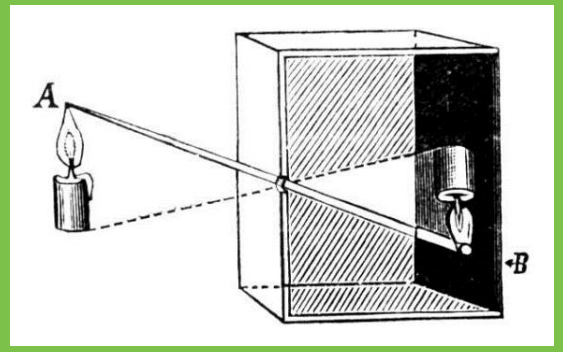
Visual Computing – Kamera & Graphik-Pipeline

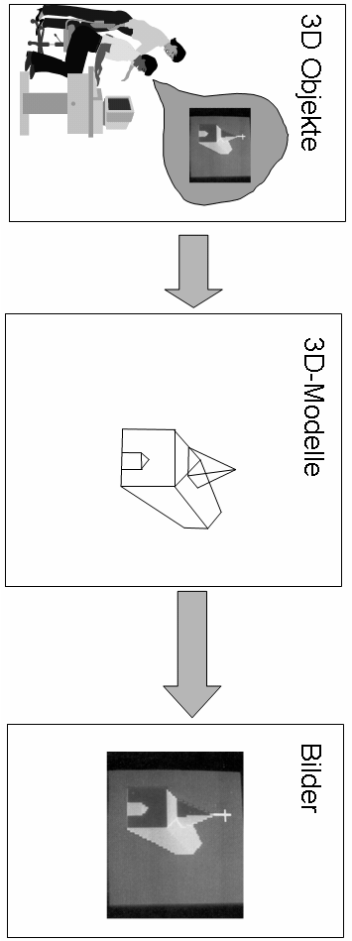


Yvonne Jung

3D-Graphik-Pipeline

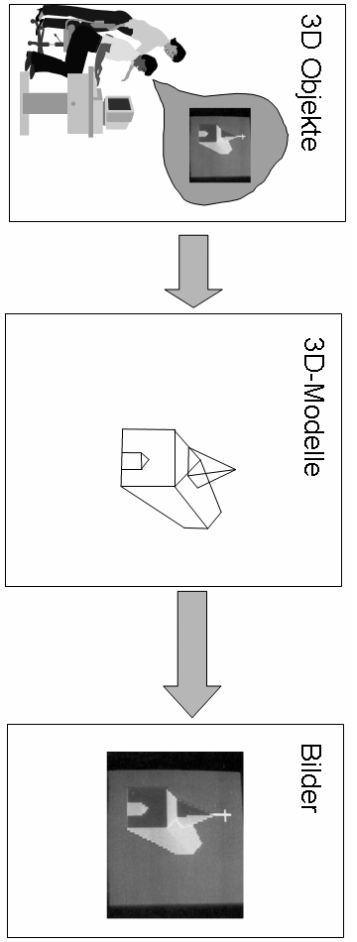
- Anwendungsebene
  - Repräsentation von 3D-Daten
    - Graphische Primitive
    - Lage im 3D-Raum
    - Räumliche Datenstrukturen
  - Animationen
- Geometrieverarbeitung
- Rasterisierung
- Ausgabe

Typischerweise in sog. Szenengraph o.ä. organisiert



3D-Graphik-Pipeline

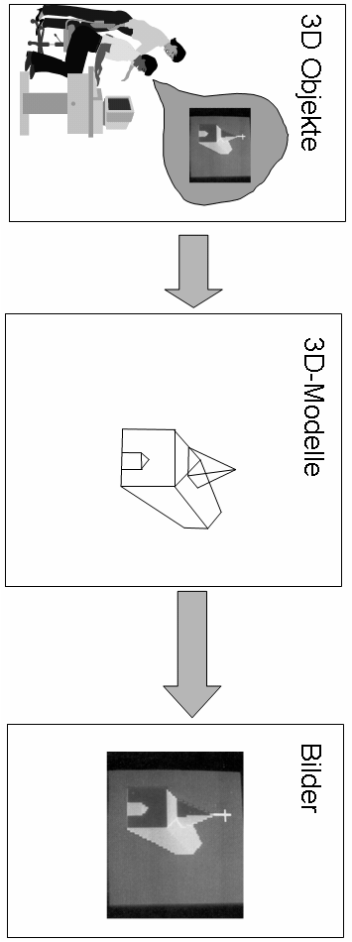
- Anwendungsebene
- Geometrieverarbeitung
  - 3D-Transformationen
    - Von Objekt- zu Kamerakoordinaten
  - Projektion
  - Clipping
  - Beleuchtungssimulation
- Rasterisierung
- Ausgabe



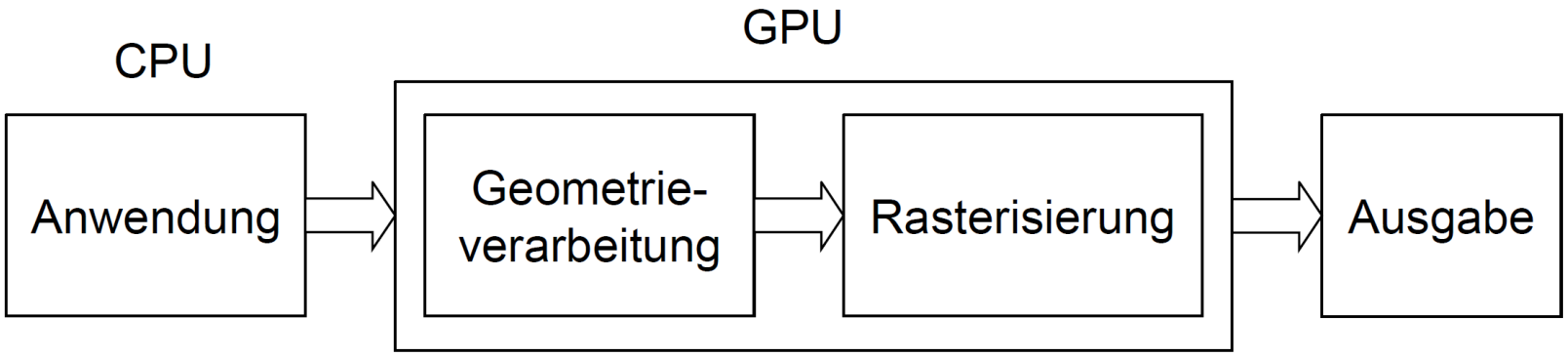
3D-Graphik-Pipeline

- Anwendungsebene
- Geometrieverarbeitung
- Rasterisierung
  - Scankonvertierung
  - Farbwertinterpolation
  - Texturierung
  - Tiefentest/Verdeckung
- Ausgabe
  - Monitor u.ä.

Geschieht auf Graphikkarte

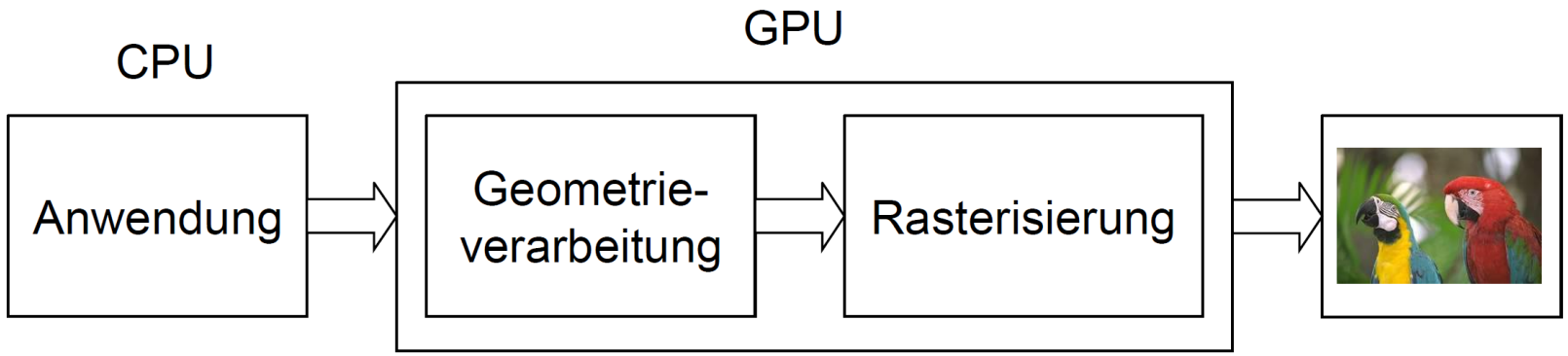


3D-Graphik-Pipeline



- Drei Hauptstufen: Anwendung (CPU), Geometrie, Rasterisierung (GPU)
- **Anwendungsebene:** Repräsentation der 3D-Daten (z.B. als Szenengraph), Kollisionsbehandlung, Viewfrustum-Culling, …
- **Geometriestufe:** Modell-/ Kameratransformation, Projektion, Clipping, …
- **Rasterisierung:** Triangle-Setup, Scankonvertierung, Pixel-Shading (z.B. Texturierung), Per-Fragment Operationen (z.B. Depth-Test, Blending)

