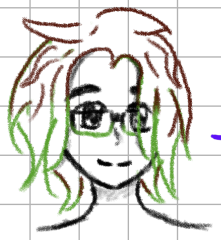
Disclaimer

Das ist nur eine Zusammenfassung und kein Ersatz für das Skriptum / die VOs. Keine Garantie darauf, dass alles so stimmt, wie es hier steht. Das ist nur meine Interpretation der Inhalte. Falls etwas unklar sein sollte, bitte im Skriptum nachschauen. Einige Unterkapitel (wie z.B. "Andere Objektrepräsentationen") könnten fehlen, da ich sie als unwichtig/nicht prüfungsrelevant erachtet hab. Jegliche Bilder gehören den Urhebern des Skriptums.



Viel Spaß beim Lernen!

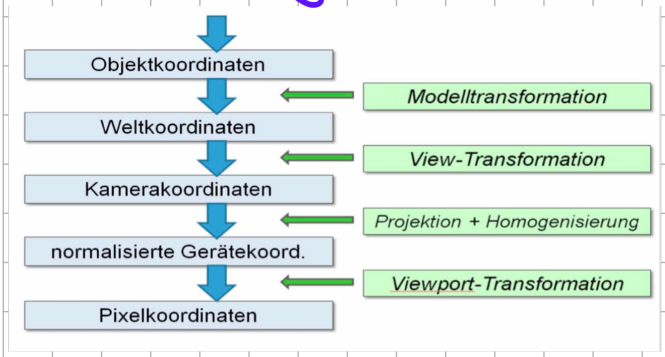
Inhaltsverzeichnis

(nur 6/7 Kapitel, da ich die Einführung bzw. das 1. Kapitel ausgelassen hab)

CG-Zusammenfassung

Graphikpipeline

Viewing-Pipeline

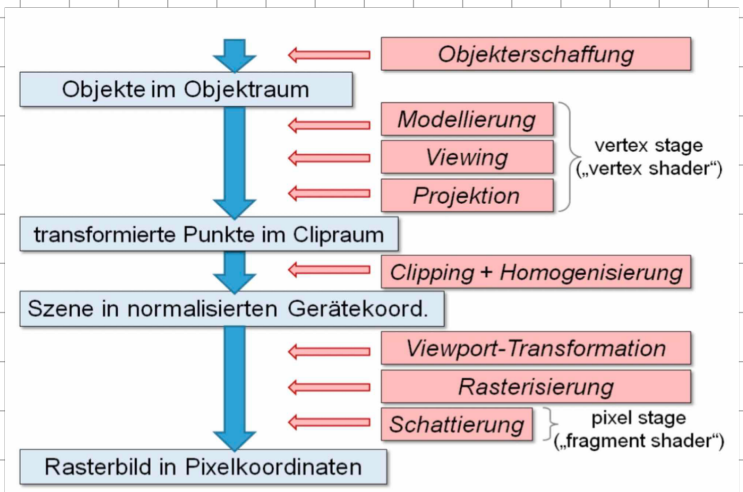


Objekte & Szene beschreiben
↓
Blickrichtung festlegen (View-Tr.)
↓
Objekte projizieren (Projektion)
↓
in Rasterpunkte umwandeln

Objekterschaffung:
- durch Modellierung
- durch Scanning

Programmierbare Teile der Graphikkarte/GPU:
- "vertex shader"
- "fragment shader"

Rendering-Pipeline

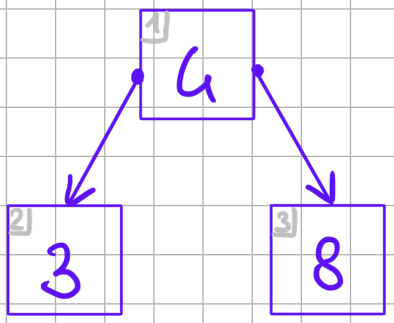


Graphen & Bäume (auch erklärt in EP2, AlgoDat & ttw. ADM)

zeigerverkettete Strukturen, ähnlich wie verkettete Listen
rekursive Datenstruktur
Bearbeitungsreihenfolge (nach Aufruf der Nachfolger):

"pre-order"

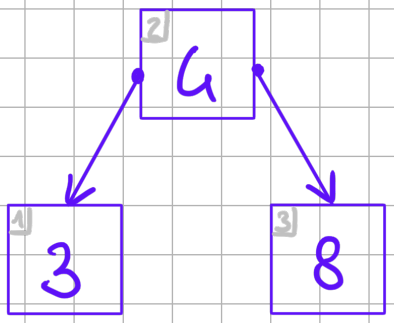
1. Wurzel, 2. links, 3. rechts



⇒ G38

"in-order"

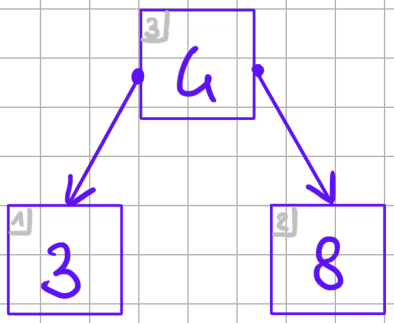
1. links, 2. Wurzel, 3. rechts



⇒ 3G8

"post-order"

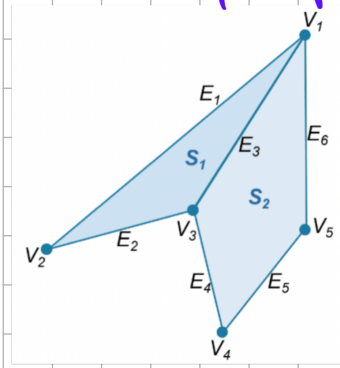
1. links, 2. rechts, 3. Wurzel



⇒ 38G

Polygon-Listen (B-Reps)

Boundary-Representation (B-Rep): beschreibt Oberfläche eines Objekts durch Polygone



