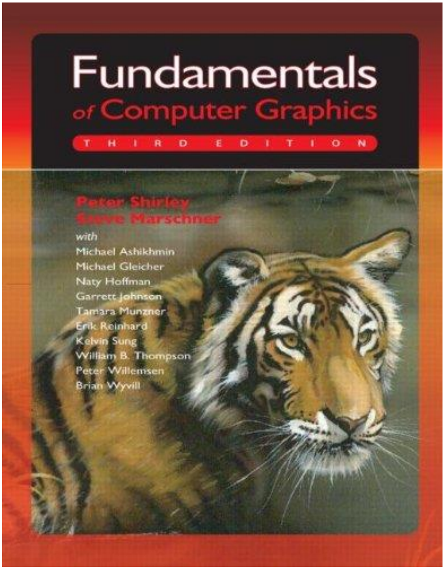
Computergrafik

Vorlesung im Wintersemester 2024/25
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



Bildsynthese

- 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

Interaktive/Echtzeit-Computergrafik

- Rasterisierung: (noch) Teil 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

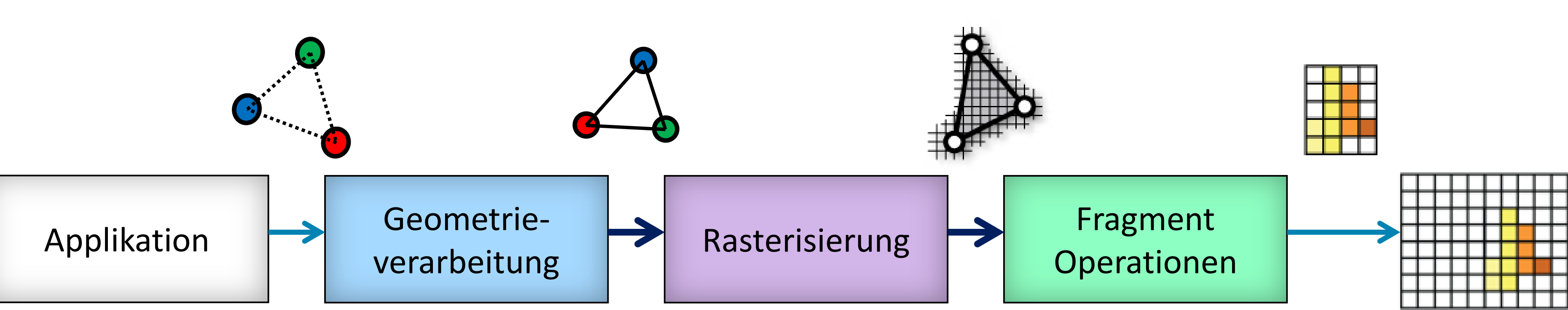
Betrachter

Bildebene

Interaktive/Echtzeit-Computergrafik

- Rasterisierung: (noch) Teil 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

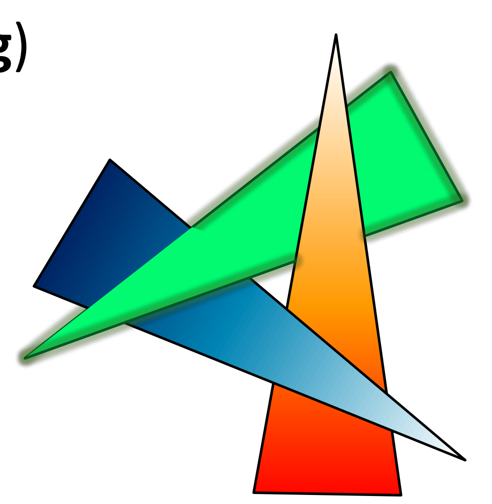
Grafik-Pipeline

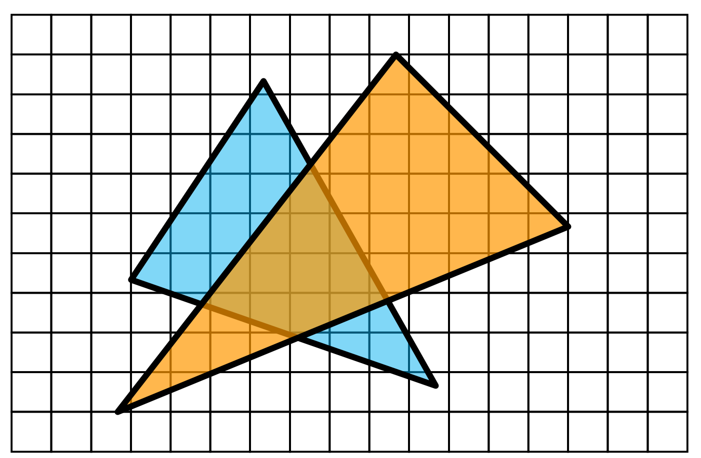


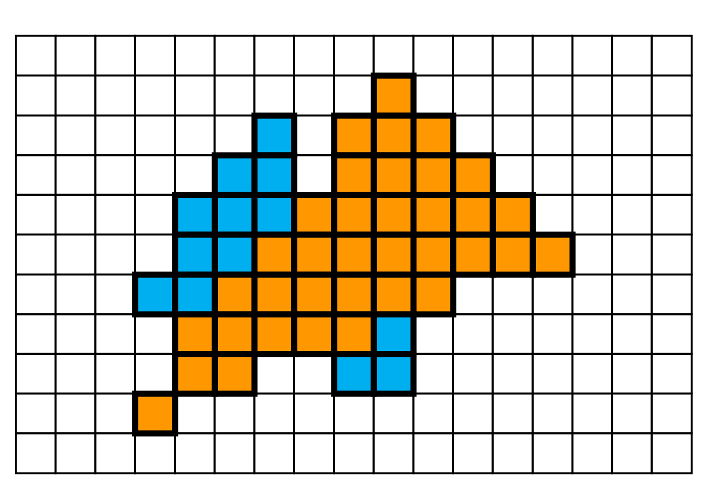
Inhalt

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 (Prinzip)
  - perspektivische Interpolation

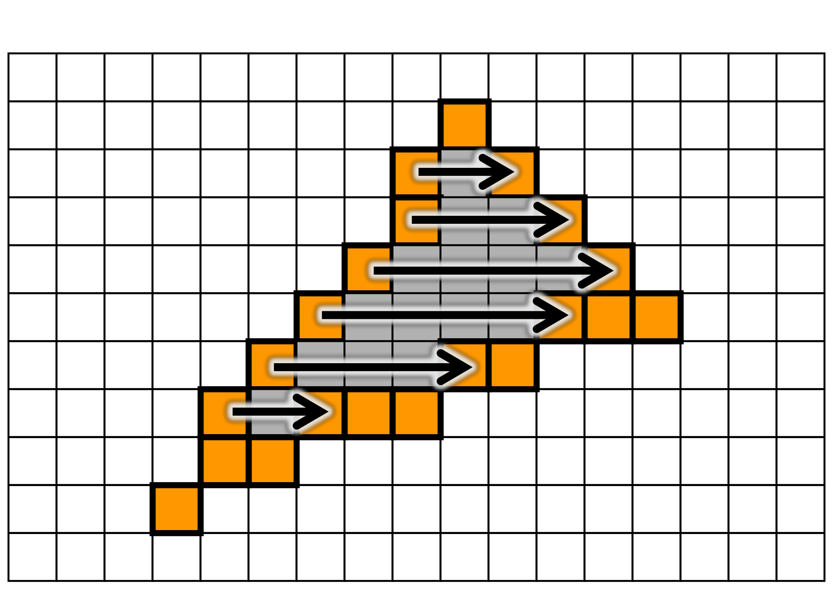


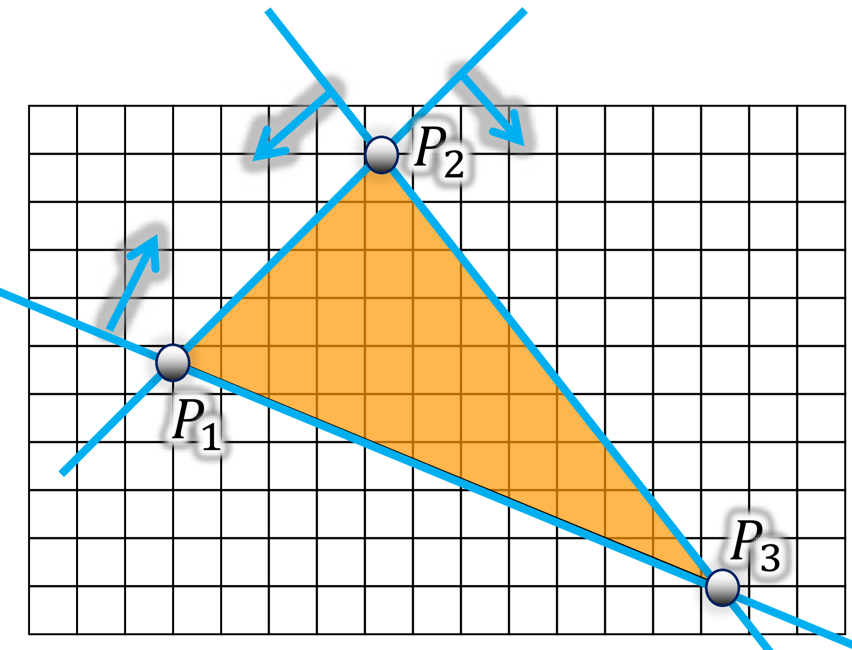




Rasterisierung von konvexen Polygonen/Dreiecken

- 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



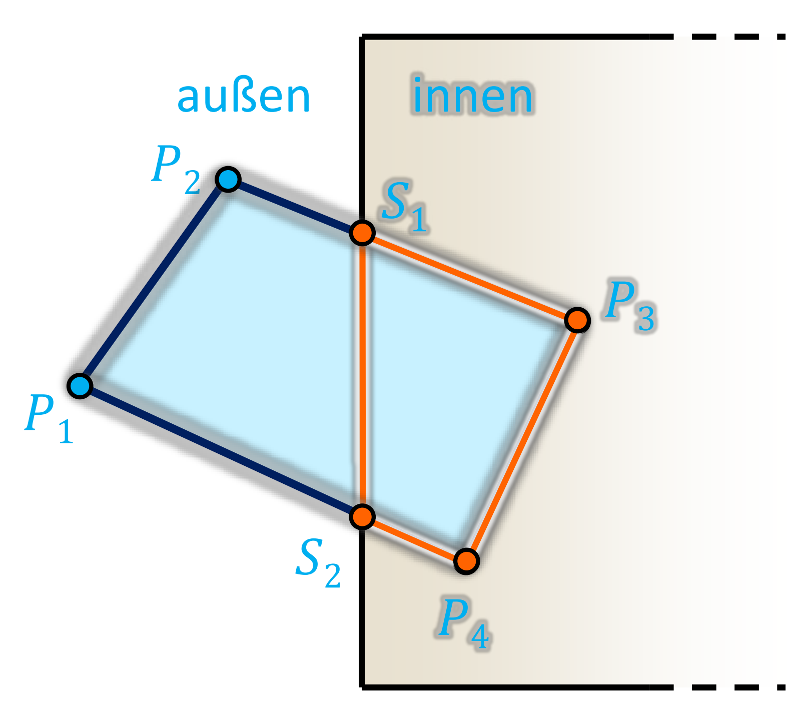


Clipping

- Clipping: Abschneiden von Linien/Polygon-Teilen die außerhalb des Bildschirms liegen (oder allg. außerhalb eines gewünschten Bereichs)
  - Clipping kann wichtig sein für 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

- Clipping: Abschneiden von Linien/Polygon-Teilen die außerhalb des Bildschirms liegen (oder allg. außerhalb eines gewünschten Bereichs)
  - Clipping kann wichtig sein für 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 Dreiecke – funktioniert nicht immer
- … und ist ineffizient: Sortierung benötigt Zugriff auf alle Dreiecke, Überschreiben von verdeckten Flächen(teilen)

