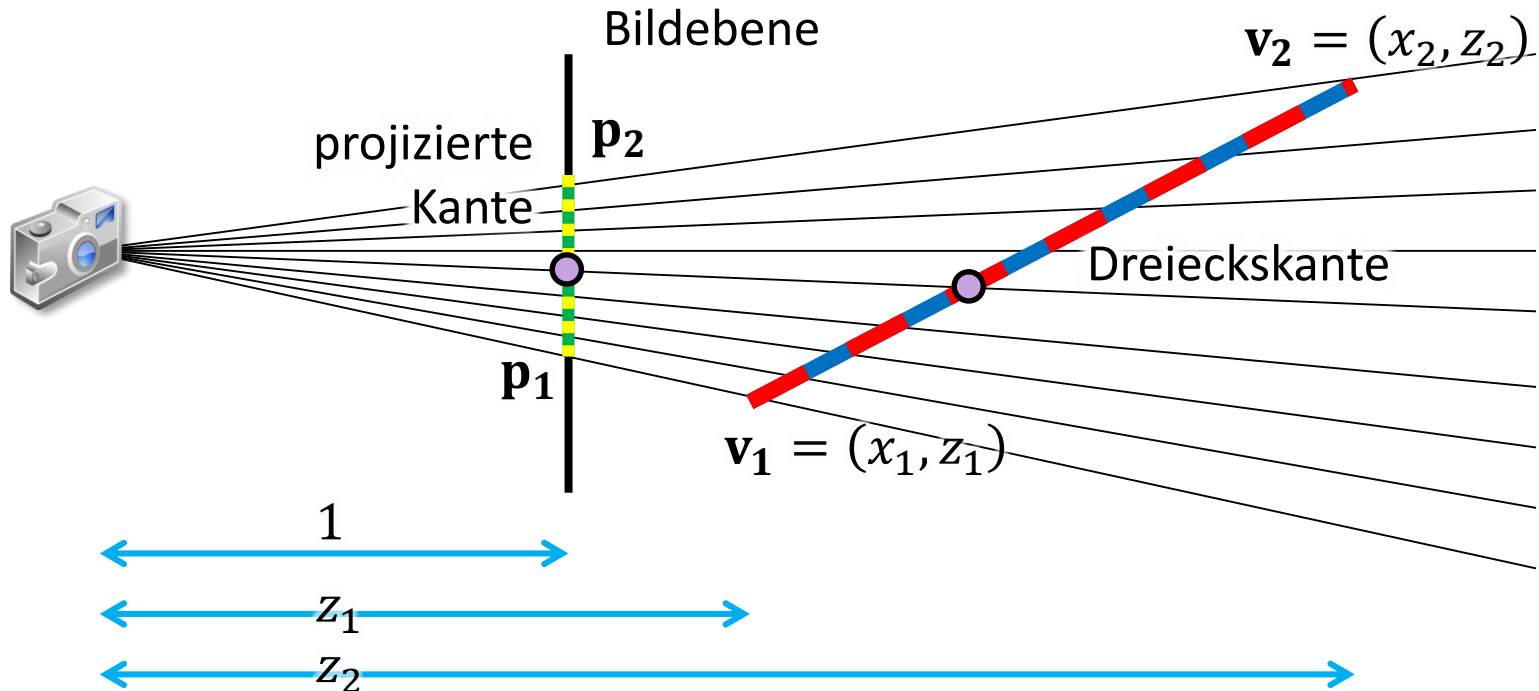
$$\begin{pmatrix} x \\ z \end{pmatrix} = \begin{pmatrix} x_1 \\ z_1 \end{pmatrix} + t \left(\begin{pmatrix} x_2 \\ z_2 \end{pmatrix} - \begin{pmatrix} x_1 \\ z_1 \end{pmatrix} \right) \Rightarrow \mathbf{p}_{\text{Persp}}(t) = \frac{x_1 + t(x_2 - x_1)}{z_1 + t(z_2 - z_1)}$$



- ▶ im Allgemeinen **gilt natürlich**

$$\frac{x_1}{z_1} + t \left(\frac{x_2}{z_2} - \frac{x_1}{z_1} \right) = \mathbf{p}_{\text{Bild}}(t) \neq \mathbf{p}_{\text{Persp}}(t) = \frac{x_1 + t(x_2 - x_1)}{z_1 + t(z_2 - z_1)}$$

- ▶ wenn dies nicht für Punkte auf der Kante gilt, dann auch nicht für Attribute (Anm. affine Abbildung Punkt \leftrightarrow Attribut)

