

Computergrafik

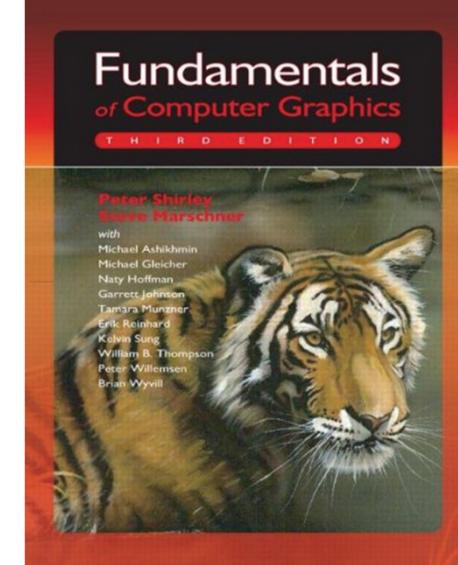
**Vorlesung im Wintersemester 2014/15
Kapitel 6: Rasterisierung, Clipping und
Projektionstransformationen**

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

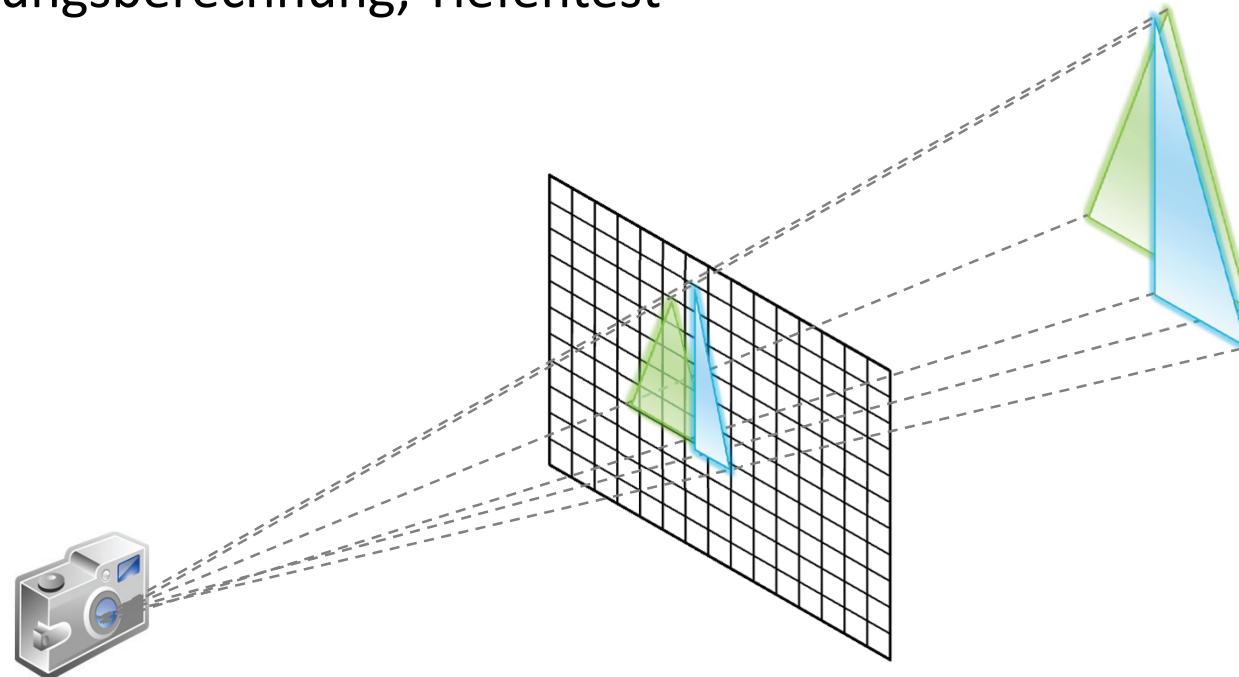


Literatur

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

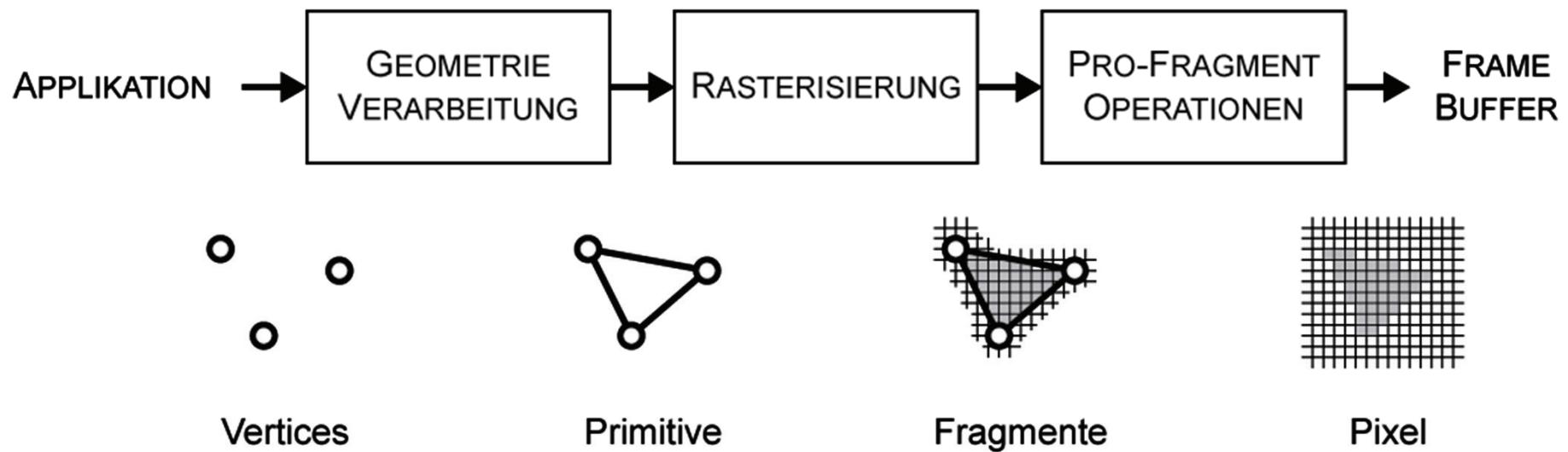


- ▶ basiert meist auf Rasterisierung
 - ▶ Repräsentation von Oberflächen meist durch Dreiecke
 - ▶ Abbilden der Dreiecke auf 2D Bildschirmkoordinaten
 - ▶ Rasterisierung der Dreiecke
 - ▶ Interpolation von Farben, Texturkoordinaten, Tiefenwert, etc.
(an den Eckpunkten gegeben, für jeden Pixel)
 - ▶ Verdeckungsberechnung, Tiefentest



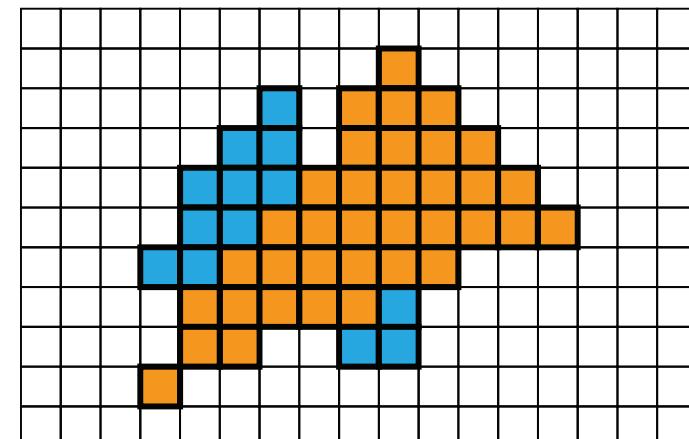
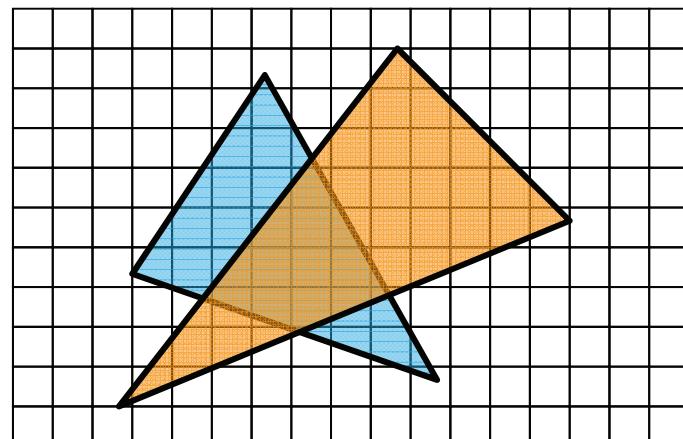
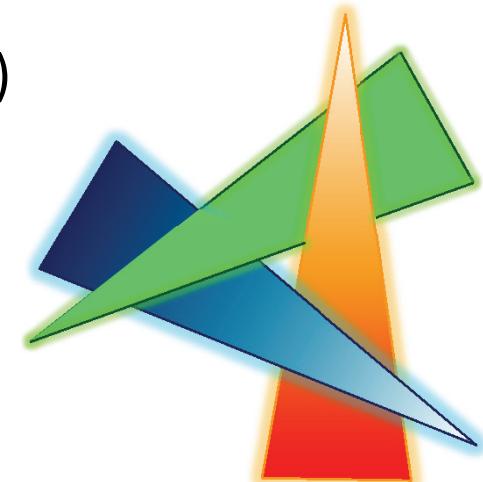
Interaktive/Echtzeit-Computergrafik

- ▶ basiert meist auf Rasterisierung
 - ▶ Repräsentation von Oberflächen meist durch Dreiecke
 - ▶ Abbilden der Dreiecke auf 2D Bildschirmkoordinaten
 - ▶ Rasterisierung der Dreiecke
 - ▶ Interpolation von Farben, Texturkoordinaten, Tiefenwert, etc.
(an den Eckpunkten gegeben, für jeden Pixel)
 - ▶ Verdeckungsberechnung, Tiefentest



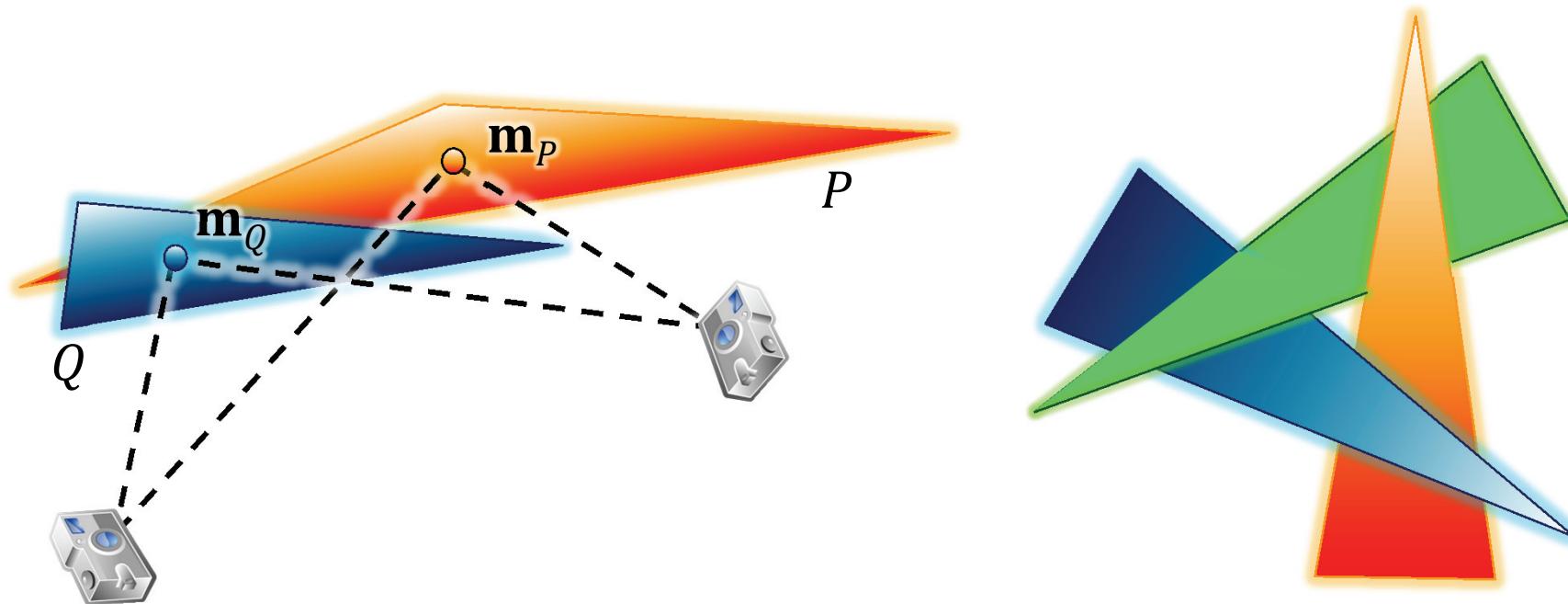
Inhalt

- ▶ Grundlagen für Rasterisierungsverfahren (Basis von OpenGL, Direct3D, ...)
- ▶ Grundlegende Rasterisierungsansätze (in der **Übung**)
- ▶ Sichtbarkeit und Tiefenpuffer (Z-Buffer)
- ▶ Clipping von Linien und Polygonen (in der **Übung**)
 - ▶ Cohen-Sutherland und α -Clipping
 - ▶ Sutherland-Hodgeman
- ▶ Projektive Abbildungen
 - ▶ Clipping in homogenen Koordinaten
 - ▶ perspektivische Interpolation



Sichtbarkeit

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

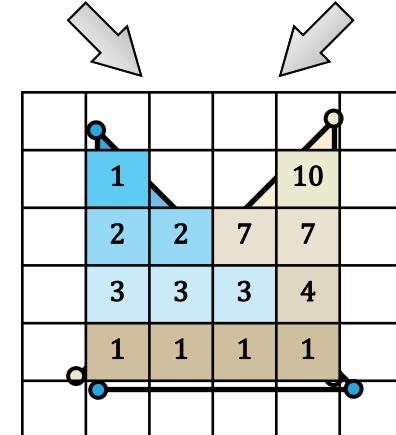
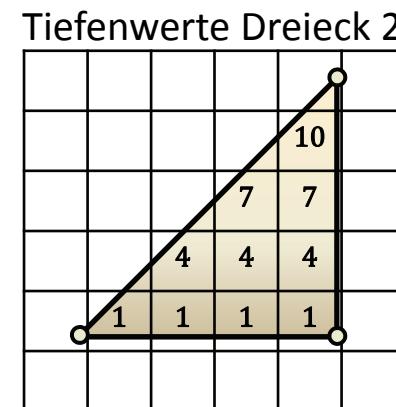
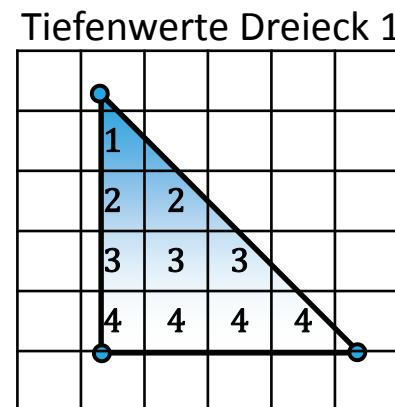


- ▶ wichtig für effiziente Verarbeitung (v.a. in Hardware): Pipelining
 - ▶ Bearbeitung eines Dreiecks nach dem anderen
 - ▶ unabhängig von allen anderen Dreiecken

Tiefenpuffer, Z-Buffering

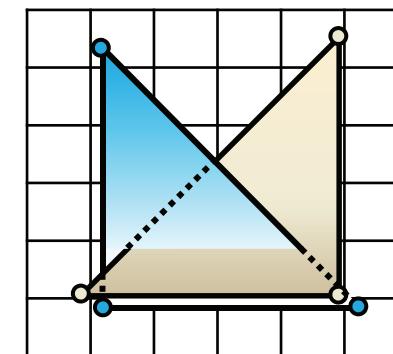
Idee, Prinzip

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



Sichtbarkeit mit Tiefenpuffer

zum Vergleich:
Sichtbarkeit exakt



Tiefenpuffer, Z-Buffering

Rasterisierung mit Tiefenpuffer

- ▶ Initialisierung
 - ▶ fülle Framebuffer mit Hintergrundfarbe
 - ▶ fülle Tiefenpuffer mit maximalem Tiefen-Wert
(entspricht „Far-Plane“, gleich mehr)

- ▶ Rasterisierung

```
// in beliebiger Reihenfolge
foreach Triangle {

    foreach Pixel(x,y) im projizierten Triangle {

        // 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);
        }
    }
}
```

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



Tiefenpuffer als Graustufenbild dargestellt

Nachteile

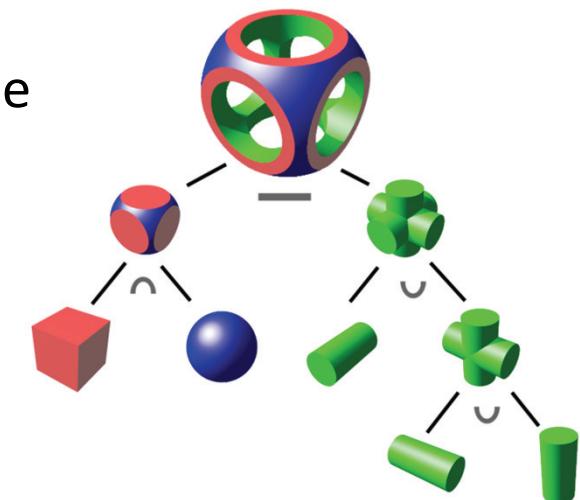
- ▶ Speicherbedarf (heutzutage nicht mehr)
- ▶ begrenzte Genauigkeit und z-Aliasing
- ▶ transparente Flächen können nicht behandelt werden
- ▶ viel unnötiger Aufwand in Szenen mit hoher Tiefenkomplexität
 - ▶ Tiefenkomplexität = Anzahl der Schnitte entlang eines Primärstrahls mit den Oberflächen der Szene
 - ▶ unnötiger Aufwand, weil verdeckte Flächen rasterisiert werden

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

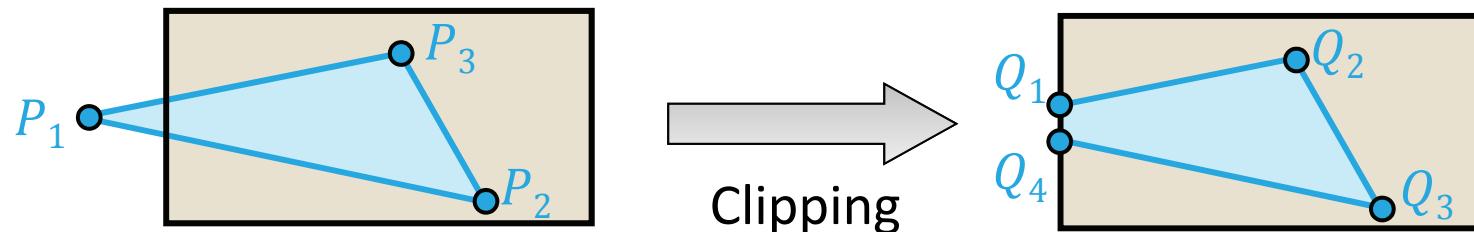
Clipping

- ▶ Clipping: Abschneiden von Linien/Polygon-Teilen die außerhalb des Bildschirms liegen (oder allg. außerhalb eines gewünschten Bereichs)
- ▶ warum ist Clipping wichtig?
 - ▶ Effizienz: zeichne nichts außerhalb des Bildschirms/View Frustums, keine Objekte hinter der Kamera (in 3D)
 - ▶ (vermeide problematische Fälle bei 3D Projektionen)
- ▶ wichtig nicht nur für Rasterisierung, beispielsweise auch für Constructive Solid Geometry, Boolesche Operationen in 2D und 3D
- ▶ unterschiedliche Ansätze
 - ▶ Clipping on-the-fly während der Rasterisierung
 - ▶ z.B. Laufvariable für x -Achse beschränken auf $0 \leq x \leq$ Breite
 - ▶ besser: analytisches Clipping (dieses Kapitel)



