

Generative Computergrafik



Prof. Dr. Ulrich Schwanecke

RheinMain University of Applied Sciences
Wiesbaden Rüsselsheim

11. April 2018



Organisatorisches

- Vorlesung (Gebäude D, Raum 14) Achtung: **Inverted Classroom!**
 - Mittwoch, 08:15 – 09:45 (**Videos:** <https://video.cs.hs-rm.de/>)
- Übungen (Gruppen A – C, Gebäude D, Raum 12)
 - Mittwoch, C: 11:45 – 13:15, B: 14:15 – 15:45, A: 16:00 – 17:30
- Geplante Termine (insgesamt 13)

	April	11.	18.	25.	
2018	Mai	02.	09.	16.	23. 30.
	Juni	06.	13.	20.	27.
	Juli	04.			

- Vorlesungsfolien, Übungsblätter, weitere Informationen unter cvmr.info/teaching/computergrafik/
 - Name: GenCG Kennwort: ss18

Bewertung

Vorlesung:	Klausur
Praktikum:	Drei bewertete Abgaben



Ziele der Veranstaltung

- *Überblick* über das Gebiet der Computergrafik
 - Geschichte/Entwicklung (Einordnung/Vergleich im CV Umfeld)
 - Anwendungsgebiete
- *Verständnis* grundlegender Konzepte
 - Geometrie-/Render-Pipeline (Lokale Beleuchtungsmodelle)
 - Euklidische, Affine, Projektive Transformationen
 - Darstellung von Punkten, Linien und Polygonen auf Rasterbildschirmen
 - Farben, Licht, Beleuchtungsmodelle, Schattierungsverfahren, Texturen
 - Unterschied zwischen Grafik API und Szenengraph
 - Darstellung von Kurven und Flächen
 - Polynome in Monom, Lagrange und Bézier Darstellung, B-Splines
 - Photorealistische Darstellungsmethoden (Globale Beleuchtung)
 - Raytracing, Radiosity
- *Erwerb praktischer Kenntnisse*
 - Grafik API OpenGL
 - Globale Beleuchtung mittels Raycasting/Raytracing



Bibliographie

- Vorlesungsskript (cvmr.info/teaching/computergrafik/)
- J. D. Foley, A. van Dam, S. K. Feiner: *Computer Graphics: Principles and Practice*, Addison Wesley, 2012
- E. Angel, D. Shreiner : *Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL*, Addison Wesley, 2011
- R. S. Wright, N. Haemel, G. Sellers, B. Lipchak: *OpenGL SuperBible: Comprehensive Tutorial and Reference*, Addison Wesley, 2011
- T. Akenine-Möller, E. Haines, N. Hoffman, *Real-time Rendering*, Peters, Wellesley, 2008
- S. Riley, T. Riley: *Game Programming with Python*, Cengage Learning Emea, 2003



Online Quellen

- Alles
 - www.google.de
 - www.bing.com
- Python
 - www.python.org
- Grafik API OpenGL/Mesa, GLUT
 - www.opengl.org
 - www.mesa3d.org
 - freeglut.sourceforge.net
 - nehe.gamedev.net
 - www.xmission.com/~nate/opengl.html
- Szenengraphen OpenInventor/Coin, Java3D
 - www.coin3d.org
 - java3d.java.net



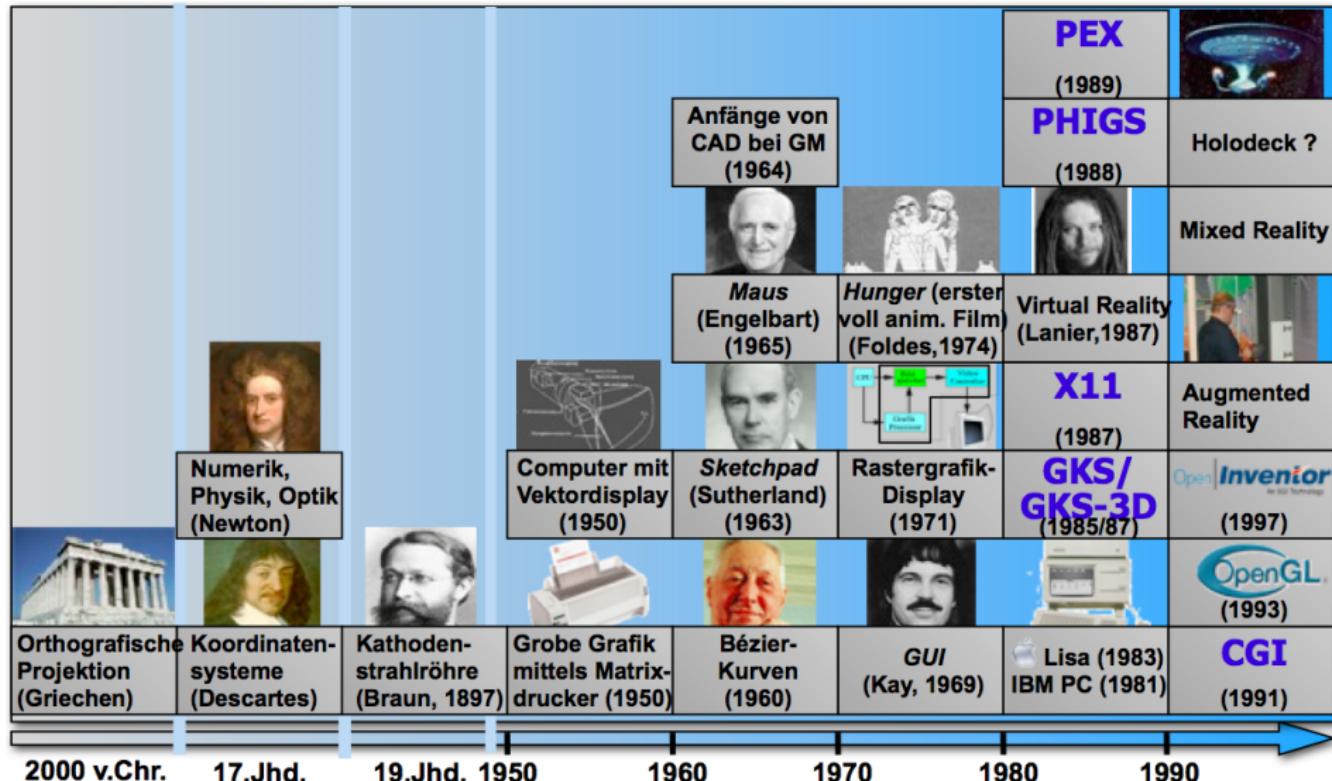
Was ist generative Computergrafik?

Klassisches Schema nach Rosenfeld



- Ursprünglich strenge Trennung zwischen *Computergrafik* und *Bildverarbeitung*
- Heute *integrierte Betrachtung* durch verbindende Anwendungen (z.B. Mixed Reality, ...)

Eine kurze Geschichte der Computergrafik



Typische Anwendungsgebiete von Computergrafik

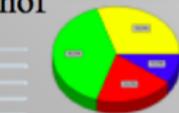
Kartografie

Landkarte,
Flächen-
nutzungs-
plan,
...



Business Grafik

Balken & Torten statt
Zahlenfriedhof



100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

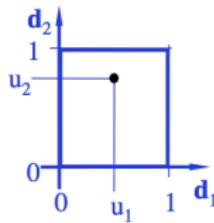
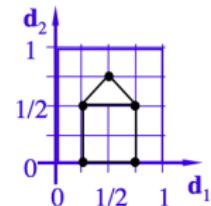
100%

100%

100%

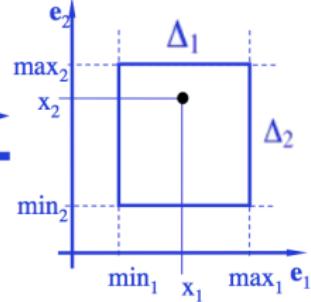
Koordinatensysteme

- Koordinatensysteme wurden von René Descartes (1596-1650) eingeführt
 - Festlegung eines Ursprungs und mehrere (zueinander senkrechter) Koordinaten-Achsen
- Einzelne Objekte werden in *lokalen Koordinatensystemen* dargestellt
 - Unabhängig von der späteren Größe, Position und Orientierung
- Alle Objekte zusammen werden in einem *globalen Koordinatensystem* dargestellt
 - Umrechnung zwischen lokalem und globalem Koordinatensystem



$$\begin{aligned}x_1 &= \min_1 + u_1 \Delta_1 \\x_2 &= \min_2 + u_2 \Delta_2\end{aligned}$$

$$\begin{aligned}u_1 &= \frac{x_1 - \min_1}{\Delta_1} \\u_2 &= \frac{x_2 - \min_2}{\Delta_2}\end{aligned}$$



Lokales Koordinatensystem



Beispiel: Auswahl eines Icons

Ein Maus-Klick liefert die *Bildschirmkoordinaten* (380,220)

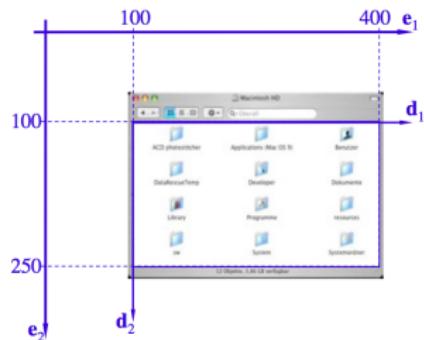
Frage: Welches Icon wird aktiviert?

Lösung

- Berechne lokale (normalisierte) Koordinaten

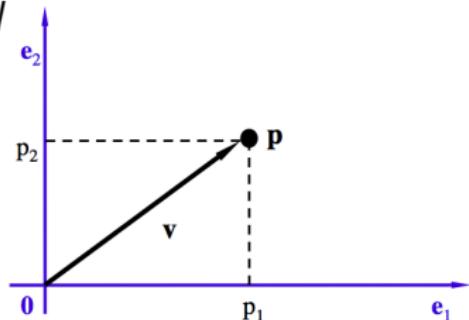
$$u_1 = \frac{380 - 100}{300} \approx 0.93, \quad u_2 = \frac{220 - 100}{150} \approx 0.80$$

- u_1 -Zerlegung der normalisierten Koordinaten ist $0, 1/3, 2/3, 1$
 - 0.93 liegt zwischen $2/3$ und 1, also wurde ein Icon in der *dritten Spalte* getroffen
- u_2 -Zerlegung der normalisierten Koordinaten ist $0, 1/4, 1/2, 3/4, 1$
 - 0.8 liegt zwischen $3/4$ und 1, also wurde ein Icon in der *vierten Zeile* getroffen

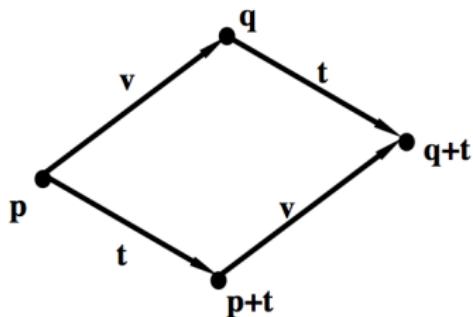


Punkte und Vektoren

- Ein *Punkt* \mathbf{p} wird durch ein *Koordinatentupel* repräsentiert: $\mathbf{p} = (p_1, p_2)$, $\mathbf{p} = (p_1, p_2, p_3)$
- Ein (*Ort-*)*Vektor* \mathbf{v} von \mathbf{p} wird *ebenfalls* durch ein Koordinatentupel repäsentiert:
 $\mathbf{v} = \mathbf{p} - \mathbf{0}$



- Zu zwei Punkten \mathbf{p}, \mathbf{q} existiert ein *eindeutiger Vektor* $\mathbf{v} = \mathbf{q} - \mathbf{p}$ von \mathbf{p} nach \mathbf{q}
- Zu einem Vektor \mathbf{v} existieren *unendlich viele verschiedene* Punkte \mathbf{p}, \mathbf{q} mit $\mathbf{v} = \mathbf{q} - \mathbf{p}$
- Vektoren sind *invariant* (ändern sich nicht) unter *Translation*
- Punkte sind *nicht translationsinvariant*



Addition von Vektoren und Multiplikation mit einem Skalar

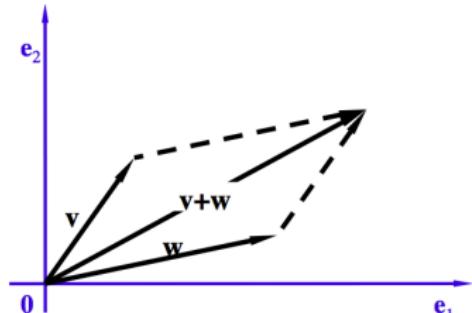
Addition/Subtraktion von Vektoren

- Algebraisch: Addition der Komponenten

$$\begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} + \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} v_1 + w_1 \\ v_2 + w_2 \\ v_3 + w_3 \end{pmatrix}$$

- Geometrisch: Aneinandersetzen von "Pfeilen"

Anwendung in der Computergrafik: Verschieben von Objekten (Translation)



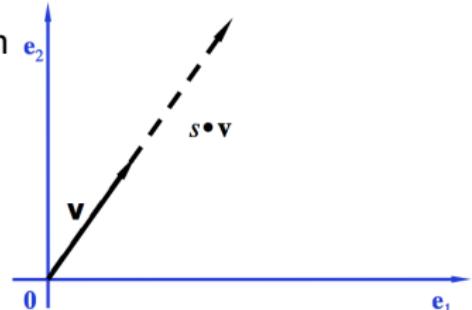
Multiplikation mit einem Skalar s

- Algebraisch: Komponentenweise Multiplikation

$$s \cdot \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} s \cdot v_1 \\ s \cdot v_2 \\ s \cdot v_3 \end{pmatrix}$$

- Geometrisch: Änderung der Vektorlänge um s

Anwendung in der Computergrafik: Skalieren, Spiegeln, Zoomen



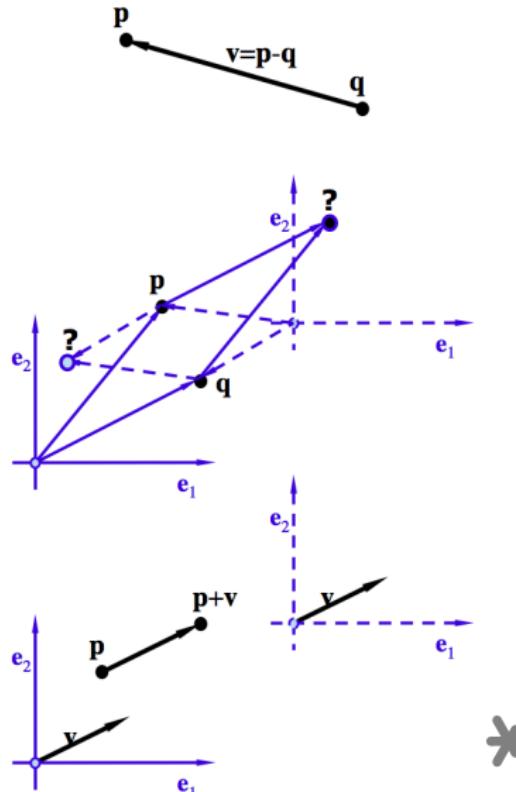
Addition von Punkten

Subtraktion und Addition von Punkten

- Durch Subtraktion zweier Punkte p und q erhält man einen Vektor v
- Die (einfache) *Addition von Punkten ist nicht definiert*
- Die Addition von Punkt und Vektor ist definiert und liefert einen Punkt

Gibt es eine sinnvolle Addition von Punkten?

Ja, die *baryzentrische Kombination*



Baryzentrische Kombination

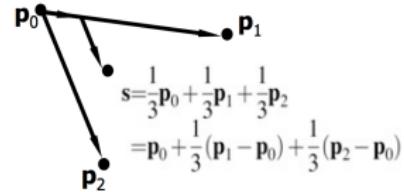
Punkte lassen sich addieren, d.h. $\mathbf{q} = \sum_{i=0}^n \alpha_i \mathbf{p}_i$ macht Sinn, wenn

- 1 Koeffizientensumme $\sum_{i=0}^n \alpha_i = 1 \implies \mathbf{q} = \mathbf{p}_0 + \sum_{i=1}^n \alpha_i (\mathbf{p}_i - \mathbf{p}_0)$

- *Ergebnis ist ein Punkt* (Summe eines Punktes und eines Vektors)
- Man spricht dann von *baryzentrischer Kombination*
- Sind alle $\alpha_i > 0$ spricht man von einer *Konvexitätskombination*

- *Anwendung:* Für den *Schwerpunkt* \mathbf{s} eines Objekts mit den Punkten $\mathbf{p}_0, \dots, \mathbf{p}_n$ gilt:

$$\mathbf{s} = \frac{1}{n+1} \sum_{i=0}^n \mathbf{p}_i$$



- 2 Koeffizientensumme $\sum_{i=0}^n \alpha_i = 0 \implies \mathbf{q} = \sum_{i=1}^n \alpha_i (\mathbf{p}_i - \mathbf{p}_0)$
- *Ergebnis ist ein Vektor* (Summe von Vektoren)

Das Skalarprodukt

- Länge l eines Vektors $\mathbf{v} = (v_1, v_2, v_3)$:

$$l = \|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2 + v_3^2}$$

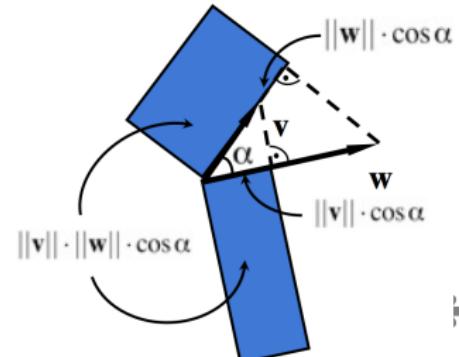
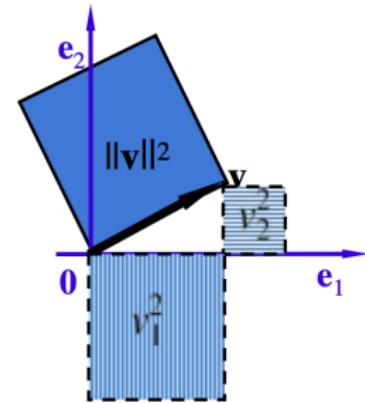
- Skalarprodukt zweier Vektoren \mathbf{v}, \mathbf{w} :

$$\langle \mathbf{v}, \mathbf{w} \rangle = v_1 w_1 + v_2 w_2 + v_3 w_3$$

- Es gilt $l = \|\mathbf{v}\| = \sqrt{\langle \mathbf{v}, \mathbf{v} \rangle}$
- Winkel α zwischen zwei Vektoren \mathbf{v}, \mathbf{w} :

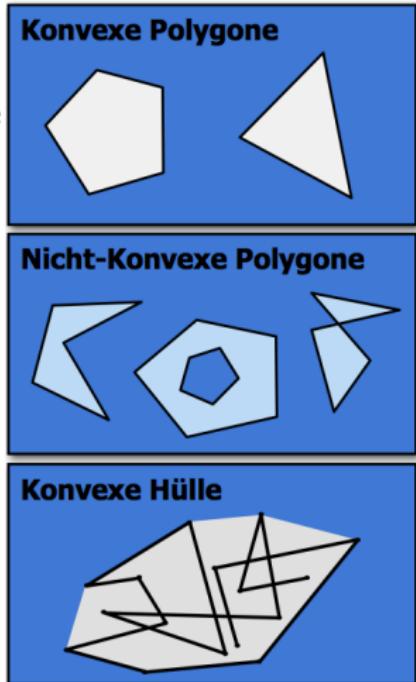
$$\langle \mathbf{v}, \mathbf{w} \rangle = \|\mathbf{v}\| \|\mathbf{w}\| \cdot \cos \alpha$$

- Vektoren \mathbf{v}, \mathbf{w} senkrecht, wenn $\langle \mathbf{v}, \mathbf{w} \rangle = 0$
- Drehtrick (nur in 2D):
 - $\mathbf{v} = (v_1, v_2)$ steht senkrecht auf $\mathbf{w} = (-v_2, v_1)$ (90° Drehung gegen den Uhrzeigersinn)



Konvexe Polygone und konvexe Hülle

- Ein Polygon P heißt *konvex*, wenn die folgenden äquivalenten Bedingungen erfüllt sind
 - Mit 2 Punkten liegt auch deren Verbindung in P
 - Der Schnitt von P und einer Gerade ist immer eine Strecke
 - Das Innere von P ist durch ein lineares Ungleichungssystem beschreibbar
- Alle *Dreiecke sind konvexe Polygone*
 - Die meisten Grafiksystem unterstützen nur konvexe Polygone (nur Dreiecke)
- Vorteile konvexer Polygone
 - Einfache Verwaltung der Schnitte von P und Geraden
 - Einfacher Test von “Punkt in Polygon”
- *Konvexe Hülle* einer Punktmenge (eines Polygons)
 - Das konvexe Polygon durch die Punkte, das alle Punkte in seinem Inneren enthält



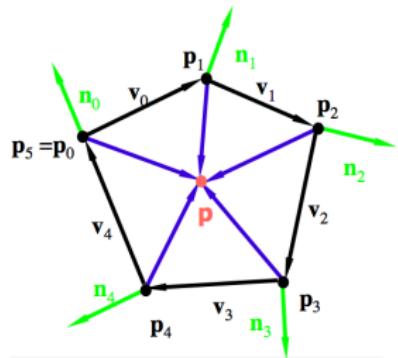
Anwendung: Punkt \mathbf{p} in konvexem Polygon?

Test: Punkt \mathbf{p} in konvexem Polygon $\mathbf{p}_0, \dots, \mathbf{p}_5 = \mathbf{p}_0$ (im Uhrzeigersinn) ?

- 1 Die Kanten des Polygons sind $\mathbf{v}_i = \mathbf{p}_{i+1} - \mathbf{p}_i$
- 2 Berechne die Normalen \mathbf{n}_i (Drehtrick)
- 3 \mathbf{p} liegt innerhalb des Polygons, wenn

$$\langle \mathbf{n}_i, (\mathbf{p} - \mathbf{p}_i) \rangle < 0 \quad \forall i = 0, \dots, n-1$$

- Gilt $\langle \mathbf{n}_i, (\mathbf{p} - \mathbf{p}_i) \rangle > 0$ für ein i , liegt \mathbf{p} außerhalb
- \mathbf{p} liegt auf dem Rand des Polygons, wenn
 $\langle \mathbf{n}_i, (\mathbf{p} - \mathbf{p}_i) \rangle \leq 0 \quad \forall i = 0, \dots, n-1$, wobei
mindestens einmal “=” gilt



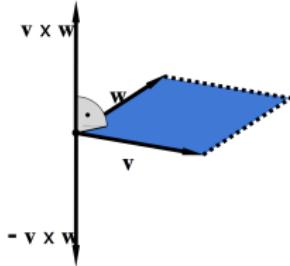
```
def pointInPolygon(point, convexPolygon):
    """_Test,_ob,_point,_in,_convexPolygon,_liegt,_"""
    for (p,q) in zip(convexPolygon, convexPolygon[1:]):
        if (q-p).Normal()*(point-p) > 0:
            return False
    return True
```

Das Vektorprodukt und der Axiator

Das Vektorprodukt für zwei Vektoren \mathbf{v}, \mathbf{w} im (3D) Raum

$$\begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \times \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{vmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{vmatrix} = \begin{pmatrix} v_2 w_3 - v_3 w_2 \\ v_3 w_1 - v_1 w_3 \\ v_1 w_2 - v_2 w_1 \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{pmatrix}}_{=:[\mathbf{v}] \times} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix}$$

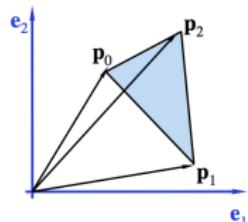
- Die Matrix $[\mathbf{v}] \times$ heißt *Axiator* von \mathbf{v}
- Es gilt $\mathbf{v} \times \mathbf{w} = [\mathbf{v}] \times \mathbf{w} = -[\mathbf{w}] \times \mathbf{v} = -\mathbf{w} \times \mathbf{v}$
- $\mathbf{v} \times \mathbf{w}$ steht *senkrecht* auf der durch \mathbf{v} und \mathbf{w} definierten Ebene
- Die Länge von $\mathbf{v} \times \mathbf{w}$ ist gleich dem *Flächeninhalt* des durch \mathbf{v} und \mathbf{w} definierten Parallelogramms
- Gilt $\mathbf{v} \times \mathbf{w} = 0$, so sind die Vektoren \mathbf{v} und \mathbf{w} kollinear (parallel)



Anwendung: Flächeninhalt eines Polygons

Für den Flächeninhalt $F(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)$ eines Dreiecks $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ gilt

$$\begin{aligned} 2 \cdot F(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2) &= \|(\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)\| \\ &= \|\mathbf{p}_0 \times \mathbf{p}_1 + \mathbf{p}_1 \times \mathbf{p}_2 + \mathbf{p}_2 \times \mathbf{p}_0\| \end{aligned}$$



Für den Flächeninhalt $F(\mathbf{p}_0, \dots, \mathbf{p}_n)$ eines Polygons gilt analog

$$\begin{aligned} 2 \cdot F(\mathbf{p}_0, \dots, \mathbf{p}_n) &= \|\mathbf{p}_0 \times \mathbf{p}_1 + \mathbf{p}_1 \times \mathbf{p}_2 + \dots + \mathbf{p}_{n-1} \times \mathbf{p}_n + \mathbf{p}_n \times \mathbf{p}_0\| \\ &= \sum_{i=0}^n (p_i)_1 ((p_{i+1})_2 - (p_{i+1})_1 (p_i)_2) \text{ (modulo } n+1) \end{aligned}$$

Durch Umformung erhält man

$$F(\mathbf{p}_0, \dots, \mathbf{p}_n) = \frac{1}{2} \sum_{i=0}^n (p_i)_1 ((p_{i+1})_2 - (p_{i-1})_2) \text{ (modulo } n+1)$$

Beispiel zur Berechnung des Flächeninhalts

Für den Flächeninhalt von $\mathbf{P} = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 2 \\ 0 \end{pmatrix}, \begin{pmatrix} 2 \\ 2 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 2 \end{pmatrix} \right\}$

erhält man mit

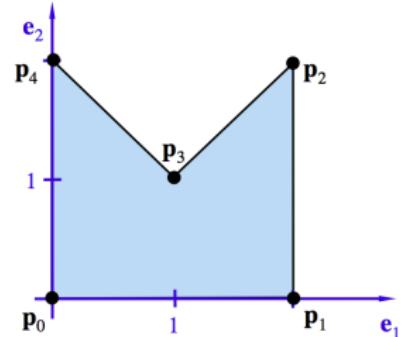
$$x_i = (p_i)_1, \quad y_i = (p_i)_2$$

als Ergebnis

$$F(\mathbf{P}) = \frac{1}{2} \sum_{i=0}^4 x_i(y_{i+1} - y_{i-1})$$

$$= \frac{1}{2}(0 \cdot (0 - 2) + 2 \cdot (2 - 0) + 2 \cdot (1 - 0) + 1 \cdot (2 - 2) + 0 \cdot (0 - 1))$$

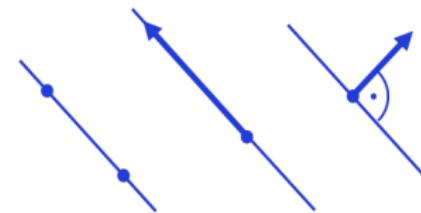
$$= 3$$



Geraden in der Ebene

Eine *Gerade* ist definiert durch Angabe von

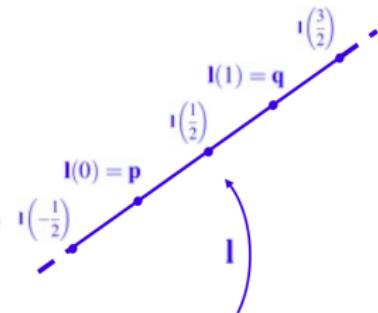
- zwei Punkten oder
- Punkt und Vektor parallel zur Geraden oder
- Punkt und Vektor senkrecht zur Geraden



Parameterdarstellung einer Geraden

$$\mathbf{l}(t) = (1 - t)\mathbf{p} + t\mathbf{q}$$

- \mathbf{p}, \mathbf{q} sind Punkte (2D oder 3D), $t \in \mathbb{R}$
- $\mathbf{l}(t_0)$ ist der Geradenpunkt zum Parameter t_0
(*Lineare Interpolation* zwischen \mathbf{p}, \mathbf{q} für $t_0 \in [0, 1]$)
- Mit dem Vektor $\mathbf{v} = \mathbf{q} - \mathbf{p}$ erhält man



$$\mathbf{l}(t) = \mathbf{p} + t\mathbf{v}$$

$\mathbf{---} t < 0 \mathbf{---} 0 \leq t \leq 1 \mathbf{---} t > 1 \mathbf{---}$

Implizite Darstellung einer Geraden

Mit Punkt \mathbf{p} und Vektor \mathbf{w} senkrecht zur Geraden erhält man die *implizite Geradendarstellung*

$$\langle \mathbf{w}, \mathbf{x} - \mathbf{p} \rangle = 0$$

- Für $\|\mathbf{w}\| = 1$ (Normalenvektor) heißt diese Darstellung *Hesse'sche Normalform*
- Komponentenschreibweise:

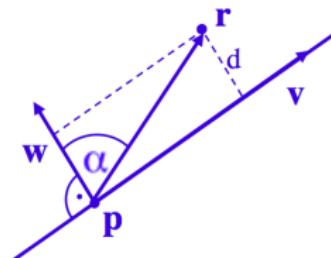
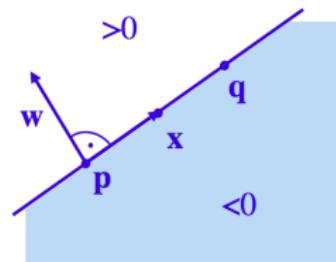
$$ax_1 + bx_2 + c = 0 \text{ mit}$$

$$a = w_1, b = w_2, c = -(w_1 p_1 + w_2 p_2)$$

- Für den Abstand d von Punkt \mathbf{r} zur Geraden gilt:

$$d = \frac{\langle \mathbf{w}, \mathbf{r} - \mathbf{p} \rangle}{\|\mathbf{w}\|}$$

denn $\langle \mathbf{w}, \mathbf{r} - \mathbf{p} \rangle = \|\mathbf{w}\| \cdot \|\mathbf{r} - \mathbf{p}\| \cdot \cos \alpha$ und $\cos \alpha = d / \|\mathbf{r} - \mathbf{p}\|$



Konvertierung zwischen parametrischer & impliziter Form

Von *parametrisch* $\mathbf{l}(t) = \mathbf{p} + t\mathbf{v}$ nach *implizit* $ax_1 + bx_2 + c = 0$

- 1 Vektor $\mathbf{w} = (-v_2, v_1)^T$ senkrecht zu \mathbf{v} (man erhält $a = -v_2, b = v_1$)
- 2 Durch Auflösen nach c erhält man $c = v_2 p_1 - v_1 p_2$

Beispiel: Eine implizite Darstellung von $\mathbf{l}(t) = \binom{2}{2} + t\binom{4}{2}$ ist
 $-2x_1 + 4x_2 - 4 = 0$

Von *implizit* $ax_1 + bx_2 + c = 0$ nach *parametrisch* $\mathbf{l}(t) = \mathbf{p} + t\mathbf{v}$

- 1 Wähle $\mathbf{v} = (-b, a)^T$ parallel zu \mathbf{l}
- 2 Wähle $\mathbf{p} = \begin{cases} \left(\begin{smallmatrix} -\frac{c}{a} \\ 0 \end{smallmatrix}\right) & \text{falls } |a| > |b| \\ \left(\begin{smallmatrix} 0 \\ -\frac{c}{b} \end{smallmatrix}\right) & \text{sonst} \end{cases}$

Beispiel: Eine Parameterdarstellung von $-2x_1 + 4x_2 - 4 = 0$ ist
 $\mathbf{l}(t) = \binom{0}{1} + t\binom{-4}{2} = \binom{2}{2} + t\binom{4}{2}$

Homogene Geraden-Vektoren

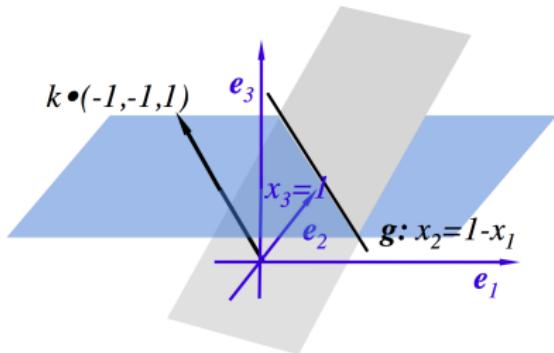
Implizite Geradengleichung $ax_1 + bx_2 + c = 0$

Alle Vektoren $k \cdot (a, b, c)^T, k \neq 0$ repräsentieren die gleiche Gerade

Ein Vektor $k \cdot (a, b, c)^T, k \neq 0$ heißt *homogener (Geraden-)Vektor*

Ein Punkt $\mathbf{p} = (p_1, p_2)$ liegt genau dann auf der Geraden (a, b, c) , wenn

$$ap_1 + bp_2 + c = 0$$

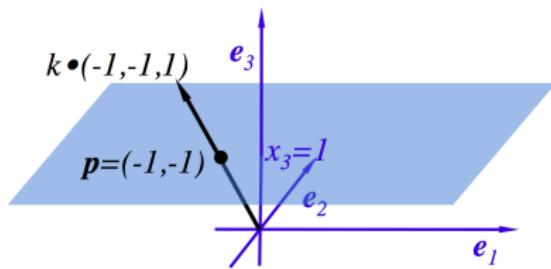


- Schreibweise mit Skalarprodukt: $\langle (p_1, p_2, 1)^T, (a, b, c)^T \rangle = 0$
- Bedingung gilt $\iff k \cdot \langle (p_1, p_2, 1)^T, (a, b, c)^T \rangle = 0, k \neq 0$

Homogene Punkt-Vektoren

Analog zu den Geraden können auch Punkte durch *homogene Vektoren* dargestellt werden

- Jeder homogener Vektor $k \cdot (x_1, x_2, x_3)^T$ mit $k, x_3 \neq 0$ repräsentiert den *Punkt* $(x_1/x_3, x_2/x_3)^T$
- Ein homogener Vektor $(x_1, x_2, 0)^T$ repräsentiert den *Vektor* $(x_1, x_2)^T$

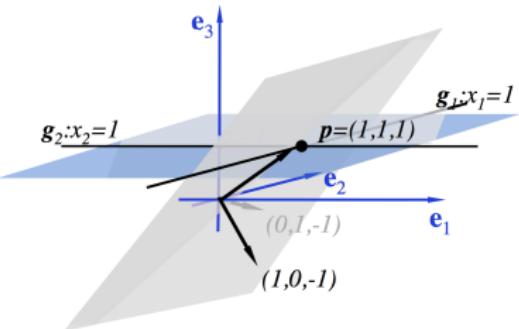


Schnittpunkt zweier Geraden

Zwei Geraden $\mathbf{g}_1, \mathbf{g}_2$ (in homogener Darstellung) schneiden sich im Punkt
 $\mathbf{p} = \mathbf{g}_1 \times \mathbf{g}_2 = [\mathbf{g}_1] \times \mathbf{g}_2 = -[\mathbf{g}_2] \times \mathbf{g}_1$

Beispiel

- Die Gerade $\mathbf{g}_1 : x = 1$ wird repräsentiert durch den homogenen Vektor $(1, 0, -1)$
- Die Gerade $\mathbf{g}_2 : y = 1$ wird repräsentiert durch den homogenen Vektor $(0, 1, -1)$
- $\mathbf{g}_1, \mathbf{g}_2$ schneiden sich im Punkt
 $\mathbf{p} = (1, 0, -1)^T \times (0, 1, -1)^T = (1, 1, 1)$

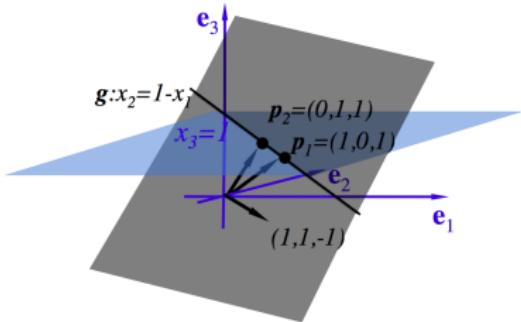


Gerade durch zwei Punkte

Durch die Punkte $\mathbf{p}_1, \mathbf{p}_2$ (in homogener Darstellung) geht die Gerade
 $\mathbf{g} = \mathbf{p}_1 \times \mathbf{p}_2 = [\mathbf{p}_1] \times \mathbf{p}_2 = -[\mathbf{p}_2] \times \mathbf{p}_1$

Beispiel

- Der Punkt $\mathbf{p}_1 = (1, 0)$ wird repräsentiert durch den homogenen Vektor $(1, 0, 1)$
- Der Punkt $\mathbf{p}_2 = (0, 1)$ wird repräsentiert durch den homogenen Vektor $(0, 1, 1)$
- Durch $\mathbf{p}_1, \mathbf{p}_2$ geht die Gerade
 $\mathbf{g} = (1, 0, 1)^T \times (0, 1, 1)^T = (1, 1, -1)^T$
 also $y = 1 - x$



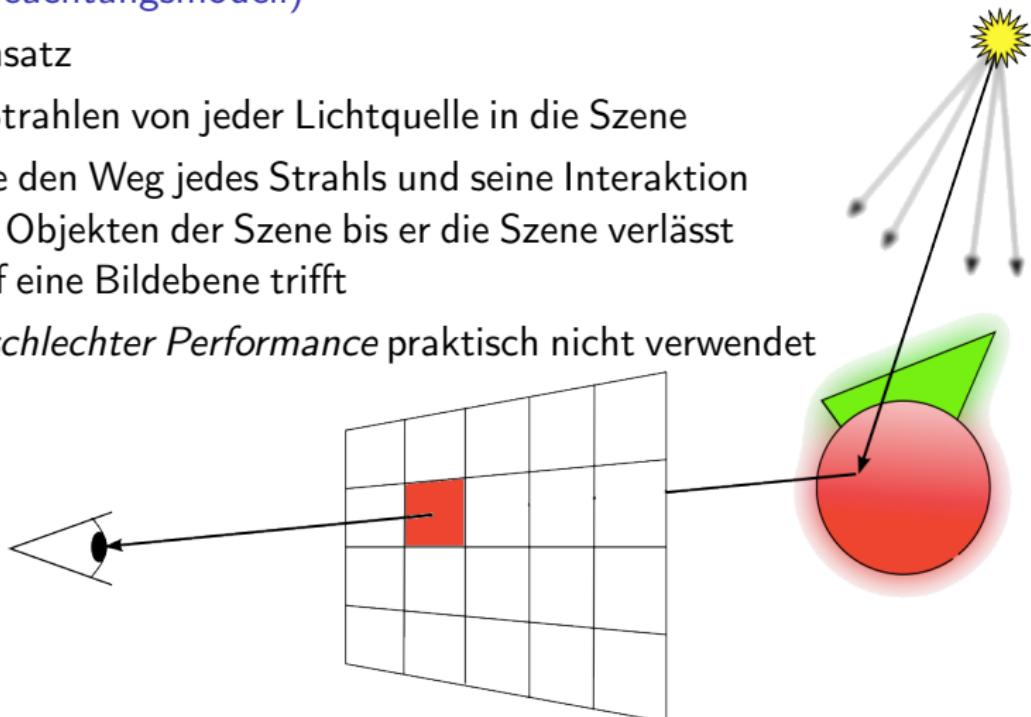
Raytracing als globales Beleuchtungsmodell

Raytracing (Strahlverfolgung) bestimmt die Lichtausbreitung in einer Szene (globales Beleuchtungsmodell)

Der naive Ansatz

- 1 Sende Strahlen von jeder Lichtquelle in die Szene
- 2 Verfolge den Weg jedes Strahls und seine Interaktion mit den Objekten der Szene bis er die Szene verlässt oder auf eine Bildebene trifft

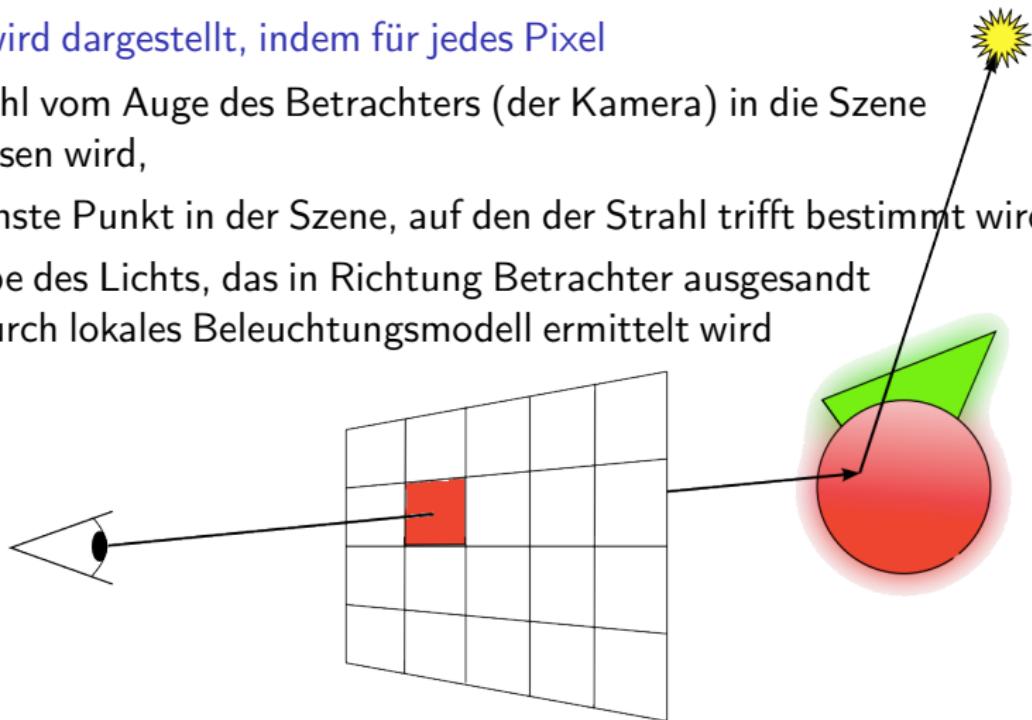
wird wegen *schlechter Performance* praktisch nicht verwendet



(Camera) Ray Casting / Backward Raytracing¹

Eine Szene wird dargestellt, indem für jedes Pixel

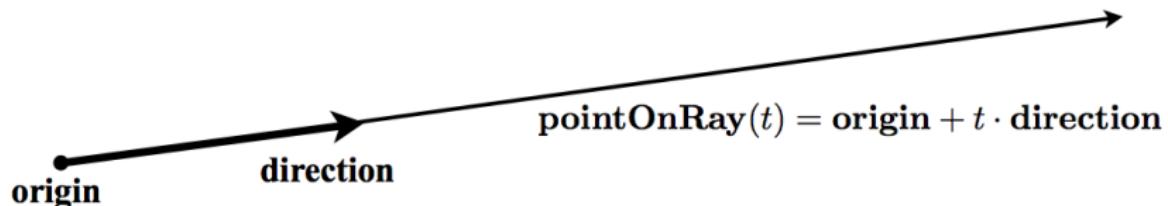
- 1 ein Strahl vom Auge des Betrachters (der Kamera) in die Szene geschossen wird,
- 2 der nächste Punkt in der Szene, auf den der Strahl trifft bestimmt wird,
- 3 die Farbe des Lichts, das in Richtung Betrachter ausgesandt wird, durch lokales Beleuchtungsmodell ermittelt wird



¹Arthur Appel, *Some techniques for machine rendering of solids*, in AFIPS Joint Computer Conferences, New York. 1968. pp. 37-45

Definition eines Strahls

Ein Strahl ist definiert durch seinen *Ursprung* und seine (normierte) *Richtung*



Eine *Strahl-Klasse* könnte wie folgt aussehen

```
class Ray(object):
    def __init__(self, origin, direction):
        self.origin = origin # point
        self.direction = direction.normalized() #vector

    def __repr__(self):
        return 'Ray(%s,%s)' % (repr(self.origin), repr(self.direction))

    def pointAtParameter(self, t):
        return self.origin + self.direction.scale(t)
```

Der Ray Casting Algorithmus

Für jedes Pixel

- 1 Berechne einen Strahl vom Auge (von der Kamera) durch das Pixel
- 2 Für jedes Objekt der Szene
 - Teste, ob der Strahl das Objekt trifft
- 3 Schattiere das Pixel unter Verwendung des nächstgelegenen Objekts
(oder setze die Pixelfarbe auf die Hintergrundfarbe)

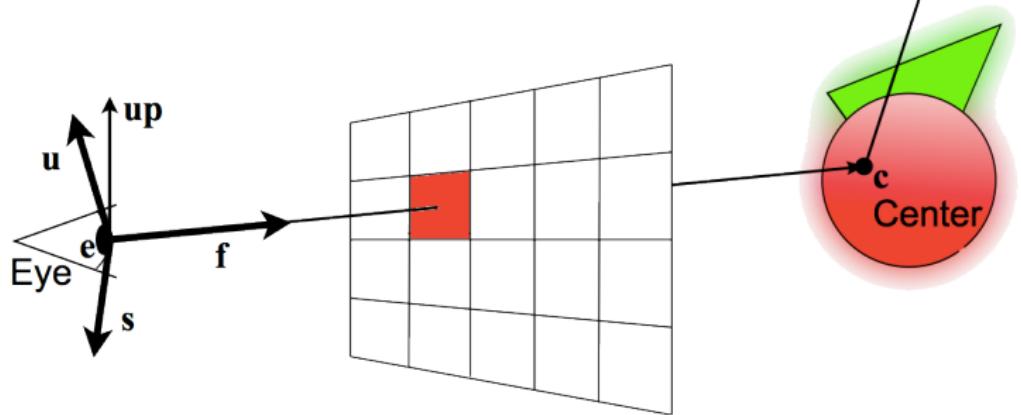
```
for x in range(imageWidth):
    for y in range(imageHeight):
        ray = calcRay(x, y)
        maxdist = float('inf')
        color = BACKGROUND_COLOR
        for object in objectlist:
            hitdist = object.intersectionParameter(ray)
            if hitdist:
                if hitdist < maxdist:
                    maxdist = hitdist
                    color = object.colorAt(ray)
        image.putpixel((x,y), color)
```

Definition eines Koordinatensystems

Kamera, positioniert im Ursprung, mit Blick in Richtung $-z$ liefert das Kamera (Augen) Koordinatensystem^a (extrinsische Kameraparameter)

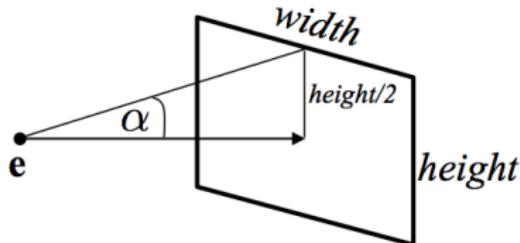
^awird oft auch *Look-At-Transformation*, abhängig von \mathbf{e} , \mathbf{c} , \mathbf{up} , genannt

$$\mathbf{f} = \frac{\mathbf{c} - \mathbf{e}}{\|\mathbf{c} - \mathbf{e}\|}, \quad \mathbf{s} = \frac{\mathbf{f} \times \mathbf{up}}{\|\mathbf{f} \times \mathbf{up}\|}, \quad \mathbf{u} = \mathbf{s} \times \mathbf{f}$$



Definition der Betrachtergeometrie

Definition eines Abbildungsbereichs abhängig von *Öffnungswinkel (field of view)* und *Seitenverhältnis (aspect ratio)* (intrinsische Kameraparameter)



$$\begin{aligned}\alpha &= \frac{\text{fieldofview}}{2} \\ \text{height} &= 2 \cdot \tan \alpha \\ \text{width} &= \text{aspectratio} \cdot \text{height}\end{aligned}$$

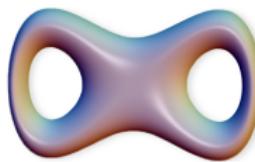
Sichtstrahl (pro Pixel) abhängig von der Auflösung (*wRes, hRes*)

```

pixelWidth = width / (wRes - 1)
pixelHeight = height / (hRes - 1)
for y in range(hRes):
    for x in range(wRes):
        xcomp = s.scale(x * pixelWidth - width / 2)
        ycomp = u.scale(y * pixelHeight - height / 2)
        ray = Ray(e, f + xcomp + ycomp) # evtl. mehrere Strahlen pro Pixel
    
```

Mögliche Primitivtypen

- Mittels Raytracing lassen sich Objekt (beleuchtet) darstellen,
 - für die der Schnittpunkt mit einem Strahl berechnet werden kann (z.B. Kugel, Ebene, Polygone (Dreiecke), implizit definierte Objekte, ...)
 - und die überall eine eindeutige Oberflächennormale besitzen
- Vorteil
 - Objekte mit "glatter" Oberfläche müssen nicht durch Polygone (Dreiecke) approximiert werden
 - Beispiel: $(x^2(1 - x^2) - y^2)^2 + \frac{1}{2}z^2 - \frac{1}{40}(1 + (x^2 + y^2 + z^2)) = 0^2$



- Nachteil
 - Jeder Primitivtyp erfordert eigenes Verfahren (oder numerische Lösung) zur Bestimmung der Schnittpunkte mit einem Strahl

²<http://virtualmathmuseum.org/Surface/bretzel2/bretzel2.html>



Implizite Definition einer Kugel

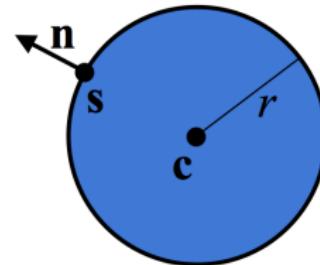
Eine *Kugel* ist definiert durch *Zentrum $\mathbf{c} \in \mathbb{R}^3$* und *Radius $r \in \mathbb{R}$*

- Für Punkte \mathbf{s} auf der Kugel gilt

$$\|\mathbf{c} - \mathbf{s}\| = r \implies \langle \mathbf{c} - \mathbf{s}, \mathbf{c} - \mathbf{s} \rangle = r^2$$

- Für die Normale \mathbf{n} der Kugel im Punkt \mathbf{s} gilt

$$\mathbf{n} = \frac{\mathbf{c} - \mathbf{s}}{\|\mathbf{c} - \mathbf{s}\|}$$

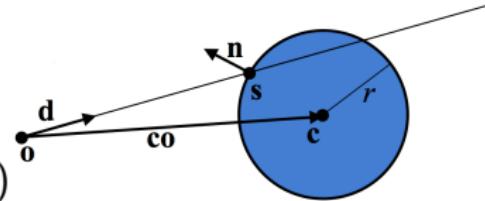


Schnittpunkt von Strahl und Kugel

Berechnung des Schnittpunktes von Strahl $\mathbf{r}(t) = \mathbf{o} + t \cdot \mathbf{d}$ und Kugel durch Einsetzen in die Kugelgleichung $\langle \mathbf{c} - \mathbf{s}, \mathbf{c} - \mathbf{s} \rangle = r^2$