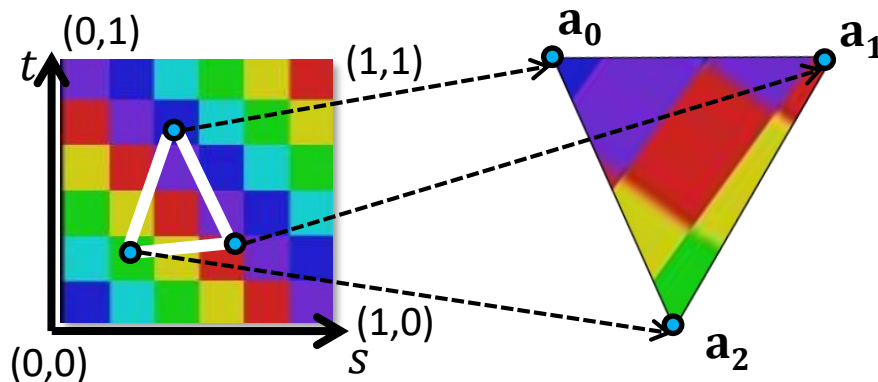


Affine Abbildung von Texturraum \leftrightarrow Objektraum

- ▶ die Texturkoordinaten der Vertices repräsentieren die Texturparametrisierung eines Dreiecks
- ▶ Zusammenhang zwischen Orts- und Texturkoordinaten lässt sich als eindeutige affine Abbildung beschreiben:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} s \\ t \\ 1 \end{pmatrix}$$

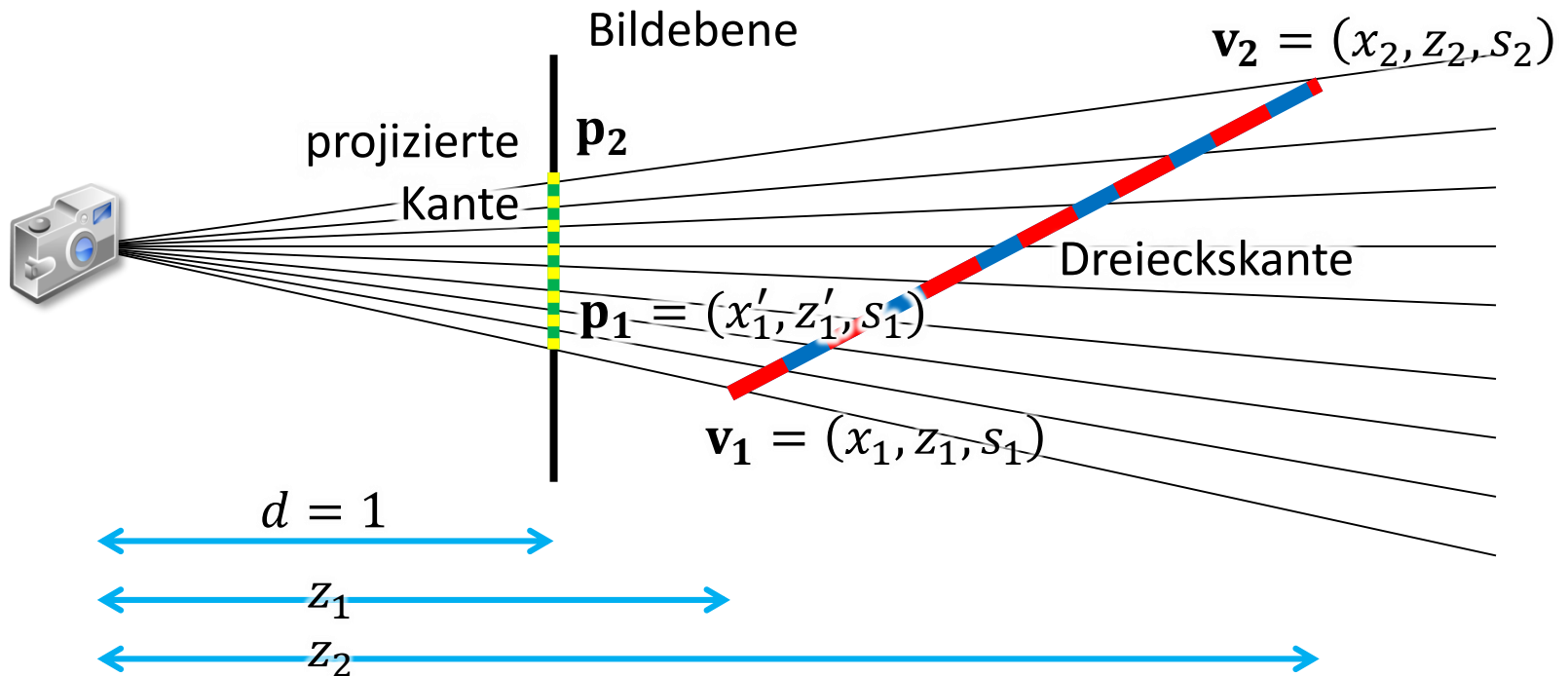
- ▶ Beispiel: Texturkoordinaten (baryzentrische Interpolation = ParallelKoSys)