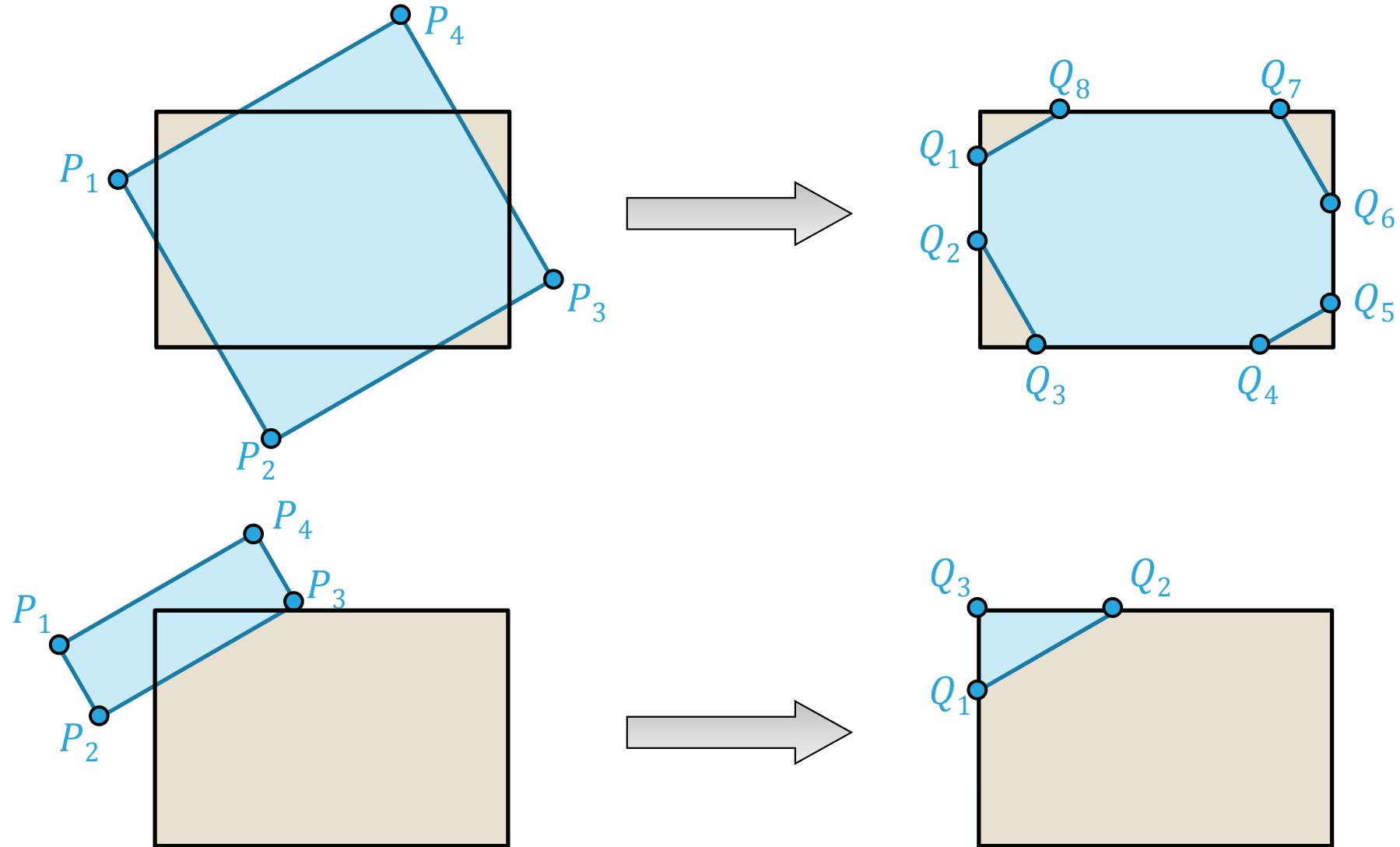
Clipping von Polygonen

- ▶ naives Clipping in den bisherigen Rasterisierungsverfahren
 - ▶ Brute Force Ansatz: teste nur Pixel am Bildschirm
 - ▶ Scanline Rasterisierung: zeichne nur Scanlines innerhalb des Bildes
 - ▶ aber: fixe Setup-Kosten pro Dreieck/pro Scanline
- ▶ warum ist Clipping **unbedingt notwendig?**
 - ▶ **Vermeidung problematischer Fälle bei Projektionen** (gleich mehr)
- ▶ geg. Polygon P_1, P_2, \dots, P_n
- ▶ ges. Polygon Q_1, Q_2, \dots, Q_n nach Clipping gegen ein Rechteck



Clipping von Polygonen

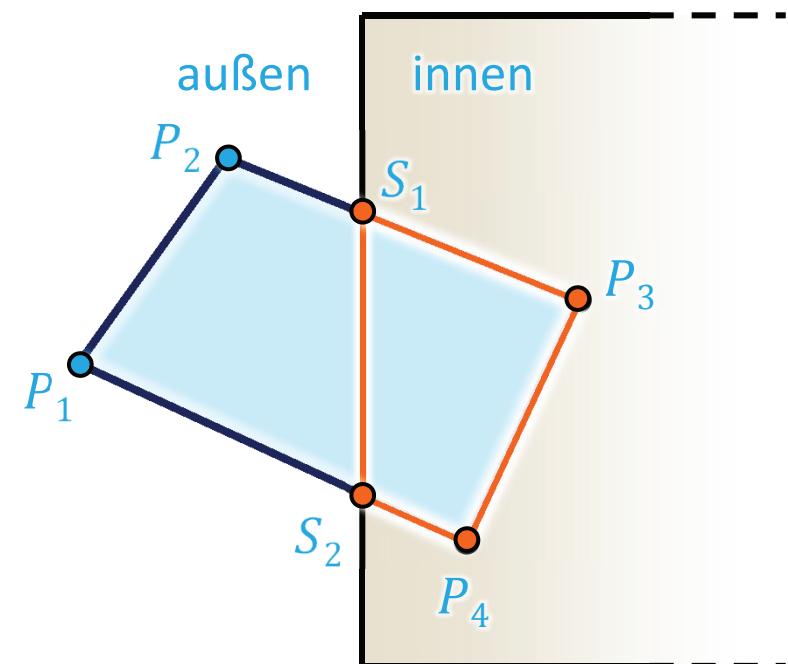
- wir brauchen mehr als nur Liniensegmente, die durch Clipping der Polygonkanten entstehen → Algorithmus von Sutherland-Hodgeman



Sutherland-Hodgeman Polygon Clipping

Algorithmus für (konvexe) Viewports in 2D

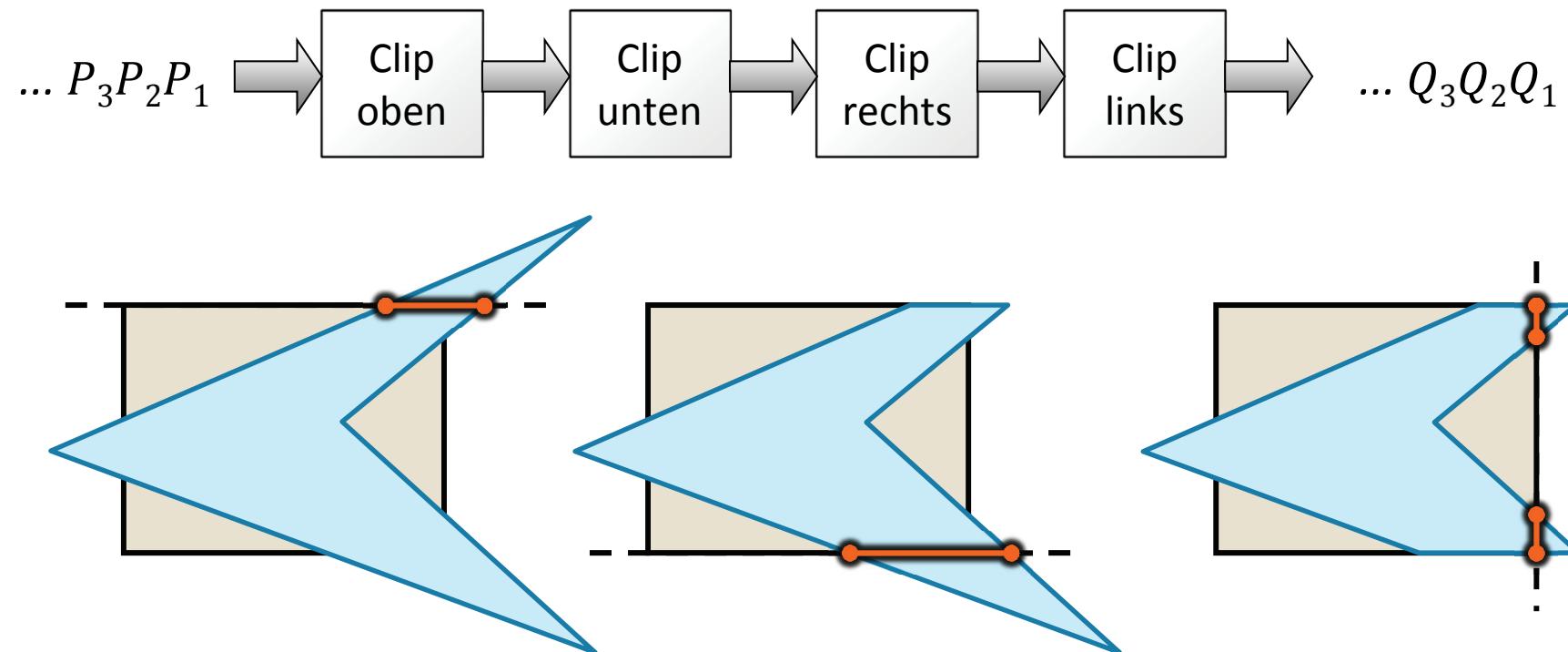
- ▶ Clipping gegen eine Kante nach der Anderen
- ▶ für jede Kante
 - ▶ Eingabe: Liste der Vertizes des Polygons
 - ▶ Ausgabe: Liste von Vertizes des resultierenden Polygons
 - ▶ Beispiel:
 - ▶ Eingabe: P_1, P_2, P_3, P_4
 - ▶ Ausgabe: S_1, P_3, P_4, S_2



Sutherland-Hodgeman Polygon Clipping

Algorithmus für (konvexe) Viewports in 2D

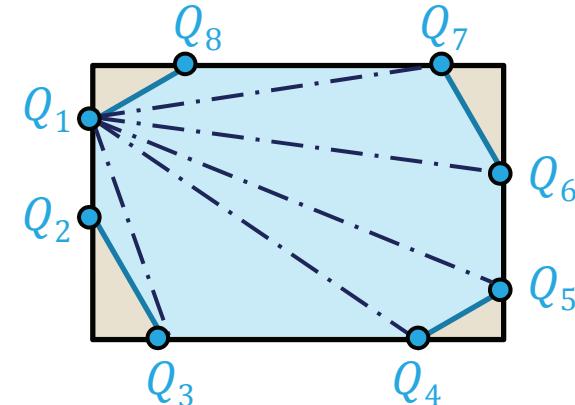
- ▶ Clipping gegen eine Kante nach der Anderen ist einfacher und effizienter
 - ▶ nach jeder Kante erhält man ein geschlossenes Polygon
 - ▶ erlaubt Pipelining (wichtig für Hardware-Umsetzungen)



Sutherland-Hodgeman Polygon Clipping

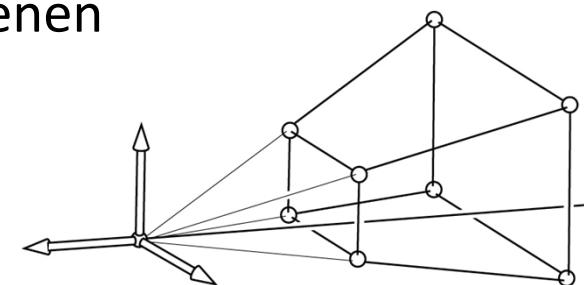
Clipping mit Attributen

- ▶ Vertex-Attribute (z.B. Texturkoordinaten oder Farben) werden mit und für die Schnittpunkte berechnet
- ▶ ist das Eingabepolygon konvex, dann auch das Ausgabepolygon
 - ▶ kann bei Bedarf einfach in Dreiecke zerlegt werden:
 Q_1, Q_i, Q_{i+1} mit $1 < i < n$



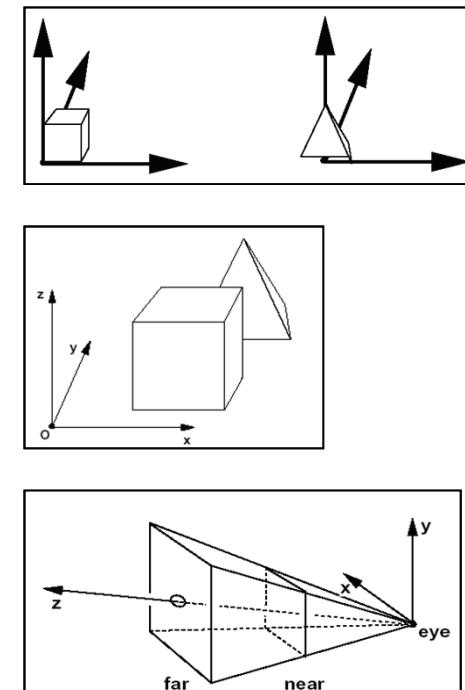
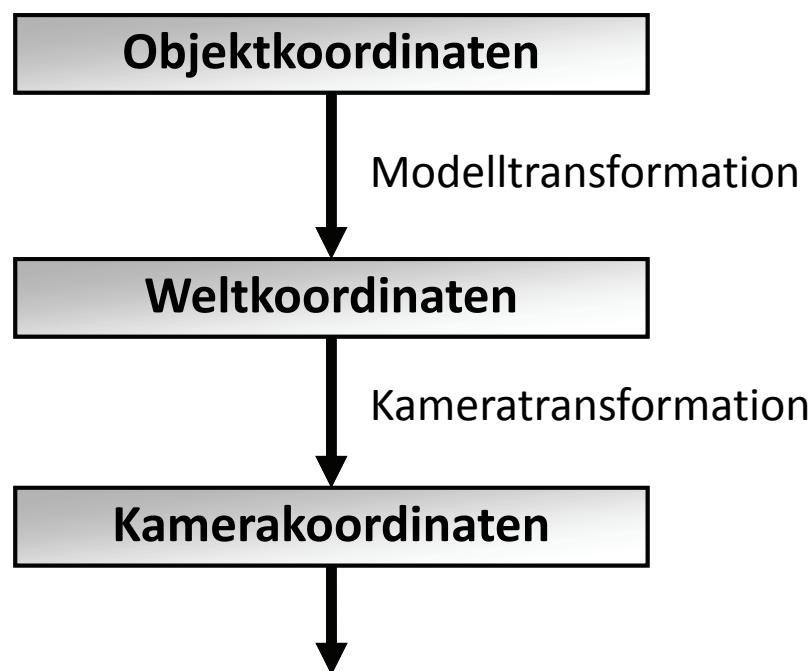
Clipping in 3D

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



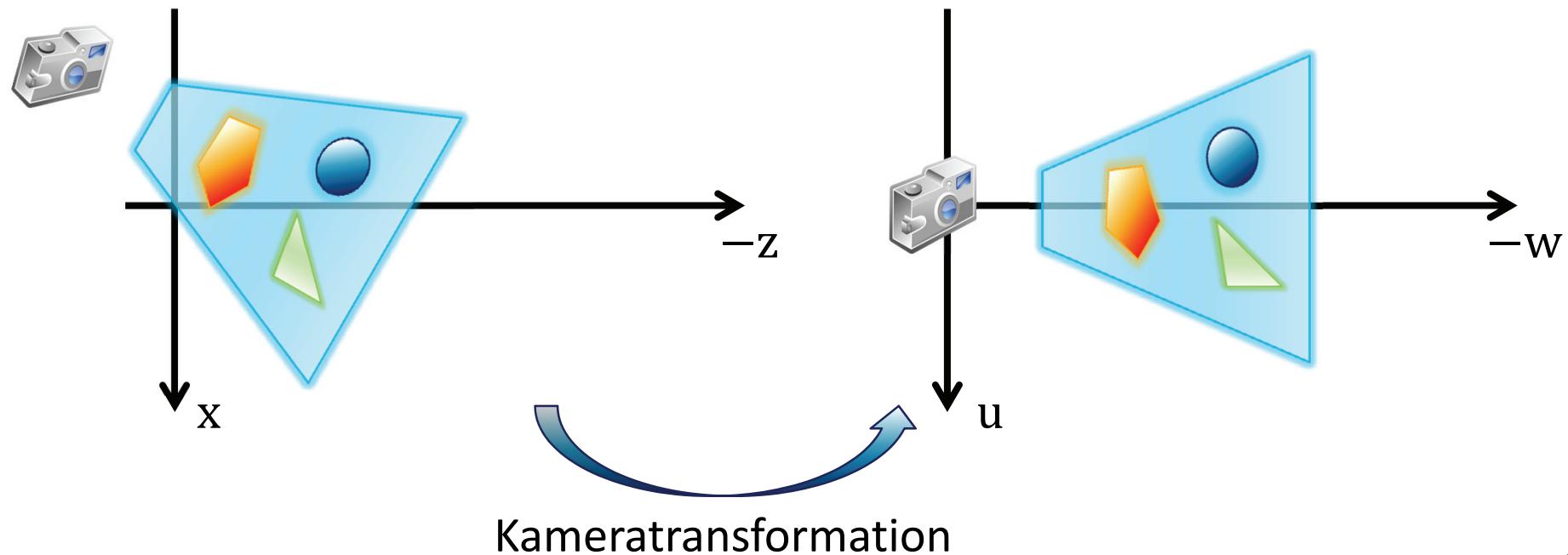
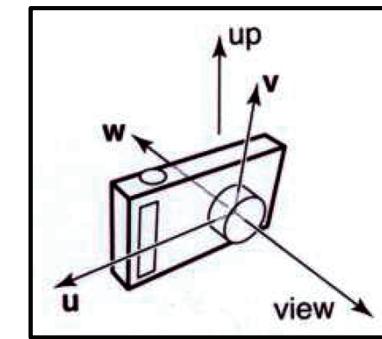
Koordinatensysteme in der CG

- ▶ ...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

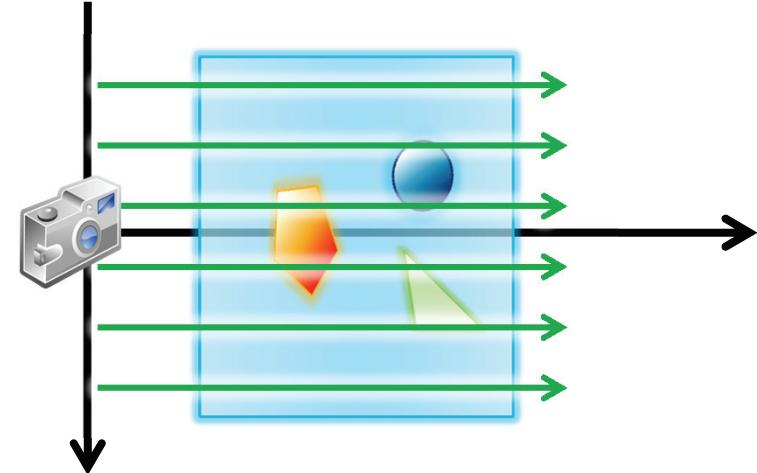
- ▶ virtuelle Kamera definiert durch
 - ▶ Position e und (negative) Blickrichtung w
 - ▶ „Up-Vektor“ up
 $\Rightarrow u = up \times w$ und $v = w \times u$
 - ▶ zuerst Translation um $-e$, dann Transformation in das Kamera-Koordinatensystem dessen Basis von u, v, w gebildet wird