VERTEX TABLE

V_1:	x_1, y_1, z_1
V_2:	x_2, y_2, z_2
V_3:	x_3, y_3, z_3
V_4:	x_4, y_4, z_4
V_5:	x_5, y_5, z_5

EDGE TABLE

E_1:	V_1, V_2
E_2:	V_2, V_3
E_3:	V_3, V_1
E_4:	V_3, V_4
E_5:	V_4, V_5
E_6:	V_5, V_1

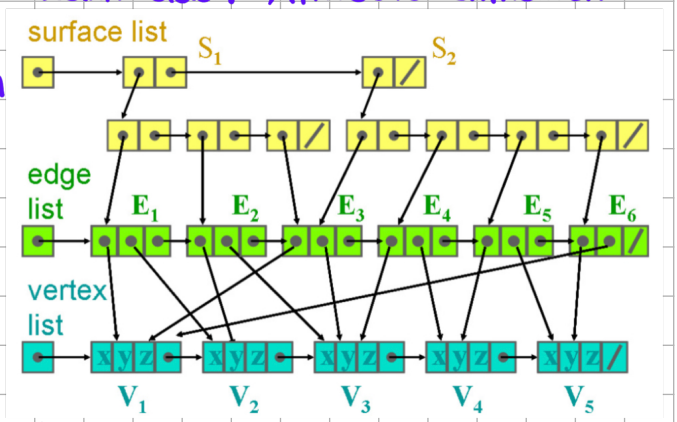
POLYGON TABLE

S_1:	E_1, E_2, E_3
S_2:	E_3, E_4, E_5, E_6

⇒ Geometrie: siehe Abb. links

Datenstruktur eines B-Reps kann auch Attribute enthalten

↳ Lässt sich auch mit Zeigerlisten darstellen (siehe Abb. rechts)



Vorteil: Lineare Verkettung erleichtert die Bearbeitung

Backface: Rückseite des Polygons / Frontface: Vorder-/Außenseite

Trägerebene: $Ax + By + Cz + D = 0$

Dreiecks-Mesh: Datenstruktur, die nur aus Dreiecken besteht

Dreiecks-Strip: Lineare Abfolge von Dreiecken

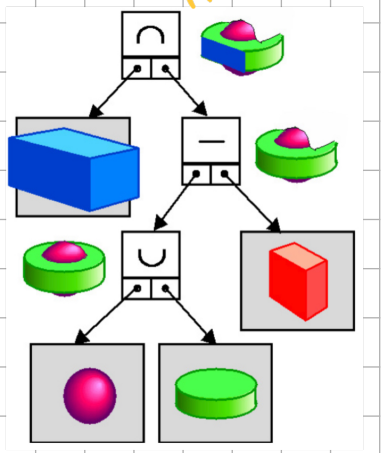
Constructive Solid Geometry (CSG)

⇒ Objekte werden aus 3D-Primitiven & Mengenoperationen gemacht

CSG-Baum: hirarchiche Datenstruktur, kreisfreier Graph

3D-Primitiven: geometrische Formen (Kugel, Würfel, Zylinder, ...)

Mengenoperationen: Vereinigung (∪), Durchschnitt (∩), Differenz (−)



⇒ alle Objekte immer konsistent & wohldefiniert

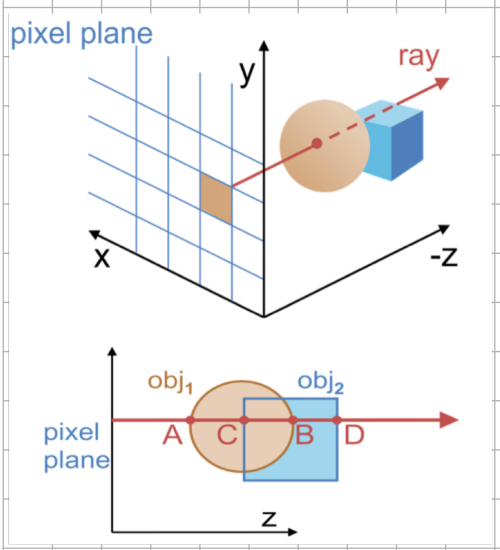
⇒ Transformation auch im Nachhinein möglich, da solche in Form von Matrizen gespeichert werden

Vorteile: exakte Repräsentation, geringer Speicherbedarf, Einfachheit von Transformationen

Nachteile: aufwändige Berechnung von Bildern, kompliziertes Rendering

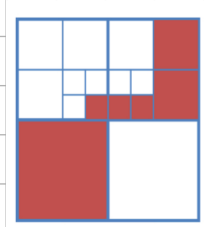
Raycasting von CSG-Objekten

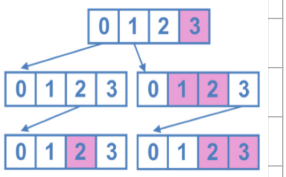
⇒ gebräuchlichste Methode, um CSG-Objekte abzubilden



- Strahl wird in Blickrichtung auf jedes Pixel gelegt und mit allen Objekten geschnitten
- vorderste Schnittpunkt = sichtbares Objekt
- Pixelfarbe = Objektfarbe
- Berechnung der Schnittpunkte rekursiv

Quadtrees und Octrees

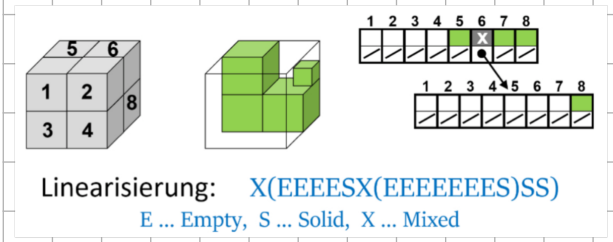




Quadtree: zur Repräsentation 2D-Strukturen

Octtree: Erweiterung auf 3D

⇒ ähnlich wie Binärbaum, komplizierte Würfel werden aufgeteilt



Vorteil: repräsentation beliebiger Formen, Leichte Untersuchung davon

Nachteil: ungenaue Repräsentation, hoher Speicherbedarf, komplizierte Transformationen