Perspektivisch-korrekte Attribut-Interpolation



- ▶ beim Rasterisieren finden wir alle Pixel, die das Dreieck bedeckt
 - ▶ wir berechnen zunächst zu jedem Pixel (Punkt auf der Bildebene) die Position in Kamerakoordinaten vor der Projektion
(nur mal so, um herauszufinden was wir eigentl. interpolieren sollten!)
- ▶ ab hier weisen wir Punkten \mathbf{v} , noch eine Texturkoordinate s zu

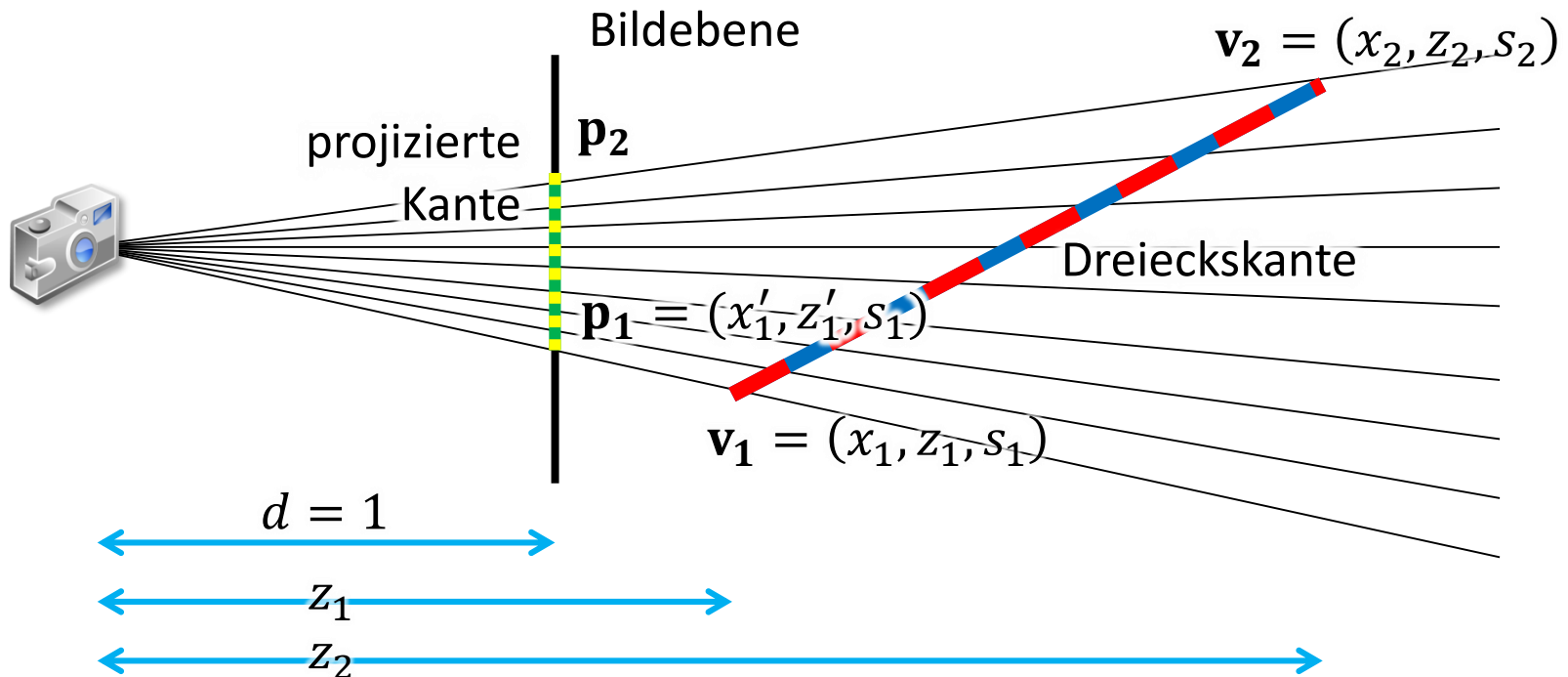


