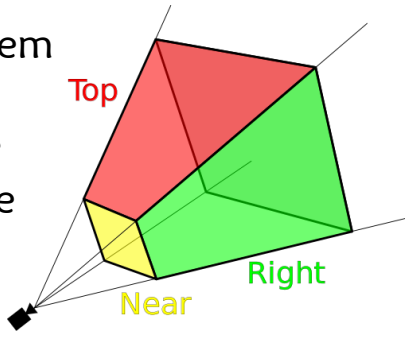
Rendering-Pipeline (funktional)

1. Objekte mit lokalen Koordinaten
2. Objekte in Weltkoordinaten
3. Kamera definiert Sichtvolumen
4. Entfernen von Rückseiten (*culling*)
5. Zuschneiden auf Sichtvolumen (*clipping*)
6. Rastern
7. Schattieren (*shading*)
8. Entfernen verdeckter Flächen (*z-buffer*)

(Loch-)Kamera

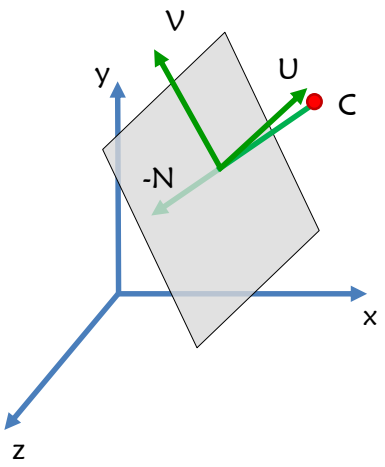
- Position, Blickrichtung, Drehung, Öffnungswinkel
- Ansichtskoordinatensystem
  - 0 ist Kameraposition
  - Richtung entlang z-Achse
- Sehr nahe und sehr ferne Objekte auslassen
- Bildebene
- Sichtvolumen: Pyramidenstumpf (*frustum*)



Bilder: Wikipedia

Kameraparameter

- Kameraposition C in Weltkoordinaten
- Blickrichtung der Kamera als Vektor -N
- U und V spannen Bildebene auf
- U, V, N sind paarweise senkrecht



Wo ist oben?

- V zeigt nach „oben“ in der Bildebene
- Häufig ist N nicht parallel zum „Boden“
- Einfacher wäre „oben“ in Weltkoordinaten anzugeben – meist konstant $V_{\text{up}} = (0,1,0)$
- $V = V_{\text{up}} - \left(V_{\text{up}} \cdot N\right) N$ [und normalisiert]
- $U = V \times N$
- Es reicht oft N und V_{up} anzugeben

Ansichtskoordinaten

- U, V, N sind paarweise senkrecht
- Sind sie auch normiert, so bilden sie eine Orthonormalbasis unseres Vektorraums
- Weltkoordinaten zu Ansichtskoordinaten ist Basistransformation
- Alle Weltkoordinaten transformieren:

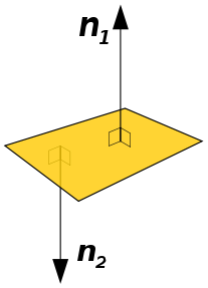
$$\begin{pmatrix} x_v \\ y_v \\ z_v \\ 1 \end{pmatrix} = \begin{pmatrix} U_x & U_y & U_z & 0 \\ V_x & V_y & V_z & 0 \\ N_x & N_y & N_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -C_x \\ 0 & 1 & 0 & -C_y \\ 0 & 0 & 1 & -C_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix}$$

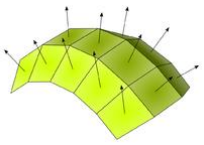
Ansichtsraum

- Kamera liegt bei (0,0,0)
- Blickrichtung entlang der negativen z-Achse
- Standardsituation bei OpenGL
- Objekte mit positiver z-Koordinate sind hinter der Kamera = nicht zu sehen
- z=0 ist direkt in der Kamera und somit „unendlich groß“
  - Kann nicht korrekt berechnet werden, daher ausgeschlossen

Normalenvektor

- Vektor senkrecht zur Oberfläche
- Üblicherweise normalisiert
- Genau zwei Normalenvektoren
  - $n_1 = -n_2$
- Vielseitig hilfreich
  - Definition innen/außen
  - Lichtberechnung
  - Beschreiben Neigung von Ebenen

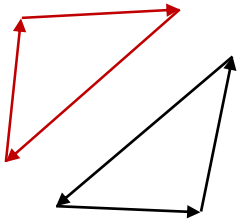




Bilder: Wikipedia

Orientiertes Polygon

- Reihenfolge der Punkte definiert Orientierung
- Beliebig, aber inoffizieller Standard: Vorderseite gegen Uhrzeigersinn (mathematisch positiver Sinn)
- Normale = Kreuzprodukt
- Normale zeigt nach „außen“



Rückseiten (*backface culling*)

- Ziel: Nicht sichtbare Polygone von weiteren Berechnungen ausschließen
- Einfach und schnell, möglichst früh
- In Weltkoordinaten: Skalarprodukt aus N und Normalenvektor
  - Polygon P hat Orientierung
  - Normalenvektor N_P steht senkrecht auf Polygon P und zeigt nach außen
  - P ist Rückseite wenn $N \cdot N_P < 0$
- In Ansichtskoordinaten noch einfacher – Wie?