Szenengraphen

⇒ objektorientierte Datenstruktur, die logische/räumliche Anordnungen einzelner Elemente hirarchisch beschreibt

⇒ Oberbegriff für hirarchische Beschreibungsformen

⇒ Graphentheoretisch: baumartig, gerichtet & kreisfrei

⇒ Wurzelknoten = gesamte Szene, Endknoten = einfachstes Objekt

Einfache 2D-Transformationen

Translation (Verschiebung): $(x', y') = (x + t_x, y + t_y)$

Rotation (Drehung): $(x', y') = (x \cdot \cos\Theta - y \cdot \sin\Theta, x \cdot \sin\Theta + y \cdot \cos\Theta)$

Skalierung (Verkleinerung/Vergrößerung): $(x', y') = (s_x \cdot x, s_x \cdot y)$

Reflexion (Spiegelung): Skalierung mit $s_x = (-1)$ oder $s_y = (-1)$

Transformationsmatrizen (für Translation keine vorhanden):

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$ Skalierung

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$ Rotation (gegen Uhrzeigersinn)

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$ Spiegelung um x-Achse

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$ Spiegelung um y-Achse

Homogene Koordinaten

nötig, damit Translation in Matrizenform berechnet werden kann:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$ 2D-Rotation

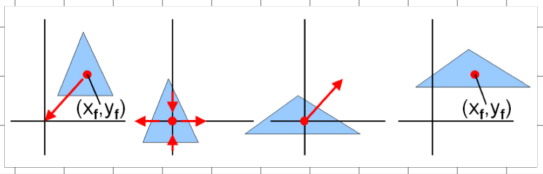
$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$ 2D-Skalierung

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$ 2D-Translation

hier: $h=1$ (ist meistens der Fall)

Vorteil an Matrixtransformation: meist größere Teile auf einmal transformiert ⇒ geringerer Rechenaufwand mit Matrizen

Komplexe 2D-Transformationen

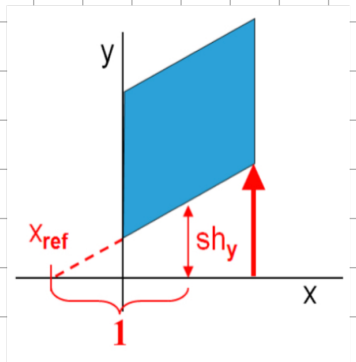


Skalierung mit (x_f, y_f) als Zentrum:

$$S(x_f, y_f, s_x, s_y) = T(x_f, y_f) \cdot S(x_f, y_f) \cdot T(-x_f, -y_f)$$

Spiegelung an $y = mx + b$ Achse:

$$X(m,b) = T(0,b) \cdot R(\Theta) \cdot S(1,-1) \cdot R(-\Theta) \cdot T(0,b)$$



Scherung: $\begin{pmatrix} 1 & 0 & 0 \\ sh_x & 1 & -sh_y \cdot x_{ref} \\ 0 & 0 & 1 \end{pmatrix}$ in y Richtung

Affine Transformationen

⇒ Koordinaten lassen sich durch lineare Funktionen + Translation in einander umwandeln, d.h.:

- Punkte auf einer Linie sind auch auf der abgebildeten Linie
- Proportionalität von Abständen bleibt gleich

- Verhältnis von Längen bleibt erhalten
- Winkel, Parallelität, Endlichkeit bleiben erhalten

3D-Transformationen

Hier sind einmal die wichtigsten 3D-Transformationen:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3D-Translation

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3D-Skalierung

$$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Spiegelung um yz-

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

xz-

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

xy-Ebene

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3D-Rotation um x-Achse

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

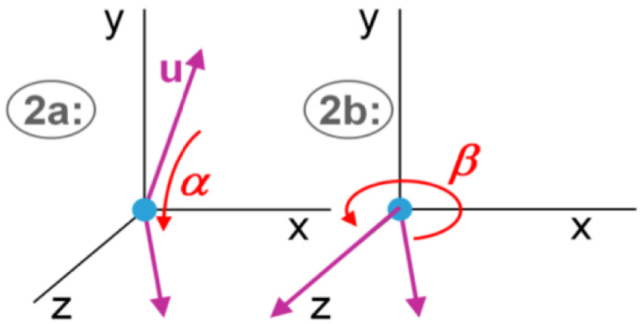
3D-Rotation um y-Achse

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3D-Rotation um z-Achse

komplexe Tf. – Drehung um Winkel θ um beliebige Achse:

$$R(\theta) = T^{-1}(-x_1,-y_1,-z_1)\cdot R_x^{-1}(\alpha)\cdot R_y^{-1}(\beta)\cdot R_z(\theta)\cdot R_y(\beta)\cdot R_x(\alpha)\cdot T(-x_1,-y_1,-z_1) =$$
$$= T(x_1,y_1,z_1)\cdot R_x(-\alpha)\cdot R_y(-\beta)\cdot R_z(\theta)\cdot R_y(\beta)\cdot R_x(\alpha)\cdot T(-x_1,-y_1,-z_1)$$



Scherung in 3D: $\begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ (parallel zur xy-Ebene um den Wert a_x bzw. b_y)