3D-Graphik-Pipeline



- Eingabe grafischer Daten
- Repräsentation von 3D Daten (Primitive,Transformationen, räumliche Datenstrukturen)

- Modell-Transformation
- Simulation der Beleuchtung
- Perspektivische Transformation
- Clipping
- Culling
- Projektion

- Rasterisierung der Primitive nacheinander:
  - Scan-Konvertierung
  - Verdeckungsrechnung
  - Farbwertinterpolation

- Speichern des Bildes
- Display
- Hardcopy

- Application Code
  - Wichtige Eingangsdaten: Kamera, Lichter, Drawables
- High-Level Graphiksystem behandelt Drawables (Geometrie mit Material und Transformationen) sowie Frustum Culling
  - Dabei Aufbau einer „Draw List“ (wird an Low-Level Graphiksystem übergeben)
- Anzeige des fertigen Bildes

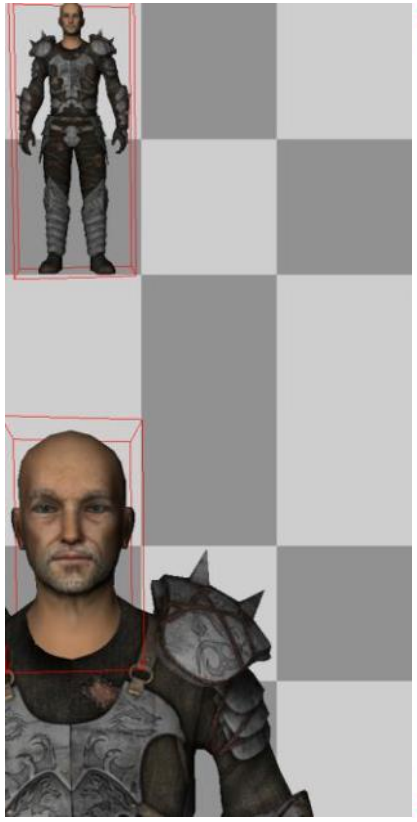
Virtuelle Kamera

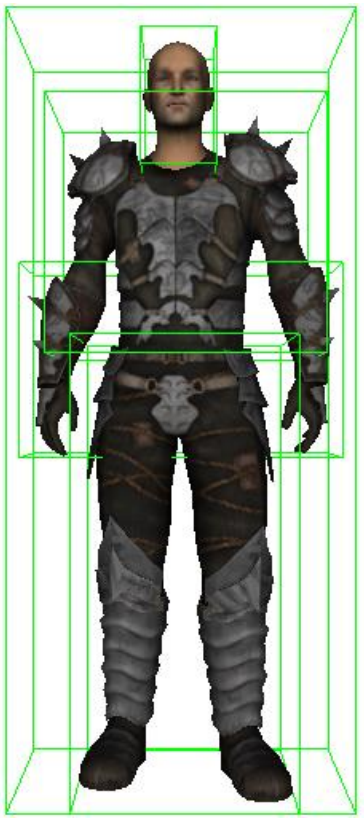
h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Kamera def. Blickpunkt (z.B. First Person Shooter)

Third Person View

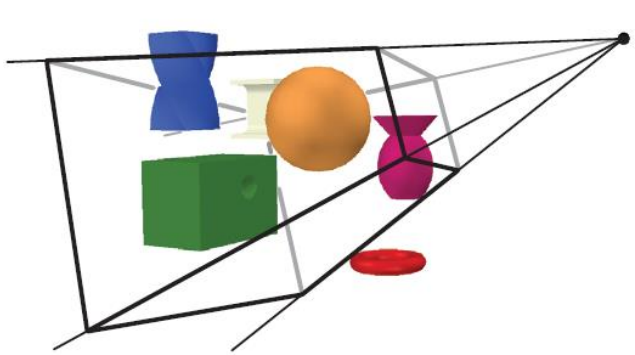
Cinematographische Ansätze

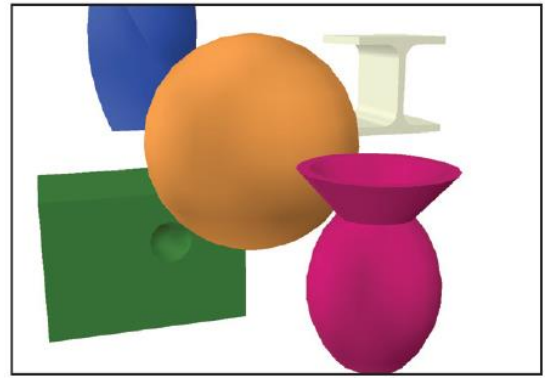




Linkes und mittleres Bild entnommen aus "Left for Dead 2"