- Mit $\mathbf{co} = \mathbf{c} - \mathbf{o}$ und $\|\mathbf{d}\| = 1$ gilt

$$\begin{aligned} 0 &= \langle \mathbf{co} - t \cdot \mathbf{d}, \mathbf{co} - t \cdot \mathbf{d} \rangle - r^2 \\ &= t^2 - 2 \cdot \langle \mathbf{co}, \mathbf{d} \rangle \cdot t + (\langle \mathbf{co}, \mathbf{co} \rangle - r^2) \end{aligned}$$



- Als Lösung(en) dieser quadratischen Gleichung erhält man

$$t_{1/2} = \langle \mathbf{co}, \mathbf{d} \rangle \pm \sqrt{\langle \mathbf{co}, \mathbf{d} \rangle^2 - (\langle \mathbf{co}, \mathbf{co} \rangle - r^2)}$$

- Für den "ersten" Schnittpunkt gilt

$$t = \langle \mathbf{co}, \mathbf{d} \rangle - \sqrt{\langle \mathbf{co}, \mathbf{d} \rangle^2 - (\langle \mathbf{co}, \mathbf{co} \rangle - r^2)}$$

Eine mögliche Kugelklasse

```
class Sphere(object):
    def __init__(self, center, radius):
        self.center = center # point
        self.radius = radius # scalar

    def __repr__(self):
        return 'Sphere(%s,%s)' %(repr(self.center), self.radius)

    def intersectionParameter(self, ray):
        co = self.center - ray.origin
        v = co.dot(ray.direction)
        discriminant = v*v - co.dot(co) + self.radius*self.radius
        if discriminant < 0:
            return None
        else:
            return v - math.sqrt(discriminant)

    def normalAt(self, p):
        return (p - self.center).normalized()
```

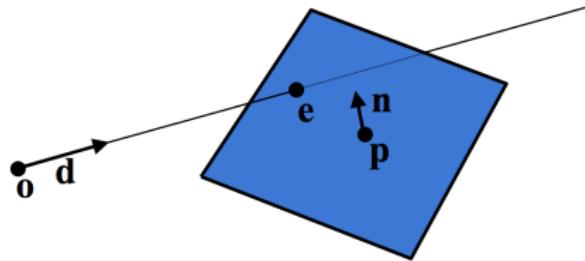


Schnittpunkt von Strahl und Ebene

Berechnung des Schnittpunktes von Strahl $\mathbf{r}(t) = \mathbf{o} + t \cdot \mathbf{d}$ und Ebene durch Einsetzen in die Ebenengleichung $0 = \langle \mathbf{e} - \mathbf{p}, \mathbf{n} \rangle$

Aus

$$\begin{aligned} 0 &= \langle \mathbf{e} - \mathbf{p}, \mathbf{n} \rangle \\ &= \langle \mathbf{o} + t \cdot \mathbf{d} - \mathbf{p}, \mathbf{n} \rangle \\ &= \langle \mathbf{o} - \mathbf{p}, \mathbf{n} \rangle + t \cdot \langle \mathbf{d}, \mathbf{n} \rangle \end{aligned}$$



folgt

$$t = -\frac{\langle \mathbf{o} - \mathbf{p}, \mathbf{n} \rangle}{\langle \mathbf{d}, \mathbf{n} \rangle}$$

Eine mögliche Ebenen-Klasse

```
class Plane(object):
    def __init__(self, point, normal):
        self.point = center # point
        self.normal = normal.normalized() # vector

    def __repr__(self):
        return 'Plane(%s,%s)' %(repr(self.point), repr(self.normal))

    def intersectionParameter(self, ray):
        op = ray.origin - self.point
        a = op.dot(self.normal)
        b = ray.direction.dot(self.normal)
        if b:
            return -a/b
        else:
            return None

    def normalAt(self, p):
        return self.normal
```

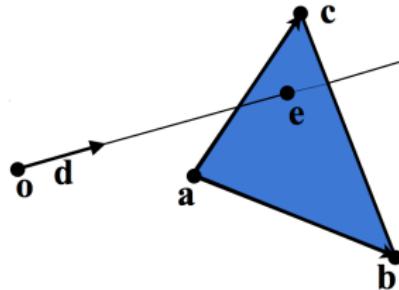


Schnittpunkt von Strahl und Dreieck

Berechnung des Schnittpunktes von Strahl $\mathbf{r}(t) = \mathbf{o} + t \cdot \mathbf{d}$ und Dreieck durch Einsetzen in die Ebenengleichung $\mathbf{e} = \mathbf{a} + r \cdot (\mathbf{b} - \mathbf{a}) + s \cdot (\mathbf{c} - \mathbf{a})$

führt zu dem Gleichungssystem

$$\begin{bmatrix} -\mathbf{d}, \underbrace{(\mathbf{b} - \mathbf{a})}_{=: \mathbf{u}}, \underbrace{(\mathbf{c} - \mathbf{a})}_{=: \mathbf{v}} \end{bmatrix} \begin{pmatrix} t \\ r \\ s \end{pmatrix} = \underbrace{(\mathbf{o} - \mathbf{a})}_{=: \mathbf{w}}$$



Mit der Cramerschen Regel und $\det(\mathbf{a}, \mathbf{b}, \mathbf{c}) = \langle \mathbf{a}, \mathbf{b} \times \mathbf{c} \rangle$ gilt

$$\begin{pmatrix} t \\ r \\ s \end{pmatrix} = \frac{1}{\langle \mathbf{d} \times \mathbf{v}, \mathbf{u} \rangle} \begin{pmatrix} \langle \mathbf{w} \times \mathbf{u}, \mathbf{v} \rangle \\ \langle \mathbf{d} \times \mathbf{v}, \mathbf{w} \rangle \\ \langle \mathbf{w} \times \mathbf{u}, \mathbf{d} \rangle \end{pmatrix}$$

Der Punkt \mathbf{e} liegt im Dreieck, wenn $r, s \in [0, 1]$ und $r + s \leq 1$

Eine mögliche Dreiecks-Klasse

```
class Triangle(object):
    def __init__(self, a, b, c):
        self.a = a # point
        self.b = b # point
        self.c = c # point
        self.u = self.b - self.a # direction vector
        self.v = self.c - self.a # direction vector

    def __repr__(self):
        return 'Triangle(%s,%s,%s)' % (repr(self.a), repr(self.b), repr(self.c))

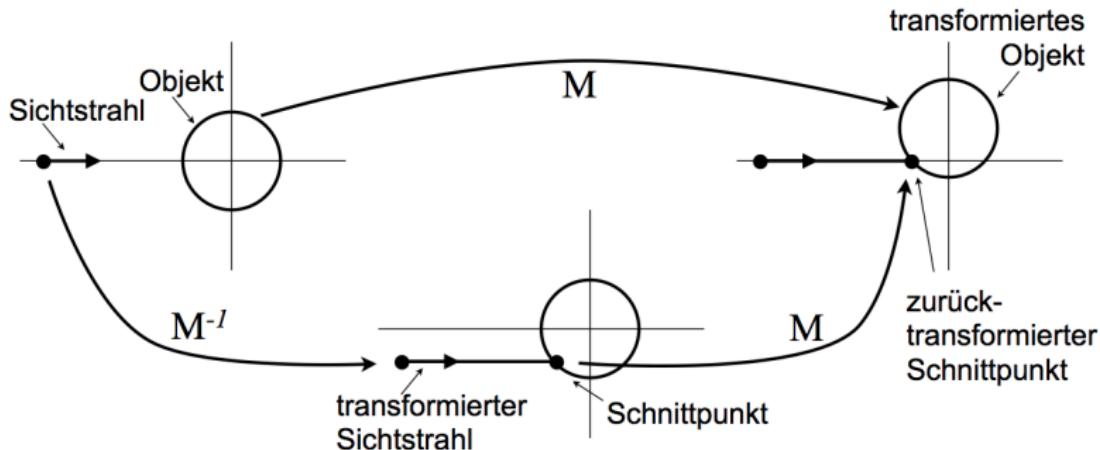
    def intersectionParameter(self, ray):
        w = ray.origin - self.a
        dv = ray.direction.cross(self.v)
        dvu = dv.dot(self.u)
        if dvu == 0.0:
            return None
        wu = w.cross(self.u)
        r = dv.dot(w) / dvu
        s = wu.dot(ray.direction) / dvu
        if 0 <= r and r <= 1 and 0 <= s and s <= 1 and r+s <= 1:
            return wu.dot(self.v) / dvu
        else:
            return None

    def normalAt(self, p):
        return self.u.cross(self.v).normalized()
```

Transformation von Objekten

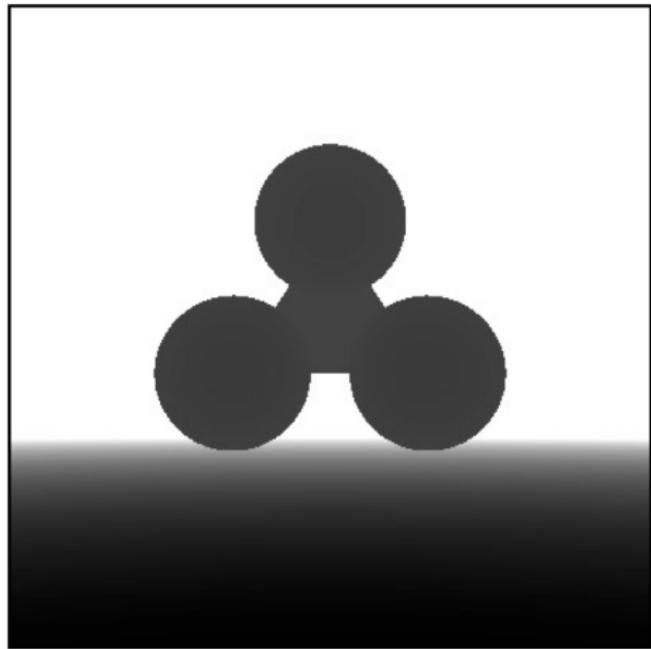
Um ein Objekt mit \mathbf{M} zu transformieren

- transformiert man den Sichtstrahl mit \mathbf{M}^{-1} ,
- führt den bekannten Strahl-Objekt Schnitttest durch und
- transformiert den Schnittpunkt anschließend mit \mathbf{M} zurück in das Weltkoordinatensystem



Beispiel (Tiefenbild)

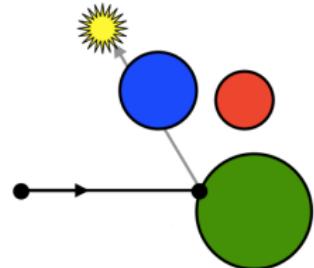
- Bildgröße
 - 400×400 Pixel
- Kamera
 - $\mathbf{e} = (0, 1.8, 10)^T$
 - $\mathbf{c} = (0, 3, 0)^T$
 - $\mathbf{up} = (0, 1, 0)^T$
 - 45° field of view
- Geometrie
 - Drei Kugeln
 - Eine Ebene
 - Ein Dreick



Beleuchtungsberechnung

Schatten lassen sich berechnen, indem

- für jeden Strahl-Objekt-Schnittpunkt ein Strahl in Richtung der Lichtquelle(n) geschickt wird und
- überprüft wird, ob dieser ein anderes Objekt schneidet

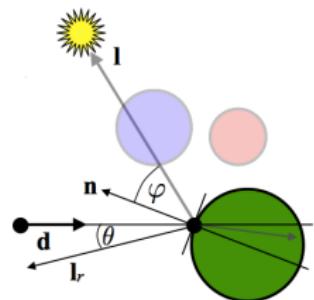


Schnittpunktfarben lassen sich z. B. berechnen mittels

- lokalem Beleuchtungsmodell nach Phong

$$\begin{aligned} \mathbf{C}_{out} &= \underbrace{\mathbf{C}_a \cdot k_a}_{\text{ambienter Anteil}} + \underbrace{\mathbf{C}_{in} \cdot k_d \cdot \cos \varphi}_{\text{diffuser Anteil}} + \underbrace{\mathbf{C}_{in} \cdot k_s \cdot (\cos \theta)^n}_{\text{spekularer Anteil}} \\ &= \mathbf{C}_a \cdot k_a + \mathbf{C}_{in} \cdot k_d \cdot \langle \mathbf{l}, \mathbf{n} \rangle + \mathbf{C}_{in} \cdot k_s \cdot \langle \mathbf{l}_r, -\mathbf{d} \rangle^n \end{aligned}$$

- $\mathbf{l}, \mathbf{n}, \mathbf{l}_r, \mathbf{d}$ normiert
- Lichtfarben(vektoren) $\mathbf{C}_a, \mathbf{C}_{in}$
- Materialkonstanten(vektoren)
- $k_a \leq a, k_d + k_s \leq 1$
- und evtl. Refractionsterm

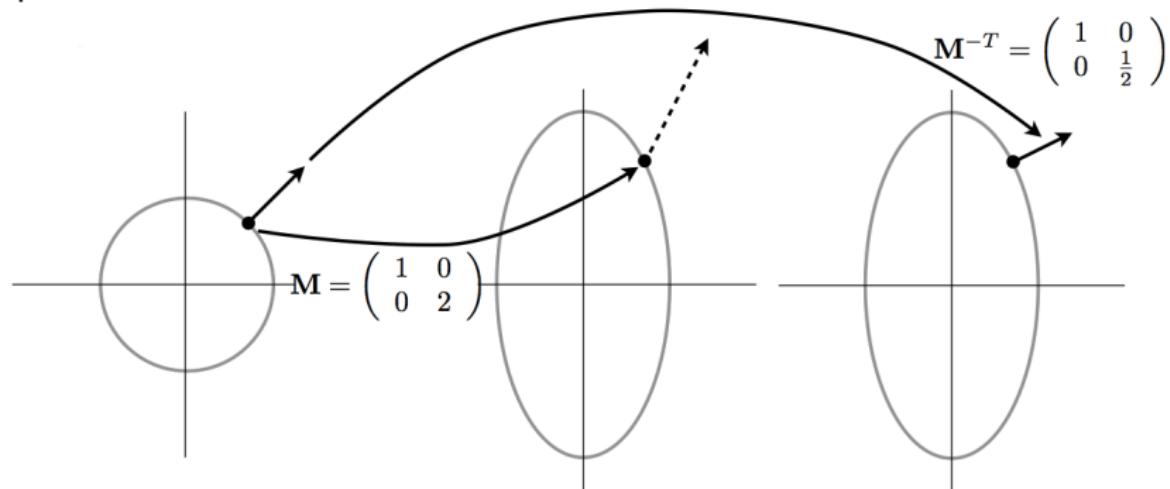


Transformation der Normalen

Werden Objektpunkte mit \mathbf{M} transformiert, dann müssen die Normalen mit \mathbf{M}^{-T} transformiert werden, denn

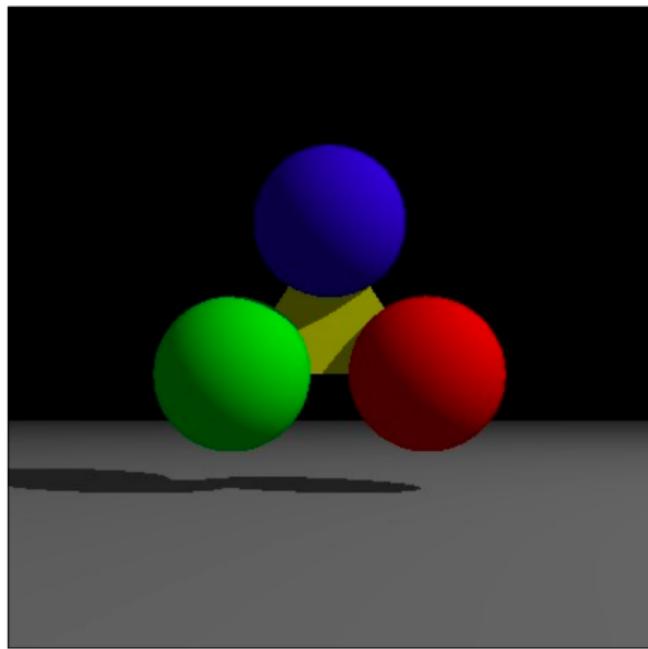
für Punkte \mathbf{p} auf der Ebene \mathbf{e} gilt $\langle \mathbf{e}, \mathbf{p} \rangle = \mathbf{e}^T \mathbf{p} = 0$ und nach der Transformation mit \mathbf{M} muss gelten $\langle \mathbf{Ne}, \mathbf{Mp} \rangle = \mathbf{e}^T \mathbf{N}^T \mathbf{M} \mathbf{p} = 0$, also $\mathbf{N} = \mathbf{M}^{-T}$

Beispiel:



Beispiel (Beleuchtung)

- Bildgröße
 - 400×400 Pixel
- Lichtquellenposition
 - $\mathbf{l} = (30, 30, 10)^T$
- Kamera
 - $\mathbf{e} = (0, 1.8, 10)^T$
 - $\mathbf{c} = (0, 3, 0)^T$
 - $\mathbf{up} = (0, 1, 0)^T$
 - 45° field of view
- Geometrie
 - Drei Kugeln
 - Eine Ebene
 - Ein Dreick



Prinzip des rekursiven Ray Tracing

```
for x in range(imageWidth):
    for y in range(imageHeight):
        ray = calcRay(x, y, cameraParameter)
        color = traceRay(0, ray)
        image.putpixel((x,y), color)
```

```
def traceRay(level, ray):
    hitPointData = intersect(level, ray, maxlevel) # maxLevel = maximale Rekursions-Tiefe
    if hitPointData:
        return shade(level, hitPointData)
    return BACKGROUND_COLOR
```

```
def shade(level, hitPointData):
    directColor = computeDirectLight(hitPointData)

    reflectedRay = computeReflectedRay(hitPointData)
    reflectColor = traceRay(level+1, reflectedRay)

    #refractedRay = computeRefractedRay(hitPointData)
    #refractColor = traceRay(level+1, refractedRay)

    return directColor + reflection*reflectColor # + refraction*refractColor
```



Gerichtet reflektiertes / transmittiertes Licht

Bei *spiegelnden Oberflächen* wird das auftreffende Licht in genau eine Richtung weitergesandt

- Einfallswinkel = Ausfallwinkel, also gilt $\mathbf{d}_r = \mathbf{d} - 2 \cdot \langle \mathbf{n}, \mathbf{d} \rangle \cdot \mathbf{n}$
- Es muss jetzt also das Licht aus Richtung \mathbf{d}_r bestimmt werden

Bei *transparenten Oberflächen* wird das auftreffende Licht ebenfalls in genau eine Richtung weitergeleitet

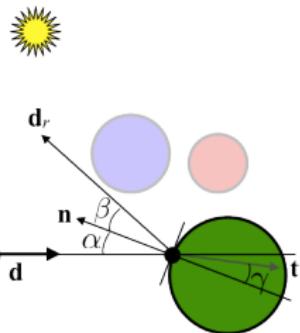
- Winkel der Lichtbrechung ist abhängig von den Brechungsindizes η_1, η_2 der beiden

Materialien (Snell's Gesetz: $\frac{\sin \alpha}{\sin \gamma} = \eta_1/\eta_2$):

$$\mathbf{t} = \frac{\eta_1}{\eta_2} \mathbf{d} + \left(\frac{\eta_1}{\eta_2} \cos \alpha - \sqrt{1 + \left(\frac{\eta_1}{\eta_2} \right)^2 \cdot (\cos \alpha - 1)} \right) \mathbf{n}$$

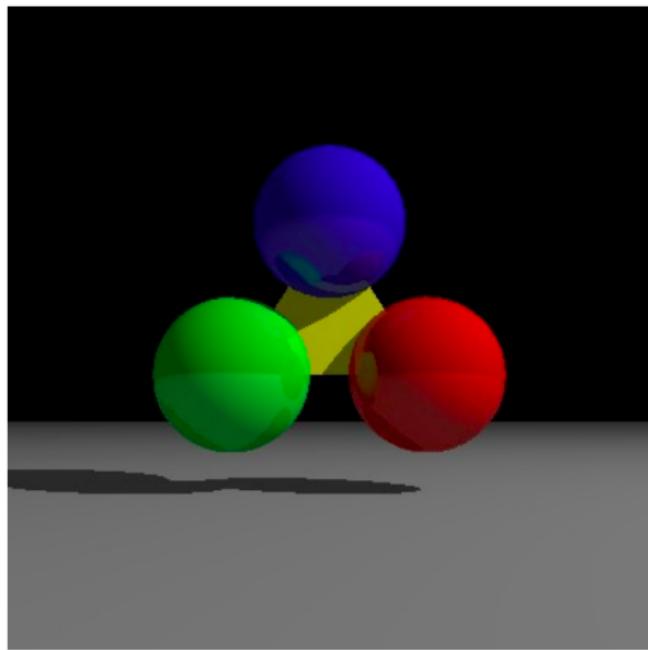
evtl. gibt es eine Totalreflexion

- Es muss jetzt also das Licht aus Richtung \mathbf{t} bestimmt werden



Beispiel (spiegelnde Oberflächen)

- Bildgröße
 - 400×400 Pixel
- Lichtquellenposition
 - $\mathbf{l} = (30, 30, 10)^T$
- Kamera
 - $\mathbf{e} = (0, 1.8, 10)^T$
 - $\mathbf{c} = (0, 3, 0)^T$
 - $\mathbf{up} = (0, 1, 0)^T$
 - 45° field of view
- Geometrie
 - Drei Kugeln
 - Eine Ebene
 - Ein Dreick



Materialien (Texturen)

Objekte können "beliebige" Materialien (Texturen, Bilder) zugewiesen bekommen

Einfaches, klassisches Beispiel: *Schachbrettmuster*

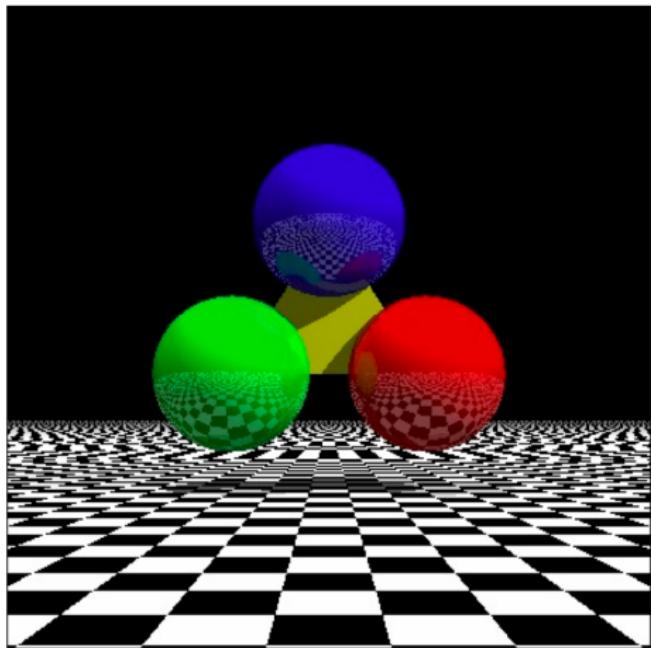
```
class CheckerboardMaterial(object):
    def __init__(self, a, b, c):
        self.baseColor = (1, 1, 1)
        self.otherColor = (0, 0, 0)
        self.ambientCoefficient = 1.0
        self.diffuseCoefficient = 0.6
        self.specularCoefficient = 0.2
        self.checkSize = 1

    def baseColorAt(self, p):
        v = Vector(p)
        v.scale(1.0 / self.checkSize)
        if (int(abs(v.x) + 0.5) + int(abs(v.y) + 0.5) + int(abs(v.z) + 0.5)) % 2:
            return self.otherColor
        return self.baseColor
```



Beispiel (texturierte Oberflächen)

- Bildgröße
 - 400×400 Pixel
- Lichtquellenpositionen
 - $\mathbf{l}_1 = (30, 30, 10)^T$
 - $\mathbf{l}_2 = (-10, 100, 30)^T$
- Kamera
 - $\mathbf{e} = (0, 1.8, 10)^T$
 - $\mathbf{c} = (0, 3, 0)^T$
 - $\mathbf{up} = (0, 1, 0)^T$
 - 45° field of view
- Geometrie
 - Drei Kugeln
 - Eine Ebene
 - Ein Dreick



Raytracing Laufzeitanalyse

Der aufwändigste Teil beim Raytracing ist die Berechnung der Schnittpunkte zwischen Strahlen und Objekten

- Bei naiver Implementierung muss *für jeden Strahl* der Schnittpunkt *mit jedem der n Objekte* einer Szene berechnet werden
- Bei einem Primärstrahl pro Pixel und einer maximalen Strahlverfolgungstiefe von d ergibt sich eine *Gesamtaufzeit* von

$$O(\text{width} \cdot \text{height} \cdot 2^d \cdot n)$$

- Für ein Bild mit einer *Auflösung* von 640×480 Pixeln, einer *Szene* mit $n = 100$ Objekten und einer *Strahlverfolgungstiefe* von $d = 3$ ergibt das schon 245.760.000 Schnittpunkttests



Beschleunigung von Raytracing

Beschleunigung von Raytracing zum Beispiel durch

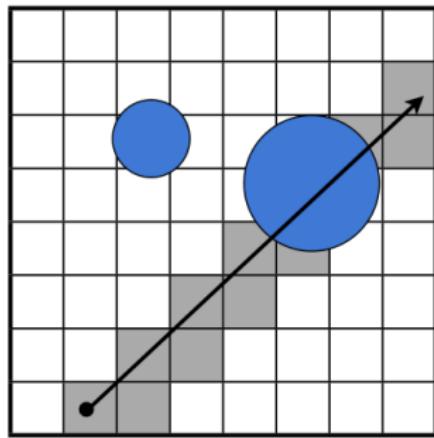
- Effiziente Algorithmen zur Schnittstellenberechnung
- "Cache-Optimierungen" (Vermeiden von Hauptspeicherzugriffen)
- Parallelisierung (auf Clustern, auf CPU-Kernen, auf GPU-Einheiten)
- Effiziente (räumliche) Datenstrukturen zur Einschränkung der Liste, der pro Strahl auf Schnittpunkt zu testenden Objekte
 - Gleichmässiges Gitter
 - Octrees
 - k-d-Bäume
 - Binary Space Partitioning (BSP)
 - Hüllkörperhierarchien
 - ...



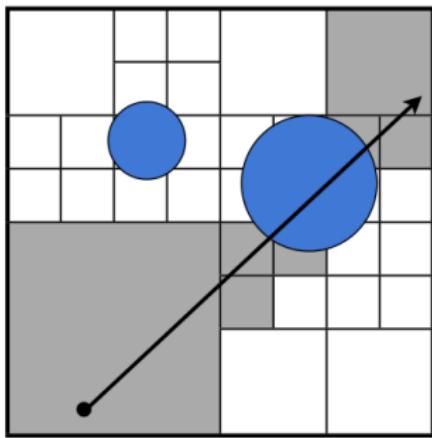
Gleichmässiges Gitter / Octree

Unterteile den Raum in Voxel (Volumen Elemente) und speichere zu jedem Bereich die dort vorhandenen Objekte

Bei der Schnittpunktberechnung nur Objekte testen, die sich im Bereich um den Strahl herum befinden und Voxelgitter z.B. mittels 3D Bresenham durchlaufen



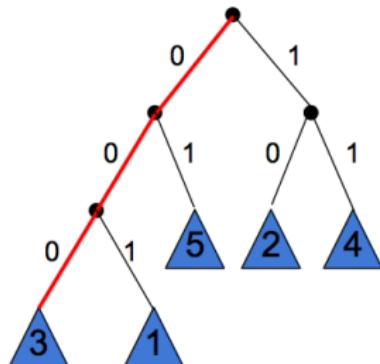
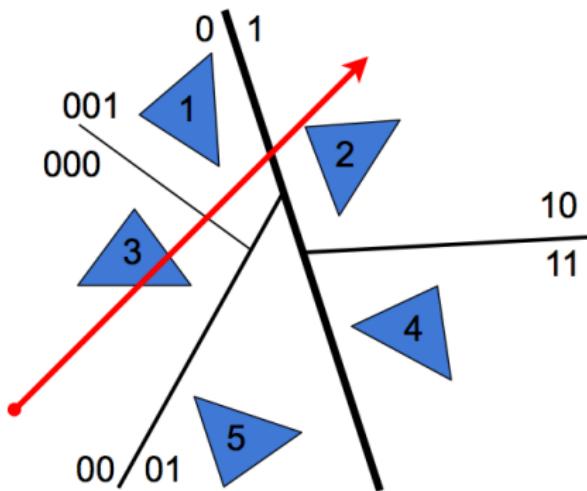
Gleichmässiges Gitter



Octree (bzw. Quadtree)

Binary Space-Partitioning

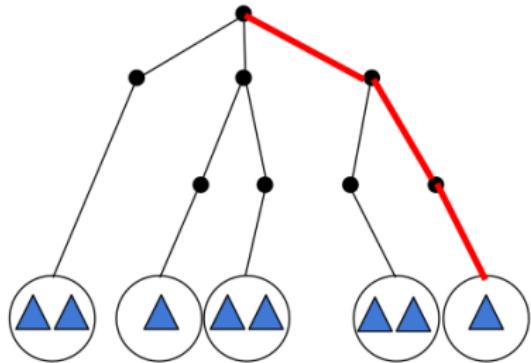
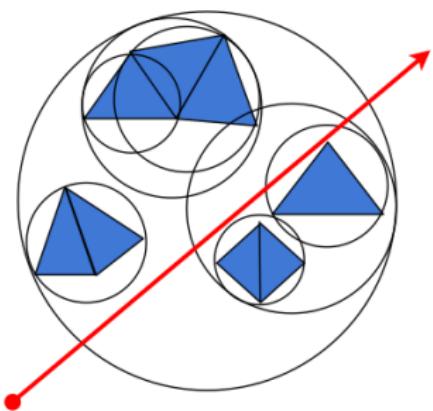
Teile den Raum in Halbräume



- Steige bei der Schnittpunktberechnung zunächst in den Unterbaum ab, der näher am Blickpunkt liegt
- Mit Erweiterungen nur noch $O(\log n)$ Schnittpunkttests

Hüllkörperhierarchien

Die Objekte einer Szene werden in einfachen Hüll-Objekten hierarchisch verschachtelt



- Hüll-Objekte (z.B. Kugeln) können effizient auf Schnitt getestet werden
- Bei der Schnittpunktberechnung wird nur dann in einen Unterbaum abgestiegen, wenn Schnitt mit Wurzelobjekt vorliegt

Diffus reflektiertes / transmittiertes Licht

- Bei der *diffusen Reflexion* wird einfallendes Licht in unterschiedliche Richtungen gestreut
- Bei der *diffusen Transmission* wird einfallendes Licht im Objekt (durch kleine Partikel) ständig umgelenkt (Transluzenz)
- Problem
 - In beiden Fällen gibt es keine eindeutige Richtung, in die Strahlen weiter verfolgt werden können
- Lösungsmöglichkeiten
 - Stochastisches Raytracing
 - Path Tracing
 - Photon Mapping
 - ...



Berücksichtigung weitere physikalischer Effekte

Einfaches Lochkameramodell

Alles in einer Szene wird scharf abgebildet

Reale Kameras haben Linsen / Aperturen und Belichtungszeit

Dadurch endliche Schärfentiefe, Bewegungsunschärfe, ...

Beispiel endliche Schärfentiefe



Quelle: Povray Hall of Fame

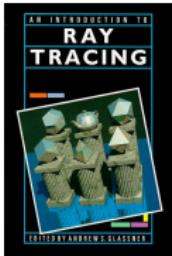
Beispiel Bewegungsunschärfe



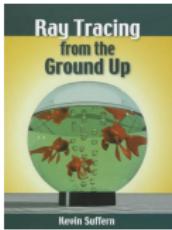
AUTORAYTRACING 2018 - Raytracing mit Rayshade 3.3 - Seite 11

Weitere Informationen

- Andrew S. Glassner; *An Introduction to Ray Tracing*, Academic Pr. Inc., 1989



- Kevin Suffner; *Ray Tracing from the Ground Up*, Taylor & Francis Ltd., 2007



- POV-Ray (freier Raytracer): www.povray.org



Die verschiedenen Geometrien und ihre Transformationen

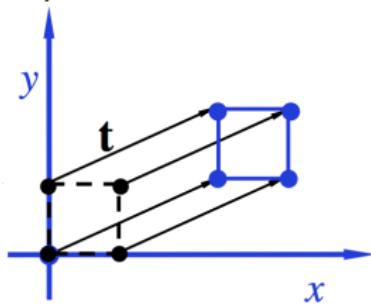
	Geometrien			
	Euklidisch	Ähnlichkeit	Affin	Projektiv
<i>Transformationen</i>				
Translation	X	X	X	X
Rotation	X	X	X	X
Gleichmässige Skalierung		X	X	X
Ungleichmässige Skalierung			X	X
Scherung			X	X
Perspektivische Projektion				X
Verknüpfung von Projektionen				X
<i>Invarianten</i>				
Länge	X			
Winkel	X	X		
(Teil-)Verhältnis von Längen	X	X	X	
Parallelität	X	X	X	
Inzidenz	X	X	X	X
Doppelverhältnis (cross ratio)	X	X	X	X



Euklidische (starre) Transformationen (Translation)

Translation (Verschiebung) eines Objekts

- Jeder Punkt \mathbf{p} eines Objekts wird um den Vektor \mathbf{t} verschoben und es entsteht ein neues Objekt mit den Punkten $\mathbf{p}' = \mathbf{p} + \mathbf{t}$
- Der Nullpunkt wird verschoben



$$\begin{pmatrix} p'_x \\ p'_y \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

- Inverse Operation: $\mathbf{p}' = \mathbf{p} - \mathbf{t}$



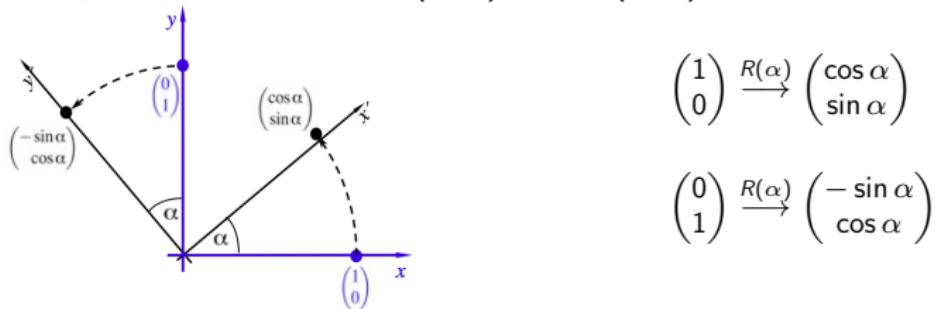
Euklidische (starre) Transformationen (Rotation in \mathbb{R}^2)

Rotation eines Objekts um den Winkel α

Jeder Punkt \mathbf{p} eines Objekts wird mit $R(\alpha)$ transformiert und es entsteht ein neues Objekt mit den Punkten $\mathbf{p}' = R(\alpha)\mathbf{p}$

Rotation in \mathbb{R}^2 um den Ursprung gegen den Uhrzeigersinn

- Abbildung der Basisvektoren $(1, 0)^T$ und $(0, 1)^T$



- Für die *Abbildungsmatrix $R(\alpha)$ gilt also*

$$R(\alpha) = \begin{pmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix} \quad \text{und} \quad R(\alpha)^{-1} = \begin{pmatrix} \cos \alpha & \sin \alpha \\ -\sin \alpha & \cos \alpha \end{pmatrix}$$

Homogene Darstellung von Transformationen

Allgemeine Darstellung einer starren Transformation des Punktes \mathbf{p} lautet

$$\mathbf{p}' = \mathbf{R}\mathbf{p} + \mathbf{t} \text{ mit der Rotation } \mathbf{R} \text{ und der Verschiebung } \mathbf{t}$$

Sollen z.B. zwei Euklidische Transformationen $(\mathbf{R}_1, \mathbf{t}_1), (\mathbf{R}_2, \mathbf{t}_2)$ hintereinander ausgeführt werden, führt das zu dem komplizierte Ausdruck

$$\mathbf{p}'' = \mathbf{R}_2\mathbf{p}' + \mathbf{t}_2 = \mathbf{R}_2(\mathbf{R}_1\mathbf{p} + \mathbf{t}_1) + \mathbf{t}_2 = \mathbf{R}_2\mathbf{R}_1\mathbf{p} + \mathbf{R}_2\mathbf{t}_1 + \mathbf{t}_2$$

Zur einheitlichen Darstellung von (Euklidischen) Transformationen mittels Matrizen verwendet man *homogene Vektoren* (siehe auch Vorlesung (I))

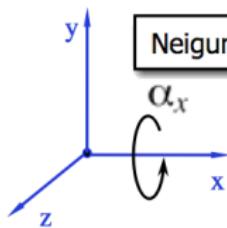
- $\tilde{\mathbf{p}} = k(x, y, 1)^T, k \in \mathbb{R} \setminus \{0\}$ repräsentieren den Punkt $\mathbf{p} = (x, y)$
- $\tilde{\mathbf{p}} = k(x, y, 0)^T, k \in \mathbb{R} \setminus \{0\}$ repräsentieren den Vektor $\mathbf{p} = (x, y)$
- Homogene Beschreibung von (Euklidischen) Transformationen:

$$\tilde{\mathbf{p}}' = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \tilde{\mathbf{p}} = \begin{pmatrix} r_{00} & r_{01} & t_x \\ r_{10} & r_{11} & t_y \\ 0 & 0 & 1 \end{pmatrix} \tilde{\mathbf{p}} = \begin{pmatrix} \cos \alpha & -\sin \alpha & t_x \\ \sin \alpha & \cos \alpha & t_y \\ 0 & 0 & 1 \end{pmatrix} \tilde{\mathbf{p}}$$

Rotation in \mathbb{R}^3 mittels Euler-Winkeln

Im *Rechtssystem* positive Drehwinkel (gegen den Uhrzeigersinn)

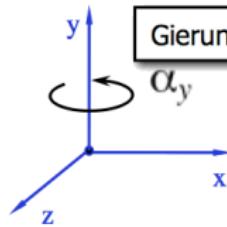
Rotation um
die x-Achse
(*Pitch*)



Neigungswinkel

$$\tilde{\mathbf{R}}_x(\alpha_x) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha_x & -\sin \alpha_x & 0 \\ 0 & \sin \alpha_x & \cos \alpha_x & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

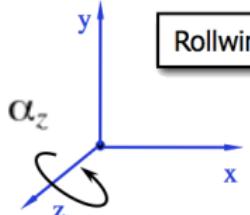
Rotation um
die y-Achse
(*Yaw*)



Gierungswinkel

$$\tilde{\mathbf{R}}_y(\alpha_y) = \begin{pmatrix} \cos \alpha_y & 0 & \sin \alpha_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \alpha_y & 0 & \cos \alpha_y & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation um
die z-Achse
(*Roll*)



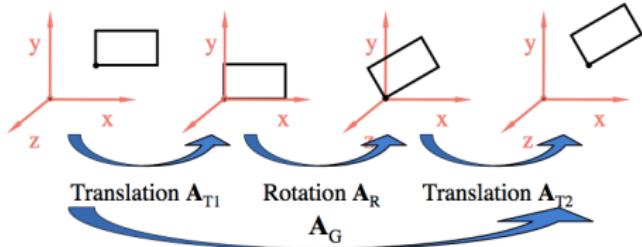
Rollwinkel

$$\tilde{\mathbf{R}}_z(\alpha_z) = \begin{pmatrix} \cos \alpha_z & -\sin \alpha_z & 0 & 0 \\ \sin \alpha_z & \cos \alpha_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Beispiel: Rotation um einen beliebigen Punkt \mathbf{p}

Vorgehensweise bei der Rotation eines Objekts um eine *Parallele zur z-Achse* an einem seiner eigenen Punkte $(p_x, p_y, 0)^T$

- 1 $\tilde{\mathbf{A}}_{T1}$: Objekt in den Ursprung verschieben
- 2 $\tilde{\mathbf{A}}_R$: Rotation durchführen
- 3 $\tilde{\mathbf{A}}_{T2}$: Objekt zurückverschieben



Die Matrix $\tilde{\mathbf{A}}_G$ der *Gesamt-Transformation* lautet damit

$$\begin{aligned}\tilde{\mathbf{A}}_G &= \tilde{\mathbf{A}}_{T2} \cdot \tilde{\mathbf{A}}_R \cdot \tilde{\mathbf{A}}_{T1} = \begin{pmatrix} 1 & 0 & 0 & p_x \\ 0 & 1 & 0 & p_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 & 0 \\ \sin \alpha & \cos \alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 & -p_x \cos \alpha + p_y \sin \alpha + p_x \\ \sin \alpha & \cos \alpha & 0 & -p_x \cos \alpha - p_y \sin \alpha + p_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\end{aligned}$$

Eulerwinkel und Eulertransformationen

Die Winkel $\alpha_x, \alpha_y, \alpha_z$ werden auch *Eulerwinkel* genannt

Das Ergebnis der Multiplikation dreier Matrizen für die Drehungen um die (kartesische) $x-, y-$ und z -Achse nennt man *Eulertransformation*

Beispiel

Drehe nacheinander um die $x-, y-$ und z -Achse

$$\begin{aligned}\tilde{\mathbf{R}}(\alpha, \beta, \gamma) &= \tilde{\mathbf{R}}_z(\gamma) \cdot \tilde{\mathbf{R}}_y(\beta) \cdot \tilde{\mathbf{R}}_x(\alpha) \\ &= \begin{pmatrix} \cos \beta \cos \gamma & \sin \alpha \sin \beta \cos \gamma - \cos \alpha \sin \gamma & \cos \alpha \sin \beta \cos \gamma + \sin \alpha \sin \gamma & 0 \\ \cos \beta \sin \gamma & \sin \alpha \sin \beta \sin \gamma + \cos \alpha \cos \gamma & \cos \alpha \sin \beta \sin \gamma - \sin \alpha \cos \gamma & 0 \\ -\sin \beta & \sin \alpha \cos \beta & \cos \alpha \cos \beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\end{aligned}$$

Verbreitetste Darstellung von Rotationen

Vorteil: Anschaulichkeit

Nachteile der Eulertransformation

Nachteile der Beschreibung von Rotationen mittels Eulerwinkel

- Reihenfolge der Drehungen ist relevant

$$\mathbf{R}_x(\alpha) \cdot \mathbf{R}_y(\beta) \cdot \mathbf{R}_z(\gamma) \neq \mathbf{R}_z(\gamma) \cdot \mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha)$$

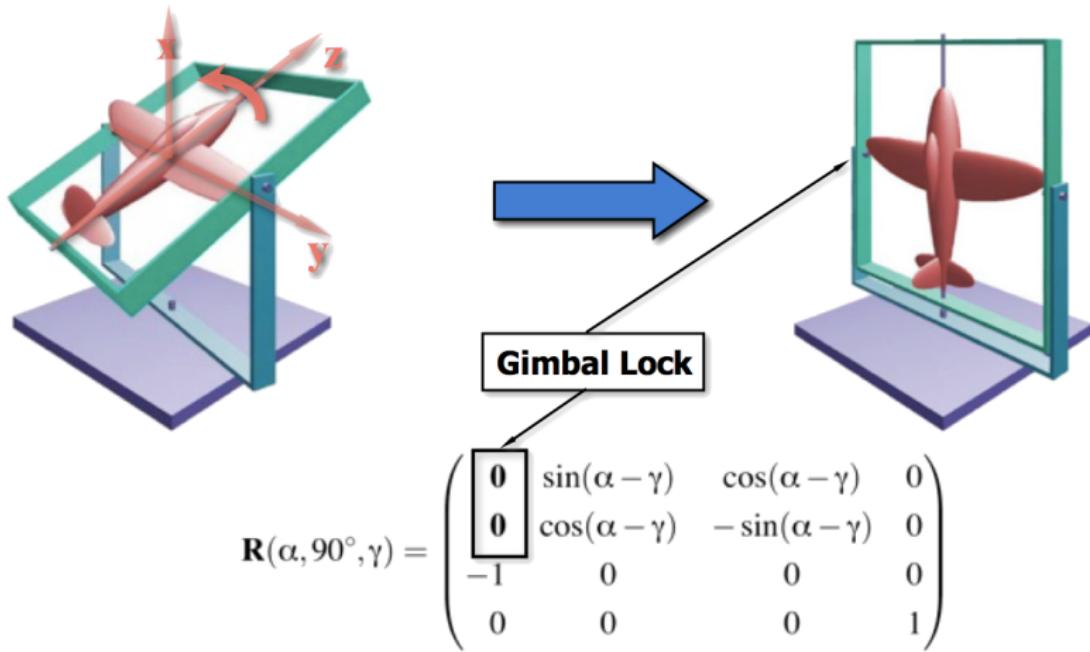
- Nicht eindeutig: $\mathbf{R}(\alpha, \beta, \gamma) = \mathbf{R}(\alpha \pm 180^\circ, \beta \pm 180^\circ, \gamma \pm 180^\circ)$
Beispiel

$$\tilde{\mathbf{R}}(0, 180^\circ, 180^\circ) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \tilde{\mathbf{R}}(180^\circ, 0, 0)$$

- Für Animation (Interpolation von Bewegungen) *ungeeignet*
 - Inbetweening durch Spherical Linear Interpolation (SLERP)
- *Gimbal Lock* (Gimbal = Kardanring (Kompass, Gyroscope))
 - Rotation einer Achse um 90° überlagert eine andere Achse.
 - Es geht ein *Freiheitsgrad verloren*

Gimbal Lock Beispiel

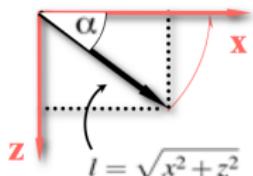
Rotation um die y -Achse um 90°



Rotation um beliebige Achse (I)

Drehung gegen den Uhrzeigersinn, um den (normierten) Vektor $\mathbf{n} = (x, y, z)$ ($\|\mathbf{n}\| = 1$) und den Winkel φ durch Zerlegung in "Standard"-Rotationen:

1. Rotation von \mathbf{n} um die y -Achse auf die xy -Ebene

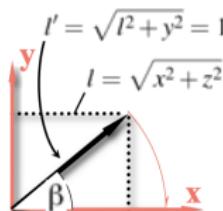


$$\sin \alpha = \frac{z}{l} \quad \Rightarrow \quad \tilde{\mathbf{R}}_1 = \begin{pmatrix} \frac{x}{l} & 0 & \frac{z}{l} & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{z}{l} & 0 & \frac{x}{l} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\cos \alpha = \frac{x}{l}$$

Ergebnis: Vektor \mathbf{n}' in der xy -Ebene

2. Rotation von \mathbf{n}' um die z -Achse auf die x -Achse

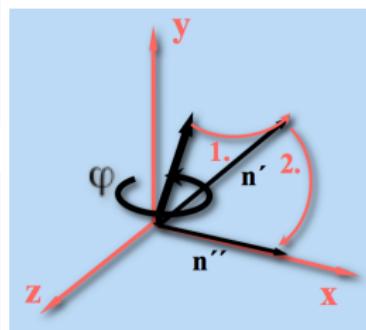


$$l' = \sqrt{l^2 + y^2} = 1 \quad \sin \beta = y$$

$$\cos \beta = l \quad \Rightarrow \quad \tilde{\mathbf{R}}_2 = \begin{pmatrix} l & y & 0 & 0 \\ -y & l & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation
gegen den
Uhrzeigersinn

Ergebnis: Vektor \mathbf{n}'' parallel zur x -Achse



Rotation um beliebige Achse (II)

3. Rotation um den Winkel φ um die x -Achse

$$\tilde{\mathbf{R}}_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \varphi & -\sin \varphi & 0 \\ 0 & \sin \varphi & \cos \varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

4. Inverse Rotation zur Rotation \mathbf{R}_2

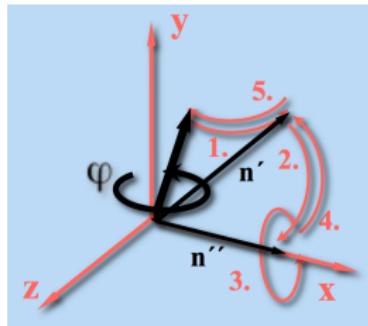
- Rotation um den Winkel β um die z -Achse
- Rotation \mathbf{R}_2 war gegen den Uhrzeigersinn!

$$\tilde{\mathbf{R}}_2^{-1} = \begin{pmatrix} l & -y & 0 & 0 \\ y & l & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

5. Inverse Rotation zur Rotation \mathbf{R}_1

- Rotation um den Winkel $-\alpha$ um die y -Achse
- $\sin(-\alpha) = -\frac{z}{l}, \cos(-\alpha) = \frac{x}{l}$

$$\tilde{\mathbf{R}}_1^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{x}{l} & \frac{z}{l} & 0 \\ 0 & -\frac{z}{l} & \frac{x}{l} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Rotation um beliebige Achse (III)

Gesamt-Matrix der Rotation um den normierten Vektor $\mathbf{n} = (x, y, z)^T$ und den Winkel φ :

$$\begin{aligned}\tilde{\mathbf{R}}(\mathbf{n}, \varphi) &= \tilde{\mathbf{R}}_1^{-1} \cdot \tilde{\mathbf{R}}_2^{-1} \cdot \tilde{\mathbf{R}}_3 \cdot \tilde{\mathbf{R}}_2 \cdot \tilde{\mathbf{R}}_1 \\ &= \begin{pmatrix} x^2(1 - \cos \varphi) + \cos \varphi & xy(1 - \cos \varphi) - z \sin \varphi & xz(1 - \cos \varphi) + y \sin \varphi & 0 \\ xy(1 - \cos \varphi) + z \sin \varphi & y^2(1 - \cos \varphi) + \cos \varphi & yz(1 - \cos \varphi) - x \sin \varphi & 0 \\ xz(1 - \cos \varphi) - y \sin \varphi & yz(1 - \cos \varphi) + x \sin \varphi & z^2(1 - \cos \varphi) + \cos \varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\end{aligned}$$

Da \mathbf{R} eine Orthonormal-Matrix ist, gilt weiterhin $\mathbf{R}^{-1}(\mathbf{n}, \varphi) = \mathbf{R}^T(\mathbf{n}, \varphi)$

- "Standard"-Darstellung von Rotationen in der Computergrafik
 - Wird auch in *OpenGL* verwendet (siehe z.B. `man glRotate`)
- Kann in Euler-Winkel ungerechnet werden
- Kein Gimbal Lock
- Für Animationen (Interpolation von Rotationen) allerdings *ungeeignet*

Rotation um beliebige Achse (Beispiel)

Rotation um den Vektor $\mathbf{v} = (1, 1, 1)^T$ und den Winkel $\varphi = 90^\circ$

1 Normieren von Vektor \mathbf{v} liefert

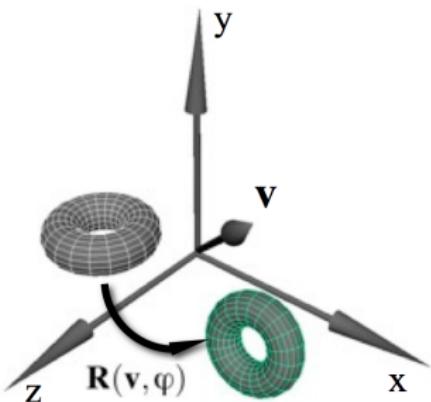
$$\mathbf{n} = \left(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}} \right)$$

2 Berechnen von sin und cos liefert

$$\sin 90^\circ = 1, \quad \cos 90^\circ = 0$$

3 Aufstellen der Rotationsmatrix liefert

$$\tilde{\mathbf{R}}(\mathbf{n}, \varphi) = \begin{pmatrix} \frac{1}{3} & \frac{1-\sqrt{3}}{3} & \frac{1+\sqrt{3}}{3} & 0 \\ \frac{1+\sqrt{3}}{3} & \frac{1}{3} & \frac{1-\sqrt{3}}{3} & 0 \\ \frac{1-\sqrt{3}}{3} & \frac{1+\sqrt{3}}{3} & \frac{1}{3} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Bestimmung der Achse aus einer Rotationsmatrix \mathbf{R}

Die Rotationsachse $\mathbf{v} = (v_1, v_2, v_3)^\top$ ist der Eigenvektor der Rotationsmatrix \mathbf{R} zum Eigenwert 1.