$\mathbf{m}_P$ P $\mathbf{m}_Q$ Q

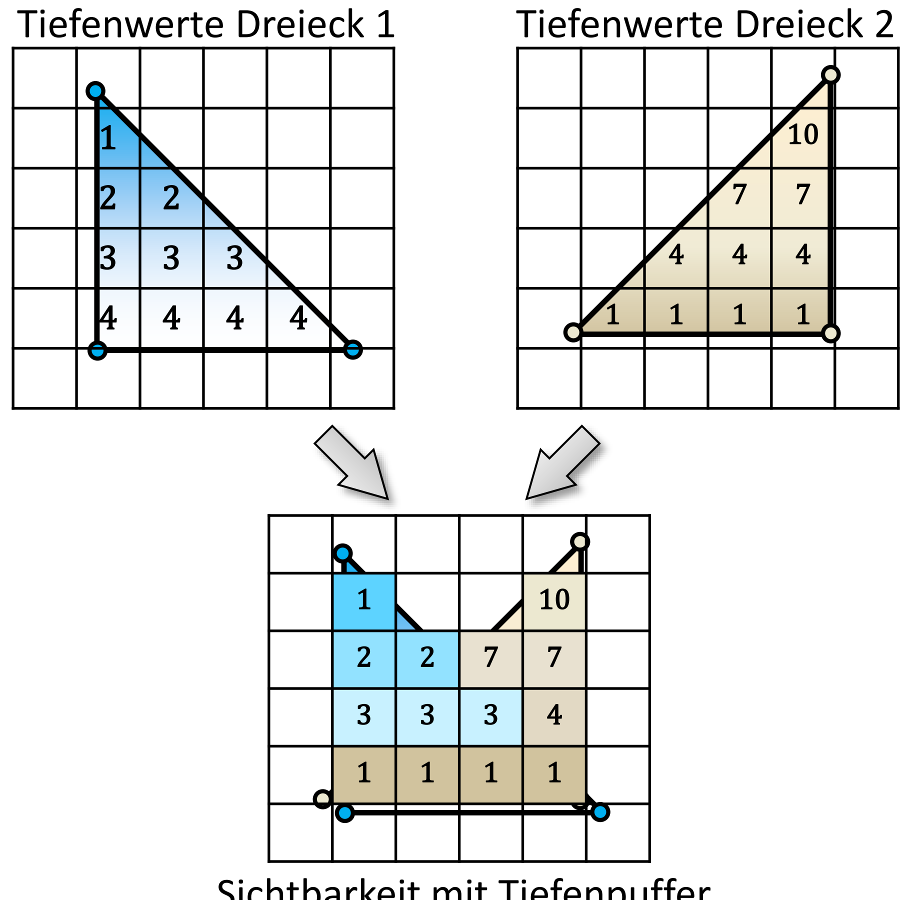
- wichtig für effiziente Verarbeitung (v.a. in Hardware): Pipelining
  - Bearbeitung eines Dreiecks nach dem anderen
  - unabhängig von allen anderen Dreiecken

Tiefenpuffer, Z-Buffering

parallel entwickelt von
E. Catmull und W. Straßer (1974)

Idee, Prinzip

- bildbasierter Ansatz: speichere für jeden Pixel Distanz zur nahsten Fläche
- Entfernung-/Tiefen-Wert wird pro Vertex berechnet und interpoliert
- zusätzlich zum Farbpuffer 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 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

Tiefenpuffer, Z-Buffering

Nachteile (oder besser: was man einfach bedenken sollte)

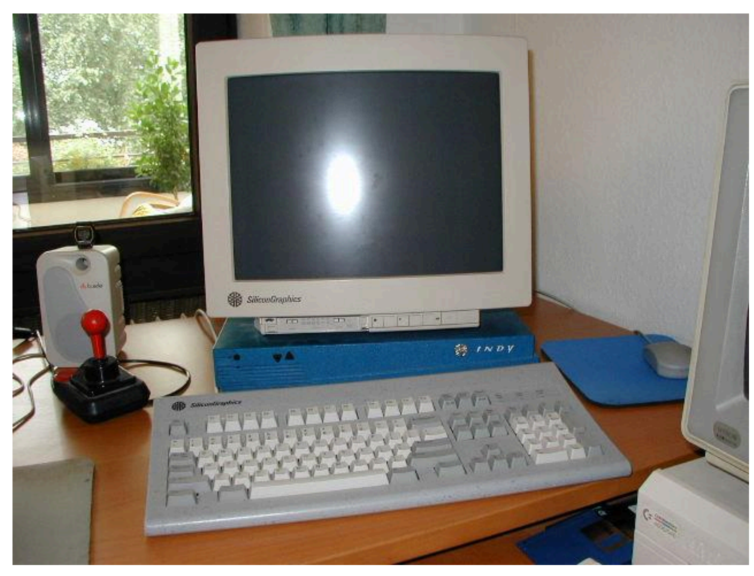
- zusätzlicher Speicherbedarf und -bandbreite (heute nicht mehr kritisch)
- begrenzte Genauigkeit und z-Aliasing (wichtig!)
- transparente Flächen können nicht ohne Weiteres behandelt werden
- bei naiver Umsetzung: unnötiger Aufwand bei hoher Tiefenkomplexität
  - Tiefenkomplexität = Anzahl der Schnitte 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

Grafik-Hardware

- SGI Indy (1993):
  Hardware rasterisiert Dreiecke
  (Gouraud Shading, keine Texturen)
- low-cost: 5000 US-$
  (umgerechnet ca. 10500€ in 2024)



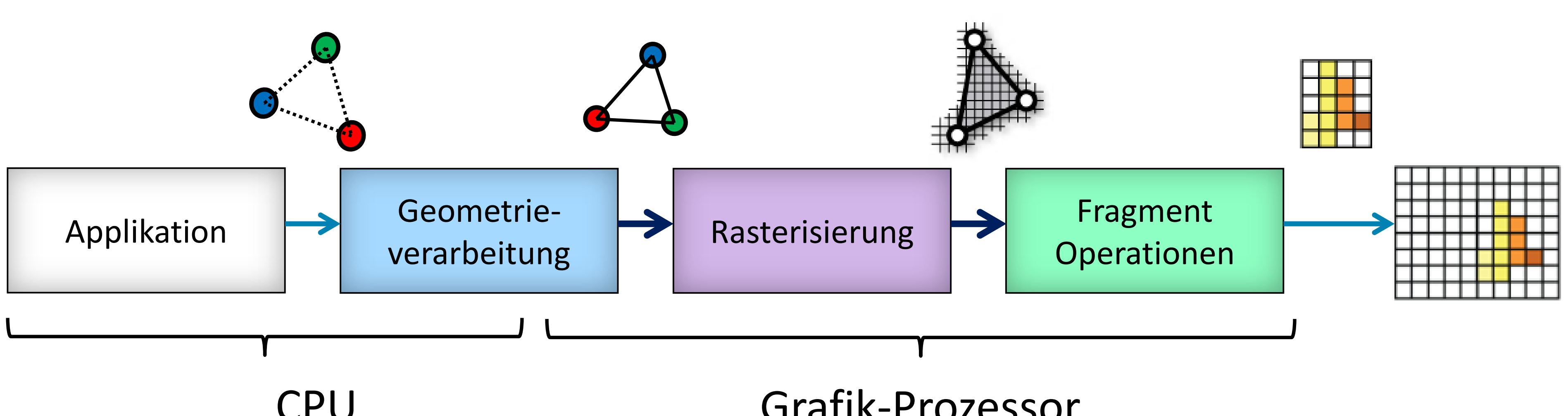
Applikation → Geometrie-verarbeitung → Rasterisierung → Fragment Operationen →

CPU

Rasterisierungs-HW

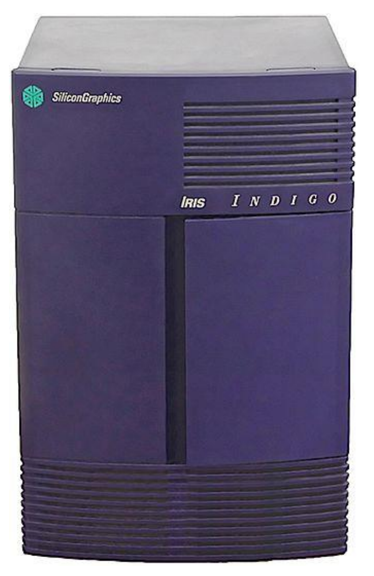
Grafik-Hardware

- SGI O_2 (1996):
  Hardware für Rasterisierung
  und Transformation
- Texture Mapping und Unified Memory
- low-cost, Nachfolger der SGI Indy



Grafik-Hardware

- SGI Indigo (1991): Hardware für Rasterisierung und Transformation → GPU (graphics processing unit)
- “one of the most capable graphics workstations of its era […] essentially peerless in the realm of hardware-accelerated three-dimensional graphics rendering” (Wikipedia)
- mit Indigos wurde bspw. Jurassic Park gerendert!
- Preis: 8000 US-$, umgerechnet 18000€ in 2024



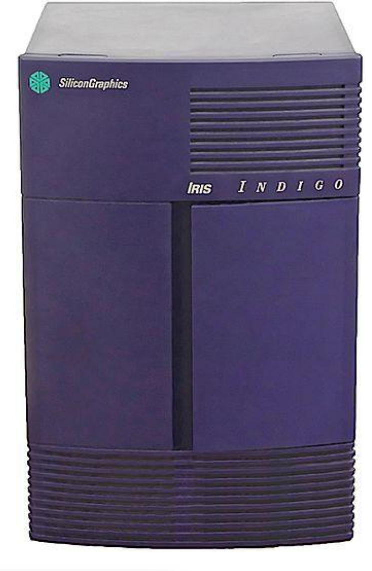
Applikation → Geometrie-verarbeitung → Rasterisierung → Fragment Operationen →

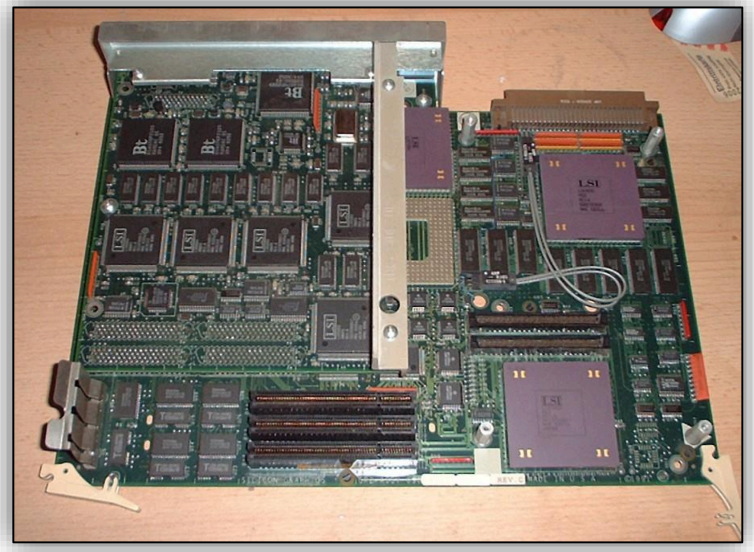
CPU: Applikation

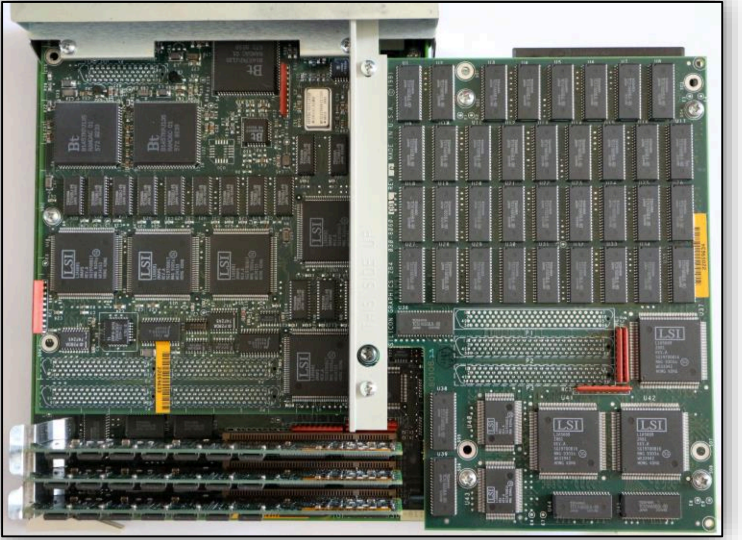
GPU: Geometrie-verarbeitung, Rasterisierung, Fragment Operationen

Grafik-Hardware

- SGI Indigo (1991): Hardware für Rasterisierung und Transformation → GPU (graphics processing unit)
- “one of the most capable graphics workstations of its era […] essentially peerless in the realm of hardware-accelerated three-dimensional graphics rendering” (Wikipedia)
- mit Indigos wurde bspw. Jurassic Park gerendert!