Problemstellung



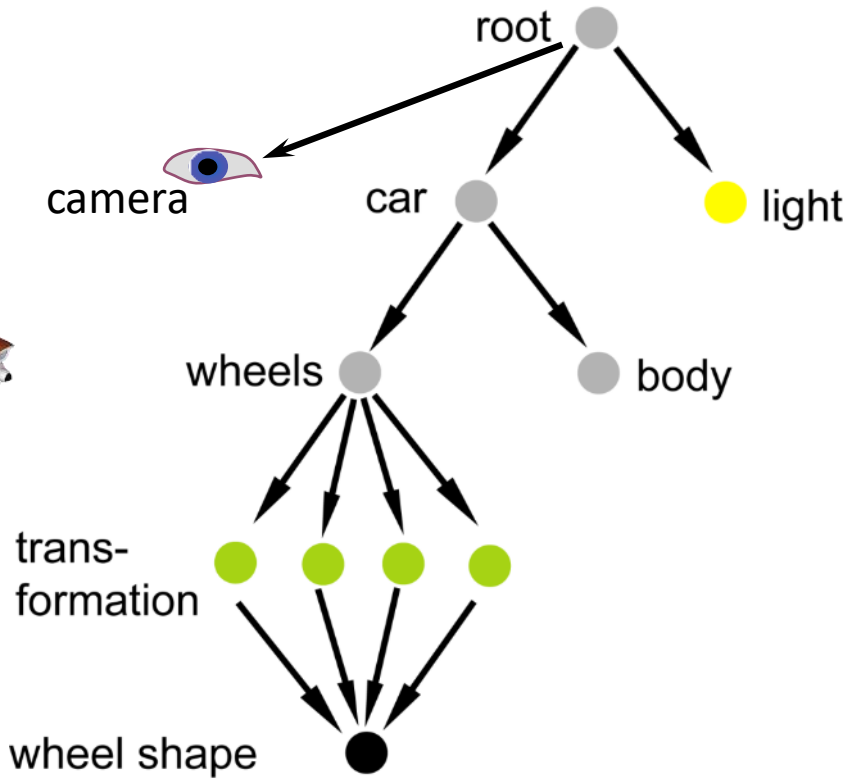


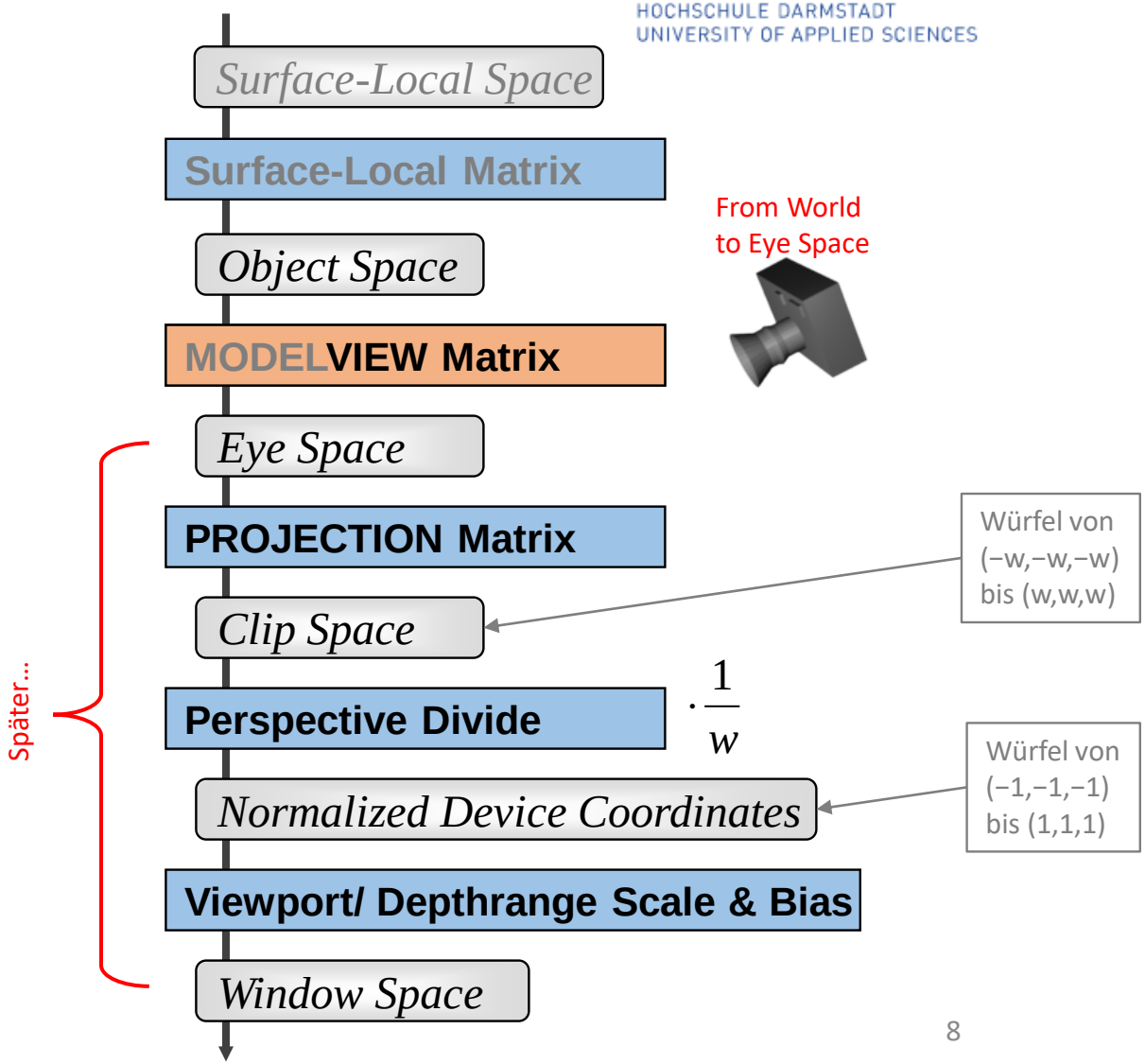
- 3D-Welt auf 2D-Bildebene abbilden
  - Vor.: 3D-Objekte bereits in Weltkoordinaten transformiert (mit Model-Matrix)
- Lösung: Kameratransformationen
  - Zunächst Transformation von Welt- in Kamerakoordinaten (mit View-Matrix)
  - Dann Projektion auf Bildebene (mit Transformation auf gewählten Viewport)
    - Erst von View/Eye Space (d.h. Kamerakoordinaten) zu Clip Space (mit Projection-Matrix)
      - Beinhaltet neben Projektion i.d.R. perspektivische Abbildung
      - Daher keine affine Abbildung (Parallelität bleibt i.A. nicht erhalten)
    - Nun von Clip Space zu Normalized Device Coordinates (NDC)
      - Geschieht mit Hilfe des sog. "Perspective Divides" (durch Division durch *w*)
    - Von NDC zum sog. Window Space (d.h. zum eigentlichen Viewport)

Koordinatentransformationen im 3D

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

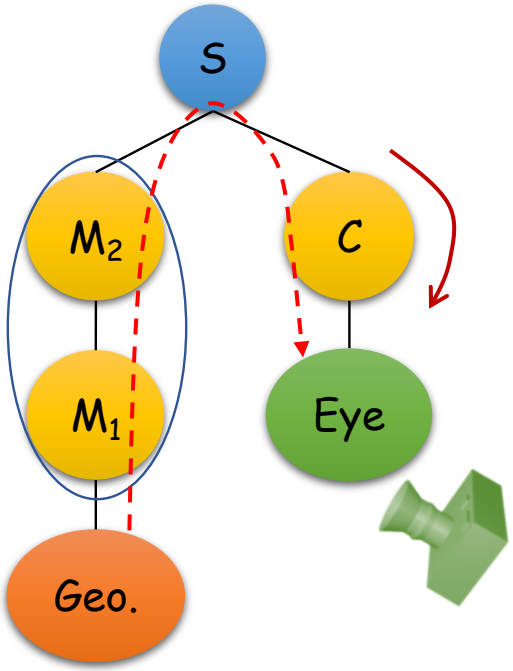
Umrechnung in Kamerakoordinaten

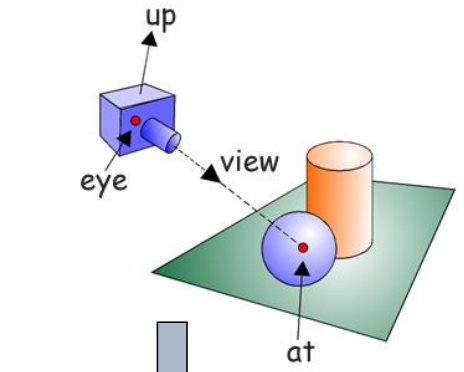


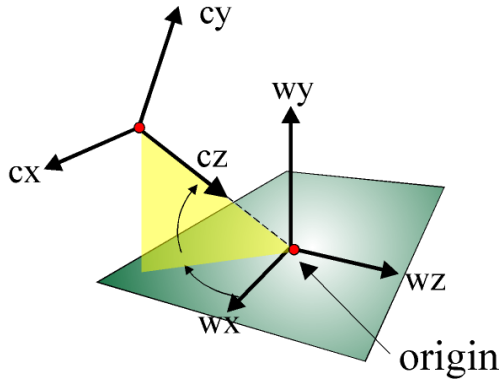


View Matrix

- Kameramatrix C definiert Position und Orientierung der Kamera in Weltkoordinaten
  - View Matrix ist Inverse der Kameramatrix: $V = C^{-1}$
- Navigation (z.B. Fly) bedeutet Manipulation der Kameramatrix C (mit $C = T \cdot R$)
  - Positionsänderungen durch Translation T
  - Orientierungsänderungen durch Rotation R
- Modelview-Matrix: $p' = V \cdot M \cdot p$







View-Matrix:
$V = C^{-1} = (T \cdot R)^{-1}$

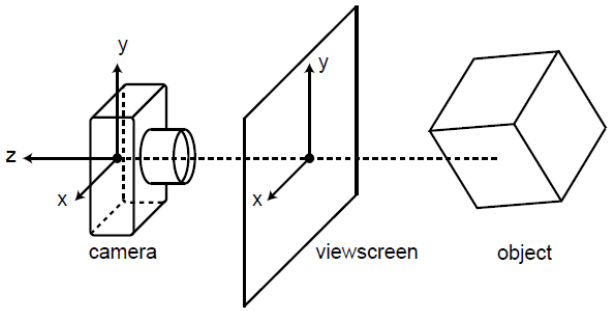
Transformiert Welt in kameralokale Koordinaten (Kamera im Ursprung, schaut entlang negativer z-Achse)