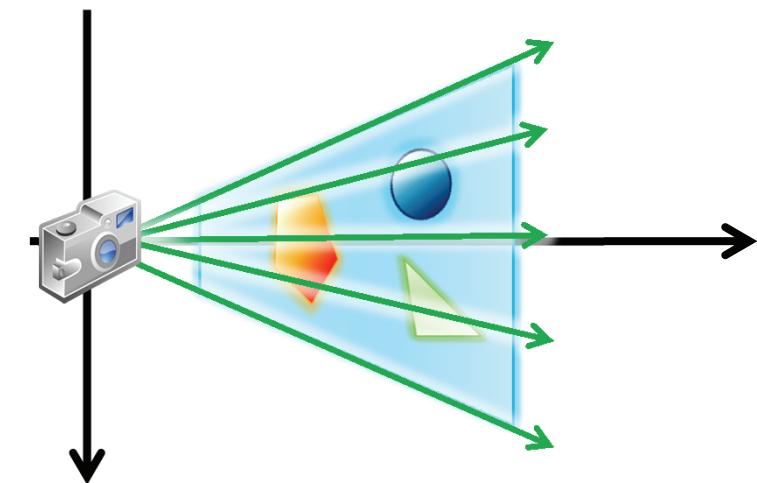
Projektion

- ▶ wir betrachten nur zwei Arten von Projektionen

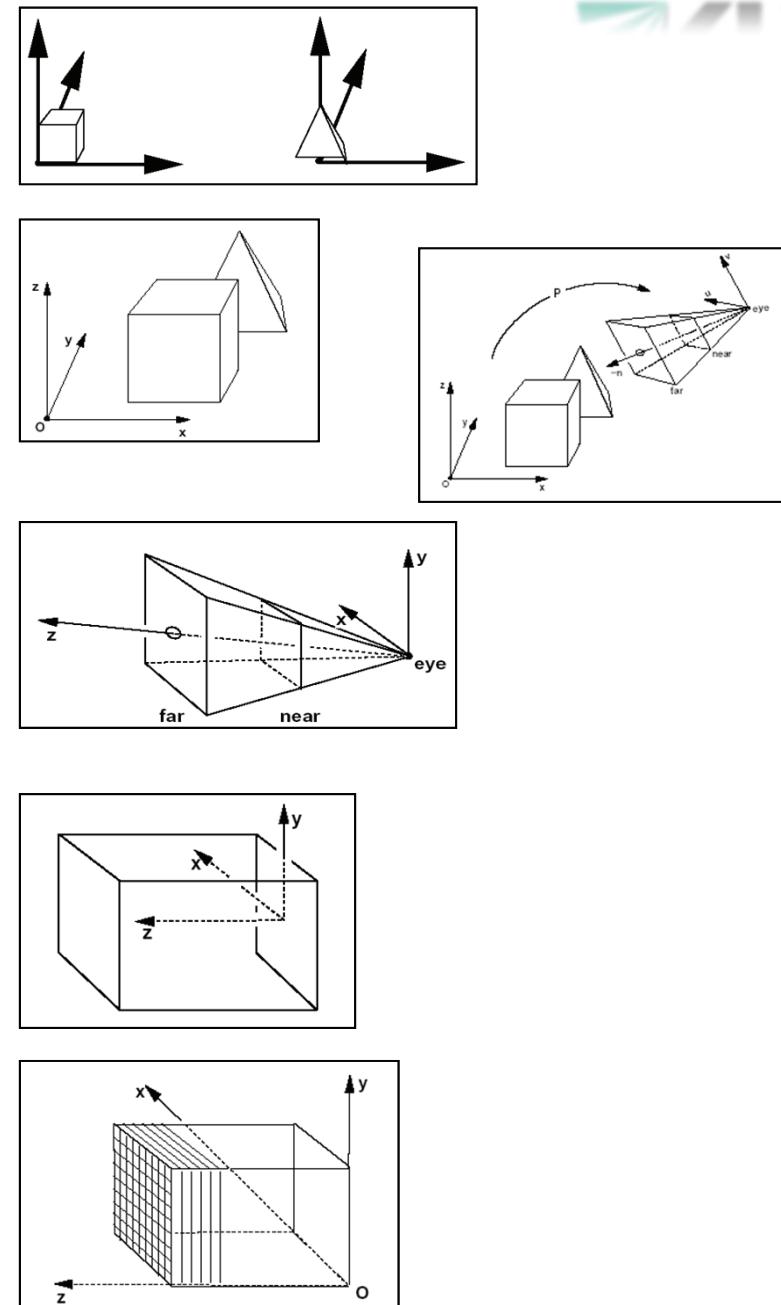
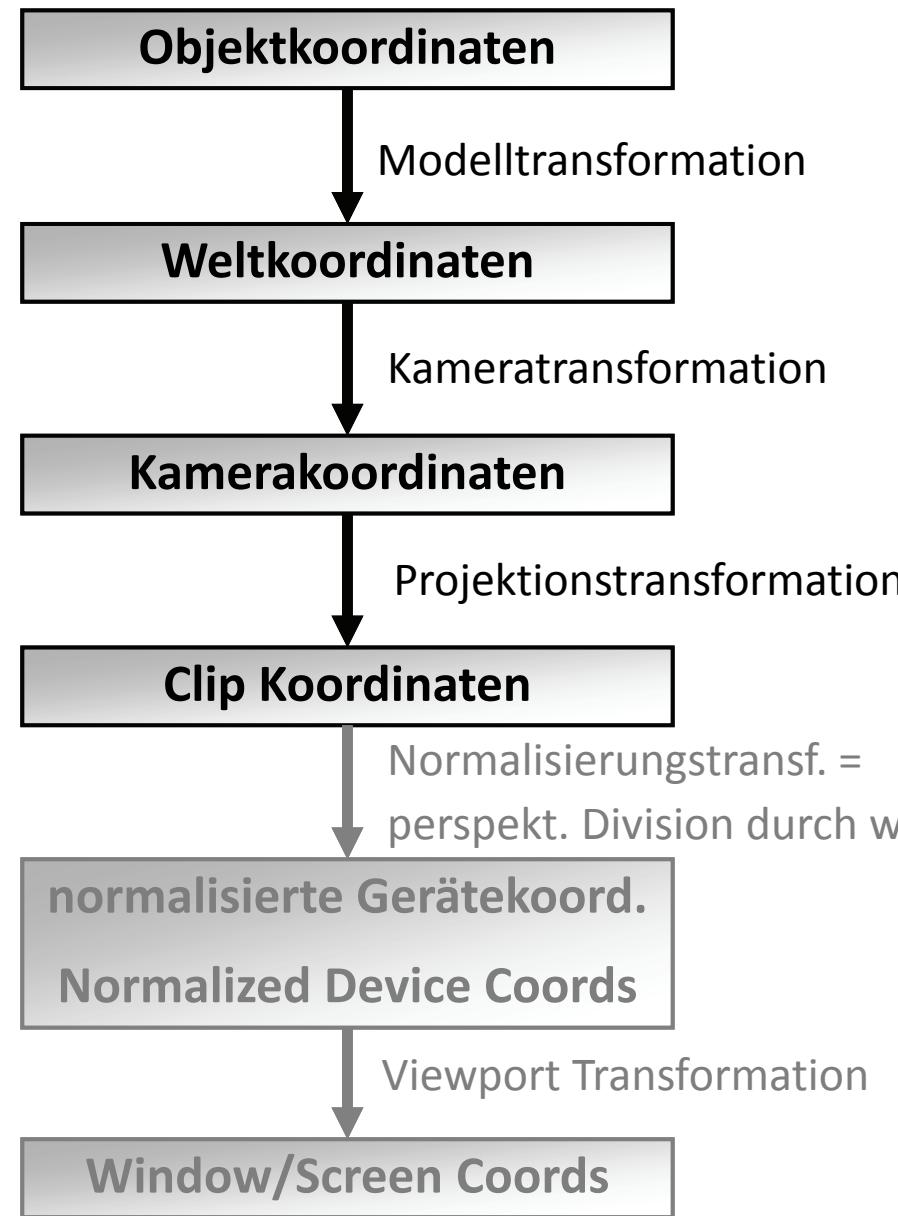
- ▶ orthographische Kamera:
parallele **Sichtstrahlen**
(Anm. wir betrachten nur senkrechte
Projektionen auf die Bildebene, es
gibt auch schiefe Projektionen
mit parallelen Sichtstrahlen)



- ▶ perspektivische Kamera:
Sichtstrahlen ausgehend von
einem Projektionszentrum

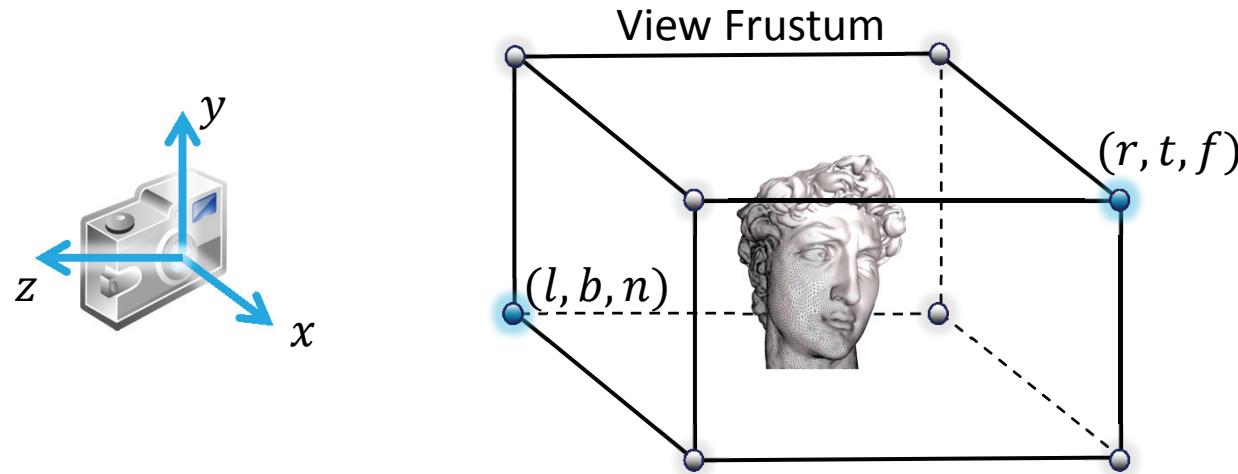


(Noch mehr) Koordinatensysteme in der CG



Orthographische Kamera/Projektion

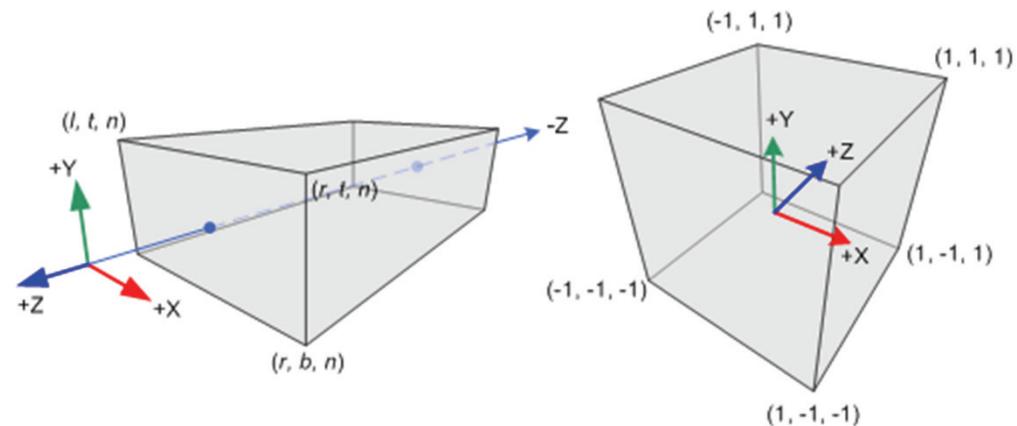
- ▶ übliche Konvention bei Rasterisierungs-APIs:
 - ▶ x -Achse zeigt nach rechts, y -Achse nach oben, Blick in neg. z -Richtung
- ▶ das Sichtvolumen (View Frustum) ist ein Quader $[l; r] \times [b; t] \times [n; f]$
 - ▶ left l , right r , bottom b , top t , near n und far f
 - ▶ Achtung: $n > f$
- ▶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

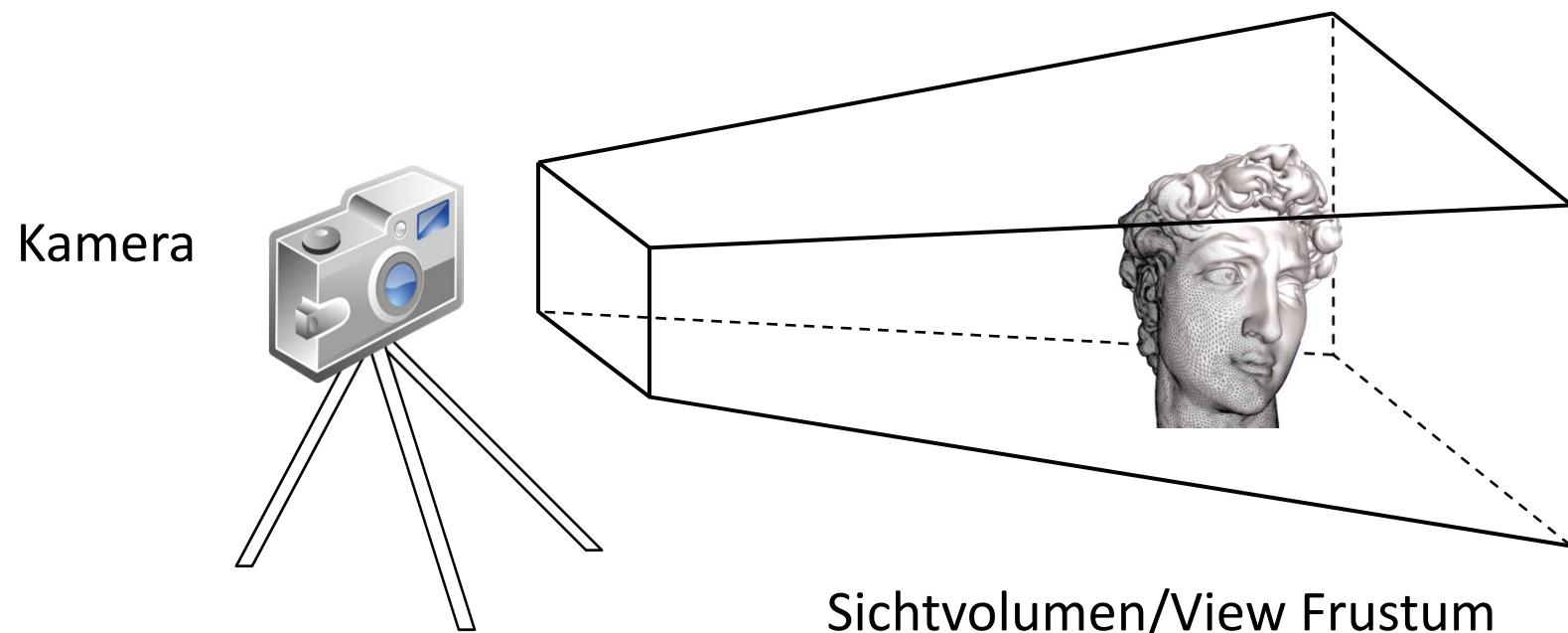
$$\mathbf{M}_{orth} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



- ▶ 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

- ▶ Modellierung: bewege und skaliere Modell/Objekt
- ▶ Blickpunkt: Position und Orientierung der Kamera
- ▶ Projektion: Zoom-Objektiv einstellen
- ▶ Viewport: Ausschnittsvergrößerung
- ▶ auch hier suchen wir ebenfalls 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 neg. z-Achse, Blick in Richtung der pos. z-Achse
- ▶ Projektion auf die Bildebene mit $z = 0$

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

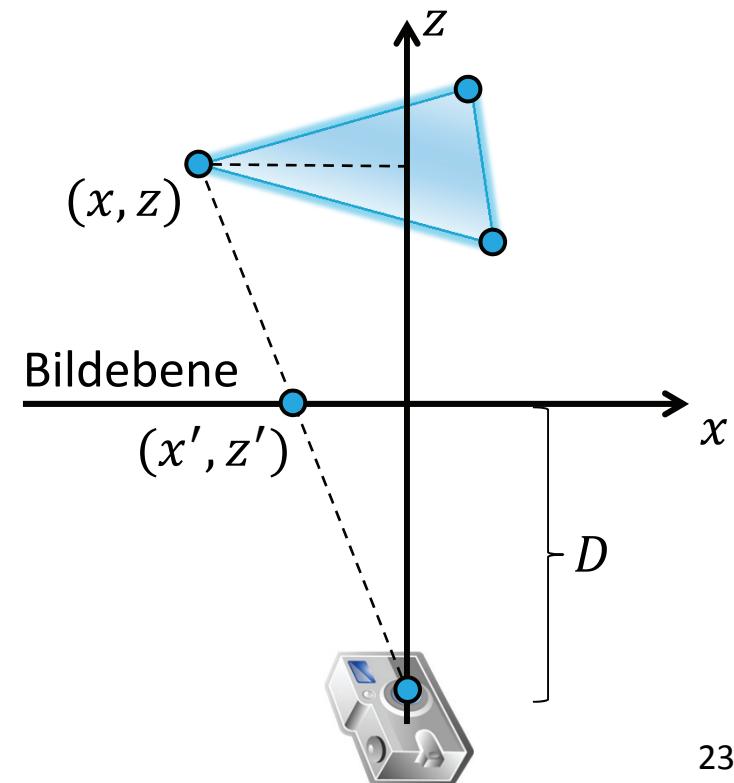
- ▶ nach einer „herkömmlichen“ Projektion wäre $z' = 0$

- ▶ wir heben z' als Tiefe auf, um später die Sichtbarkeit zu bestimmen

- ▶ Berechnung analog zu x'
(ein Grund: eine Matrix für Alles)

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

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



Perspektivische Projektion in 2D

- ▶ Konvention in diesem Beispiel: Projektionszentrum liegt bei $(0, -D)$ auf der neg. z-Achse, Blick in Richtung der pos. z-Achse
- ▶ Projektion auf die Bildebene mit $z = 0$

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

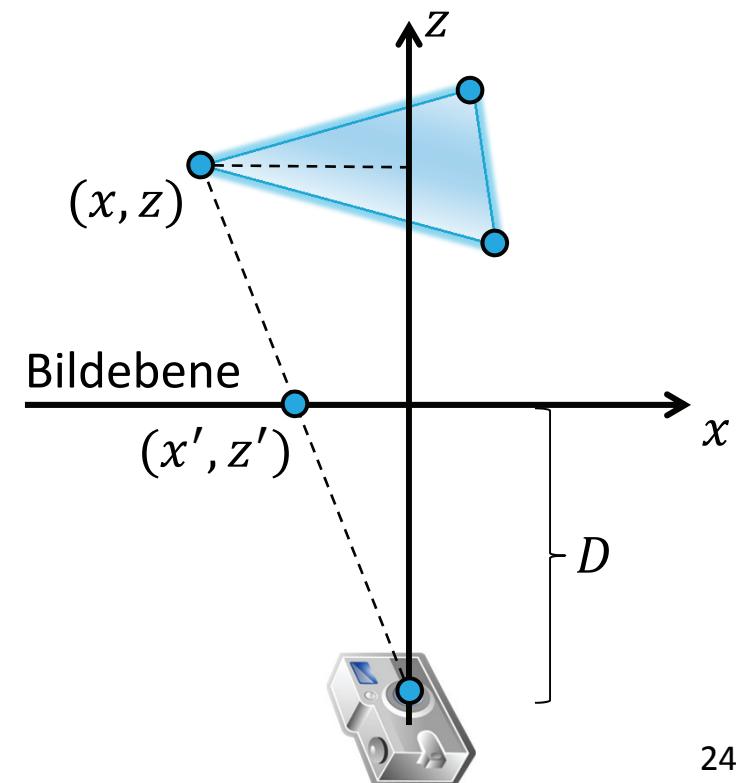
- ▶ nach einer „herkömmlichen“ Projektion wäre $z' = 0$

- ▶ wir heben z' als Tiefe auf, um später die Sichtbarkeit zu bestimmen

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

- ▶ warum Tiefe z' und nicht einfach z für Z-Buffering?

- ▶ Genauigkeit bei der Repräsentation der Tiefenwerte
 - ▶ perspektivisch-korrekte Attributinterpolation

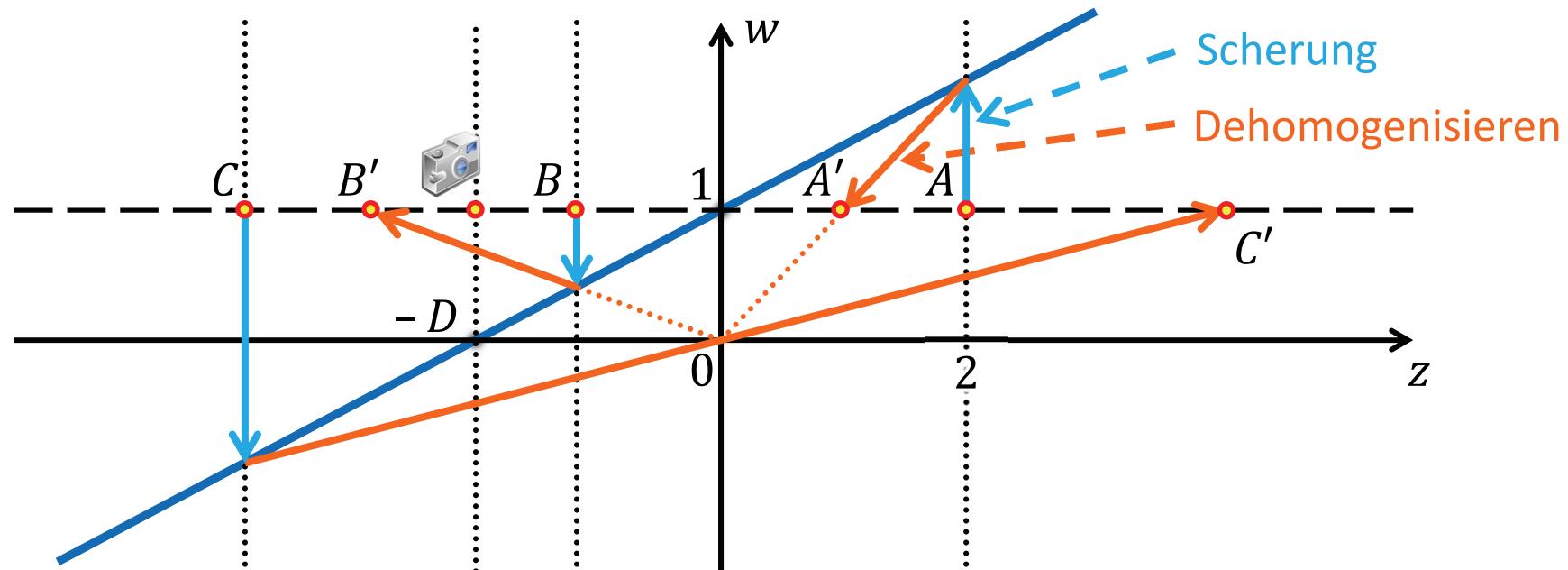


Perspektivische Projektion in 2D

Betrachtung des Tiefenwerts

- ▶ wie hängt z und $z' = \frac{z}{z/D+1}$ zusammen?
- ▶ Scherung in w -Richtung mit $D = 2$ und anschließende Dehomogenisierung