Mit $(\mathbf{R} - \mathbf{I}) \cdot \mathbf{v} = \mathbf{0}$ gilt also

$$\begin{aligned}(r_{11} - 1) \cdot v_1 + r_{12} \cdot v_2 + r_{13} \cdot v_3 &= 0 \\ r_{21} \cdot v_1 + (r_{22} - 1) \cdot v_2 + r_{23} \cdot v_3 &= 0 \\ r_{31} \cdot v_1 + r_{32} \cdot v_2 + (r_{33} - 1) \cdot v_3 &= 0\end{aligned}$$

und somit

$$\begin{aligned}v_1 &= r_{12} \cdot r_{23} - (r_{22} - 1) \cdot r_{13} \\ v_2 &= r_{21} \cdot r_{13} - (r_{11} - 1) \cdot r_{23} \\ v_3 &= (r_{11} - 1) \cdot (r_{22} - 1) - r_{12} \cdot r_{21}\end{aligned}$$



Bestimmung des Winkels aus einer Rotationsmatrix \mathbf{R}

- \mathbf{R} lässt sich zerlegen (siehe Folie 71) in $\mathbf{R} = \mathbf{R}_1^{-1} \cdot \mathbf{R}_2^{-1} \cdot \mathbf{R}_3 \cdot \mathbf{R}_2 \cdot \mathbf{R}_1$, wobei \mathbf{R}_3 die Rotation um die x-Achse und den Winkel φ beschreibt.
- Für die Spur einer Matrix $Tr(\mathbf{A}) = a_{11} + a_{22} + \dots + a_{nn}$ gilt
 $Tr(\mathbf{A} \cdot \mathbf{B}) = Tr(\mathbf{B} \cdot \mathbf{A})$
- Insgesamt erhält man
 $Tr(\mathbf{R}) = Tr(\mathbf{R}_1^{-1} \cdot \mathbf{R}_2^{-1} \cdot \mathbf{R}_3 \cdot \mathbf{R}_2 \cdot \mathbf{R}_1) = Tr(\mathbf{R}_3) = 1 + 2 \cos \varphi$ und damit

Für den Rotationswinkel φ zu einer Rotationsmatrix \mathbf{R} gilt

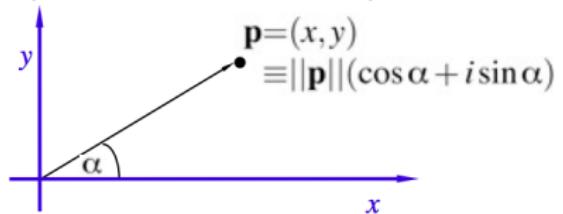
$$\varphi = \arccos\left(\frac{Tr(\mathbf{R}) - 1}{2}\right)$$



Rotation in \mathbb{R}^2 mittels komplexer Zahlen

Ein Punkt $\mathbf{p} = (x, y)^T$ in der Ebene kann (in Polarkoordinaten) repräsentiert werden durch die *komplexe Zahl*

$$\mathbf{p} = \|\mathbf{p}\|(\cos \alpha + i \sin \alpha) = \|\mathbf{p}\|e^{i\alpha}$$

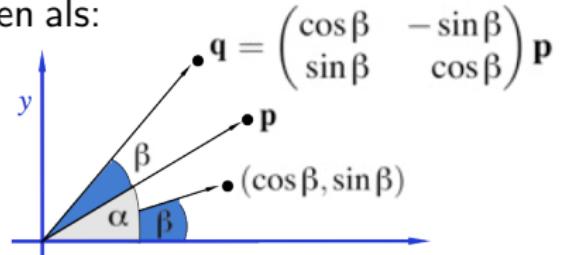


Multiplikation von $\|\mathbf{p}\|e^{i\alpha}$ mit $e^{i\beta} = \cos \beta + i \sin \beta$ liefert

$$\mathbf{q} = \|\mathbf{p}\|e^{i\alpha} \cdot e^{i\beta} = \|\mathbf{p}\|e^{i(\alpha+\beta)} = \|\mathbf{p}\|(\cos(\alpha + \beta) + i \sin(\alpha + \beta))$$

also \mathbf{p} um den Winkel β gegen den Uhrzeigersinn rotiert. Mit $x = \|\mathbf{p}\| \cos \alpha$, $y = \|\mathbf{p}\| \sin \alpha$ kann dies dargestellt werden als:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \beta & -\sin \beta \\ \sin \beta & \cos \beta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$



Quaternionen

Eine Quaternion \mathbf{q} hat eine reelle und drei imaginäre Komponenten

Mit den reellen Zahlen s, x, y, z und den imaginären Einheiten i, j, k gilt

$$\mathbf{q} := \underbrace{s}_{\text{Realteil}} \cdot 1 + \underbrace{x}_{\text{Imaginärteil}} \cdot i + \underbrace{y}_{\text{Imaginärteil}} \cdot j + \underbrace{z}_{\text{Imaginärteil}} \cdot k \iff \mathbf{q} := \begin{bmatrix} s \\ \begin{pmatrix} x \\ y \\ z \end{pmatrix} \end{bmatrix} = [s, \mathbf{v}]$$

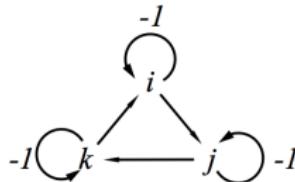
mit

$$i^2 = j^2 = k^2 = i \cdot j \cdot k = -1$$

$$i \cdot j = -j \cdot i = k$$

$$j \cdot k = -k \cdot j = i$$

$$k \cdot i = -i \cdot k = j$$



W. R. Hamilton
1805-1865

- Erstmals 1853 von William Rowan Hamilton beschrieben
- Vierdimensionales Analogon zu den komplexen Zahlen
 - Multiplikation mit komplexer Zahl der Länge 1, $z = e^{i\varphi}$ beschreibt eine Drehung um den Winkel φ

Eigenschaften von Quaternionen

Addition: $\mathbf{q}_1 + \mathbf{q}_2 = [s_1 + s_2, \mathbf{v}_1 + \mathbf{v}_2]$

assoziativ und kommutativ.

Neutrales Element: $\mathbf{0} = [0, (0, 0, 0)]$

Multiplikation: $\mathbf{q}_1 \cdot \mathbf{q}_2 = [s_1 \cdot s_2 - \langle \mathbf{v}_1, \mathbf{v}_2 \rangle, s_1 \mathbf{v}_2 + s_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2]$

nicht kommutativ, d.h. $\mathbf{q}_1 \cdot \mathbf{q}_2 \neq \mathbf{q}_2 \cdot \mathbf{q}_1$. Neutrales Element: $\mathbf{1} = [1, (0, 0, 0)]$

Distributivgesetze gelten: $\mathbf{q}(\mathbf{r} + \mathbf{s}) = \mathbf{qr} + \mathbf{qs}$ und $(\mathbf{r} + \mathbf{s})\mathbf{q} = \mathbf{rq} + \mathbf{sq}$

Norm ("Länge"): $\|\mathbf{q}\| = \sqrt{s^2 + x^2 + y^2 + z^2}$

Konjugierte: $\bar{\mathbf{q}} = [s, -\mathbf{v}]$

Analog zu den komplexen Zahlen gilt $\mathbf{q} \cdot \bar{\mathbf{q}} = s^2 + \|\mathbf{v}\|^2 = \|\mathbf{q}\|^2$

Inverse: $\mathbf{q}^{-1} = \|\mathbf{q}\|^{-2} \cdot \bar{\mathbf{q}}$. Für \mathbf{q} mit $\|\mathbf{q}\| = 1$ gilt: $\mathbf{q}^{-1} = \bar{\mathbf{q}}$

Quaternionen-Multiplikation (Beispiel)

Gegeben:

$$\mathbf{q} = [s, (x, y, z)], \mathbf{p}_x = [0, (1, 0, 0)], \mathbf{p}_y = [0, (0, 1, 0)], \mathbf{p}_z = [0, (0, 0, 1)]$$

Berechne:

$$\begin{aligned}\mathbf{R}(\mathbf{p}_x) &= \mathbf{q} \cdot \mathbf{p}_x \cdot \bar{\mathbf{q}} = [s, (x, y, z)] \cdot [0, (1, 0, 0)] \cdot [s, (-x, -y, -z)] \\ &= [-x, (s, z, -y)] \cdot [s, (-x, -y, -z)] \\ &= [0, (s^2 + x^2 - y^2 - z^2, 2(xy + sz), 2(xz - sy))]\end{aligned}$$

$$\begin{aligned}\mathbf{R}(\mathbf{p}_y) &= \mathbf{q} \cdot \mathbf{p}_y \cdot \bar{\mathbf{q}} = [s, (x, y, z)] \cdot [0, (0, 1, 0)] \cdot [s, (-x, -y, -z)] \\ &= [-y, (-z, s, x)] \cdot [s, (-x, -y, -z)] \\ &= [0, (2(xy - sz), s^2 + y^2 - x^2 - z^2, 2(yz + sx))]\end{aligned}$$

$$\begin{aligned}\mathbf{R}(\mathbf{p}_z) &= \mathbf{q} \cdot \mathbf{p}_z \cdot \bar{\mathbf{q}} = [s, (x, y, z)] \cdot [0, (0, 0, 1)] \cdot [s, (-x, -y, -z)] \\ &= [-z, (y, -x, s)] \cdot [s, (-x, -y, -z)] \\ &= [0, (2(xz + ys), 2(yz - xs), s^2 + z^2 - x^2 - y^2)]\end{aligned}$$

Quaternionen-Multiplikation und Rotation

Wählt man s, x, y, z , so dass \mathbf{q} eine Einheitsquaternion ist ($\|\mathbf{q}\| = 1$), setzt

$$s = \cos \frac{\varphi}{2}, \quad x' = x \sin \frac{\varphi}{2}, \quad y' = y \sin \frac{\varphi}{2}, \quad z' = z \sin \frac{\varphi}{2}$$

und beachtet $x^2 + y^2 + z^2 = 1$ sowie

$$\cos 2\varphi = \cos^2 \varphi - \sin^2 \varphi \quad \text{und} \quad \sin 2\varphi = 2 \cos \varphi \sin \varphi$$

dann gilt

$$\tilde{\mathbf{R}} = \begin{pmatrix} x^2(1 - \cos \varphi) + \cos \varphi & xy(1 - \cos \varphi) - z \sin \varphi & xz(1 - \cos \varphi) + y \sin \varphi & 0 \\ xy(1 - \cos \varphi) + z \sin \varphi & y^2(1 - \cos \varphi) + \cos \varphi & yz(1 - \cos \varphi) - x \sin \varphi & 0 \\ xz(1 - \cos \varphi) - y \sin \varphi & yz(1 - \cos \varphi) + x \sin \varphi & z^2(1 - \cos \varphi) + \cos \varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Die Einheitsquaternion $\mathbf{q} = [\cos \frac{\varphi}{2}, \sin \frac{\varphi}{2} \cdot \mathbf{v}]$ beschreibt also eine Drehung um den Winkel φ und die Achse \mathbf{v}

Quaternionen und Rotation

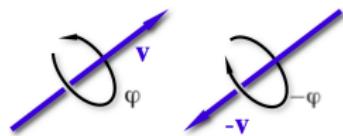
Rotationen um eine Achse \mathbf{v} und einen Winkel φ können mit Einheitsquaternionen $\mathbf{q} = [\cos \frac{\varphi}{2}, \sin \frac{\varphi}{2} \cdot \mathbf{v}]$, $\|\mathbf{q}\| = 1$ beschrieben werden.

Für die Rotation des Punktes \mathbf{p} mit der Quaternion \mathbf{q} gilt

$$\mathbf{R}(\mathbf{p}) = \mathbf{q} \cdot \mathbf{p}_q \cdot \bar{\mathbf{q}} \text{ mit } \mathbf{p}_q = [0, \mathbf{p}]$$

Die Quaternionen \mathbf{q} und $-\mathbf{q}$ beschreiben die gleiche Drehung, denn

$$\begin{aligned} -\mathbf{q} \cdot \mathbf{p}_q \cdot \bar{-\mathbf{q}} &= -1 \cdot \mathbf{q} \cdot \mathbf{p}_q \cdot (\overline{-1 \cdot \mathbf{q}}) \\ &= -1 \cdot \mathbf{q} \cdot \mathbf{p}_q \cdot \overline{-1} \cdot \bar{\mathbf{q}} = \mathbf{q} \cdot \mathbf{p}_q \cdot \bar{\mathbf{q}} \end{aligned}$$



Konkatenation von Rotationen = Multiplikation von Quaternionen

$$\mathbf{R}_2(\mathbf{R}_1(\mathbf{p})) = \mathbf{q}_2 \cdot (\mathbf{q}_1 \cdot \mathbf{p}_q \cdot \bar{\mathbf{q}}_1) \cdot \bar{\mathbf{q}}_2 = (\mathbf{q}_2 \cdot \mathbf{q}_1) \cdot \mathbf{p}_q \cdot \overline{(\mathbf{q}_1 \cdot \mathbf{q}_2)} = \mathbf{R}_{21}(\mathbf{p})$$

Konvertierung Einheitsquaternion in Rotationsmatrix

Für $\mathbf{q} = [s, (x, y, z)]$ gilt

$$\mathbf{q} \begin{bmatrix} 0, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \end{bmatrix} \bar{\mathbf{q}} = \begin{bmatrix} 0, \begin{pmatrix} s^2 + x^2 - y^2 - z^2 \\ 2(xy + sz) \\ 2(xz - sy) \end{pmatrix} \end{bmatrix}$$

$$\mathbf{q} \begin{bmatrix} 0, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \end{bmatrix} \bar{\mathbf{q}} = \begin{bmatrix} 0, \begin{pmatrix} 2(xy - sz) \\ s^2 + y^2 - x^2 - z^2 \\ 2(yz + sx) \end{pmatrix} \end{bmatrix}$$

$$\mathbf{q} \begin{bmatrix} 0, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \end{bmatrix} \bar{\mathbf{q}} = \begin{bmatrix} 0, \begin{pmatrix} 2(xz + sy) \\ 2(yz - sx) \\ s^2 + z^2 - x^2 - y^2 \end{pmatrix} \end{bmatrix}$$

Mit $s^2 + x^2 + y^2 + z^2 = 1$ erhält man dann

$$\tilde{\mathbf{R}} = \begin{pmatrix} 1 - 2(y^2 + z^2) & 2(xy - sz) & 2(xz + sy) & 0 \\ 2(xy + sz) & 1 - 2(x^2 + z^2) & 2(yz - sx) & 0 \\ 2(xz - sy) & 2(yz + sx) & 1 - 2(x^2 + y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Konvertierung Rotationsmatrix in Einheitsquaternion

Für die Einträge der Rotationsmatrix $\mathbf{R} = \begin{pmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{pmatrix}$ gelten

$$\begin{aligned} r_{10} + r_{01} &= 4xy, & r_{02} + r_{20} &= 4xz, & r_{21} + r_{12} &= 4yz \\ r_{10} - r_{01} &= 4sz, & r_{02} - r_{20} &= 4sy, & r_{21} - r_{12} &= 4sx \end{aligned} \quad (1)$$

Mit $t = r_{00} + r_{11} + r_{22} = 4s^2 - 1$ erhält man die Gleichungen

$$\begin{aligned} 2r_{00} - t &= 4x^2 - 1 \implies x &= \pm \frac{1}{2} \sqrt{2r_{00} - t + 1} \\ 2r_{11} - t &= 4y^2 - 1 \implies y &= \pm \frac{1}{2} \sqrt{2r_{11} - t + 1} \\ 2r_{22} - t &= 4z^2 - 1 \implies z &= \pm \frac{1}{2} \sqrt{2r_{22} - t + 1} \end{aligned} \quad (2)$$

- Nachteile $t = 4s^2 - 1 \implies s = \pm \frac{1}{2} \sqrt{t + 1}$

- Wahl der Vorzeichen ist nicht klar
- Es müssen vier Wurzeln berechnet werden

- Besser

- Wähle aus (2) die Gleichungen mit dem größten Ergebnis und benutze (1), um die übrigen Werte zu berechnen

Konvertierung Rotationsmatrix in Einheitsquaternion

```

def rotmat2quat(r):
    """Return the unit quaternion q=[s,x,y,z] to a given 3D rotation matrix r"""
    t = r[0][0] + r[1][1] + r[2][2]
    l = [r[0][0], r[1][1], r[2][2], t]
    i = l.index(max(l))
    if i==0:
        x = sqrt(2*r[0][0]-t+1)/2
        y = (r[1][0]+r[0][1])/(4*x); z = (r[0][2]+r[2][0])/(4*x); s = (r[2][1]-r[1][2])/(4*x)
    elif i==1:
        y = sqrt(2*r[1][1]-t+1)/2
        x = (r[1][0]+r[0][1])/(4*y); z = (r[2][1]+r[1][2])/(4*y); s = (r[0][2]-r[2][0])/(4*y)
    elif i==2:
        z = sqrt(2*r[2][2]-t+1)/2
        x = (r[0][2]+r[2][0])/(4*z); y = (r[2][1]+r[1][2])/(4*z); s = (r[1][0]-r[0][1])/(4*z)
    else: # if i==3:
        s = sqrt(t+1)/2
        x = (r[2][1]-r[1][2])/(4*s); y = (r[0][2]-r[2][0])/(4*s); z = (r[1][0]-r[0][1])/(4*s)
    return [s, x, y, z]

```



Rodrigues Rotationen (Exponential Map)

Die Rotation um einen Vektor $\mathbf{v} = (v_x, v_y, v_z)^T$ und den Winkel $\varphi = \|\mathbf{v}\|$ kann beschrieben werden, durch

$$\exp([\mathbf{v}]_\times) = \mathbf{I} + [\mathbf{v}]_\times + \frac{1}{2!}[\mathbf{v}]_\times^2 + \frac{1}{3!}[\mathbf{v}]_\times^3 + \dots \quad (3)$$

Diese Reihenentwicklung der Exponentialfunktion kann mit der *Formel von Rodrigues*

$$\mathbf{R}([\mathbf{v}]_\times) = \exp([\mathbf{v}]_\times) = \mathbf{I} + \frac{\sin \varphi}{\varphi} [\mathbf{v}]_\times + \frac{1 - \cos \varphi}{\varphi^2} [\mathbf{v}]_\times^2 \quad (4)$$

ausgewertet werden.

- Für den Fall $\|\mathbf{v}\| = 2n\pi$ mit $n \geq 1$ hat (4) eine Singularität und beschreibt keine Rotation
- In der Praxis ersetzt man \mathbf{v} durch $\left(1 - \frac{2\pi}{\|\mathbf{v}\|}\right)\mathbf{v}$, wenn $\|\mathbf{v}\| > \pi$, was die gleiche Rotation mit kleinerem Winkel beschreibt^a

^aEine Rotation um φ um \mathbf{v} ist äquivalent zu einer Rotation um $2\pi - \varphi$ um $-\mathbf{v}$

Rodrigues Rotationen (Beispiel)

Rotation um $\mathbf{v} = \frac{\pi}{2\sqrt{3}}(1, 1, 1)^T$ (d.h. mit $\varphi = \|\mathbf{v}\| = \frac{\pi}{2}$)^a:

^aVergleiche Folie 72

$$\begin{aligned}
 \mathbf{R}([\mathbf{v}]_{\times}) &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + \frac{\sin \frac{\pi}{2}}{\frac{\pi}{2}} \cdot \frac{\pi}{2\sqrt{3}} \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix} + \frac{1 - \cos \frac{\pi}{2}}{\frac{\pi^2}{4}} \cdot \frac{\pi^2}{4 \cdot 3} \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix}^2 \\
 &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + \frac{\sin \frac{\pi}{2}}{\sqrt{3}} \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix} + \frac{1 - \cos \frac{\pi}{2}}{3} \begin{pmatrix} -2 & 1 & 1 \\ 1 & -2 & 1 \\ 1 & 1 & -2 \end{pmatrix} \\
 &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + \frac{1}{\sqrt{3}} \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix} + \frac{1}{3} \begin{pmatrix} -2 & 1 & 1 \\ 1 & -2 & 1 \\ 1 & 1 & -2 \end{pmatrix} \\
 &= \begin{pmatrix} \frac{1}{3} & \frac{1-\sqrt{3}}{3} & \frac{1+\sqrt{3}}{3} \\ \frac{1+\sqrt{3}}{3} & \frac{1}{3} & \frac{1-\sqrt{3}}{3} \\ \frac{1-\sqrt{3}}{3} & \frac{1+\sqrt{3}}{3} & \frac{1}{3} \end{pmatrix}
 \end{aligned}$$

Linearisierung kleiner Rotationen

Aus Gleichung (3) ergibt sich folgende lineare *Approximation einer Rotation um einen kleinen Winkel um die Achse \mathbf{v}*

$$\mathbf{p}' = \mathbf{R}\mathbf{p} = (\mathbf{I} + [\mathbf{v}]_{\times})\mathbf{p} = \mathbf{p} + [\mathbf{v}]_{\times}\mathbf{p} \quad (5)$$

Beispiel: Rotation des Punktes $\mathbf{p} = (1, 1, 0)^T$ um $1^\circ = \frac{\pi}{180}$ um die z -Achse
 Euler-Rotation liefert

$$\begin{pmatrix} \cos \frac{\pi}{180} & -\sin \frac{\pi}{180} & 0 \\ \sin \frac{\pi}{180} & \cos \frac{\pi}{180} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \cos \frac{\pi}{180} & -\sin \frac{\pi}{180} \\ \sin \frac{\pi}{180} & \cos \frac{\pi}{180} \\ 0 & 0 \end{pmatrix} \approx \begin{pmatrix} 0.98239529 \\ 1.01730010 \\ 0 \end{pmatrix}$$

Nach (5) ergibt sich mit $\mathbf{v} = \frac{\pi}{180}(0, 0, 1)^T \approx (0, 0, 0.01745329)^T$

$$\begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 & -0.01745329 & 0 \\ 0.01745329 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.98254671 \\ 1.01745329 \\ 0 \end{pmatrix}$$

Die verschiedene Repräsentationen von Rotationen

Euler-Rotationen

- Repräsentation einer Rotation als Produkt von Matrizen, die Rotationen um die x , y und z Achse beschreiben
- Gimbal Lock Problematik

Quaternione

- Repräsentation einer Rotation durch einen Vektor $\mathbf{q} \in \mathbb{R}^4$ mit der zusätzlichen Bedingung $\|\mathbf{q}\| = 1$
- Vermeidet *Gimbal Lock Problematik* der Euler-Winkel Repräsentation

Rodrigues Rotationen (Exponential Map)

- Repräsentation einer Rotation durch einen Vektor $\mathbf{v} \in \mathbb{R}^3$
- Vermeidet *Gimbal Lock Problematik* der Euler-Winkel Repräsentation
- Benötigt im Gegensatz zu Quaternione keine zusätzliche Bedingung

Invarianten von Euklidischen (starren) Transformationen

Euklidische Transformationen werden auch *starre Transformationen* oder *Isometrien*^a genannt, da Sie ein Objekt "starr" lassen. Sie ändern also nicht (lassen *invariant*)

^a*Iso* = Gleich, *Metrik* = Messung

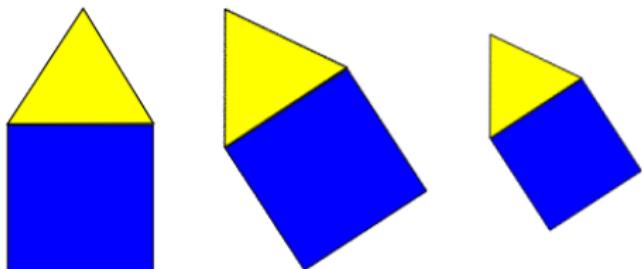
- die Länge (Größe) eines Objekts
- die Winkel eines Objekts
- das (Teil-)Verhältnis $TV(\mathbf{a}, \mathbf{b}, \mathbf{t}) := \frac{l_g(\mathbf{at})}{l_g(\mathbf{tb})}$ dreier Punkte $\mathbf{a}, \mathbf{t}, \mathbf{b}$ auf einer Strecke, wobei $l_g(\mathbf{at})$ die vorzeichenbehaftete Länge von \mathbf{a} nach \mathbf{t} ist.
- die Parallelität von Geraden eines Objekts
- die Inzidenz von Punkten und Geraden eines Objekts
- das Doppelverhältnis $DV(\mathbf{a}, \mathbf{b}, \mathbf{t}, \mathbf{u}) := \frac{TV(\mathbf{a}, \mathbf{b}, \mathbf{t})}{TV(\mathbf{a}, \mathbf{b}, \mathbf{u})}$ von vier Punkten $\mathbf{a}, \mathbf{t}, \mathbf{u}, \mathbf{b}$ auf einer Strecke

Ähnlichkeits-Transformationen

Kommen zur *Translation* und *Rotation* noch eine *gleichmässige (isotrope) Skalierung* hinzu, so spricht man von *Ähnlichkeits- Transformationen*

Homogene Repräsentation einer gleichmässigen Skalierung um den Faktor $s \in \mathbb{R}$ eines Objekts in \mathbb{R}^3 :

$$\tilde{\mathbf{S}} = \begin{pmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Invariant bleiben

- die Winkel eines Objekts
- das (Teil-)Verhältnis von Längen eines Objekts
- die Parallelität von Geraden eines Objekts
- die Inzidenz von Punkten und Geraden eines Objekts
- das Doppelverhältnis eines Objekts

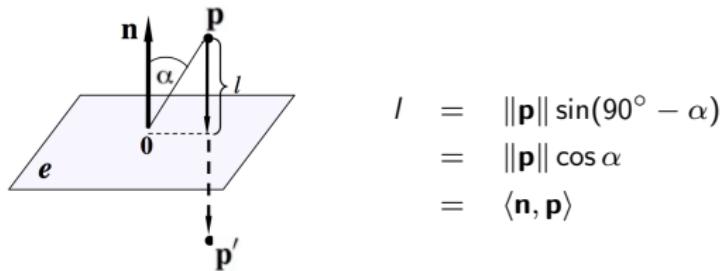
Spiegelungen

Eine *Punktspiegelung* am Ursprung bildet den Punkt $\mathbf{p} = (x, y, z)^T$ ab auf den Punkt $\mathbf{p}' = (-x, -y, -z)^T$. In Matrix-Schreibweise gilt also

$$\mathbf{p}' = -\mathbf{I} \cdot \mathbf{p} \quad (\text{isotrope Skalierung mit dem Faktor } -1)$$

Für eine *Spiegelung an der Ebene* \mathbf{e} : $\langle \mathbf{n}, \mathbf{x} \rangle = 0$ ($\|\mathbf{n}\| = 1$) durch den Ursprung gilt:

$$\mathbf{p}' = \mathbf{p} - 2\langle \mathbf{n}, \mathbf{p} \rangle \mathbf{n}$$



Durch Abbildung der Basisvektoren $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ erhält man die Matrix

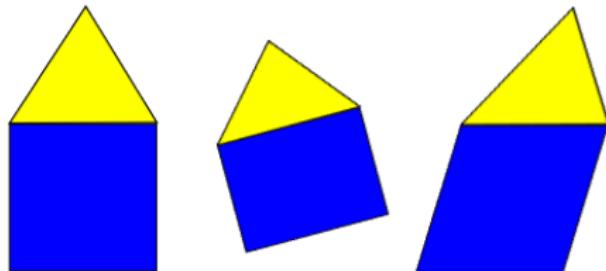
$$\mathbf{S} = \begin{pmatrix} 1 - 2n_x^2 & -2n_x n_y & -2n_x n_z \\ -2n_y n_x & 1 - 2n_y^2 & -2n_y n_z \\ -2n_z n_x & -2n_z n_y & 1 - 2n_z^2 \end{pmatrix} = \mathbf{I} - 2 \cdot \mathbf{n} \cdot \mathbf{n}^T$$

Affine Transformationen

Affine Transformationen sind Ähnlichkeits-Transformationen plus

- *anisotrope Skalierung*

$$\tilde{\mathbf{S}} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



- und *Scherung*

$$\tilde{\mathbf{A}} = \begin{pmatrix} 1 & s_{yx} & s_{zx} & 0 \\ s_{xy} & 1 & s_{zy} & 0 \\ s_{xz} & s_{yz} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ und } \tilde{\mathbf{A}}^{-1} = \frac{1}{\det} \begin{pmatrix} 1 - s_{zy}s_{yz} & s_{zx}s_{yz} - s_{yx} & s_{yx}s_{zy} - s_{zx} & 0 \\ s_{zy}s_{xz} - s_{xy} & 1 - s_{zx}s_{xz} & s_{zx}s_{xy} - s_{zy} & 0 \\ s_{xy}s_{yz} - s_{xz} & s_{yx}s_{xz} - s_{yz} & 1 - s_{yx}s_{xy} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\text{mit } \det = 1 + s_{xy}s_{yz}s_{zx} + s_{xz}s_{zy}s_{yx} - s_{xy}s_{yx} - s_{xz}s_{zx} - s_{yz}s_{zy}$$

Invariant bleiben: *Teilverhältnis, Parallelität, Inzidenz und Doppelverhältnis.*

Projektive Transformationen

Projektive Transformationen

bilden Punkte im Unendlichen (*Fernpunkte*) auf Punkte im Endlichen ab und umgekehrt, d.h. sie ändern die vierte Komponente des homogenen Koordinatenvektors eines Punktes im Raum.

Invariant bleiben: *Inzidenz* und *Doppelverhältnis*

Beispiel

Der *Fernpunkt* $\tilde{\mathbf{p}} = (\mathbf{p}^T, 0)^T$ wird durch die projektive Abbildung $\begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{v}^T & v \end{bmatrix}$ abgebildet auf $\begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{v}^T & v \end{bmatrix} \begin{pmatrix} \mathbf{p} \\ 0 \end{pmatrix} = \begin{pmatrix} \mathbf{Ap} \\ \langle \mathbf{v}, \mathbf{p} \rangle \end{pmatrix}$

was für $\langle \mathbf{v}, \mathbf{p} \rangle \neq 0$ kein Fernpunkt mehr ist. Umgekehrt wird der endliche Punkt $\tilde{\mathbf{q}} = (\mathbf{q}^T, 1)^T$ durch

$$\begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{v}^T & v \end{bmatrix} \begin{pmatrix} \mathbf{q} \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{Aq} + \mathbf{t} \\ \langle \mathbf{v}, \mathbf{q} \rangle + v \end{pmatrix}$$

für $\langle \mathbf{v}, \mathbf{q} \rangle = -v$ auf einen Fernpunkt abgebildet.

Übersicht räumliche Transformationen

Euklidische (starre) Transformationen haben 6 Freiheitsgrade (6 DOF)

$$\tilde{\mathbf{p}}' = \tilde{\mathbf{A}}_E \tilde{\mathbf{p}} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \tilde{\mathbf{p}} \quad \text{mit} \quad \mathbf{R}^T \mathbf{R} = \mathbf{I}$$

Ähnlichkeits-Transformationen haben 7 Freiheitsgrade (7 DOF)

$$\tilde{\mathbf{p}}' = \tilde{\mathbf{A}}_S \tilde{\mathbf{p}} = \begin{bmatrix} s\mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \tilde{\mathbf{p}} \quad \text{mit} \quad \mathbf{R}^T \mathbf{R} = \mathbf{I}$$

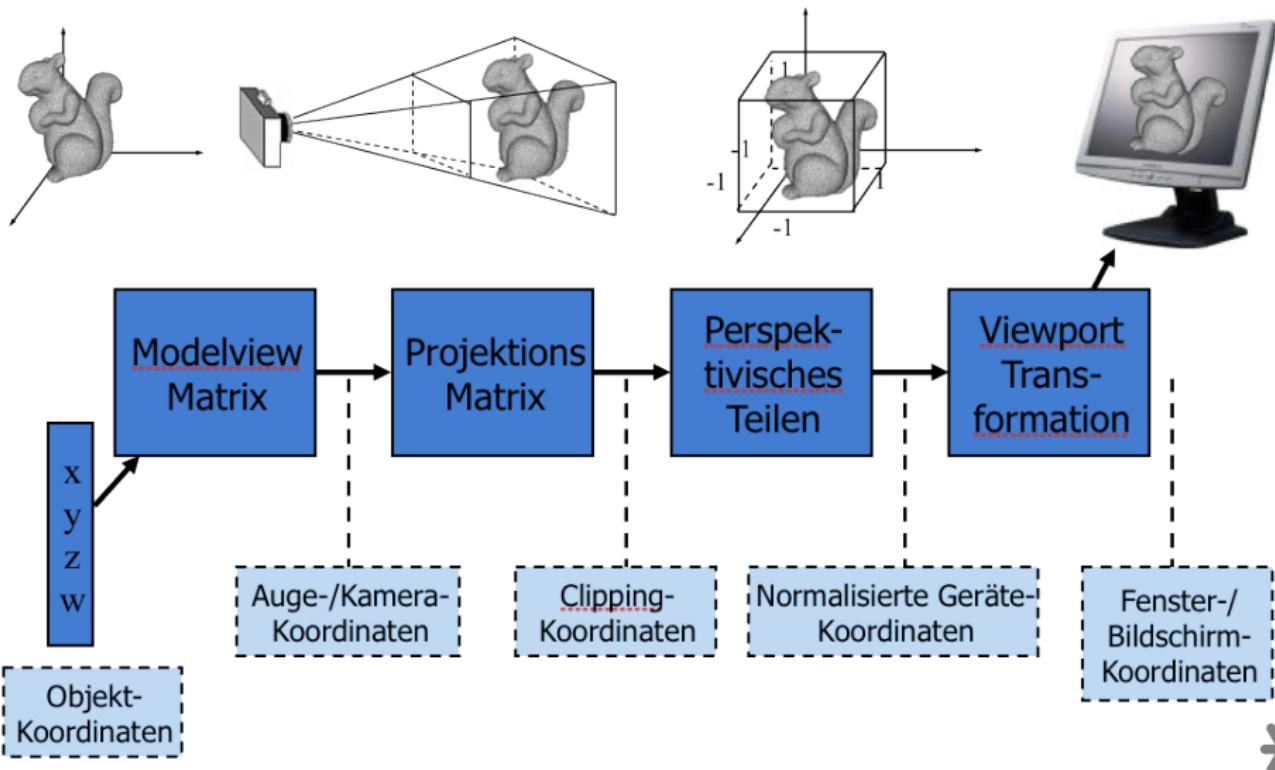
Affine Transformationen haben 12 Freiheitsgrade (12 DOF)

$$\tilde{\mathbf{p}}' = \tilde{\mathbf{A}}_A \tilde{\mathbf{p}} = \begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \tilde{\mathbf{p}} \quad \text{mit beliebigem} \quad \mathbf{A} \in \mathbb{R}^{3 \times 3}$$

Projektive Transformationen haben 15 Freiheitsgrade (15 DOF)

$$\tilde{\mathbf{p}}' = \tilde{\mathbf{A}}_P \tilde{\mathbf{p}} = \begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{v}^T & v \end{bmatrix} \tilde{\mathbf{p}} \quad \text{mit} \quad \mathbf{v} = (v_1, v_2, v_3)^T, v \in \mathbb{R}$$

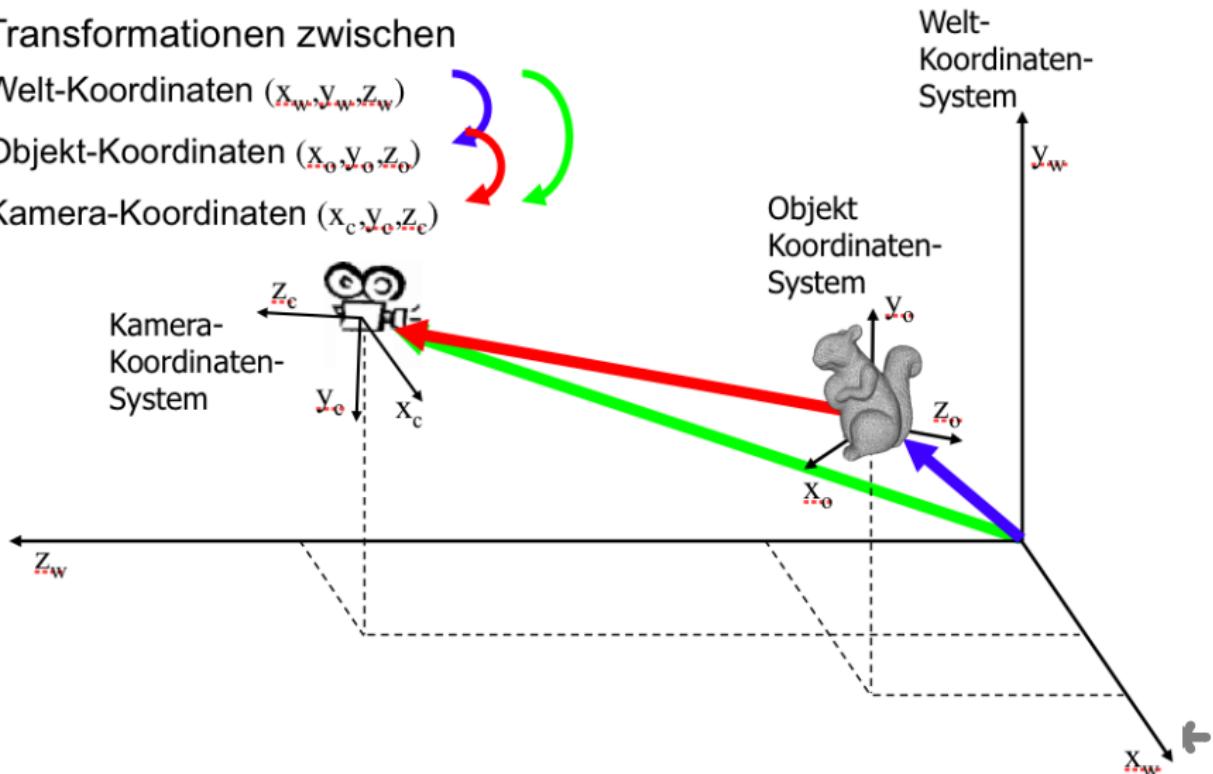
Die Geometrie Pipeline



Verschiedene Koordinatensysteme

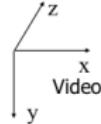
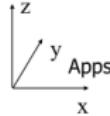
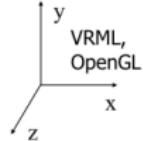
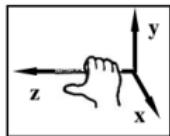
3D Transformationen zwischen

- Welt-Koordinaten (x_w, y_w, z_w)
- Objekt-Koordinaten (x_o, y_o, z_o)
- Kamera-Koordinaten (x_c, y_c, z_c)

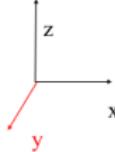
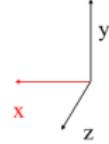
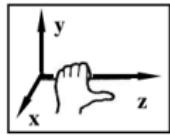


Wechsel zwischen Rechts-/Links-Systemen (Spiegelung)

Rechtshändige Systeme



Linkshändige Systeme



Transformationsmatrizen zum Wechsel zwischen links- und rechtshändigen Koordinatensystemen

$$\mathbf{M}_{R \leftarrow L}^{yz} = \mathbf{M}_{L \leftarrow R}^{yz} = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Spiegelung an der yz -Ebene

$$\mathbf{M}_{R \leftarrow L}^{xz} = \mathbf{M}_{L \leftarrow R}^{xz} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Spiegelung an der xz -Ebene

$$\mathbf{M}_{R \leftarrow L}^{xy} = \mathbf{M}_{L \leftarrow R}^{xy} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

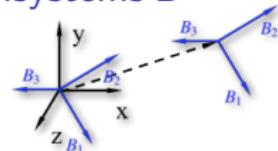
Spiegelung an der xy -Ebene

Transformation kartesischer Koordinatensysteme

Für die Transformation $\mathbf{M}_{E \leftarrow B}$ des kartesischen Koordinatensystems B in das kartesische Einheitskoordinatensystem E gilt

$$\mathbf{M}_{E \leftarrow B} = \mathbf{T}_{E \leftarrow B} \cdot \mathbf{R}_{E \leftarrow B}$$

$$= \begin{pmatrix} 1 & 0 & 0 & T_{(E \leftarrow B)_x} \\ 0 & 1 & 0 & T_{(E \leftarrow B)_y} \\ 0 & 0 & 1 & T_{(E \leftarrow B)_z} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} B_{1x} & B_{2x} & B_{3x} & 0 \\ B_{1y} & B_{2y} & B_{3y} & 0 \\ B_{1z} & B_{2z} & B_{3z} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} B_{1x} & B_{2x} & B_{3x} & T_{(E \leftarrow B)_x} \\ B_{1y} & B_{2y} & B_{3y} & T_{(E \leftarrow B)_y} \\ B_{1z} & B_{2z} & B_{3z} & T_{(E \leftarrow B)_z} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Für die inverse Transformation $\mathbf{M}_{B \leftarrow E} = \mathbf{M}_{E \leftarrow B}^{-1}$ gilt

$$\mathbf{M}_{B \leftarrow E} = \mathbf{M}_{E \leftarrow B}^{-1} = (\mathbf{T}_{E \leftarrow B} \cdot \mathbf{R}_{E \leftarrow B})^{-1} = \mathbf{R}_{E \leftarrow B}^{-1} \cdot \mathbf{T}_{E \leftarrow B}^{-1} = \mathbf{R}_{E \leftarrow B}^T \cdot \mathbf{T}_{E \leftarrow B}^{-1}$$

$$= \begin{pmatrix} B_{1x} & B_{1y} & B_{1z} & 0 \\ B_{2x} & B_{2y} & B_{2z} & 0 \\ B_{3x} & B_{3y} & B_{3z} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -T_{(E \leftarrow B)_x} \\ 0 & 1 & 0 & -T_{(E \leftarrow B)_y} \\ 0 & 0 & 1 & -T_{(E \leftarrow B)_z} \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} B_{1x} & B_{1y} & B_{1z} - \langle \mathbf{B}_1, \mathbf{T}_{(E \leftarrow B)} \rangle & 0 \\ B_{2x} & B_{2y} & B_{2z} - \langle \mathbf{B}_2, \mathbf{T}_{(E \leftarrow B)} \rangle & 0 \\ B_{3x} & B_{3y} & B_{3z} - \langle \mathbf{B}_3, \mathbf{T}_{(E \leftarrow B)} \rangle & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Koordinatensystemtransformation (Beispiel I)

Gegeben sind die Koordinatensysteme

- E mit dem Ursprung $E_u = (0, 0, 0)$ und den Basisvektoren $E_1 = (1, 0, 0)$, $E_2 = (0, 1, 0)$, $E_3 = (0, 0, 1)$
- B mit dem Ursprung $B_u = (5, 6, 0)$ und den Basisvektoren $B_1 = (1/\sqrt{2}, 1/\sqrt{2}, 0)$, $B_2 = (-1/\sqrt{2}, 1/\sqrt{2}, 0)$, $B_3 = (0, 0, 1)$

Die Transformationsmatrizen $\mathbf{M}_{E \leftarrow B}$ und $\mathbf{M}_{B \leftarrow E} = \mathbf{M}_{E \leftarrow B}^{-1}$ lauten

$$\mathbf{M}_{E \leftarrow B} = \begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 5 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 6 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{M}_{B \leftarrow E} = \mathbf{M}_{E \leftarrow B}^{-1} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & -\frac{11}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Koordinatensystemtransformation (Beispiel II)

Gegeben sind die Koordinatensysteme

- A mit dem Ursprung $A_u = (2, 3, 4)$ und den Basisvektoren
 $A_1 = (1, 0, 0)$, $A_2 = (0, 1/\sqrt{2}, 1/\sqrt{2})$, $A_3 = (0, -1/\sqrt{2}, 1/\sqrt{2})$
- B mit dem Ursprung $B_u = (5, 6, 0)$ und den Basisvektoren
 $B_1 = (1/\sqrt{2}, 1/\sqrt{2}, 0)$, $B_2 = (-1/\sqrt{2}, 1/\sqrt{2}, 0)$, $B_3 = (0, 0, 1)$

Die Transformationsmatrizen $\mathbf{M}_{B \leftarrow A}$ lautet

$$\mathbf{M}_{B \leftarrow A} = \mathbf{M}_{B \leftarrow E} \cdot \mathbf{M}_{E \leftarrow A}$$

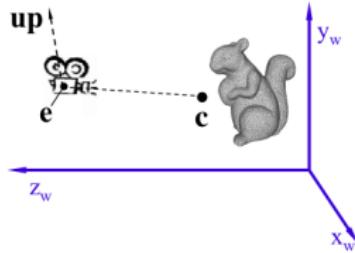
$$= \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & -\frac{11}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 3 \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 4 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{2} & -\frac{1}{2} & -\frac{4}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{2} & -\frac{1}{2} & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 4 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Betrachten einer Szene

Durch die *Modelltransformation* liegen
alle Punkte einer Szene in Weltkoordinaten vor

Betrachten der Szene aus verschiedenen Positionen und Richtungen durch
Transformation aller Punkte in ein *Ansichts-*
oder *Kamera-Koordinatensystem*

$$(x_c, y_c, z_c, 1)^T = \mathbf{M}_{LookAt} \cdot (x_w, y_w, z_w, 1)^T$$



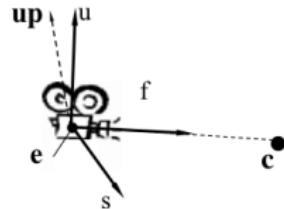
Die *LookAt* (*Ansichts-*)Transformation bzw. die Matrix \mathbf{M}_{LookAt} kann mit Hilfe einer *Kameraanalogie* beschrieben werden, d.h. durch

- die *Kameraposition/Augpunkt e* (eyepoint)
- den *Blickbezugspunkt c* (center)
- den *Obenvektor up* (upward, darf *nicht parallel* zum Vektor $c - e$ sein!)