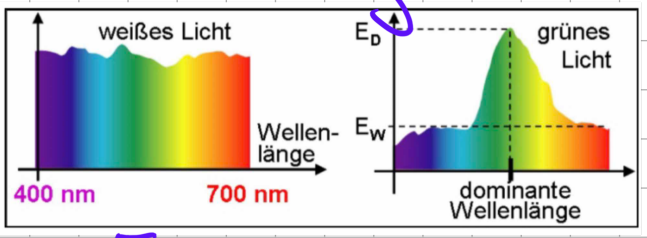
Farbe

= elektromagnetische Strahlung unterschiedlicher Frequenzen

spektralreines Licht = Licht hat genau eine Wellenlänge

- bei mehreren Farben:

dominanteste Wellenlänge sichtbar

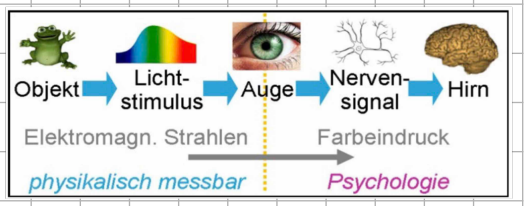


Reinheit einer Farbe: $(E_D - E_W)/E_D$, mit E_D = dom. Wellenlänge,

E_W = durchschnittliche Energie der anderen Farben

Helligkeit: Integral der Spektralkurve

Kolorimetrie



= technische Beschreibung von Farben

Farbe = empfundener Sinneseindruck, daher nur folgendes definiert:

- Stimuli mit gleichen Spezifikation sehen unter gleichen Bedingungen gleich aus
- gleich aussehende Stimuli = gleiche Spezifikationen
- verwendete Zahlen = stetige Funktionen von physikalischen Parametern

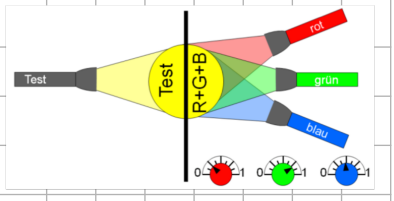
⇒ berücksichtigt nur visuelle Unterscheidbarkeit

Farbfehlsichtigkeit

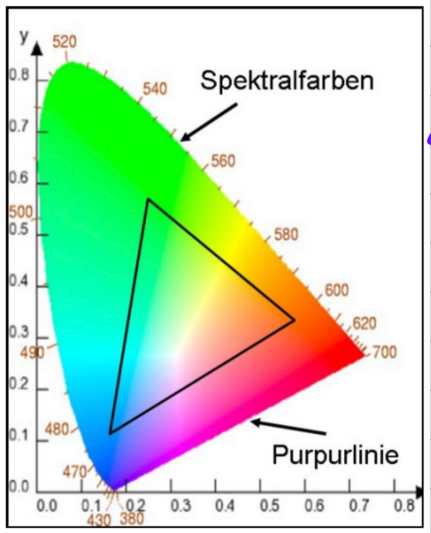
= Zäpfchenart fehlt oder Empfindlichkeitskurven zu ähnlich

häufigste Art: Rot-Grün-Schwäche

CIE 1931 XYZ-Farbmodell



Testpersonen müssen mit RGB-Reglern spektralreine Farben additiv "nachbauen" ⇒ "imaginäre Grundfarben" X, Y, Z



Ergebnis nach Normierung auf Helligkeit 1 und Projizierung auf XY-Ebene: CIE-Diagramm

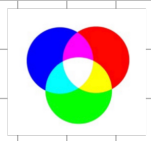
Helligkeit kann zusätzlich als Y angegeben werden

vollständige Definition: (x, y, Y)

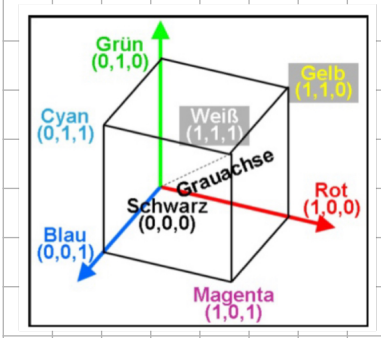
Purpurlinie = Gegenteil der spektralreinen Farben

Dreieck: mit RGB-Monitor darstellbare Farben

RGB-Farbmodell



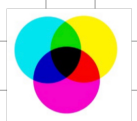
⇒ Farbraum zur Beschreibung von Farben auf einem Gerät



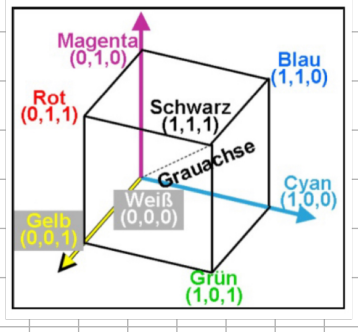
Zusammensetzung der Pixel aus drei additiven Lichtpunkten; Grundfarben von Monitor abhängig

Gamut: vom Gerät erzeugbarer Farbraum

CMY-Farbmodell



⇒ subtraktive Farbmischung von Licht; Komplement von RGB



[C, M, Y] = [1, 1, 1] - [R, G, B]

CMYK-Modell: K = Key = Schwarzanteil

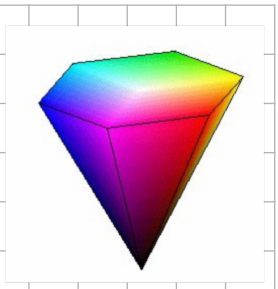
⇒ Schwarz nicht durch CMY gemischt

HSV- & HSL-Farbmodelle

⇒ an benutzerempfinden angepasste Modelle

HSV = Hue (Farbton), Saturation (Sättigung), Value (Helligkeit)

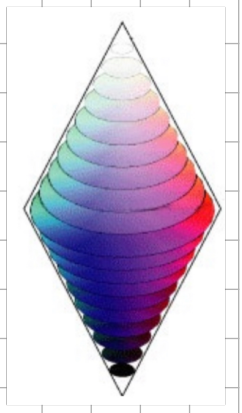
je dunkler, desto weniger Sättigungsabstufungen



Farbe = Grad entlang Basiskante (R=0°, G=120°, B=240°)

Sättigung = Abstand von der Pyramidenachse (in %)

Helligkeit = Abstand der Grundfläche von der Spitze (in %)