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

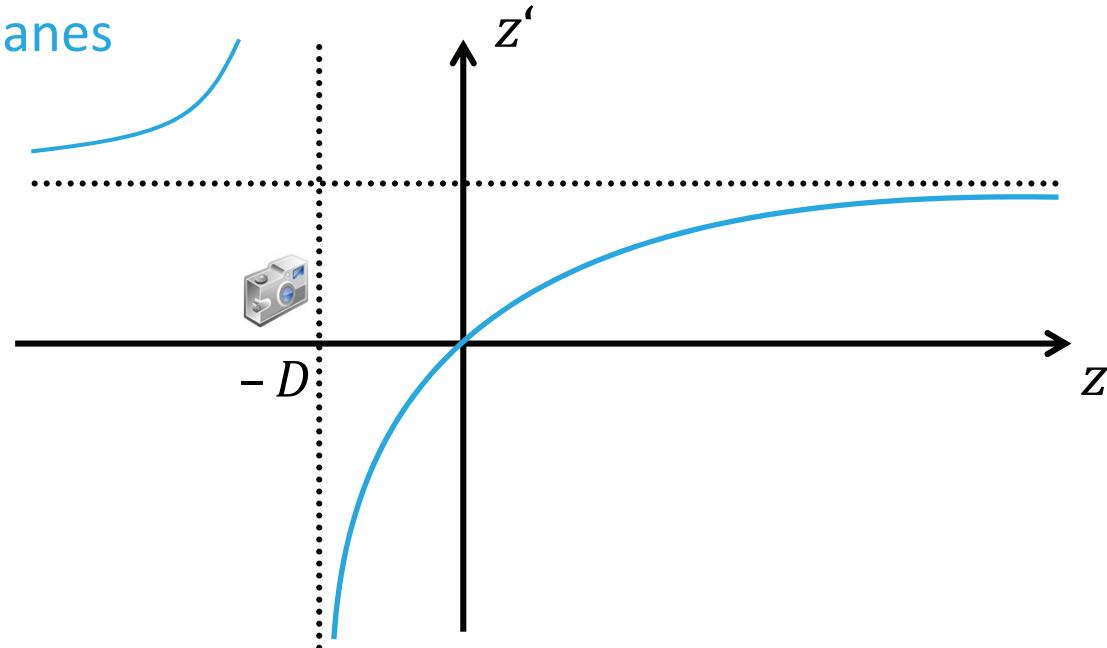


- A: vor der Kamera, $z > 0$ → kleinerer, positiver Tiefenwert
 B: vor der Kamera, aber $z < 0$ → Abbildung auf negative Werte
 C: hinter der Kamera, $z < -D$ → Abbildung auf positive Werte
 Kameraposition → $-\infty$

Perspektivische Projektion in 2D

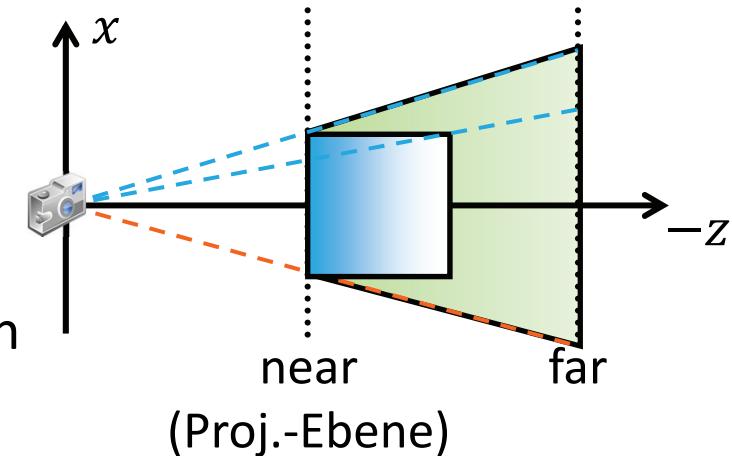
Betrachtung des Tiefenwerts

- ▶ zur Erinnerung: wir möchten z' verwenden, um herauszufinden welches Primitiv vor welchem liegt, also z.B. für den Tiefentest
- ▶ Nicht-Linearität der z -Transformation ...
 - ▶ **gut: höhere Genauigkeit für nahe Objekte** (= der zweite Grund für die Transformation mit einer Matrix), dafür wenig Präzision für entfernte Objekte (große Werte)
 - ▶ positive Werte für Objekte hinter der Kamera (zu kleine z)
- ▶ Lösung: beschränke Tiefenbereich des View Frustum mit **Near und Far (Clipping) Planes**



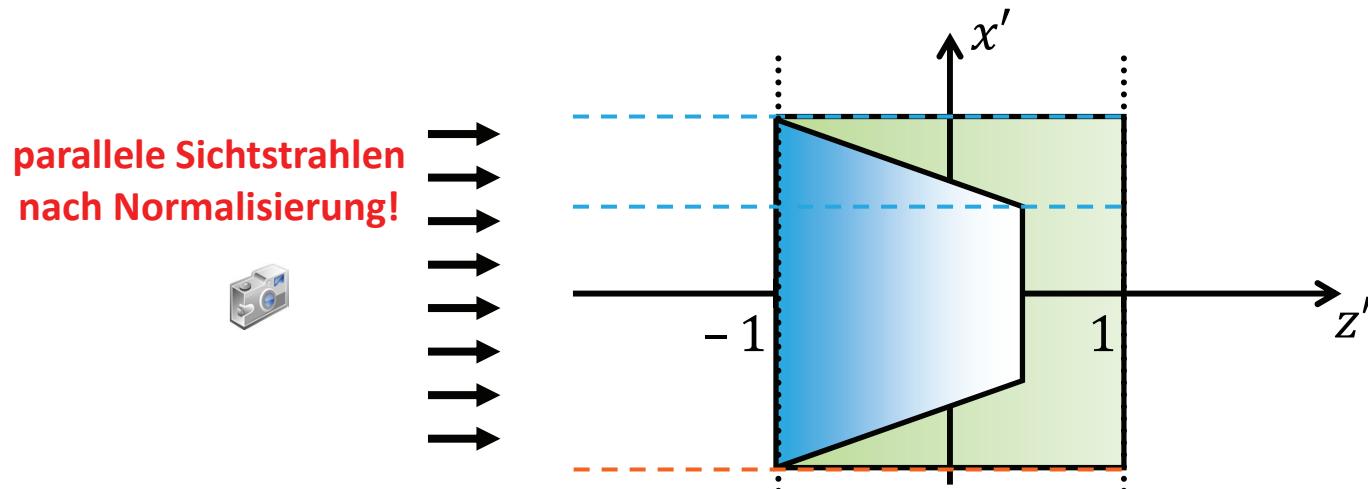
Perspektivische Projektion in 2D

- 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



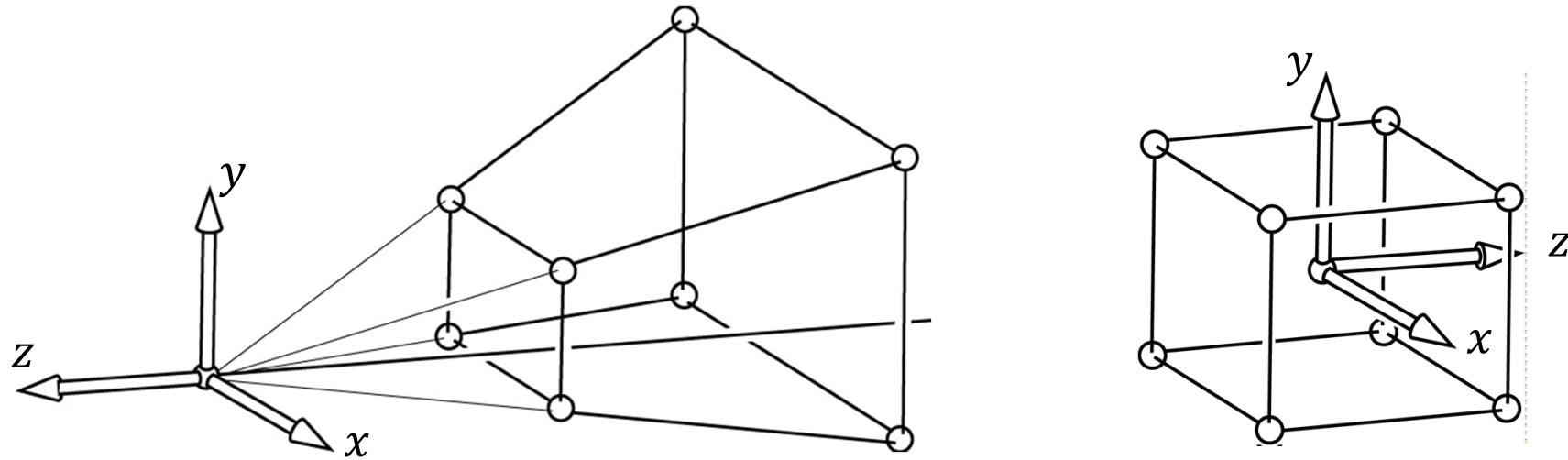
- nach der Projektionstransformation haben wir noch homogene Koordinaten!

- Dehomogenisieren danach nennt man „Normalisierungstransformation“
 - Bildschirmkoordinaten (x', y') erhalten wir danach durch Weglassen der Tiefe (=orthographische Projektion)



Perspektivische Projektion in 3D

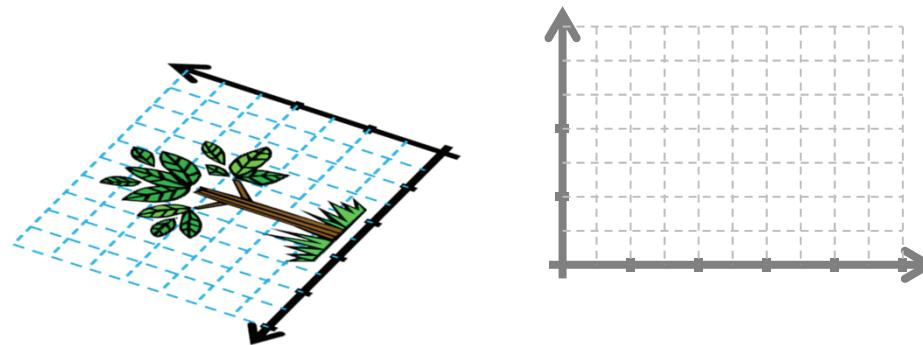
- ▶ Dehomogenisieren nach der Projektionstransformation ist die „Normalisierungstransformation“
- ▶ das View Frustum wird durch *beide* Schritte auf den Einheitswürfel $[-1; 1]^3$ (in kartesischen Koordinaten) abgebildet



- ▶ eine oft gebrauchte Konvention (z.B. auch bei OpenGL) ist
 - ▶ Kamera befindet sich im Ursprung
 - ▶ Blickrichtung entlang der negativen z -Achse

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 skalierungs invariant
 - ▶ $5 \times 3 = 4 \times 4 - 1$ Elemente

Bestimmung der Projektionsmatrix (in 2D)

2D Projektionstransformation: $3 \times 3 - 1$ Unbekannte

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

- Kameraposition nach $-\infty$

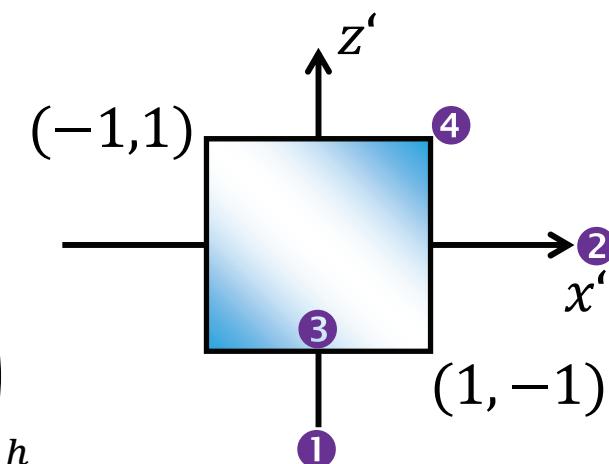
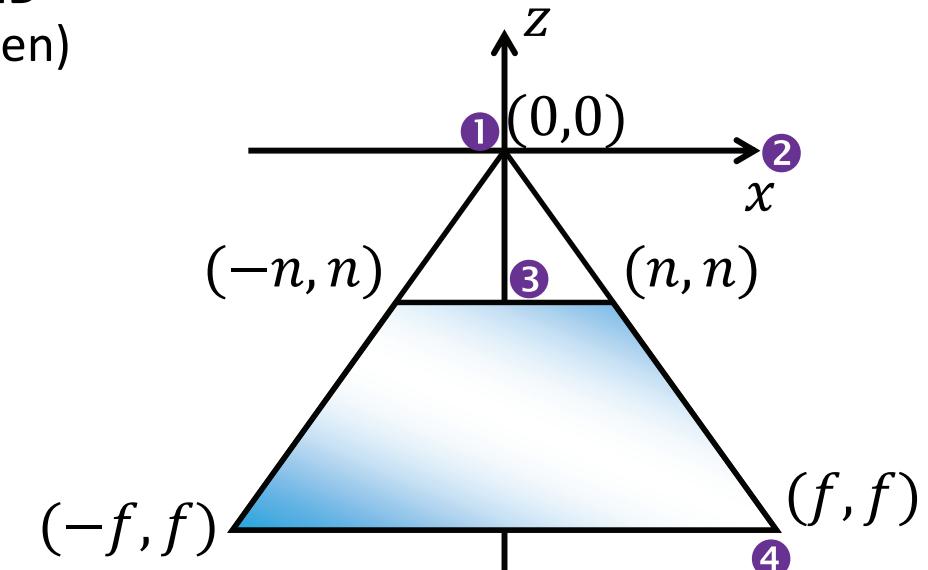
$$① \quad \mathbf{M} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}_h = k_1 \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix}_h$$

- x -Richtung beibehalten

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

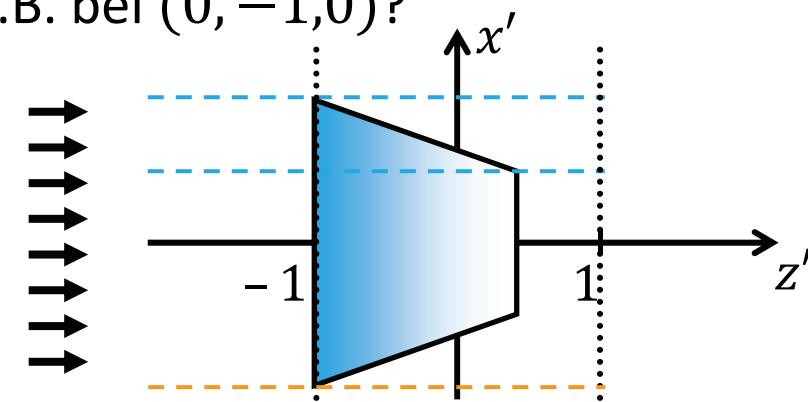
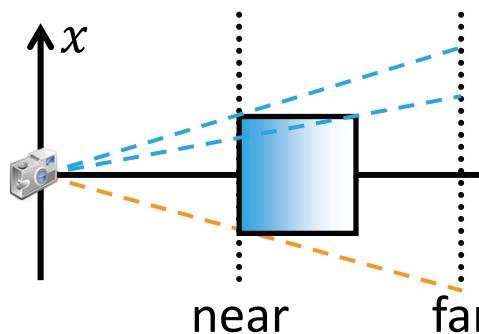
- zwei weitere Punkte (Achtung: nicht 3 Punkte auf einer Geraden wählen)

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



Bestimmung der Projektionsmatrix (in 2D)

- die Sichtstrahlen von der Kamera sind nach der Projektionstransformation und Dehomogenisieren parallel – warum „schneiden“ sich parallele Linien im Unendlichen, also z.B. bei $(0, -1, 0)$?



- implizite Darstellung zweier paralleler Linien im affinen Raum

$$Ax + By + C_1 = 0 \text{ bzw. } Ax + By + C_2 = 0$$

- offensichtlich: wenn $C_1 \neq C_2$, dann gibt es keinen Schnittpunkt
- Darstellung zweier paralleler Linien in homogenen Koordinaten

$$Ax + By + C_1w = 0 \text{ bzw. } Ax + By + C_2w = 0$$

- Schnittpunkt für $w = 0$ bei $(-B, A, 0) = (B, -A, 0)$



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:

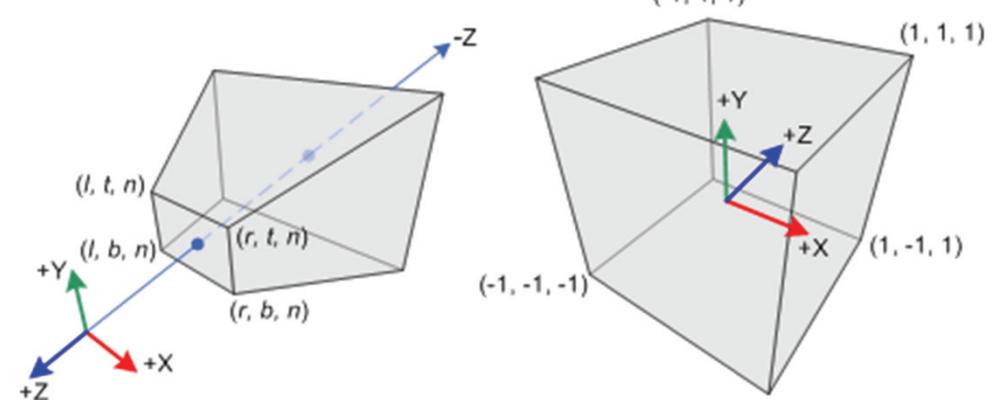
$$\mathbf{M}_{2D} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 1 & 0 \end{pmatrix}$$

- ▶ Projektionstransformation in 3D (mit beliebig asymmetrischem Frustum)

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

Achtung mit dieser Konvention:

$n = 1$ bedeutet die Near-Plane liegt im Abstand 1 entlang der negativen z-Achse (Far-Plane analog)



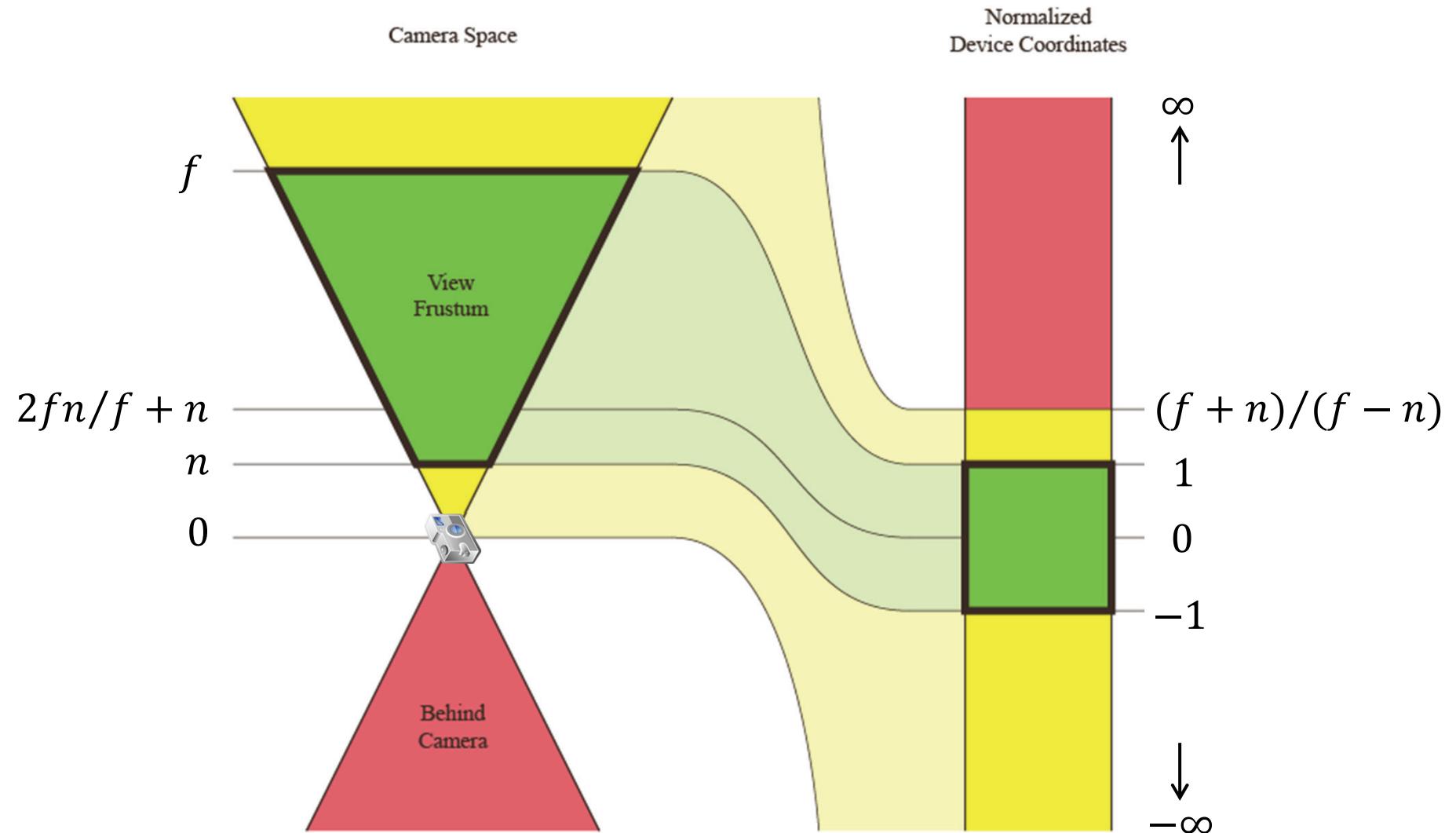
http://www.songho.ca/opengl/gl_projectionmatrix.html