Definition des Kamera-Koordinatensystems

Aus dem Punkten e , c und dem Vektor up kann man das *kartesische Kamera-Koordinatensystem* s, u, f erzeugen:

- 1 Normiere den Vektor up : $\text{up}' = \frac{\text{up}}{\|\text{up}\|}$
- 2 Berechne die (normierte) *Blickrichtung* $f = \frac{c-e}{\|c-e\|}$
- 3 Berechne die beiden Achsen $s = \frac{f \times \text{up}'}{\|f \times \text{up}'\|}$ und $u = s \times f$

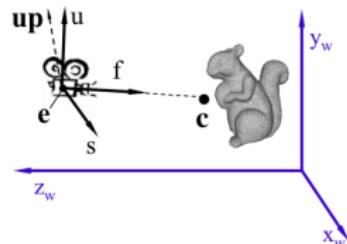


Für die Abbildung des Kamera-Koordinatensystems in das Welt-Koordinatensystem erhält man

$$\mathbf{M}_{W \leftarrow C} = \begin{pmatrix} 1 & 0 & 0 & e_x \\ 0 & 1 & 0 & e_y \\ 0 & 0 & 1 & e_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} s_x & u_x & -f_x & 0 \\ s_y & u_y & -f_y & 0 \\ s_z & u_z & -f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} s_x & u_x & -f_x & e_x \\ s_y & u_y & -f_y & e_y \\ s_z & u_z & -f_z & e_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Die Look-At (Ansichts) Transformation

Die Abbildung von Welt-Koordinaten in Kamera-Koordinaten geschieht also mit Hilfe der Matrix



$$\begin{aligned}
 \mathbf{M}_{\text{LookAt}} &= \mathbf{M}_{C \leftarrow W} = \mathbf{M}_{W \leftarrow C}^{-1} \\
 &= \left(\begin{pmatrix} 1 & 0 & 0 & e_x \\ 0 & 1 & 0 & e_y \\ 0 & 0 & 1 & e_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} s_x & u_x & -f_x & 0 \\ s_y & u_y & -f_y & 0 \\ s_z & u_z & -f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \right)^{-1} \\
 &= \begin{pmatrix} s_x & s_y & s_z & 0 \\ u_x & u_y & u_z & 0 \\ -f_x & -f_y & -f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} s_x & s_y & s_z & -\langle \mathbf{s}, \mathbf{e} \rangle \\ u_x & u_y & u_z & -\langle \mathbf{u}, \mathbf{e} \rangle \\ -f_x & -f_y & -f_z & \langle \mathbf{f}, \mathbf{e} \rangle \\ 0 & 0 & 0 & 1 \end{pmatrix}
 \end{aligned}$$

In OpenGL (GLU) siehe `gluLookAt(ex,ey,ez, cx,cy,cz, ux,uy,uz)`

Die Look-At (Ansichts) Transformation (Beispiel)

Ansichtskoordinaten der Punkte $\mathbf{p} = (0, 0, 0)^T$ und $\mathbf{q} = (1, 1, 1)^T$ bei den Kameraparametern Augpunkt $\mathbf{e} = (1, 2, 2)^T$, Blickbezugspunkt $\mathbf{e}' = (0, 0, 0)^T$ und Obenvektor $\mathbf{up} = (0, 1, 0)^T$

Es gilt

$$\mathbf{up}' = \frac{\mathbf{up}}{\|\mathbf{up}\|} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad \mathbf{f} = \frac{\mathbf{c} - \mathbf{e}}{\|\mathbf{c} - \mathbf{e}\|} = \begin{pmatrix} -\frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \\ -\frac{2}{\sqrt{3}} \end{pmatrix}, \quad \mathbf{s} = \frac{\mathbf{f} \times \mathbf{up}'}{\|\mathbf{f} \times \mathbf{up}'\|} = \begin{pmatrix} -\frac{2}{\sqrt{5}} \\ 0 \\ -\frac{1}{\sqrt{5}} \end{pmatrix}, \quad \mathbf{u} = \mathbf{s} \times \mathbf{f} = \begin{pmatrix} -\frac{2}{3\sqrt{5}} \\ \frac{5}{3\sqrt{5}} \\ \frac{4}{3\sqrt{5}} \end{pmatrix}$$

und damit

$$\mathbf{M}_{LookAt} = \begin{pmatrix} s_x & s_y & s_z & -\langle \mathbf{s}, \mathbf{e} \rangle \\ u_x & u_y & u_z & -\langle \mathbf{u}, \mathbf{e} \rangle \\ -f_x & -f_y & -f_z & \langle \mathbf{f}, \mathbf{e} \rangle \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \frac{2}{\sqrt{5}} & 0 & -\frac{1}{\sqrt{5}} & 0 \\ \frac{-2}{3\sqrt{5}} & \frac{5}{3\sqrt{5}} & \frac{-4}{3\sqrt{5}} & 0 \\ \frac{1}{3} & \frac{2}{3} & \frac{2}{3} & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

also

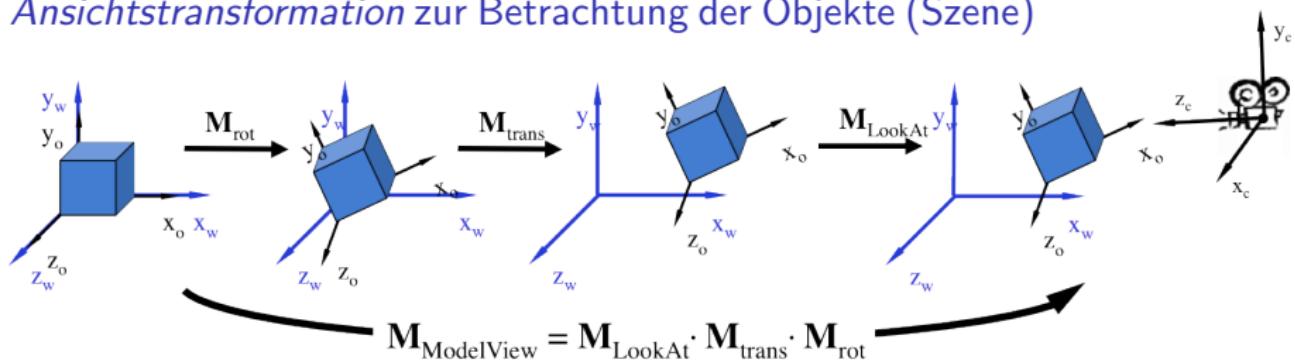
$$\mathbf{p}_c = \mathbf{M}_{LookAt} \cdot \mathbf{p} = (0, 0, -3)^T, \quad \mathbf{q}_c = \mathbf{M}_{LookAt} \cdot \mathbf{q} = \left(\frac{1}{\sqrt{5}}, -\frac{1}{3\sqrt{5}}, -\frac{4}{3}\right)^T$$

Modell-View-Transformationen

Positionierung von Objekten im Raum durch Modelltransformationen

- Zu Beginn sind Objekt- und Welt-Koordinatensystem deckungsgleich
- Schrittweise Ausführung von Einzeltransformationen (Rotation, Translation, Skalierung, ...) bringt Objekt in die gewünschte Lage

Ansichtstransformation zur Betrachtung der Objekte (Szene)



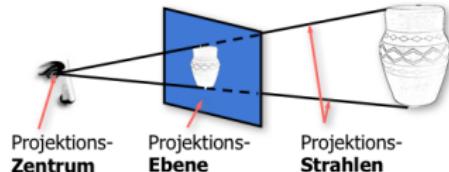
Verringelter Rechenaufwand durch Zusammenfassen aller Einzeltransformationen zu einer Gesamt-Modell-View-Matrix

Abbildung von 3D Objekten auf 2D Bildschirme

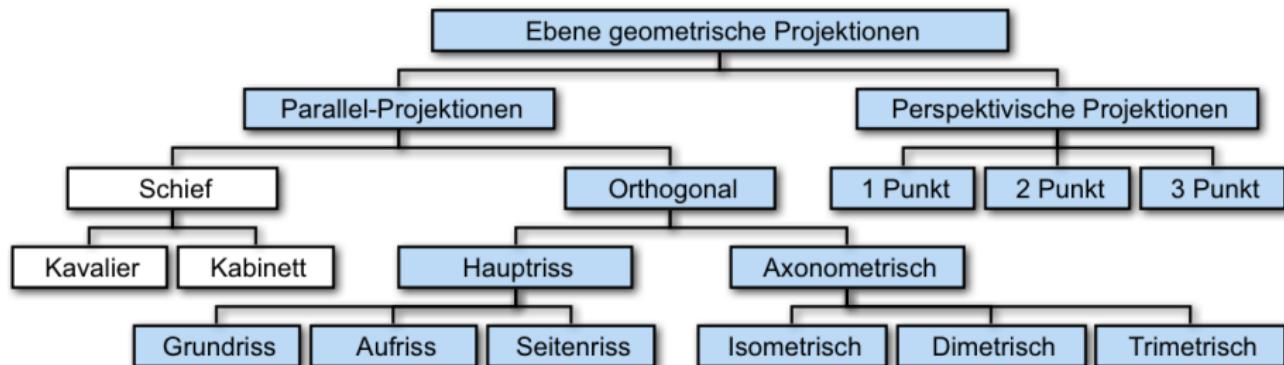
Projektionen (in der CG) bilden 3D Objekte auf 2D (Bild-)Ebene ab

Kennzeichen dieser *ebenen geometrischen Projektionen*:

- Ein (oder mehrere) *Projektions-Zentrum*
- Abbildung auf eine *Projektions-Ebene*
- Nur *Geraden als Projektionsstrahlen*



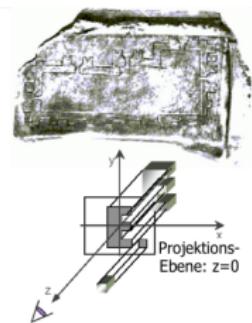
Klassifikation ebener Projektionen



Orthogonale Paralellprojektionen (Hauptrisse)

Parallelität bleibt erhalten

- Verwendung in Ingenieurwesen und Architektur
- Schlechter räumlicher Eindruck
 - Auch Objekte hinter dem „Betrachter“ werden auf die „Bildebene“ projiziert
- Je nach Projektionsrichtung spricht man von
 - *Aufriss* (Projektion Richtung x -Achse: $x = 0$)
 - *Seitenriss* (Projektion Richtung y -Achse: $y = 0$)
 - *Grundriss* (Projektion Richtung z -Achse: $z = 0$)



Zugehörige Projektionsmatrizen

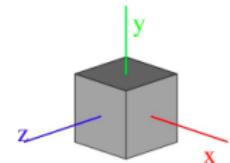
$$\mathbf{P}_{\text{Aufriss}} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{P}_{\text{Seitenriss}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{P}_{\text{Grundriss}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Orthogonale Parallelprojektionen (Axonometrien)

Isometrische Projektion

Alle Winkel zwischen Projektionsrichtung und Koordinatenachsen des Modellkoordinatensystems sind gleich

(OpenGL) Beispiel:
Isometrie

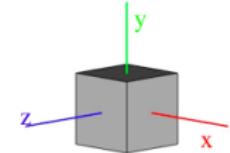


```
glOrtho(-2,2,-2,2,1,100)
gluLookAt(2,2,2,0,0,0,0,1,0)
```

Dimetrische Projektion

Winkel zwischen Projektionsrichtung und zwei Koordinatenachsen des Modellkoordinatensystems sind gleich

(OpenGL) Beispiel:
Dimetrie

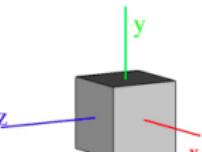


```
glOrtho(-2,2,-2,2,1,100)
gluLookAt(2,0.5,2,0,0,0,0,1,0)
```

Trimetrische Projektion

Winkel zwischen Projektionsrichtung und allen drei Koordinatenachsen des Modellkoordinatensystems sind verschieden

(OpenGL) Beispiel:
Trimetrie

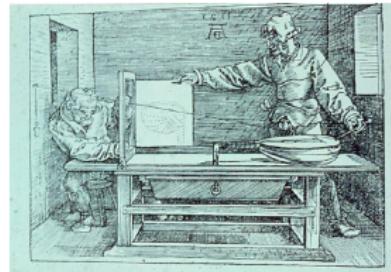


```
glOrtho(-2,2,-2,2,1,100)
gluLookAt(5,1,3,0,0,0,0,1,0)
```

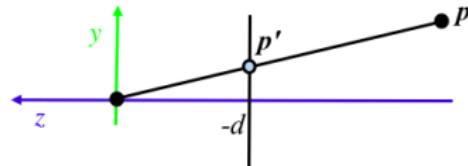
Perspektivische (Zentral-)Projektion

Realistischer durch *perspektivische Verzerrung*

- Änderung von Längenverhältnissen und Winkeln
- Parallel Geraden konvergieren
- Was passiert mit Objekten hinter Betrachter?



Beispiel: Projektion auf Ebene parallel zur xy–Ebene mit Abstand d



Nach *Strahlensatz* gilt

$$\frac{y'}{d} = \frac{y}{-z} \implies y' = \frac{y}{-z/d}$$

In *homogenen Koordinaten* gilt

$$(x', y', z', 1)^T = (d \cdot x, d \cdot y, d \cdot z, -z)^T$$

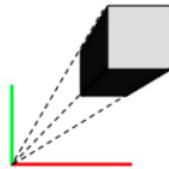
und die zugehörige Projektionsmatrix lautet dann

$$P_{Z_{xy}}(d) = \begin{pmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Perspektivische (1, 2, 3 Punkt) Projektionen

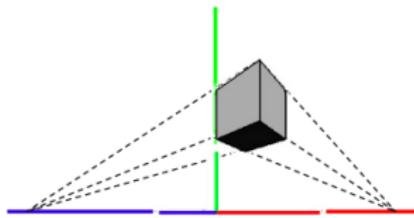
Schneidet die Projektionsebene *eine*, *zwei* oder *drei* Koordinatenachsen des Ansichtskoordinatensystems, erhält man

1-Punkt-Perspektive



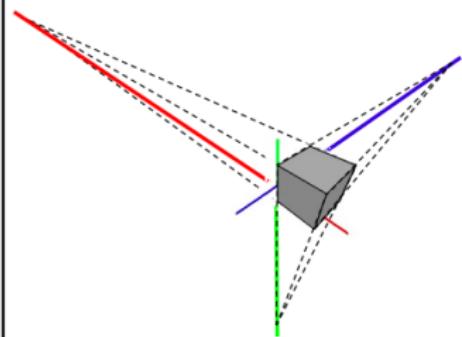
```
gluPerspective(120,  
              1,1,100)  
gluLookAt(0,0,4,  
          0,0,0,  
         0,1,0)
```

2-Punkt-Perspektive



```
gluPerspective(120,  
              1,1,100)  
gluLookAt(4,0,4,  
          0,0,0,  
         0,1,0)
```

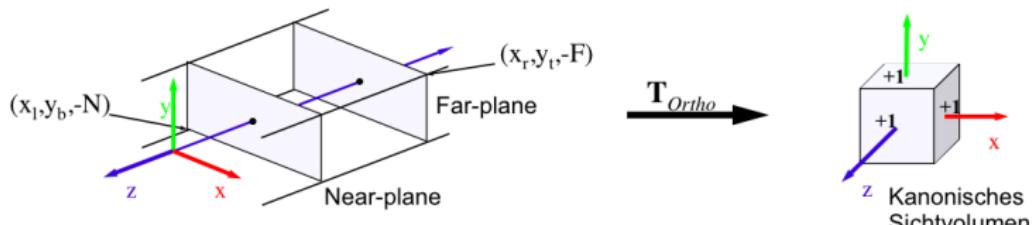
3-Punkt-Perspektive



```
gluPerspective(120,  
              1,1,100)  
gluLookAt(3,4,3,  
          0,0,0,  
         0,1,0)
```

Orthogonale View Frustum Transformation

Orthogonale Transformation des sichtbaren Bereichs (View Frustum) in das kanonische Sichtvolumen $[-1, 1]^3$



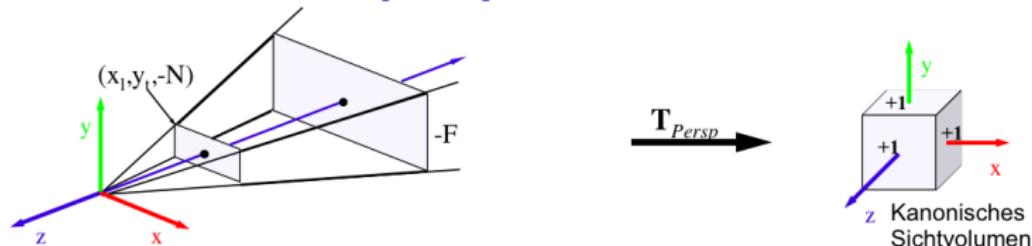
- Als Transformations-Matrix erhält man

$$\mathbf{T}_{Ortho} = \mathbf{T}_{Scale} \mathbf{T}_{Trans} = \begin{pmatrix} \frac{2}{x_r - x_l} & 0 & 0 & 0 \\ 0 & \frac{2}{y_t - y_b} & 0 & 0 \\ 0 & 0 & \frac{-2}{F - N} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{x_r + x_l}{2} \\ 0 & 1 & 0 & -\frac{y_t + y_b}{2} \\ 0 & 0 & 1 & -\frac{F + N}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \frac{2}{x_r - x_l} & 0 & 0 & -\frac{x_r + x_l}{x_r - x_l} \\ 0 & \frac{2}{y_t - y_b} & 0 & -\frac{y_t + y_b}{y_t - y_b} \\ 0 & 0 & \frac{-2}{F - N} & -\frac{F + N}{F - N} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

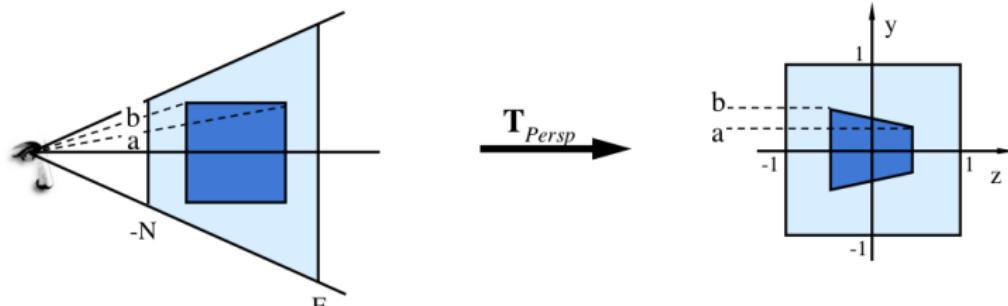
- In OpenGL: `glOrtho(xl, xr, yb, yt, zn, zf)`
- Anschließend einfaches *Clippen* nicht sichtbarer Objekte

Perspektivische View Frustum Transformation

Perspektivische Transformation des sichtbaren Bereichs (View Frustum) in das kanonische Sichtvolumen $[-1, 1]^3$



Die perspektivische View Frustum Transformation verformt sichtbare Objekte so, dass sie noch orthogonaler Transformationen aussehen, wie die originalen Objekte nach perspektivischer Transformation



Perspektivische View Frustum Transformation

- Nach Strahlensatz gilt $x' = \frac{N \cdot x}{-z}$ und $y' = \frac{N \cdot y}{-z}$
- Mit dem Ansatz $z' = \frac{a \cdot z + b}{-z}$ und $-1 = \frac{a \cdot (-N) + b}{N}, 1 = \frac{a \cdot (-F) + b}{F}$ erhält man

$$a = -\frac{F + N}{F - N}, b = -\frac{2 \cdot F \cdot N}{F - N} \implies \mathbf{P}_{Z_{xy}}(N, F) = \begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & -\frac{F+N}{F-N} & -\frac{2 \cdot F \cdot N}{F-N} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

- Insgesamt gilt dann

$$\mathbf{T}_{Persp} = \begin{pmatrix} \frac{2}{x_r - x_l} & 0 & 0 & 0 \\ 0 & \frac{2}{y_t - y_b} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{x_r + x_l}{2} \\ 0 & 1 & 0 & -\frac{y_t + y_b}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \mathbf{P}_{Z_{xy}}(N, F) = \begin{pmatrix} \frac{2 \cdot N}{x_r - x_l} & 0 & \frac{x_r + x_l}{x_r - x_l} & 0 \\ 0 & \frac{2 \cdot N}{y_t - y_b} & \frac{y_t + y_b}{y_t - y_b} & 0 \\ 0 & 0 & -\frac{F+N}{F-N} & -\frac{2 \cdot F \cdot N}{F-N} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

- In OpenGL: `glFrustum(xl.xr, yb,yt, n,f)`



Perspektivische View Frustum Transformation (Beispiel)

Gegeben sei ein Sichtvolumen mit

$$x_l = -2, x_r = 2, y_b = -4, y_t = 4, N = 1, F = 100$$

Welche Clip-Koordinaten hat der Punkt $\mathbf{p} = (10, 10, -10)^T$?

Mit der Matrix für die perspektivische Frustum Transformation

$$\mathbf{T}_{Persp} = \begin{pmatrix} \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{4} & 0 & 0 \\ 0 & 0 & -\frac{101}{99} & -\frac{200}{99} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

erhält man als Clip-Koordinaten $\mathbf{T}_{Persp} \cdot (10, 10, -10, 1)^T = (5, \frac{5}{2}, \frac{90}{11}, 10)^T$



Clipping am kanonischen Sichtvolumen

Orthogonale/perspektivische Frustum Transformationen überführen die Kamera-Koordinaten eines Punktes in *homogene Clip-Koordinaten*

Ein Punkt \mathbf{p} mit Clip-Koordinaten (x_c, y_c, z_c, w_c) liegt genau dann im kanonischen Sichtvolumen $[-1, 1]^3$, wenn gilt

$$\underbrace{(w_c + x_c) > 0}_{\text{linke Seite}} \wedge \underbrace{(w_c - x_c) > 0}_{\text{rechte Seite}} \wedge \underbrace{(w_c + y_c) > 0}_{\text{untere Seite}} \wedge \underbrace{(w_c - y_c) > 0}_{\text{obere Seite}} \wedge \underbrace{(w_c + z_c) > 0}_{\text{vordere Seite}} \wedge \underbrace{(w_c - z_c) > 0}_{\text{hintere Seite}} = \text{True}$$

Beispiel: Wo liegen die Punkte \mathbf{p}, \mathbf{q} mit Clip-Koordinaten

$$\mathbf{p}_c = \left(5, \frac{5}{2}, \frac{90}{11}, 10\right)^T, \mathbf{q}_c = \left(2, 10, \frac{1}{9}, 1\right)^T?$$

- \mathbf{p} liegt *im* Sichtvolumen
- \mathbf{q} liegt wegen $(w_c - x_c) = (1 - 2) < 0$ *rechts* und wegen $(w_c - y_c) = (1 - 10) < 0$ *oberhalb* des (kanonischen) Sichtvolumens

Perspektivisches Teilen

Durch Frustum Transformationen entstehen *homogene (Clip-)Koordinaten*

- Homogene Komponente ist bei orthogonaler Frustum Transformation gleich 1
- *Perspektivisches Teilen* (Division mit w_c -Koordinate) erzeugt aus den homogenen Clip-Koordinaten nicht homogene normalisierte Gerätekordinaten
- Fass man alle Schritte der perspektivischen Transformation eines Punktes $\mathbf{p} = (x, y, z)^T$ (in Kamerakoordinaten) in normalisierte Gerätekordinaten zusammen, erhält man

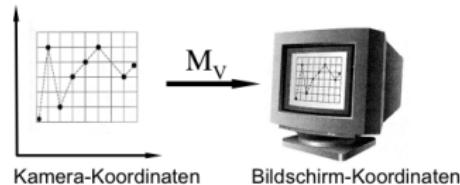
$$\begin{pmatrix} x_d \\ y_d \\ z_d \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{x_c}{w_c} \\ \frac{y_c}{w_c} \\ \frac{z_c}{w_c} \\ 1 \end{pmatrix} = \begin{pmatrix} -\frac{2 \cdot N}{x_r - x_f} \cdot \frac{x}{z} - \frac{x_r + x_f}{x_r - x_f} \\ -\frac{2 \cdot N}{y_t - y_b} \cdot \frac{y}{z} - \frac{y_t + y_b}{y_t - y_b} \\ -\frac{F+N}{F-N} + \frac{2 \cdot F \cdot N}{F-N} \cdot \frac{1}{z} \\ 1 \end{pmatrix}$$

- Beispiel: Die normalisierten Gerätekordinaten des Punktes \mathbf{p} mit den Clip-Koordinaten $\mathbf{p}_c = (5, \frac{5}{2}, \frac{90}{11}, 10)^T$ lauten $\mathbf{p}_d = (\frac{1}{2}, \frac{1}{4}, \frac{9}{11}, 1)^T$

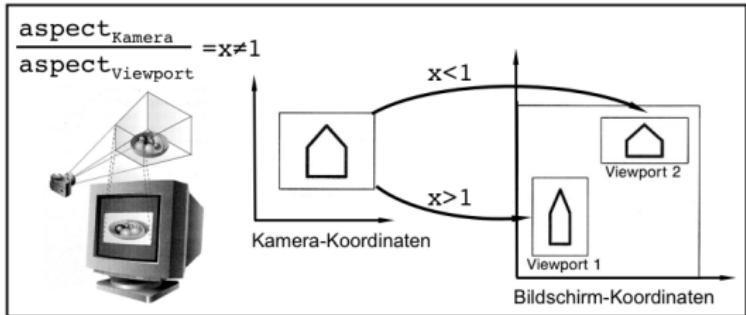
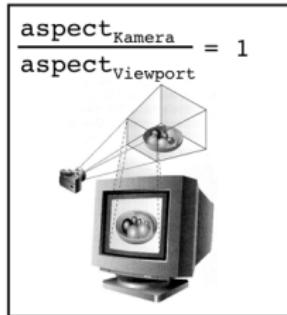
Window-to-Viewport Transformation

Die (*Window-to-*-)Viewport Transformation bestimmt die Bildgröße

- Bei einer Kamera ist dies die Größe des Abzugs (Poster, Postkarte, ...)
- In der Computergrafik ist der Viewport (das Darstellungsfeld) der rechtwinkelige Bereich, auf dem das Bild erscheint

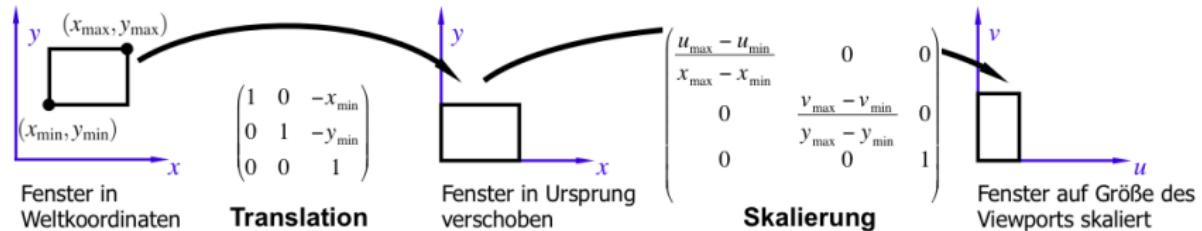


Haben der Bereich in Kamera-Koordinaten und der Viewport nicht das gleiche Seitenverhältnis aspect = width/height entstehen Verzerrungen



Matrix der Window-to-Viewport Transformation

Für die Transformation des Fensters $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ in den Viewport $[u_{\min}, u_{\max}] \times [v_{\min}, v_{\max}]$ gilt



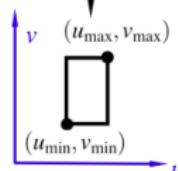
Gesamttransformation

$$\mathbf{M}_{WV} = \begin{pmatrix} 1 & 0 & u_{\min} \\ 0 & 1 & v_{\min} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \frac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}} & 0 & 0 \\ 0 & \frac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}} & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & -x_{\min} \\ 0 & 1 & -y_{\min} \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}} & 0 & -x_{\min} \cdot \frac{u_{\max} - u_{\min}}{x_{\max} - x_{\min}} + u_{\min} \\ 0 & \frac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}} & -y_{\min} \cdot \frac{v_{\max} - v_{\min}}{y_{\max} - y_{\min}} + v_{\min} \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & u_{\min} \\ 0 & 1 & v_{\min} \\ 0 & 0 & 1 \end{pmatrix}$$

Translation



Fenster in Bildschirmkoordinaten (skaliert & verschoben)

2D/3D Clipping

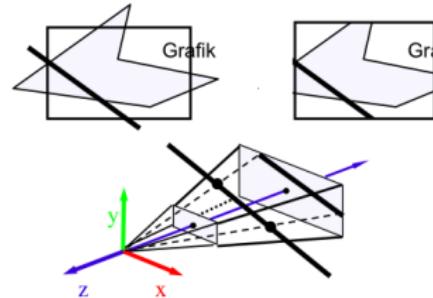
Unter *Clipping* versteht man das Beschneiden grafischer Darstellungselemente an Begrenzungs-Linien/Kurven bzw. -Ebenen/Flächen
 Clipping von *Bitmaps* und *analytisch beschriebenen Objekten*

Man unterscheidet

- 2D-Clipping
 - Viewport oder allgemeine Polygone
- 3D-Clipping
 - View-Frustum oder allgemeine Clip-Ebenen

und

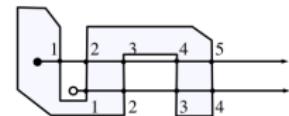
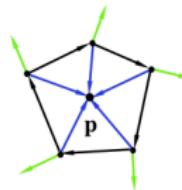
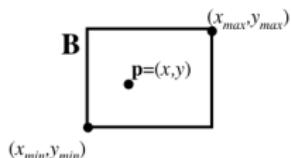
- Analytische Techniken
 - Schnittpunktberechnung und Bestimmung der sichtbaren Segmente
- Scissor Techniken
 - Nur Pixel innerhalb einer *Scissor-Box* werden in Bildspeicher geschrieben
- Stencil Techniken
 - Nur Pixel innerhalb eines *maskierten Bereichs* werden in Bildspeicher



Point Clipping

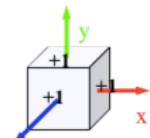
2D Point Clipping

- An *Bounding-Box* $\mathbf{B} = [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$
 - $\mathbf{p} = (x, y)^T$ liegt in \mathbf{B} , genau dann, wenn gilt
 $(x - x_{\min}) > 0 \wedge (x_{\max} - x) > 0 \wedge (y - y_{\min}) > 0 \wedge (y_{\max} - y) > 0 \wedge == \text{True}$
- An *konvexem Polygon* $\mathbf{P} = \{\mathbf{p}_0, \dots, \mathbf{p}_n\}$
 - z. B. mittels Skalarprodukt (siehe Vorlesung I)
- An *beliebigem Polygon* $\mathbf{P} = \{\mathbf{p}_0, \dots, \mathbf{p}_n\}$
 - *Beobachtung:* Der Punkt \mathbf{p} liegt im Polygon \mathbf{P} , wenn ein von \mathbf{p} ausgehender Strahl eine *ungerade Anzahl von Schnittpunkten mit \mathbf{P}* hat



3D Point Clipping

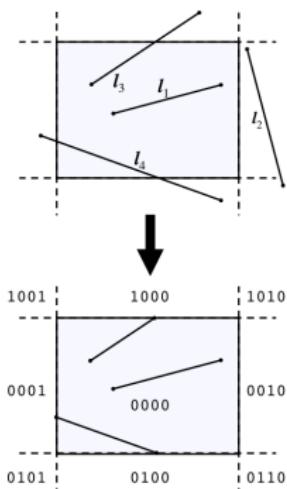
Clipping am kanonischen Sichtvolumen (siehe Vorlesung IV)



2D Line Clipping nach Cohen Sutherland

Drei Endpunkt-Konstellationen einer Linie

- 1 Beide Endpunkte innerhalb des Clip-Rechtecks
 - Linie ist komplett sichtbar (Beispiel: I_1)
- 2 Beide Endpunkte auf einer Seite einer Kante
 - Linie ist komplett unsichtbar (Beispiel: I_2)
- 3 Es gelten weder 1. noch 2.
 - Man kann wiederum zwei Fälle unterscheiden
 - Linie schneidet das Clip-Rechteck (Beispiel: I_3)
 - Linie ausserhalb des Clip-Rechtecks (Beispiel: I_4)

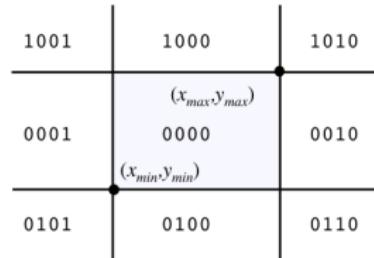
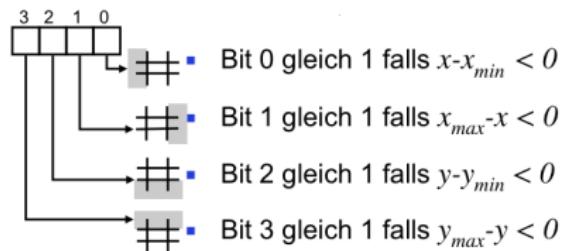


Algorithmus von Cohen und Sutherland

- 4-Bit-Bereichscode Algorithmus
- Ermittelt effizient die Fälle 1. und 2.
- Fall 3. erfordert zusätzliche Tests

Der Cohen Sutherland Algorithmus

4-Bit-Bereichscode für jeden Punkt $\mathbf{p} = (x, y)$



Testen einer Linie mit Startpunkt $\mathbf{p}_1 = (x_1, y_1)$, Bereichscode $c1$ und Endpunkt $\mathbf{p}_2 = (x_2, y_2)$, Bereichscode $c2$

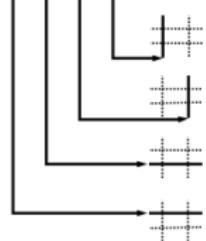
- 1 Fall: *Triviales Annehmen*, wenn $c1 | c2 == 0$
 - Linie liegt *komplett innerhalb* des Clip-Rechtecks (ist sichtbar)
- 2 Fall: *Triviales Abweisen*, wenn $c1 \& c2 != 0$
 - Linie liegt *komplett ausserhalb* des Clip-Rechtecks (ist unsichtbar)
- 3 Fall: weder 1. noch 2. Fall, d.h. $c1 | c2 != 0$ und $c1 \& c2 == 0$
 - Linie teilweise oder vollständig unsichtbar (*weitere Tests notwendig*)

Der Fall $c1|c2 \neq 0$ und $c1\&c2 == 0$

Schnittpunktberechnung abhängig vom Linecode $lc = c1|c2$

- Betrachte

$$lc = \begin{array}{cccc} 3 & 2 & 1 & 0 \\ \boxed{} & \boxed{} & \boxed{} & \boxed{} \end{array}$$



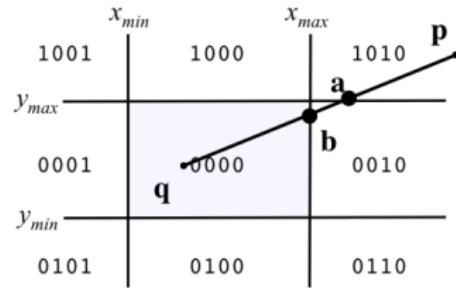
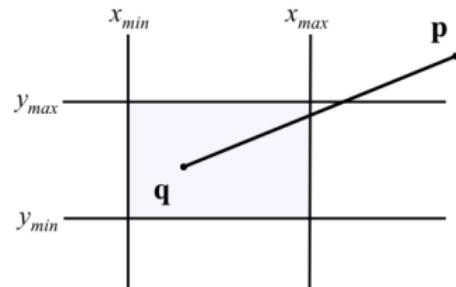
- Bit 0 = 1 : Schnittpunkt mit x_{min} berechnen
- Bit 1 = 1 : Schnittpunkt mit x_{max} berechnen
- Bit 2 = 1 : Schnittpunkt mit y_{min} berechnen
- Bit 3 = 1 : Schnittpunkt mit y_{max} berechnen

- y -Koordinate des Schnittpunktes mit x -Achse ist $y = \frac{y_2 - y_1}{x_2 - x_1}(x - x_2) + y_2$
- x -Koordinate des Schnittpunktes mit y -Achse ist $x = \frac{x_2 - x_1}{y_2 - y_1}(y - y_2) + x_2$
- Anschließend wird solange bis $c1|c2 == 0$ oder $c1\&c2 \neq 0$
 - der äußere Punkt durch den Schnittpunkt ersetzt
 - der Bereichscode des neuen Schnittpunktes ermittelt

Der Cohen Sutherland Algorithmus (Beispiel)

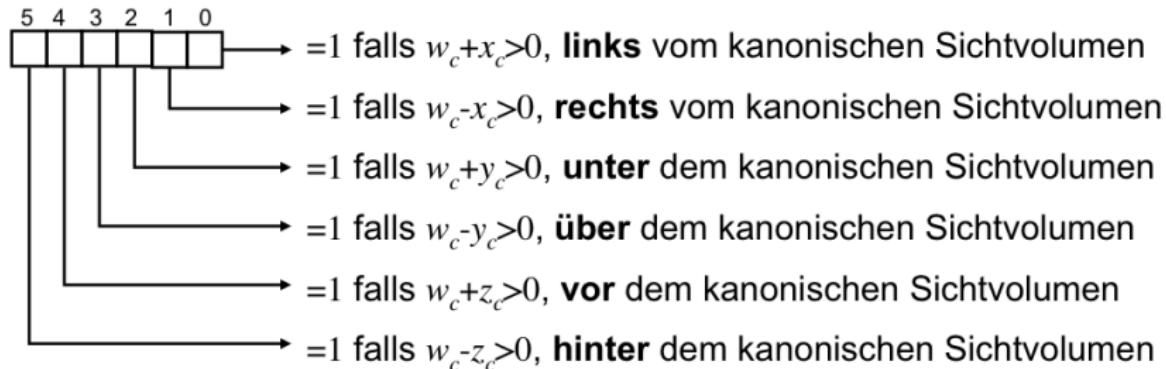
Es soll die Linie mit Startpunkt $\mathbf{p} = (8, 5)^T$ und Endpunkt $\mathbf{q} = (2, 2)^T$ am Clipping-Rechteck $\mathbf{B} = [1, 1] \times [5, 4]$ geclipt werden

- Bereichscodes: \mathbf{p} : $c_1=1010$, \mathbf{q} : $c_2=0000$
- $c_1 | c_2=1010$: *kein triviales Annehmen*
- $c_1 \& c_2=0000$: *kein triviales Abweisen*
- Linecode: $lc=1010$
 - Ersetze \mathbf{p} durch Schnittpunkt \mathbf{a} mit y_{max}
 - Bereichscode von \mathbf{a} : $c_3=0010$
 - $c_3 | c_2=0010$ und $c_3 \& c_2=0000$
 - Linecode: $lc_2=0010$
 - Ersetze \mathbf{a} durch Schnittpunkt \mathbf{b} mit x_{max}
 - Bereichscode von \mathbf{b} : $c_4=0000$
 - $c_4 | c_2=0000$: *triviales Annehmen* der Linie von \mathbf{b} nach \mathbf{q}



3D Line Clipping nach Cohen Sutherland

6-Bit Bereichscode für $\mathbf{p} = (x_c, y_c, z_c, w_c)$ (Verallgemeinerung des 2D Falls)

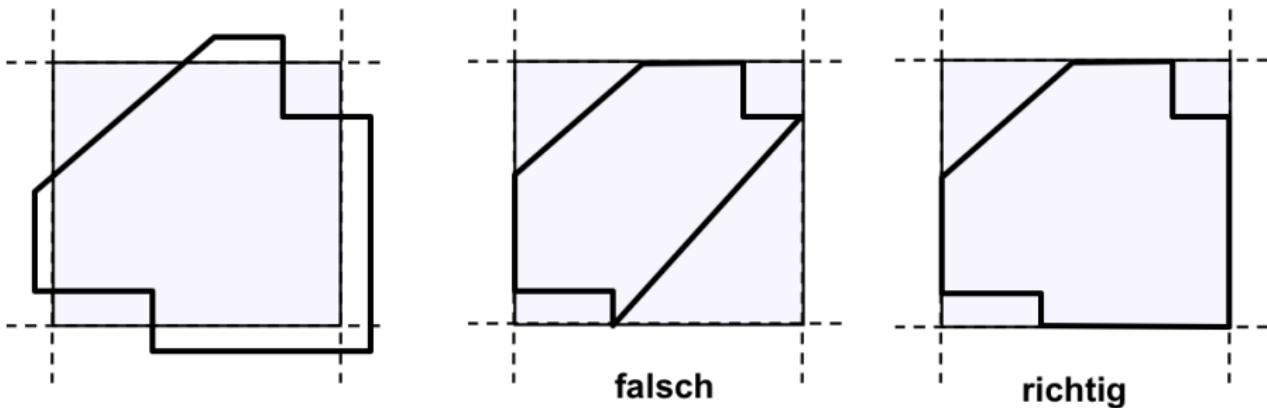


Wieder drei Fälle bei Test der Bereichscodes c_1, c_2 von Anfangs- und Endpunkt einer zu testenden Linie

- 1 Fall: *Triviales Annehmen*, wenn $c_1 | c_2 == 0$ (Linie komplett sichtbar)
- 2 Fall: *Triviales Abweisen*, wenn $c_1 \& c_2 != 0$ (Linie komplett unsichtbar)
- 3 Fall: weder 1. noch 2. Fall, d.h. $c_1 | c_2 != 0$ und $c_1 \& c_2 == 0$
 - Linie teilweise oder vollständig unsichtbar (*weitere Tests notwendig*)

2D Clipping von Polygonen

Einfaches Verbinden der Schnittpunkte zwischen Polygonseiten und Fenster führt in der Regel zum falschen Ergebnis

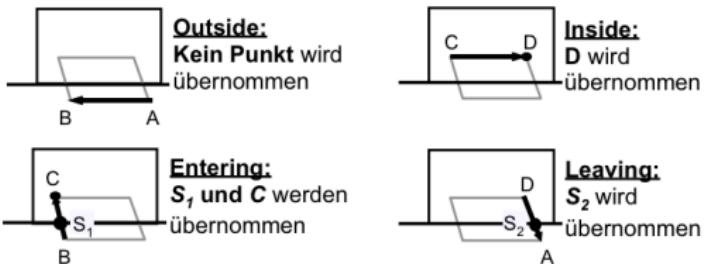


Es werden zusätzliche Informationen benötigt, da Flächen und nicht Strecken zugeschnitten werden

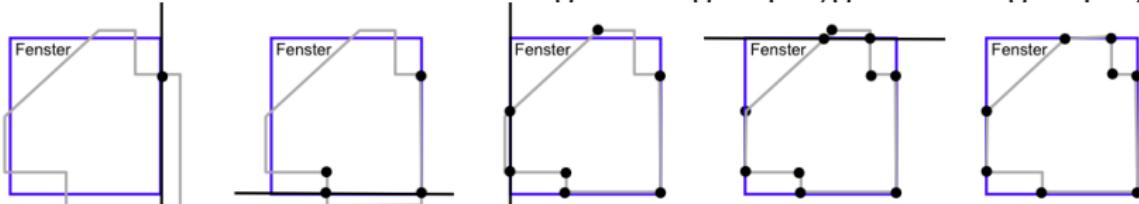
Der Sutherland-Hodgman Algorithmus

Verallgemeinerung des Cohen-Sutherland Algorithmus

- Statt gegen Fenster, nacheinander gegen 4 Fenstergeraden zuschneiden
- Zuschneiden einer Kante liefert 0, 1 oder 2 Punkte, welche in das Ausgabepolygon übernommen werden, abhängig von den vier Fällen

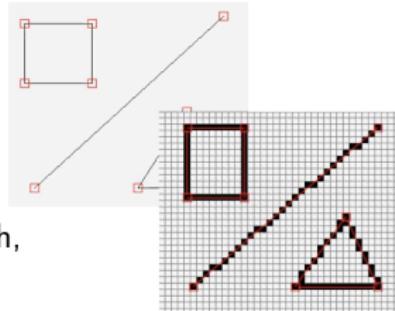
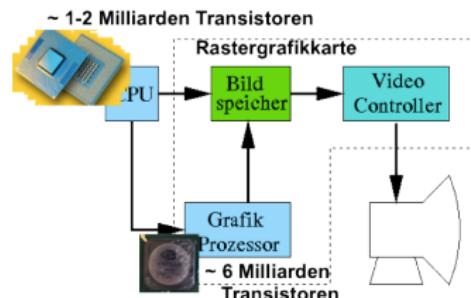


- Überprüfe nacheinander alle Polygonkanten an den 4 Fenstergeraden
 - Schritt für Schritt Überführung des Eingabepolygons in Ausgabepolygon



Rastergrafik

- *Rasterbildschirme* seit Anfang 1970
- Basis bilden diskrete Bildpunkte
 - Pixel (*Picture element*)
- Kontinuierliche Grafikobjekte (Linien, ...) müssen konvertiert werden
 - Rasterkonvertierung (Scan conversion)
- Vorteile
 - Günstiger als Vektorgrafik-Bildschirme
 - Darstellung (mit Mustern) gefüllter Flächen
- Nachteile
 - Diskrete Darstellung durch Pixel
 - Grafikobjekte (Linien, ...), meist kontinuierlich, müssen in Rasterbild konvertiert werden
 - Durch (grobe) Rasterung entstehen Treffeneffekte (Aliasing, Abtastproblematik)



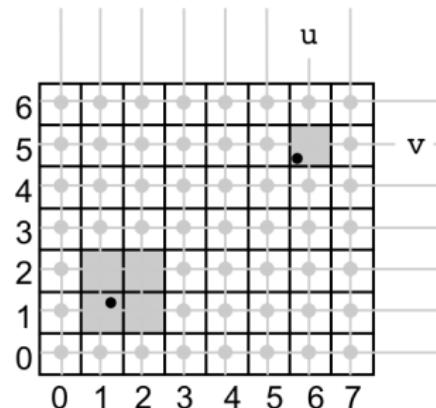
Rasterisieren von Punkten

Darstellung eines Punktes $\mathbf{p} = (x, y)$ mit (reellen) Koordinaten in (ganzzahligen) Bildschirmkoordinaten (u, v) durch Runden

```
u = round(x)
y = round(y)
```

Festlegung (ganzzahliger) Punktgrößen n durch quadratische Anordnung von $n \times n$ Pixeln der Größe s

```
def setPoint(x, y, n):
    dx = 2*(x - floor(x)))>1 and not n%2
    dy = 2*(y - floor(y)))>1 and not n%2
    startX = x - s*(n/2 - dx)
    startY = y - s*(n/2 - dy)
    for i in range(n):
        for j in range(n):
            u = round(startX + i*s)
            v = round(startY + j*s)
            setPixel(u, v)
```



Rasterisieren von Geraden (naiv und DDA)

Rasterisierung der Strecke mit Endpunkten $\mathbf{p}_0 = (x_0, y_0)$ und $\mathbf{p}_1 = (x_1, y_1)$

- Steigung der Gerade durch die Strecke ist $m = \Delta y / \Delta x = y_1 - y_0 / x_1 - x_0$
- Für $m \in [0, 1]$ und $x_0 < x_1$ erhält man mit der Darstellung $y = mx + t$

```
for x in range(x0, x1+1):
    y = m*x + t
    setzePixel(x, round(y))
```

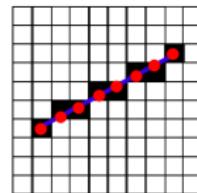
- Nachteile: Pro Durchgang
 - 1 Multiplikation
 - 2 Runden

- Beheben von Nachteil 1

- Addition statt Multiplikation (Digital Differential Analysis (DDA)) durch Betrachtung des Schritts von Berechnung von x_i zu $x_{i+1} = x_i + dx$

$$(x_{i+1}, y_{i+1}) = (x_i + dx, m \cdot (x_i + dx) + t) = (x_i + dx, y_i + m \cdot dx)$$

```
y = y0
for x in range(x0, x1+1): # dx = 1
    y += m
    setzePixel(x, round(y))
```

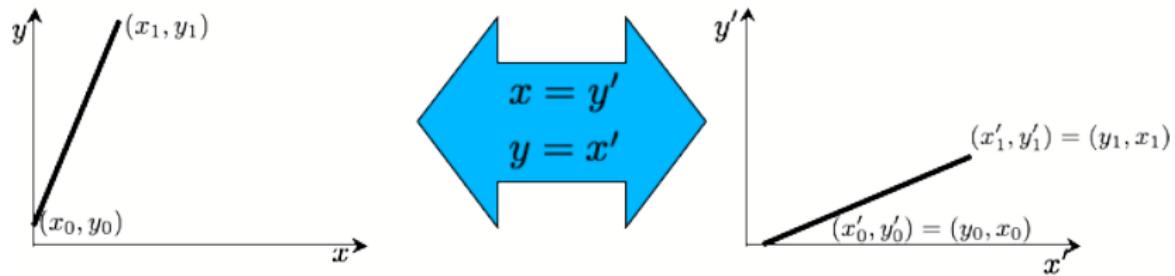


- Immer noch eine Rundungsoperation pro Durchgang :-)

Mittelpunkt–Algorithmus für Geraden (Bresenham, 1965)

Gerade durch zwei Punkte $\mathbf{p}_0 = (x_0, y_0), \mathbf{p}_1 = (x_1, y_1)$

O.B.d.A. gelte $x_0 \leq x_1$ sowie $m = \frac{y_1 - y_0}{x_1 - x_0} \in [0, 1]$



$$\text{Implizite Form } f(x, y) = \underbrace{(y_1 - y_0)}_a x + \underbrace{(x_0 - x_1)}_b y + \underbrace{(x_1 y_0 - x_0 y_1)}_c$$

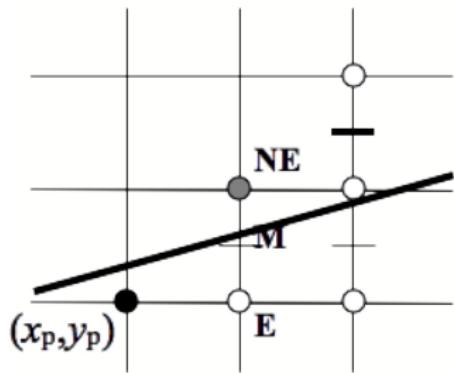
- *Positiv* für Punkte (x, y) unterhalb der Geraden
- *Negativ* für Punkte (x, y) oberhalb der Geraden

Mittelpunkt-Algorithmus (M liegt unter der Geraden)

Schritt von bekanntem (x_p, y_p) zum nächsten Punkt

Entscheidung, ob als nächster Punkt E oder NE gesetzt wird, abhängig von

$$\begin{aligned} d &= f(M) = f(x_p + 1, y_p + \frac{1}{2}) \\ &= a(x_p + 1) + b(y_p + \frac{1}{2}) + c \\ &= f(x_p, y_p) + a + \frac{b}{2} \\ &= a + \frac{b}{2} \end{aligned}$$



- Ist $d > 0$, wird NE gewählt und für das neue d_{new} gilt

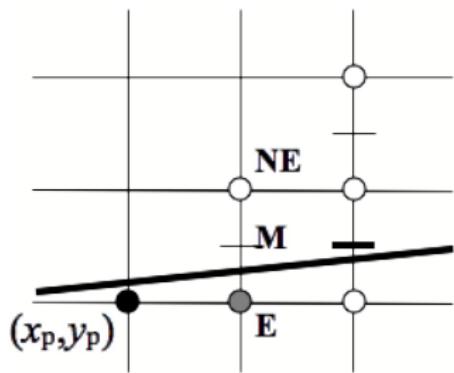
$$\begin{aligned} d_{new} &= f(x_p + 2, y_p + \frac{3}{2}) = a(x_p + 2) + b(y_p + \frac{3}{2}) + c \\ &= f(x_p + 1, y_p + \frac{1}{2}) + a + b = d + a + b \end{aligned}$$

Mittelpunkt-Algorithmus (M liegt über der Geraden)

Schritt von bekanntem (x_p, y_p) zum nächsten Punkt

Entscheidung, ob als nächster Punkt E oder NE gesetzt wird, abhängig von

$$\begin{aligned} d &= f(M) = f(x_p + 1, y_p + \frac{1}{2}) \\ &= a(x_p + 1) + b(y_p + \frac{1}{2}) + c \\ &= f(x_p, y_p) + a + \frac{b}{2} \\ &= a + \frac{b}{2} \end{aligned}$$



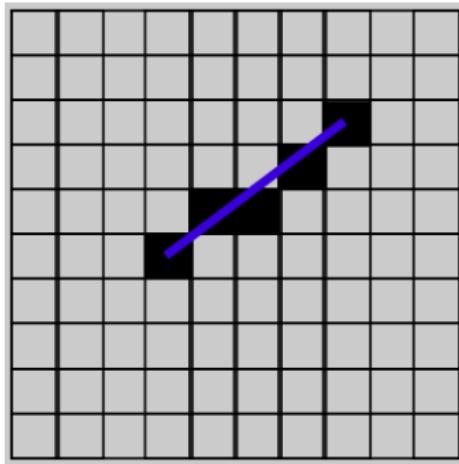
- Ist $d < 0$, wird E gewählt und für das neue d_{new} gilt

$$\begin{aligned} d_{new} &= f(x_p + 2, y_p + \frac{1}{2}) = a(x_p + 2) + b(y_p + \frac{1}{2}) + c \\ &= f(x_p + 1, y_p + \frac{1}{2}) + a = d + a \end{aligned}$$

Mittelpunkt–Algorithmus für Geraden (in Python)

Ersetzen von $f(x, y)$ durch $2f(x, y)$ ändert die Gerade nicht, es tauchen aber nur noch ganze Zahlen auf: $d = 2a + b$, $d_{new} = d + 2(a + b)$, $d_{new} = d + 2a$

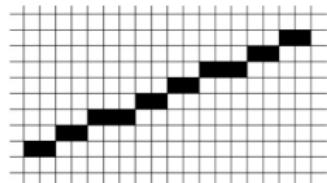
```
def zeichneLinie(x0, y0, x1, y1):
    a, b = y1 - y0, x0 - x1
    d = 2*a + b
    IncE = 2*a
    IncNE = 2*(a + b)
    y = y0
    for x in range(x0, x1+1):
        setzePixel(x, y)
        if d<=0: d += IncE
        else:
            d += IncNE
            y += 1
```



Aliaseffekte und Antialiasing

Die räumliche Auflösung eines (Raster-)Ausgabegerätes ist durch Rastergröße / Speicherkapazität beschränkt

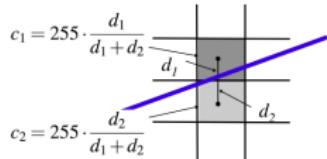
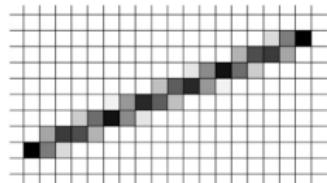
Es treten Aliaseffekte
(Treppeneffekte) auf



Einfaches Verfahren zum Glätten (Antialiasing) gerasterter Linien

- Setze für jeden x -Wert (Spalten) zwei übereinander liegende Pixel $\mathbf{p}_1, \mathbf{p}_2$
- Helligkeit c_1, c_2 der Pixel abhängig von (vertikaler) Entfernung von der Geraden

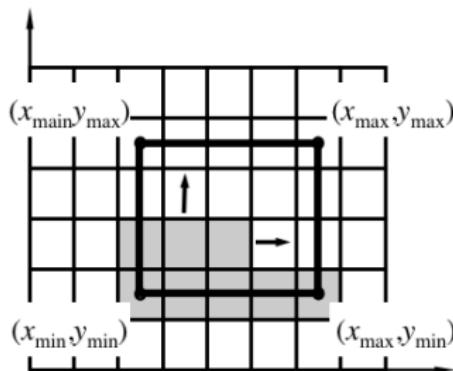
$$c_1 = 255 \cdot \frac{d_1}{d_1 + d_2}, \quad c_2 = 255 \cdot \frac{d_2}{d_1 + d_2}$$



- Gesamthelligkeit pro Spalte ist gleich

Rasterisieren achsparalleler Rechtecke

Füllen des achsparallelen Rechtecks mit Ecken $(x_{\min}, y_{\min}), (x_{\max}, y_{\max})$



```
for y in range(ymin, ymax+1)
    for x in range(xmin, xmax+1):
        setzePixel(x, y, farbwert)
```

- Problem
 - Doppeltes Kantenzeichnen bei benachbarten Rechtecken
- Lösung durch Verwendung halboffener Intervalle
 - Zeichne immer die *untere* und die *linke* Kante (nicht die obere und rechte)

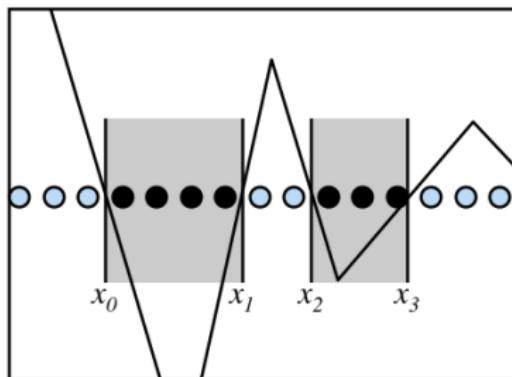
```
for y in range(ymin, ymax)
    for x in range(xmin, xmax):
        setzePixel(x, y, farbwert)
```

Rasterisieren beliebiger Polygone

Füllen eines beliebigen Polygons mit Eckpunkten $\mathbf{p}_0, \dots, \mathbf{p}_n$ mittels *Scan-line Algorithmus* (schematisch)

Traversiere alle scan-lines y (horizontale Pixel-Linien), die das Polygon schneiden, d.h. für die gilt $\min_{i \in \{0, \dots, n\}} \{(\mathbf{p}_i)_y\} \leq y \leq \max_{i \in \{0, \dots, n\}} \{(\mathbf{p}_i)_y\}$

- 1 Finde alle Schnittpunkte der scan-line mit den Kanten des Polygons
- 2 Sortiere die Schnittpunkte x_0, x_1, \dots nach wachsender x-Koordinate
- 3 Setze alle Pixel zwischen den Schnittpunkten mit gerader und ungerader Nummer



Genaue Beschreibung mit *Beispiel-Implementierung* findet sich z.B. in Foley et al., *Computer Graphics: Principles and Practice*

Spezialfälle des Scanline Algorithmus

1. Gemeinsame Kanten benachbarter Polygone?

- Teilintervalle sind immer *rechtseitig halboffen*, d.h.
 - Nach Schnittpunktberechnung Runden zum kleineren Integerwert (floor)
 - Ganzzahlige rechte Schnittpunkte werden nicht berücksichtigt

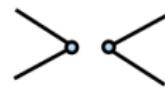
2. Wie oft wird der Schnittpunkt mit einem Eckpunkt gezählt?