Model-Matrix:
$M = M_2 \cdot M_1$

Akkumulierte Transformationen, positioniert 3D-Objekte in Welt

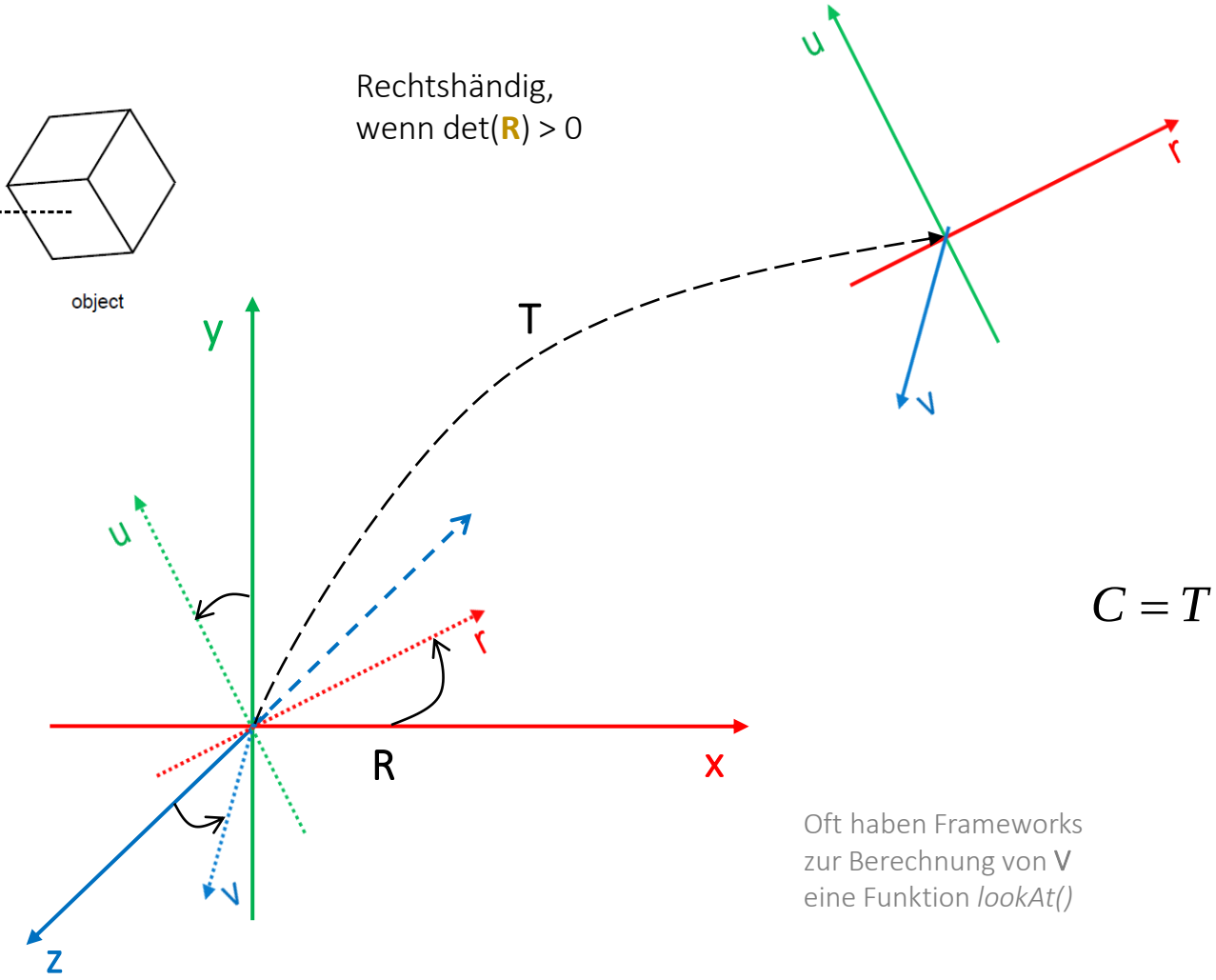
$$(T \cdot R)^{-1} = R^{-1} \cdot T^{-1}$$
$$\overset{R\ orthog.}{=} R^T \cdot T_{\vec{t}}^{-1} = R^T \cdot T_{-\vec{t}}$$

Matrix definiert Koordinatensystem



Im View/Eye Space schaut Kamera von Nullpunkt entlang negativer z-Achse

Rechtshändig, wenn det(**R**) > 0



Linke, obere 3x3-Matrix beinhaltet typischerweise Rotation R
Spaltenvektoren stehen je zu einander senkrecht und sind auf 1 normiert

Orthonormal-Basis des $\mathbb{R}^3$

$$C = T \cdot R = \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

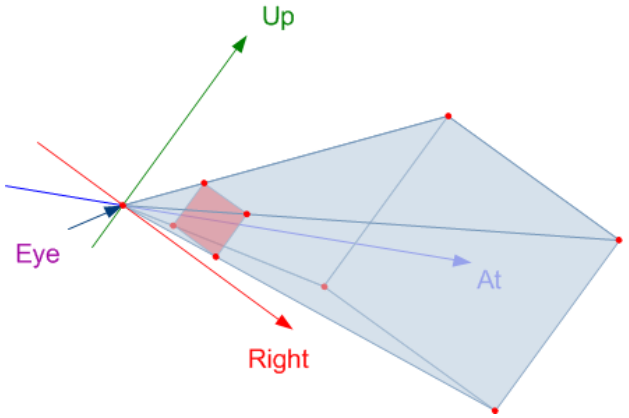
x (side) | y (up) | z (view) | O (pos.)

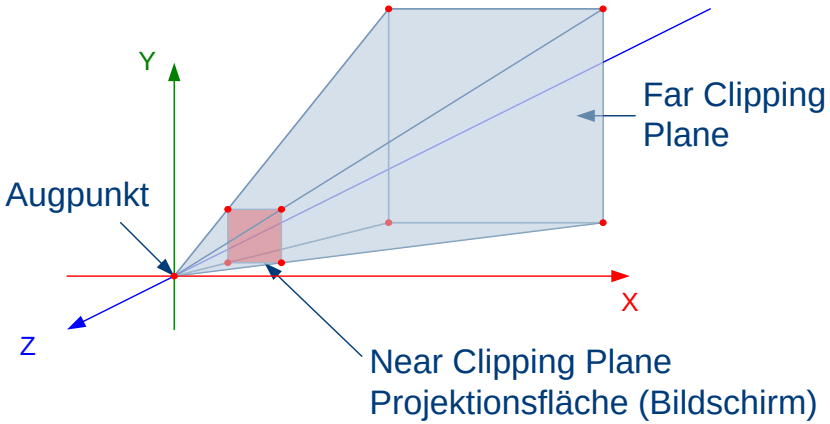
Die Transformation der Szene wird wieder als Matrixmultiplikation ausgedrückt mit $V = C^{-1}$

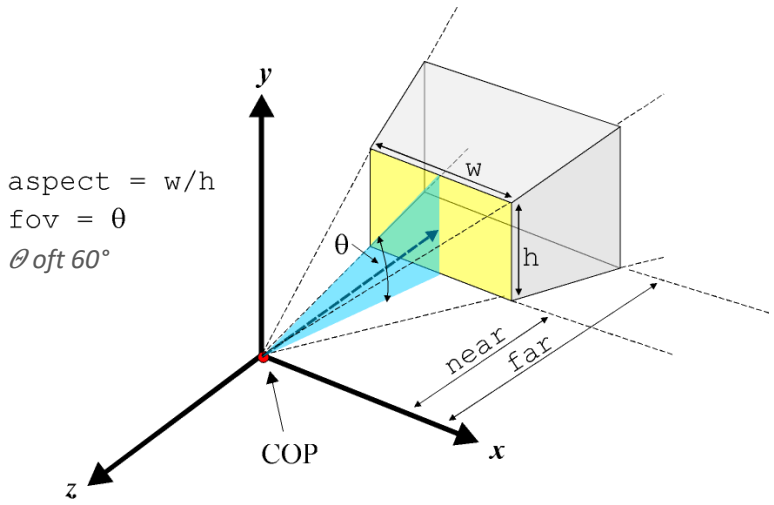
Oft haben Frameworks zur Berechnung von V eine Funktion *lookAt()*

Viewfrustum

- Sichtbereich durch Sichtpyramide (Viewfrustum) definiert
  - Die sechs eingrenzenden Flächen heißen Clipping Planes
    - Augpunkt (d.h. Kamera) liegt in Spitze der Sichtpyramide
  - Bereich zwischen Augpunkt und Projektionsfläche nicht sichtbar
- Definition der Sichtpyramide durch vier Parameter
  - Vertikaler Öffnungswinkel (*fov*), Aspect Ratio sowie Abstand von Near und Far Clipping Plane