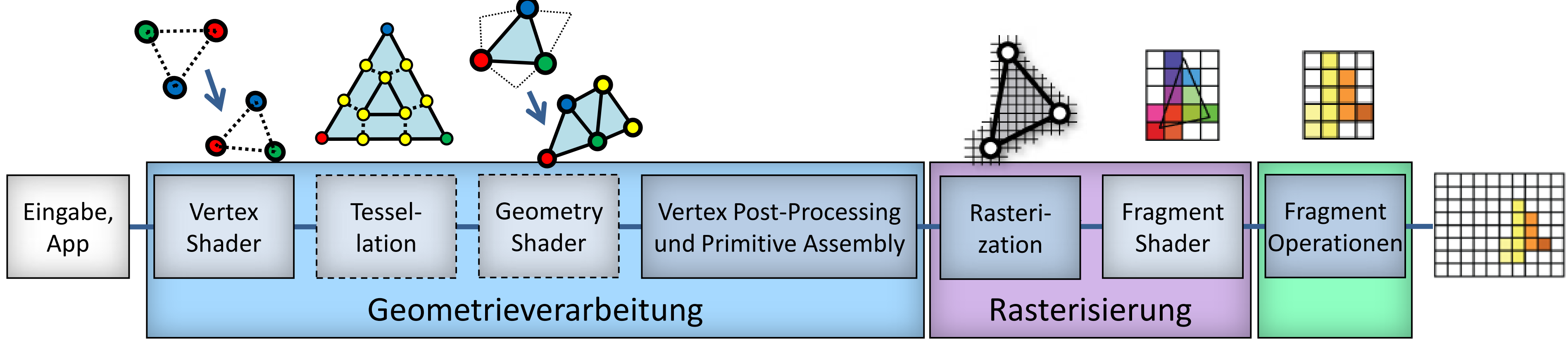
Bilder XS24 / XS24Z Grafik-Hardware:
https://bukosek.si/hardware/collection/sgi-indigo.html
https://hardware.majix.org/computers/sgi.indigo/indigo4k.shtml

Klassisches OpenGL: konfigurierbare Pipeline

```
// Initialisiere OpenGL States
glClearColor( 0.0f, 0.0f, 0.0f, 1.0f );
glEnable( GL_LIGHTING );
glEnable( GL_LIGHT0 );
glEnable( GL_DEPTH_TEST );
...

// Transformationen (inkl. Projektion)
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
gluPerspective( 65.0, (Glfloat)w / h, 1.0, 100.0 );

glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
gluLookAt( 0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0 );
...

// Rendering
glRotate( 90.0, 0.0, 1.0, 0.0 );
glBegin( GL_TRIANGLES );
  glVertex3f( 0, 0, 0 );
  glVertex3f( 1, 0, 0 );
  glVertex3f( 1, 1, 0 );
glEnd();
```

Applikation → Geometrie-verarbeitung → Rasterisierung → Fragment Operationen →

Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)



- Grafikprozessor (GPU):
  massiv-paralleler Prozessor, spezielle Hardware-Einheiten für Grafik
- teils programmierbar: Geometrieverarbeitung und Schattierung
- teils „fixed-function“: Rasterisierung selbst, Tiefentest, …
- Pipeline-Verarbeitung:
  - Daten werden von einer Stufe zur nächsten weitergegeben
  - gut geeignet für hohen Durchsatz: viel Geometrie, viele Pixel

Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)

Eingabe, App | Vertex Shader | Tessel-lation | Geometry Shader | Vertex Post-Processing und Primitive Assembly | Rasteri-zation | Fragment Shader | Fragment Operationen

Vertices, Texturen, …

„Transform Feedback“: Vertices, …

Texturen

GPU Memory

FBOs

```
uniform mat4 matrixMVP, matrixMV, matrixNrml;
uniform vec3 lightSourcePos;
in  vec4 in_position, in_normal;
out vec3 L, N; // Ausgabe des Vertex Shader
void main() {
    gl_Position = matrixMVP * in_position;
    L = lightSourcePos – vec3( matrixMV * in_position );
    N = vec3( matrixNrml * in_normal );
}
```

Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)

Eingabe, App → Vertex Shader → Tessel-lation → Geometry Shader → Vertex Post-Processing und Primitive Assembly → Rasteri-zation → **Fragment Shader** → Fragment Operationen

Vertices, Texturen, …

„Transform Feedback“: Vertices, …

Texturen

GPU Memory

FBOs

```
in vec3 L, N;                  // Eingabe aus dem Vertex Shader
out vec4 out_color;            // Ausgabe des Fragment Shader
void main() {
    float kd = max( 0.0, dot( normalize(L),
                              normalize(N) ) );
    out_color = vec4( kd );
}
```

OpenGL Selbststudium

Ziel des Selbststudiums / Anmerkungen

- OpenGL-Foliensatz enthält Teile, die Sie sich bitte zuhause ansehen
  - verschaffen Sie sich einen Eindruck, wie die Grafik-Pipeline in klassischem OpenGL umgesetzt war (konfigurierbare Verarbeitung, Shading etc.) und wie einfache Shader funktionieren
  - einiges an Funktionalität der Fixed-Function Pipeline nur angedeutet
- in der Vorlesung besprechen wir v.a. die Funktionalität und Konzepte von modernem OpenGL und Grafik-Hardware
  - aber auch bspw. Fragment Operationen
  - wichtig für die Praxisteile / Übungsaufgaben

Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)

Eingabe, App → Vertex Shader → Tessel- lation → Geometry Shader → Vertex Post-Processing und Primitive Assembly → Rasteri- zation → Fragment Shader → Fragment Operationen

Vertices, Texturen, …

„Transform Feedback“: Vertices, …

Texturen

GPU Memory

FBOs

- Eingabedaten für die Grafik-Pipeline
  - Strom von Vertices und ggf. Indizes für Dreiecksnetze
  - Daten werden im Speicher der GPU abgelegt

Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)

Eingabe, App
Vertex Shader
Tessel-lation
Geometry Shader
Vertex Post-Processing und Primitive Assembly
Rasteri-zation
Fragment Shader
Fragment Operationen

Vertices, Texturen, ...
„Transform Feedback“: Vertices, ...
Texturen
GPU Memory
FBOs

- programmierbare Stufe für Verarbeitung **eines** Vertex
  - Transformation des Vertex
  - z.B. auch Berechnung der Beleuchtung oder Texturkoordinaten pro Vertex (wird anschließend über der Dreiecksfläche interpoliert)

Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)

Eingabe, App

Vertex Shader

Tessel-lation

Geometry Shader

Vertex Post-Processing und Primitive Assembly

Rasteri-zation

Fragment Shader

Fragment Operationen

Vertices, Texturen, ...

„Transform Feedback“: Vertices, ...

Texturen

GPU Memory

FBOs

- programmierbare Stufe für Unterteilung von Primitiven (optional)
  - Tessellation Control Shader: bestimmt notwendige Unterteilung
  - eigentliche Unterteilung ist nicht programmierbar
  - Tessellation Evaluation Shader: Berechnung pro generiertem Vertex

Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)

Eingabe, App → Vertex Shader → Tessel-lation → Geometry Shader → Vertex Post-Processing und Primitive Assembly → Rasteri-zation → Fragment Shader → Fragment Operationen

Vertices, Texturen, …

„Transform Feedback“: Vertices, …

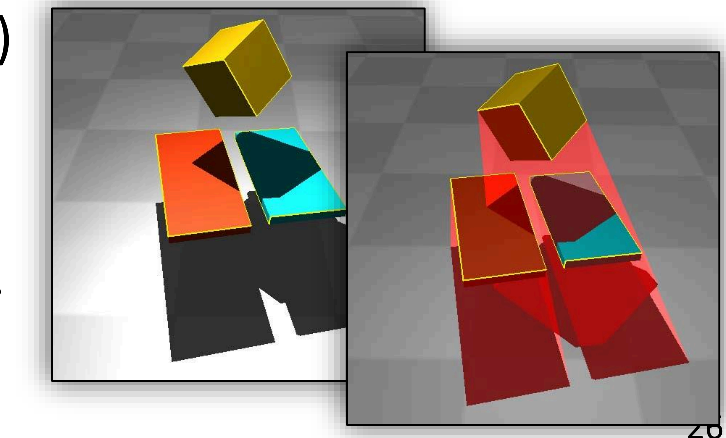
Texturen

FBOs

GPU Memory

Texturen, Vertex/Index Buffer, Konstanten, …

- programmierbare Stufe für Bearbeitung von Primitiven (optional)
  - Eingabe: **ein** Primitiv (Dreieck, Linie, Punkt)
  - Ausgabe: ein oder mehrere Primitive
  - Beispiel: Instanziierung, Dreieck → Kantenzug, Schattenvolumen, …



Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)

Eingabe, App
Vertex Shader
Tessel-lation
Geometry Shader
Vertex Post-Processing und Primitive Assembly
Rasteri-zation
Fragment Shader
Fragment Operationen

Vertices, Texturen, …
„Transform Feedback“: Vertices, …
Texturen
GPU Memory
FBOs
Texturen, Vertex/Index Buffer, Konstanten, …

- nicht-programmierbare Stufe („fixed-function“)
  - Transform Feedback: Ausgabe transformierter Vertices, kein Rendering
  - Primitive Assembly:
    Erzeugung der Primitive für die Rasterisierung aus dem Vertex-Strom
  - Clipping und perspektivische Division
  - Backface Culling

Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)

Eingabe, App
Vertex Shader
Tessel-lation
Geometry Shader
Vertex Post-Processing und Primitive Assembly
Rasteri-zation
Fragment Shader
Fragment Operationen

Vertices, Texturen, …
„Transform Feedback“: Vertices, …
Texturen
GPU Memory
FBOs
Texturen, Vertex/Index Buffer, Konstanten, …

- Rasterisierung: nicht-programmierbare Stufe („fixed-function“)
  - Eingabe: Primitive
  - Eingabe: Fragmente mit 2D-Position, interpolierten Attributen, …

Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)

Eingabe, App

Vertex Shader

Tessel-lation

Geometry Shader

Vertex Post-Processing und Primitive Assembly

Rasteri-zation

Fragment Shader

Fragment Operationen

Vertices, Texturen, …

„Transform Feedback“: Vertices, …

Texturen

GPU Memory

FBOs

Texturen, Vertex/Index Buffer, Konstanten, …

- programmierbare Stufe
  - Eingabe: Fragmente mit 2D-Position, interpolierten Attributen, …
  - Ausgabe: Farbe, Opazität, optional Tiefe, …

Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)

Eingabe, App → Vertex Shader → Tessel-lation → Geometry Shader → Vertex Post-Processing und Primitive Assembly → Rasteri-zation → Fragment Shader → Fragment Operationen