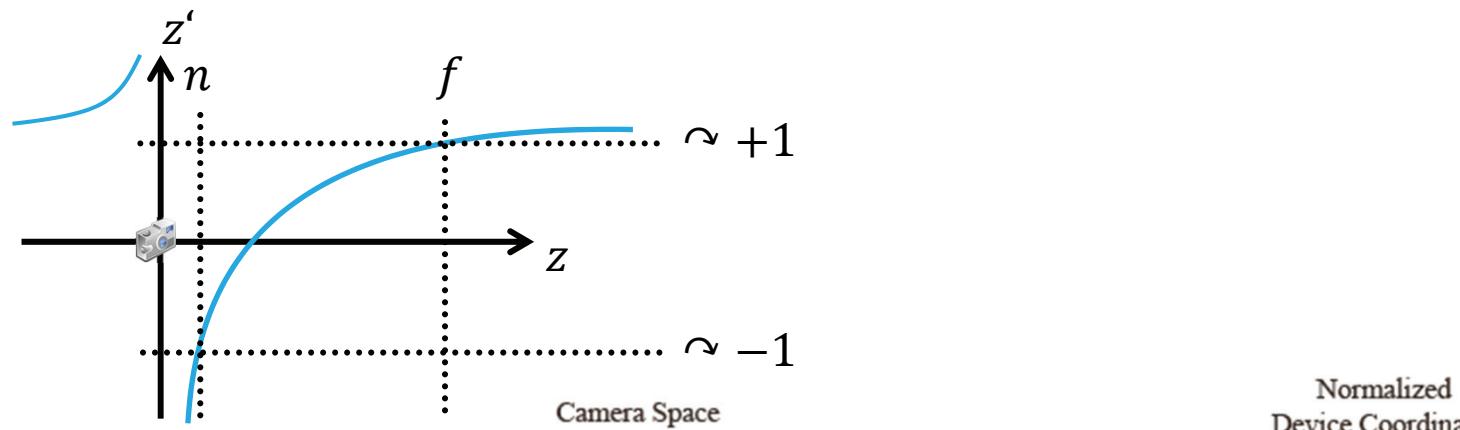
<http://www.scratchapixel.com/lessons/3d-advanced-lessons/perspective-and-orthographic-projection-matrix/opengl-perspective-projection-matrix/>

Perspektivische Projektion in 2D

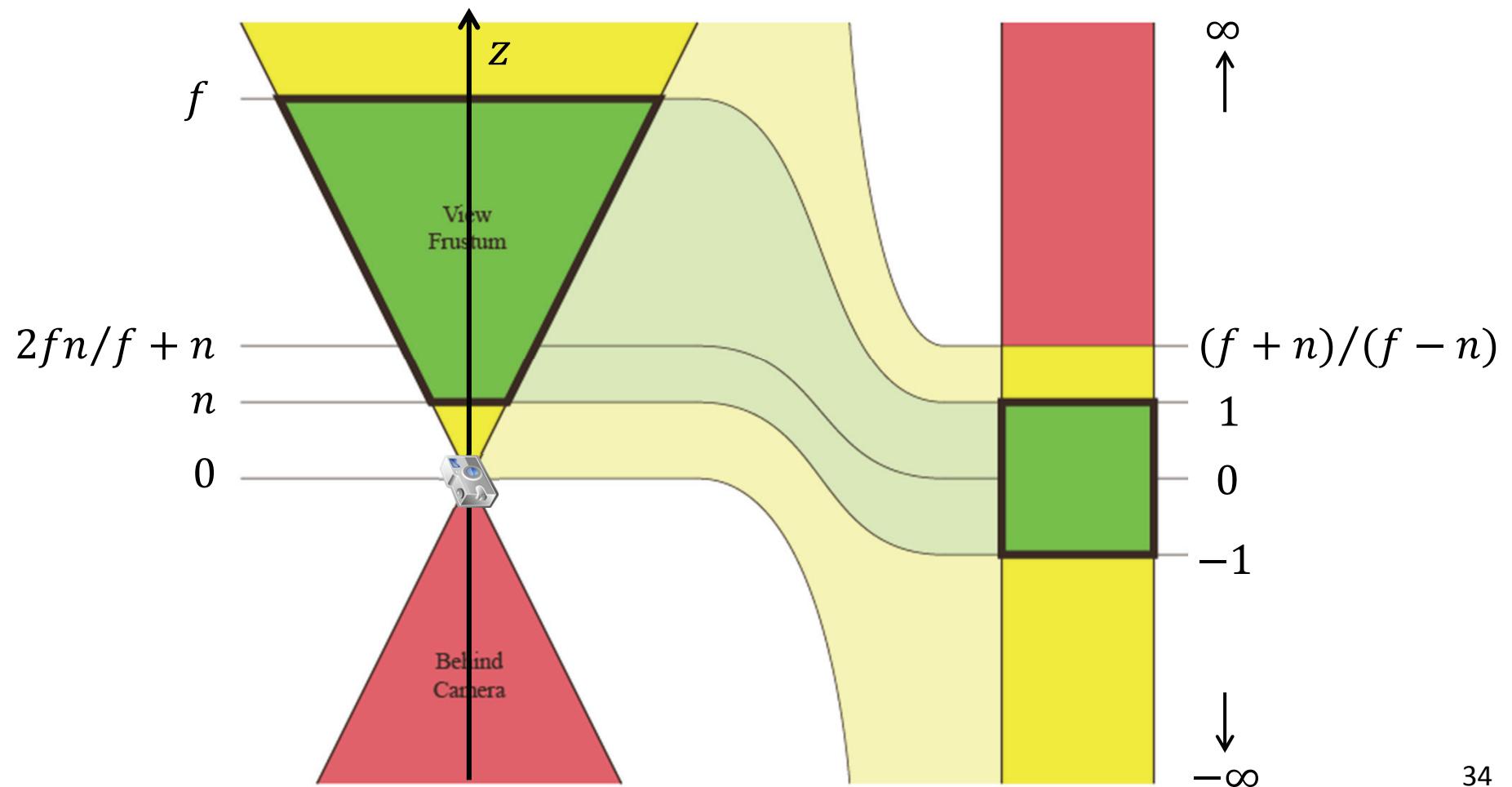
Normalisierungstransformation und Clipping

► betrachten wir nochmals die Abbildung der Tiefe...





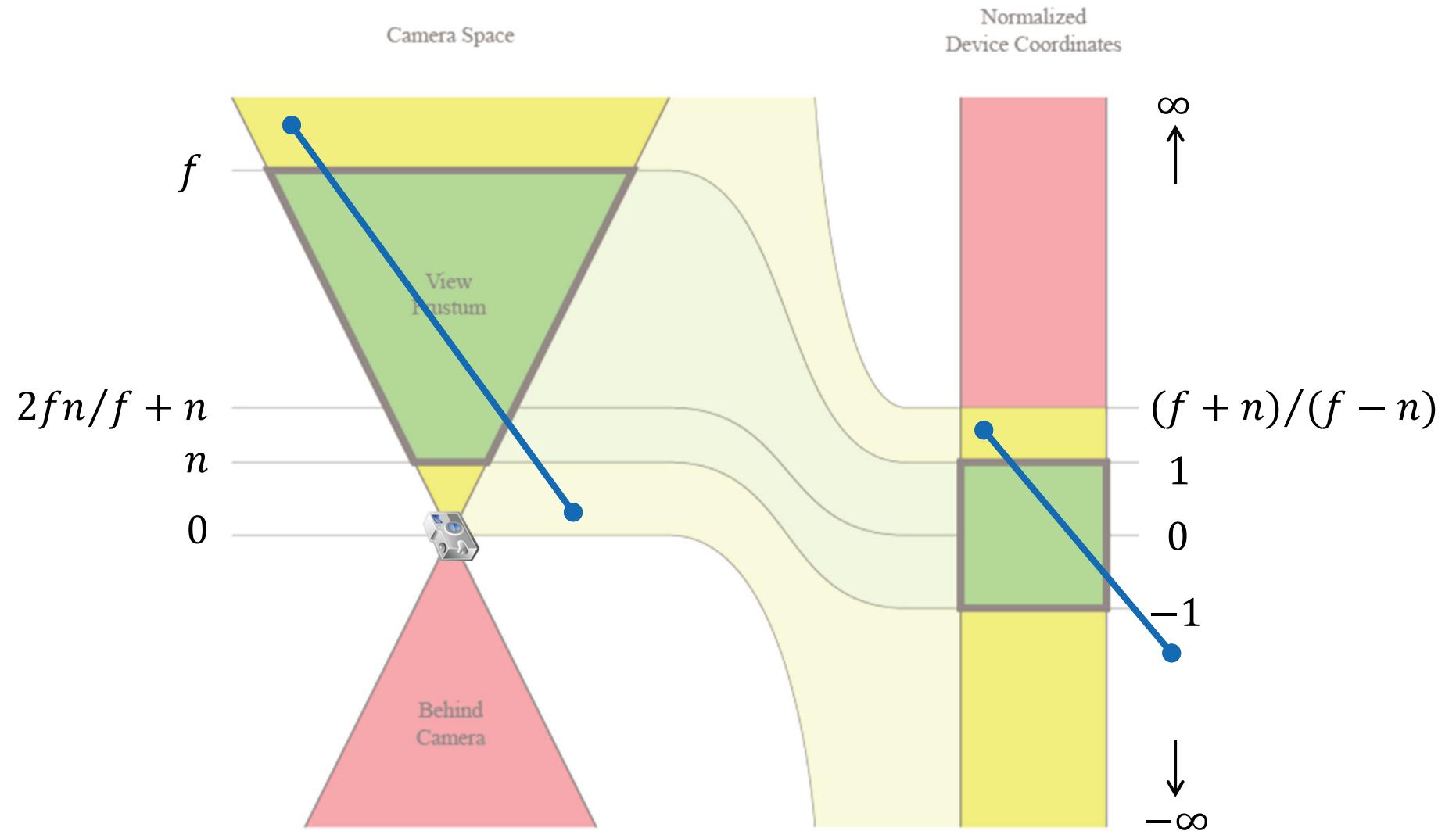
Normalized
Device Coordinates



Perspektivische Projektion in 2D

Normalisierungstransformation und Clipping

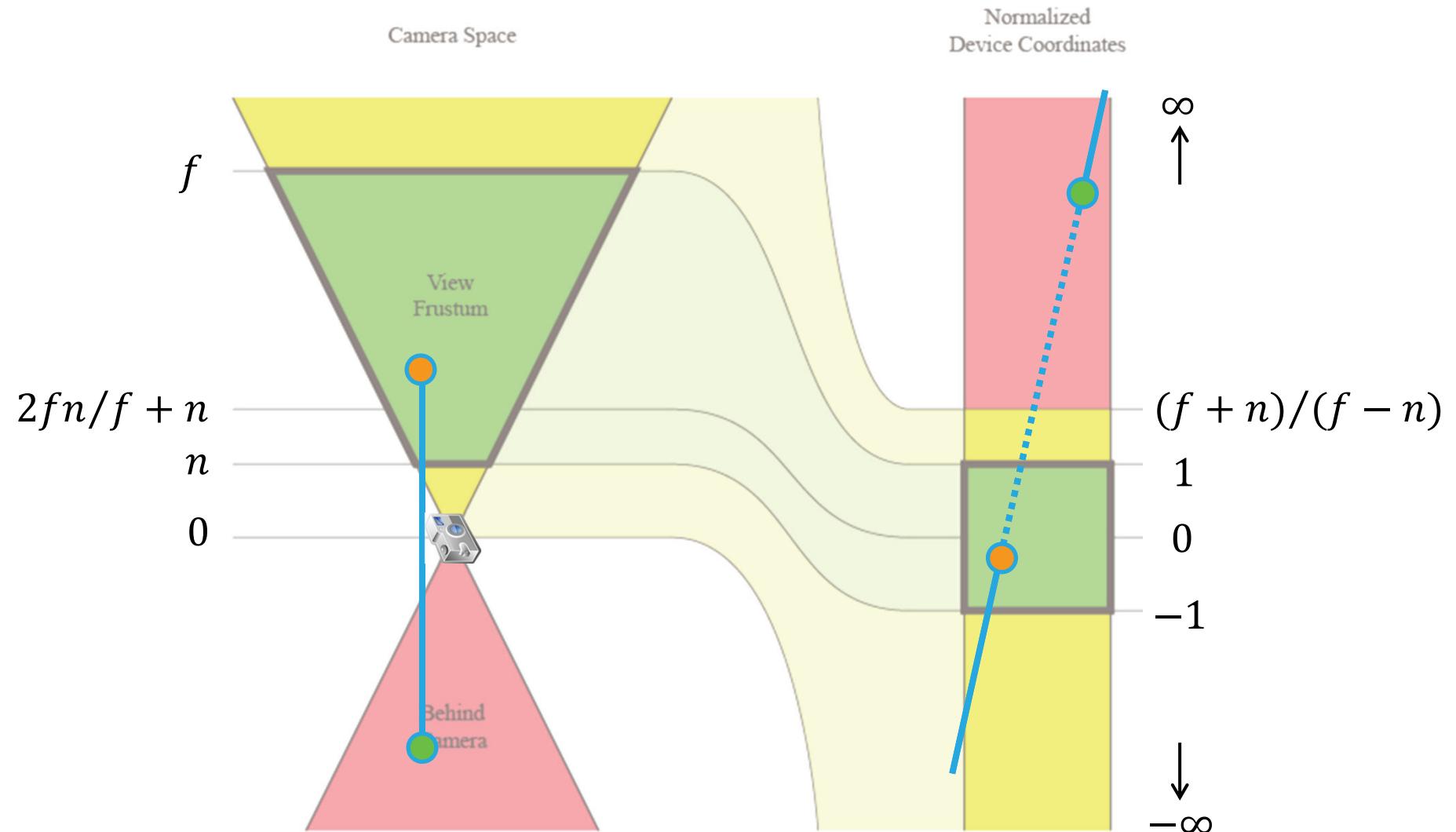
- ▶ Abbildung der Tiefe bereitet beim Clipping Schwierigkeiten



Perspektivische Projektion in 2D

Normalisierungstransformation und Clipping

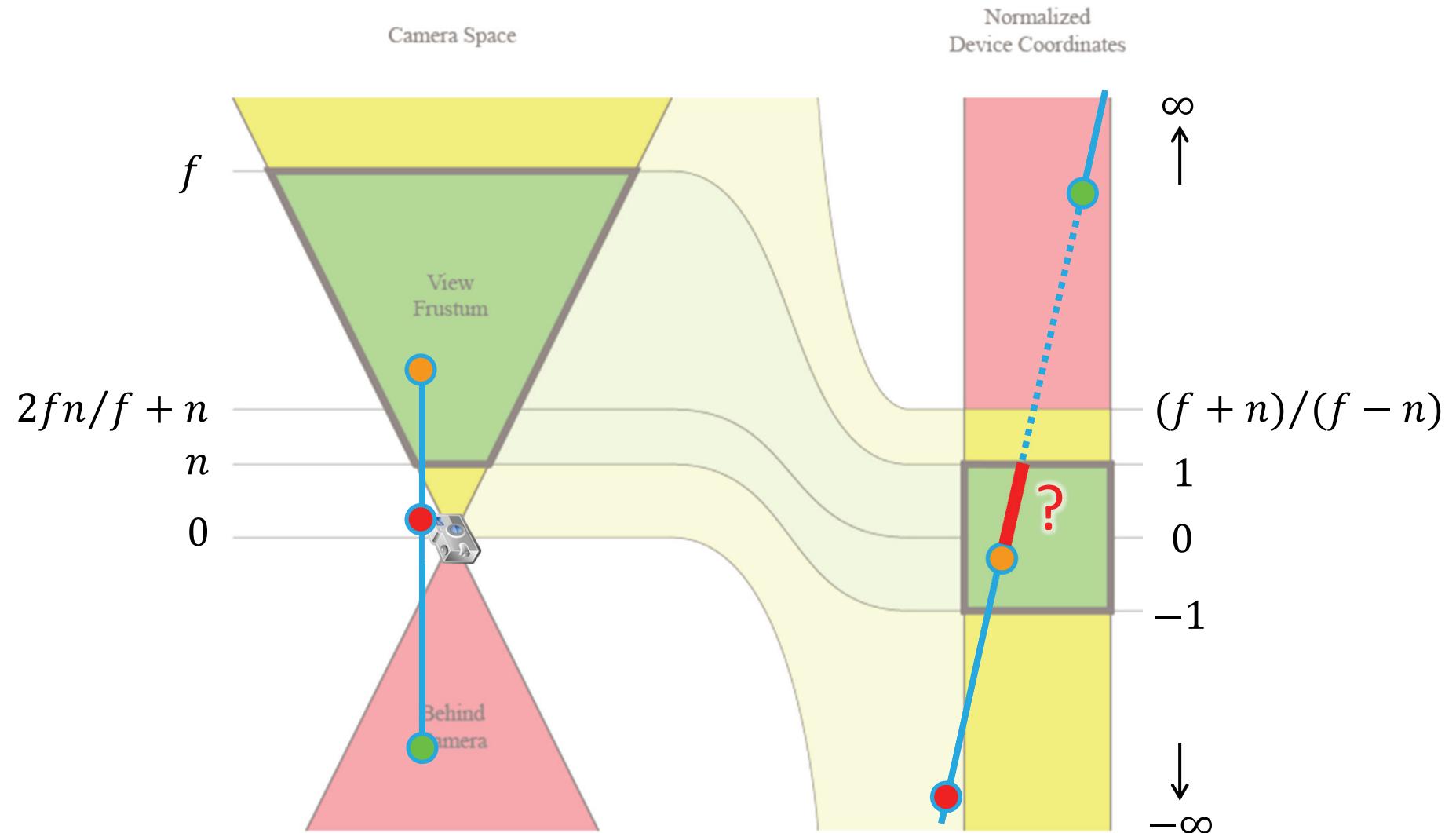
- ▶ Abbildung der Tiefe bereitet beim Clipping Schwierigkeiten



Perspektivische Projektion in 2D

Normalisierungstransformation und Clipping

- ▶ Abbildung der Tiefe bereitet beim Clipping Schwierigkeiten



Clipping und Projektionen

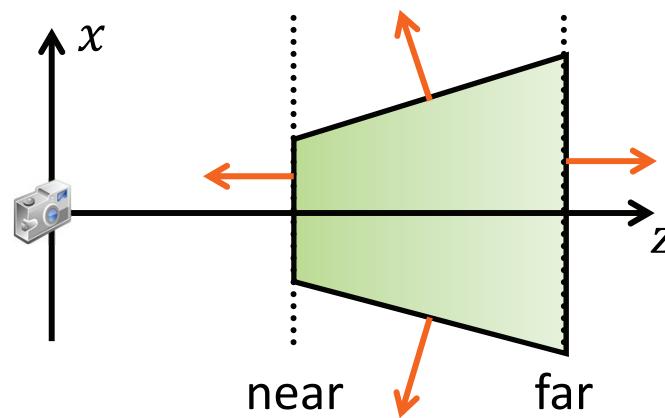
- ▶ Clipping in kartesischen Koordinaten **nach der Normalisierung ist falsch:**
Ambiguitäten und falsche Liniensegmente
- ▶ was passiert bei der Projektionstransformation?
 - ▶ w ist nichts anderes als die Tiefe: $w = -z$

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

- ▶ Punkte $(1,1,1,1)$ und $(-1, -1, -1, 1)$ sind – ohne Betrachtung des z -Werts – nach der Dehomogenisierung ($x' = {}^x/w$) nicht mehr zu unterscheiden
- ▶ es kann auch passieren, dass $w = -z = 0$ ist \rightarrow Division durch 0?
- ▶ wir können einen Test für einzelne Punkte durchführen, aber wie funktioniert Clipping für eine Linie (oder ein Polygon)?