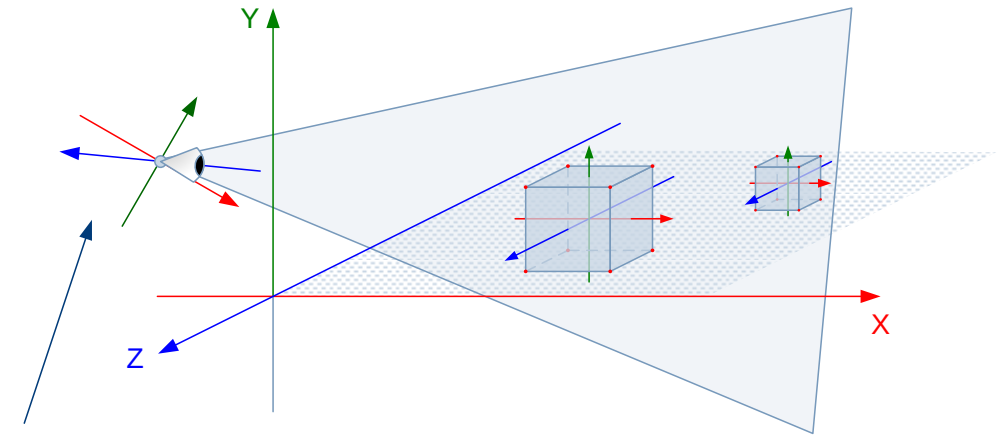




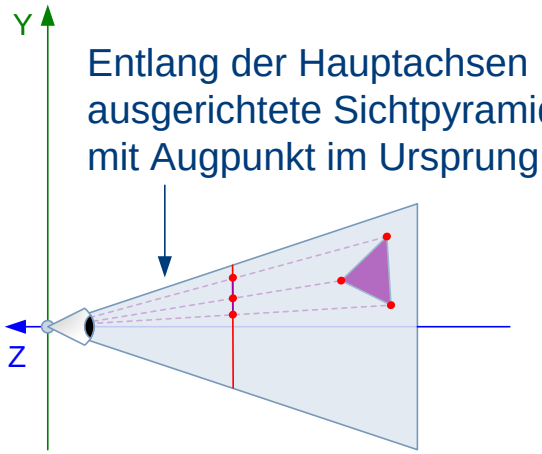


Viewfrustum

- Abbildung der 3D-Objekte auf 2D-Projektionsfläche erforderlich
  - Berechnung der Schnittpunkte der Sehstrahlen mit Projektionsfläche
    - Sehstrahlen entsprechen Strecken zwischen Kameraursprung und Eckpunkten
  - Schnittbestimmung einfacher bei ausgerichtetem Viewfrustum
- Sichtpyramide so ausgerichtet, dass Spitze in Ursprung und Blickrichtung entlang negativer z-Achse liegt



Transformiertes Koordinatensystem der Sichtpyramide



Entlang der Hauptachsen ausgerichtete Sichtpyramide mit Augpunkt im Ursprung

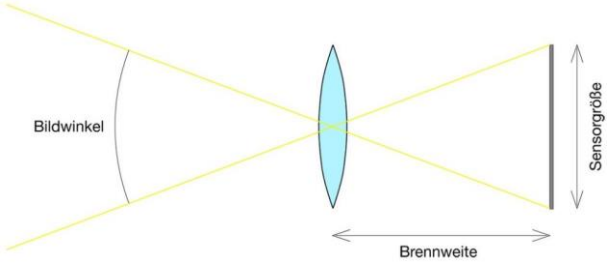
Perspektivische Wahrnehmung

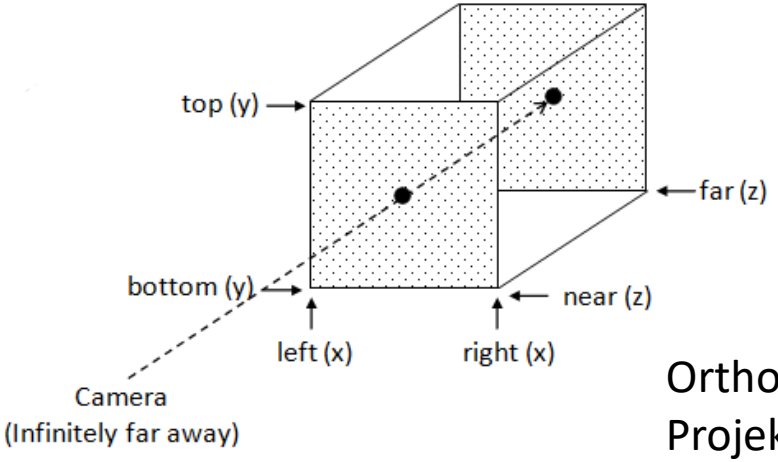
- Monokulare Raumwahrnehmung
  - Dazu zählen Perspektive, Verdeckung, Schatten und relative Größe
  - Je weiter hinten das Objekt ist, desto kleiner erscheint es uns
  - Wird diese Regel gebrochen, erhalten wir fehlerhafte Größeninformation
- Perspektivische Projektion
  - „Natürliche Darstellung“
  - Am meisten verwendete Projektionsart
  - Sehstrahlen treffen sich in Fluchtpunkt

Quelle: Wikipedia – Optische Täuschungen

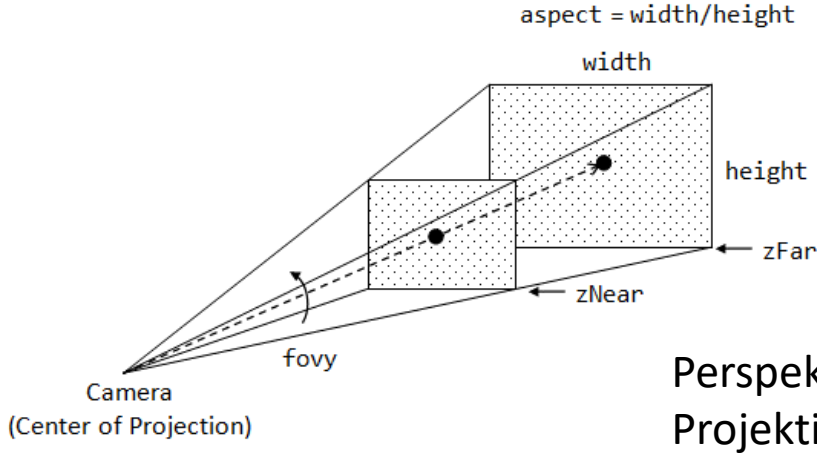
Kameraparameter

- View-Matrix definiert *extrinsische* Kameraparameter (Position und Orientierung)
- Es fehlen noch die *intrinsischen* Parameter
  - Vereinfacht: Welcher Bereich der Szene soll dargestellt werden?
  - Bei echter Kamera entspräche das Öffnungswinkel und Brennweite
- Projektionsmöglichkeiten





Orthografische Projektion

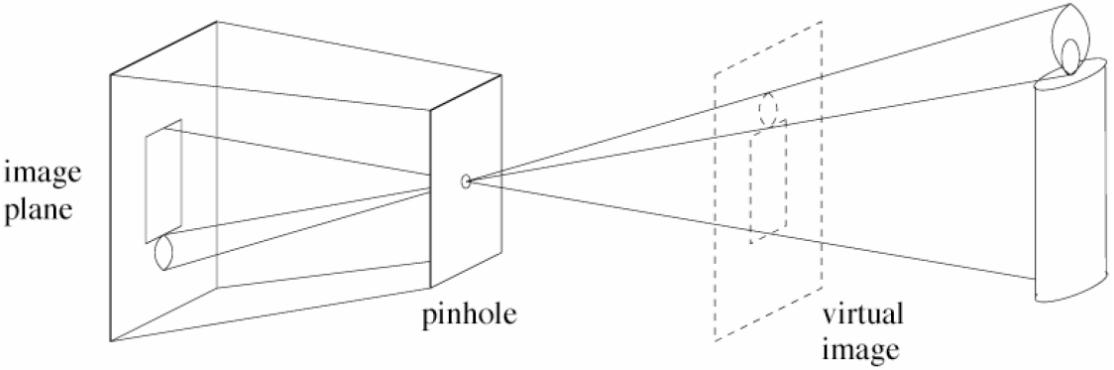


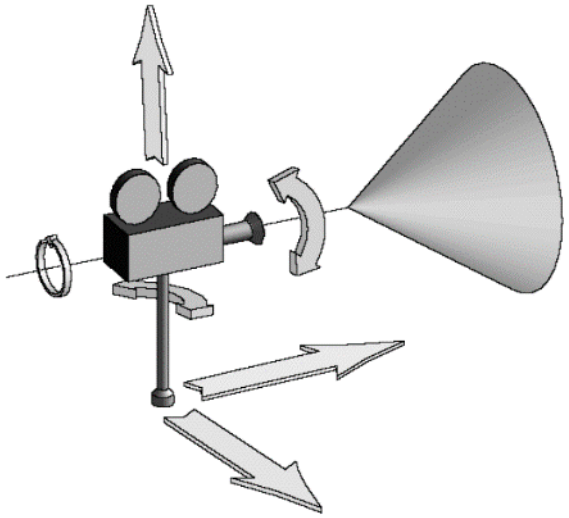
Perspektivische Projektion

Kameramodell der Computergraphik

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- Einfaches Lochkamera-Modell
  - Abbildung über Projektionsmatrix
- Projiziert 3D-Szene auf 2D-Bildebene
  - Modelliert perspektivische Projektion
  - Keine Tiefenschärfe, alles im Fokus
    - Da keine echte Linse sondern Lochblende
- Parameter (7 Freiheitsgrade):
  - Position (Projektionszentrum (cop))
  - Orientierung (Projektionsrichtung)
  - Öffnungswinkel (Field of View (fov))





fov → Zoom

Abbildung auf Bildebene

HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Orthographische Projektion

Parallelprojektion

Perspektivische Projektion

Zentralprojektion

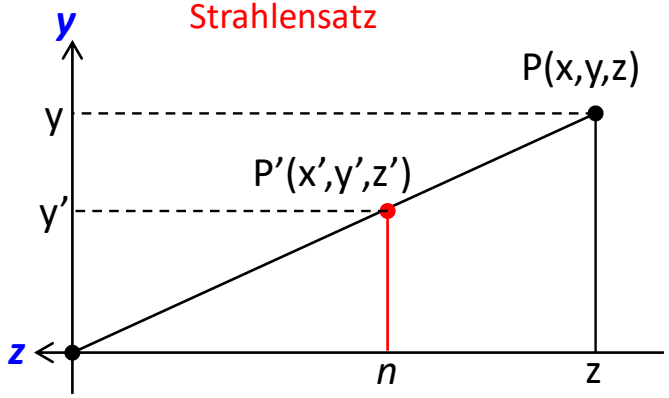
Abbildung auf Bildebene (vereinfacht)

HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Orthographische Projektion

Parallelprojektion

- Abbildung *3D* auf *2D*
  - Matrix: $$P_{Ortho} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Perspektivische Projektion

Zentralprojektion

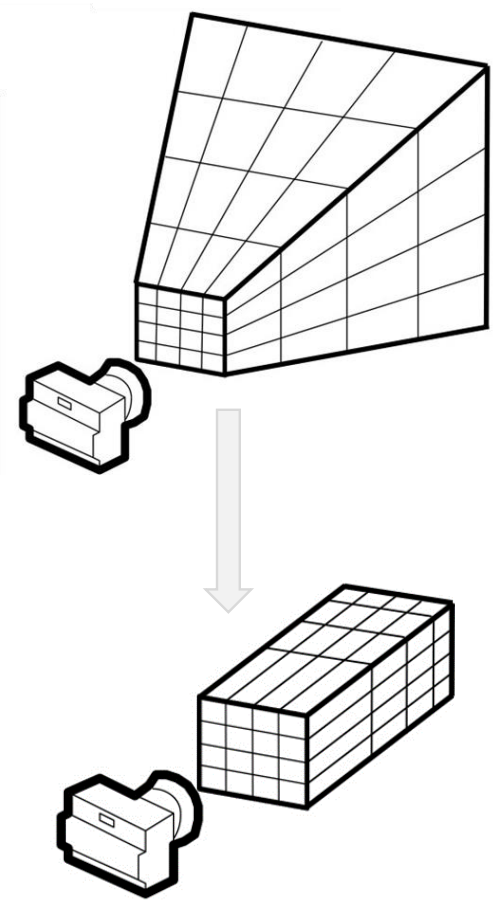
- Abbildung *3D* auf *2D*
  - Matrix: $$P_{Persp} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/n & 0 \end{pmatrix}$$
  - $$\frac{y}{y'} = \frac{z}{z'} \Rightarrow y' = \frac{y}{z/n}$$ analog für *x*

$$p' = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/n & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z/n \end{pmatrix} \xrightarrow{\frac{1}{z/n}} \begin{pmatrix} n \cdot x/z \\ n \cdot y/z \\ n \\ 1 \end{pmatrix}$$

Projektionsmatrix

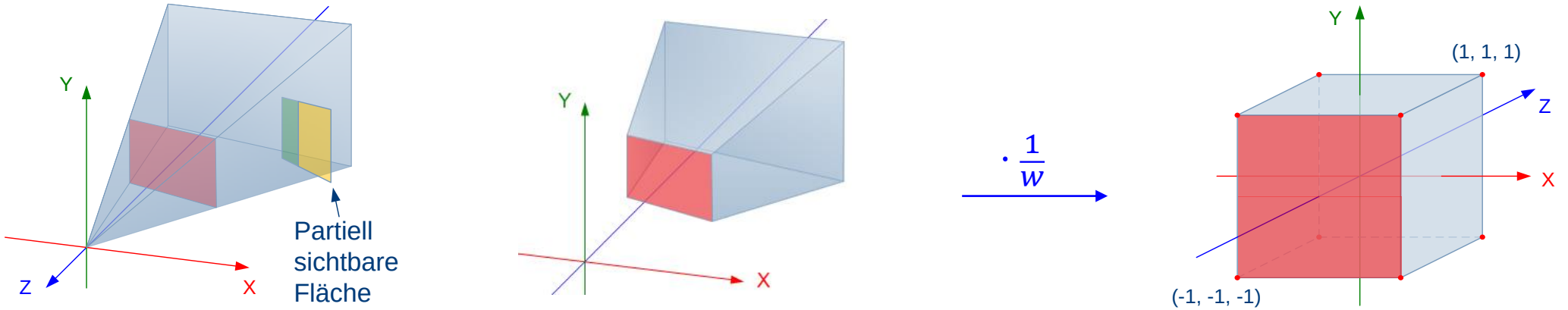
$w_{Clip\ Space} = -z_{Eye\ Space}$

- Bei vereinfachter Version geht *z*-Wert (Tiefe) verloren
- Projection Matrix *P*
  (mit $e = \frac{1}{\tan\left(\frac{\Theta}{2}\right)}$ und $a = \frac{w}{h}$): $P = \begin{pmatrix} e/a & 0 & 0 & 0 \\ 0 & e & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{pmatrix}$ *(ohne Herleitung)*
  - Abbildung auf 2D-Projektionsfläche gemäß Strahlensatz
  - Ergebnis abhängig vom *z*-Abstand eines Eckpunkts zur Kamera
    - Nichtlineare Skalierung der *z*-Koordinaten (vorne höhere Präzision)
  - Koordinaten auf Near Plane bleiben unverändert
- Transformation eines homogenen Punktes mit *P* ergibt Vektor, dessen *w*-Komponente ungleich 1 ist
  - Teilen durch *w* ergibt 3D-Punkt (vierte Dimension nun obsolet)

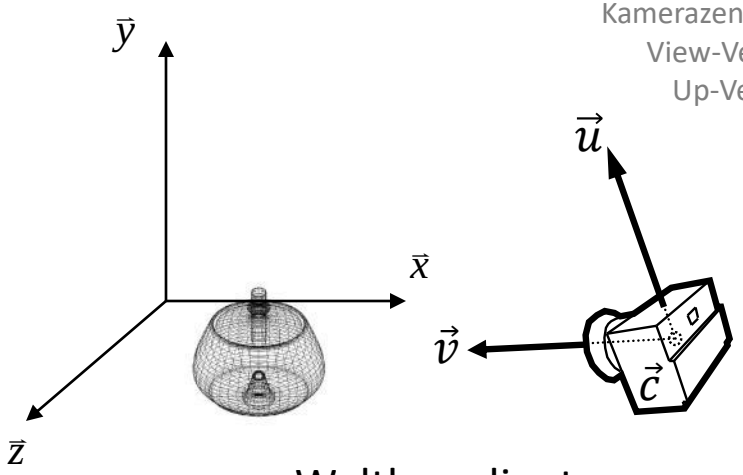


Identifikation unsichtbarer Bereiche

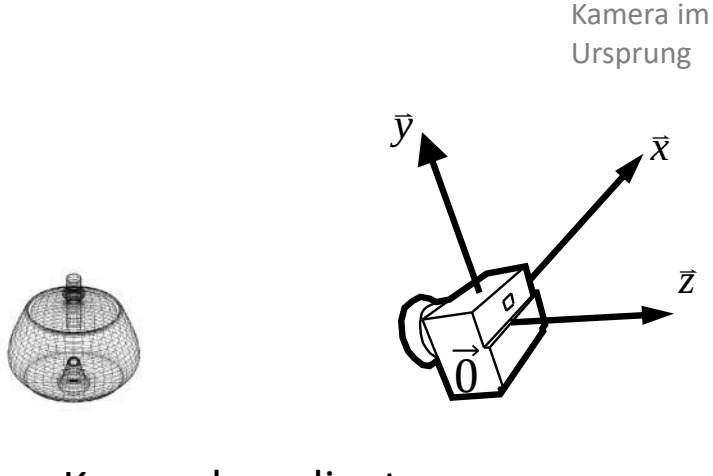
- Clipping von Polygonen gegen 3D-Sichtpyramide aufwendig
  - Seitliche Clipping Planes nicht achsparallel, Inside-Tests nicht durch Vergleich möglich
  - Daher Transformation des Sichtbereiches in Würfel (Clip Space)
    - Eckpunkt P(x, y, z) sichtbar, wenn $-w \leq x, y, z \leq w$
- Für Darstellung Übergang von rechts- zu linkshändigem Koordinatensystem
  - Dabei Transformation des Sichtbereiches in Einheitswürfel in Bereich $[-1,1]^3$
  - Geschieht durch perspektivische Division, d.h. Teilen durch w-Koordinate
    - Tiefe wächst damit bei zunehmendem z-Wert