Vertices, Texturen, …

„Transform Feedback“: Vertices, …

Texturen

FBOs

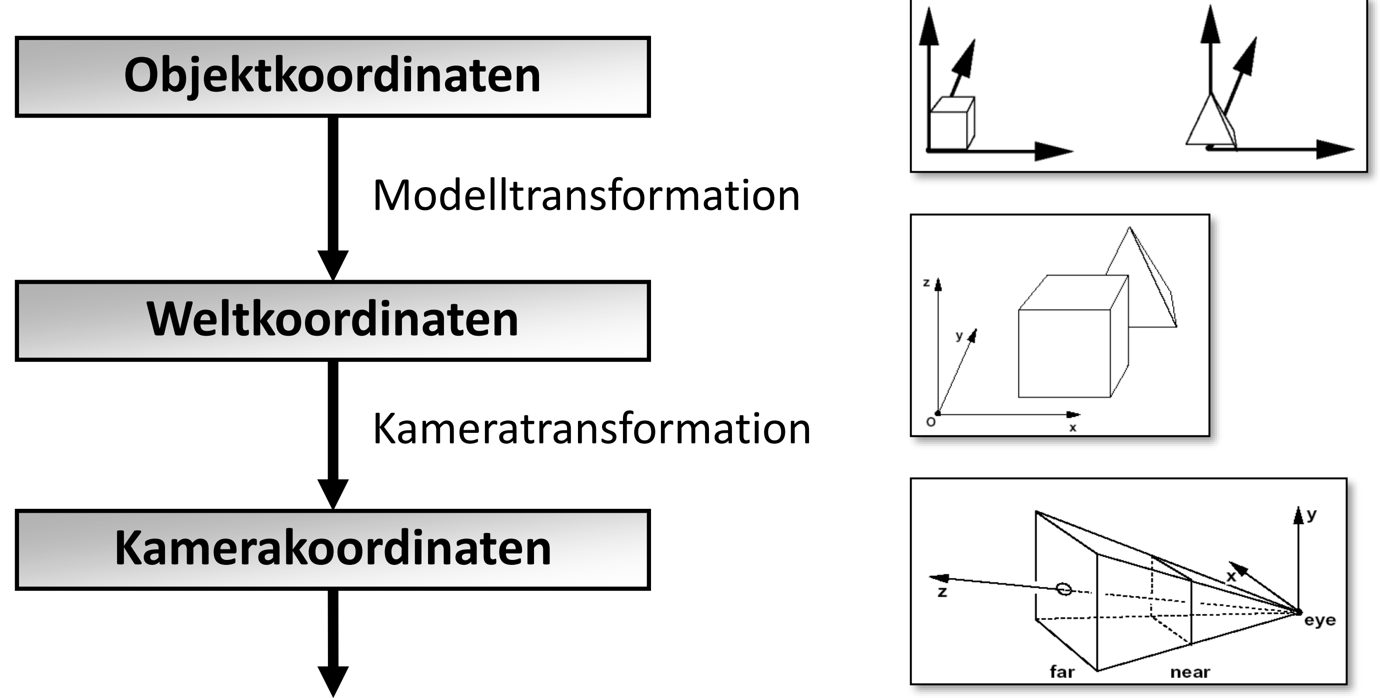
GPU Memory

Texturen, Vertex/Index Buffer, Konstanten, …

- nicht-programmierbare Stufe („fixed-function“)
  - Tiefentest mit Z-Buffer
  - Maskierung und Blending
  - Schreiben in den Frame Buffer

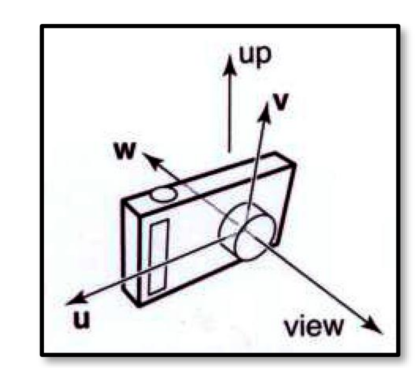
Koordinatensysteme in der CG

- …jetzt aber nochmal 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)



Kameratransformation

y
x
z
x
$-\mathbf{z}$

Kamera und Sichtpyramide

nach der Kameratransformation

(Noch mehr) Koordinatensysteme in der CG

Objektkoordinaten

↓ Modelltransformation

Weltkoordinaten

↓ Kameratransformation

Kamerakoordinaten

↓ Projektionstransformation

Clip Koordinaten

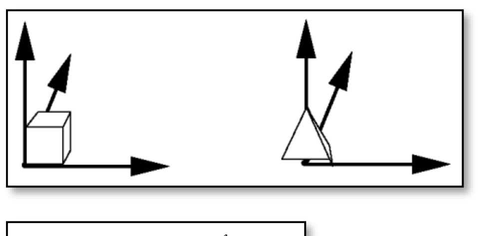
↓ Normalisierungstransf. = perspekt. Division durch w

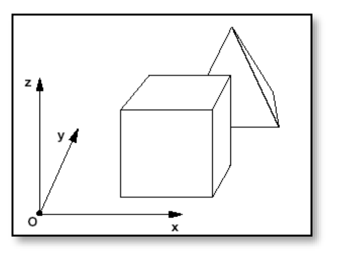
normalisierte Gerätekoord.

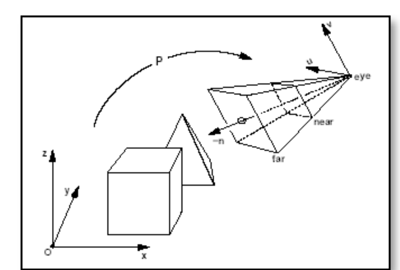
Normalized Device Coords

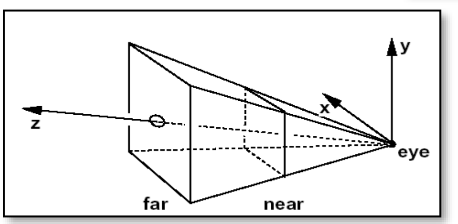
↓ Viewport Transformation

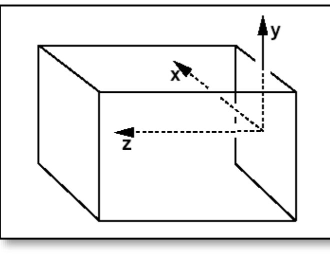
Window/Screen Coords



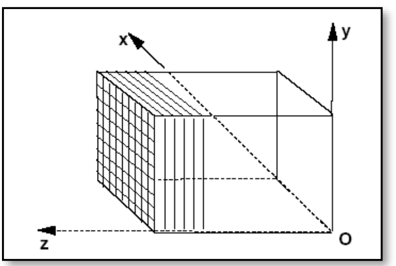






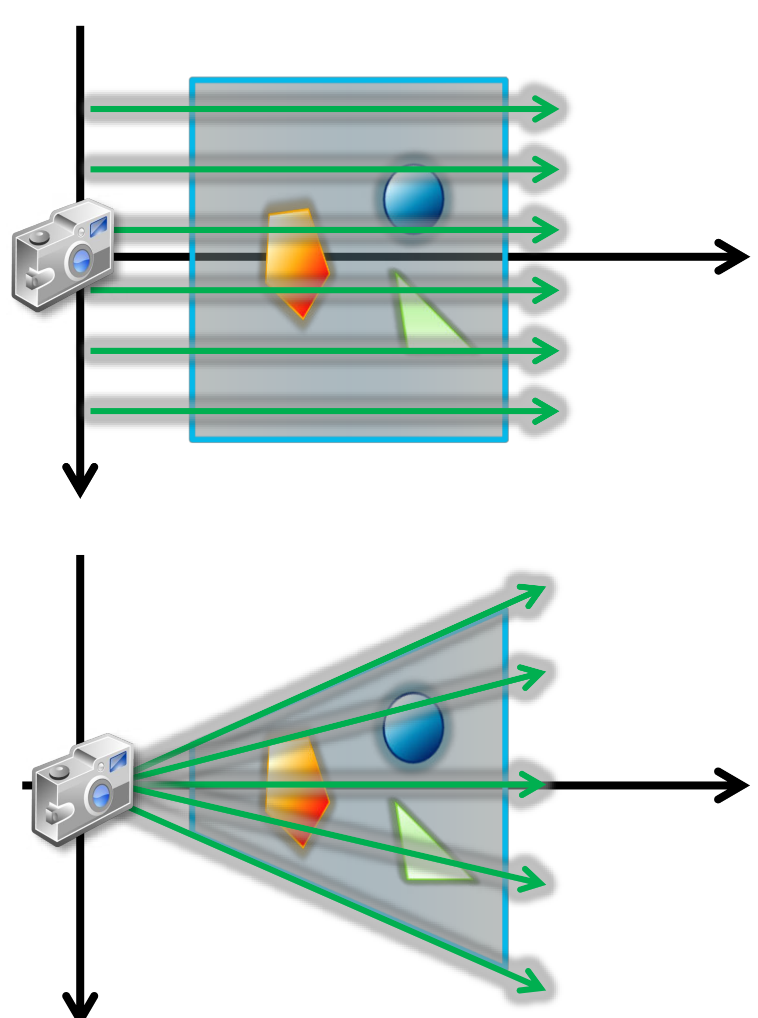


Canonical Viewing Volume



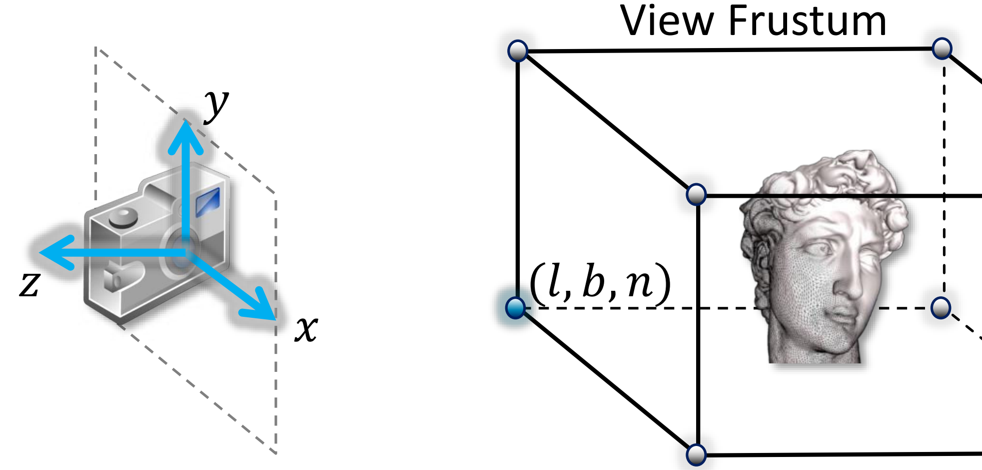
Projektion

- 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



Orthographische Kamera/Projektion

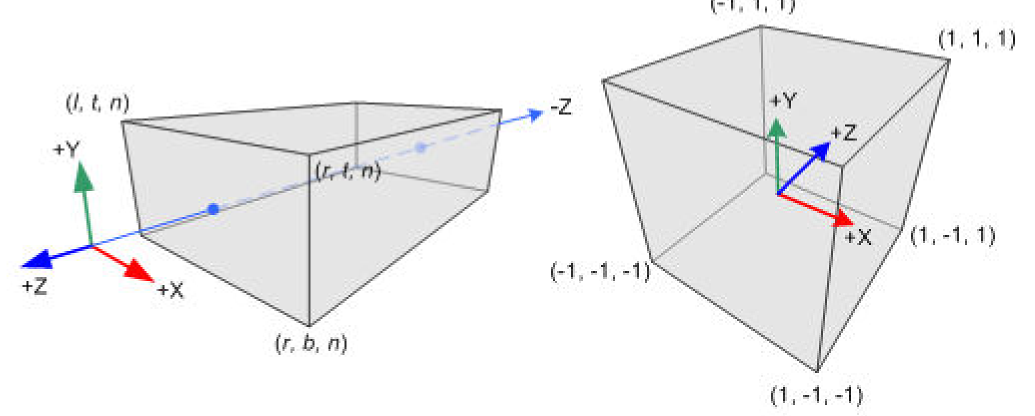
- übliche Konvention bei Rasterisierungs-APIs (in Kamerakoordinaten):
  - x-Achse zeigt nach rechts, y-Achse nach oben, Blick in neg. z-Richtung
- das Sichtvolumen (View Frustum) ist ein Quader $[l;r] \times [b;t] \times [n;f]$
  - left l, right r, bottom b, top t, near n und far f
  - Achtung: $n >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