HLS = Hue, Luminance/Lightness, Saturation

Grund dafür: Annahme, dass weiß = heller als Farben

normalerweise steht die Wahl zwischen Farbmodellen frei

Farbsymbolik

wesentliche Gebiete: Sprachgebrauch, Religion, Politik, Kennzeichnung, Verkehr, Technik, Natur, Farbassoziationen

Befehle für Graphikprimitiven

wichtigste in 2D:
- Punkte, Listen
- Polygone, Kreise, Ellipsen & Kurven (auch gefüllt)
- Bitmap-Operationen
- Buchstaben & Zeichen

wichtigste in 3D:
- Dreiecke & weitere Polygone
- Freiformflächen

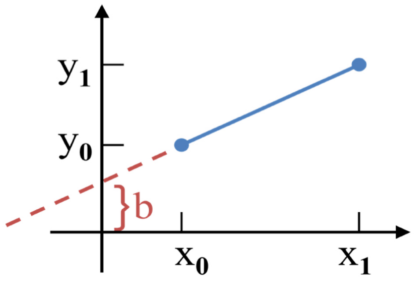
⇒ zusätzlich Befehle für Eigenschaften (Farbe, Füllmuster, ...)

Digital Differencial Analyser (DDA)

```
dx = x1 - x0; dy = y1 - y0;
m = dy / dx;

x = x0;   y = y0;
setPixel (round(x), round(y));

for (k = 0; k < dx; k++)
  { x += 1; y += m;
  setPixel (round(x), round(y)}
```

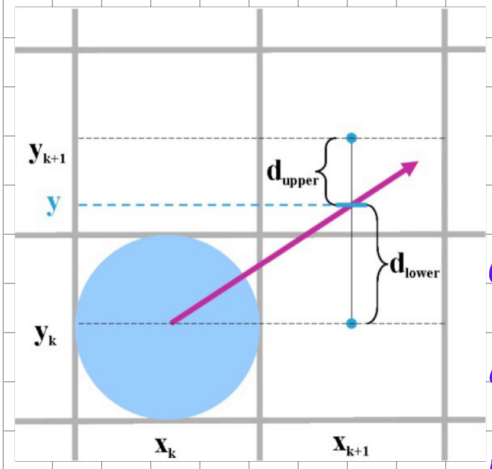


$|m|<1$: for-Schleife wird normal ausgeführt

$|m|>1$: x und y werden vertauscht (Ausführung in senkrechter Richtung)

Bresenham-Verfahren

wie DDA, aber verwendet nur Integer / lässt sich für Kurven anpassen



für $0<|m|<1$ folgt aus $y = mx + b$:

$d_{lower}: y - y_k = m(x_k + 1) + b - y_k$

$d_{upper}: (y_k + 1) - y = y_k + 1 - m(x_k + 1) - b$

$d_{lower} - d_{upper} = 2m \cdot (x_k + 1) - 2y_k + 2b - 1$

$p_k = \Delta x \cdot (d_{lower} - d_{upper}) = 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c$ (Entscheidungsvariablen)

$p_{k+1} = p_k + 2\Delta y - 2\Delta x \cdot (y_{k+1} - y_k)$; $p_0 = 2\Delta y - \Delta x$

```
1. store left line endpoint in (x0,y0)
2. plot pixel (x0,y0)
3. calculate constants Δx, Δy, 2Δy, 2Δy - 2Δx,  and obtain p0 = 2Δy - Δx
4. At each xk along the line, perform test:
      if pk<0
      then plot pixel (xk+1,yk);  pk+1 = pk+ 2Δy
     else plot pixel (xk+1,yk+1);  pk+1= pk+ 2Δy - 2Δx
5. perform step 4 (Δx - 1) times.
```

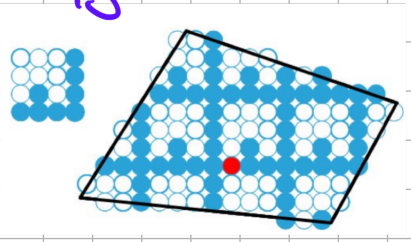
Attribute

Linien/Punkte: Strichdicke, Strichlierung, Farbe, Pinseltyp, Linienenden,

Eckenform:

Text: Font, Größe, Richtung, Farbe, Bündigkeit (links, rechts, mitte)

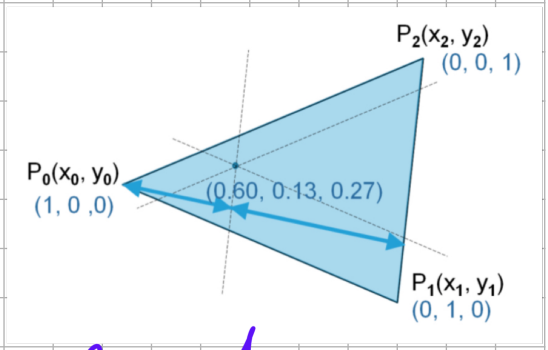
Polygone & Flächen: bei Muster - Aneinanderreihung des Grundmusters



ausgehend von Referenzpunkt (seed-point)

Mischen von Farben B&F: $P = t \cdot F + (1-t) \cdot B$

Dreiecke rasterisieren



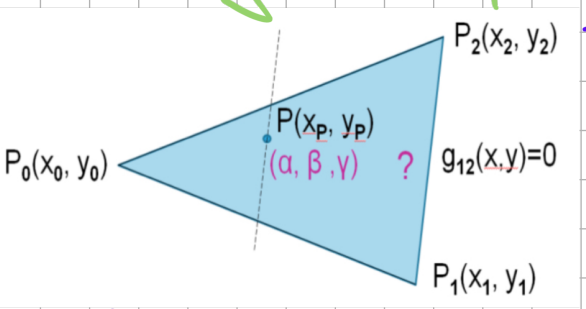
barycentrische Koordinaten (α, β, γ)