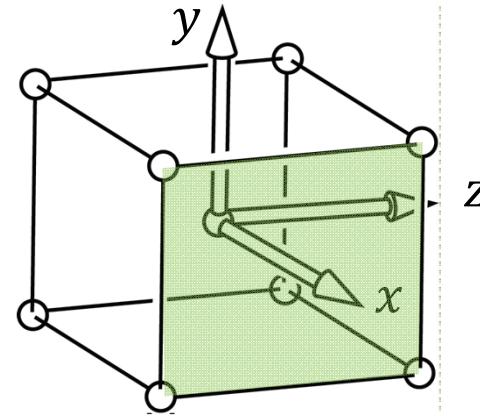
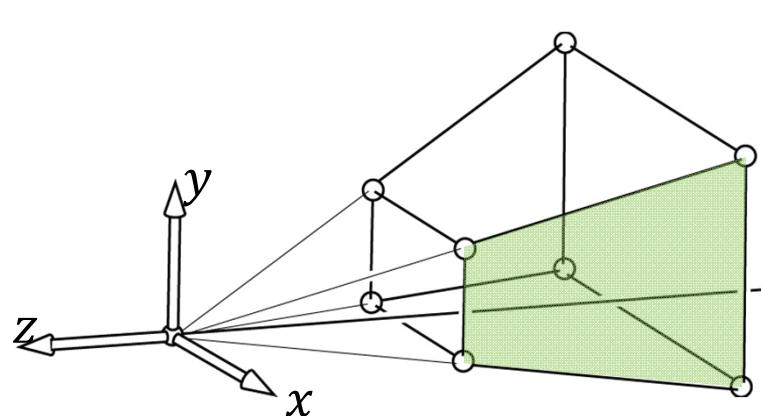
Clipping und Projektionen

- ▶ Option 1: Clipping gegen das Frustum in Welt- oder Kamerakoordinaten
 - ▶ vor der Projektionstransformation
 - ▶ stelle **6 Ebenengleichungen** auf
 - ▶ 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

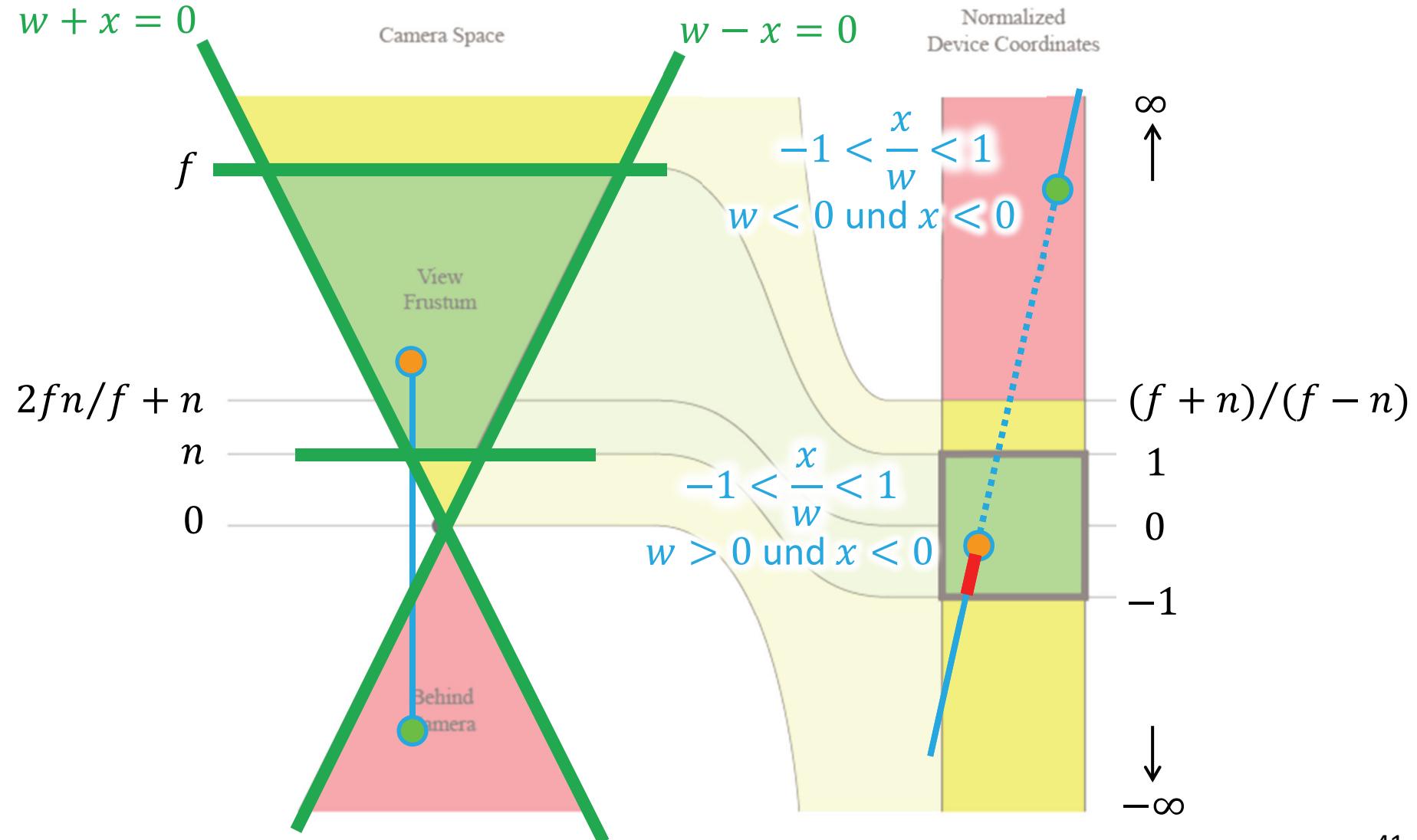


- ▶ für alle Punkte auf der grünen Seite gilt
nach dem Dehomogenisieren: $x' = x/w = 1$
- ▶ vor dem Dehomogenisieren gilt: $x = w$

Perspektivische Projektion in 2D

Normalisierungstransformation und Clipping

- Clipping vor der Normalisierungstransformation löst Mehrdeutigkeiten



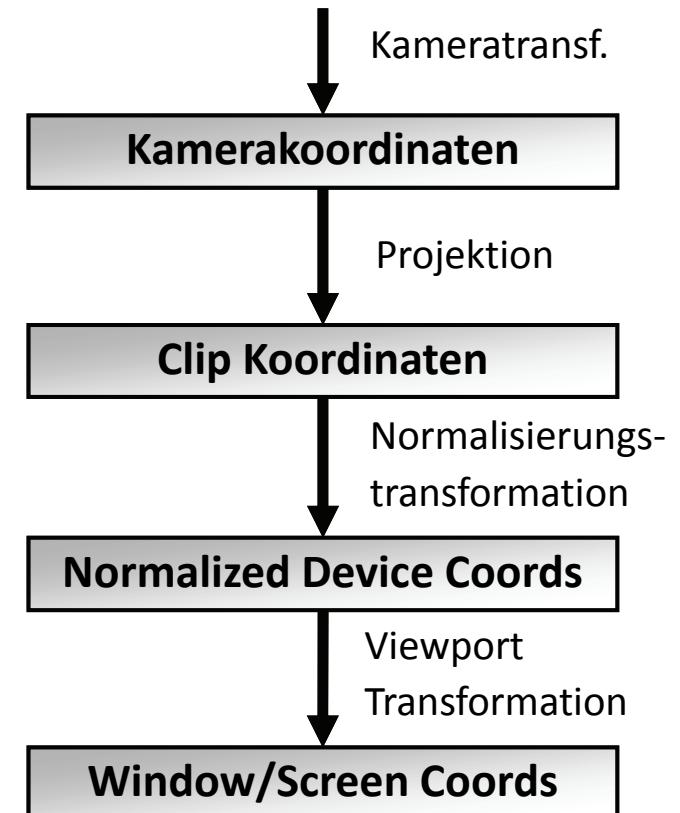
Clipping in homogenen Koordinaten

- ▶ Option 2: Clipping in homogenen Clip-Koordinaten
 - ▶ Achtung: das Frustum ist 4D, begrenzt durch 3D-Hyperebenen
 - ▶ Bsp. linke Seite des Frustum
 - ▶ Clip-Ebene in projizierten Koordinaten: $x' = -1$ (Annahme $w > 1$)
 - ▶ in homogenen Koordinaten $x/w = -1$ bzw. $x + w = 0$
 - ▶ Clip-Ebenen
 - ▶ links: $w + x = 0$ rechts: $w - x = 0$
 - ▶ unten: $w + y = 0$ oben: $w - y = 0$
 - ▶ near: $w + z = 0$ far: $w - z = 0$
 - ▶ α -Clipping (siehe Übung)
 - ▶ Ebenengl. sind gleichzeitig homogene WECs, z.B. $WEC_L(P) = x + w$
 - ▶ Outcodes und trivial accept/reject wie bisher
 - ▶ Bsp. $L(\alpha) = P_1 + \alpha(P_2 - P_1)$: $\alpha_L = \frac{WEC_L(P_1)}{WEC_L(P_1) - WEC_L(P_2)}$

Clipping in homogenen Koordinaten

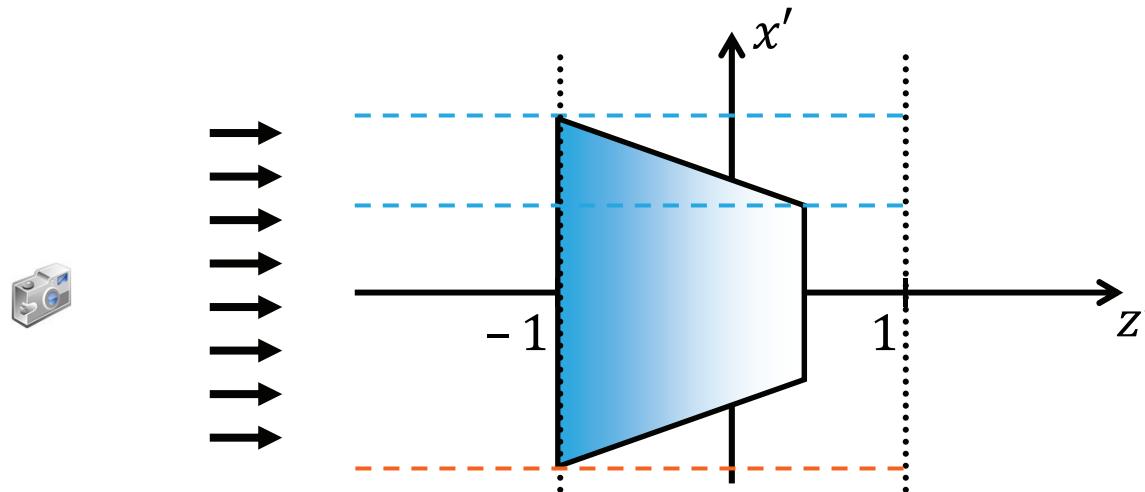
- ▶ Option 2: Clipping in homogenen Clip-Koordinaten
 - ▶ nach der Projektionstransformation, aber vor dem Dehomogenisieren

- ▶ 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

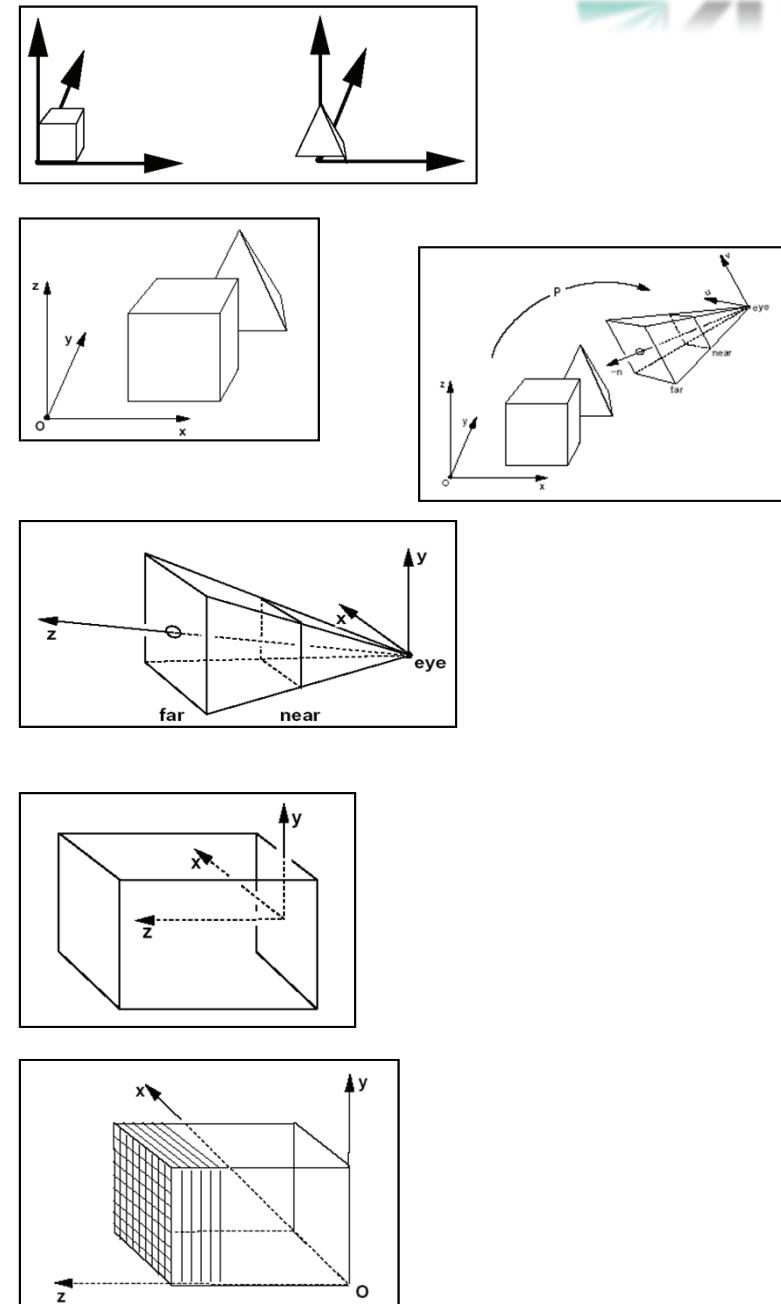
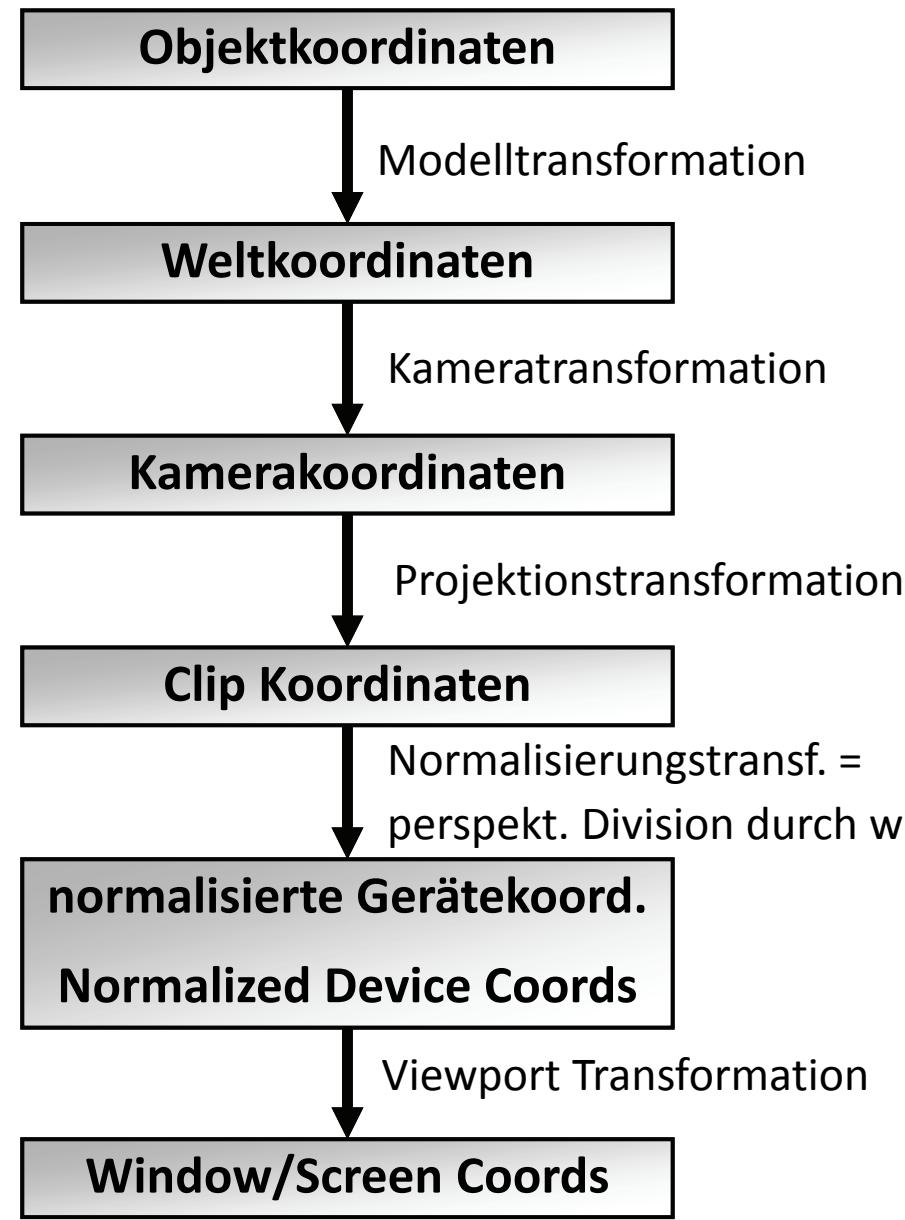
- der Viewport ist der rechteckige Bereich in dem das Bild dargestellt wird
- projizierte Koordinaten (x', y') durch Weglassen der Tiefe
(=orthographische Projektion)



- (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:

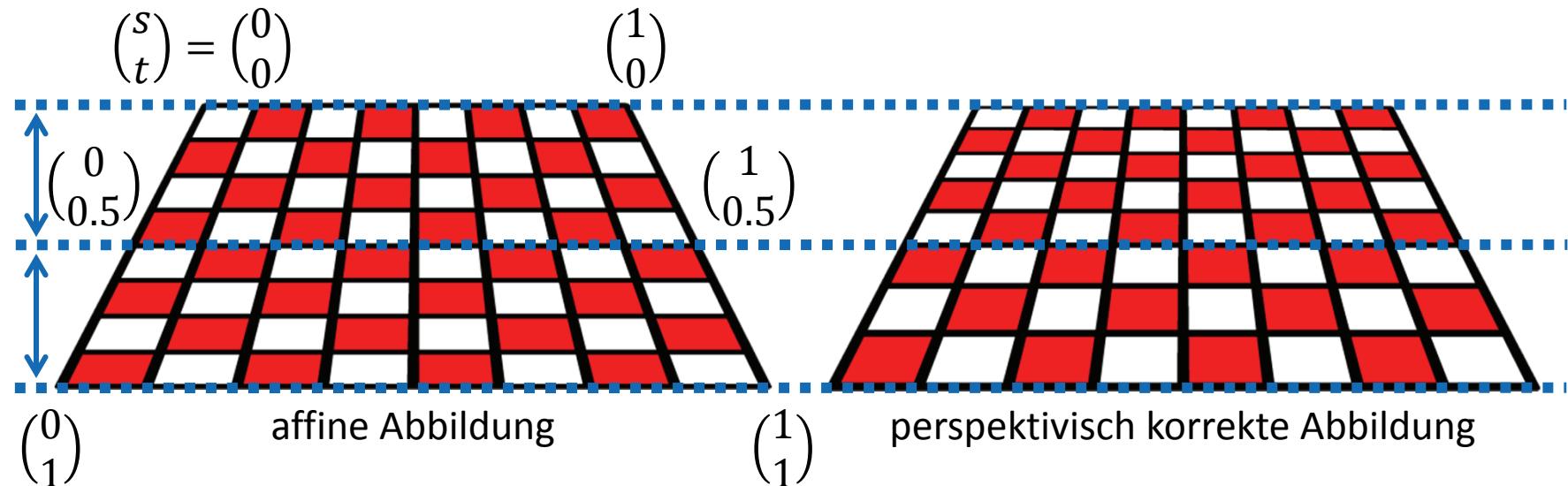
$$x_w = (x' + 1) \frac{width}{2} + x \quad \text{bzw.} \quad y_w = (y' + 1) \frac{height}{2} + y$$