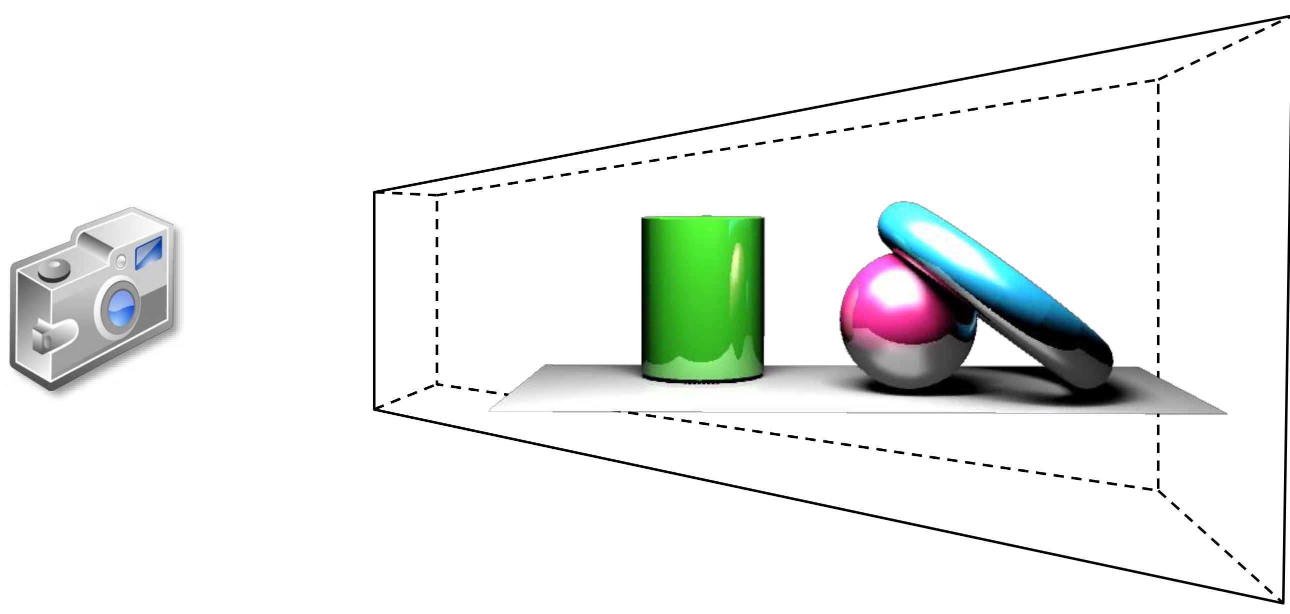
$$\mathbf{M}_{orth} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{n-f} & \frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



- 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
  $\rightarrow$ einheitliche Behandlung unterschiedlicher Projektionen

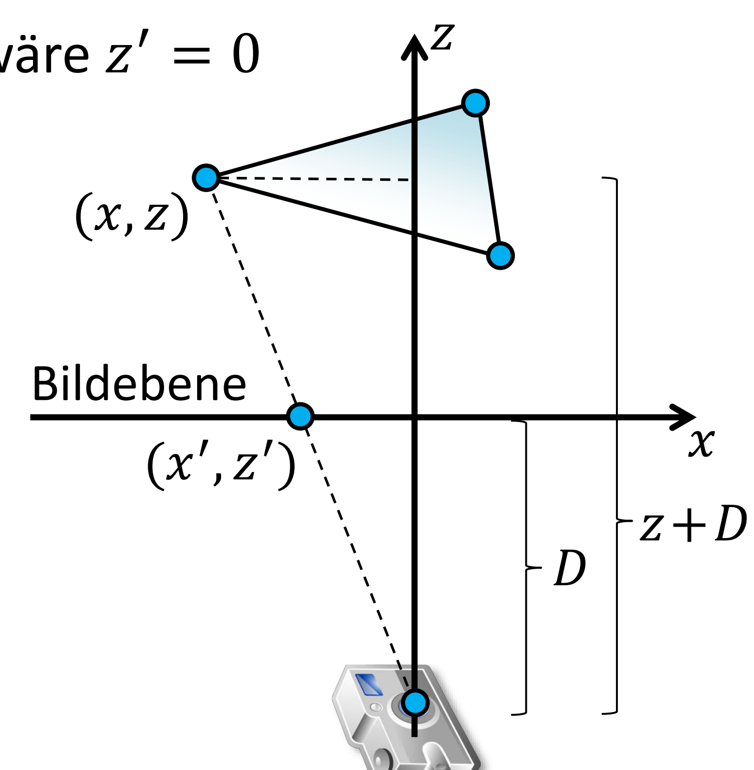


Perspektivische Projektion in 2D

- 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 $z = 0$

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

- nach einer „herkömmlichen" Projektion wäre $z' = 0$
  - wir berechnen z' analog zu x' mit $z' = \frac{z}{z/_D + 1} = \frac{z_h}{w_h}$
  - ... und heben z' als Tiefe für den späteren Z-Buffer-Test auf
  - $\begin{pmatrix} x_h \\ z_h \\ w_h \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/D & 1 \end{pmatrix} \begin{pmatrix} x \\ z \\ 1 \end{pmatrix}$

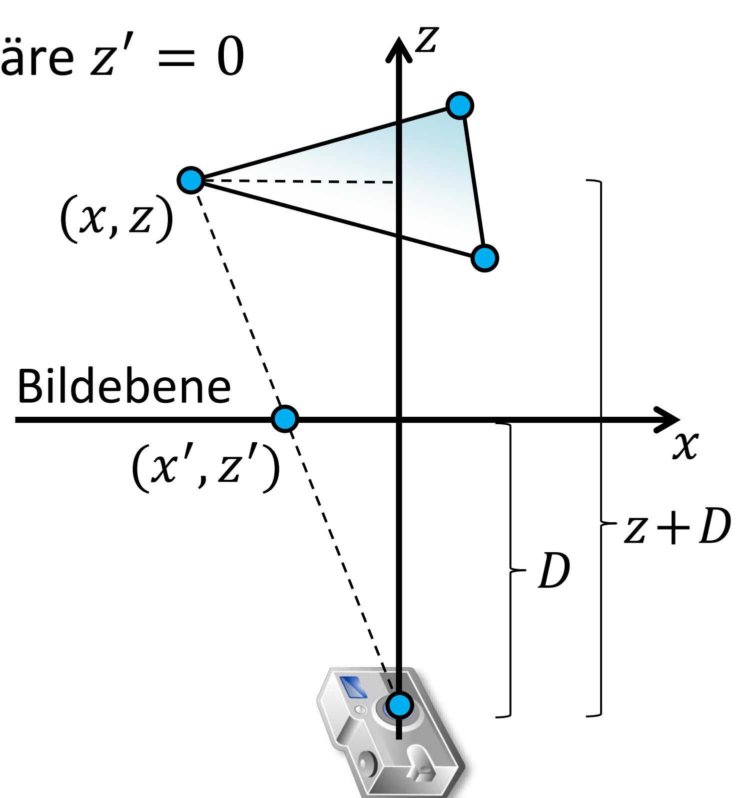


Perspektivische Projektion in 2D

- 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 $z = 0$

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

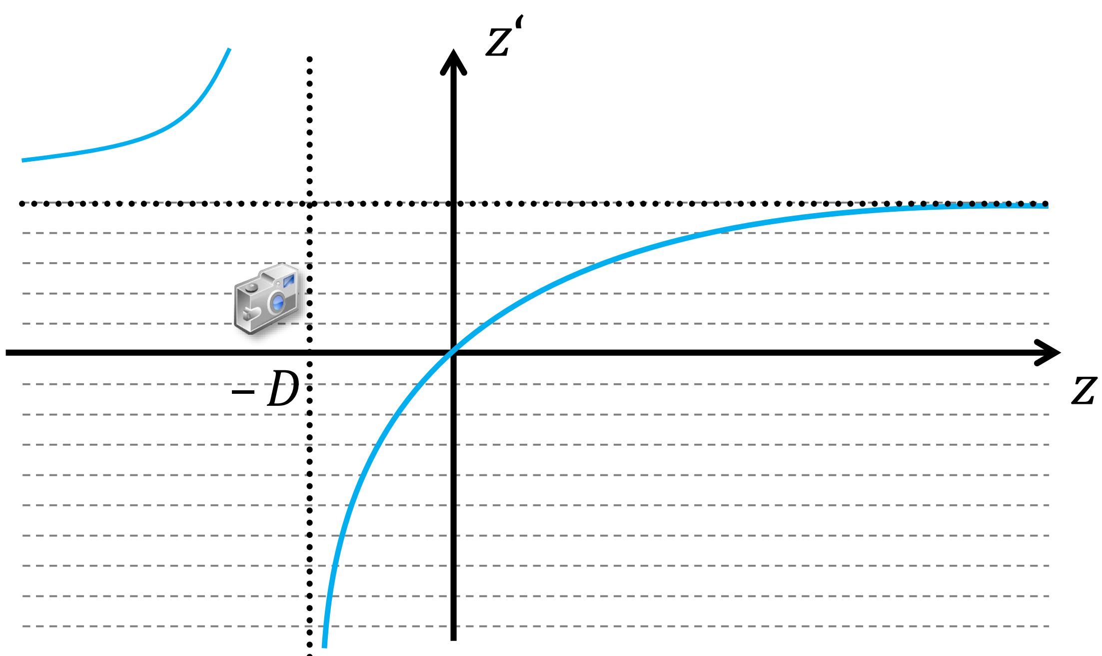
- nach einer „herkömmlichen“ Projektion wäre $z' = 0$
  - warum Tiefe $z' = \frac{z}{z/_D+1}$ und nicht einfach z für Z-Buffering?
    - Genauigkeit bei der Repräsentation der Tiefenwerte
    - perspektivisch-korrekte Attributinterpolation



Perspektivische Projektion in 2D

Betrachtung der Abbildung des Tiefenwerts $z'(z) = \frac{z_h}{w_h} = \frac{z}{z/_D+1} = \frac{zD}{z+D}$

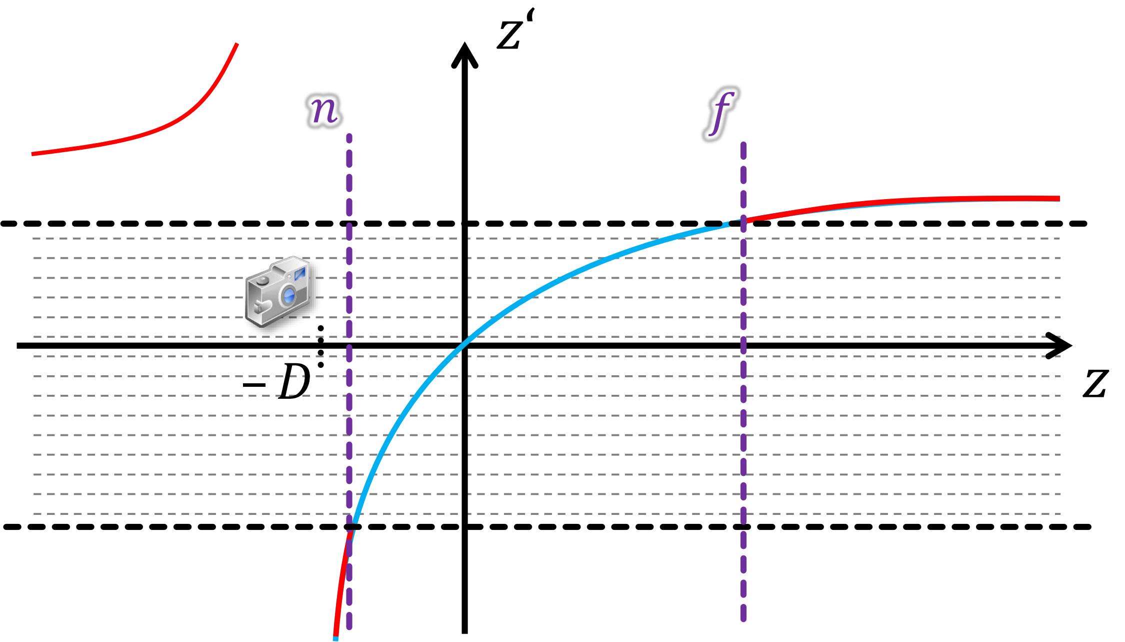
- Nicht-Linearität der z-Transformation
  - wir möchten z' diskretisieren und für den Tiefentest verwenden, um herauszufinden welches Primitiv in einem Pixel vor welchem liegt
  - **gut: höhere Genauigkeit für nahe Oberflächen**, dafür wenig Präzision für entfernte Oberflächen (große Werte)
- positive Werte für Objekte hinter der Kamera, gegen $-\infty$ für $z \to -D$