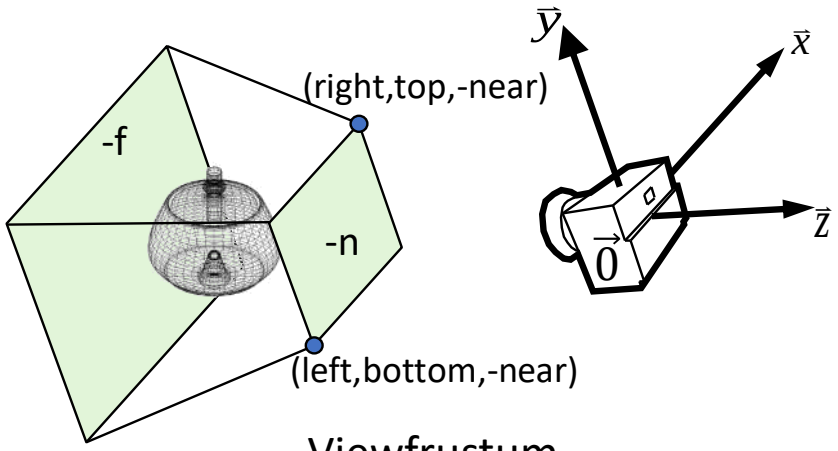
Transformationspipeline



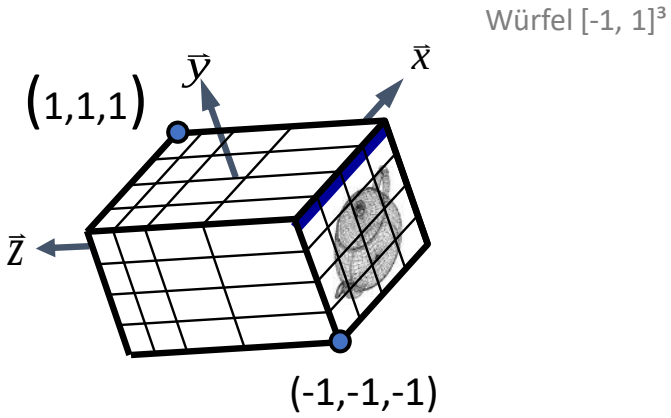
Weltkoordinaten



Kamerakoordinaten

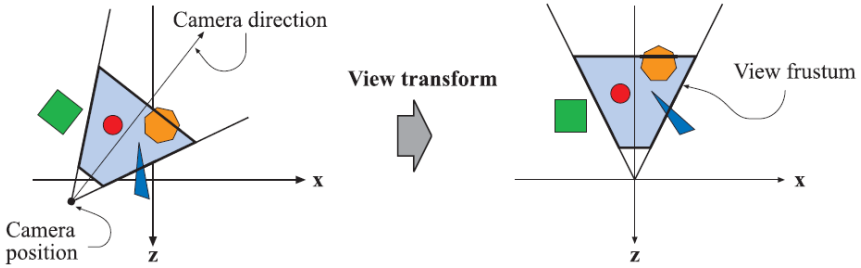


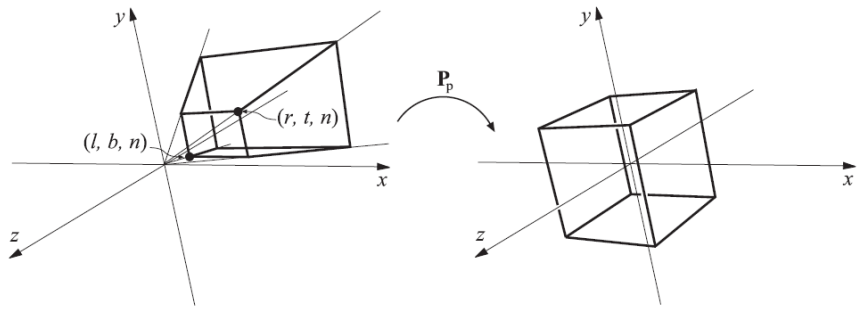
Viewfrustum



Normalized Device Space (NDC)

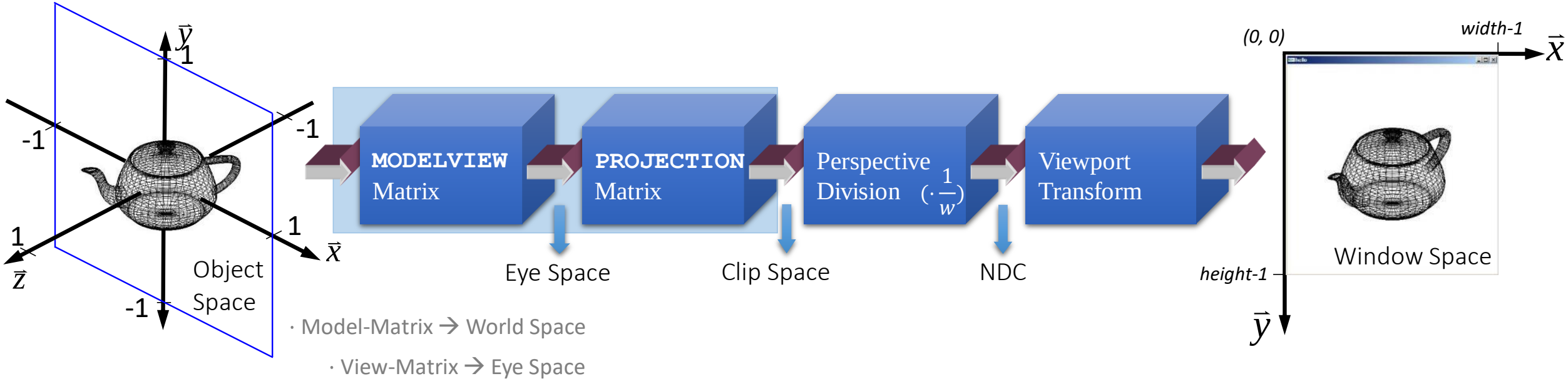
Model-View-Transformation





Projektion

Viewport-Transformation



· Model-Matrix → World Space

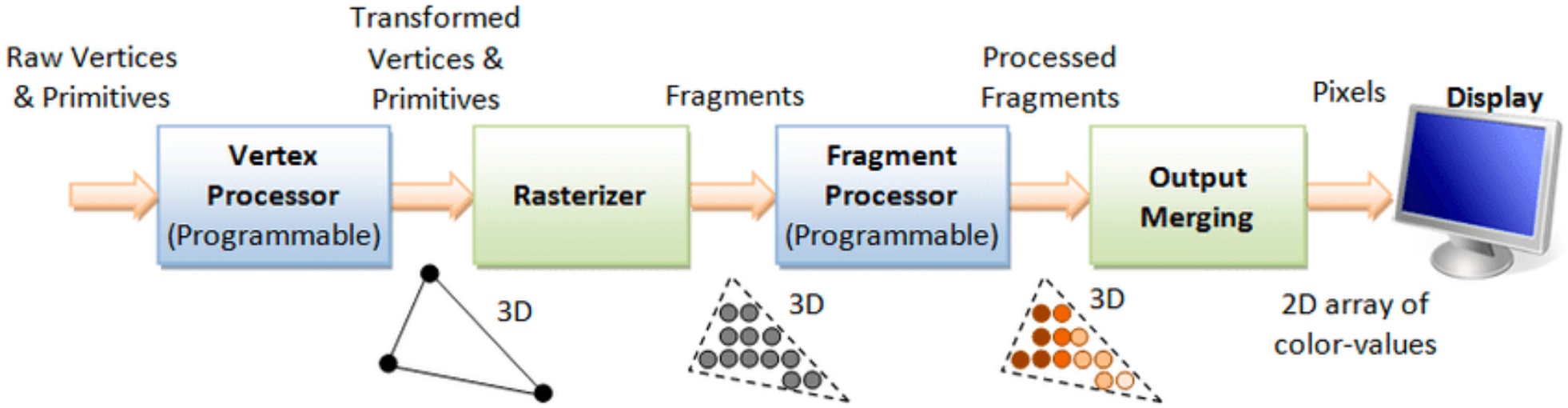
· View-Matrix → Eye Space

- Transformation der *xy*-Koordinaten des kanonischen Sichtvolumens (Normalized Device Space) aus Bereich [-1, 1] × [-1, 1] in Pixelkoordinaten (Viewport) aus Bereich [0, *width*-1] × [0, *height*-1]
- Transformation der *z*-Koordinaten aus [-1, 1] in Bereich [0, 1] für Tiefenpuffer (→ Verdeckungen)