- Polygonkanten sind *oben halboffen*
 - Schnittpunkt mit Ecke zählt nur, wenn Ecke *unterer Punkt einer Kante*
- Für die Anzahl der Schnittpunkte mit einer Ecke gibt es die Fälle

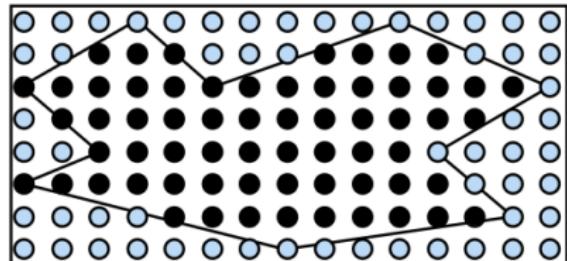
1. 0 Schnittpunkte



2. 1 Schnittpunkt



3. 2 Schnittpunkte

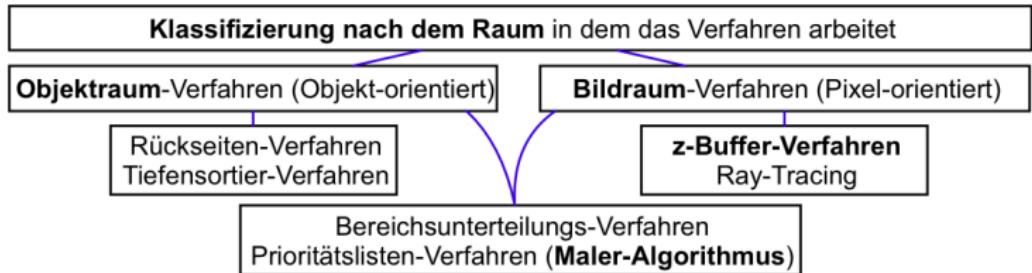


Sichtbarkeit

Projektion von 3D Szenen auf 2D Geräte kann unterschiedliche Objekte auf dieselbe Stelle abbilden



Verfahren zur Sichtbarkeitsbestimmung



Der Maler-Algorithmus (Prioritätslisten-Algorithmus)

- 1 Sortiere Objekte nach abnehmendem Abstand vom Betrachter
- 2 Gib die sortierten Objekte (*von hinten nach vorne*) aus

Spezialfälle (erfordern Unterteilung einzelner Objekte)

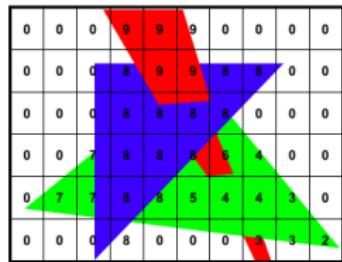
- Durchdringung, Überlappung (zyklische und wechselnde)



z-Buffer Algorithmus (Tiefenspeicher-Verfahren)

Der z-Buffer Algorithmus ist ein *Bildraum-Verfahren*

- Die Sichtbarkeits-Bestimmung erfolgt mittels *zusätzlichem Speicher* (z-Buffer), welcher das *Höhenprofil der Szene* (in der Regel) aus Sicht der Kamera enthält, d.h.
 - Alle Sichttransformationen sind durchgeführt
 - Szene ist in das Standardsichtvolumen der Parallelprojektion überführt
 - Kamera blickt in (negative) z-Richtung
- Der z-Buffer hat in der Regel
 - die Auflösung des Framebuffers (Bildschirmspeichers)
 - 8 oder 32 Bit pro Pixel zur Speicherung des Tiefenwertes
- Der z-Buffer Algorithmus arbeitet scan-line orientiert, d.h.
 - die Pixel des Bildschirmspeichers werden zeilenweise bearbeitet



Der z-Buffer Algorithmus

1 Initialisiere den Bildschirmspeicher mit der Hintergrundfarbe

2 Initialisiere den z-Buffer mit minimalem z-Wert

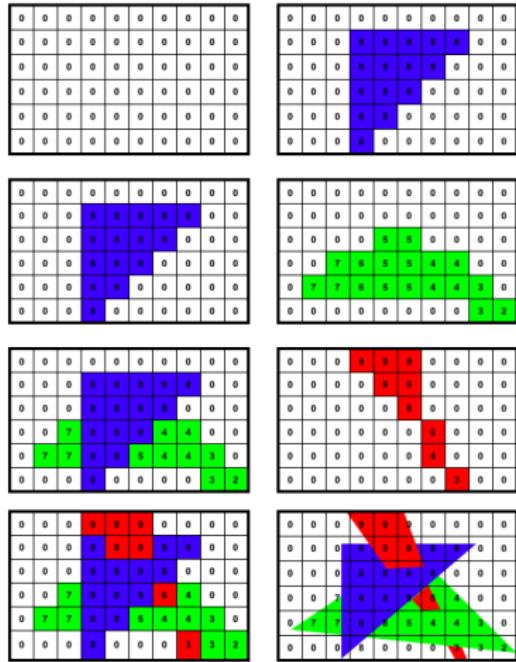
- i.A. z-Wert der Back-Clipping-Plane
- z-Wert der Front-Clipping-Plane = Größter zu speichernder z-Wert

3 Rasterzeilenkonvertierung pro Polygon

1 Berechne den z-Wert $z(x, y)$ für jedes Pixel (x, y) der Projektion des Polygons

2 Ist $z(x, y)$ größer als z-Buffer an der Stelle (x, y) , liegt das Pixel (x, y) des aktuellen Polygons vor allen bisherigen Polygone

- Trage die Farbe des Polygons im Bildspeicher an der Stelle (x, y) ein
- Setze den z-Buffer bei Adresse (x, y) auf den Wert $z(x, y)$



z-Buffer und Rasterisieren planarer Polygone (Dreiecke)

Beim Rasterisieren planarer Polygone müssen z-Werte nur an Schnittpunkten von Scan-Linie und linker Polygonkante berechnet werden

Alle weiteren Tiefen entlang der Scan-Linie im Polygon lasse sich mit Hilfe der zum Polygon gehörenden Ebenengleichung inkrementell berechnen:

Aus der Ebenengleichung $ax+by+cz+d = 0$ folgt

$$z(x, y) = -\frac{1}{c}(a \cdot x + b \cdot y + d)$$

und damit

$$z(x + d_x, y) = -\frac{1}{c}(a \cdot (x + d_x) + b \cdot y + d)$$

$$= z(x, y) - d_x \cdot \frac{a}{c}$$

- Für die nächste Rasterposition entlang der Scan-Linie gilt $d_x = 1$
- a/c ist konstant

ist zur **Bestimmung des nächsten z-Werts** lediglich **die Subtraktion des aktuellen z-Werts mit der Konstanten a/c** notwendig

Beispiel

Wie sieht der z-Buffer für das Dreieck $\mathbf{a} = (3, 2, 3)^T$, $\mathbf{b} = (3, 6, 8)^T$, $\mathbf{c} = (9, 2, 15)^T$ (Ebenengleichung $8x + 5y - 4z - 22 = 0$) aus?

0	0	0	0	0	0	0	0	0	0
0	0	3						15	0
0	0							0	0
0	0						0	0	0
0	7					0	0	0	0
0	0	8	0	0	0	0	0	0	0



(Open Graphics Library)

- Spezifikation für plattform- und programmiersprachenunabhängige Programmierschnittstelle zur Entwicklung von 2D- und 3D-Computergrafik-Anwendungen
- OpenGL realisiert einen Zustandsautomaten
 - Elemente werden entsprechend aktuellem „*Rendering-State*“ dargestellt
- Der OpenGL-Standard
 - enthält etwa 250 Befehle, zur Echtzeit-Darstellung von 3D-Szenen,
 - beschränkt sich auf die Darstellung von *Punkten*, *Linien*, *Polygonen* und
 - ist (in einem vierstufigen Verfahren) erweiterbar durch „*Extensions*“
 - 1 Neue Funktionsnamen und Konstanten erhalten herstellerspezifisches Suffix (z.B. ATI für ATI, NV für NVIDIA)
 - 2 Einigen sich mehrere Hersteller auf die Funktionalität bekommen die Namen das Suffix EXT (für Extension)
 - 3 Einigt sich das **Architecture Review Board** darauf, eine Erweiterung zu standardisieren, erhalten alle Namen das Suffix ARB
 - 4 Die meisten ARB-Erweiterungen werden in der nächsten OpenGL-Spezifikation Bestandteil von OpenGL und erhalten kein Suffix mehr



OpenGL versus Direct3D

Vorteile von OpenGL

- Plattformunabhängigkeit (*Windows, Mac OS, X Window System, BeOS, Amiga, Symbian, Bada, Apple iOS, Android, ...*)
- Erweiterbar
- Client-Server-Modell
- In der Regel bessere Treiberunterstützung für professionelle Grafikhardware
- Draw Call-Kosten sind niedriger als in Direct3D, was zu einer besseren Performance führen kann (siehe auch www.nvidia.com/object/opengl-nvidia-extensions-gdc-2006.html)

Nachteile

- Trotz ARB teilweise „Extension-Chaos“

OpenGL Historie (I) ³

Version	Datum	Beschreibung
1.0	01.07.1992	<ul style="list-style-type: none"> ■ Erste Veröffentlichung
2.0	07.09.2004	<ul style="list-style-type: none"> ■ <i>Shaderprogramme: OpenGL Shading Language</i> ■ Multiple Render Targets ■ Texturen beliebiger Größe (auch nicht 2^n)
3.0	11.08.2008	<ul style="list-style-type: none"> ■ Codebasis aufgeräumt ■ OpenGL Shading Language 1.30 ■ Entfernung von Altlasten (<i>glBegin</i>, <i>glEnd</i>, <i>Fixed-Function-Pipeline</i>, <i>Transform & Lighting</i>, ...) ■ Architektur insgesamt (Schichtmodell) nähert sich an DirectX an ■ Erstmals weitgehender Verzicht auf explizite Abwärtskompatibilität
4.0	11.03.2010	<ul style="list-style-type: none"> ■ Einbindung von OpenCL ■ Tesselation ■ OpenGL Shading Language 4.00

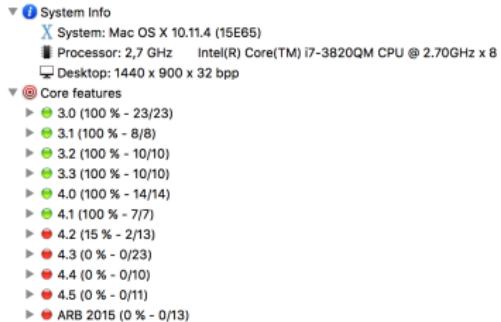
³www.wikipedia.org

OpenGL Historie (II)⁴

Version	Datum	Beschreibung
4.5	11.08.2014	<ul style="list-style-type: none"> ■ Direct State Access (DSA) und Flush Control ■ OpenGL ES 3.1 API und Shader Kompatibilität ■ DX11 Emulations Features
Vulkan	26.02.2016	<ul style="list-style-type: none"> ■ Vereinigung von OpenGL und OpenGL ES ■ Nicht Abwärtskompatibel zur OpenGL

Aktueller Stand

- Großer Teil vorhandener OpenGL Anwendungen < 3.0
- Apple hat Standard nur bis 4.1 vollständig umgesetzt



⁴www.wikipedia.org

OpenGL und ergänzende Bibliotheken

OpenGL Utility Library (GLU)

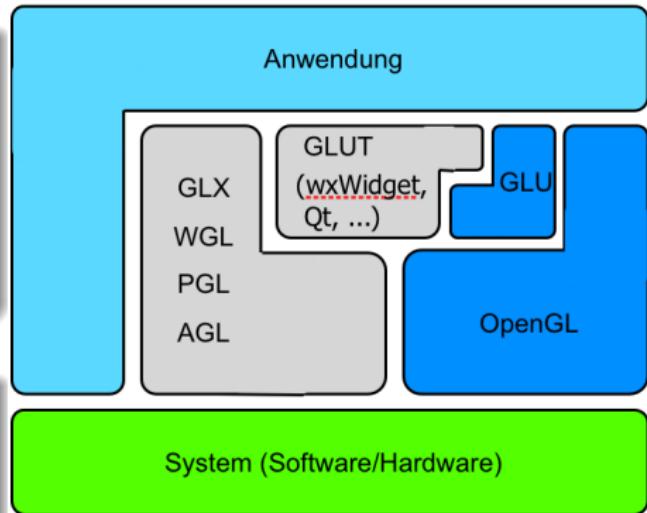
- Darstellung komplexer Objekte (Quadriken, NURBS, ...)
- Kamerahandling
- Fehlerbehandlung

Plattformabhängige Fenster-API

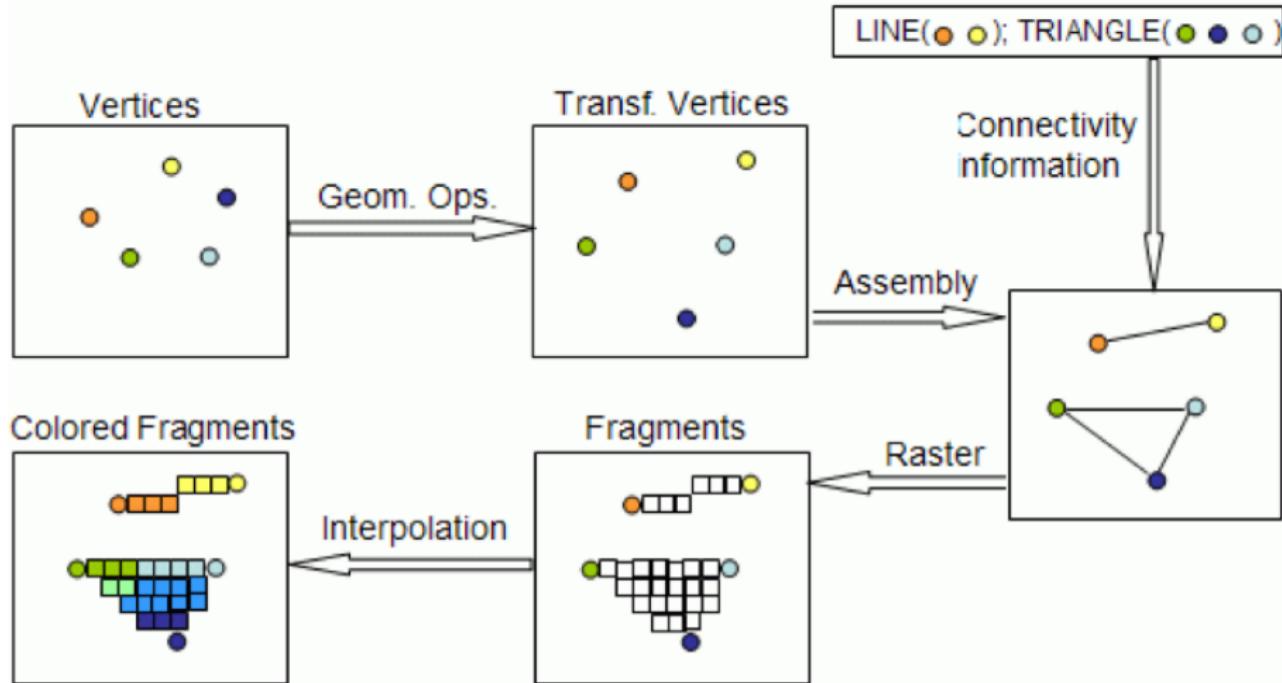
- GLX für X-Windows, WGL für Windows, AGL für Mac OS

OpenGL Utility Toolkit (GLUT)

- Einfaches plattformunabhängiges GUI Toolkit
- Unterstützung von Eingabegeräten (Maus, Tastatur, ...)
- Fertige Solid- und Wireframe-Objekte (Würfel, Kegel, Teapot, ...)



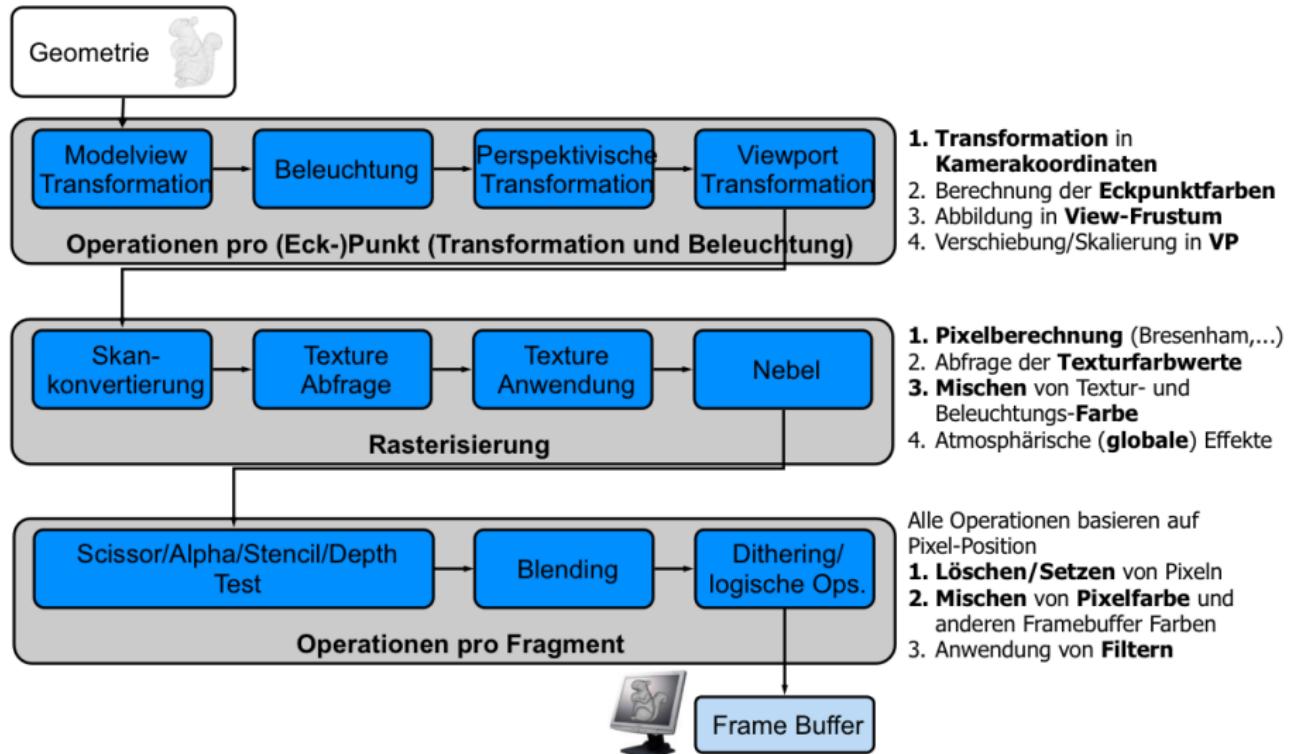
Erinnerung: Die Geometrie-Pipeline



Bildquelle: <http://www.lighthouse3d.com/opengl/glsl>



Die (vereinfachte) OpenGL Fixed-Function-Renderpipeline



(Deprecated in OpenGL 3.0. Nicht mehr im Kern seit OpenGL 3.1)



Die OpenGL Datentypen

OpenGL definiert (in Anlehnung an C) eigene Datentypen

OpenGL Typ	Datentyp	Suffix
GLbyte	8-bit integer	b
GLshort	16-bit integer	s
GLint	32-bit integer	i
GLfloat, GLclampf	32-bit float	f
GLdouble, GLclampd	64-bit double	d
GLubyte, GLboolean	8-bit unsigned byte	ub
GLshort	16-bit unsigned short	us
GLuint, GLenum, GLbitfield	32-bit unsigned integer	ui

Die OpenGL Nomenklatur

OpenGL unterstützt kein Überladen von Funktionen

- Die Namen aller Funktionen von OpenGL und der Hilfsbibliotheken GLU und GLUT werden gebildet nach dem Schema

{gl, glu, glut}<funktionsname>[{d, f, u, ...}] [v]

- Das Präfix gibt an, ob es sich um eine OpenGL, GLU oder GLUT Funktion handelt
- Das Suffix gibt den Datentyp der Parameter der Funktion an
- Werden Parameter als Vektor erwartet, endet der Funktionsname mit einem v
- Konstanten werden groß geschrieben (Präfixe: GL_, GLU_, GLUT_)
- Beispiele für Namen von OpenGL Konstanten und Funktionen
 - GL_POINTS, GLU_FILL, GLUT_DOUBLE
 - glFrustum, glVertex3f, glMaterialfv, glMaterialfv
 - gluLookAt, gluPerspective
 - glutSolidTeapot, glutSwapBuffers

Die OpenGL Nomenklatur (Beispiel)

OpenGL-Funktion

glVertex3fv(...)

Anzahl der
Komponenten

2- (x,y)

3- (x,y,z)

4- (x,y,z,w)

Daten-Typ

b - byte

ub - unsigned byte

s - short

us - unsigned short

i - int

ui - unsigned int

f - float

d - double

Vektor

Bei der skalaren
Form des Befehls
wird das "v"
weggelassen



Modelview-/Projektions-Matrix

OpenGL speichert geometrische Transformationen in zwei verschiedenen Matrizen

- Modelview-Matrix M (in der Regel affine Transformationen)
 - Transformationen der Objekte (Translation, Rotation, ...)
 - Transformationen der Kamera (Translation, Rotation, ...)
- Projektions-Matrix P (in der Regel projektive Transformationen)
 - Art der projektiven Abbildung
 - Orthographische, perspektivische, ... Projektionen
- Nur die aktive Matrix wird verändert
 - Die aktive Matrix wird ausgewählt durch
`glMatrixMode(GL_MODELVIEW)`
`glMatrixMode(GL_PROJECTION)`
 - Alle nachfolgenden Matrix-Operationen beziehen sich auf die aktive Transformations-Matrix
- Alle Punkte werden mit $P \cdot M$ transformiert

Kamera-Parameter

Extrinsische Kamera-Parameter (Positionierung der Kamera durch)

- Manipulation der Modelview-Matrix oder
- komfortabler mittels GLU-Funktion (vergleiche Vorlesung 4)
`gluLookAt(ex, ey, ez, cx, cy, cz, ux, uy, uz)`

Intrinsische Kamera-Parameter (Festlegen des View-Frustums durch)

- Manipulation der Projektions-Matrix
- Orthographische Projektion (vergleiche Vorlesung 4)
`glOrtho(left, right, bottom, top, near, far)`
- Komfortabler mittels GLU-Funktion
`gluOrtho2D(left, right, bottom, top)`
- Perspektivische Projektion (vergleiche Vorlesung 4)
`glFrustum(left, right, bottom, top, near, far)`
- Komfortabler mittels GLU-Funktion
`gluPerspective(fovy, aspect, near, far)`

Ein OpenGL „Hello World“

```

from OpenGL.GL import *
from OpenGL.GLU import *

import sys, math

def initGL(width, height):
    # Set background color to blue
    glClearColor(0.0, 0.0, 1.0, 0.0)
    # Set orthographic projection
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    glOrtho(-1.5, 1.5, -1.5, 1.5, -1.0, 1.0)
    glMatrixMode(GL_MODELVIEW)

def display():
    # Clear framebuffer
    glClear(GL_COLOR_BUFFER_BIT)
    # Set color to light gray
    glColor3f(.75, .75, .75)
    # Set drawstyle
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)
    # Set primitive type to polygon
    glBegin(GL_POLYGON)
    for i in range(6):
        glVertex2f(math.cos(i*math.pi/3), \
                  math.sin(i*math.pi/3))
    glEnd()
    # Flush commands
    glFlush()

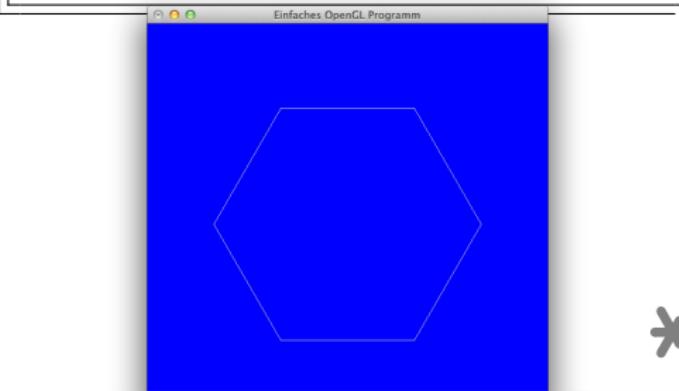
```

```

def main():
    # Initialize GLUT
    glutInit(sys.argv)
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(500, 500)
    glutCreateWindow("Einfaches_OpenGL_Programm")
    # Register display callback function
    glutDisplayFunc(display)
    # Initialize OpenGL context
    initGL(500, 500)
    # Start GLUT mainloop
    glutMainLoop()

if __name__ == "__main__":
    main()

```



Direct Mode nicht mehr im OpenGL-Kern

Bei

```
...
glBegin(GL_POLYGON)
for i in range(6):
    glVertex2f(math.cos(i*math.pi/3), math.sin(i*math.pi/3))
glEnd()
...
```

wird jeder Punkt (`glVertex2f`) einzeln zur Grafikkarte geschickt. Das ist

- aufwändig und langsam und
- seit OpenGL 3.0 nicht mehr im OpenGL Kern.

Mittels Vertex Buffer Objekts (VBOs) kann Speicher auf der Grafikkarte reserviert werden, in den dann (Punkt-)Daten kopiert werden können:

```
points = [[math.cos(i*math.pi/3), math.sin(i*math.pi/3)] for i in range(6)]
vbo = vbo.VBO(array(points, 'f')) # copy data

def display():
    ...
    vbo.bind()
    glVertexPointerf(vbo)
    glEnableClientState(GL_VERTEX_ARRAY)
    glDrawArrays(GL_POLYGON, 0, 6)
    vbo.unbind()
    glDisableClientState(GL_VERTEX_ARRAY)
    ...
```



Ein OpenGL „Hello World“ mittels VBOs

```

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.arrays import vbo
from numpy import array
import sys, math

points = [[math.cos(i*math.pi/3), \
           math.sin(i*math.pi/3)] \
           for i in range(6)]
vbo = vbo.VBO(array(points, 'f'))

def initGL(width, height):
    glClearColor(0.0, 0.0, 1.0, 0.0)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    glOrtho(-1.5, 1.5, -1.5, 1.5, -1.0, 1.0)
    glMatrixMode(GL_MODELVIEW)

def display():
    glClear(GL_COLOR_BUFFER_BIT)
    glColor3f(.75, .75, .75)
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)
    # Render Vertex Buffer Object
    vbo.bind()
    glVertexPointerf(vbo)
    glEnableClientState(GL_VERTEX_ARRAY)
    glDrawArrays(GL_POLYGON, 0, 6)
    vbo.unbind()
    glDisableClientState(GL_VERTEX_ARRAY)
    glFlush() # Flush commands

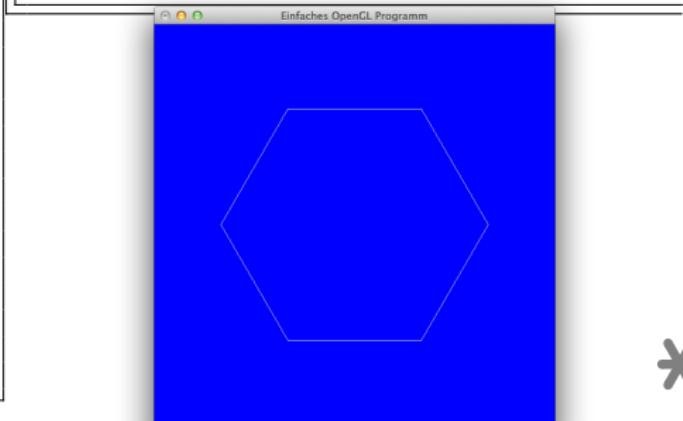
```

```

def main():
    # Initialize GLUT
    glutInit(sys.argv)
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(500, 500)
    glutCreateWindow("Einfaches_OpenGL_Programm")
    # Register display callback function
    glutDisplayFunc(display)
    # Initialize OpenGL context
    initGL(500, 500)
    # Start GLUT mainloop
    glutMainLoop()

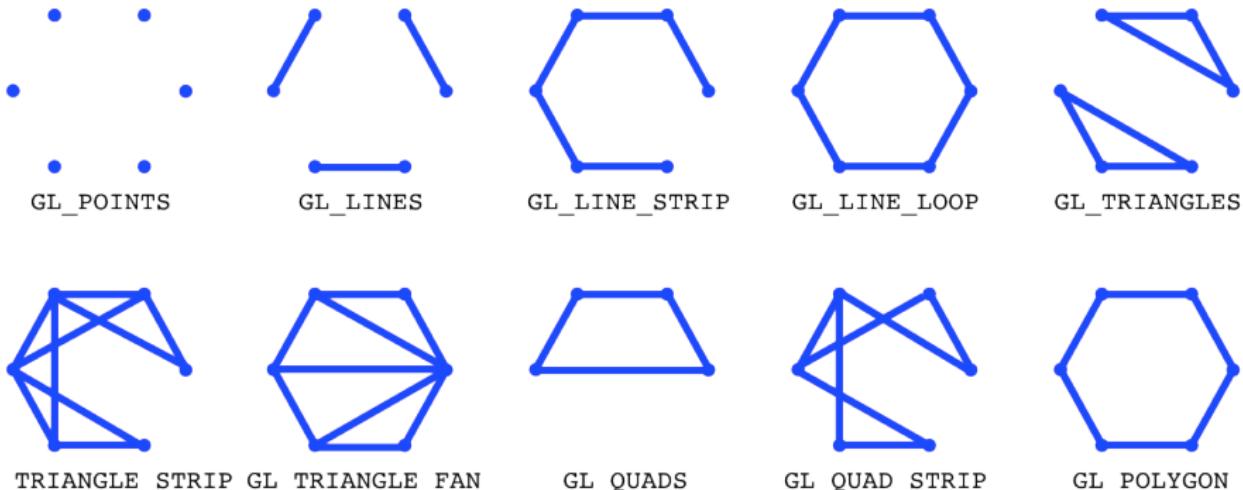
if __name__ == "__main__":
    main()

```



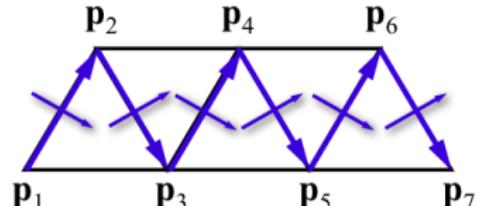
Die OpenGL Primitive

```
points = [[math.cos(i*math.pi/3), math.sin(i*math.pi/3)] for i in range(6)]
vbo = vbo.VBO(array(points, 'f'))
...
def display():
    ...
    glDrawArrays(GL_POINTS, 0, 6)
```

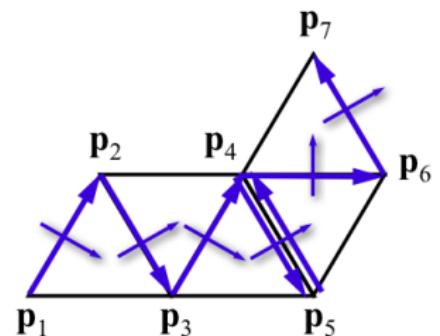


(Verallgemeinerte) Triangle Strips

- Zur Darstellung einer „zusammenhängenden“ Dreiecksmenge müssen die gemeinsamen Dreiecksseiten abwechselnd „links und rechts“ liegen
- Diese Einschränkung ist behebbar durch *verallgemeinerte Triangle Strips*
 - Neue Punkte können mit rechts und links Drehung verbunden sein
- Verallgemeinerte Triangle Strips können durch „Kanten-Swaps“ (umdrehen der Kantenrichtung) beschrieben werden
- In OpenGL müssen Kanten-Swaps durch Verdopplung des gemeinsamen Eckpunkts der Kanten mit gleicher Richtung dargestellt werden



Triangle Strip = (1,2,3,4,5,6,7)



Triangle Strip = (1,2,3,4,5,4,6,7)

Der Kanten-Swap wird durch das **degenerierte Dreieck** $\Delta_{4,5,4}$ simuliert

Die GLUT Primitive

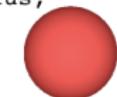
GLUT enthält 18 Funktionen zum Erzeugen einfacher 3D Objekte

Die Namen der Funktionen werden nach dem folgenden Schema gebildet:

`glut{Wire|Solid}{Sphere, Cube, ...}(<parameter>)`

Überblick über alle GLUT-Primitive: `glut{Wire|Solid}`

`Sphere(GLdouble radius,
GLint slices,
GLint stacks)`



`Cone(GLdouble base,
GLdouble height,
GLint slices,
GLint stacks)`



`Cube(GLdouble size)`



`Octahedron(void)`



`Tetrahedron(void)`



`Icosahedron(void)`



`Dodecahedron(void)`



`Torus(GLdouble innerRadius,
GLdouble outerRadius,
GLint nsides,
GLint rings)`



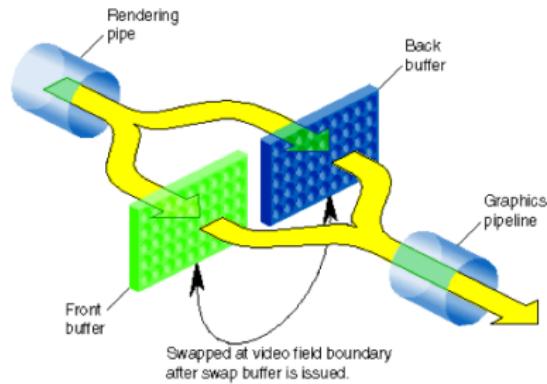
`Teapot(GLdouble size)`



Double Buffering (Bewegte Bilder)

Simulation einer Filmkamera (zeige nur fertige Bilder) durch Double-Buffering vermeidet Flackern

- Realisierung durch zwei Bildschirmspeicher: Front-Buffer und Back-Buffer
 - 1 Zeige den Inhalt des Front -Buffers an
 - 2 Rendere das nächste Bild in den Back-Buffer
 - 3 Vertausche Front- und Back-Buffer
 - 4 Gehe zu 1.
- Realisierung des Double-Buffering hängt von Hardware und GUI ab
- OpenGL selbst hat keine Funktion zum Vertauschen von Front- und Back-Buffer
- GLUT stellt die Funktion `glutSwapBuffers()` zur Verfügung



Beispiel

```

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *

import sys, math

def initGL(width, height):
    # Set background color to blue
    glClearColor(0.0, 0.0, 1.0, 0.0)
    # Enable clearing of the depth buffer
    glEnable(GL_DEPTH_TEST)
    # Set perspective transformation
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    gluPerspective(45., float(width)/height,\n                  0.1, 100.0)
    glMatrixMode(GL_MODELVIEW)

def display():
    # Clear framebuffer and depth buffer
    glClear(GL_COLOR_BUFFER_BIT | \
            GL_DEPTH_BUFFER_BIT)
    # Reset the modelview matrix
    glLoadIdentity()
    # Move down the z-axis
    glTranslate(0.0 ,0.0, -4.0)
    # Set glutWireTeapot
    glutWireTeapot(1.0)
    # Swap double buffer buffers
    glutSwapBuffers()

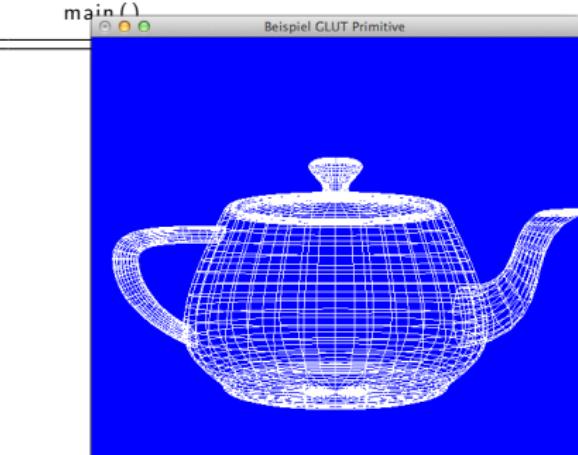
```

```

def main():
    glutInit(sys.argv)
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | \
                        GLUT_DEPTH)
    glutInitWindowSize(500, 500)
    glutCreateWindow("Beispiel_GLUT_Primitive")
    # Register display callback function
    glutDisplayFunc(display)
    # Initialize OpenGL context
    initGL(500, 500)
    # Start GLUT mainloop
    glutMainLoop()

if __name__ == "__main__":
    main()

```



GLUT Callback-Funktionen

GLUT stellt verschiedene Callback-Funktionen zur Verfügung. z.B.

- `glutDisplayFunc(void (*func)(void))`
- `glutReshapeFunc(void (*func)(int width, int height))`
- `glutKeyboardFunc(void (*func)(int width, int height))`
- `glutIdleFunc(void (*func)(void))`
 - Die Idle-Callback Funktion hat im Allgemeinen die Gestalt
 - ```
def idle():
 ...
 glutPostRedisplay() # display Callback wird aufgerufen
```
- `glutTimerFunc(unsigned int msec, void (*func)(int value), value)`
  - Die Timer-Callback Funktion hat im Allgemeinen die Gestalt
  - ```
def timer(vis):  
    ...  
    glutTimerFunc(msec, timer, value)
```
- `glutVisibilityFunc(void (*func)(int state))`
 - Die Visible-Callback Funktion hat im Allgemeinen die Gestalt
 - ```
def visible(vis):
 if vis == GLUT_VISIBLE: glutIdleFunc(idle)
 else: glutIdleFunc(None)
```

# GLUT Callback Beispiel

```

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *

import sys, math

animate = False
angle = 0

def initGL(width, height):
 # Wie auf Folie 24

def display():
 # Clear framebuffer and depth buffer
 glClear(GL_COLOR_BUFFER_BIT | \
 GL_DEPTH_BUFFER_BIT)
 # Reset the modelview matrix
 glLoadIdentity()
 # Move down the z-axis
 glTranslate(0.0, 0.0, -4.0)
 # Rotate around y-Axis
 glRotate(angle, 0.0, 1.0, 0.0)
 # Set glutWireTeapot
 glutWireTeapot(1.0)
 # Swap double buffer buffers
 glutSwapBuffers()

...

```

```

...
def keyPressed(key, x, y):
 global animate
 if key == "a":
 animate ^= True
 glutPostRedisplay()

def animation():
 global angle
 if animate:
 angle = (angle+1)%360
 glutPostRedisplay()

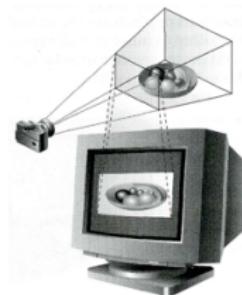
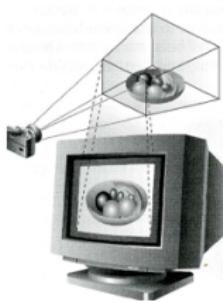
def main():
 # Initialize GLUT
 ...
 # Register display callback function
 glutDisplayFunc(display)
 # Register reshape callback function
 glutReshapeFunc(resizeViewport)
 # Register keyboard callback function
 glutKeyboardFunc(keyPressed)
 # Register Idle Callback
 glutIdleFunc(animation)
 # Initialize OpenGL Context
 initGL(500, 500)
 # Start GLUT mainloop
 glutMainLoop()
...

```

# Der Viewport

## Die Größe des Viewports

- entspricht in der Regel der Fenstergröße beim Programmstart
- kann geändert werden mittels `glViewport(x, y, width, height)`
- sollte an die intrinsischen Kameraparameter angepasst werden



```
gluPerspective(fovy, 1.0, near, far) gluPerspective(fovy, 1.0, near, far)
glViewport(0,0,400,400) glViewport(0,0,400,200)
```

- Keine Verzerrung, wenn `width/height = aspect`

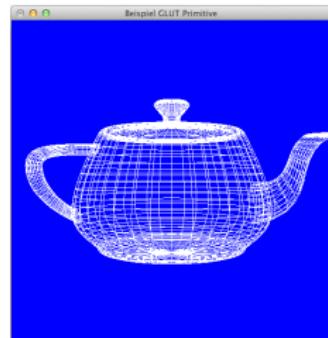
# Die reshape Funktion

Ändern der Viewportgröße erfordert  
Anpassung des Kameraseitenverhältnisses

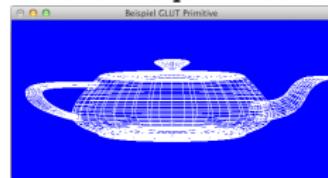
```
...
def resizeViewport(width, height):
 # Prevent division by zero
 if height == 0:
 height = 1
 # Reset current viewport
 glViewport(0, 0, width, height)
 # Reset perspective transformation
 glMatrixMode(GL_PROJECTION)
 glLoadIdentity()
 gluPerspective(45.0, float(width)/height,\n 0.1, 100.0)
 # Activate model view matrix
 glMatrixMode(GL_MODELVIEW)
 # Swap double buffer buffers
 glutSwapBuffers()

...
def main():
 ...
 # Register reshape callback function
 glutReshapeFunc(resizeViewport)
 ...

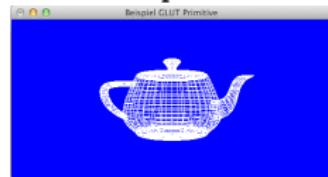
```



Ohne reshape Funktion



Mit reshape Funktion

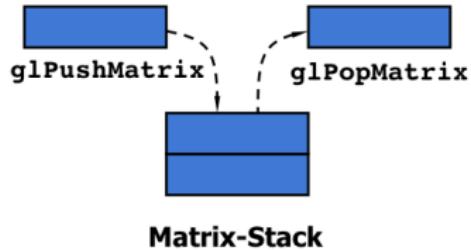


# Der Matrix Stack

OpenGL verwaltet (mindestens) 32 Modelview- und 2 Projektions-Matrizen auf jeweils einem Stack

- Der Matrix-Stack erleichtert die Verwaltung hierarchischer Szenen
- Aktiver Matrix-Stack wird mit Hilfe von `glMatrixMode()` bestimmt
- Oberste Matrix ist die aktuelle Modellview- bzw. Projektions-Matrix
- Zwei Methoden zur Manipulation der obersten Matrix auf dem Stack
  - `glPushMatrix()` (kopieren und auf den Stack legen)
    - Die beiden obersten Matrizen auf dem Stack sind danach gleich
  - `glPopMatrix()` (vom Stack holen)
    - Inhalt der Matrix geht verloren

```
def zeichneRadUndSchrauben():
 zeichneRad()
 for i in range(0,6):
 glPushMatrix()
 glRotatef(60.0*i, 0.0, 0.0, 1.0)
 glTranslatef(6.0, 0.0, 0.0)
 zeichneSchraube()
 glPopMatrix()
```



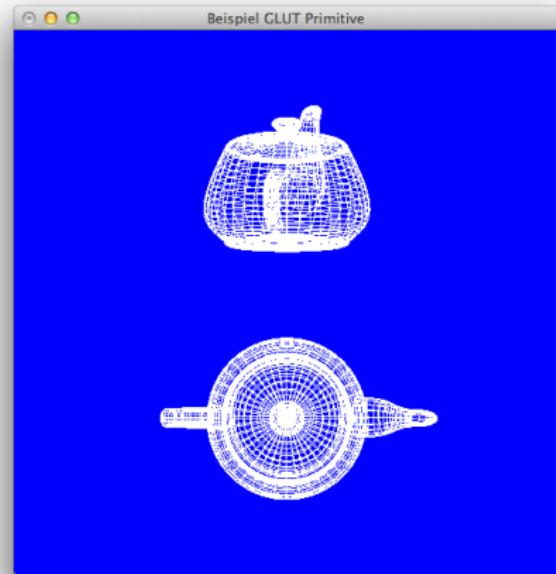
# Beispiel

```

...
Wie Folie 26 nur

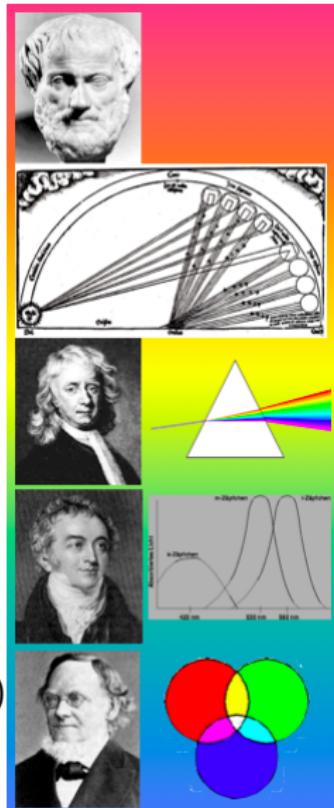
def display():
 # Clear framebuffer and depth buffer
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
 # Reset the modelview matrix
 glLoadIdentity()
 # Save Matrix
 glPushMatrix()
 # Move
 glTranslate(0.0 ,0.7, -4.0)
 # Rotate around y-Axis
 glRotate(angle, 0.0, 1.0, 0.0)
 # Set glutWireTeapot
 glutWireTeapot(.5)
 # Throw away matrix. CAUTION: You have to have as
 # many glPushMatrix() as glPopMatrix() to
 # prevent stack overflow or underflow
 glPopMatrix()
 # Move
 glTranslate(0.0 ,-0.7, -4.0)
 # Rotate around y-Axis
 glRotate(-angle, 1.0, 0.0, 0.0)
 # Set glutWireTeapot
 glutWireTeapot(.5)
 # Swap double buffer buffers
 glutSwapBuffers()
...