Sichtvolumen (*frustum culling*)

- Objekte/Polygone komplett außerhalb des Sichtvolumens von weiteren Berechnungen ausschließen
- Komplexe Objekte in einfache „verpacken“ (*bounding box/sphere*) und diese mit Sichtvolumen schneiden
- Noch einfacher im Bildschirmraum

Bildschirmraum: Warum?

- Vereinfachen!
- Schneiden mit Sichtvolumen möglich, aber noch zu aufwändig
- Verdeckungstest möglich, aber noch zu aufwändig (erfordert Strahlverfolgung)
- Transformation auf die Bildebene wird danach trivial

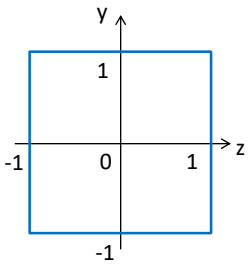
Bildschirmraum

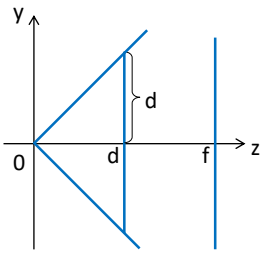
- Idee: Pyramide auf Würfel projizieren
  - Sichtvolumentest wird Koordinatenvergleich
  - Verdeckungstest: Punkte liegen hintereinander
- Bildebene hat Dimension (b,h)
- Erinnerung: Homogene Koordinaten

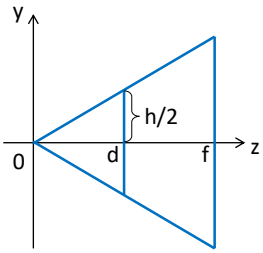
$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} wx \\ wy \\ wz \\ w \end{pmatrix} \qquad \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x/w \\ y/w \\ z/w \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix}$$

Perspektivische Projektion

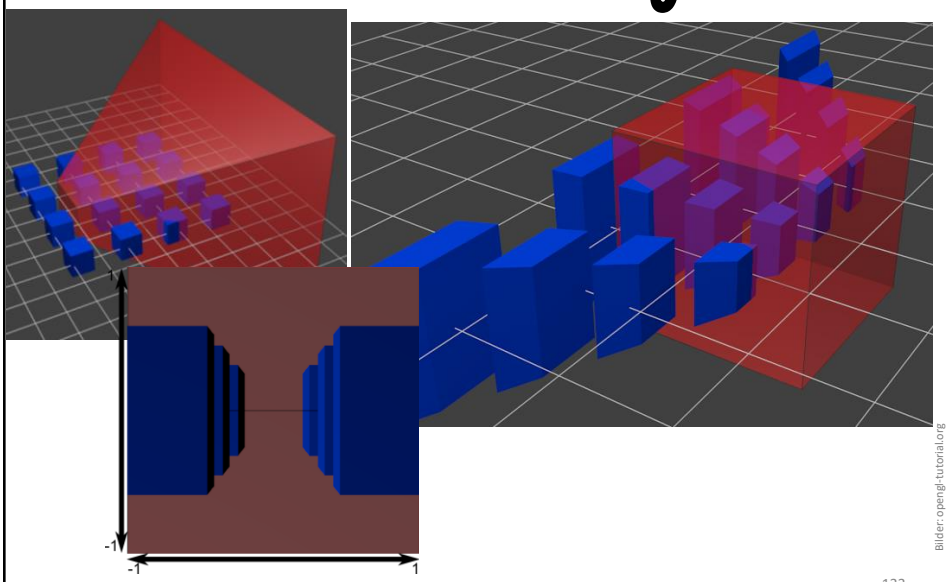
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{-f-d}{f-d} & \frac{-2df}{f-d} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} \frac{2d}{b} & 0 & 0 & 0 \\ 0 & \frac{2d}{h} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_v \\ y_v \\ z_v \\ 1 \end{pmatrix}$$







Was verformt die Projektion?



Bilder: opengl-tutorial.org

Ansichtsraum zu Bildschirmraum

- $\begin{pmatrix} x_S \\ y_S \\ z_S \end{pmatrix} \overset{\frac{1}{w}}{\leftarrow} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} \frac{2d}{b} & 0 & 0 & 0 \\ 0 & \frac{2d}{h} & 0 & 0 \\ 0 & 0 & \frac{-f-d}{f-d} & \frac{-2df}{f-d} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_v \\ y_v \\ z_v \\ 1 \end{pmatrix}$
- $x_S = \frac{x}{w} = \frac{2d}{b}\frac{x_v}{-z_v},\ y_S = \frac{2d}{h}\frac{y_v}{-z_v},\ z_S = \frac{f+d+\frac{2df}{z_v}}{f-d}$

Alternative: Parallelprojektion

- Orthogonal: z-Koordinate „löschen“
- $\begin{pmatrix} x \\ y \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$

- Keine perspektivische Verzerrung
  - Nahe und ferne Objekte haben gleiche Größe

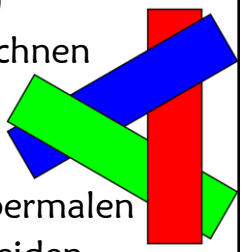
Bilder: Wikipedia

Aller guten Dinge...