GPU: Geometriestufe & Rasterisierung

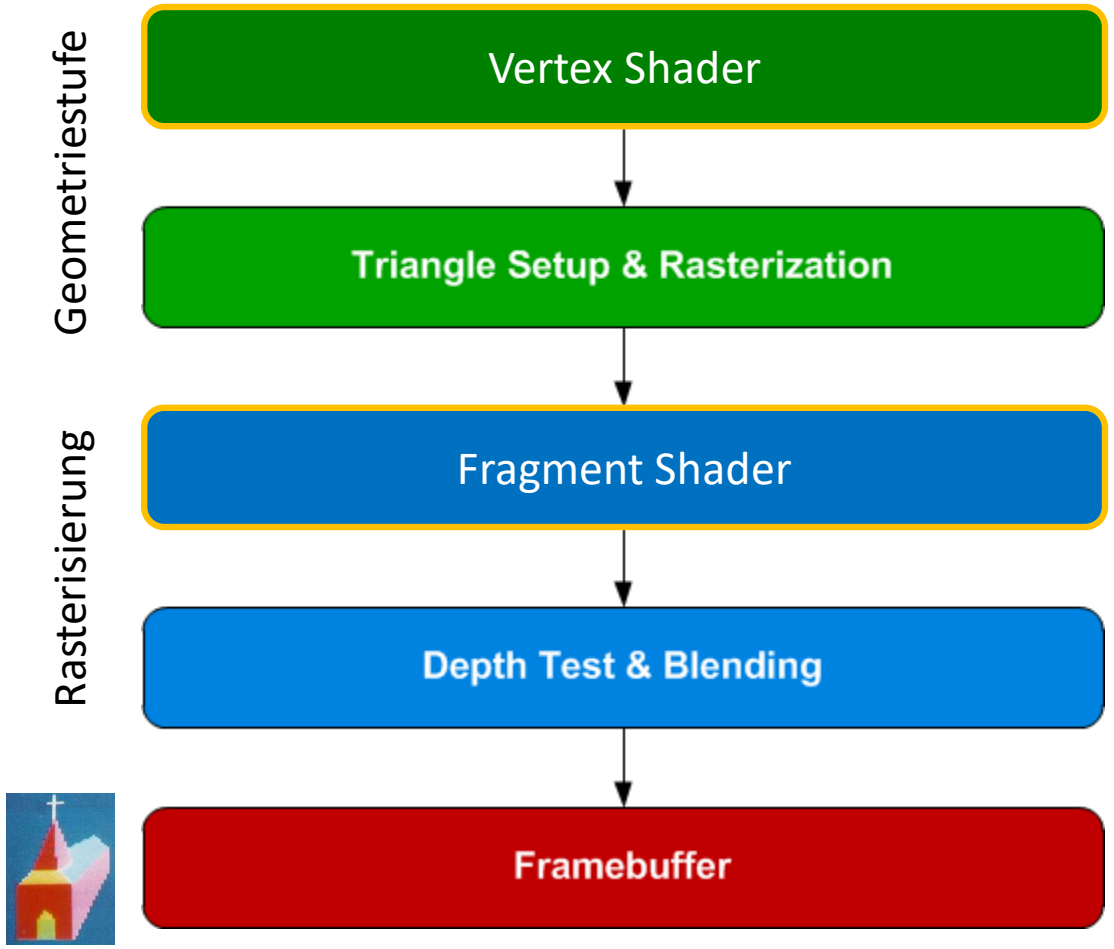
h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- Positionieren von 3D-Objekten in Szene durch Transformationen mit Integration der virtuellen Kamera sind die erste Schritte einer ganzen Kette von Berechnungen
  - Diese werden automatisch auf GPU ausgeführt oder können z.B. per OpenGL-Befehl aktiviert/deaktiviert werden
- Unterschied zwischen Fragment und Pixel: Fragment ist nur potentielles Pixel, d.h. Vorstufe dazu
  - Manchmal bilden mehrere Fragmente ein Pixel (Transparenz) und nicht alle Fragmente werden zum Pixel (Z-Buffer)



Bildquelle: https://www.researchgate.net/publication/334129575_Efficient_Spatial_Anti-Aliasing_Rendering_for_Line_Joins_on_Vector_Maps

Überblick: Vom Eckpunkt zum Bild

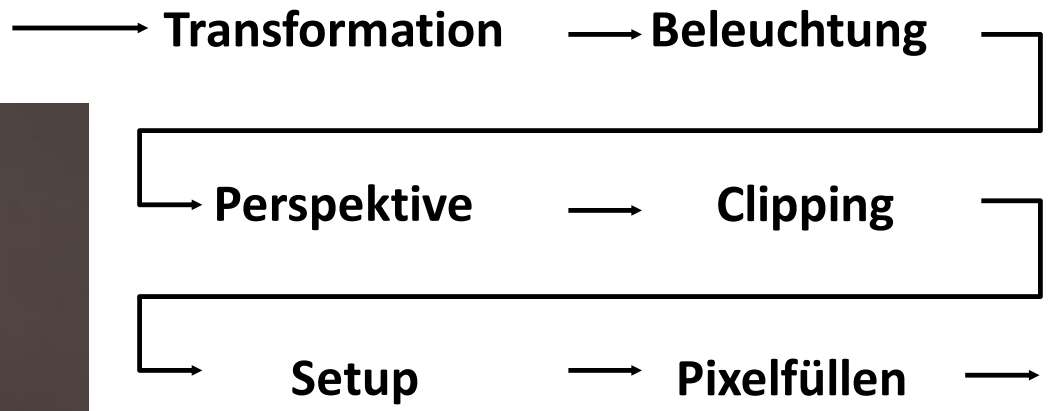


- Grafikkarteninterner Workflow wird vom Entwickler mit geeigneten Befehlen und Grafikdaten angestoßen
  - Grafikengines abstrahieren davon für einfachere Entwicklung
- Auf GPU hochgeladene Grafikdaten können vor Rendern noch manipuliert werden
  - Berechnungen pro Vertex bzw. Pixel parallel ausgeführt
  - Wird heutzutage in sog. Shadern implementiert: kleine Programme, die direkt auf GPU ausgeführt werden
- Jeder Shader hat hierbei eine genau definierte Aufgabe
  - Vertex Shader berechnet finale Position jedes Vertex im Bild
  - Fragment Shader berechnet Farbe jedes Pixels im Ausgabebild

Rendering Pipeline (Fixed Function)

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

GPU-interner Workflow – muss vom Anwendungsentwickler mit passenden Grafikdaten je "befüttert" werden



Transformation, Beleuchtung, Projektion
Clipping
Triangle Setup
Rasterisierung
Farbe / Texturierung
Stencil-, Tiefentest, Blending

Früher

Rendering Pipeline (Programmierbar)

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

GPU-interner Workflow – muss vom Anwendungsentwickler mit passenden Grafikdaten je "befüttert" werden

Transformation → **Beleuchtung**

Perspektive → **Clipping**

Setup → **Pixelfüllen** →

Vertex-Shader → Anweisungen zur Eckpunkttransformation

Clipping

Triangle Setup

Rasterisierung

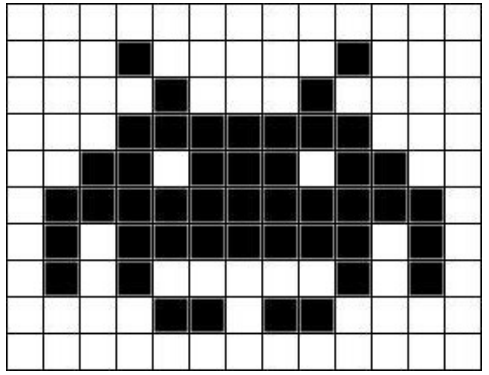
Fragment-Shader → Anweisungen zur Farbgebung eines Pixels

Stencil-, Tiefentest, Blending

heute

Recap: Rasterisierungsansatz

HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES



```
for each 3d object:
   frustum culling
   (occlusion culling)
   transform into clip space
   backface-cull, clip
   homogeneous divide
   find out which pixels are covered (i.e., rasterize)
   for each covered pixel:
      compute color
      depth-test, alpha-test, etc.
      blend pixel value with previous value
```

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Vielen Dank!

Noch Fragen?