Koordinatensysteme in der CG



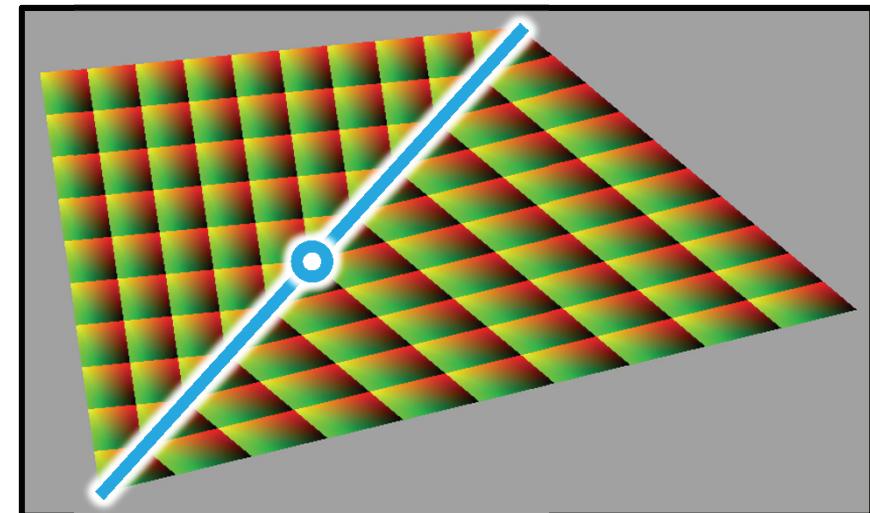
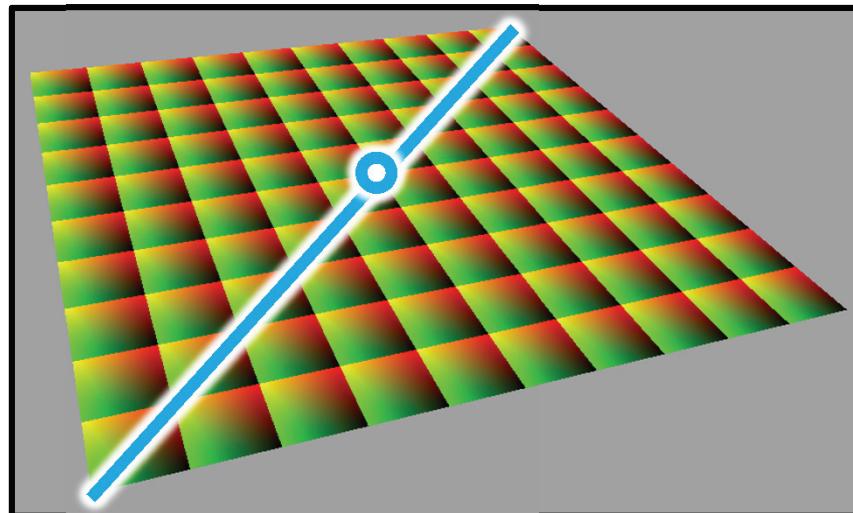
Perspektivisch-korrekte Attribut-Interpolation

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



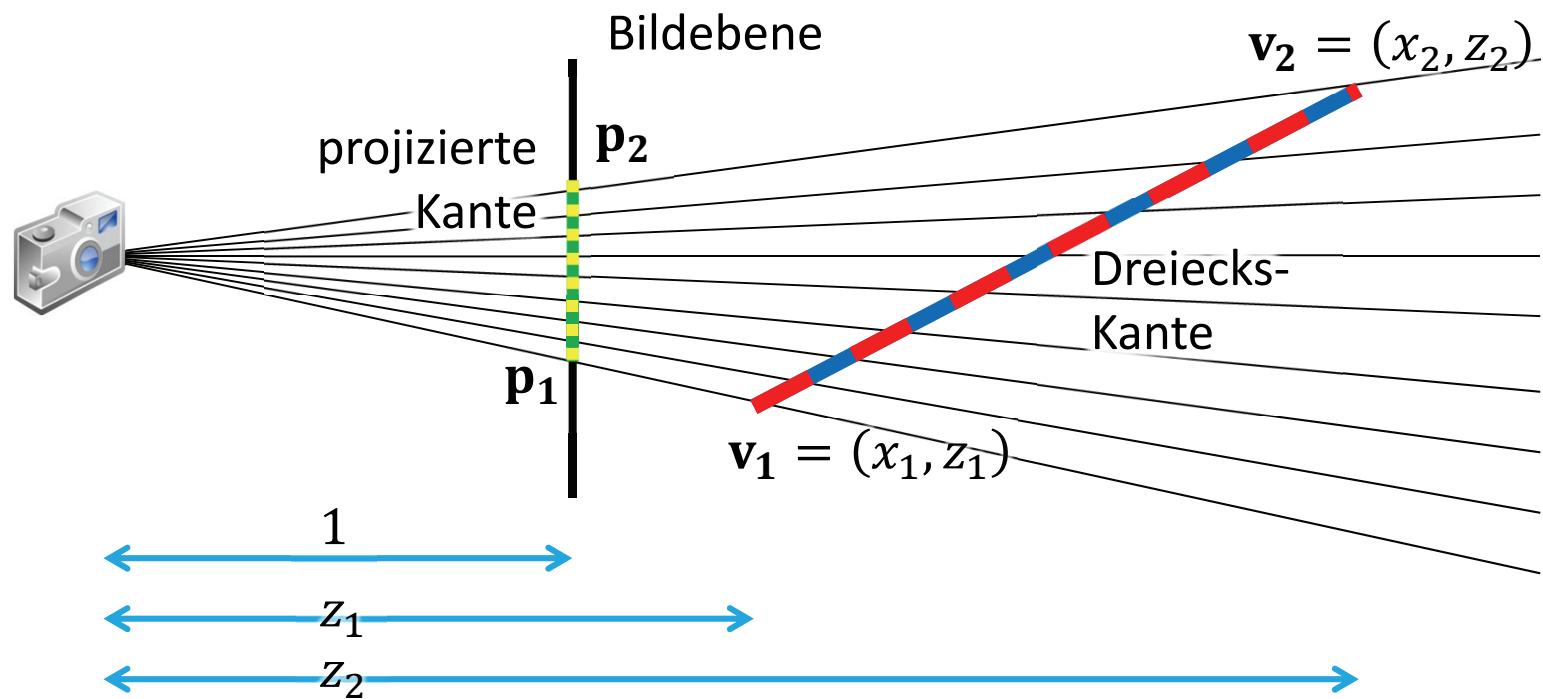
Perspektivisch-korrekte Attribut-Interpolation

- ▶ lineare Interpolation im Bildraum (bisher) 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 (links) und Interpolation im Bildraum (rechts), das Quadrat besteht je aus 2 Dreiecken



Perspektivisch-korrekte Attribut-Interpolation

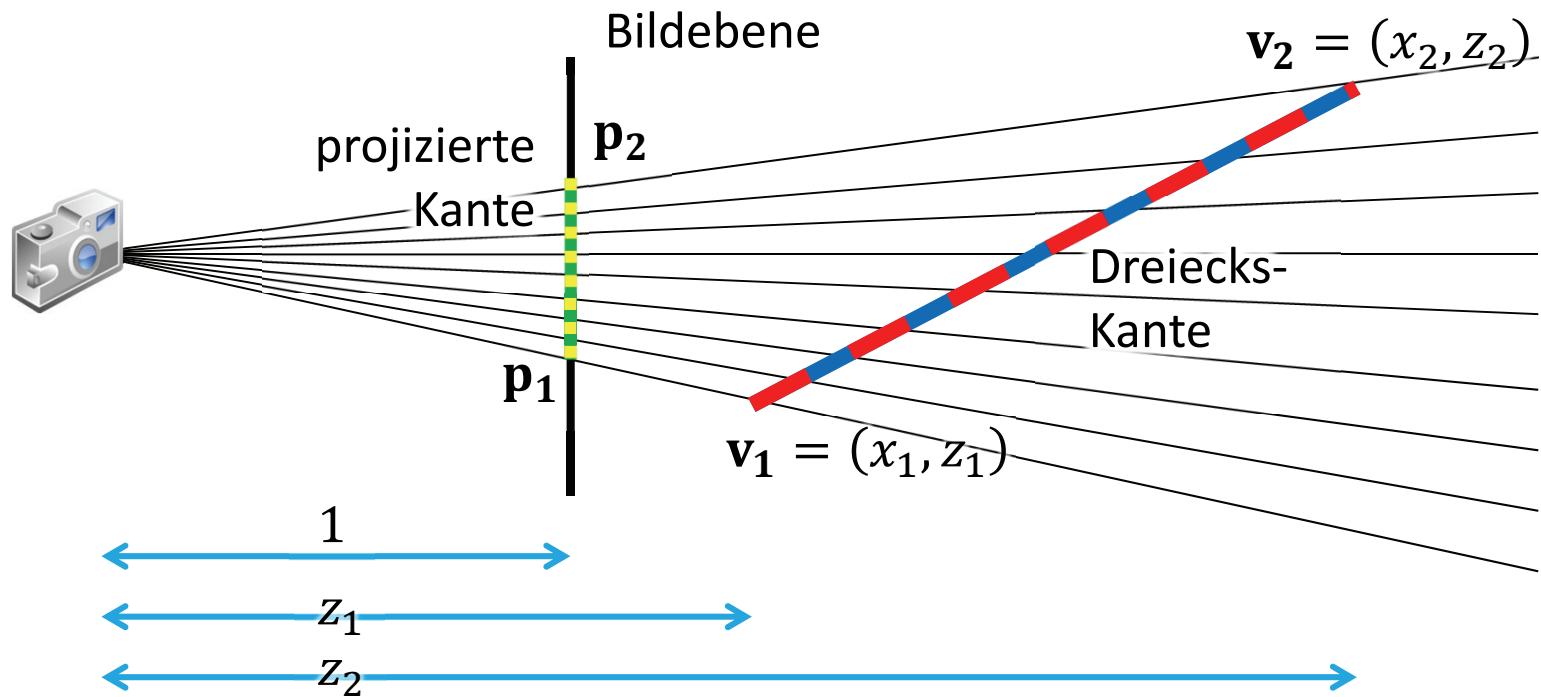
- ▶ lineare Interpolation im Bildraum (bisher) führt zu Verzerrungen
 - ▶ lineare Interpolation (bzw. linearer Teil der affinen Interpol.) ändert Längenverhältnisse nicht, perspektivische Projektionen aber schon!
- ▶ Veranschaulichung in 2D:



Perspektivisch-korrekte Attribut-Interpolation

- ▶ Veranschaulichung in 2D:
 - ▶ Bildebene bei $z = 1$ (obdA)
 - ▶ lineare Interpolation entlang der Kante $\mathbf{v}_1 = (x_1, z_1)$ nach $\mathbf{v}_2 = (x_2, z_2)$ im Bildraum:

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

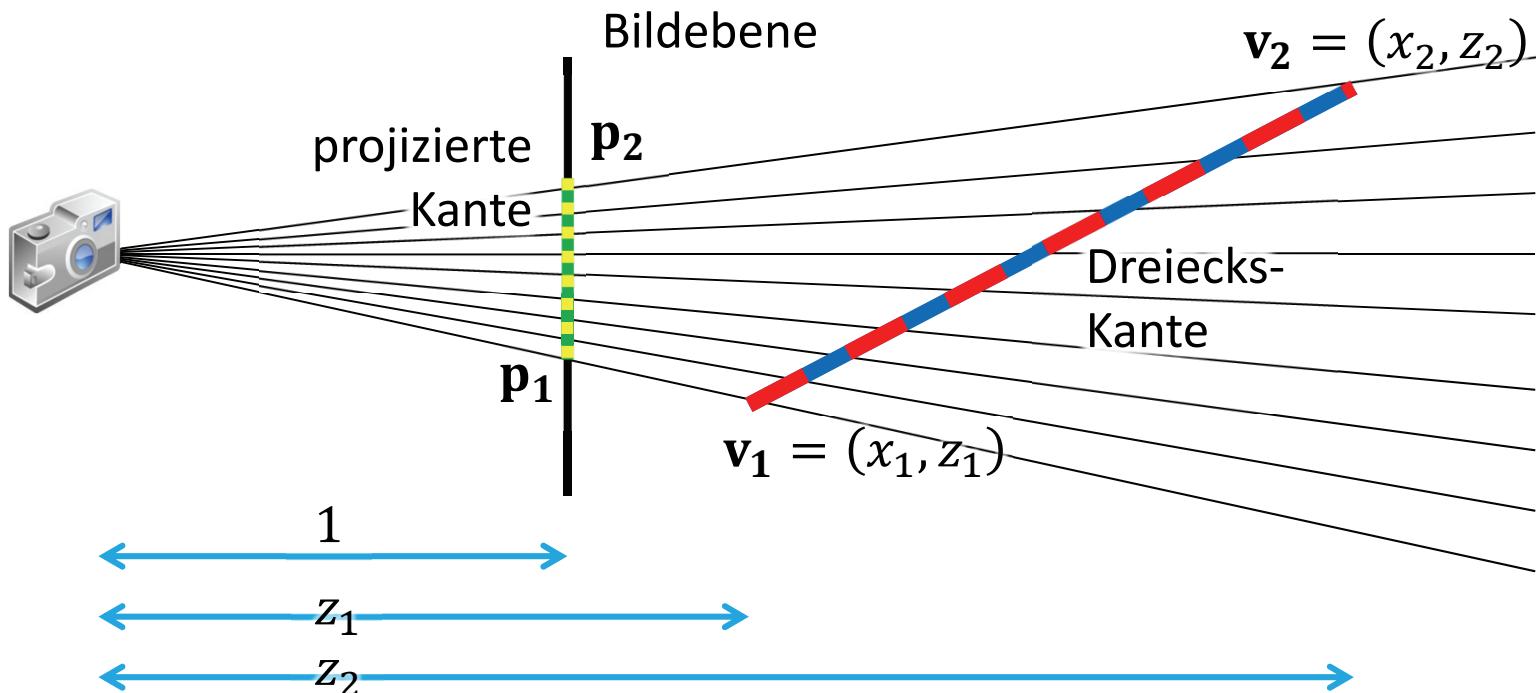


Perspektivisch-korrekte Attribut-Interpolation

- Veranschaulichung in 2D:

- lineare Interpolation entlang der Kante $\mathbf{v}_1 = (x_1, z_1)$ nach $\mathbf{v}_2 = (x_2, z_2)$ vor der Projektion:

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

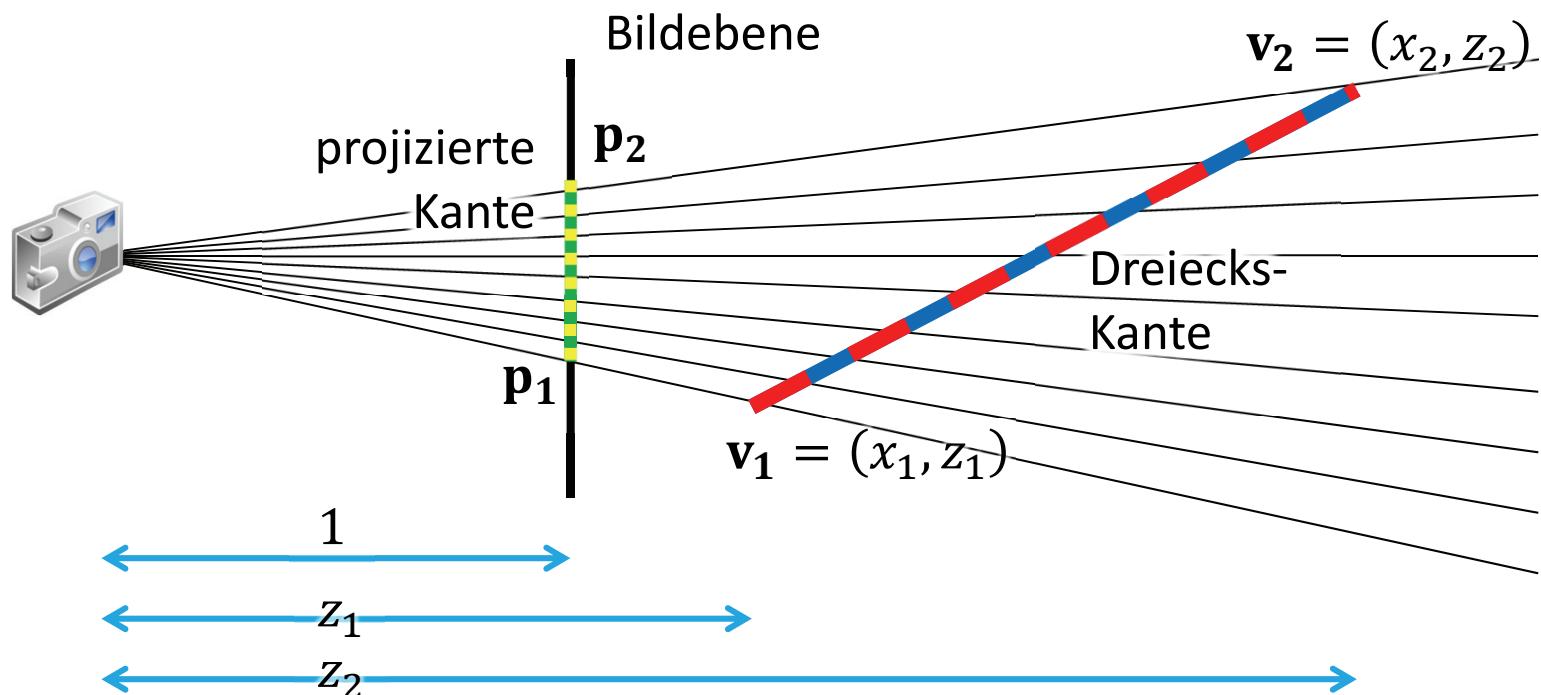


Perspektivisch-korrekte Attribut-Interpolation

- im Allgemeinen gilt natürlich

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

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



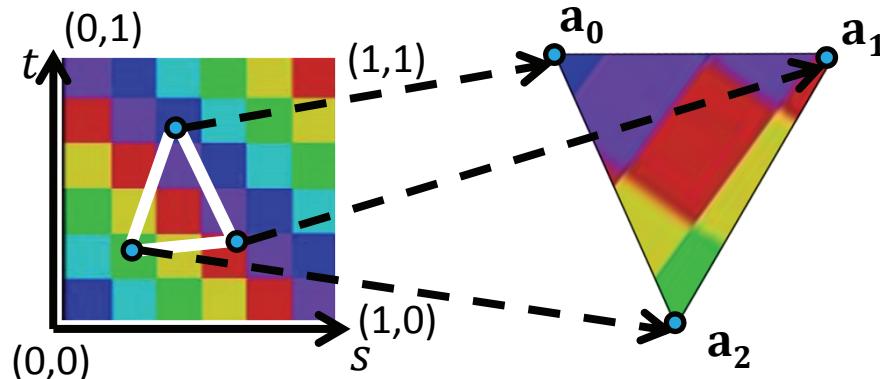
Perspektivisch-korrekte Attribut-Interpolation

Affine Abbildung von Texturraum in Objektraum

- die Texturkoordinaten der Vertizes repräsentieren die Texturparametrisierung eines Dreiecks
- es handelt sich dabei um eine affine Abbildung, die sich eindeutig bestimmen lässt:

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \\ 0 & 0 & j \end{pmatrix} \begin{pmatrix} s \\ t \\ 1 \end{pmatrix}$$

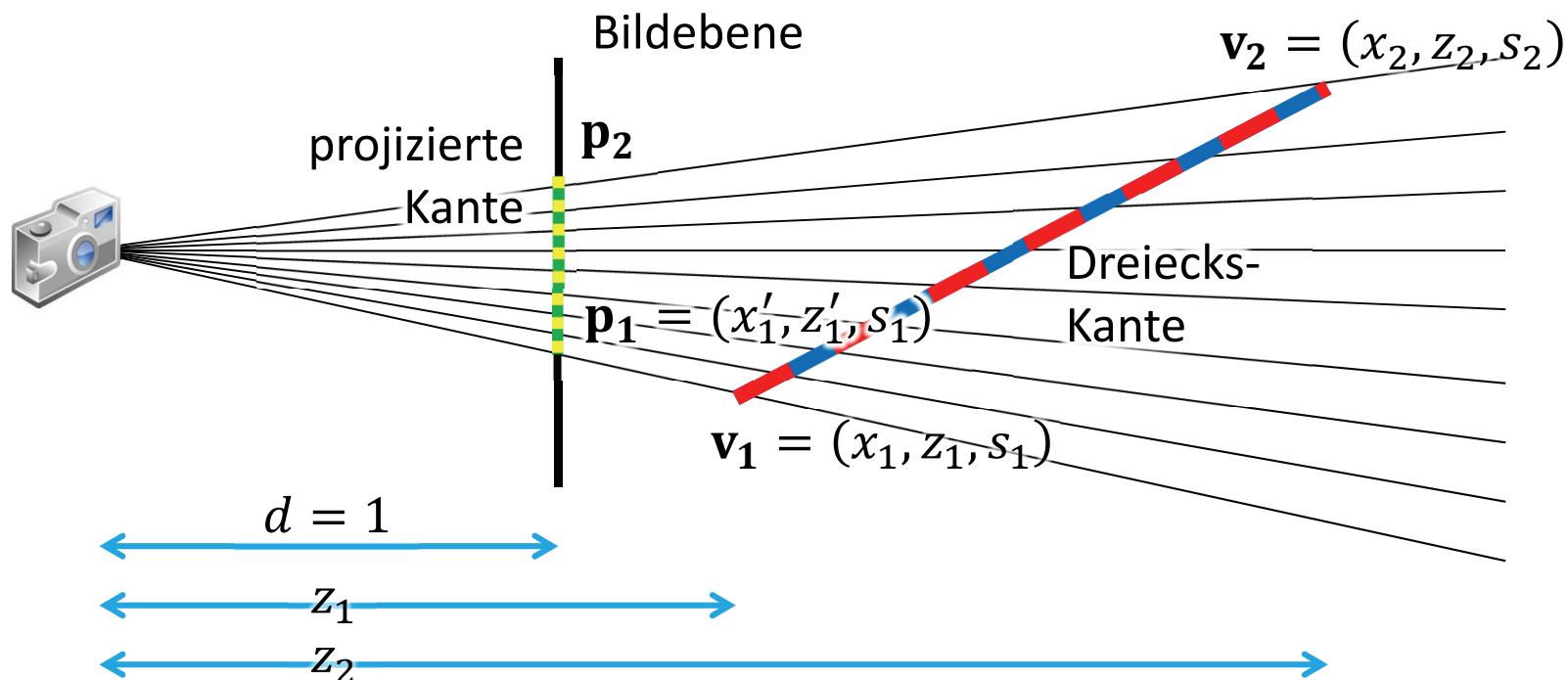
- Beispiel: Texturkoordinaten (baryzentrische Interpolation = ParallelKoSys)