Perspektivische Projektion in 2D

Betrachtung der Abbildung des Tiefenwerts $z'(z) = \frac{z_h}{w_h} = \frac{z}{z/_D + 1} = \frac{zD}{z+D}$

- 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 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 $+1$

x

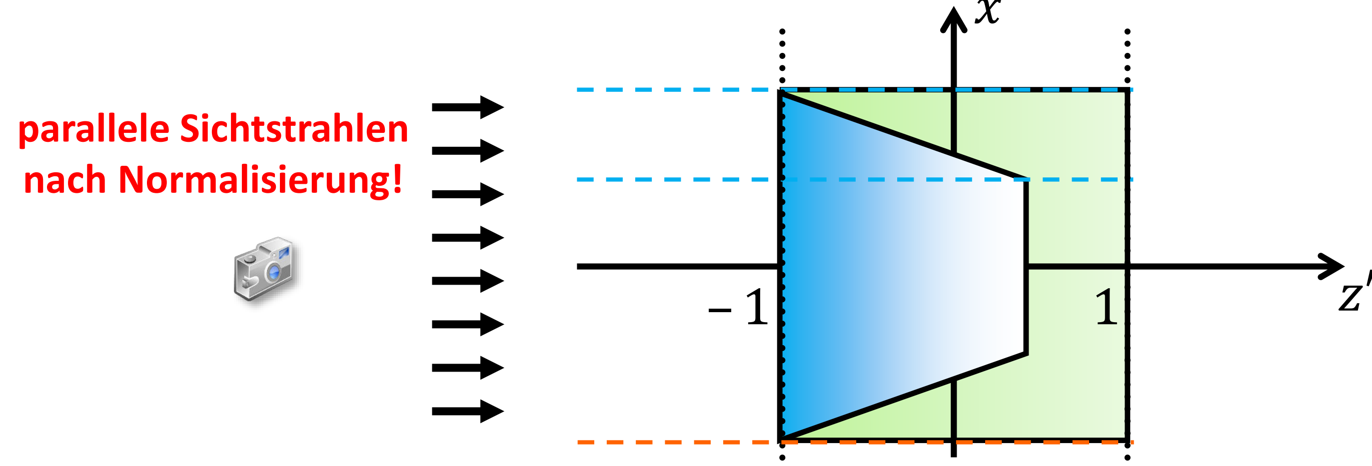
$-z$

near
(Proj.-Ebene)

far

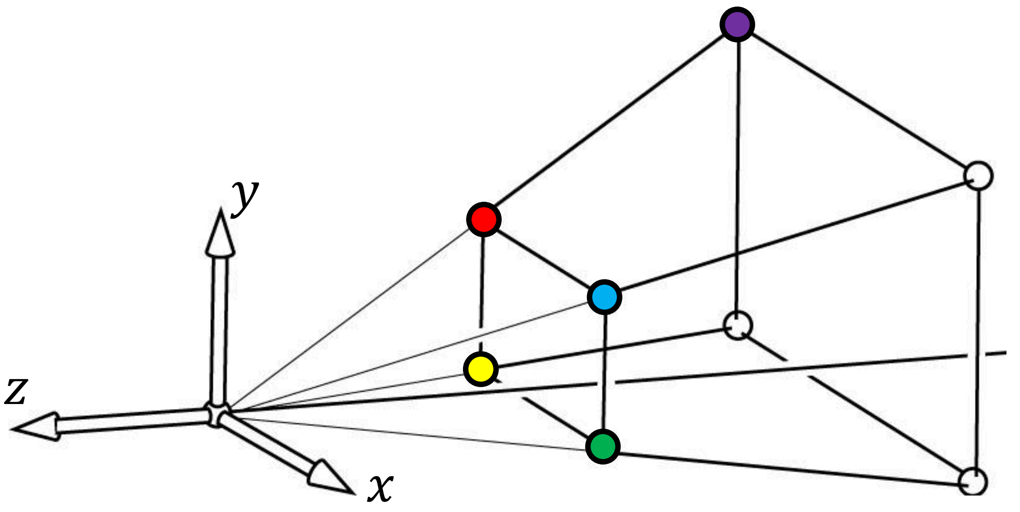
- nach der Projektionstransformation haben wir noch homogene Koordinaten!

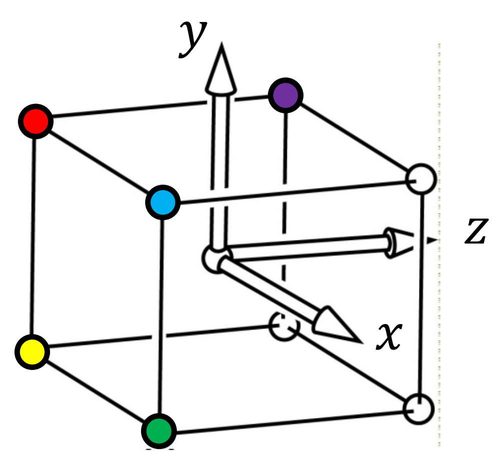
- Dehomogenisieren danach nennt man „Normalisierungstransformation“
  - ergibt sog. **Normalized Device Coordinates**, Bildschirmkoordinaten (x', y') erhalten wir danach durch Weglassen der Tiefe



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

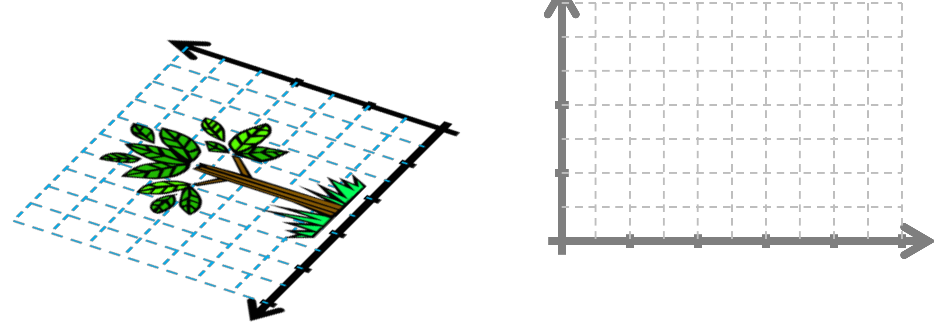




- eine oft gebrauchte Konvention (z.B. auch bei OpenGL) ist
  - Kamera befindet sich im Ursprung
  - Blickrichtung entlang der negativen z-Achse
- wie viele Punkte bestimmen eindeutig die Projektionstransformation?

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

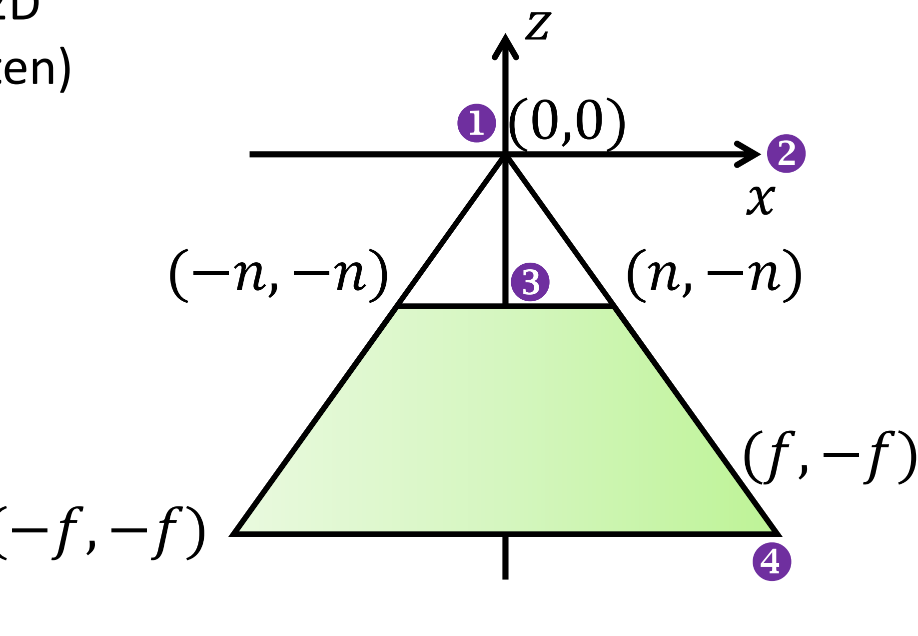
- Projektion in 2D: $3 \times 3 - 1 = 8$ Elemente, also 4 Punkte im $\mathbb{R}^2$

Bestimmung der Projektionsmatrix (in 2D)

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

- betrachte Abbildung von 4 Punkten in 2D (Faktoren k_i wg. homogenen Koordinaten)
  - Kameraposition nach $-\infty$

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



  - x-Richtung beibehalten

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

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

$$❸ \quad \mathbf{M}\begin{pmatrix}0\\-n\\1\end{pmatrix}_h = k_3\begin{pmatrix}0\\-1\\1\end{pmatrix}_h \qquad ❹ \quad \mathbf{M}\begin{pmatrix}f\\-f\\1\end{pmatrix}_h = k_4\begin{pmatrix}1\\1\\1\end{pmatrix}_h$$

z'

(−1,1) ❹

❷ x'

❸

(1, −1)

❶

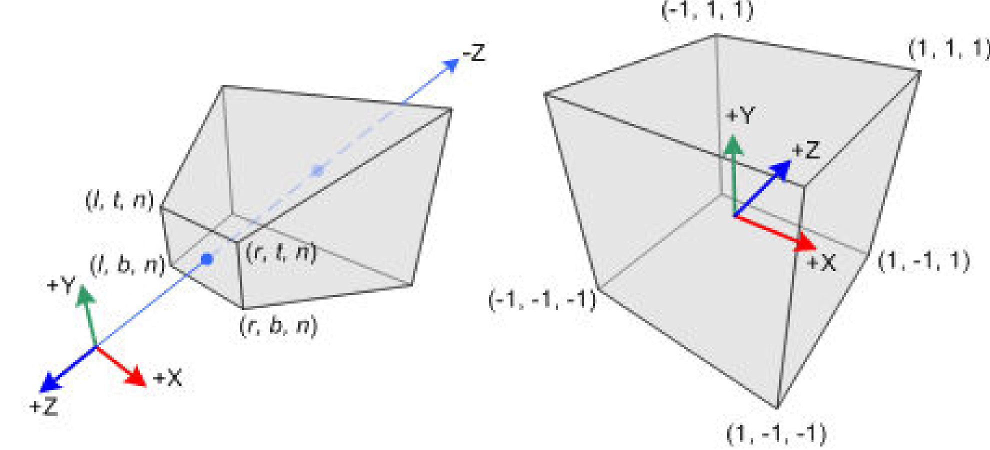
Bestimmung der Projektionsmatrix (in 2D)

- 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 /

Zusammenfassung: Projektionstransformation

- ges. Abbildung des Sichtbereichs auf den Einheitswürfel $[-1;1]^3$
  - einheitliche Handhabung für unterschiedliche Projektionen
  - gleichzeitig Abbildung der Tiefe/z (Genauigkeit, perspekt. Korrektur)
  - 2 Schritte: Projektions- und Normalisierungstransformation

y z x (r,t,f) (l,b,n) y z x

x $-z$ near (Proj.-Ebene) far x' -1 1 z'

parallele Sichtstrahlen nach Normalisierung!