- ...und Matrizen sind gute Dinge...
- Auf jeden Fall haben wir es immer mit (mindestens) drei zu tun:
- *Model matrix*
  - Lokale Koordinaten zu Weltkoordinaten
- *View matrix*
  - Weltkoordinaten zu Ansichtskoordinaten
- *Projection matrix*
  - Ansichtskoordinaten zu Bildschirmkoordinaten

Zeichenreihenfolge

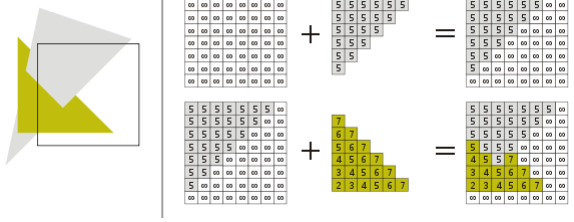
- *Painter's algorithm* (*back to front*)
  - Hinten liegende Elemente zuerst zeichnen
  - Vordere überdecken später
- Probleme
  - Rechenaufwand durch vielfaches Übermalen
  - Vorne/hinten nicht immer zu entscheiden
- Lösung 1: Polygone zerschneiden
- Lösung 2: z-buffer / depth buffer
- Für Transparenz Lösung 1 notwendig



Bilder: Wikipedia

Verdeckungstest (*z-buffer*)

- Zu jedem Pixel: Farbe und z speichern
- Nur Übermalen, wenn z kleiner/gleich
- Effizient wenn nach Tiefe aufsteigend sortiert (*front to back*)



Bilder: Wikipedia

Binary Space Partition

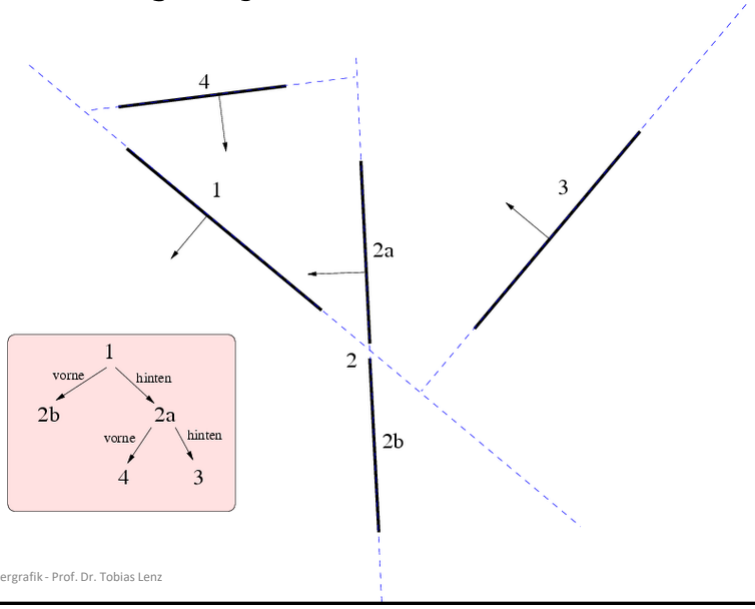
- Rechenzeit/Speicher für z-Buffer war früher nicht vorhanden
- Bei Transparenz ist die Zeichenreihenfolge wichtig, z-Buffer keine Lösung
- Wie richtige Reihenfolge sicherstellen?
- Eben gesehen: Zerteilen (*partition*) des Zeichenraumes (*space*) erforderlich, hier in „vorne“ und „hinten“ (*binary*)

Binary Space Partition Tree

- Ergebnis kann in einem Binärbaum festgehalten werden: BSP Tree

- bsp(Menge M)
  - Wähle beliebige Strecke/Fläche a aus M
  - Teile alle Elemente b aus M in Menge V wenn vor a oder H wenn hinter a
  - Falls nicht eindeutig, dann zerschneide b
  - Lege Knoten für a an mit linkem Kind bsp(V) und rechtem Kind bsp(H)

Binary Space Partition Tree



Bilder: Wikipedia (Marco Bakera)

Suche im BSP Tree

- Beispiel: Zeichenreihenfolge von hinten nach vorne

- Betrachte Fläche a im Wurzelknoten
- Kamera davon?
  - Zeichne rekursiv alle aus rechtem Kind, dann a, dann rekursiv aus linkem Kind
- Kamera dahinter?
  - Zeichne rekursiv alle aus linkem Kind, dann a, dann rekursiv aus rechtem Kind

BSP Tree: Qualität

- Baum evtl. schlecht balanciert
  - Hängt von der Wahl der Teilungsflächen ab
- Anzahl der Elemente im Baum vorher unklar
  - Je nach Teilungsfläche werden andere Flächen zerschnitten oder auch nicht
- Mögliche Lösung: mehrfach probieren, bestes Ergebnis behalten