= Darstellung der Punkte als gewichtetes Mittel der Ecken: $P = \alpha P_0 + \beta P_1 + \gamma P_3$, wobei $\alpha + \beta + \gamma = 1$

innerhalb des Dreiecks: $(0<\alpha<1, 0<\beta<1, 0<\gamma<1)$

Füllen des Dreiecks: Zeichnen all dieser ↑ Punkte

Berechnung der Baryzentrischen Koordinaten

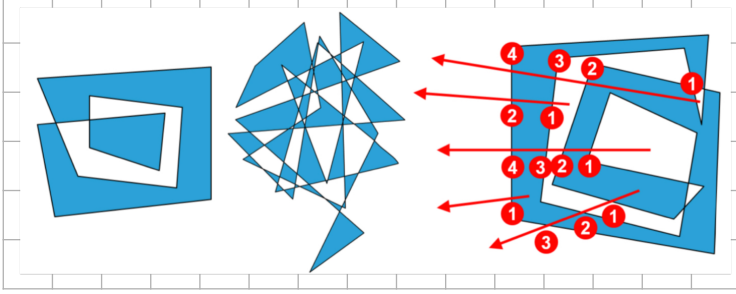


Trägergerade: $g_{12}(x,y) = a_{12}x + b_{12}y + c_{12} = 0$

α von $P(x_p, y_p)$: $\alpha = g_{12}(x_p, y_p) / g_{12}(x_0, y_0)$

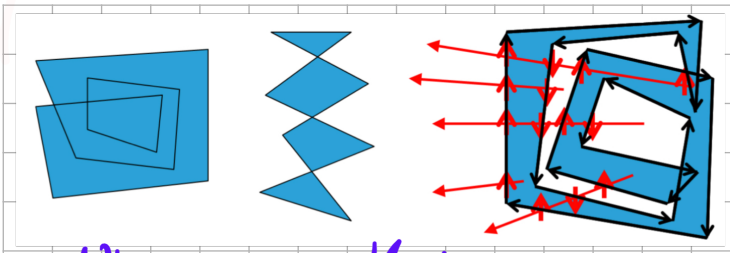
damit Pixel nur einmal zeichnen: nur Pixel mit Mittelpunkt innerhalb zeichnen, Pixel auf Kanten speziell behandeln

Rasterisierung von Polygonen



Odd-Even-Rule:

innerhalb, wenn Schnittanzahl ungerade, sonst innerhalb



Nonzero-Winding-Number-Rule: außerhalb, wenn Kantenanzahl im Uhrzeigersinn = Kantenanzahl gegen Uhrzeigersinn, sonst innen

All-In-Rule: Alles, was umschlossen ist, ist drinnen

konvexes Polygon: alle inneren Winkel < 180°, andernfalls konkav

Scanline-Flächenfüllen

= Scanline

⇒ zeilenweises Füllen, getrennte Berechnung für jede Rasterzeile

Inkrementelle Schnittpunktberechnung:

⇒ Wiederverwendung von möglichst viel Info

⇒ Bearbeitung von unten nach oben

y = max. y-Wert (dient für sortierung)

x = x-Wert des 1. Schnittpkt., $\frac{1}{m}$ = inverse Steigung

zweite kurze Liste: Kanten, die Scanline schneiden

aktive Kantenliste: zu Beginn leer → Zeilenwechsel → überprüfen, ob

(hier a.K.)

Sortierung nach Schnittpunkten von links nach rechts

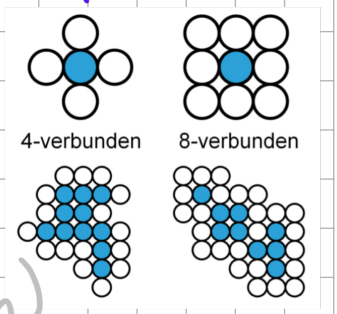
- vorderste Kante auf Scanline beginnt → nein = alle anderen Kanten irrelevant, ja = in a.K. übernehmen
- & Löschen → Kanten, mit überschrittenem max y löschen

inkrementelle Schnittpunktberechnung: $x_{k+1} = x_k + \frac{1}{m}$ & $y_{k+1} = y_k + \frac{1}{m}$

falls Eckpunkt auf Scanline: Verschiebung um kleinen Wert ϵ

4-verbunden vs. 8-verbunden

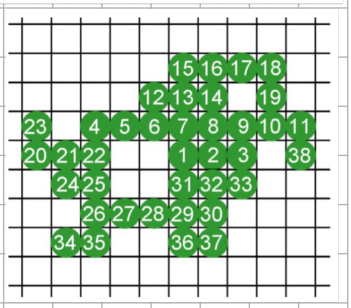
⇒ analog zu Nachbarschaften (siehe CV: lokale Operationen)
8-verbundene Grenze reicht für 4-verbundene Fläche, aber nicht umgekehrt (8=schwächer als 4)



rekursiver Foodfill Algorithmus (für 4-verb. Flächen)

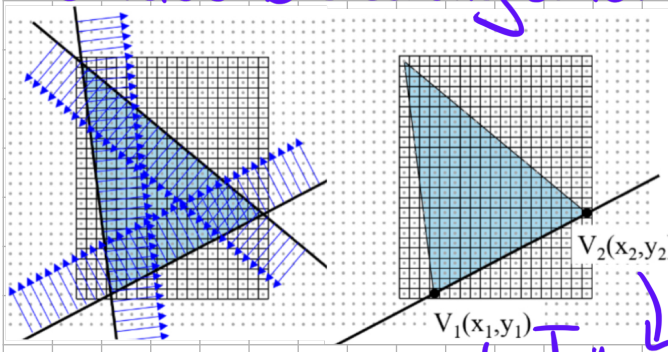
```
void floodFill4 (int x, int y, int new, int old)
{ int color;
    /* set current color to new */
    getPixel (x, y, color);
    if (color = old) {
        setPixel (x, y);
        floodFill4 (x-1, y, new, old);   /* left */
        floodFill4 (x, y+1, new, old);   /* up */
        floodFill4 (x+1, y, new, old);   /* right */
        floodFill4 (x, y-1, new, old)    /* down */
}}
```