```



# Farbtheorien

- Aristoteles (384-322v.Chr.)
  - Experimente mit farbigen Gläsern
  - Farben sind ein Gemisch aus Schwarz und Weiß?!
- Dietrich von Freiberg (1250-1310)
  - Erklärt die Entstehung von primärem und sekundärem Regenbogen
- Sir Isaac Newton (1643-1727)
  - Experimente mit Prismen
  - Sonnenlicht ist in farbige Bänder zerlegbar
- Thomas Young (1773-1829)
  - Erste drei Farben Theorie
    - Netzhaut besteht aus drei verschiedenen Nervenelementen
    - Erweitert durch Hermann von Helmholtz (1821-1894)
- Hermann Günther Grassmann (1809-1877)
  - Regeln für das Mischen von Farben



# Farben als elektromagnetische Wellen

Farben = Elektromagnetische Wellen unterschiedlicher Länge

Sichtbare Wellenlängen zwischen ca. 380 nm und 780 nm

**Frequenz ( $f$ ):**

|        |        |     |        |        |     |        |        |     |           |           |           |           |           |           |           |           |           |           |           |           |           |           |           |
|--------|--------|-----|--------|--------|-----|--------|--------|-----|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| $10^1$ | $10^2$ | $1$ | $10^4$ | $10^5$ | $1$ | $10^7$ | $10^8$ | $1$ | $10^{10}$ | $10^{11}$ | $10^{12}$ | $10^{13}$ | $10^{14}$ | $10^{15}$ | $10^{16}$ | $10^{17}$ | $10^{18}$ | $10^{19}$ | $10^{20}$ | $10^{21}$ | $10^{22}$ | $10^{23}$ | $10^{24}$ |
| Hz     | Hz     | KHz | Hz     | MHz    | Hz  | Hz     | GHz    | Hz  | Hz        | Hz        | Hz        | Hz        | Hz        | Hz        | Hz        | Hz        | Hz        | Hz        | Hz        | Hz        | Hz        | Hz        | Hz        |

| Wechselstrom & Tonfrequenz | Rundfunk     | Funkwellen (DZM-meterwellen) | Infrarot                   | Ultra-violett              | Röntgenstrahlung           | Gammastrahlung             | Höhenstrahlung             |
|----------------------------|--------------|------------------------------|----------------------------|----------------------------|----------------------------|----------------------------|----------------------------|
| $10^4$<br>km               | $10^3$<br>km | $10^2$<br>km                 | $10$<br>km                 | $1$<br>km                  | $10^2$<br>m                | $10$<br>m                  | $1$<br>m                   |
| $10^1$<br>cm               | $10^0$<br>cm | $10^{-1}$<br>mm              | $10^{-2}$<br>$\mu\text{m}$ | $10^{-3}$<br>$\mu\text{m}$ | $10^{-4}$<br>$\mu\text{m}$ | $10^{-5}$<br>$\mu\text{m}$ | $10^{-6}$<br>$\mu\text{m}$ |
| $10^2$<br>nm               | $10^1$<br>nm | $10^{-1}$<br>nm              | $10^{-2}$<br>nm            | $10^{-3}$<br>nm            | $10^{-4}$<br>nm            | $10^{-5}$<br>nm            | $10^{-7}$<br>nm            |

**Wellenlänge ( $\lambda$ ):**

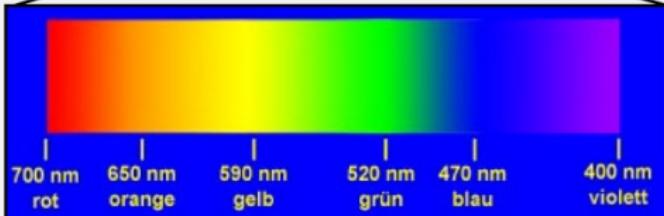
Erinnerung:

Es gilt  $f \cdot \lambda = c$

mit der Lichtgeschwindigkeit

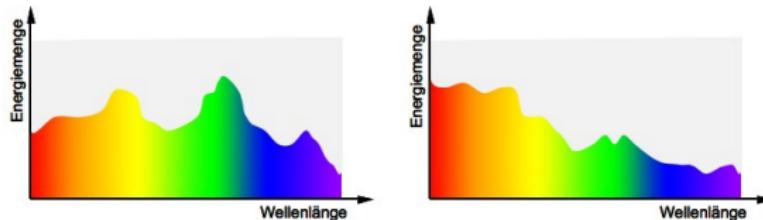
$$c = 299.792.458 \frac{\text{m}}{\text{s}}$$

Sichtbares Licht

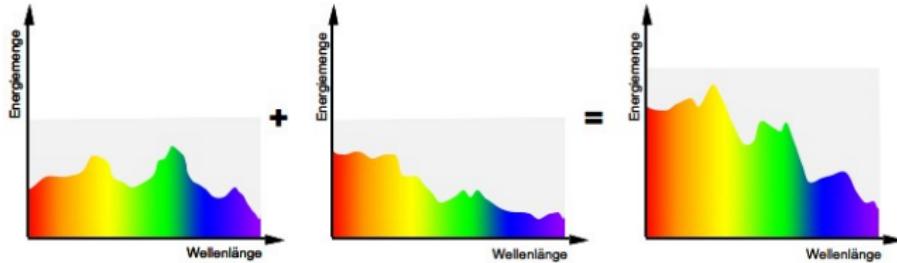


# Licht und additive Farbmischung

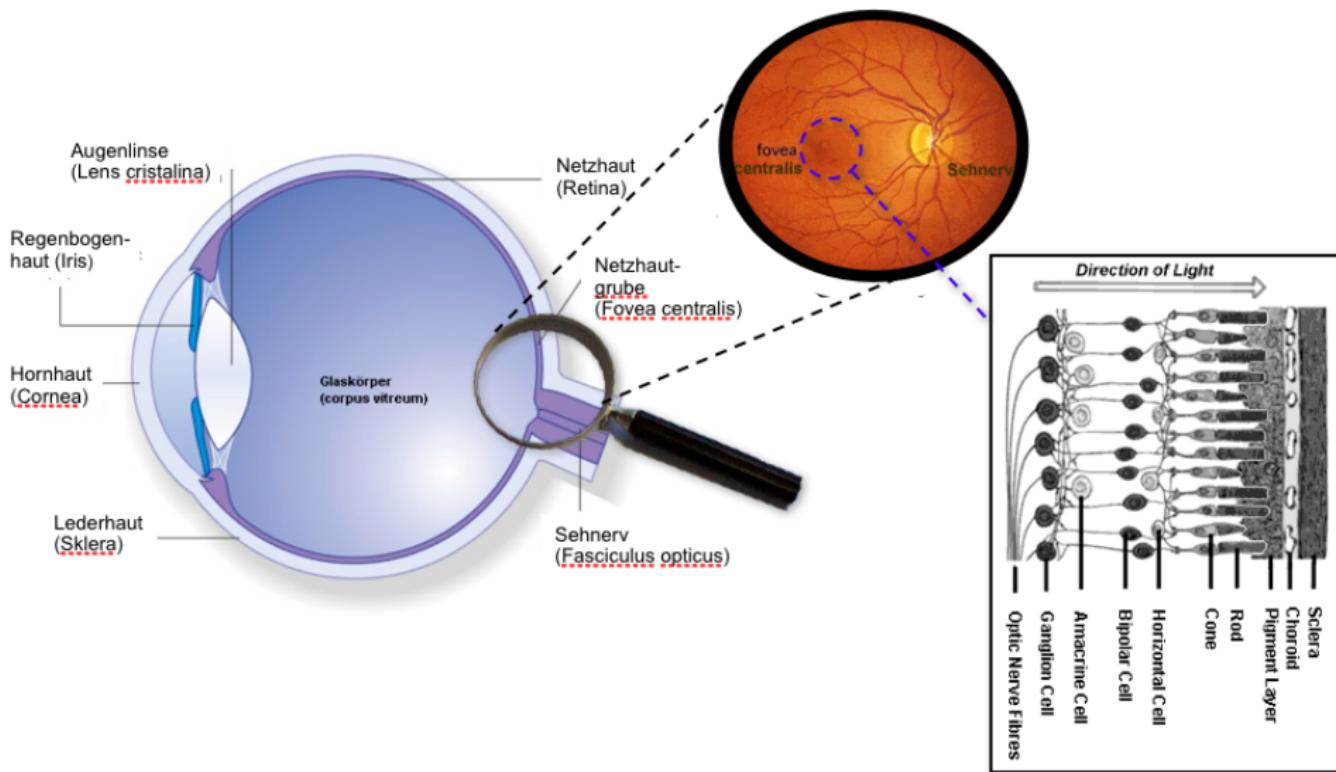
Verschiedene Lichtquellen (Tageslicht, Dämmerungslicht, Glühbirne, Neonröhre, LED, ...) haben unterschiedliche Energieverteilungen



Gesamtfarbe zweier Lichter ergibt sich durch Addition der Energiespektren



# Das Auge



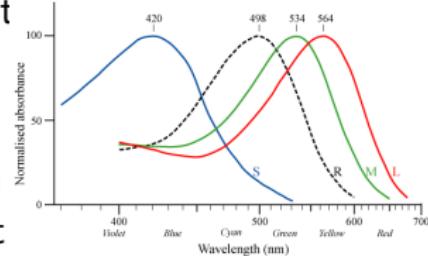
# Lichtempfindliche Nervenzellen auf der Netzhaut

Stäbchen (Rods) – ca. 120 Millionen pro Auge – Kein Farbsehen

- Große Lichtempfindlichkeit (Dämmerungssehen) durch Verschaltung als rezeptives Feld (ca. 1000 Stäbchen pro Ganglienzelle)
- Höchste Dichte um die Fovea herum, daher geringe Sehschärfe aber sehr empfindliche Bewegungswahrnehmung

Zapfen (Cones) – ca. 6 Millionen pro Auge – Farbsehen

- Drei unterschiedlich empfindliche Zelltypen mit Sehpigmenten für **Rot** (l-Zapfen), **Grün** (m-Zapfen), **Blau** (k-Zapfen)
- Geringe Lichtempfindlichkeit (Tagessehen), da pro Ganglienzelle nur ein Zapfen verschaltet ist
- Maximum der spektralen Hellempfindlichkeitskurve bei 555nm (**grün**)
- Höchste Dichte in der Fovea, daher große Sehschärfe

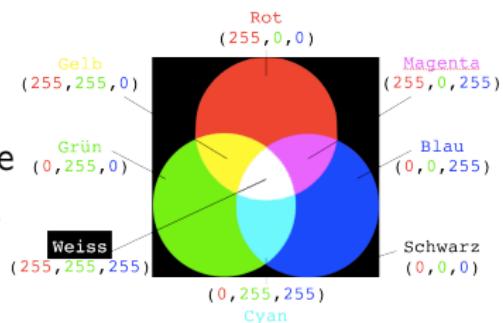


Bildquelle: [http://de.wikipedia.org/wiki/Zapfen\\_\(Auge\)](http://de.wikipedia.org/wiki/Zapfen_(Auge))

# Additive Farbmischung

Regeln für das Mischen von farbigem Licht (Grassamen 1853)

- Drei Grundfarben additiver Farbmischung
  - Rot, Grün, Blau
- Jeder Grundfarbe wird (maximale) Energie bzw. Intensität (z.B. 1 o. 255) zugeordnet
- Komplementärfarben
  - Cyan, Magenta, Gelb
- Farben werden als Linearkombination der drei Grundfarben dargestellt



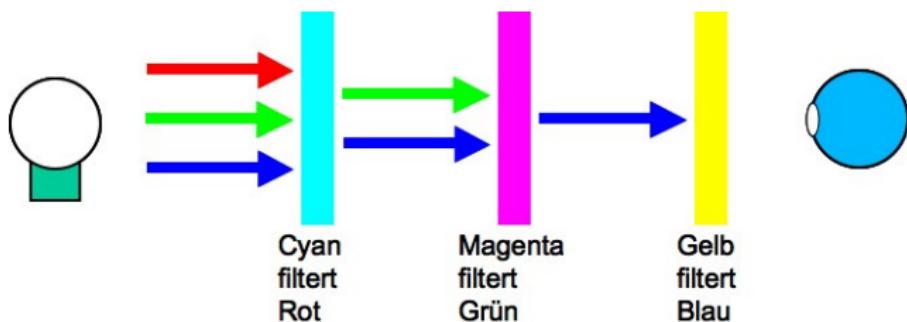
$$F = r \cdot R + g \cdot G + b \cdot B = (r, g, b)$$

- Addition zweier Farben  $F_1 = (r_1, g_1, b_1)$  und  $F_2 = (r_2, g_2, b_2)$ :

$$\begin{aligned} F_1 + F_2 &= (r_1 + r_2) \cdot R + (g_1 + g_2) \cdot G + (b_1 + b_2) \cdot B \\ &= (r_1 + r_2, g_1 + g_2, b_1 + b_2) \end{aligned}$$

# Subtraktive Farbmischung

Subtraktive Farbmischung: Farbfilter absorbieren Teile des Farbspektrums



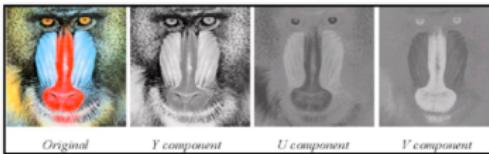
- Die Farben (Cyan, Magenta, Gelb) der Filter bezeichnet man als Primärfarben der subtraktiven Farbmischung
- Die dabei erzeugten (Komplementär-)Farben Rot, Grün, Blau als Sekundärfarben
- Subtraktive Farbmischung ist eine physikalische Farbmischung (unabhängig von der Farbwahrnehmung)



# Geräteabhängige Farbsysteme

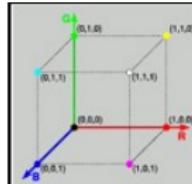
## ■ Bildschirmausgabe (additive Farbsysteme)

- RGB–System (Rot, Grün, Blau Anteil)
- YUV–System (Y: Helligkeit, U,V: Abweichungen zu Y)



$$\begin{aligned} Y &= R + G + B \\ U &= B - Y \\ V &= R - Y \end{aligned}$$

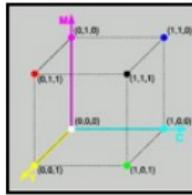
$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.30 & 0.59 & 0.11 \\ 0.44 & -0.15 & -0.29 \\ 0.36 & -0.26 & -0.51 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$



## ■ Druckausgabe (subtraktive Farbsysteme)

- CMY–System (Cyan, Magenta, Yellow)
  - Komplementärfarben zu Rot(=1-C), Grün(=1-M), Blau(=1-Y)
  - Mischen von CMY liefert in der Praxis kein schwarz sondern braun
- CMYK–System
  - Erweitert CMY (c,m,y) um schwarz (black)
  - Definition des CMYK–Quadrupels ( $c'$ ,  $m'$ ,  $y'$ ,  $k'$ ):

$$k' = \min\{c, m, y\}, \quad c' = c - k', \quad m' = m - k', \quad y' = y - k'$$



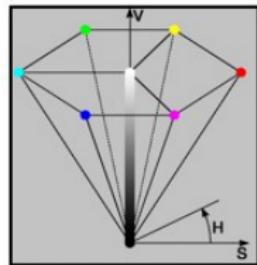
# Perzeptionsorientierte Farbsysteme (HSV)

- An menschliche *Farbwahrnehmung* angepasst

- Welche *Farbe*?
- Wie *intensiv* ist die Farbe?
- Wie *hell* oder *dunkel* ist die Farbe?

- *HSV-Farbmodell* (Hue / Saturation / Value)

- *Hue* (Farbton): Eigentliche Farbe
  - Als Winkel in  $[0^\circ, 360^\circ]$  gemessen.  $0^\circ$ =Rot,  $60^\circ$ =Gelb, ...
- *Saturation* (Sättigung): Weißanteil (Werte  $\in [0, 1]$ )
  - $< 1$ : Pastelltöne,  $1$ : Farbe liegt auf Pyramidenfläche
- *Value* (Helligkeit) (Werte  $\in [0, 1]$ )
  - $< 1$ : Dunkle Farben,  $0$ : Farbe schwarz
- Beispiele (\* können beliebige Werte sein)
  - Weiß:  $(*, 0, 1)$ , Schwarz:  $(*, *, 0)$ , Rot:  $(0, 1, 1)$ , Blau:  $(240, 1, 1)$
- Umrechnung zwischen RGB und HSV
  - Foley et. al., Computer Graphics, Principles and Practice



Zusammenhang RGB-HSV

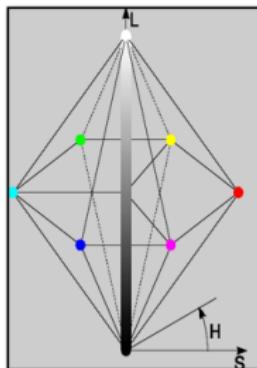
RGB-Würfel entlang der Diagonalen von Weiß nach Schwarz betrachtet



# Perzeptionsorientierte Farbsysteme (HLS)

## HLS-Farbmodell (Hue / Lightness / Saturation)

- Wird auch HSI (Intensity) genannt
- Strategie eines Malers
  - Nimm reines Pigment ( $H$ )
  - Mische Weiß dazu ( $S$ )
  - Mische Schwarz dazu ( $1-L$ )
- HLS-Doppelpyramide entsteht aus der HSV-Pyramide durch herausziehen der Pyramidengrundfläche bei Weiß
  - Hue (Farbton)  $\in [0^\circ, 360^\circ]$  wie im HSV Modell
  - Saturation (Farbsättigung/Grauanteil) wie bei HSV
  - Lightness entspricht in etwa dem V im HSV Modell
- Voll gesättigte Farben für  $L = 0.5$  und  $S = 1$
- Beispiele (\* können beliebige Werte sein)
  - Weiß= $(*, 1, *)$ , Schwarz= $(*, 0, *)$ , Rot= $(0, 0.5, 1)$ , Grün= $(120, 0.5, 1)$ , Blau= $(240, 0.5, 1)$



# Konvertierung zwischen RGB- und HLS-Farben

Konvertierung von HLS-Tripel (H,L,S) in RGB-Tripel (R,G,B)

Durch Rotation und Umrechnung in Zylinderkoordinaten erhält man:

Mit  $m_1 = \sqrt{\frac{2}{3}} \cdot S \cdot \cos(H)$ ,  $m_2 = \sqrt{\frac{2}{3}} \cdot S \cdot \sin(H)$  und  $l = \frac{4 \cdot L}{\sqrt{3}}$  gilt

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} \frac{2}{\sqrt{6}} & 0 & \frac{1}{\sqrt{3}} \\ -\frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{3}} \\ -\frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{3}} \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ l \end{pmatrix}$$

Konvertierung von RGB-Tripel (r,g,b) in HLS-Tripel (h,l,s)

Mit  $\begin{pmatrix} m_1 \\ m_2 \\ l \end{pmatrix} = \begin{pmatrix} \frac{2}{\sqrt{6}} & -\frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{6}} \\ 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$  erhält man

$$H = \arctan\left(\frac{m_2}{m_1}\right), S = \sqrt{\frac{3}{2}(m_1^2 + m_2^2)} \text{ und } L = \frac{l \cdot \sqrt{3}}{4}$$

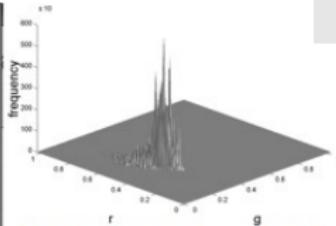
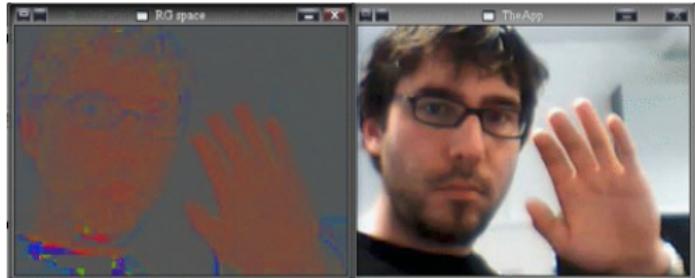
# Problemorientierte Farbsysteme

## Beispiel: Chromaticity-(rg-)Space

- Farbwertanteile  $r, g, b$  werden aus  $R, G, B$  berechnet als

$$r = \frac{R}{R + G + B}, g = \frac{G}{R + G + B}, b = \frac{B}{R + G + B} = 1 - r - g$$

- Normalisierung bezüglich Intensität
  - Robust gegenüber Beleuchtungsänderungen
- Anwendung: Klassifikation von Haut



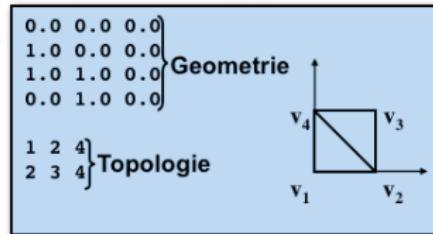
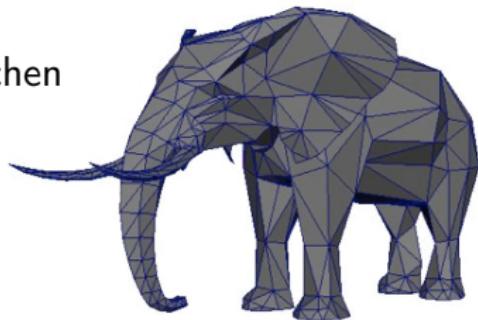
Quelle: Caetano et al., Performance Evaluation of Single and Multiple-Gaussian Models for Skin Color Modeling, 2002



# Polygonale (Netz-) Darstellung von Objekten

Objekte werden als eine Menge von Polygonen dargestellt

- Trennung von Geometrie (beschreibt die Position der Eckpunkte) und Topologie (beschreibt, welche Punkte zu einem Polygon gehören)
- Jedem Eckpunkt werden meist mehrere Attribute zugeordnet, die in unterschiedlichen Arrays gespeichert sind
  - Eckpunktkoordinaten (Vertex-Array)
  - Farben (Color-Array)
  - Normalenvektoren (Normal-Array)
  - ...
- Objekte werden in der Regel als Indexed-Face-Set gespeichert
  - Liste von Eckpunkten, Normalen, ...
  - Liste von Polygonen (enthält Listen von Indizes zu den Vertex-, Normal-, ... Arrays)



# OpenGL Beispiel (GL\_COLOR\_ARRAY)

```

from OpenGL.GL import *
from OpenGL.GLUT import *
from math import *
from OpenGL.arrays import vbo
from numpy import array
...
myVBO = None
...
def HLS2RGB(h, l, s):
 """_convert_HLS_to_RGB_color"""
 m1 = sqrt(2./3)*s*cos(h)
 m2 = sqrt(2./3)*s*sin(h)
 s2, s3, s6 = sqrt(2), sqrt(3), sqrt(6)
 i = 4*l/s3
 r = 2*m1/s6 + i/s3
 g = -m1/s6 + m2/s2 + i/s3
 b = -m1/s6 - m2/s2 + i/s3
 return (r,g,b)
...
def initGeometry():
 global myVBO
 cR = []
 for c in range(0,362, 2):
 r, g, b = HLS2RGB(pi*c/180, 0.5, 1)
 cR.append([-0.5, 1-c/180., r, g, b])
 cR.append([0.5, 1-c/180., r, g, b])
 cR.append([0.5, 1-(c+2)/180., r, g, b])
 cR.append([-0.5, 1-(c+2)/180., r, g, b])
 myVBO = vbo.VBO(array(cR, 'f'))
...

```

```

def display():
 glClear(GL_COLOR_BUFFER_BIT |\
 GL_DEPTH_BUFFER_BIT)

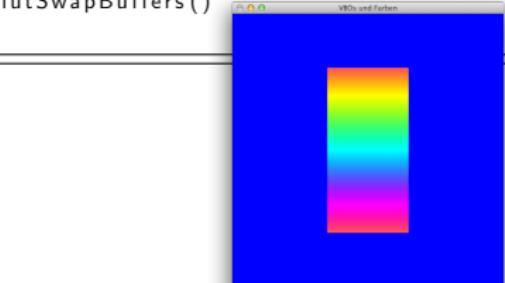
 myVBO.bind()
 glEnableClientState(GL_VERTEX_ARRAY)
 glEnableClientState(GL_COLOR_ARRAY)

 glVertexPointer(2, GL_FLOAT, 20, myVBO)
 glColorPointer(3, GL_FLOAT, 20, myVBO+8)

 glLoadIdentity()
 glDrawArrays(GL_QUADS, 0, len(cR))
 # or using an index list
 # glDrawElementsui(GL_QUADS, |
 # range(0,4*180+1))

 myVBO.unbind()
 glDisableClientState(GL_VERTEX_ARRAY)
 glDisableClientState(GL_COLOR_ARRAY)
 glutSwapBuffers()
...

```

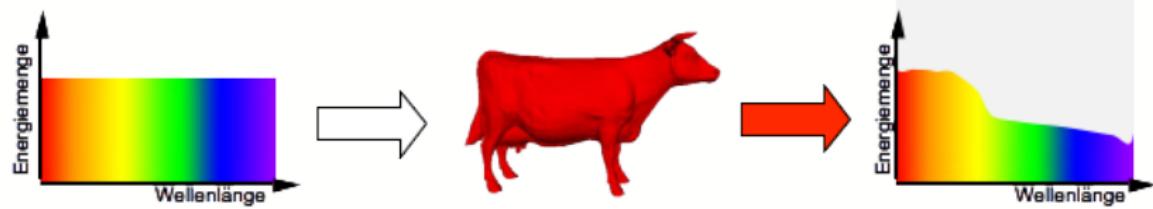


# Farbe eines Objekts

Farbe des von einem Objekt zurückgeworfenen Lichts ist abhängig von der Beleuchtung und den Oberflächeneigenschaften des Objekts

Pro Wellenlänge strahlt das Objekt einen bestimmten Anteil des einfallenden Lichts in der selben Wellenlänge in alle Richtungen zurück.

Beispiel: Rotes Objekt strahlt im roten Bereich mehr zurück als im Rest des Spektrums, also kommt rotes Licht zurück, wenn man weißes einstrahlt.



Oft wird ein Teil des Lichts unabhängig von der Wellenlänge aber abhängig von der lokalen Flächennormalen reflektiert (Einfallsinkel  $\approx$  Ausfallsinkel) und es entsteht ein heller Glanzfleck (abhängig von der Betrachterposition)



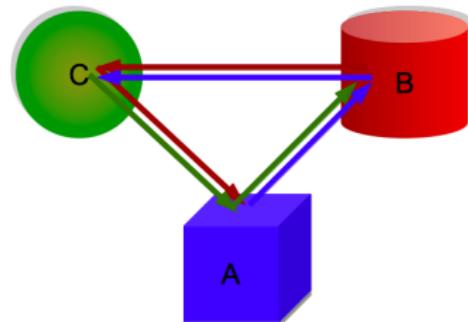
# Rendergleichung (Rendering Equation)<sup>5</sup>

Licht ist Energie und genügt damit dem Energieerhaltungssatz

- In geschlossenen Systemen bleibt die Gesamtenergie gleich
- Jede Fläche kann Licht absorbieren und abstrahlen
- Von Fläche abgestrahltes Licht besteht aus emittiertem Licht (Lichtquelle) und Reflexionen von anderen Flächen

Beispiel:

- Das von A nach B reflektierte Licht ...
  - ... kommt teilweise auch von C und wird an A reflektiert.
  - Das von C kommende Licht ...
  - ... kommt teilweise von B und ...
  - ... das Licht von B teilweise von A.
- 
- Die Rendergleichung beschreibt das obige Abhängigkeitssystem im allgemeinen Fall



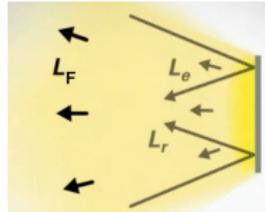
<sup>5</sup>James T. Kajiya: The rendering equation. SIGGRAPH 1986

# Rendergleichung (Rendering Equation)

Für das insgesamt von einer Fläche ausgestrahlte Licht  $L_F$  gilt

$$L_F = L_e + L_r \quad \text{mit}$$

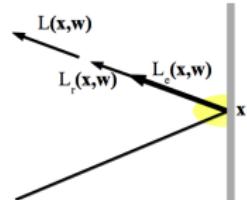
- dem von der Fläche selbst emittierten Licht  $L_e$  und
- dem an der Fläche reflektierten Licht  $L_r$



Für das von einem Punkt  $x$  in Richtung  $w$  ausgestrahlte Licht  $L(x, w)$  gilt

$$L(x, w) = L_e(x, w) + L_r(x, w) \quad \text{mit}$$

- von  $x$  in Richtung  $w$  selbst emittiertem Licht  $L_e(x, w)$
- an  $x$  in Richtung  $w$  reflektiertem Licht  $L_r(x, w)$



# Rendergleichung (Rendering Equation)

Für das reflektierte Licht  $L_r(\mathbf{x}, \mathbf{w})$  gilt

$$L_r(\mathbf{x}, \mathbf{w}) = \underbrace{\int_{\Omega_x}}_{\substack{\text{Integral über alle} \\ \text{Einfallslichtrichtungen} \\ (\text{Einheits-Hemisphäre um } \mathbf{x})}} \underbrace{f(\mathbf{x}, \mathbf{w}', \mathbf{w})}_{\substack{\text{Streuungsterm} \\ (\text{BRDF am} \\ \text{Punkt } \mathbf{x})}} \cdot \underbrace{L(\mathbf{x}, \mathbf{w}')}_{\substack{\text{Lichtstärke aus} \\ \text{Einfallsrichtung} \\ \mathbf{w}'}} \cdot \underbrace{\langle \mathbf{w}', \mathbf{n} \rangle}_{\substack{\text{Kosinus des Winkels} \\ \text{zwischen Ober-} \\ \text{flächenormalen } \mathbf{n} \\ \text{und Einfalls-} \\ \text{lichtrichtung } \mathbf{w}'}} d\mathbf{w}'$$

wobei die *Bidirectional Reflectance Distribution Function* (BRDF)  $f(\mathbf{x}, \mathbf{w}', \mathbf{w})$  das Reflexionsverhalten der Fläche im Punkt  $\mathbf{x}$  unter beliebigen Lichteinfallenrichtungen  $\mathbf{w}'$  und Lichtausfallsrichtungen  $\mathbf{w}$  beschreibt.

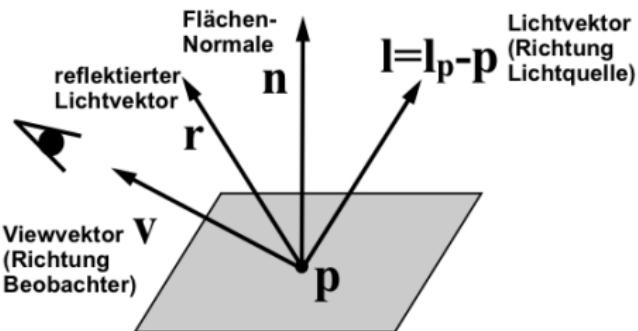
Insgesamt erhält man die Rendergleichung

$$L(\mathbf{x}, \mathbf{w}) = L_e(\mathbf{x}, \mathbf{w}) + \int_{\Omega_x} f(\mathbf{x}, \mathbf{w}', \mathbf{w}) \cdot L(\mathbf{x}, \mathbf{w}') \cdot \langle \mathbf{w}', \mathbf{n} \rangle d\mathbf{w}'$$

# Lokale Beleuchtungsmodelle

Lichtintensität (Farbe) eines (Flächen-)Punktes ist *nur* abhängig von

- der Materialeigenschaft (Farbe) des Punktes
- der Punktposition  $p$
- der Flächennormalen  $n$
- der Position der Lichtquelle  $I_p$
- der Blickrichtung  $v$  des Betrachters



Globale Beleuchtungseffekte (Schatten, Spiegelungen, ... anderer Objekte) werden nicht berücksichtigt



# Das Phong-Beleuchtungsmodell<sup>6</sup>

## Lokales Beleuchtungsmodell mit Vereinfachungen

- Lichtquellen sind punktförmig
- Geometrie der Oberflächen, außer Oberflächennormalen, wird ignoriert
- diffuse und spiegelnde Reflexion wird nur lokal modelliert
- ambiente Reflexion wird global modelliert

Die Intensität des reflektierten Lichts  $I_{out}$  wird berechnet als

$$I_{out} = I_a \cdot k_{ambient} + I_{in} \cdot \left( k_{diffuse} \cdot \langle \mathbf{l}, \mathbf{n} \rangle + k_{specular} \cdot \frac{n+2}{2\pi} \cdot \langle \mathbf{r}, \mathbf{v} \rangle^n \right) \text{ mit}$$

- den Materialkonstanten  $k_{diffuse} + k_{specular} \in [0, 1]$  und  $k_{ambient} \in [0, 1]$
- dem Spiegelexponenten  $n$  (rau:  $n \leq 32$ , glatt:  $n > 32$ )
- dem Normalisierungsfaktor  $\frac{n+2}{2\pi}$ , der dafür sorgt, dass die Helligkeit bei großen Spiegelexponenten nicht abnimmt

---

<sup>6</sup>Bui Tuong Phong: Illumination for Computer Generated Pictures. Communications of the ACM 18, 6 (Jun. 1975)

# Das Blinn-Beleuchtungsmodell<sup>7</sup>

Die Berechnung des Reflexionsvektors  $\mathbf{r} = (2\mathbf{n}\mathbf{n}^t - I)\mathbf{l}$  im Phong-Beleuchtungsmodell ist sehr aufwändig.

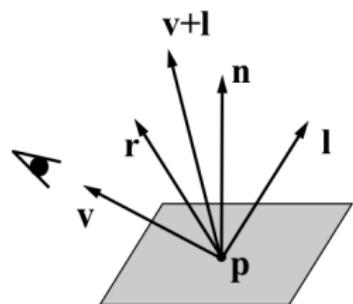
## Das Blinn-Beleuchtungsmodell

- ersetzt  $\mathbf{r}$  durch den *Halfway-Vektor*  $\mathbf{h} = \frac{\mathbf{v}+\mathbf{l}}{\|\mathbf{v}+\mathbf{l}\|}$
- ändert die Berechnung des spekularen Anteils im Phong-Beleuchtungsmodell in

$$I_{specular} = I_{in} \cdot k_{specular} \cdot \frac{n+2}{2\pi} \langle \mathbf{h}, \mathbf{n} \rangle^n$$

- wurde in der Fixed-Function-Pipeline von OpenGL verwendet

Um mit dem Blinn-Modell ähnliche Ergebnisse zu erzielen wie mit dem Phong-Modell, muss der Spiegelexponent  $n$  viermal so groß sein.



<sup>7</sup>James F. Blinn: Models of light reflection for computer synthesized pictures. SIGGRAPH 1977

# Shading (Schattierung)

Unter Shading (Schattierung) versteht man

- 1 die Simulation der Oberflächeneigenschaften von Objekten
- 2 Interpolationsverfahren zur Berechnung der Normalen von beliebigen Punkten eines Polygonnetzes. Objektoberflächen erscheinen „glatter“. Man unterscheidet:

- *Flat-Shading (constant shading):*  
Farbwert des ersten Eckpunkts des zu zeichnenden Polygons wird für das gesamte *Polygon* verwendet
- *Gouraud-Shading<sup>a</sup> (color-interpolation shading):*  
Farbwerte der einzelnen Pixel werden aus Farbwerten der Eckpunkte des Polygons *interpoliert*
- *Phong-Shading (normal-vector interpolation shading):*  
Pro Pixel wird eine Normale aus den Eckpunkt-Normalen des Polygons *interpoliert* und Beleuchtungsmodell mit dieser Normalen ausgewertet

---

<sup>a</sup>Henri Gouraud: Continuous Shading of Curved Surfaces. IEEE Transactions on Computers C-20, 6 (Jun. 1971)

# Beispiele für die verschiedenen Interpolationsverfahren

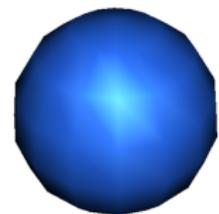
## Flat-Shading

- Vor-/Nachteile
  - + Ressourcenschonend (Eine Lichtberechnung pro Fläche)
  - „Unschöner“ Look (Facetten deutlich sichtbar)



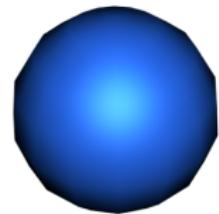
## Gouraud-Shading

- Vor-/Nachteile
  - + Bessere Qualität als Flat-Shading bei überschaubarem Rechenaufwand (Beleuchtungsberechnung pro Vertex)
  - Ergebnis abhängig von Polygon-Anzahl (und Art)



## Phong-Shading

- Vor-/Nachteile
  - + Hohe visuelle Qualität, „unabhängig“ von Polygonanzahl
  - Rechenaufwand (Beleuchtungsberechnung pro Pixel)



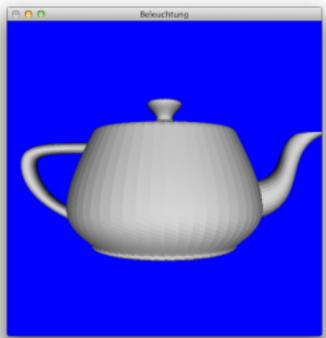
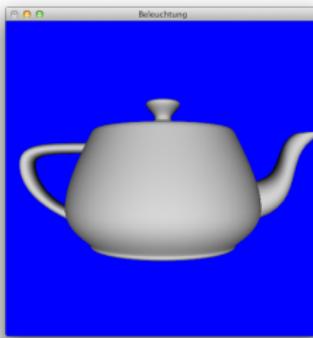
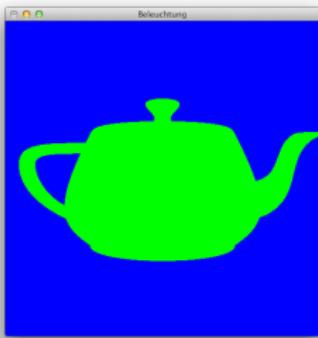
# Beleuchtung und Shading in OpenGL

Die *Fixed-Function-Pipeline* von OpenGL vor 3.0 realisierte das *Bleuchtungsmodell von Blinn* und *Flat- bzw. Gouraud-Shading*

```
def display():
 ...
 glColor(0.0, 1.0, 0.0)
 glutSolidTeapot(1.0)
 ...
 glutSwapBuffers()
```

```
def initGL():
 ...
 glEnable(GL_DEPTH_TEST)
 glEnable(GL_NORMALIZE)
 glEnable(GL_LIGHTING)
 glEnable(GL_LIGHT0)
```

```
def initGL():
 ...
 glShadeModel(GL_FLAT)
 ...
 ...
```



Seit OpenGL 3.0 wird die Fixed-Funktion-Pipeline nicht mehr unterstützt.  
Beleuchtung durch Implementierung eigener Shader (siehe Vorlesung VIII)

# OpenGL Beispiel (Beleuchtung und GL\_NORMAL\_ARRAY)

```

...
myVBO = None

...
def initGL(width, height):
 ...
 glEnable(GL_LIGHTING)
 glEnable(GL_LIGHT0)
 glEnable(GL_DEPTH_TEST)
 glEnable(GL_NORMALIZE)
 ...

def initGeometryFromObjFile():
 global myVBO
 ...
 # load obj file containing
 # vertices, normals, faces
 ...
 # calculate bounding box, center and scale
 ...
 data = []
 for face in faces:
 for vertex in face:
 vn = int(vertex[0])-1
 nn = int(vertex[2])-1
 data.append(vertices[vn]+normals[nn])
 myVBO = vbo.VBO(array(data, 'f'))
...

```

```

def display():
 glClear(GL_COLOR_BUFFER_BIT |\
 GL_DEPTH_BUFFER_BIT)

 myVBO.bind()
 glEnableClientState(GL_VERTEX_ARRAY)
 glEnableClientState(GL_NORMAL_ARRAY)

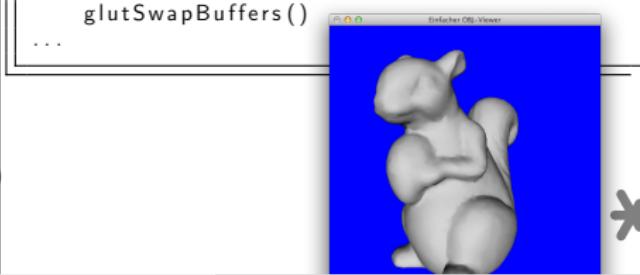
 glVertexPointer(3, GL_FLOAT, 24, myVBO)
 glNormalPointer(GL_FLOAT, 24, myVBO+12)

 glLoadIdentity()
 gluLookAt(0,0,4, 0,0,0, 0,1,0)

 glScale(scale, scale, scale)
 glTranslate(-c[0], -c[1], -c[2])
 glDrawArrays(GL_TRIANGLES, 0, len(data))

 myVBO.unbind()
 glDisableClientState(GL_VERTEX_ARRAY)
 glDisableClientState(GL_NORMAL_ARRAY)
 glutSwapBuffers()
...

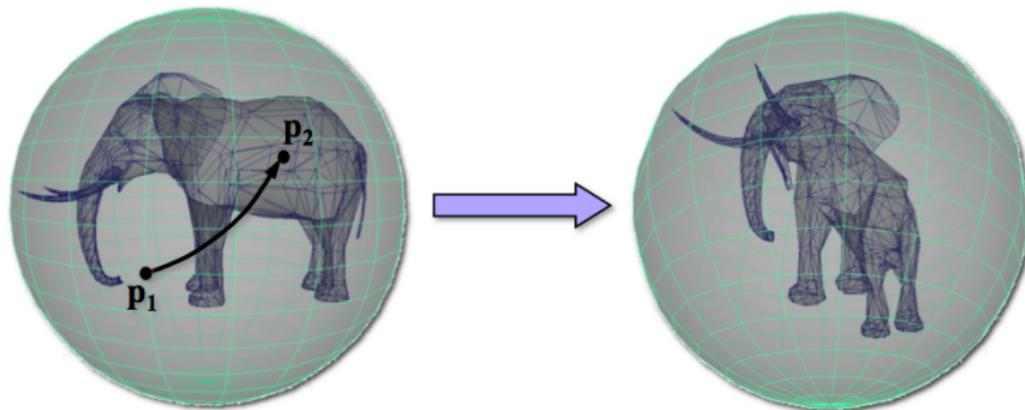
```



# Arball Rotation (intuitive Rotation einer 3D Szene)

„Intuitive“ Rotation eines Objekt um eine beliebige Achse mit der Maus

Durch zwei Punkte  $\mathbf{p}_1$  und  $\mathbf{p}_2$  auf der Einheitskugel wird eine Rotation um die Achse  $\mathbf{v} = \mathbf{p}_1 \times \mathbf{p}_2$  und den Winkel  $\alpha = \arccos(\mathbf{p}_1, \mathbf{p}_2)$  beschrieben



- Position der Maus bei Mausklick definiert  $\mathbf{p}_1$
- Veränderung der Mausposition bei gedrückter Maustaste definiert  $\mathbf{p}_2$

# OpenGL Beispiel (Arcball Rotation)

```

...
def projectOnSphere(x, y, r):
 x, y = x-WIDTH/2.0, HEIGHT/2.0-y
 a = min(r*r, x**2 + y**2)
 z = sqrt(r*r - a)
 l = sqrt(x**2 + y**2 + z**2)
 return x/l, y/l, z/l

def mousebuttonpressed(button, state, x, y):
 global startP, actOri, angle, doRotation
 r = min(WIDTH, HEIGHT)/2.0
 if button == GLUT_LEFT_BUTTON:
 if state == GLUT_DOWN:
 doRotation = True
 startP = projectOnSphere(x, y, r)
 if state == GLUT_UP:
 doRotation = False
 actOri = actOri*rotate(angle, axis)
 angle = 0

def mousemoved(x, y):
 global angle, axis, scaleFactor
 if doRotation:
 r = min(WIDTH, HEIGHT)/2.0
 moveP = projectOnSphere(x, y, r)
 angle = acos(dot(startP, moveP))
 axis = cross(startP, moveP)
 glutPostRedisplay()
...

```

```

def rotate(angle, axis):
 c, mc = cos(angle), 1-cos(angle)
 s = sin(angle)
 l = sqrt(dot(array(axis), array(axis)))
 x, y, z = array(axis)/l
 r = matrix([
 [x*x*mc+c, x*y*mc-z*s, x*z*mc+y*s, 0], \
 [x*y*mc+z*s, y*y*mc+c, y*z*mc-x*s, 0], \
 [x*z*mc-y*s, y*z*mc+x*s, z*z*mc+c, 0], \
 [0, 0, 0, 1]])
 # OpenGL uses column major order
 # -> transpose matrix
 return r.transpose()

def display():
 ...
 glMultMatrixf(actOri*rotate(angle, axis))
 ...
 glDrawArrays(GL_TRIANGLES, 0, len(data))
 ...
 glutSwapBuffers()
 ...

def main():
 # Initialize GLUT
 ...
 glutMouseFunc(mousebuttonpressed)
 glutMotionFunc(mousemoved)
 ...

```



# (Hardware-)Shader

*Shader* sind Programme, die die *Oberflächeneigenschaften* (das Aussehen) von (gerenderten) Objekten bestimmen

- Lichtabsorption
- Reflexion
- Brechungen
- Schatten
- ...



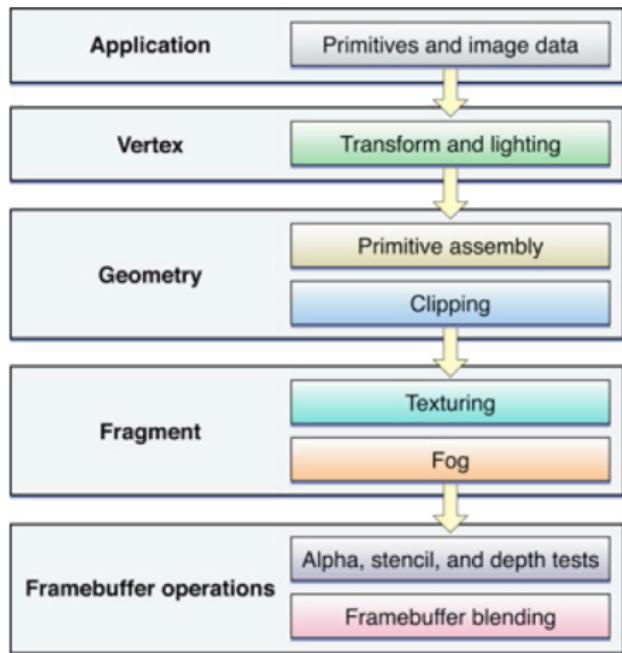
Bildquelle: [https://renderman.pixar.com/products/whats\\_renderman/showcase\\_ratatouille.html](https://renderman.pixar.com/products/whats_renderman/showcase_ratatouille.html)

*Hardware-Shader* sind Programme, die *direkt auf der Grafik-Hardware (GPU)* ausgeführt werden

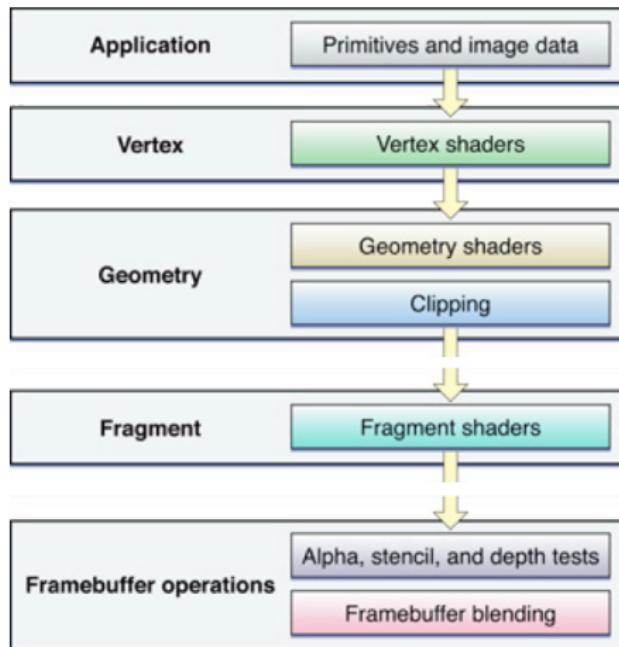
- Werden auch *Echtzeit-Shader* genannt
- Ersetzen Teile der statischen *Fixed-Function-/Standard-Grafikpipeline*

# (Hardware-)Shader und die Standard-Grafikpipeline

## OpenGL Fixed-Function Pipeline

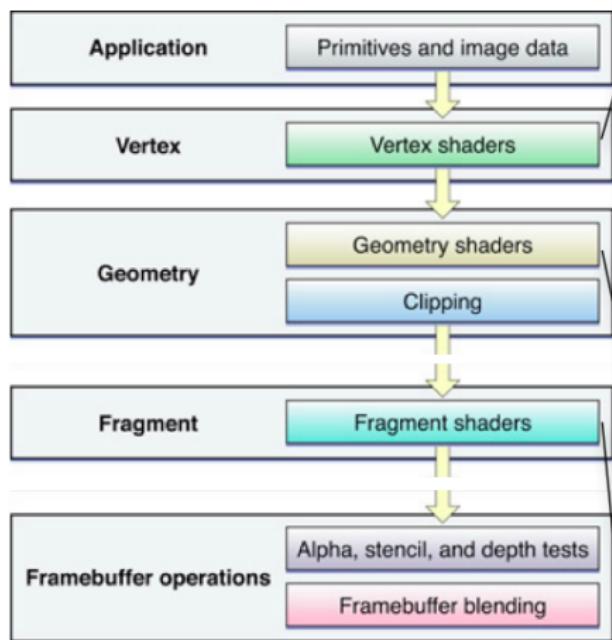


## OpenGL Shader Pipeline



# Hardware-Shader und die Grafikpipeline

## OpenGL Shader Pipeline



```
void mainO {
 gl_FrontColor = gl_Color;
 gl_TexCoord[0] = gl_MultiTexCoord0;
 //gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
 //gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
 gl_Position = ftransform();
}
```

```
void mainO {
 gl_FrontColor = gl_FrontColorIn[0];
 gl_TexCoord[0] = gl_TexCoordIn[0][0];
 gl_Position = gl_PositionIn[0];
 EmitVertexO;
 gl_FrontColor = gl_FrontColorIn[1];
 gl_TexCoord[0] = gl_TexCoordIn[1][0];
 gl_Position = gl_PositionIn[1];
 EmitVertexO;
 gl_FrontColor = gl_FrontColorIn[2];
 gl_TexCoord[0] = gl_TexCoordIn[2][0];
 gl_Position = gl_PositionIn[2];
 EmitVertexO;
 EndPrimitiveO;
}
```

**Optional**

```
uniform sampler2D tex;

void mainO {
 gl_FragColor = gl_Color * texture2D(tex, gl_TexCoord[0].xy);
}
```

# Unified Shaders

Vier Shader-Typen, die „nacheinander“ arbeiten

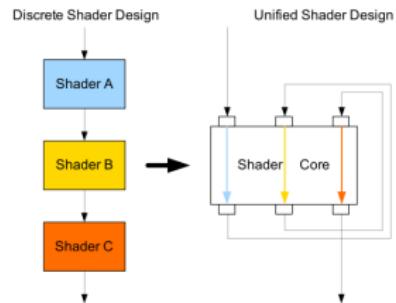
- *Vertex-Shader* transformieren Eckpunkte
- *Tesselation-Shader* unterteilen Flächen in kleinere Flächen
- *Geometrie-Shader* erzeugen aus Primitiven neue Primitive
- *Fragment-Shader* berechnen Attribute (z.B. Farbe) von Pixeln

Unified Shader Modell

- Alle Shader-Typen haben (annähernd) den gleichen Befehlssatz

Unified Shader Architektur

- Flexiblere Nutzung der Recheneinheiten einer GPU
- Jede Recheneinheit kann jeden Shader-Typ bearbeiten



# Unterschiedliche *Shading (Hoch-)Sprachen*

Für das Offline-Rendern photorealistischer Grafiken

- RenderMan (<https://renderman.pixar.com/>)
- Gelato (<http://www.nvidia.co.uk/page/gelato.html>)

Für das Echtzeit-Rendern auf spezieller Grafik-Hardware

- HLSL (High Level Shading Language)
  - Shadersprache (nur) für Microsofts Direct3D API
- CG (C for Graphics)
  - Shaderprache von Nvidia, unabhängig von Hardware und Grafik-API
- GLSL (OpenGL Shading Language)
  - GLSL Version 4.20.6 ist fester Bestandteil von OpenGL 4.2

Für die Verwendung der GPU für wissenschaftliches Rechnen (GPGPU)

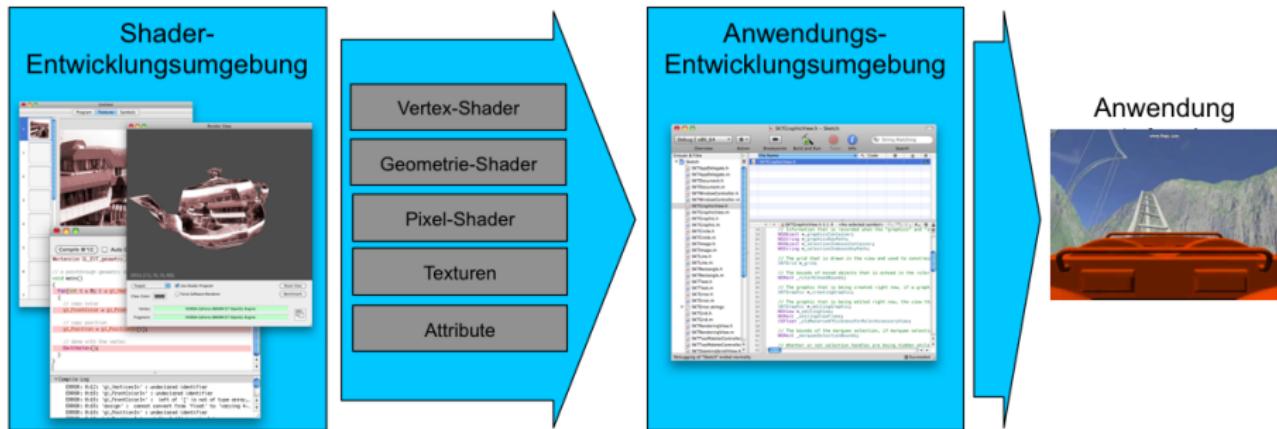
- Brook+ (Standford, AMD), CUDA (NVIDIA), OpenCL (OpenGL)

# Entwicklung von Shadern

Entwicklung von Shadern - insbesondere Debuggen und Testen - kann recht aufwendig sein. Deshalb