Perspektivisch-korrekte Attribut-Interpolation



Gesucht: Zusammenhang zwischen x' und z

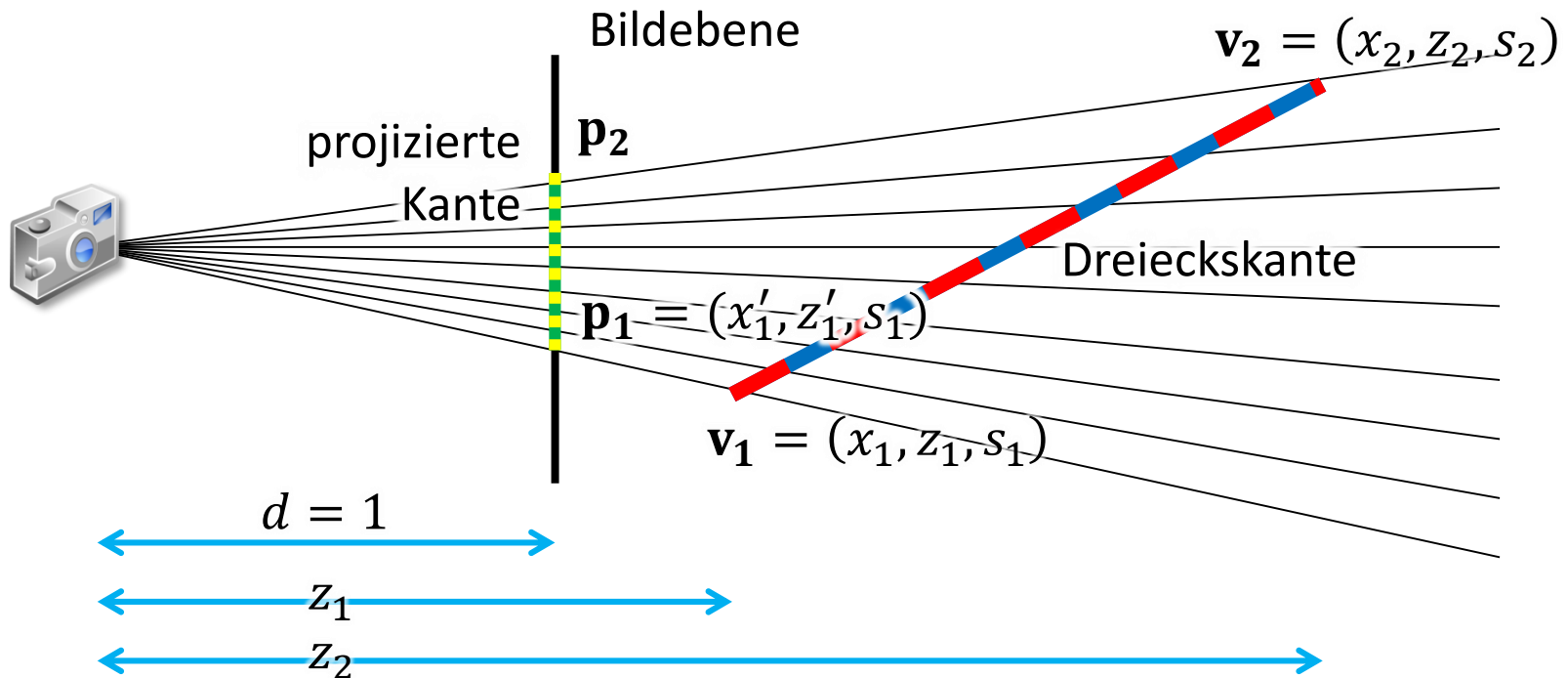
- ▶ für einen Punkt \mathbf{v} gilt $\frac{x}{z} = \frac{x'}{d}$ bzw. hier bestimmen wir die Kameraposition x für eine Pixelposition x' mit: $x = x' \cdot z$
- ▶ zweite Möglichkeit x in Abhängigkeit von z zu beschreiben: in Kamerakoordinaten lassen sich A, B so bestimmen, dass gilt $x = Az + B$