Fundamentals of Texture Mapping and Image Warping,
Paul Heckbert, Master's thesis, UCB/CSD 89/516,
CS Division, U.C. Berkeley, June 1989

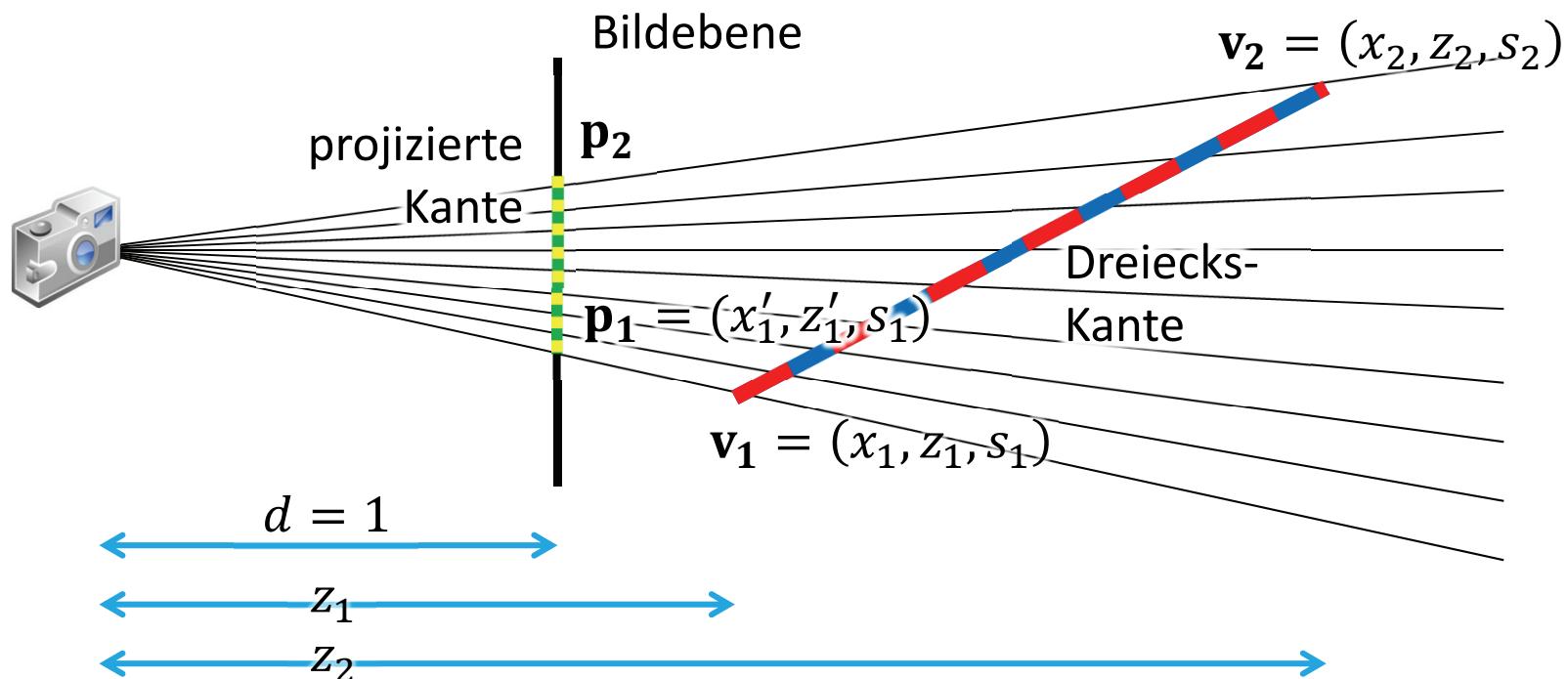
Perspektivisch-korrekte Attribut-Interpolation

- ▶ beim Rasterisieren finden wir alle Pixel, die das Polygon bedeckt
 - ▶ wir möchten zu jedem Pixel (Punkt auf der Bildebene) die Position in Kamerakoordinaten vor der Projektion berechnen
(nur mal so, um herauszufinden was wir eigentl. interpolieren sollten!)
- ▶ betrachte Punkt v_1 mit der Texturkoordinate s_1
 - ▶ es gilt $\frac{x_1}{z_1} = \frac{x'_1}{d}$ bzw. hier $x'_1 = \frac{x_1}{z_1}$ oder allgemein $x = x' \cdot z$



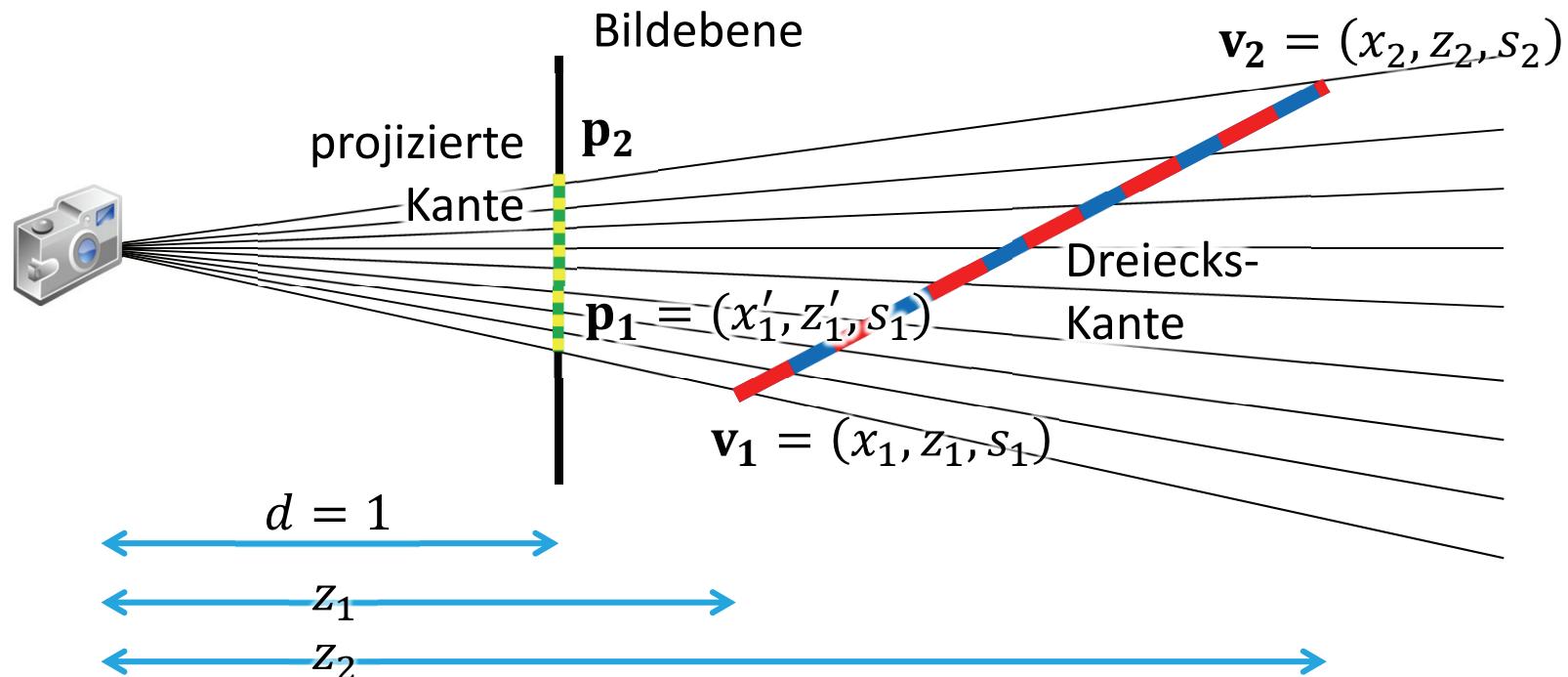
Perspektivisch-korrekte Attribut-Interpolation

- ▶ wie interpoliert man die Tiefe z bei der Rasterisierung? warum Tiefe?
 - ▶ berechnen der Kamerakoordinaten (und daraus Attribute)
 - ▶ Abstand der Fläche zur Kamera für jeden Pixel!
- ▶ Kamerakoordinaten x für eine Pixelposition x' : $x = x' \cdot z$
- ▶ in Kamerakoordinaten lässt sich A und B so bestimmen, dass gilt: $x = Az + B$



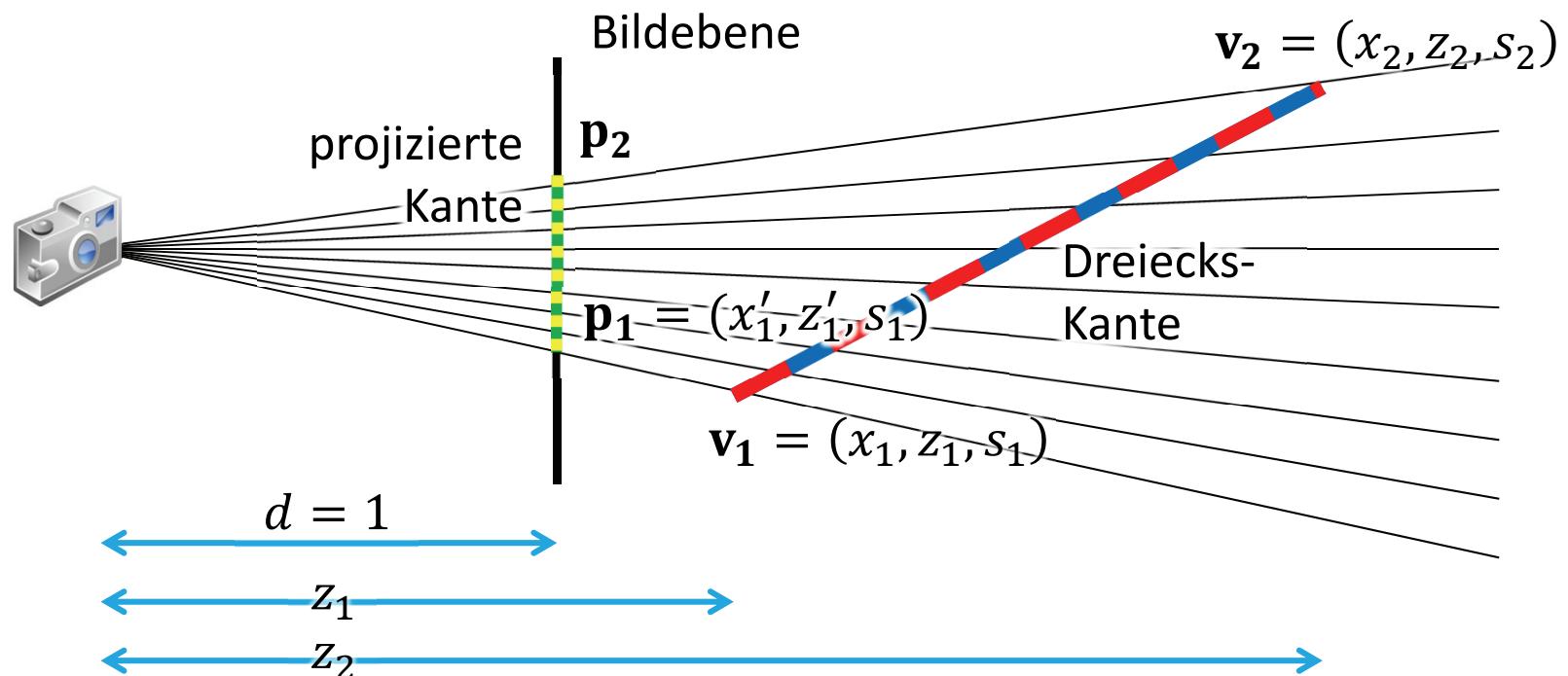
Perspektivisch-korrekte Attribut-Interpolation

- ▶ 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!



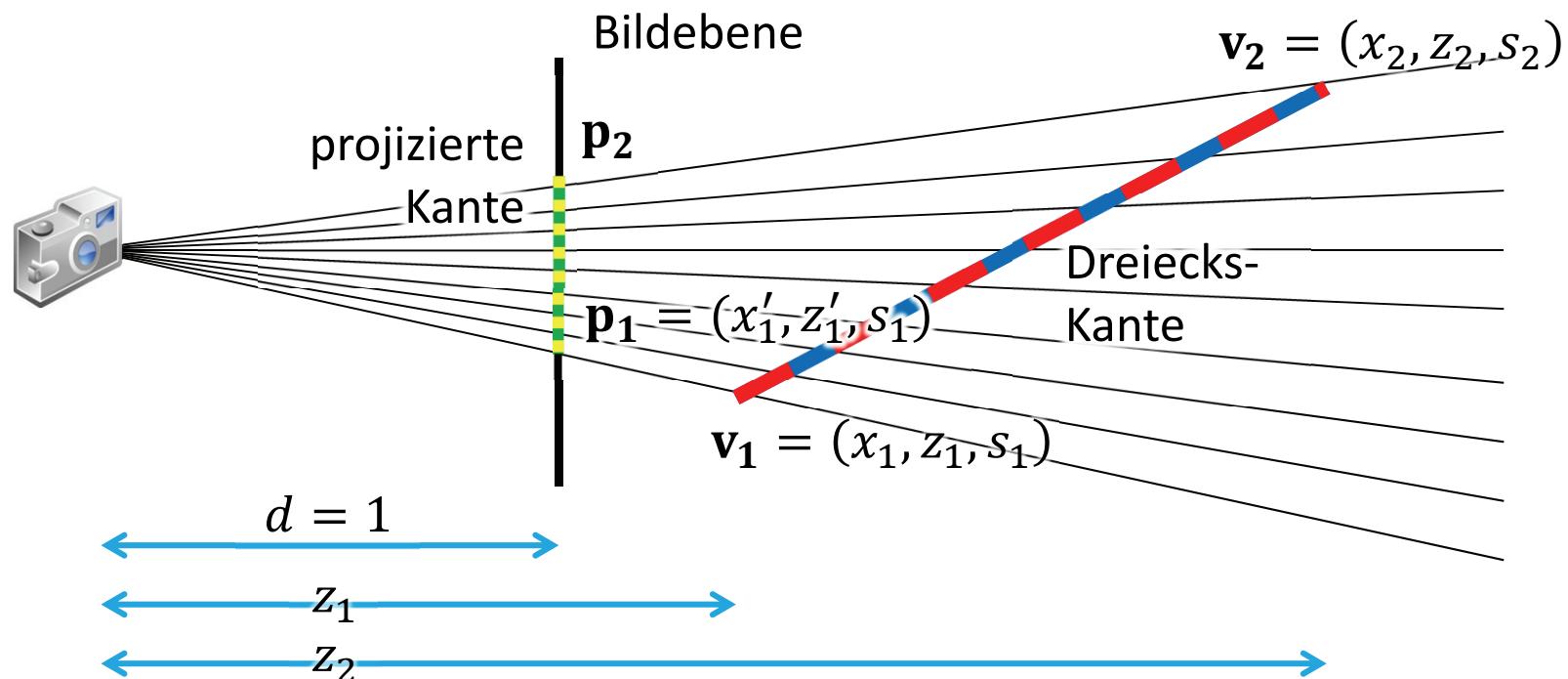
Perspektivisch-korrekte Attribut-Interpolation

- ▶ $\Rightarrow x' \cdot z = Az + B \Leftrightarrow z = \frac{B}{x' - A}$
- ▶ aber: $\frac{1}{z} = \frac{1}{B}x' - \frac{A}{B}$
- ▶ \Rightarrow man darf $\frac{1}{z}$ im Bildraum interpolieren und erhält x mit $x = \frac{x'}{1/z}$



Perspektivisch-korrekte Attribut-Interpolation

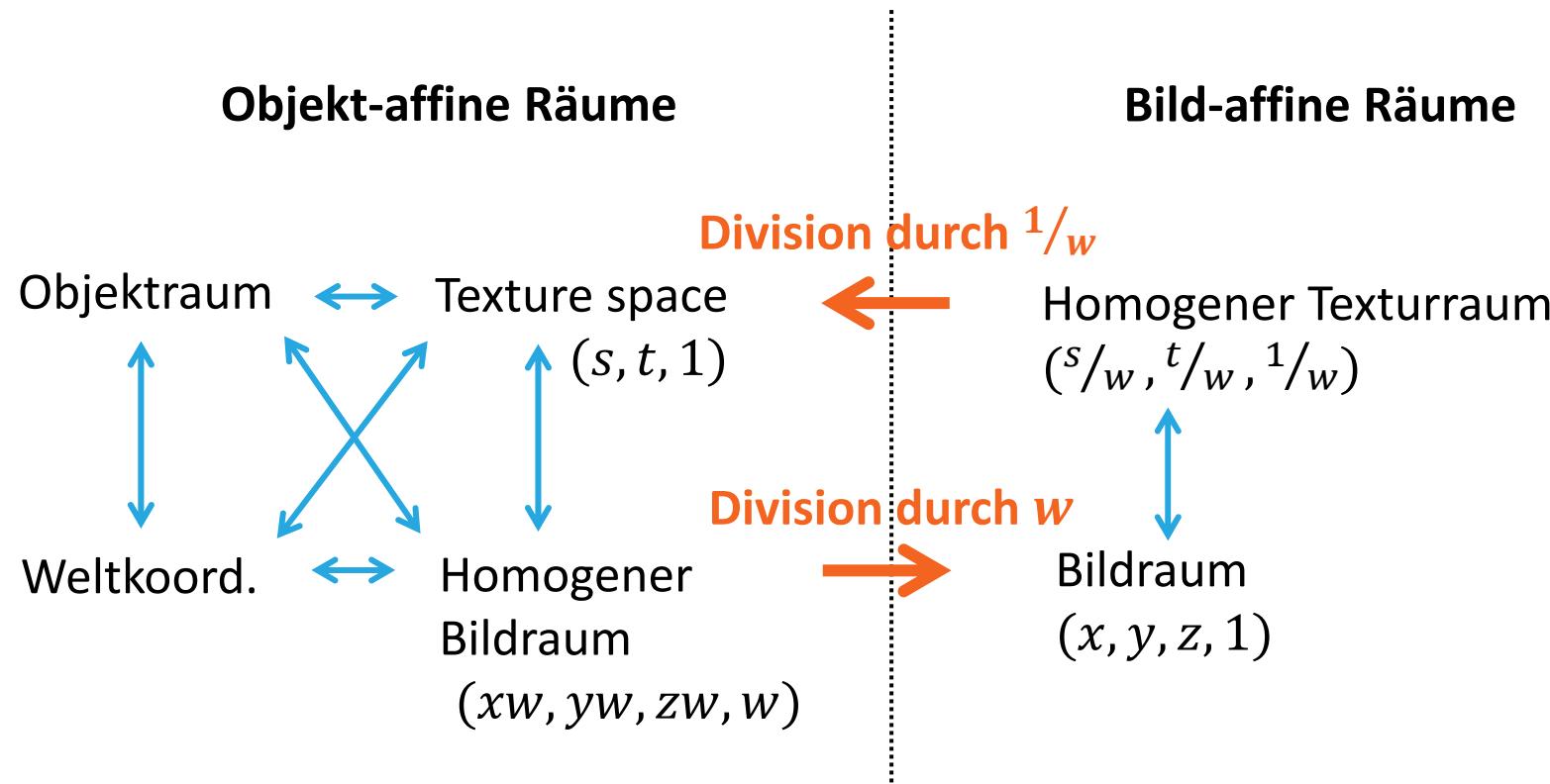
- ⇒ man darf $\frac{1}{z}$ im Bildraum interpolieren und erhält x mit $x = \frac{x'}{\frac{1}{z}}$
- die Attribute können dann über baryzentrische Koordinaten interpoliert werden (berechnet mittels x bzgl. der Vertizes in Kamerakoord.)
- ebenso darf man gleich $s/z, t/z, \dots$ im Bildraum interpolieren und erhält mit $\frac{1}{z}$ direkt die perspektivisch-korrekt interpolierten Attribute



Perspektivisch-korrekte Attribut-Interpolation

Perspektivisch-korrekte Interpolation und homogene Koordinaten

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



Rasterisierung vs. Raytracing



Rasterisierung

- ▶ sehr effizient, Hardware-Umsetzung ist einfach
- ▶ nur für „Primärstrahlen“: globale Effekte (Schatten, Spiegelungen, indirektes Licht, etc.) nur über spezielle Techniken
- ▶ 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
- ▶ Echtzeit-Rendering

Raytracing

- ▶ konzeptionell einfaches Verfahren
- ▶ Sekundärstrahlen, komplexe Beleuchtungseffekte sind einfach
- ▶ Handhabung komplexer Szenen durch räumliche Datenstrukturen
 - ▶ verdeckte Geometrie wird „automatisch“ ausgeschlossen
- ▶ Offline-Rendering, bedingt interaktives Rendering