⇒ sehr hohe Rekursionstiefe, daher horizontale Richtung meist iterativ implementiert, wie hier → Rekursionstiefe meist proportional zur Zeilenanzahl



Paralleles Füllen konvexer Polygone

⇒ schnellere Bearbeitung durch gleichzeitige Bearbeitung aller Pixel



getrennte Behandlung von diesen

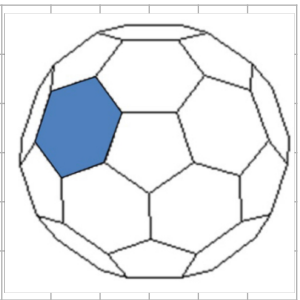
untersuchender Bereich = engstes Rechteck, was das Polygon umschließt

↳ Trägergerade: $(y_1-y_2)x+(x_2-x_1)y+x_1y_2-x_2y_1=0$

wenn Mittelpunkt > 0, dann Pixel innen, sonst außerhalb

Behandlung von 3D-Polygonen

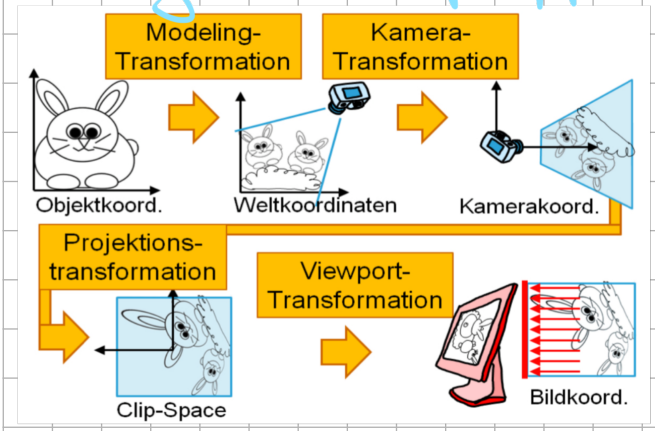
3D-Polygone = Oberflächenelemente von Objekten



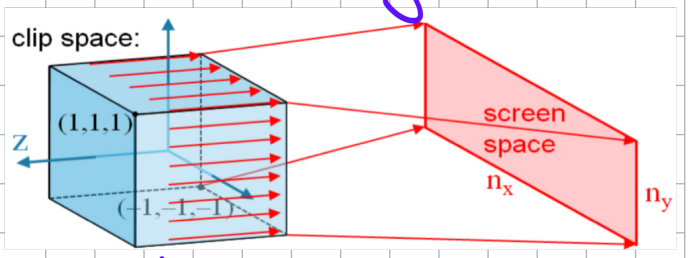
Attribute: Farbe, Materialparameter, Textur, Transparenz, geometrische Mikrostruktur, Reflexionsverhalten

⇒ an Eckpunkten Normalvektoren / Texturkoordinaten angegeben

Viewing in der Graphikpipeline



- Modellierung: Umwandlung durch Transformationen (s.S.5)
- Clip-Space = normierter Würfel (meist der Länge 2)



Viewport-Transformation

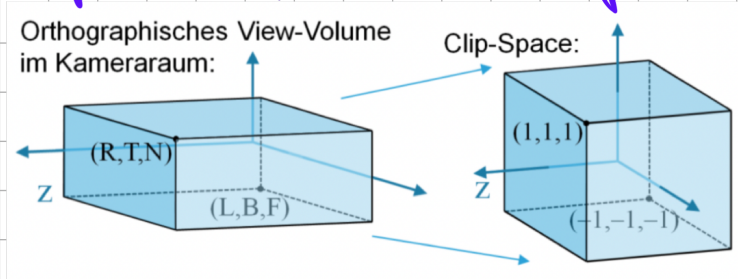
orthographische Abbildung:

$$\begin{bmatrix} x_{screen} \\ y_{screen} \\ z \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} n_x/2 & 0 & 0 & n_x/2 \\ 0 & n_y/2 & 0 & n_y/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{M_{vp}} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- z = Blickrichtung, Abmessungen: $n_x \times n_y$ Pixel
- Abbildung von $(-1,-1,z)$ auf $(0,0)$
- Abbildung von $(1,1,z)$ auf (n_x, n_y)

Projektionstransformation

⇒ achsenparallelen Quader mit Grenzen L(eft), R(ight), B(ottom), T(op), N(ear), F(ar) zu Würfel $[1,-1]^3$ verschieben & verzerren



$$M_{orth} = \begin{bmatrix} \frac{2}{R-L} & 0 & 0 & -\frac{R+L}{R-L} \\ 0 & \frac{2}{T-B} & 0 & -\frac{T+B}{T-B} \\ 0 & 0 & \frac{2}{N-F} & -\frac{N+F}{N-F} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

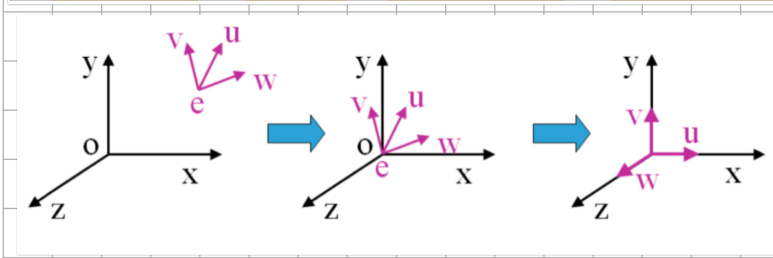
Kamera-Transformation

Viewing-Koordinaten: u (Kameraposition), v (Blickrichtung), w (Orientierung)

⇒ uv-Ebene normal auf Hauptblickrichtung (= Richtung negative w-Achse)

Festlegung:

1. Wahl einer Kameraposition (auch Augpunkt oder Viewing-Punkt genannt).
2. Wahl einer Blickrichtung, die negative Blickrichtung ergibt die w-Achse.
3. Wahl einer Richtung t „nach oben"; aus dieser lassen sich dann die u- und v-Achsen berechnen.
4. Da die Abbildungsebene normal auf die Blickrichtung liegt, ergibt das Vektorprodukt t × w die Richtung der u-Achse.
5. Berechnung der v-Achse als Vektorprodukt der w- und u-Achsen: v = w × u.