Perspektivisch-korrekte Attribut-Interpolation



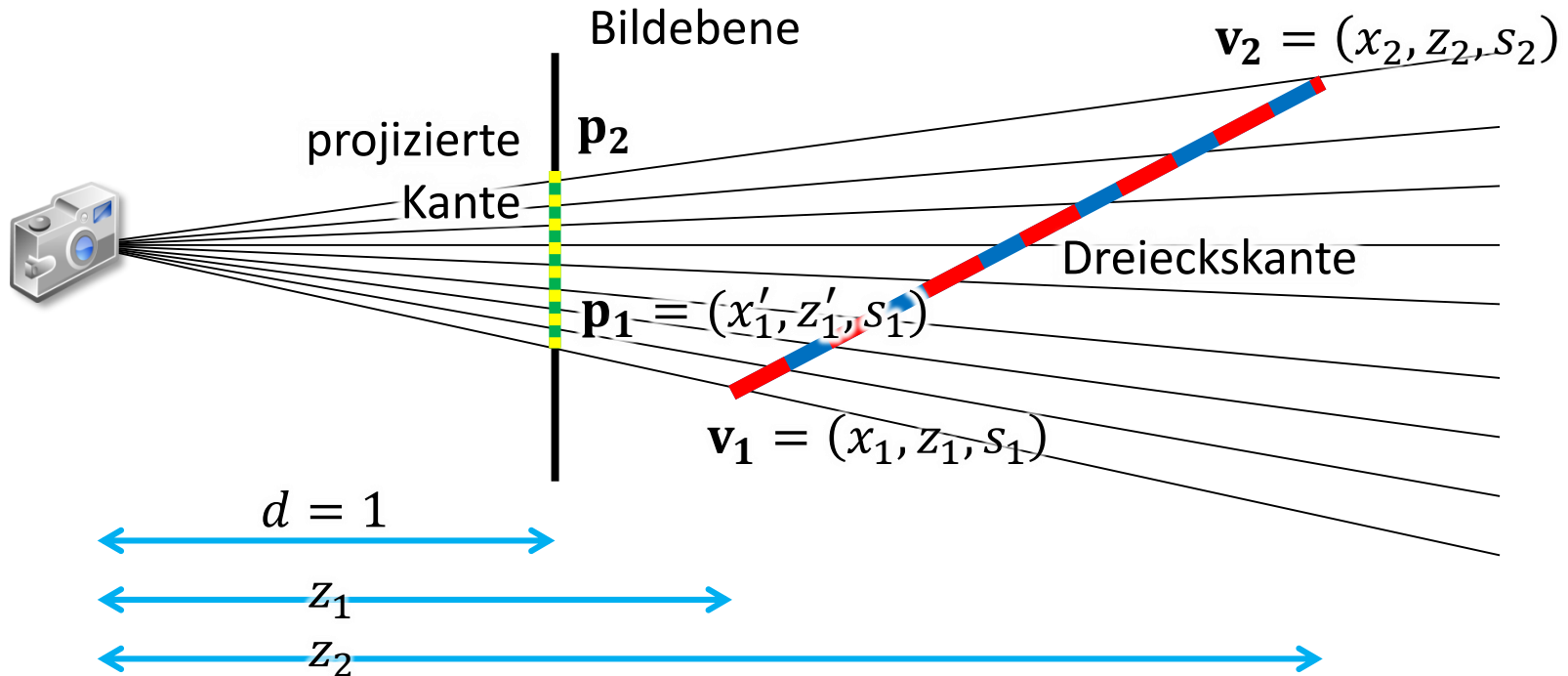
- ▶ Kamerakoordinaten für eine Pixelposition: $x = x' \cdot z$ (1)
- ▶ in Kamerakoordinaten gilt: $x = Az + B$ (2)
- ▶ setzt man (1)=(2), so erkennt man Zusammenhang zwischen x' und z
- ▶ ... und damit, wie man Tiefe z bei der Rasterisierung interpoliert?
 - ▶ Abstand der Fläche zur Kamera für jeden Pixel (Tiefenpuffer)
 - ▶ berechnen der Kamerakoord. $x = x' \cdot z$ (und daraus die Attribute)



Perspektivisch-korrekte Attribut-Interpolation



- ▶ Kamerakoordinaten für eine Pixelposition: $x = x' \cdot z$
- ▶ in Kamerakoordinaten gilt: $x = Az + B$
- ▶ $\Rightarrow x' \cdot z = Az + B \Leftrightarrow z = \frac{B}{x' - A}$
- ▶ offensichtlich ist z **nicht linear** in x' ,
d.h. eine lineare Interpolation im Bildraum ist falsch!