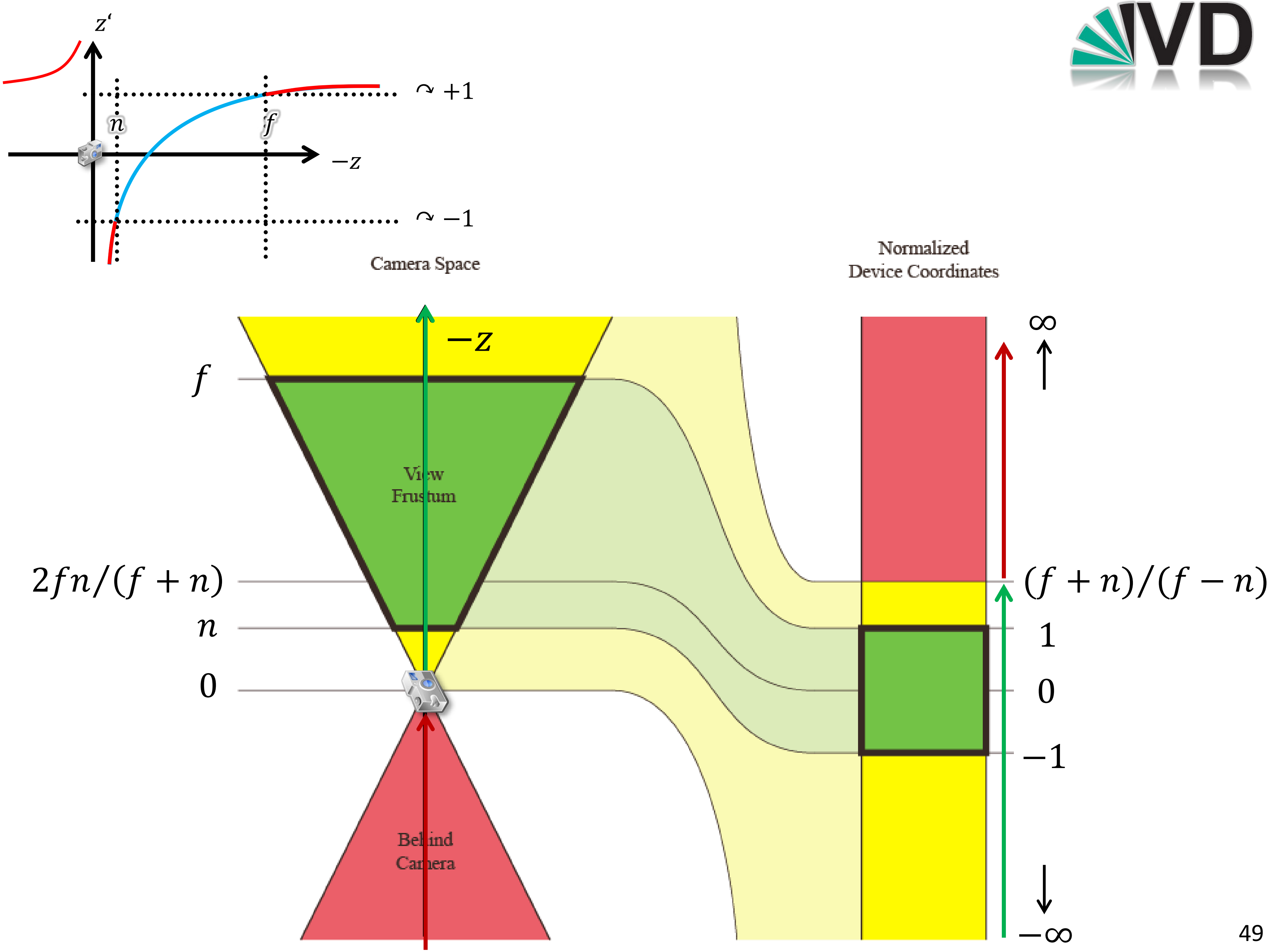
Perspektivische Projektion

Betrachtung der Abbildung des Tiefenwerts $z'(z)$

- Nicht-Linearität der z-Transformation (nur bei perspektivischer Kamera): höhere Genauigkeit für nahe Oberflächen
- beschränke Tiefenbereich Near und Far (Clipping) Planes, um die problematischen roten Bereiche wegzuschneiden
  - wähle $|n|$ immer so groß wie möglich!

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

Wichtig: das ist die Tiefe nach Projektions- und Normalisierungstransformation (also Normalized Device Coordinates)



IVD
z'
n
f
$-z$
$\sim +1$
~ -1
Camera Space
Normalized
Device Coordinates
$-z$
f
View
Frustum
$2fn/(f+n)$
n
0
Behind
Camera
∞
$(f+n)/(f-n)$
1
0
-1
$-\infty$

Perspektivische Projektion

Normalisierungstransformation und Clipping

- Abbildung der Tiefe kann beim Clipping Schwierigkeiten bereiten

Camera Space

Normalized Device Coordinates

f

$2fn/(f+n)$

n

0

View Frustum

Behind Camera

∞

$(f+n)/(f-n)$

1

0

-1

$-\infty$

Perspektivische Projektion

Normalisierungstransformation und Clipping

- Abbildung der Tiefe bereitet beim Clipping Schwierigkeiten

Camera Space

Normalized
Device Coordinates

$-z$

f

View
Frustum

$2fn/(f+n)$

n

0

Behind
Camera

was würde der
Clipping-Alg.
zurückliefern?

?

∞

$(f+n)/(f-n)$

1

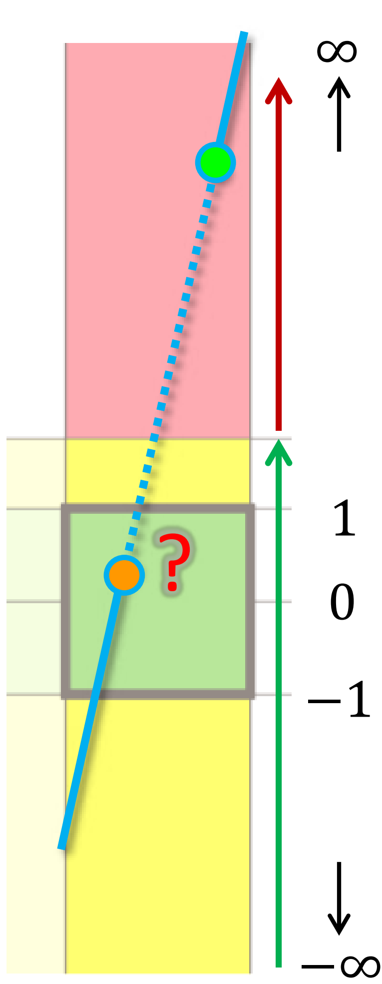
0

-1

$-\infty$

Clipping und Projektionen

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



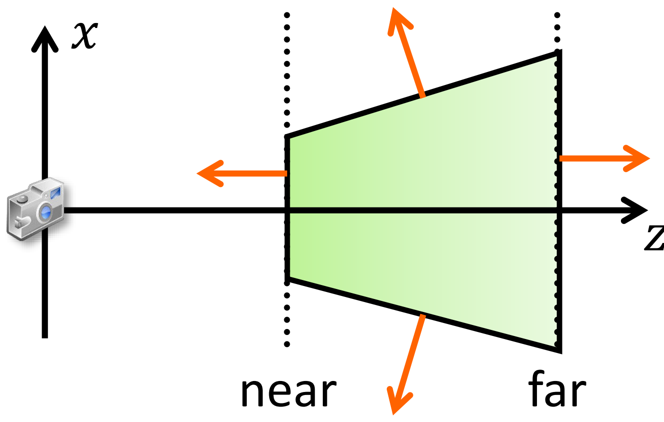
Perspektivische Projektion in 2D

Normalisierungstransformation und Clipping

- Abbildung der Tiefe bereitet beim Clipping Schwierigkeiten

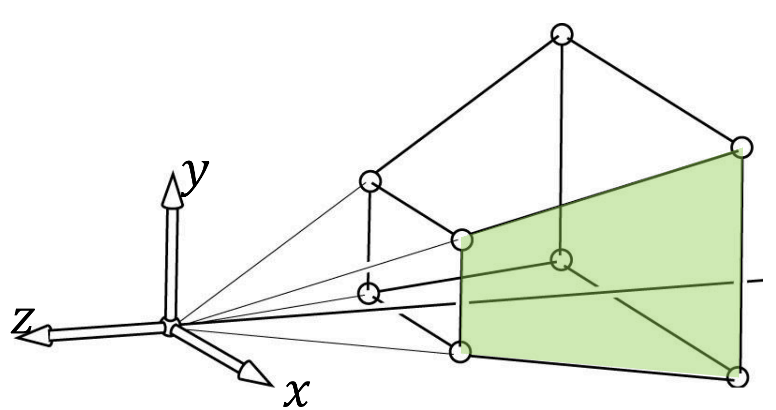
Clipping und Projektionen

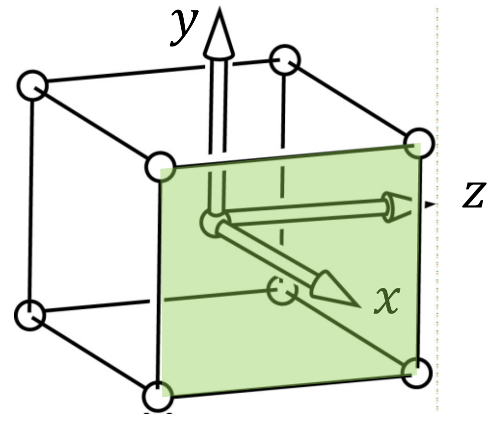
- Option 1: Clipping gegen das Frustum in Welt- oder Kamerakoordinaten **vor** der Projektionstransformation
  - stelle 6 Ebenengleichungen auf (geht direkt aus Projektionsmatrix)
  - konvexer Körper → Clipping wie bisher/wie in der Übung
  - funktioniert, aber es geht effizienter!