Welt- zu Viewingkoordinaten:

$$M_{WC,VC} = R_z \cdot R_y \cdot R_x \cdot T$$

Grenzen des darzustellenden Bereichs definieren:

6. Die Wahl von minimalen und maximalen u-, v- und w-Werten zur Eingrenzung des Ausschnittes der Szene, der abgebildet wird: L(eft), R(ight), B(ottom), T(op), N(ear), F(ar).

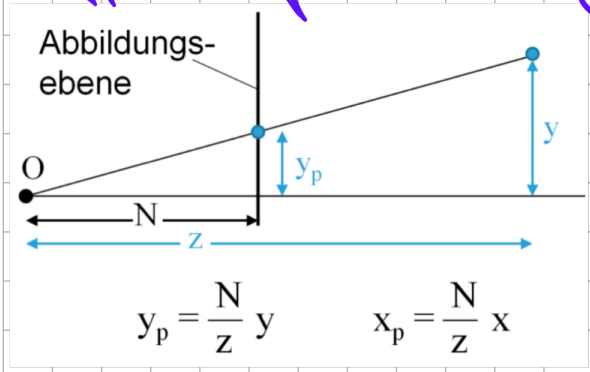
Orthographisches Viewing

$$\begin{bmatrix} x_{screen} \\ y_{screen} \\ z \\ 1 \end{bmatrix} = (M_{vp} \cdot M_{orth} \cdot M_{cam}) \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

orthographische Projektion = parallele Normalprojektion = Kamera bildet parallel ab

Perspektive

≠ affine Transformation, einziger Fall, in dem homogene Koordinate ≠ 1



O = Projektionszentrum der Abbildung

Blickrichtung = negative z-Richtung

Abbildungsebene normal auf z-Achse im Abstand N

Abgebildeter Punkt (x, y, z) hat Koordinaten $(x \cdot \frac{N}{z}, y \cdot \frac{N}{z}, N)$

$(x, y, z, 1) \cdot P = (x \cdot N, y \cdot N, z(N+F) - F \cdot N, z)$

$$\mathbf{P} = \begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & N+F & -F \cdot N \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

⇒ Homogenisierung: ↳ durch z dividieren

↳ = Verzerrung des Szenenbereichs (View Frustrum) in achsenparalleles Quader

$$\begin{bmatrix} x_{screen} \\ y_{screen} \\ z' \\ 1 \end{bmatrix} = M_{vp} \cdot \overbrace{(M_{orth} \cdot P \cdot M_{cam} \cdot M_{mod})}^{M_{per}} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

} Gesamtumwandlung von Modell- zu Gerätekoordinaten

Werkzeug: Clipping & Homogenisierung

Eigenschaften: - gerade Strecken bleiben gerade; Abbildung von diesen durch Eckpunkttransformation

- relativer Kamerabstand bleibt erhalten, wichtig für Sichtbarkeitsberechnung:

$z_1, z_2, N, F < 0$

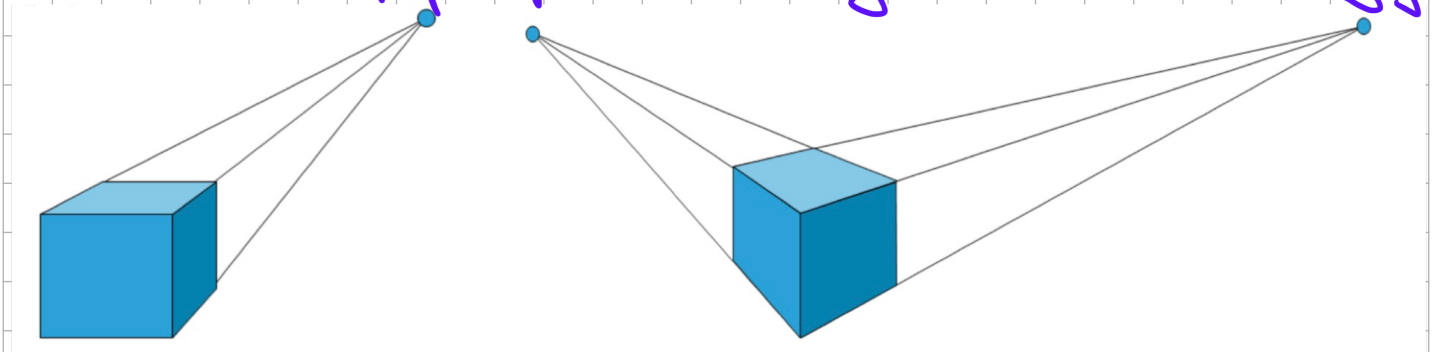
$z_1 < z_2$

$1/z_1 > 1/z_2 \quad | \cdot (-F \cdot N) \quad (<0)$

$-F \cdot N/z_1 < -F \cdot N/z_2 \quad | + (N+F)$

$(N+F) - F \cdot N/z_1 < (N+F) - F \cdot N/z_2$

Anzahl der Hauptfluchtpunkte von Lage der Bildebene abhängig:



Einpunktperspektive (1 Hauptfluchtpkt.)

Zweipunktperspektive (2 Hauptfluchtpkt.)