Perspektivisch-korrekte Attribut-Interpolation

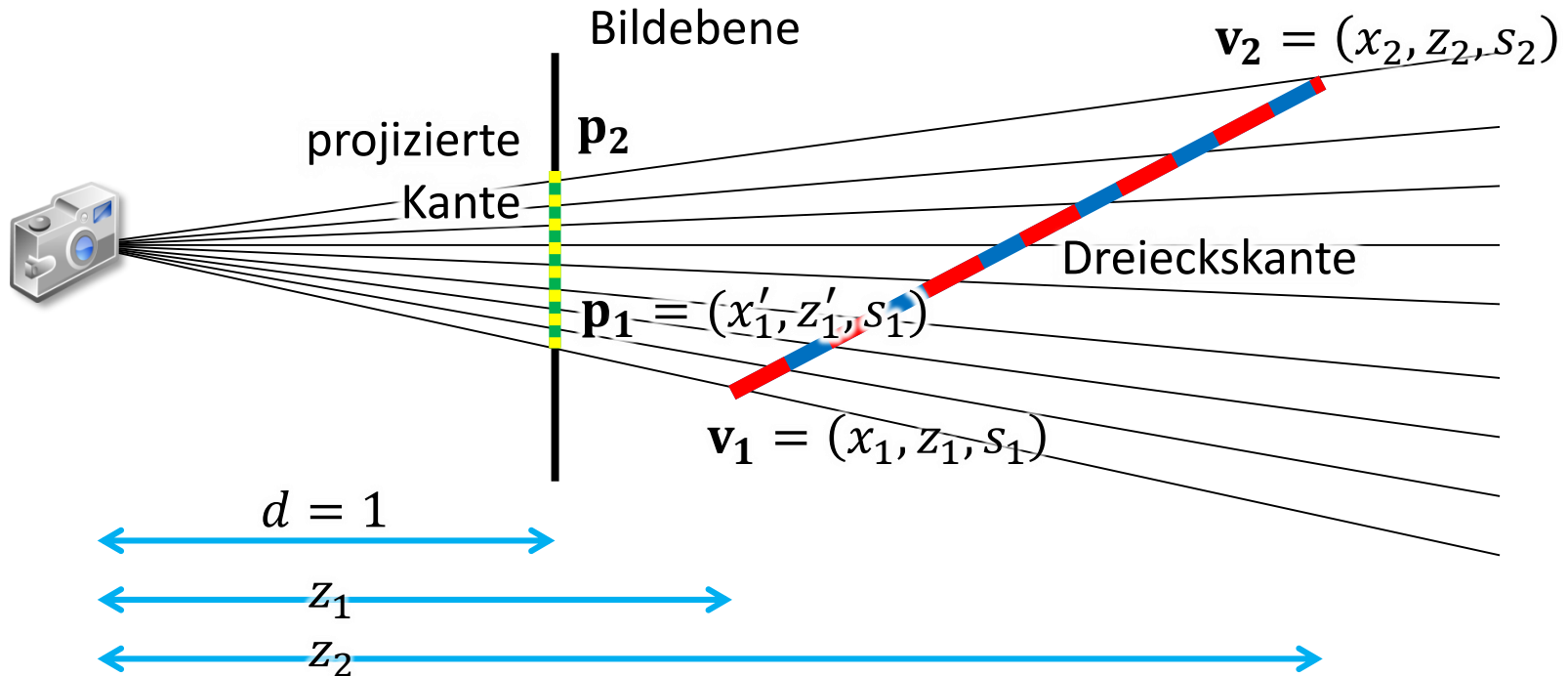


Und jetzt der Trick...