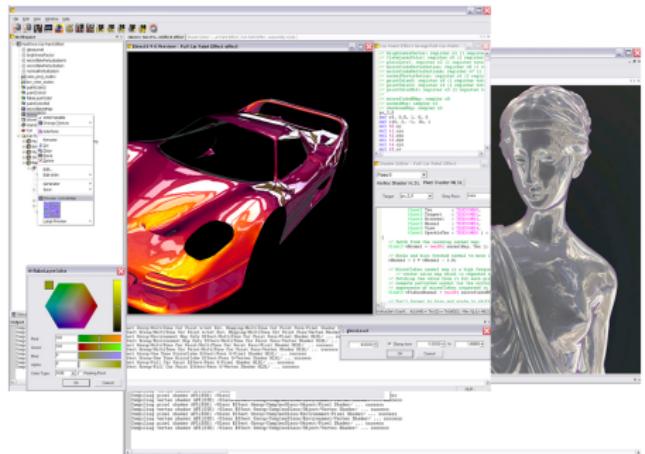
Trennung von

- Shader-Entwicklung (z.B. mit [shdr.bkcore.com](http://shdr.bkcore.com)) und
- Shader-Verwendung (Integration in OpenGL-/DirectX-Anwendung)

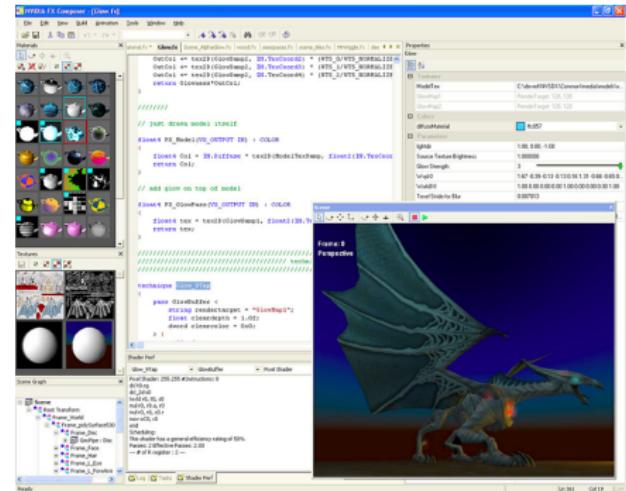


# Shader-Entwicklungsumgebungen

## Rendermonkey (AMD)



## FX Composer (NVIDIA)

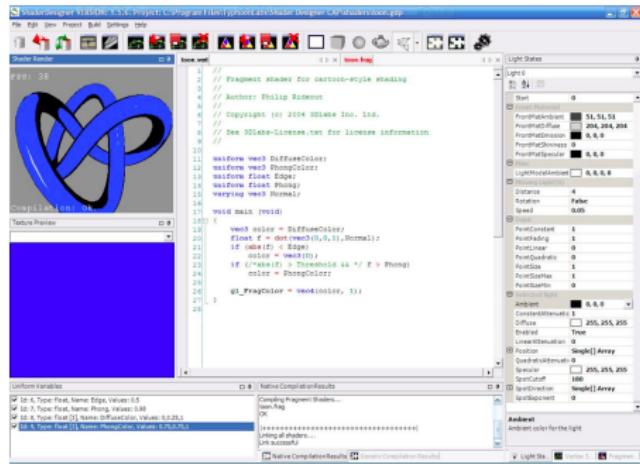


- Nur Windows
- Wird nicht mehr weiterentwickelt (letzte Version vom 18.12.2008)
- <http://developer.amd.com/ARCHIVE/GPU/RENDERMONKEY/Pages/default.aspx>

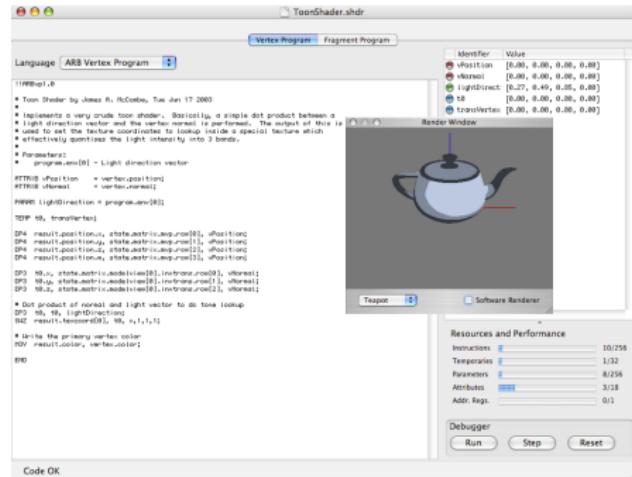
- Nur Windows
- Composer und Debugger erhältlich
- <http://developer.nvidia.com/fx-composer>

# Shader-Entwicklungsumgebungen

## Shader Designer (TyphoonLabs)



## OpenGL Shader Builder (Apple)



- Bestandteil des OpenGL Software Development Kit
- Versionen für Windows und Linux
- <http://www.opengl.org/sdk/tools/ShaderDesigner/>

- <http://developer.apple.com/library/mac/#documentation/GraphicsImaging/Conceptual/OpenGLShaderBuilderUserGuide/Introduction/Introduction.html>

# Uniform, Attribute und Varying Variable

Variablen können in GLSL mit den Schlüsselwörtern *uniform*, *attribute* und *varying* deklariert werden. Beispiel

```
uniform float timeStep;
attribute bool visible;
varying vec3 normal;
```

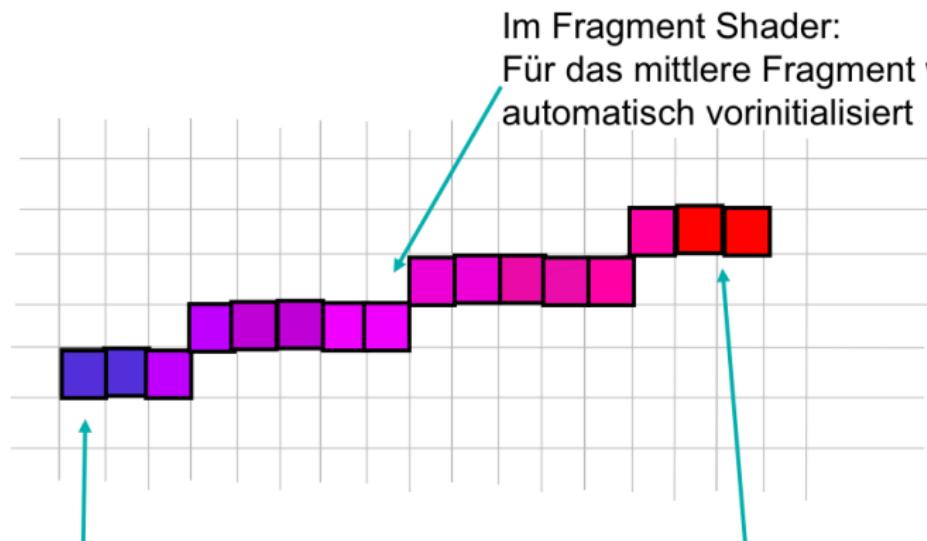
Dient dazu Daten zwischen Shadern und der „Aussenwelt“ auszutauschen

- Uniform-Variable (Daten ändern sich selten (pro Frame))
  - Daten von der Anwendung an den Vertex-/Fragment-Shader
- Attribute-Variable (Daten ändern sich häufig (pro Vertex))
  - Daten von der Anwendung an den Vertex-Shader
- Varying-Variable (Daten werden interpoliert)
  - Daten vom Vertex- an den Fragment-Shader
  - Evtl. Daten vom Vertex-Shader an den Tesselation-/Geometrie-Shader und von dort weiter an den Fragment-Shader

# Interpolation von Varying Variablen

## Beispiel

```
varying float f;
```



Im Fragment Shader:  
Für das mittlere Fragment wird  $f$  auf 0.5  
automatisch vorinitialisiert

Im Vertex Shader:  
Dieser Vertex setzt  $f = 0.0$ ;

Im Vertex Shader:  
Dieser Vertex setzt  $f = 1.0$ ;

# Die GLSL Programmiersprache

## GLSL ist eine C-ähnliche Programmiersprache

- die einheitlich verwendet werden kann für
  - Vertex-, Tesselation-, Geometrie- und Fragment-Shader
- mit
  - Präprozessor (z.B. `#define`),
  - eingebauten Typen für Vektoren, Matrizen, Texturen, ...
  - und vielen Mathematik- und Grafik-Funktionen, wobei
  - Funktionen immer *call-by-value-return* sind
- ohne
  - Chars und Strings
  - goto und switch Statement
  - sizeof Operator
  - static Variablen
  - Zeiger (keine Pointertypen und Dereferenzierung)
  - #include und Header-Dateien

# GLSL Datentypen (keine implizite Konvertierung)

## Skalare

float, int, bool, ...

## Vektoren

vec2, vec3, vec4, ivec2, ivec3, ivec4, bvec2, bvec3, ...

## Matrizen

mat2, mat3, mat4, ...

## Structs und Arrays ähnlich wie in C

```
vec4 points [];
points[2] = vec4(1.0);
```

## Texturen

sampler1D, sampler2D, sampler3D, samplerCube, ...

# GLSL Beispiele

## Strenge Überprüfung der Datentypen (keine implizite Konvertierung)

```
int i = 7;
float f = i; // Fehler, da keine implizite Typkonvertierung
float f = float(i); // Korrekt, da explizite Typkonvertierung
```

## Verwendung von 2D Bildern (Texturen)

```
vec2 coord = vec2(0.7, 0.9); // Texturkoordinaten
uniform sampler2D sp; // Textur (kommt von "aussen")
vec4 color = texture2d(sp, coord); // Auslesen der Textur
```

## Swizzling (Zugriff auf die Elemente 2-, 3- oder 4-dimensionaler Datentypen) mittels x,y,z,w oder r,g,b,a oder s,t,p,q.

```
vec3 v3 = (1.0, 2.0, 3.0, 4.0) // Okay, 4.0 wird ignoriert
vec4 v4 = v3.xxzz // v4=(1.0, 1.0, 3.0, 3.0)
v3.r = v4.p; // v3=(3.0, 2.0, 3.0)
```

# Operatoren auf den vec-Datentypen

## Komponentenweise Operationen

```
vec3 u, v; float f; mat3 m;
v = u + f; // In jeder Komponente wird f addiert
v = u + v; // Komponentenweise Addition von u und v
v = u + m[0]; // Erste Spalte der Matrix wird aufaddiert
v = 3 * u; // Jede Komponente wird mit 3 multipliziert
v = u * v; // Komponentenweise Multiplikation
```

## Vektor-Matrix-Multiplikation

```
vec4 u, v; mat4 m;
u = m * v; // v wird als Spaltenvektor behandelt
u = v * m; // v wird als Zeilenvektor behandelt
m = m * m; // Korrekte Matrixmultiplikation
```

## Skalar- und Kreuz-Produkt

```
vec4 u, v, w; float f;
f = dot(u, v); // f = <u, v>
w = cross(u, v); // w = u x v
w = normalize(w); // Bringt w auf die Laenge 1
f = length(w); // Laenge von w (hier f=1)
```

# Vergleichsoperatoren für die vec-Datentypen

Vergleichsoperatoren wie < oder >=

- sind nur für einfache Datentypen (int, float, ...) zulässig
- liefern als Resultat ein bool

Vergleichsoperatoren für die vec-Datentypen

- sind die Funktionen equal, lessThanEqual, notEqual, ...
- welche als (Vektor-)Resultat einen bvec liefern.
- Mit Hilfe der Funktionen any oder all kann überprüft werden, ob *eine* oder *alle* Komponenten true sind.

Beispiel

```
vec4 u, v;
bool b;
b = any(lessThan(u, v));
```

# (Beispiele) eingebauter Funktionen

## Trigonometrische Funktionen

`sin, cos, tan, asin, acos, atan, radians, degrees, ...`

## Exponential-Funktionen

`pow, exp2, log2, sqrt, inversesqrt, ...`

## Arithmetische Funktionen

`abs, sign, floor, ceil, fract, mod, min, max, clamp, ...`

## Vektor-/Matrix-Funktionen

`length, distance, dot, cross, matrixcompmult, ...`

Im Zweifel immer (komplett) die eingebauten Funktionen verwenden, da diese direkt und effizient in Hardware realisiert sind

```
vec3 v; float f;
f = sqrt(v.x*v.x + v.y*v.y + v.z*v.z); // schlecht
f = length(v); // gut
```

# User-definierte Funktionen

User-definierte Funktionen müssen *vor dem Aufruf* definiert oder zumindest deklariert sein

- Das Überladen von Funktionen ist nicht erlaubt
- Keine Pointer, sondern *call-by-value-return*:
  - in-Parameter (nur lesend) ist Default
  - out-Parameter (nur schreibend)
  - inout-Parameter (lesend und schreibend)
  - return (in der Funktion): Rückgabewert kann alles sein, außer Array

## Beispiel

```
vec3 computeColor(const vec3 normal, // normal ist eine Konstante
 in vec3 tangent, // tangent nur lesbar
 inout vec4 color, // color les- und schreibbar
 int number, // number nur lesbar
 inout int counter); // counter les- und schreibbar
```

# Shader-Programme ...

... liegen (in der Regel) als Textdateien vor. Beispiel *Cartoon-Shader*:

## toonShader.vert

```
varying vec3 normal;

void main()
{
 normal = gl_NormalMatrix * gl_Normal;
 gl_Position = ftransform(); // gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

## toonShader.frag

```
varying vec3 normal;

void main()
{
 float intensity;
 vec4 color;
 vec3 n = normalize(normal);
 vec3 l = normalize(gl_LightSource[0].position).xyz;
 // quantize to 8 steps (0, 1/7, 2/7, 3/7, 4/7, 5/7, 6/7, 1)
 intensity = (floor(dot(l, n)*7.0) + 1.0)/7.0;
 color = vec4(intensity, intensity, intensity, 1.0);
 gl_FragColor = color;
}
```

# Shader-Programme ...

... müssen vom Grafiktreiber kompiliert und gelinkt werden. Beispiel:

## Werden Shader unterstützt?

```
from OpenGL.GLUT import *; from OpenGL.GL import *
from OpenGL.GL.ARБ.shader_objects import *
from OpenGL.GL.ARБ.fragment_shader import *
from OpenGL.GL.ARБ.vertex_shader import *
import sys

def init(width, height):
 # creating a rendering context
 glutInit(sys.argv)
 glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH)
 glutInitWindowSize(width, height)
 glutCreateWindow("Shader test")

 # checking whether system supports ARB_(fragment,vertex)_programm extension
 if not glInitShaderObjectsARB():
 raise RuntimeError(
 """ARB Shader Objects extension is required """
 """but not supported on this machine!""")
```

OpenGL  
Initialisieren

Werden Shader  
unterstützt?

# Shader-Programme ...

... müssen vom Grafiktreiber kompiliert und gelinkt werden. Beispiel:

## Initialisieren von Shadern

- 1 Erzeuge Programm-Handle
- 2 Binde Shader-Programm an Programm-Handle
- 3 Kompiliere Shader-Programm

```
def initShader(type, program):
 # 1. create (fragment/vertex) shader program name (handle)
 shaderHandle = glCreateShaderObjectARB(type)

 # 2. generate shader source
 glShaderSourceARB(shaderHandle, 1, [program])

 # 3. compile shader
 glCompileShaderARB(shaderHandle)
 success = glGetObjectParameterivARB(shaderHandle, GL_OBJECT_COMPILE_STATUS_ARB)
 if not success:
 print glGetInfoLogARB(program)
 sys.exit(-1)
 return shaderHandle
```

The diagram illustrates the three steps of initializing a shader program, each associated with a specific line of code:

- Step 1: Programm-Handle erzeugen** (Create Program Handle) is associated with the line: `shaderHandle = glCreateShaderObjectARB(type)`.
- Step 2: Shader an Handle binden** (Bind Shader to Handle) is associated with the line: `glShaderSourceARB(shaderHandle, 1, [program])`.
- Step 3: Shader-Programm kompilieren** (Compile Shader Program) is associated with the line: `glCompileShaderARB(shaderHandle)`.

# Shader-Programme ...

... müssen vom Grafiktreiber kompiliert und gelinkt werden. Beispiel:

## Linken der kompilierten Shader

```
def linkShaders(vsHandle , fsHandle):
 if vsHandle or fsHandle: # generate shader program handle
 shaderHandle = glCreateProgramObjectARB()
 if vsHandle: # attach vertex shader
 glAttachObjectARB(shaderHandle, vsHandle)
 glDeleteObjectARB(vsHandle) # must not
 if fsHandle: # attach fragment shader
 glAttachObjectARB(shaderHandle, fsHandle)
 glDeleteObjectARB(fsHandle) # must not

 # link shader program
 glLinkProgramARB(shaderHandle)

 # print link status
 success = glGetObjectParameterivARB(shaderHandle, GL_OBJECT_LINK_STATUS_ARB)
 if not success:
 print glGetInfoLogARB(handle)
 sys.exit(-1)
 return shaderHandle
```

Mindestens ein GLSL-Shader geladen?

Shader-Handle erzeugen

Vertex-/Fragment-Shader an Shader-Programm binden

Shader-Programm erzeugen

Fehler-Ausgabe, wenn Linken misslungen

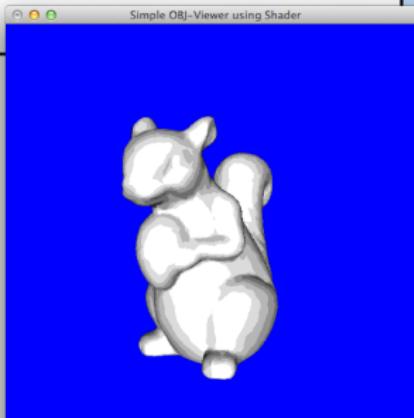
# Shader-Programme ...

... müssen vom Grafiktreiber kompiliert und gelinkt werden. Beispiel:

## Binden eines Shaders

```
...
use GLSL shader
if shaderHandle:
 glUseProgramObjectARB(shaderHandle)
else: # use fixed function pipeline
 glUseProgramObjectARB(0)
...
...
```

Umschalten auf Standard OpenGL-Renderpipeline



# Shader-Programme ab PyOpenGL 3.0.1

PyOpenGL 3.0.1 führt das Modul `OpenGL.GL.shaders` ein, welches das Initialisieren, Binden und Linken von Shadern stark vereinfacht:

```
...
from OpenGL.GL.shaders import *
...
shaderProgram = None
...
def initGL(width, height):
 ...
 if not glUseProgram:
 print "Missing_Shader_Objects!"
 sys.exit(1)
 vertexShader = open("toonShader.vert", "r").read()
 fragmentShader = open("toonShader.frag", "r").read()
 global shaderProgram
 shaderProgram = compileProgram(compileShader(vertexShader, GL_VERTEX_SHADER), \
 compileShader(fragmentShader, GL_FRAGMENT_SHADER),)
...
def display():
 ...
 if shaderProgram:
 glUseProgram(shaderProgram)
 else:
 glUseProgram(0)
 glDrawElementsui(GL_TRIANGLES, indices)
 # oder glDrawArrays(GL_TRIANGLES, 0, len(data))
 ...
```

# Kommunikation zwischen Anwendung und Shader

Im Allgemeinen *Ein-Weg-Kommunikation* zwischen Applikation und Shader

- Shader können (effizient) keine Berechnungen an die Applikation zurückgeben
- Ausnahmen
  - Debugging Kanäle
  - Texturen
  - (Pixel-)Buffer

„Einfachste Kommunikation“ über den OpenGL-Kontext möglich

- Shader können auf alle OpenGL-States und -Felder zugreifen
- „Missbrauch“ der OpenGL-States-/Felder ist aber *unsauber*



# Kommunikation zwischen Anwendung und Shader

Kommunikation über benutzerdefinierte Variablen in der Anwendung

- uniform-Variablen
  - Innerhalb eines Streams nicht änderbar
- attribute-Variablen
  - Jederzeit änderbar
  - Nur innerhalb von Vertex-Shadern nutzbar

Die Anwendung muss die Position der Variablen im Grafik-Speicher ermitteln

- `glGetUniformLocation(GLuint prog, const GLchar* name)`
- `glGetAttributeLocation(GLuint prog, const GLchar* name)`

Anschließend können Werte übergeben werden

- `glUniform{1,2,3,4}f[v](varLocation, value)`
- `glUniformMatrix{1,2,3,4}f[v](matLoc, transpose, value)`
- ...

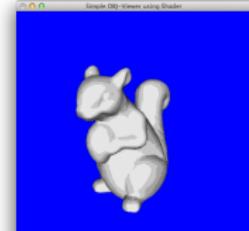
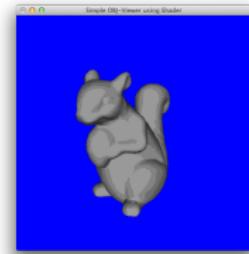
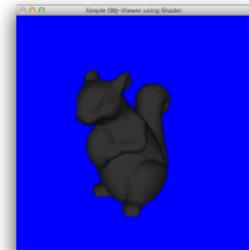
# Beispiel Cartoon-Shader mit Abschwächung

```
uniform float attenuation;
varying vec3 normal;

void main()
{
 vec3 n = normalize(normal);
 vec3 l = normalize(gl_LightSource[0].position).xyz;
 float intensity = attenuation*(floor(dot(l, n)*7.0) + 1.0)/7.0;
 gl_FragColor = vec4(intensity, intensity, intensity, 1.0);
}
```

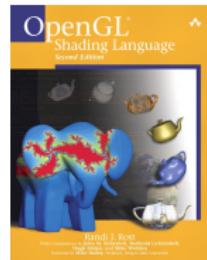
```
def sendValue(shaderProgram, varName, value):
 # determine location of uniform variable varName in shaderProgram
 varLocation = glGetUniformLocation(shaderProgram, varName)
 # pass value to shader
 glUniform1f(varLocation, value)
...
def display():
 ...
 if program:
 glUseProgram(program)
 sendValue(program, "attenuation", at)
 glDrawArrays(GL_TRIANGLES, 0, len(data))
 ...

def keypressed(key, x, y):
 ...
 if key == 'a':
 global at
 at = (at+0.1)%1
```



# Weitere Literatur und Links zum Thema Shader

- Randi J. Rost, Bill Licea-Kane, Dan Ginsburg, John M. Kessenich, Barthold Lichtenbelt, Hugh Malan, Mike Weiblen; *OpenGL Shading Language (3rd Edition)*, Addison-Wesley, 2009
  - [www.3dshaders.com](http://www.3dshaders.com)
- GLSL Homepage
  - [www.opengl.org/documentation/glsl/](http://www.opengl.org/documentation/glsl/)
- PyOpenGL Tutorial
  - <http://pyopengl.sourceforge.net/context/tutorials/index.xhtml>
- General-Purpose Computation on Graphics Hardware
  - <http://gpgpu.org>



**GPGPU**

# OpenGL 4.x

Alle Code-Teile auf der rechten Seite funktionieren in OpenGL 4.x nicht (mehr), denn es gibt keine

- Fixed-Function Pipeline
- Modell-View- und Projektions-Matrizen
- Beleuchtungs-Modell(e)
- (Standard-)Licht(er)
- ...

Man braucht also

- Matrizenhilfsfunktionen
- Shader
- ...

```
def initGL(width, height):
 ...
 # Set orthographic projection
 glMatrixMode(GL_PROJECTION)
 glLoadIdentity()
 gluPerspective(45., 1, 0.1, 100.0)
 glMatrixMode(GL_MODELVIEW)
 glLoadIdentity()
 glEnable(GL_LIGHTING)
 glEnable(GL_LIGHT0)
 ...

def display():
 ...
 glLoadIdentity()
 gluLookAt(0,0,4, 0,0,0, 0,1,0)
 glMultMatrixf(actOri*rotate(angle, axis))

 glScale(scale, scale, scale)
 glTranslate(-c[0], -c[1], -c[2])
 ...

def resizeViewport(width, height):
 ...
 glMatrixMode(GL_PROJECTION)
 glLoadIdentity()
 if height == 0: aspect = float(width)
 else: aspect = float(width)/height
 gluPerspective(45., aspect, 0.1, 100.0)
```

# Wiederholung (Modell-View-Transformationen)

```

def rotationMatrix(angle, axis):
 c, mc = cos(angle), 1-cos(angle)
 s = sin(angle)
 l = sqrt(dot(array(axis), array(axis)))
 x, y, z = array(axis)/l
 r = matrix([
 [x*x*mc+c, x*y*mc-z*s, x*z*mc+y*s, 0], \
 [x*y*mc+z*s, y*y*mc+c, y*z*mc-x*s, 0], \
 [x*z*mc-y*s, y*z*mc+x*s, z*z*mc+c, 0], \
 [0, 0, 0, 1]])
 return r

def scaleMatrix(sx, sy, sz):
 s = matrix([[sx, 0, 0, 0], \
 [0, sy, 0, 0], \
 [0, 0, sz, 0], \
 [0, 0, 0, 1]])
 return s

def translationMatrix(tx, ty, tz):
 t = matrix([[1, 0, 0, tx], \
 [0, 1, 0, ty], \
 [0, 0, 1, tz], \
 [0, 0, 0, 1]])
 return t

```

```

def lookAtMatrix(ex, ey, ez, cx, cy, cz, ux, uy, uz):
 e = array([ex, ey, ez]) # eye position
 c = array([cx, cy, cz]) # center
 up = array([ux, uy, uz]) # up vector

 # normalize up vector
 lup = sqrt(dot(up, up))
 up = up / lup
 # get view direction
 f = c - e
 lf = sqrt(dot(f, f))
 f = f / lf
 # calculate s
 s = cross(f, up)
 ls = sqrt(dot(s, s))
 s = s / ls
 # calculate u
 u = cross(s, f)
 # create lookAt matrix
 l = matrix([
 [s[0], s[1], s[2], -dot(s, e)], \
 [u[0], u[1], u[2], -dot(u, e)], \
 [-f[0], -f[1], -f[2], dot(f, e)], \
 [0, 0, 0, 1]])
 return l

```



# Wiederholung (Projektions-Transformationen)

```
...

def perspectiveMatrix(fovy, aspect, zNear, zFar):
 """Entspricht gluPerspective(fovy, aspect, zNear, zFar)"""
 f = 1.0 / tan(fovy/2.0) # cotan(fovy/2)
 aspect = float(aspect)
 zNear = float(zNear)
 zFar = float(zFar)
 p = matrix([[f / aspect, 0, 0, 0],
 [0, f, 0, 0],
 [0, 0, (zFar + zNear) / (zNear - zFar), (2 * zFar * zNear) / (zNear - zFar)],
 [0, 0, -1, 0]])

 return p

...
```



# Die *Display-Funktion*

Alt

```
def display():
 ...
 glLoadIdentity()
 gluLookAt(0,0,3, 0,0,0, 0,1,0)
 glMultMatrixf(actOri*rotate(angle, axis))
 glScale(scale, scale, scale)
 glTranslate(-c[0], -c[1], -c[2])

 if program:
 glUseProgram(program)

 myVBO.bind()
 glVertexPointer(3, GL_FLOAT, 24, myVBO)
 glNormalPointer(GL_FLOAT, 24, myVBO+12)
 glDrawArrays(GL_TRIANGLES, 0, len(data))
 myVBO.unbind()
 ...
```

Neu

```
def display():
 ...
 mvMat = lookAtMatrix(0,0,3, 0,0,0, 0,1,0)
 mvMat *= (rotationMat(angle, axis)*actOri)
 mvMat *= scaleMat(scale, scale, scale)
 mvMat *= translationMat(-c[0], -c[1], -c[2])
 # in GLS >1.4 can be done in the shader
 normalMat = inv(mvMat[0:3,0:3]).T

 # global projective Matrix pMatrix
 mvMat = pMatrix * mvMat

 glUseProgram(program)
 sendMat4(program, "mvMatrix", mvMat)
 sendMat4(program, "mvpMatrix", mvMat)
 sendMat3(program, "normalMatrix", normalMat)
 sendVec4(program, "diffuseColor", diffCol)
 sendVec4(program, "ambientColor", ambCol)
 sendVec4(program, "specularColor", specCol)
 sendVec3(program, "lightPosition", lightPos)

 myVBO.bind()
 glVertexPointer(3, GL_FLOAT, 24, myVBO)
 glNormalPointer(GL_FLOAT, 24, myVBO+12)
 glDrawArrays(GL_TRIANGLES, 0, len(data))
 myVBO.unbind()
 ...
```

# Wiederholung (Kommunikation Anwendung / Shader)

Effizienter als ...

```
def sendValue(shaderProgram, varName, value):
 varLocation = glGetUniformLocation(shaderProgram, varName)
 glUniform1f(varLocation, value)

def sendVec3(shaderProgram, varName, value):
 varLocation = glGetUniformLocation(shaderProgram, varName)
 glUniform3f(varLocation, *value)

def sendVec4(shaderProgram, varName, value):
 varLocation = glGetUniformLocation(shaderProgram, varName)
 glUniform4f(varLocation, *value)

def sendMatrix3(shaderProgram, varName, matrix):
 varLocation = glGetUniformLocation(shaderProgram, varName)
 glUniformMatrix3fv(varLocation, 1, GL_TRUE, matrix.tolist())

def sendMatrix4(shaderProgram, varName, matrix):
 varLocation = glGetUniformLocation(shaderProgram, varName)
 glUniformMatrix4fv(varLocation, 1, GL_TRUE, matrix.tolist())
```

... ist, sich alle „Variablenadressen“ *nur einmal* mittels

`varLocation = glGetUniformLocation(shaderProgram, varName)`  
zu holen (z.B. in `init()`) und anschließend, während des Renderns mittels  
`glUniform... (varLocation, value)` zu setzen.

# Ein Gouraud-Shader (Vertex-Shader)

```
// Gouraud vertex shader (borrowed from Richard S. Wright Jr., OpenGL SuperBible)
uniform vec4 ambientColor;
uniform vec4 diffuseColor;
uniform vec4 specularColor;
uniform vec3 lightPosition;
uniform mat4 mvpMatrix;
uniform mat4 mvMatrix;
uniform mat3 normalMatrix;

varying vec4 varyingColor;

void main(void)
{
 vec3 eyeNormal = normalize(normalMatrix * gl_Normal); // surface normal in eye coordinates
 // vertex position in eye coordinates
 vec4 vPosition4 = mvMatrix * gl_Vertex;
 vec3 vPosition3 = vPosition4.xyz / vPosition4.w;
 vec3 vLightDir = normalize(lightPosition - vPosition3); // vector to light source
 float diff = max(0.0, dot(eyeNormal, vLightDir)); // diffuse intensity
 varyingColor = diff * diffuseColor; // diffuse light
 varyingColor += ambientColor; // add in ambient light
 // specular light
 vec3 reflection = normalize(reflect(-vLightDir, eyeNormal));
 float spec = max(0.0, dot(eyeNormal, reflection));
 if (diff != 0.0) { // skip if diffuse light is zero
 float fSpec = pow(spec, 128.0);
 varyingColor.rgb += vec3(fSpec, fSpec, fSpec);
 }
 gl_Position = mvpMatrix * gl_Vertex; // transform geometry
}
```

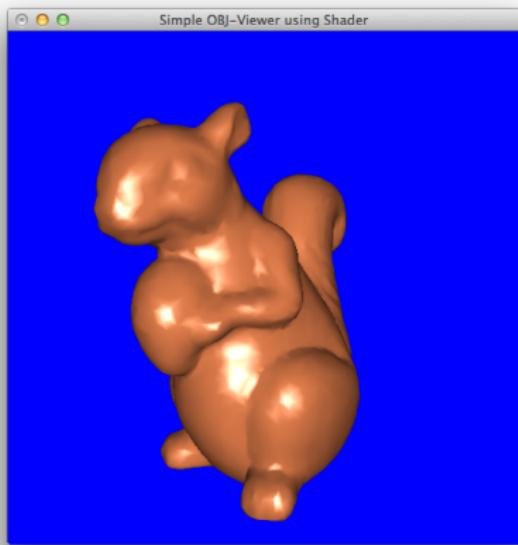


# Ein Gouraud-Shader (Fragment-Shader, Licht-Parameter)

```
// Gouraud fragment shader (borrowed from
// Richard S. Wright Jr., OpenGL SuperBible)
varying vec4 varyingColor;
```

```
void main(void)
{
 gl_FragColor = varyingColor;
}
```

```
def display():
 ...
 lightPos = [0, 0, 1]
 diffCol = [0.7059, 0.3922, 0.2353, 1]
 ambCol = [0.1765, 0.0980, 0.0588, 1]
 specCol = [0.3529, 0.1961, 0.1176, 1]
 sendVec4(program, "diffuseColor", diffCol)
 sendVec4(program, "ambientColor", ambCol)
 sendVec4(program, "specularColor", specCol)
 sendVec3(program, "lightPosition", lightPos)
 ...
```



# Ein Phong-Shader (Vertex-Shader)

```
uniform mat4 mvpMatrix;
uniform mat4 mvMatrix;
uniform mat3 normalMatrix;
uniform vec3 lightPosition;

varying vec3 varyingVertex;
varying vec3 varyingLightDir;
varying vec3 varyingNormal;

void main(void)
{
 // Get surface normal in eye coordinates
 varyingNormal = normalMatrix * gl_Normal;

 // Get vertex position in eye coordinates
 vec4 vPosition4 = mvMatrix * gl_Vertex;
 varyingVertex = vPosition4.xyz / vPosition4.w;

 // Get vector to light source
 varyingLightDir = normalize(lightPosition - varyingVertex);

 // Transform the geometry
 gl_Position = mvpMatrix * gl_Vertex;
}
```



# Ein Phong-Shader (Fragment-Shader)

```

uniform vec4 ambientColor;
uniform vec4 diffuseColor;
uniform vec4 specularColor;

varying vec3 varyingVertex;
varying vec3 varyingLightDir;
varying vec3 varyingNormal;

float intensity(vec3 u, vec3 v) {
 return max(0.0, dot(normalize(u), normalize(v)));
}

void main(void)
{
 vec3 eye = normalize(-varyingVertex); // Eye position is view direction
 vec3 reflected = normalize(reflect(-varyingLightDir, varyingNormal)); // reflected light

 // Ambient color
 gl_FragColor = gl_LightSource[0].ambient * ambientColor;

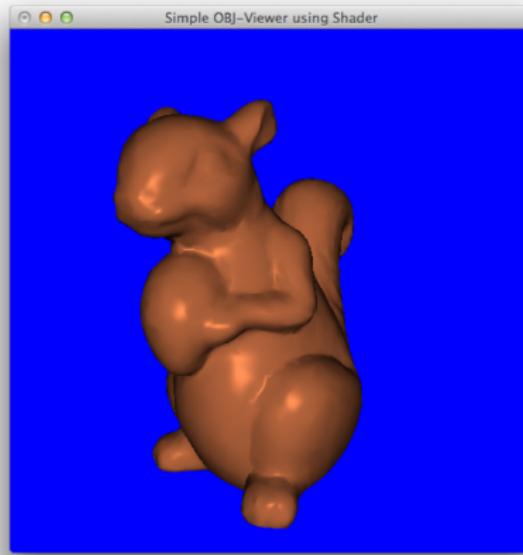
 // Dot product between normal and light direction gives diffuse intensity
 float dl = intensity(varyingNormal, varyingLightDir);
 gl_FragColor += dl * gl_LightSource[0].diffuse * diffuseColor;

 // If diffuse light is zero, don't even bother with the power function
 if(dl != 0.0) {
 // Dot product between reflected light and view direction gives specular intensity
 float shininess = 128.0;
 float sl = pow(intensity(reflected, eye), shininess);
 gl_FragColor += sl * gl_LightSource[0].specular * specularColor;
 }
}

```



# Ein Phong-Shader (Ergebnis)



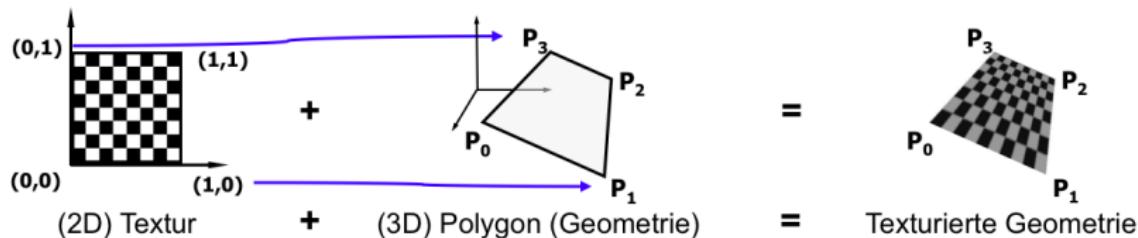
# Texturierung

## Schattierungsverfahren (Flat, Gouraud, Phong, ...)

- Oberfläche durch Dreiecke repräsentiert und mittels Shading eingefärbt
- Keine Berücksichtigung von Oberflächendetails (Strukturen, ...)

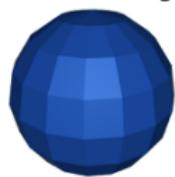
## Texturierung

- Ergänzung von Oberflächendetails durch Projektion von Bildern

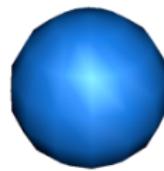


- Erhöhung der visuellen Details bei niedriger geometrischer Komplexität

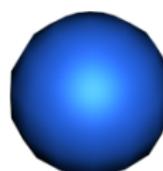
Flat-Shading



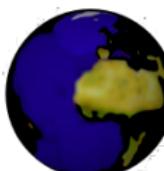
Gouraud-Shading



Phong-Shading



Texturierung



# Texturen in OpenGL (Laden von Texturen)

Bild(er) mittels *Python Image Library* laden...

```
import Image
image = Image.open("smiley-face.png")
```



... und an OpenGL übergeben

```
imagedata = numpy.array(image) # convert image to numpy array
imagedata = imagedata[::-1,:] # mirror image on x-axis
imagedata = imagedata.tostring() # get data as raw bytes

textureID = glGenTextures(1) # allocate texture object(s)
glBindTexture(GL_TEXTURE_2D, textureIDs) # bind texture object to texture state

set up texture parameter for the actual binded texture (THIS IS NOT OPTIONAL!!!)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST)

load texture data
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 512, 512, 0, GL_RGBA, GL_UNSIGNED_BYTE, image)
```

# Texturen in OpenGL (Laden von (1D, 2D, 3D)Texturen)

```
void glTexImage1D(GLenum target, GLint level, GLint internalformat, GLsizei width,
 GLint border, GLenum format, GLenum type, void *data)

void glTexImage2D(GLenum target, GLint level, GLint internalformat, GLsizei width,
 GLsizei height, GLint border, GLenum format, GLenum type, void *data)

void glTexImage3D(GLenum target, GLint level, GLint internalformat, GLsizei width,
 GLsizei height, GLsizei depth, GLint border, GLenum format, GLenum type,
 void *data)
```

**target** spezifiziert die Zieltextur:

GL\_TEXTURE\_1D, GL\_TEXTURE\_2D, GL\_TEXTURE\_3D, GL\_PROXY\_TEXTURE\_2D, ...

**level** spezifiziert den Mipmapping-Level. Ist 0 für *nonmipmapped* Texturen

**internalformat** spezifiziert das interne Texturformat:

GL\_ALPHA, GL\_LUMINANCE, GL\_LUMINANCE\_ALPHA, GL\_RGB, GL\_RGBA, ...

**width, height, depth** spezifiziert die Größe der Textur (am besten Potenzen von 2)

**border** spezifiziert die Größe eines zusätzlichen Texturrandes

**format** spezifiziert das OpenGL Pixelformat:

GL\_RGB, GL\_RGBA, ...

**type** spezifiziert den Typ der Daten:

GL\_UNSIGNED\_BYTE, GL\_BYTE, ...

**\*data** Zeiger auf die Texturdaten



# Texturen in OpenGL (Textur-Objekte)

Eine Textur ist Bestandteil eines *Textur-Zustands* (texture state), welcher sich zusammensetzt aus

- der eigentlichen Textur (Bild) und
- einer Menge von Textur-Parametern, die
  - das Verhalten der Textur-Koordinaten und Textur-Filter bestimmen und
  - mit Hilfe der Funktion `glTexParameter` gesetzt werden

## Textur-Objekte

- erlauben es mehrere Textur-Zustände zu laden und
- schnell zwischen Textur-Zuständen zu wechseln.
- werden erzeugt mittels (liefert „*handler*“ für Textur-Zustände)

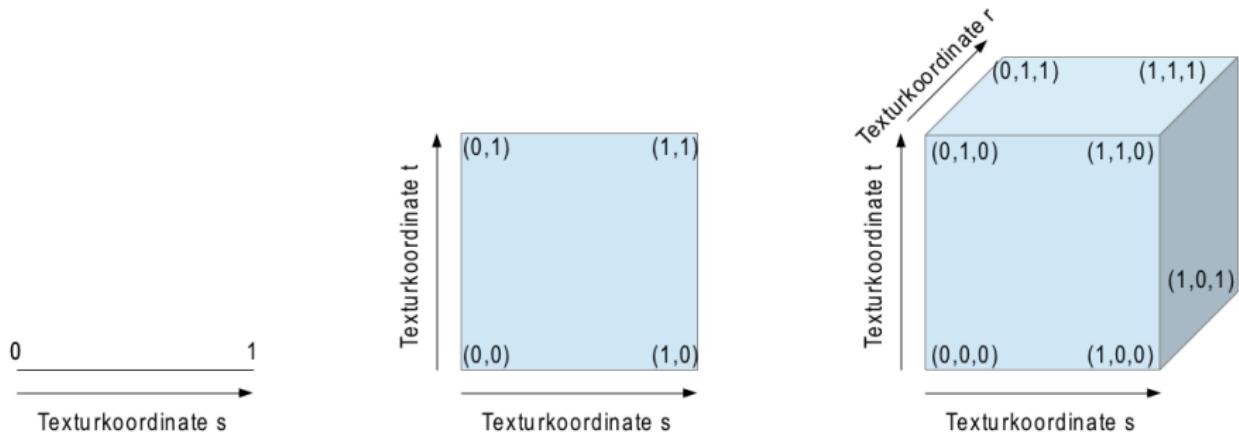
```
void glGenTextures(GLsizei n, GLuint *textures)
```

- „Binden“ eines Textur-Zustands an ein Textur-Objekt mittels

```
void glBindTexture(GLenum target, GLuint texture)
```

# Textur-Koordinaten

Jedem Vertex werden *Textur-Koordinaten*  $s, t, r, q \in [0, 1]$  zugeordnet



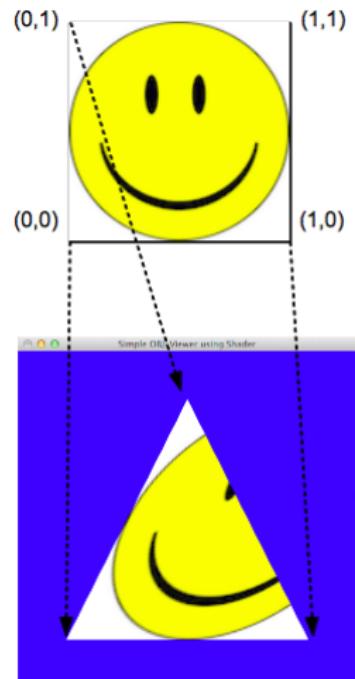
- Standardmäßig gilt für den homogene (Skalierungs-)Faktor  $q = 1$
- Als Koordinaten in der Textur werden verwendet  $s/q, t/q, r/q$

# Textur-Koordinaten

Jedem Vertex werden *Textur-Koordinaten*  $s, t, r, q \in [0, 1]$  zugeordnet

```
def init():
 ...
 data = [[-1.0, -1.0, 0.0], # Vertex-Koordinaten (-1, -1, 0)
 [0.0, 0.0], # Textur-Koordinaten (0, 0)
 [1.0, -1.0, 0.0], # Vertex-Koordinaten (1, -1, 0)
 [1.0, 0.0], # Textur-Koordinaten (1, 0)
 [0.0, 1.0, 0.0], # Vertex-Koordinaten (0, 1, 0)
 [0.0, 1.0]] # Textur-Koordinaten (0, 1)
 myVBO = vbo.VBO(array(data, 'f'))
 ...

```



# Texturen in OpenGL (Textur-Parameter)

Die *Textur-Parameter* werden eingestellt mittels

```
void glTexParameterf(GLenum target, GLenum pname, GLfloat param)
void glTexParameteri(GLenum target, GLenum pname, GLint param)
void glTexParameterfv(GLenum target, GLenum pname, GLfloat *params)
void glTexParameteriv(GLenum target, GLenum pname, GLint *params)
```

**target** spezifiziert den Textur-Modus:

GL\_TEXTURE\_1D, GL\_TEXTURE\_2D, GL\_TEXTURE\_3D, GL\_TEXTURE\_CUBE\_MAP, ...

**pname** spezifiziert den Parameter der gesetzt werden soll:

GL\_TEXTURE\_MIN\_FILTER, GL\_TEXTURE\_MAG\_FILTER, ...

**param, \*params** Wert des Parameters der gesetzt wird

Beispiele: **pname** ist

GL\_TEXTURE\_MIN\_FILTER, GL\_TEXTURE\_MAG\_FILTER: „*Minification-Filter*“ bzw.

„*Magnification-Filter*“ kann mittels GL\_NEAREST, GL\_LINEAR auf Nächste-Nachbar- bzw. Lineare-Interpolation gesetzt werden

GL\_TEXTURE\_WRAP\_S, GL\_TEXTURE\_WRAP\_T, GL\_TEXTURE\_WRAP\_R: Textur-Umbruch (texture wrap)

(was passiert mit Texturkoordinaten ausserhalb von [0, 1]) kann mittels GL\_REPEAT, GL\_CLAMP, GL\_CLAMP\_TO\_EDGE, GL\_CLAMP\_TO\_BORDER gesetzt werden

# Beispiel Texturfilterung

In der Regel wird ein Pixel in der Textur (texel) nicht auf genau ein Pixel auf dem Bildschirm abgebildet und es muss „gefiltert“ werden

```
glTexParameteri(GL_TEXTURE_2D,
 GL_TEXTURE_MAG_FILTER, GL_NEAREST)
glTexParameteri(GL_TEXTURE_2D,
 GL_TEXTURE_MIN_FILTER, GL_NEAREST)
```

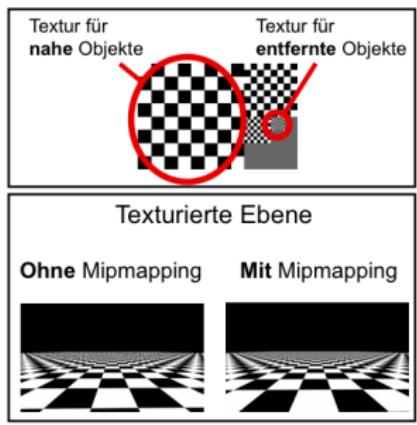
```
glTexParameteri(GL_TEXTURE_2D,
 GL_TEXTURE_MAG_FILTER, GL_LINEAR)
glTexParameteri(GL_TEXTURE_2D,
 GL_TEXTURE_MIN_FILTER, GL_LINEAR)
```



# Mipmapping

Mipmapping („Multum in parvo“ (Viele Dinge auf kleinem Raum))

- Vermeiden von Alias-Effekten durch eine Reihe von Texturen mit verschiedenen (vorberechneten) Auflösungen
  - Jede Stufe halbiert die Auflösung in jeder Richtung
- Während des Renderns wird bei der Texturierung die für eine Dreiecksgröße *optimale Auflösung* verwendet



Nachdem eine Textur in OpenGL geladen wurde können automatisch Mipmaps erstellt werden mit

```
void glGenerateMipmap(GLenum target)
```

`target` spezifiziert den Textur-Modus:

`GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, GL_TEXTURE_CUBE_MAP, ...`

# Texturierung (Vertex- und Fragment-Shader)

## Vertex-Shader

```
// Simple texture vertex shader
uniform mat4 mvpMatrix;

void main()
{
 gl_TexCoord[0] = gl_MultiTexCoord0;
 gl_Position = mvpMatrix * gl_Vertex;
}
```

## Fragment-Shader

```
// Simple texture fragment shader
uniform sampler2D colorMap;

void main(void)
{
 gl_FragColor = texture2D(colorMap, gl_TexCoord[0].st);
}
```

# Texturierung (Die Anwendung)

```
def init():
 ...
 data = [[-1.0, -1.0, 0.0, 0.0, 0.0],
 [1.0, -1.0, 0.0, 1.0, 0.0],
 [1.0, 1.0, 0.0, 1.0, 1.0],
 [-1.0, 1.0, 0.0, 0.0, 1.0]]
 myVBO = vbo.VBO(array(data, 'f'))

 image = array(Image.open("smiley-face.png"))[::-1,:,:].tostring()
 image2 = array(Image.open("sad-smiley.png"))[::-1,:,:].tostring()

 global textureIDs
 textureIDs = glGenTextures(2) # generate list of texture ids

 glBindTexture(GL_TEXTURE_2D, textureIDs[0])
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 128, 128, 0, GL_RGBA, GL_UNSIGNED_BYTE, image)
 glGenerateMipmap(GL_TEXTURE_2D)

 glBindTexture(GL_TEXTURE_2D, textureIDs[1])
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 128, 128, 0, GL_RGBA, GL_UNSIGNED_BYTE, image2)
 glGenerateMipmap(GL_TEXTURE_2D)
 ...
```



# Texturierung (Die Anwendung)

```

def display():
 ...
 glEnableClientState(GL_VERTEX_ARRAY)
 glEnableClientState(GL_TEXTURE_COORD_ARRAY)
 myVBO.bind()

 glUseProgram(program)
 sendMatrix4(program, "mvpMatrix", mvpMatrix)

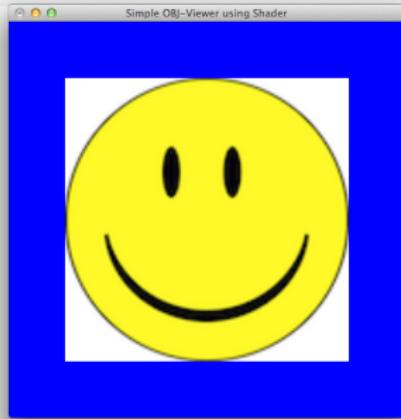
 glVertexPointer(3, GL_FLOAT, 20, myVBO)
 glTexCoordPointer(2, GL_FLOAT, 20, myVBO+12)

 glBindTexture(GL_TEXTURE_2D, textureIDs[texNr])
 glDrawArrays(GL_QUADS, 0, len(data))

 myVBO.unbind()
 glDisableClientState(GL_VERTEX_ARRAY)
 glDisableClientState(GL_TEXTURE_COORD_ARRAY)
 ...

def keypressed(key, x, y):
 ...
 global texNr
 if key == 't':
 texNr = (texNr + 1)%2
 ...

```



# Kombination von Textur und Beleuchtung (Shader)

## Vertex-Shader

```
// Phong vertex shader with texture support
uniform mat4 mvpMatrix;
uniform mat4 mvMatrix;
uniform mat3 normalMatrix;
uniform vec3 lightPosition;

varying vec3 varyingNormal;
varying vec3 varyingLightDir;

void main(void)
{
 // Pass texture coordinates
 gl_TexCoord[0] = gl_MultiTexCoord0;

 // Get surface normal in eye coordinates
 varyingNormal = normalMatrix * gl_Normal;

 // Get vertex position in eye coordinates
 vec4 vPosition4 = mvMatrix * gl_Vertex;
 vec3 vPosition3 = vPosition4.xyz / vPosition4.w;

 // Get vector to light source
 varyingLightDir = normalize(lightPosition - vPosition3);

 // Transform the geometry
 gl_Position = mvpMatrix * gl_Vertex;
}
```

# Kombination von Textur und Beleuchtung (Shader)

## Fragment-Shader

```
// Phong fragment shader with texture support
uniform vec4 ambientColor;
uniform vec4 diffuseColor;
uniform vec4 specularColor;
uniform sampler2D colorMap;

varying vec3 varyingNormal;
varying vec3 varyingLightDir;

float intensity(vec3 u, vec3 v) {
 return max(0.0, dot(normalize(u), normalize(v)));
}

void main(void)
{
 float diff = intensity(varyingNormal, varyingLightDir); // diffuse intensity
 vec4 vFragColor = diff * diffuseColor; // multiply by diffuse color
 vFragColor += ambientColor; // Add in ambient light
 vFragColor *= texture2D(colorMap, gl_TexCoord [0].st); // Modulate in the texture
 // Specular light
 vec3 vReflect = normalize(reflect(-normalize(varyingLightDir), normalize(varyingNormal)));
 float spec = intensity(varyingNormal, vReflect);
 if(diff != 0.0) { // Skip if diffuse light is zero
 float fSpec = pow(spec, 128.0);
 vFragColor.rgb += vec3(fSpec, fSpec, fSpec);
 }
 gl_FragColor = vFragColor;
}
```

# Kombination von Textur und Beleuchtung (Anwendung)

```

def display():
 ...
 glEnableClientState(GL_VERTEX_ARRAY)
 glEnableClientState(GL_NORMAL_ARRAY)
 glEnableClientState(GL_TEXTURE_COORD_ARRAY)
 myVBO.bind()

 glUseProgram(program)
 sendMatrix4(program, "mvMatrix", mvMatrix)
 sendMatrix4(program, "mvpMatrix", mvpMatrix)
 sendMatrix3(program, "normalMatrix", nMatrix)
 sendVec4(program, "diffuseColor", diffCol)
 sendVec4(program, "ambientColor", ambCol)
 sendVec4(program, "specularColor", specCol)
 sendVec3(program, "lightPosition", lightPos)

 glVertexPointer(3, GL_FLOAT, 32, myVBO)
 glNormalPointer(GL_FLOAT, 32, myVBO+12)
 glTexCoordPointer(2, GL_FLOAT, 32, myVBO+24)

 glBindTexture(GL_TEXTURE_2D, textureIDs[tNr])
 glDrawArrays(GL_QUADS, 0, len(data))

 myVBO.unbind()
 glDisableClientState(GL_VERTEX_ARRAY)
 glDisableClientState(GL_NORMAL_ARRAY)
 glDisableClientState(GL_TEXTURE_COORD_ARRAY)
 ...

```

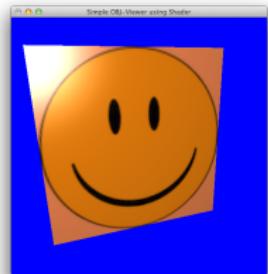
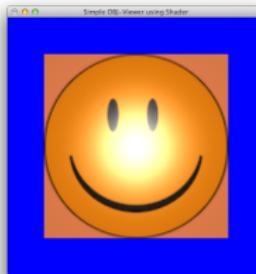
```

...
data = [
 [-1.0, -1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0],
 [1.0, -1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0],
 [1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0],
 [-1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0]]
myVBO = vbo.VBO(array(data, 'f'))

lightPos = [0, 0, 1]
diffCol = [180/255., 100/255., 60/255., 1]
ambCol = [45/255., 25/255., 15/255., 1]
specCol = [90/255., 50/255., 30/255., 1]

...