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

$w + x = 0$

Camera Space

$w - x = 0$

Normalized Device Coordinates

∞

f

View Frustum

$2fn/(f+n)$

$(f+n)/(f-n)$

n

1

0

0

-1

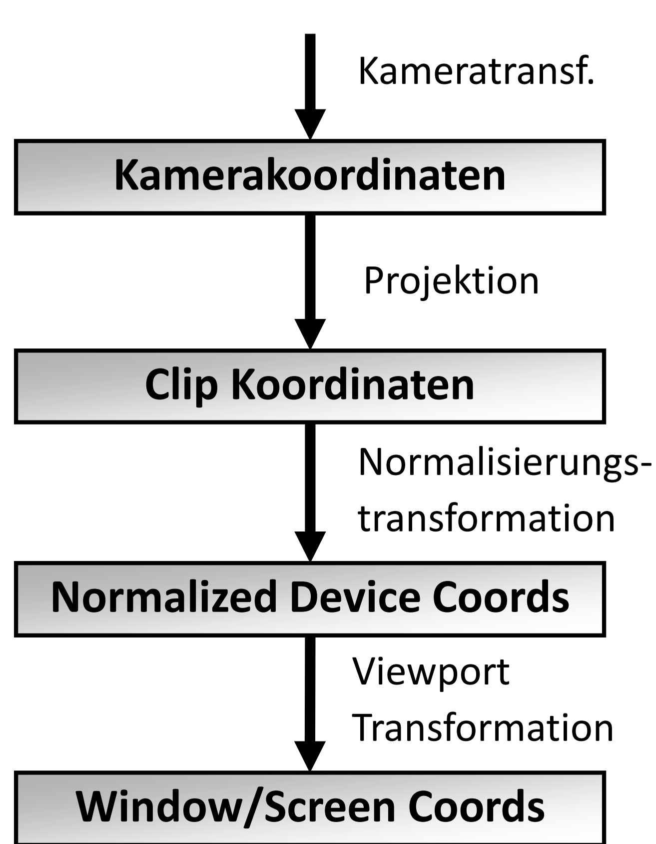
Behind Camera

$-\infty$

Clipping in homogenen Koordinaten

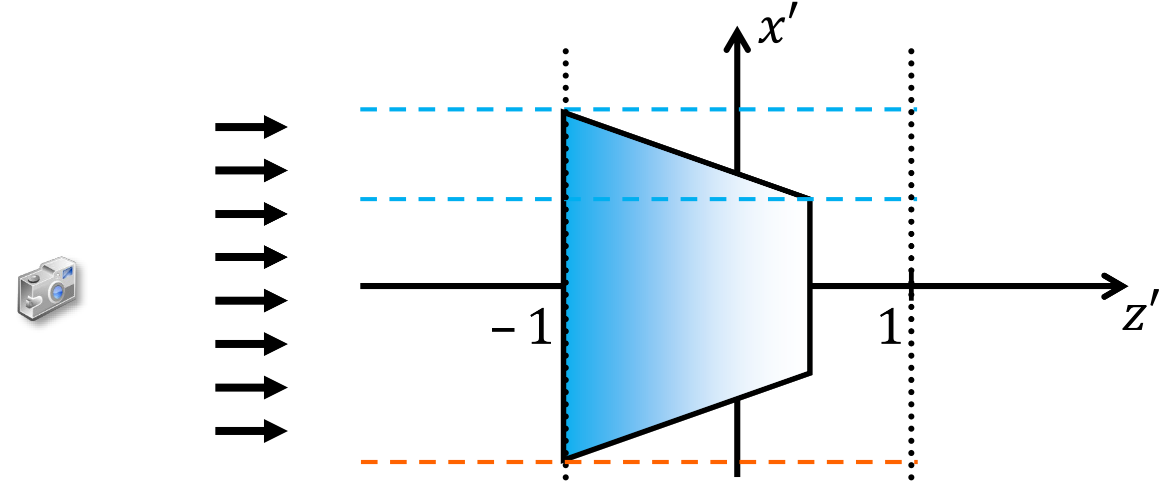
- Option 2: Clipping in homogenen Clip-Koordinaten
  - nach der Projektionstransformation, aber vor dem Dehomogenisieren

- Vorgehen (so macht man es!):
  - Projektionstransformation
  - Clipping in homogenen Koordinaten
  - dann perspektivische Division durch w $\rightarrow$ ergibt Normalized Device Coords. (Einheitswürfel)
  - dann Projektion auf Bildschirmkoordinaten und Viewport-Transformation (Abb. von $x, y \in [-1; 1]$ auf Bildschirmauflösung $\text{Breite} \times \text{Höhe}$)



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:

$$x_w = {}^1/_2\,(x' + 1) \cdot width + x \quad \text{bzw.} \quad y_w = {}^1/_2\,(y' + 1) \cdot height + y$$

Koordinatensysteme in der CG

Objektkoordinaten

↓ Modelltransformation

Weltkoordinaten

↓ Kameratransformation

Kamerakoordinaten

↓ Projektionstransformation

Clip Koordinaten

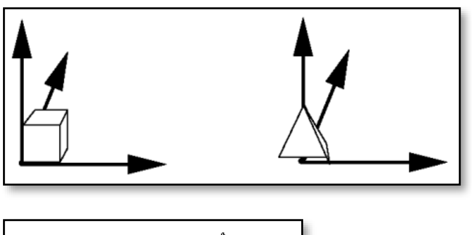
↓ Normalisierungstransf. = perspekt. Division durch w

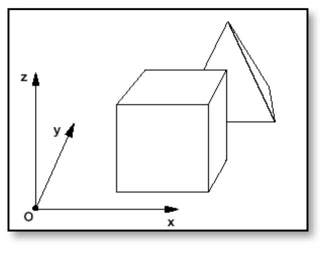
normalisierte Gerätekoord.

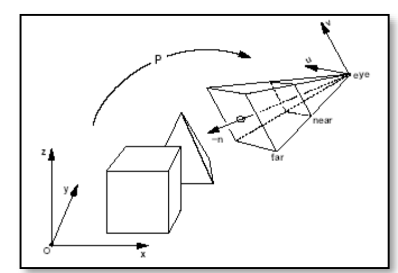
Normalized Device Coords

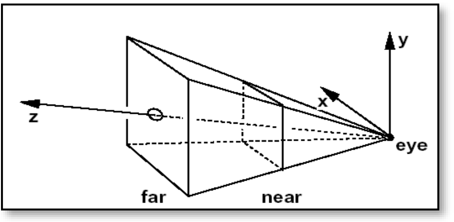
↓ Viewport Transformation

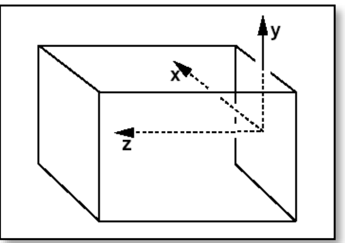
Window/Screen Coords

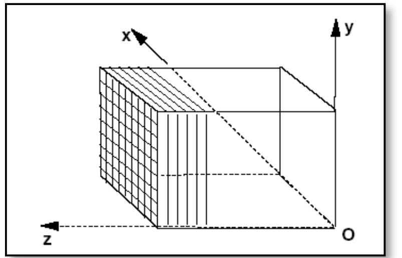












Perspektivisch-korrekte Attribut-Interpolation

- 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!

$\binom{s}{t} = \binom{0}{0}$ $\binom{1}{0}$

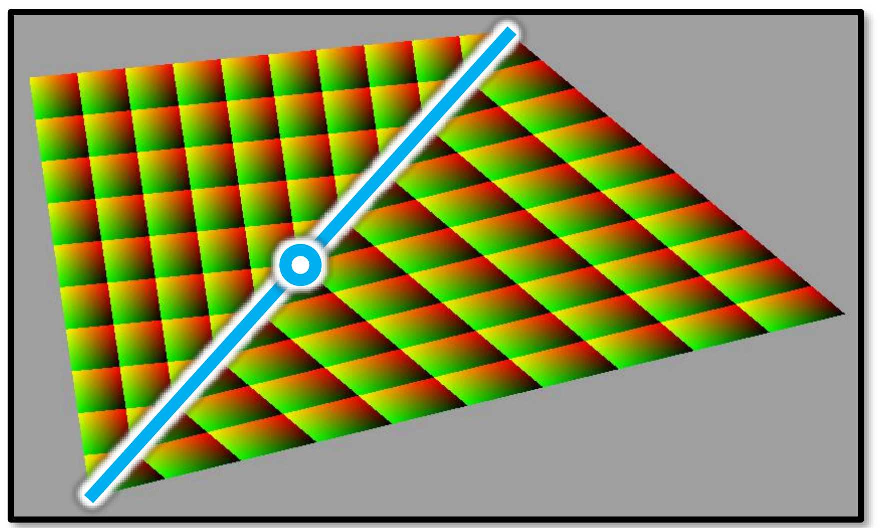
$\binom{0}{0.5}$ $\binom{1}{0.5}$ $\binom{1}{0.625}$

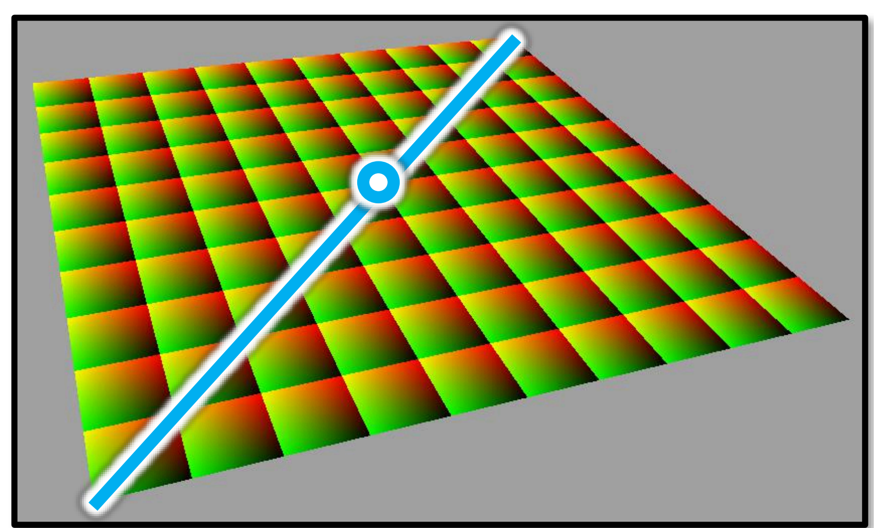
$\binom{0}{1}$ affine Interpolation $\binom{1}{1}$ perspektivisch-korrekte Interpolation

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), gilt aber ebenso für alle anderen **Attribute** und **Tiefe**

- Bsp.: perspektivisch-korrekte Interpolation (rechts) und Interpolation im Bildraum (links), das Quadrat besteht aus 2 Dreiecken





Perspektivisch-korrekte Attribut-Interpolation

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**:

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

Perspektivisch-korrekte Attribut-Interpolation

Veranschaulichung in 2D:

- lineare Interpolation entlang der Kante $\overline{\mathbf{v_1 v_2}}$ **vor der Projektion**:

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

Perspektivisch-korrekte Attribut-Interpolation

- im Allgemeinen **gilt natürlich**

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

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

Perspektivisch-korrekte Attribut-Interpolation

- Kamerakoordinaten für Pixelposition x': $x = x' \cdot z$ (1)
- Punkt auf der Kante: $x = Az + B$ (2)
- setzt man (1)=(2), so erkennt man Zusammenhang zwischen x' und z
- … und damit die perspektivisch-korrekt interpolierte Tiefe z
  - für den Tiefentest eines jeden Pixels (Z-Buffer) und
  - Attributinterpolation (zur Erinnerung: $x = x' \cdot z$ und affine Abb.)

Bildebene

$\mathbf{v_2} = (x_2, z_2)$

projizierte Kante

$\mathbf{p_2}$

Dreieckskante

$\mathbf{p_1} = (x_1', z_1')$

$\mathbf{v_1} = (x_1, z_1)$

$d = 1$

z_1

z_2

Perspektivisch-korrekte Attribut-Interpolation

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

Perspektivisch-korrekte Attribut-Interpolation

Und jetzt der Trick…

- $x' \cdot z = Az + B \quad \Leftrightarrow \quad z = \frac{B}{x' - A}$
- 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 $x = \frac{x\prime}{^1/_z}$
- die Attribute könnten dann über baryzentrische Koordinaten interpoliert werden (berechnet mittels x bzgl. der Vertices in Kamerakoord.)
- ebenso darf man gleich $^s/_z$, $^t/_z$, … (berechnet an den Vertices) im Bildraum interpolieren und erhält durch Division mit dem interpolierten $^1/_z$ direkt die perspektivisch-korrekt interpolierten Attribute

Perspektivisch-korrekte Attribut-Interpolation

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 $^1/_w$, …

Objekt-affine Räume

Bild-affine Räume

Objektraum

Texturraum $(s, t, 1)$

Division durch $^1/_w$

Homogener Texturraum $(^s/_w, ^t/_w, ^1/_w)$

Division durch w

Weltkoord. (x, y, z, w)

Homogener Bildraum (xw, yw, zw, w)

Bildraum $(x', y', z', 1)$

„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 Beschränkung der Schattierungsberechnung auf sichtbare Flächen (Deferred Shading, Early-Z-Culling, …)
- 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
- Forschung: effizientes Path Tracing, Rauschentfernung, …