▶ $x' \cdot z = Az + B \Leftrightarrow z = \frac{B}{x' - A}$

▶ aber: $\frac{1}{z} = \frac{1}{B}x' - \frac{A}{B}$

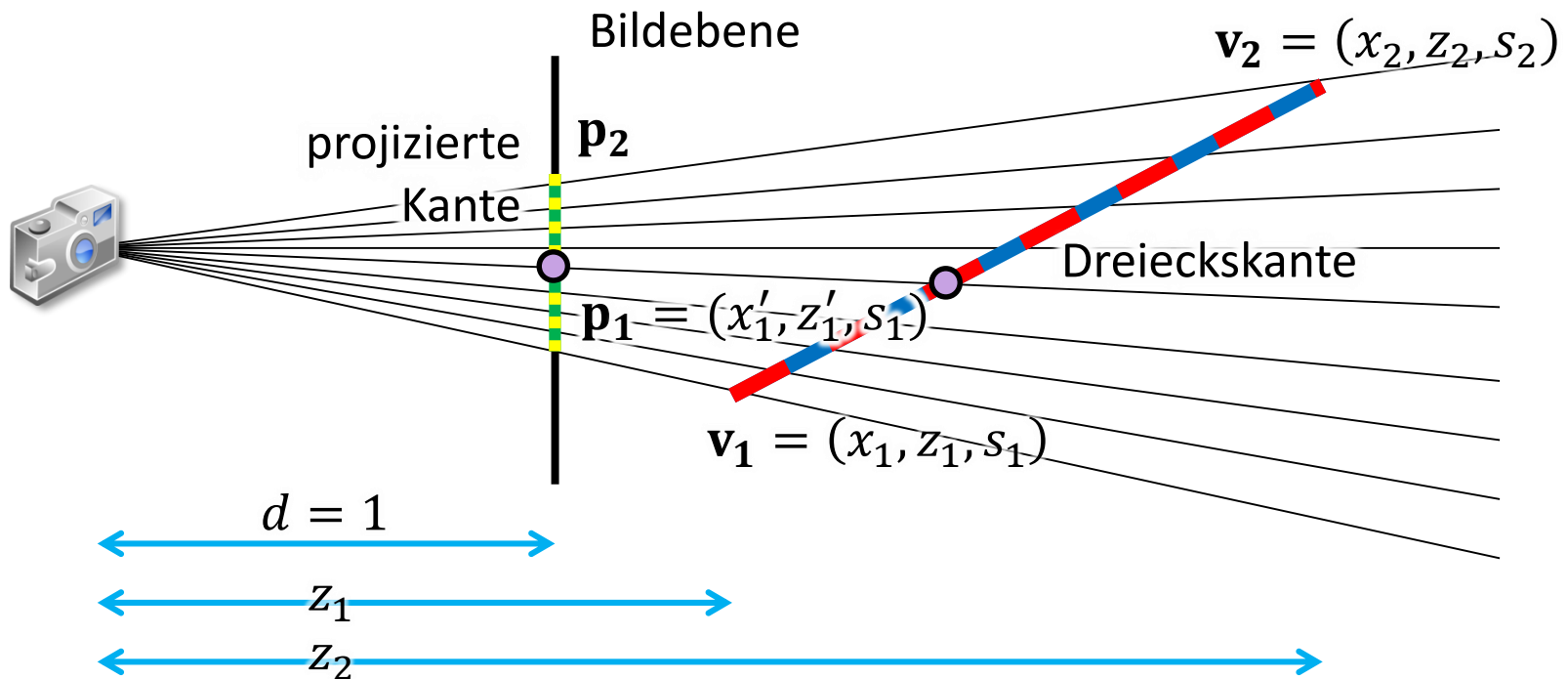
▶ \Rightarrow man darf $1/z$ linear im Bildraum interpolieren und erhält x mit $x = \frac{x'}{1/z}$



Perspektivisch-korrekte Attribut-Interpolation

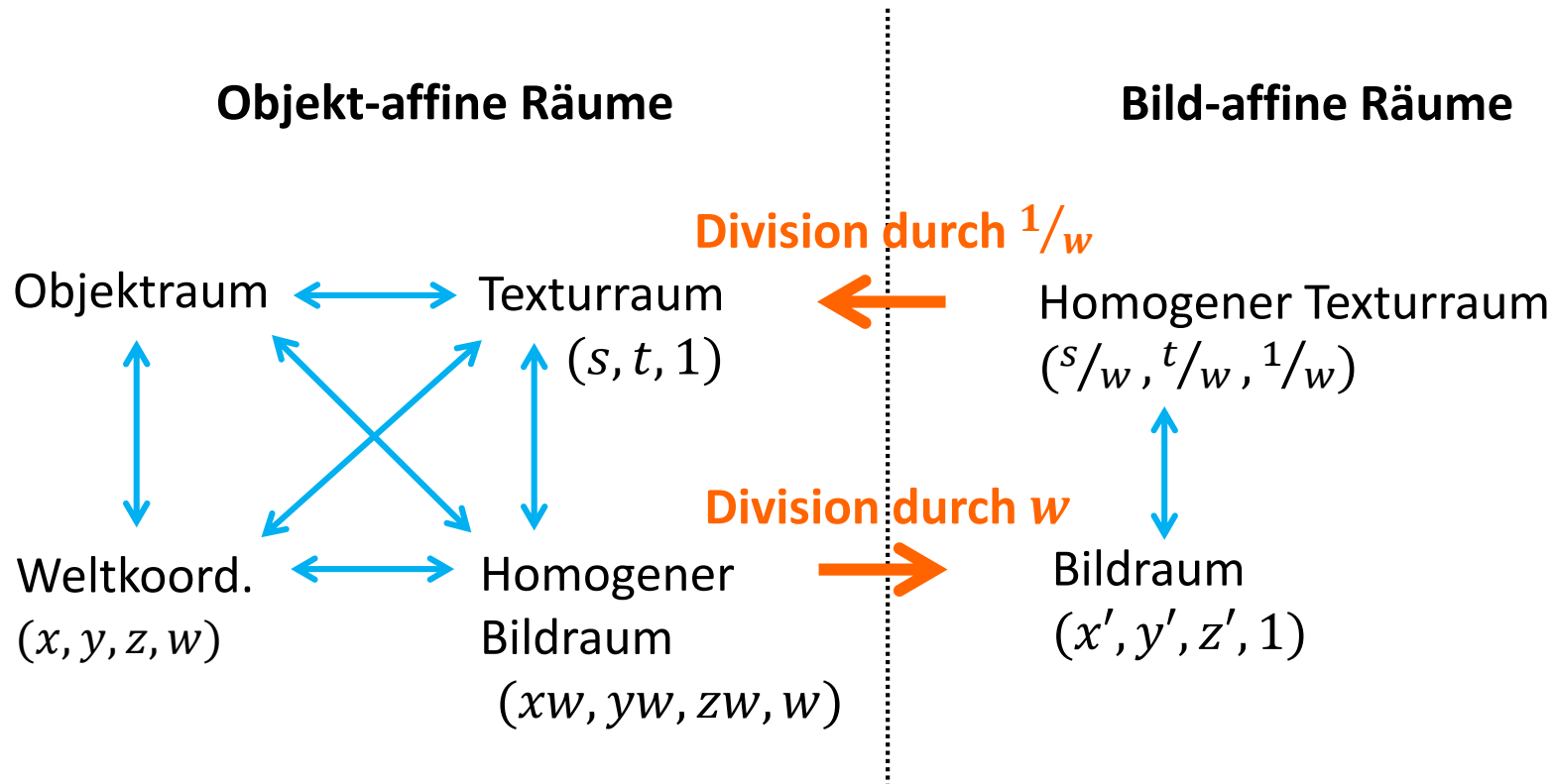


- ▶ \Rightarrow man darf $1/z$ linear im Bildraum interpolieren und erhält x mit $x = \frac{x'}{1/z}$
- ▶ die Attribute könnten dann über baryzentrische Koordinaten interpoliert werden (berechnet mittels x bzgl. der Vertices in Kamerakoord.)
- ▶ ebenso darf man gleich $s/z, t/z, \dots$ (berechnet an den Vertices) im Bildraum interpolieren und erhält durch Division mit dem interpolierten $1/z$ direkt die perspektivisch-korrekt interpolierten Attribute



Perspektivisch-korrekte Interpolation und homogene Koordinaten

- ▶ zur Erinnerung: die 3D-Projektionsmatrix erzeugt „ $w = \pm z$ “
- ▶ lineare Interpolation von s/w , t/w und $1/w$
- ▶ pro Pixel: Division s/w durch $1/w$, ...



„Fallstudie“ Quake (1996, Intel Pentium)



Rasterisierung vs. Raytracing



Rasterisierung

- ▶ sehr effizient, Hardware-Umsetzung ist einfach
- ▶ nur für „Primärstrahlen“: globale Effekte (Schatten, Spiegelungen, indirektes Licht, etc.) nur über spezielle Techniken oder Raytracing
- ▶ Handhabung komplexer Szenen durch räumliche Datenstrukturen
 - ▶ i.W. Entfernen von nicht-sichtbaren und verdeckten Szenenteilen
- ▶ spezielle Techniken, z.B. zur Einschränkung der Schattierungsberechnung auf sichtbare Flächen
- ▶ Anwendung: Echtzeit-Rendering

Raytracing

- ▶ konzeptionell einfaches Verfahren
- ▶ Sekundärstrahlen, komplexe Beleuchtungseffekte sind einfach
- ▶ Handhabung komplexer Szenen durch räumliche Datenstrukturen
- ▶ früher: nur Offline-Rendering, zunehmend interaktives Rendering bzw. ausgewählte Teile im Echtzeit-Rendering