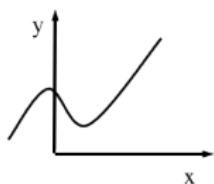
```



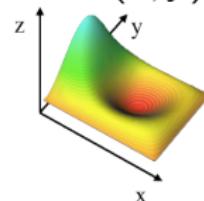
# Explizite Objektrepräsentationen

Objekt-Beschreibung (Darstellung) durch Funktion der Form

- Kurven  $y = f(x)$



- Flächen  $z = f(x, y)$



- Nachteile

- Ein  $y$ -Wert pro  $x$ -Wert (bzw. ein  $z$ -Wert pro  $(x, y)$  Wertepaar)
- Kurven/Flächen mit vertikalen Tangenten/Tangentialebenen schwierig
- Darstellung ist nicht rotationsinvariant

## Punktmengen

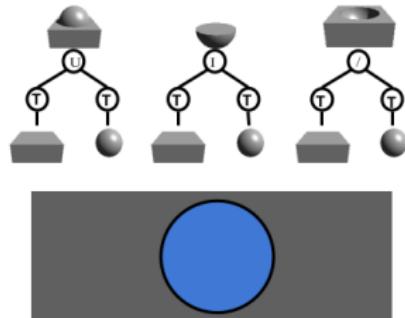
- Detaillierte Flächen haben große Anzahl Primitive
  - Objekt-Oberfläche wird durch (dichte) Punktmenge definiert
- Modellierung schwierig (fehlende Topologieinformationen)



# Implizite Objektrepräsentationen

Objekt-Beschreibung durch Funktionen der Form  $f : \mathbb{R}^{2,3} \rightarrow \mathbb{R}$

- Kurven:  $f(x, y) = 0$       Beispiel:  $K = \{(x, y) | x^2 + y^2 - r^2 = 0\}$
- Flächen:  $f(x, y, z) = 0$       Beispiel:  $S = \{(x, y, z) | x^2 + y^2 + z^2 - r^2 = 0\}$
- Wird häufig in *Constructive Solid Geometry* (*CSG*) Systemen verwendet
- Vorteile
  - Zerlegung des Raumes. Einfache Kriterien für
    - Enthalten:  $f(x, y, z) = 0$
    - Innen:  $f(x, y, z) < 0$
    - Außen:  $f(x, y, z) > 0$
  - Einfache Schnitt- und Abstandsberechnung
- Nachteile
  - Nicht geschlossene Flächen sind schwierig zu beschreiben
  - Problematisch bei „klassischer“ (Render-Pipeline) Echtzeitgrafik



# Parametrische Objektrepräsentationen

## Objekt-Beschreibung durch Funktionen der Form

- Kurven:  $\mathbb{R} \longrightarrow \mathbb{R}^{2,3}$  mit  $t \longrightarrow \mathbf{x}(t) = (x(t), y(t), z(t))^T$ 
  - Die *Richtung der Tangente* der Kurve  $\mathbf{x}(t)$  an der Parameterstelle  $t$  ist gegeben durch die erste Ableitung  $\mathbf{x}'(t)$
  - Eine parametrische Kurve heißt *n-mal stetig differenzierbar*, wenn die Abbildung  $\mathbf{x}(t)$  n-mal stetig differenzierbar ist
  - Je häufiger eine Kurve stetig differenzierbar ist, desto „*glatte*“ ist sie
- Flächen:  $\mathbb{R}^2 \longrightarrow \mathbb{R}^3$  mit  
 $(u, v) \longrightarrow \mathbf{x}(u, v) = (x(u, v), y(u, v), z(u, v))^T$

## Polygon-Netze und Unterteilungsflächen

- Polygon-Netze = Stückweise lineare Interpolation einer Fläche (Punktwolke)
- Unterteilungsflächen (Subdivision-Surfaces)  
entstehen durch Glätten von Polygon-Netzen



# Beispiel: Parameterdarstellung einer Geraden

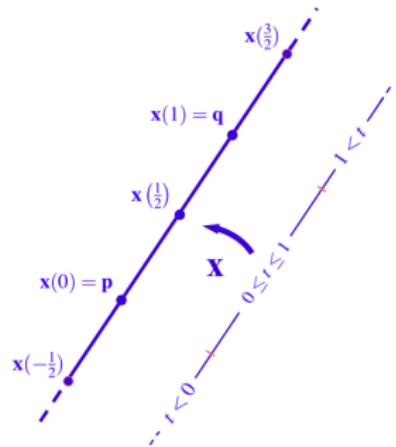
Parametrische Beschreibung einer Geraden

Lineare Interpolation der Punkte  $\mathbf{p}$  und  $\mathbf{q}$ :

$$\begin{aligned}\mathbf{x}(t) &= t\mathbf{q} + (1 - t)\mathbf{p} \\ &= \mathbf{p} + t(\mathbf{q} - \mathbf{p}) \quad \text{mit} \quad t \in \mathbb{R}\end{aligned}$$

$\mathbf{x}(t)$  ist eine lineare Funktion, d.h.

- ein Polynom in  $t$  vom Grad 1 mit
- Punkten  $\mathbf{p}, \mathbf{q}$  als Koeffizienten



**ACHTUNG:** Die Parametrisierung einer Kurve ist nicht eindeutig

Beispiel:  $\mathbf{x}_{1,2} : [0, 1] \longrightarrow \mathbb{R}^2$  mit

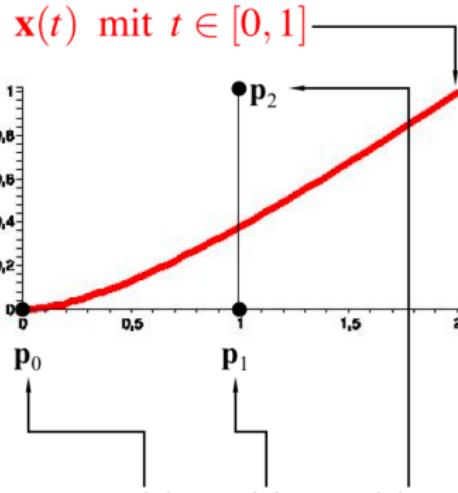
$$\mathbf{x}_1(t) = t\mathbf{q} + (1 - t)\mathbf{p} \quad \text{und} \quad \mathbf{x}_2(t) = t^2\mathbf{q} + (1 - t^2)\mathbf{p}$$

erzeugen die gleiche Kurve (Gerade).

# Monom-Darstellung von Kurven

Parametrische Darstellung von Kurven als Polynom in *Monomdarstellung*

$$\mathbf{x}(t) = \sum_{i=0}^n t^i \mathbf{p}_i$$



$$\mathbf{x}(t) = \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix}t + \begin{pmatrix} 1 \\ 1 \end{pmatrix}t^2$$

## ■ Vorteil

- Effiziente Auswertung mittels *Horner-Schema* ( $n$  Mult. +  $n$  Add.). Beispiel:

$$\begin{aligned}\mathbf{x}(t) &= t^3 \mathbf{p}_3 + t^2 \mathbf{p}_2 + t \mathbf{p}_1 + \mathbf{p}_0 \\ &= t(t(t\mathbf{p}_3 + \mathbf{p}_2) + \mathbf{p}_1) + \mathbf{p}_0\end{aligned}$$

## ■ Nachteil

- Die Koeffizienten (Punkte)  $\mathbf{p}_i$  haben keine geometrische Bedeutung. Sie bestimmen die Ableitungen der Kurve an der Stelle  $t = 0$ :  $\mathbf{p}_0 = \mathbf{x}(0)$ ,  $\mathbf{p}_1 = \mathbf{x}'(0), \dots, \mathbf{p}_k = \frac{1}{k!} \mathbf{x}^{(k)}(0)$

# Lagrange-Darstellung von Kurven



Joseph-Louis Lagrange  
1736 – 1813

Kurve, welche die  $n + 1$ , den Parameterwerten  $t_0 < \dots < t_n$  zugeordneten, Punkte  $\mathbf{p}_0, \dots, \mathbf{p}_n$  interpoliert

$$\mathbf{x}(t) = \sum_{i=0}^n L_i^n(t) \mathbf{p}_i \quad \text{mit} \quad L_i^n(t) = \prod_{\substack{j=0 \\ j \neq i}}^n \frac{t - t_j}{t_i - t_j}$$

$t_0=0, t_1=1, t_2=2, t_3=3$  und

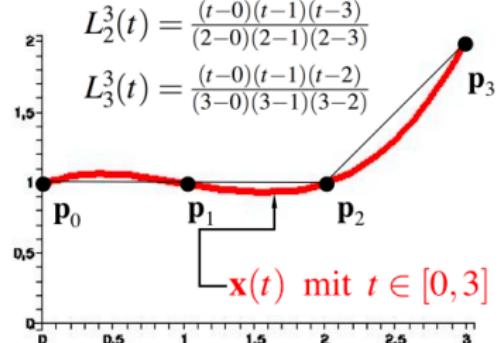
$$\mathbf{p}_0=(0,1), \mathbf{p}_1=(1,1), \mathbf{p}_2=(2,1), \mathbf{p}_3=(3,4)$$

$$L_0^3(t) = \frac{(t-1)(t-2)(t-3)}{(0-1)(0-2)(0-3)}$$

$$L_1^3(t) = \frac{(t-0)(t-2)(t-3)}{(1-0)(1-2)(1-3)}$$

$$L_2^3(t) = \frac{(t-0)(t-1)(t-3)}{(2-0)(2-1)(2-3)}$$

$$L_3^3(t) = \frac{(t-0)(t-1)(t-2)}{(3-0)(3-1)(3-2)}$$



- Vorteil
  - Punkte  $\mathbf{p}_0, \dots, \mathbf{p}_n$  haben geometrische Bedeutung (werden interpoliert)
- Nachteil
  - Ungewollte Schwingungen, speziell bei hohem Polynomgrad



# Hermite-Darstellung von Kurven



Charles Hermite  
1822 – 1901

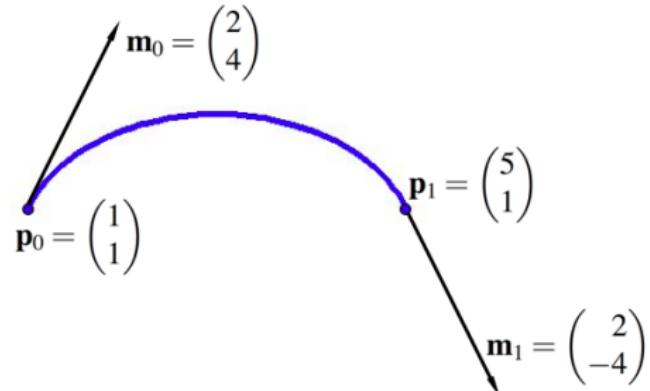
Kurven in Hermite-Darstellung interpolieren neben Punkten auch Ableitungen in Punkten

Die *kubische* Hermite-Kurve

$$\mathbf{x}(t) = H_0^3(t)\mathbf{p}_0 + H_1^3(t)\mathbf{m}_0 + H_2^3(t)\mathbf{m}_1 + H_3^3(t)\mathbf{p}_1$$

über dem (Parameter-)Intervall  $[0, 1]$ , mit den vier Basisfunktionen

$$\begin{aligned}H_0^3(t) &= (1-t)^2(1+2t) \\H_1^3(t) &= t(1-t^2) \\H_2^3(t) &= -t^2(1-t) \\H_3^3(t) &= t^2(3-2t)\end{aligned}$$



interpoliert die beiden Punkte  $\mathbf{p}_0, \mathbf{p}_1$  und die beiden Ableitungen  $\mathbf{m}_0, \mathbf{m}_1$

# Wiederholte lineare Interpolation

Durch wiederholte lineare Interpolation lässt sich aus einem Polygon eine Kurve erzeugen. Beispiel:

- Gegeben:  
 $\mathbf{p}_0 = (0, 0)^T, \mathbf{p}_1 = (1, 0)^T, \mathbf{p}_2 = (1, 1)^T$
- Berechne für beliebiges  $t \in [0, 1]$

$$\mathbf{p}_0^1 = (1 - t)\mathbf{p}_0 + t\mathbf{p}_1$$

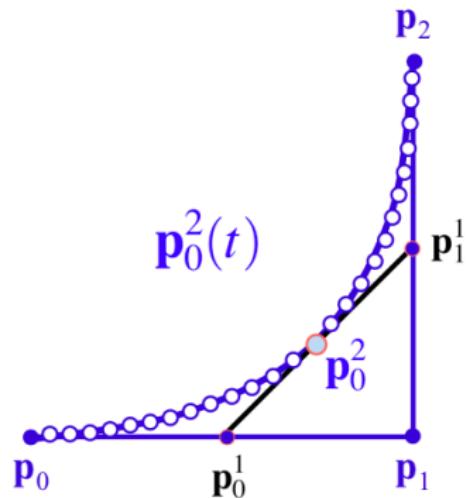
$$\mathbf{p}_1^1 = (1 - t)\mathbf{p}_1 + t\mathbf{p}_2$$

- und anschließend mit dem gleichen  $t$

$$\mathbf{p}_0^2 = (1 - t)\mathbf{p}_0^1 + t\mathbf{p}_1^1$$

- Man erhält

$$\begin{aligned}\mathbf{p}_2^0(t) &= (1 - t)((1 - t)\mathbf{p}_0 + t\mathbf{p}_1) + t((1 - t)\mathbf{p}_1 + t\mathbf{p}_2) \\ &= (1 - t)^2\mathbf{p}_0 + 2(1 - t)t\mathbf{p}_1 + t^2\mathbf{p}_2\end{aligned}$$



# Bézier-Kurven



Pierre Bézier  
1910–1999

Durch wiederholte lineare Interpolation von  $n + 1$  Punkten erhält man eine *Bézier-Kurve* vom (Polynom-)Grad  $n$  in der Form

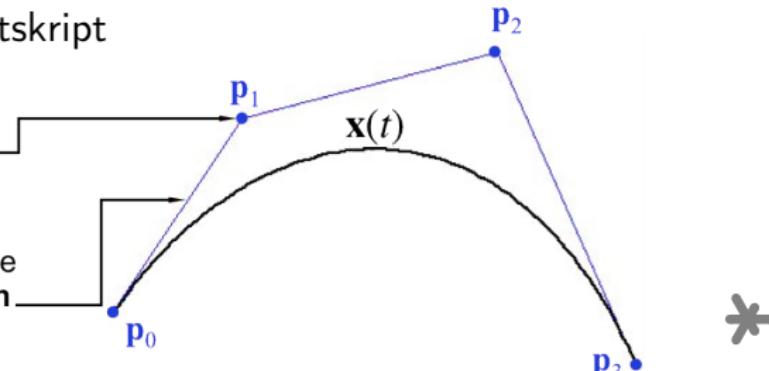
$$\mathbf{p}_0^n(t) = \mathbf{x}(t) = \sum_{i=0}^n B_i^n(t) \mathbf{p}_i, \quad t \in [0, 1]$$

mit den *Bernstein-Polynomen*  $B_i^n(t) = \binom{n}{i} (1-t)^{n-i} t^i$  als Basisfunktionen.

- Bézier-Kurven wurden  $\sim 1960$  von Pierre Bézier bei Renault entwickelt,
- unabhängig davon um 1958 von Paul de Faget de Casteljau bei Citroen
- Sie sind Grundlage von Postskript

Die Punkte  $\mathbf{p}_i$  werden  
**Kontrollpunkte**  
genannt

Das durch die Punkte gegebene  
Polygon heißt **Kontrollpolygon**



# Eigenschaften der Bernstein-Polynome

## ■ Positivität

$$B_i^n(t) \geq 0 \text{ für alle } t \in [0, 1]$$

## ■ Teilung der Eins

$$1 = ((1-t) + t)^n = \sum_{i=0}^n \binom{n}{i} (1-t)^{n-i} t^i$$

## ■ Symmetrie

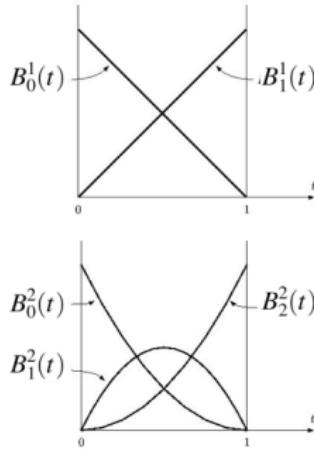
$$B_i^n(t) = B_{n-i}^n(1-t) \text{ für alle } t \in [0, 1]$$

## ■ Maxima

$B_i^n(t)$  hat genau ein Maximum in  $[0, 1]$  (bei  $t = i/n$ )

## ■ Ableitung

$$\begin{aligned} \frac{d}{dt} B_i^n(t) &= \frac{d}{dt} \binom{n}{i} (1-t)^{n-i} t^i \\ &= \frac{i n!}{i!(n-i)!} (1-t)^{n-i} t^{i-1} - \frac{(n-i)n!}{i!(n-i)!} (1-t)^{n-i-1} t^i \\ &= n \left( B_{i-1}^{n-1}(t) - B_i^{n-1}(t) \right) \end{aligned}$$



# Bézier-Kurven über dem Parameterintervall $[a, b]$

Lineare Interpolation zwischen den Punkten  $\mathbf{p}$  und  $\mathbf{q}$  mit  $t \in [a, b]$  liefert

$$\mathbf{x}(t) = \left(1 - \frac{t-a}{b-a}\right) \mathbf{p} + \frac{t-a}{b-a} \mathbf{q} = \frac{b-t}{b-a} \mathbf{q} + \frac{t-a}{b-a} \mathbf{q} \quad \text{und die}$$

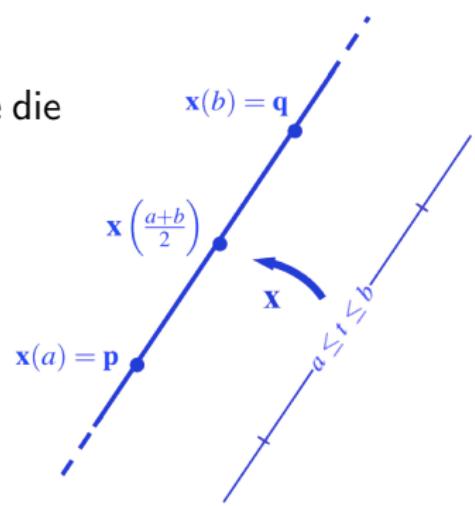
verallgemeinerten Bernstein-Polynome

$$B_i^n_{[a,b]}(t) = \binom{n}{i} \frac{(b-t)^{n-i}(t-a)^i}{(b-a)^n} \quad \text{sowie die}$$

Bézier-Kurve (vom (Polynom-)Grad  $n$ )

$$\mathbf{x}(t) = \sum_{i=0}^n B_i^n_{[a,b]}(t) \mathbf{p}_i$$

über dem Parameterintervall  $[a, b]$



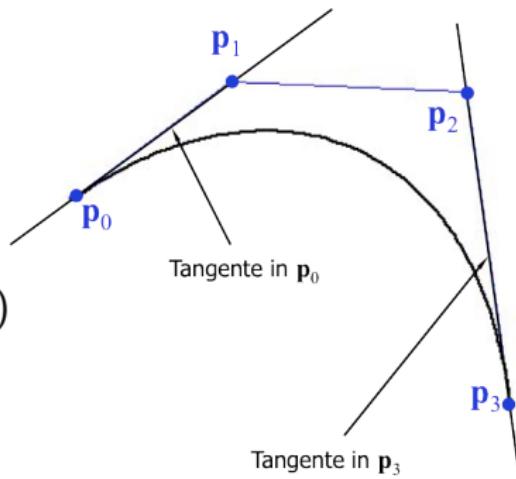
# Die Ableitung einer Bézier-Kurve

Die Ableitung der Bernstein-Polynome  $\frac{d}{dt} B_i^n(t) = n \left( B_{i-1}^{n-1}(t) - B_i^{n-1}(t) \right)$  liefert die Ableitung der Bézier-Kurve  $x(t)$  über dem Parameterintervall  $[a, b]$

$$x'(t) = \frac{n}{b-a} \sum_{i=0}^{n-1} B_i^{n-1}_{[a,b]}(t) (\mathbf{p}_{i+1} - \mathbf{p}_i)$$

mit

- $x(a) = \mathbf{p}_0$  und  $x(b) = \mathbf{p}_n$   
(Interpolation der Rand-Kontrollpunkte)
- $x'(a) = \frac{n}{b-a}(\mathbf{p}_1 - \mathbf{p}_0)$  und  
 $x'(b) = \frac{n}{b-a}(\mathbf{p}_n - \mathbf{p}_{n-1})$   
(Ableitung hängt am Rand nur von den beiden Endpunkten ab)



# Höhere Ableitungen

Für die  $k$ -te Ableitung einer Bézier-Kurve vom Grad  $n$  gilt

$$\frac{d^k}{dt^k} \mathbf{x}(t) = \frac{n!}{(n-k)!(b-a)^k} \sum_{i=0}^{n-k} B_i^{n-k}_{[a,b]}(t) \Delta^k \mathbf{p}_i$$

mit der  $k$ -ten Vorwärtsdifferenz

$$\Delta^k \mathbf{p}_i = \sum_{j=0}^k (-1)^j \binom{k}{j} \mathbf{p}_{i+k-j}$$

Am Kurvenrand gelten für die 0., 1. und 2. Ableitung einer Bézier-Kurve

$$\mathbf{x}(a) = \mathbf{p}_0$$

$$\mathbf{x}(b) = \mathbf{p}_n$$

$$\mathbf{x}'(a) = \frac{n}{b-a} (\mathbf{p}_1 - \mathbf{p}_0)$$

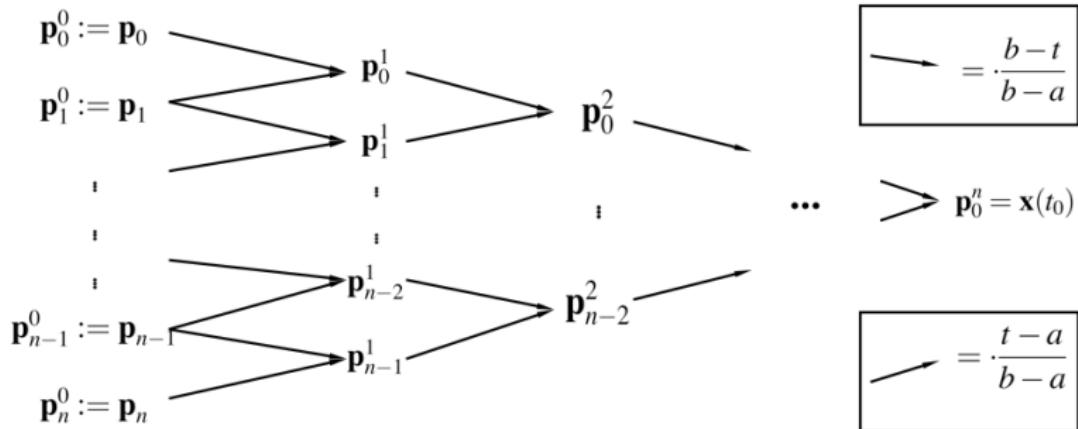
$$\mathbf{x}'(b) = \frac{n}{b-a} (\mathbf{p}_n - \mathbf{p}_{n-1})$$

$$\mathbf{x}''(a) = \frac{n(n-1)}{(b-a)^2} (\mathbf{p}_2 - 2\mathbf{p}_1 + \mathbf{p}_0)$$

$$\mathbf{x}''(b) = \frac{n(n-1)}{(b-a)^2} (\mathbf{p}_n - 2\mathbf{p}_{n-1} + \mathbf{p}_{n-2})$$

# Der Algorithmus von de Casteljau

## Wiederholte lineare Interpolation

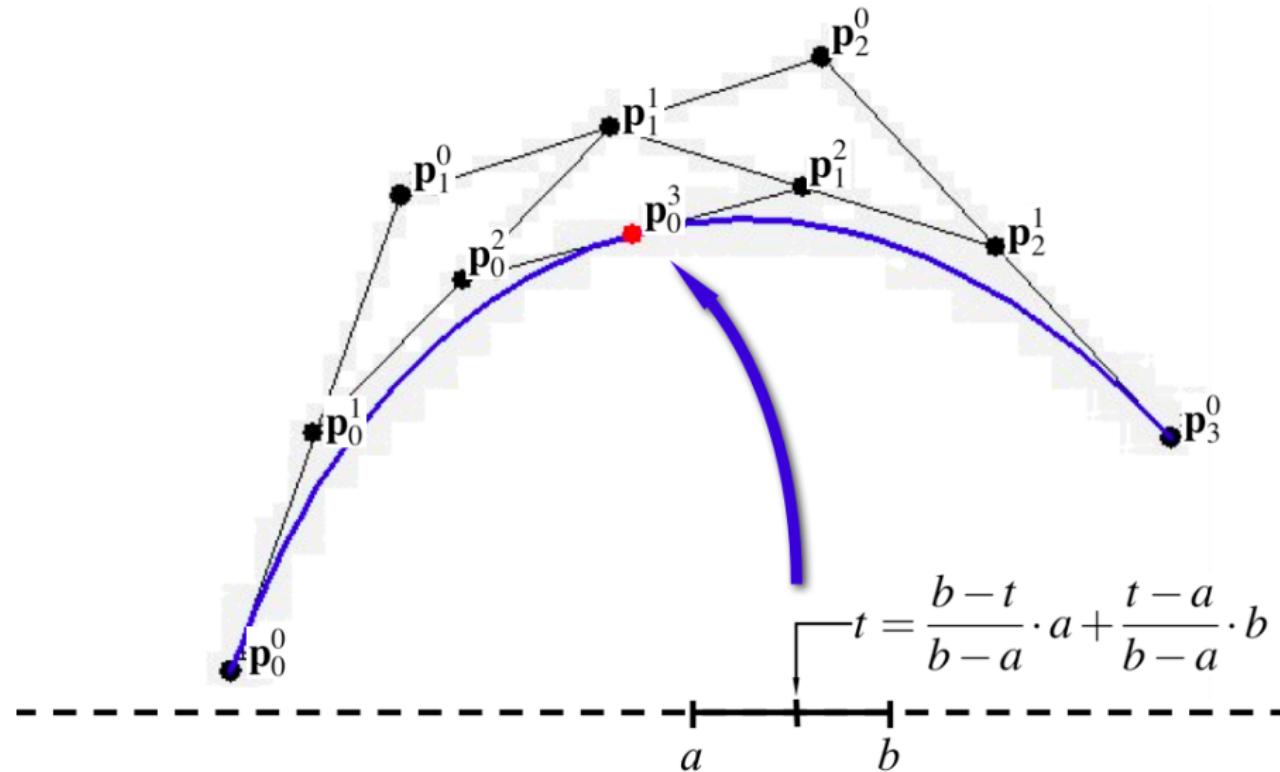


## Rekursionsformel

$$\mathbf{p}_i^0 := \mathbf{p}_i, \quad i = 0, \dots, n$$

$$\mathbf{p}_i^r = \frac{b-t}{b-a} \mathbf{p}_i^{r-1} + \frac{t-a}{b-a} \mathbf{p}_{i+1}^{r-1}, \quad r = 1, \dots, n \quad i = 0, \dots, n-r$$

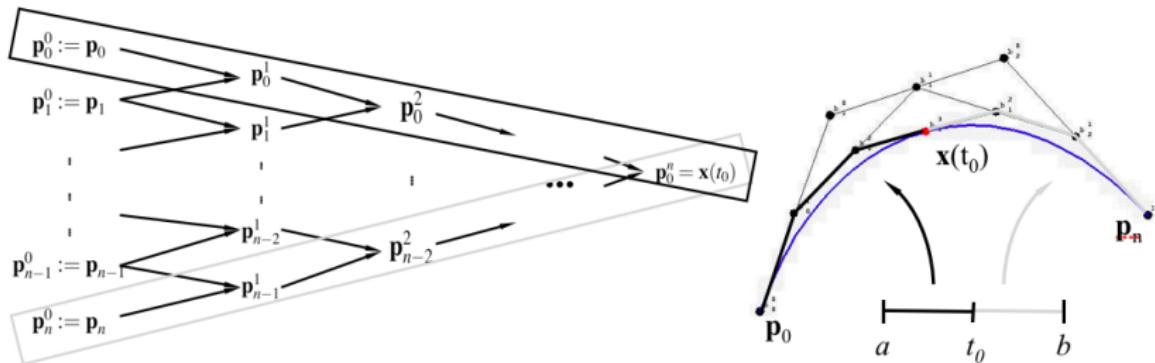
# Der Algorithmus von de Casteljau (geometrisch)



# (Wiederholte) Unterteilung

## Beobachtungen

- Die Kurve von  $\mathbf{p}_0$  bis  $\mathbf{x}(t_0)$  hat die Kontrollpunkte  $\mathbf{p}_0, \mathbf{p}_0^1, \mathbf{p}_0^2, \dots, \mathbf{p}_0^n$
- Die Kurve von  $\mathbf{x}(t_0)$  bis  $\mathbf{p}_n$  hat die Kontrollpunkte  $\mathbf{p}_0^n, \mathbf{p}_1^{n-1}, \mathbf{p}_2^{n-2}, \dots, \mathbf{p}_n$



Bei *wiederholter Unterteilung* (z.B. in der Mitte des Parameterintervalls) konvergieren die Kontrollpolygone gegen die Kurve

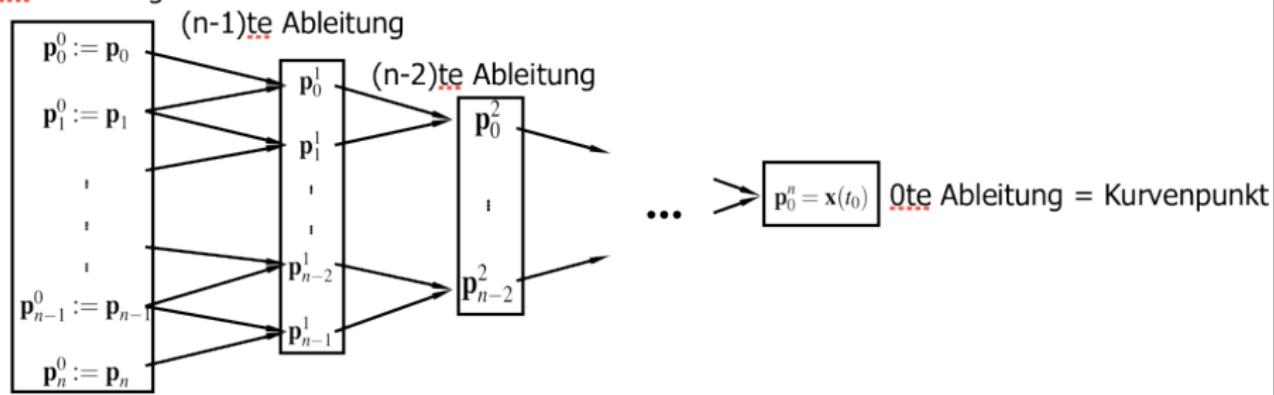
# Der Algorithmus von de Casteljau und Ableitungen

Die  $k$ -te Ableitung einer Bézier-Kurve lässt sich direkt aus den Einträgen der  $(n-k)$ -ten Stufe des de Casteljau Schemas berechnen

$$\frac{d^k}{dt^k} \mathbf{x}(t) = \frac{n!}{(n-k)!(b-a)^k} \Delta^k \mathbf{p}_0^{n-k}, \quad k = 0, \dots, n$$

mit der  $k$ -ten Vorwärtsdifferenz  $\Delta^k \mathbf{p}_0^{n-k} = \sum_{j=0}^k (-1)^j \binom{k}{j} \mathbf{p}_{k-j}^{n-k}$

n-te Ableitung



# Beispiel

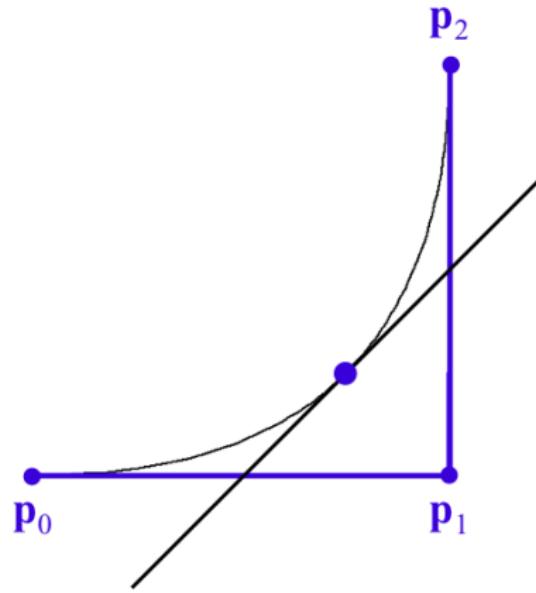
Gegeben sind die Kontrollpunkte  $\mathbf{p}_0 = (0, 0)^T$ ,  $\mathbf{p}_1 = (1, 0)^T$ ,  $\mathbf{p}_2 = (1, 1)^T$

Für den Kurvenpunkt sowie die Ableitung an der Stelle  $t_0 = 0.5$  der Bézier-Kurve

$$\mathbf{x}(t) = B_0^2(t)\mathbf{p}_0 + B_1^2(t)\mathbf{p}_1 + B_2^2(t)\mathbf{p}_2$$

erhält man mit Hilfe des de Casteljau Algorithmus

- $\mathbf{x}(0.5) = \left(\frac{3}{4}, \frac{1}{4}\right)^T$
- $\mathbf{x}'(0.5) = (1, 1)^T$



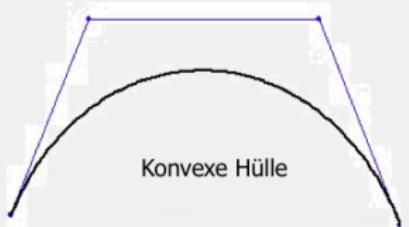
# Eigenschaften von Bézier-Kurven

- **Affine Invarianz**  
(wegen Teilung der 1)
  - Bildet man das Kontrollpolygon mit einer affinen Abbildung ab, wird die gesamte Kurve derselben Abbildung unterworfen
  
- **Variationsverminderung**  
(wegen wiederholter lin. Interpolation)
  - Kurve hat höchstens so viele Wendepunkte wie ihr Kontrollpolygon
  
- **Konvexe-Hüllen-Eigenschaft**  
(wegen positiver Teilung der 1)
  - Kurve liegt innerhalb der konvexen Hülle des Kontrollpolygons
  
- **Geometrische Beziehung** zwischen Ableitungen und Kontrollpunkten

Die Kurve rotiert mit ihrem Kontrollpolygon



Die Kurve kann **keinen Wendepunkt** haben



# Schwächen von Bézier-Kurven (I)

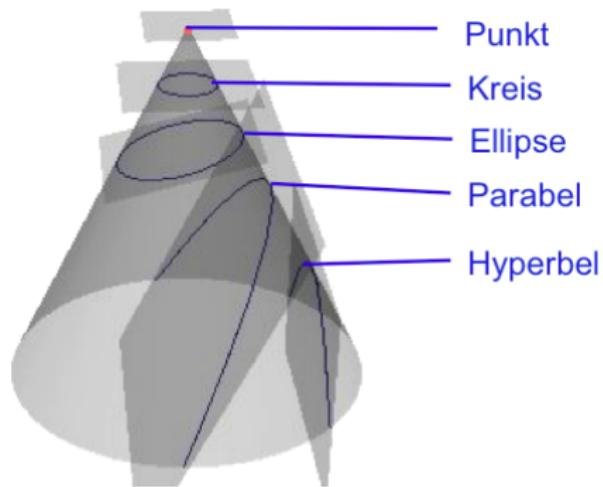
Nicht einmal alle Kegelschnitte lassen sich als Bézier-Kurven darstellen

Darstellbar als Bézier-Kurven:

- Punkt
- Gerade
- Parabel

Nicht darstellbar als Bézier-Kurven:

- Kreis
- Ellipse
- Hyperbel



# Rationale Kurven

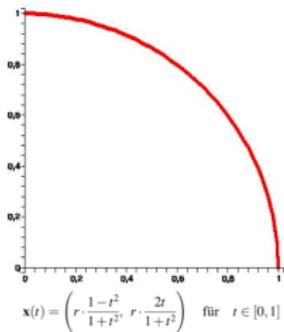
Allgemeine Kegelschnitte lassen sich nicht mit Polynomen, wohl aber mit *rationalen Funktionen* (zum Beispiel in Monomdarstellung)

$$\mathbf{x}(t) = \frac{\sum_{i=0}^n t^i \mathbf{p}_i}{\sum_{i=0}^n t^i w_i}, \quad \mathbf{p}_i \in \mathbb{R}^2, \quad w_i \in \mathbb{R}$$

darstellen

Beispiel: Darstellung eines *Kreis(bogens)* mit Radius  $r$

$$\begin{aligned}\mathbf{x}(t) &= \frac{t^0 \cdot \binom{r}{0} + t^1 \cdot \binom{0}{2r} + t^2 \cdot \binom{-r}{0}}{t^0 \cdot 1 + t^1 \cdot 0 + t^2 \cdot 1} \\ &= \left( r \cdot \frac{1 - t^2}{1 + t^2}, r \cdot \frac{2t}{1 + t^2} \right)^T, \quad t \in \mathbb{R}\end{aligned}$$



# Rationale Kurven in homogener Darstellung

Eine rationale 2D Kurve ist Zentralprojektion (Projektionszentrum **0**) einer herkömmlichen polynomiellen 3D Kurve auf die (projektive) Ebene  $z = 1$

Beispiel: Darstellung eines *Kreis(bogens)* mit Radius  $r$

Die 2D Kurve

$$\mathbf{x}(t) = \begin{pmatrix} r \cdot \frac{1-t^2}{1+t^2} \\ r \cdot \frac{2t}{1+t^2} \end{pmatrix}, \quad t \in \mathbb{R}$$

ist eine Zentralprojektion der (homogenen) 3D Kurve

$$\mathbf{X}(t) = \begin{pmatrix} r \cdot (1 - t^2) \\ r \cdot 2t \\ 1 + t^2 \end{pmatrix}, \quad t \in \mathbb{R}$$

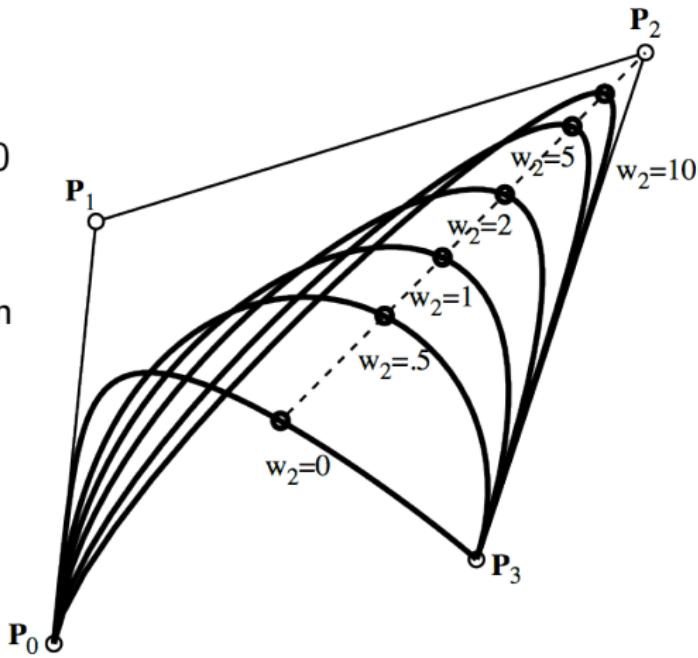


# Rationale Bézier-Kurven

Eine rationale Bézier-Kurve vom Grad  $n$  hat die Gestalt

$$\mathbf{x}(t) = \frac{\sum_{i=0}^n w_i \mathbf{p}_i B_i^n(t)}{\sum_{i=0}^n w_i B_i^n(t)}, w_i \neq 0$$

- Jedem Kontrollpunkt  $\mathbf{p}_i$  ist ein Gewicht  $w_i$  zugeordnet
- ist ein Gewicht gleich Null ( $w_i = 0$ ), so ist im Nenner  $w_i = 0$  und im Zähler anstatt  $w_i \mathbf{p}_i$  nur  $\mathbf{p}_i$  einzusetzen



# Rationale Bézier-Kurven in homogener Darstellung

Rationale Bézier-Kurven lassen sich übersichtlich mit Hilfe homogener Koordinatenvektoren beschreiben

$$\mathbf{X}(t) = \sum_{i=0}^n \mathbf{P}_i B_i^n(t) \text{ mit } \mathbf{P}_i = \begin{cases} \begin{pmatrix} w_i \cdot p_{ix} \\ w_i \cdot p_{iy} \\ w_i \cdot p_{iz} \\ w_i \end{pmatrix} & \text{falls } w_i \neq 0 \\ \begin{pmatrix} p_{ix} \\ p_{iy} \\ p_{iz} \\ 0 \end{pmatrix} & \text{sonst} \end{cases}$$

Auswertung zum Beispiel mittels normalem de Casteljau-Algorithmus<sup>a</sup>

- Aus dem resultierenden homogenen Vektor erhält man durch dehomogenisieren (perspektivisches Teilen) den Kurvenpunkt

---

<sup>a</sup>Oder über Gewichtspunkte. Siehe z.B. G. Farin, *Kurven und Flächen im Computer Aided Geometric Design: Eine praktische Einführung*, Vieweg 1994

# Eigenschaften rationaler Bézier-Kurven

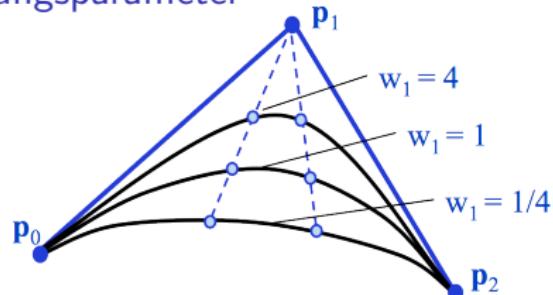
Sind *alle Gewichte*  $w_i$  gleich, so erhält man *polynomielle Bézier-Kurven*

Sind *alle Gewichte*  $w_i$  positiv, so haben rationale Bézier-Kurven alle wichtigen Eigenschaften von polynomiellen Bézier-Kurven

- Affine Invarianz
- Variationsverminderung
- Konvexe Hülle Eigenschaft

Die Gewichte dienen als zusätzliche Gestaltungsparameter

- Wird das Gewicht  $w_i$  des Kontrollpunkts  $\mathbf{p}_i$  erhöht, so wandern alle Kurvenpunkte entlang einer Zentralprojektion in Richtung  $\mathbf{p}_i$



# Beispiel: Kegelschnitte

Die (gebrochen)rationale quadratische Bézier-Kurve

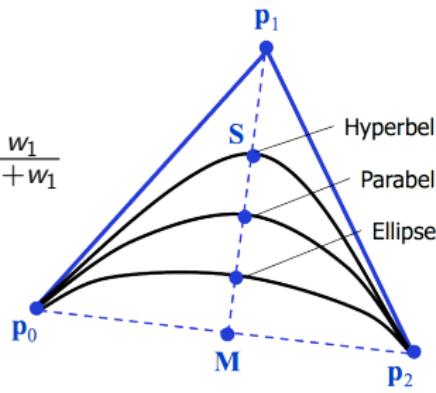
$$\begin{aligned}\mathbf{X}(t) &= \begin{pmatrix} w_0 p_{0x} \\ w_0 p_{0y} \\ w_0 p_{0z} \\ w_0 \end{pmatrix} B_0^2(t) + \begin{pmatrix} w_1 p_{1x} \\ w_1 p_{1y} \\ w_1 p_{1z} \\ w_1 \end{pmatrix} B_1^2(t) + \begin{pmatrix} w_2 p_{2x} \\ w_2 p_{2y} \\ w_2 p_{2z} \\ w_2 \end{pmatrix} B_2^2(t) \\ &= \frac{w_0 \mathbf{p}_0 B_0^2(t) + w_1 \mathbf{p}_1 B_1^2(t) + w_2 \mathbf{p}_2 B_2^2(t)}{w_0 B_0^2(t) + w_1 B_1^2(t) + w_2 B_2^2(t)}\end{aligned}$$

mit

- $w_0 = w_2 = 1$
- Mittelpunkt  $\mathbf{M} = \frac{1}{2}(\mathbf{p}_0 + \mathbf{p}_2)$
- Schulterpunkt  $\mathbf{S} = (1 - s)\mathbf{M} + s\mathbf{p}_1$  mit  $s = \frac{w_1}{1+w_1}$

liefert für

- $s = \frac{1}{2}$  einen *Parabelbogen*
- $s < \frac{1}{2}$  einen *Ellipsenbogen*
- $s > \frac{1}{2}$  einen *Hyperbelbogen*

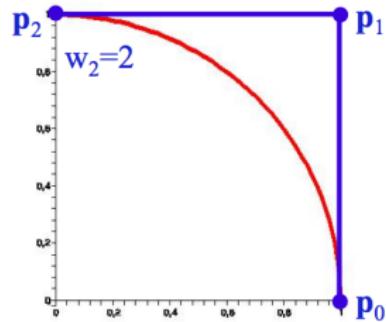


# Beispiel: Kreisdarstellung

Die rationale Parametrisierung des Kreises mit Radius  $r$  von Folie 268 erhält man mit den Gewichten  $w_0 = w_1 = 1$ ,  $w_2 = 2$  und den Kontrollpunkten

$$\mathbf{p}_0 = \begin{pmatrix} r \\ 0 \end{pmatrix}, \quad \mathbf{p}_1 = \begin{pmatrix} r \\ r \end{pmatrix}, \quad \mathbf{p}_2 = \begin{pmatrix} 0 \\ r \end{pmatrix}$$

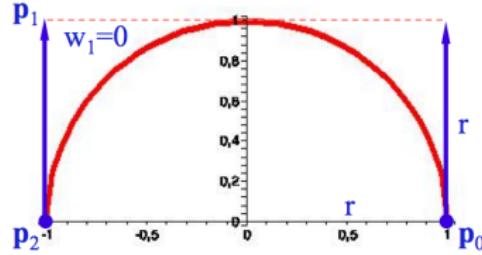
Nebenstehende Kurve entsteht für  $t \in [0, 1]$



Ein weitere rationale Parametrisierung des Kreises mit Radius  $r$  erhält man mit den Gewichten  $w_0 = w_2 = 1$ ,  $w_1 = 0$  und den Kontrollpunkten  $\mathbf{p}_0 = (r, 0)^T$ ,  $\mathbf{p}_1 = (0, r)^T$  (Fernpunkt) und  $\mathbf{p}_2 = (-r, 0)$ :

$$\mathbf{x}(t) = \left( \frac{r(1-t)^2 - rt^2}{(1-t)^2 + t^2}, \frac{2r(1-t)t}{(1-t)^2 + t^2} \right)^T$$

Nebenstehende Kurve entsteht für  $t \in [0, 1]$



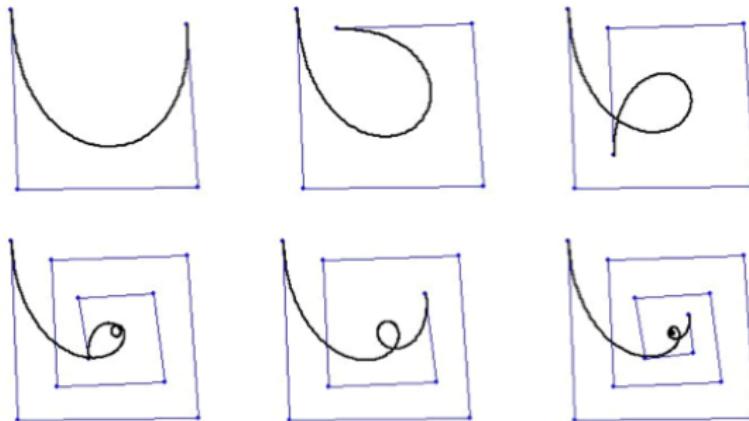
## Schwächen von Bézier-Kurven (II)

Bei komplexer Kurvengestalt muss die zugehörige Bézier-Kurve einen sehr hohen Polynomgrad haben

- In der Praxis kein Polynomgrad größer als 10 (üblicherweise 3)

Keine „lokale“ Kontrolle der Kurve

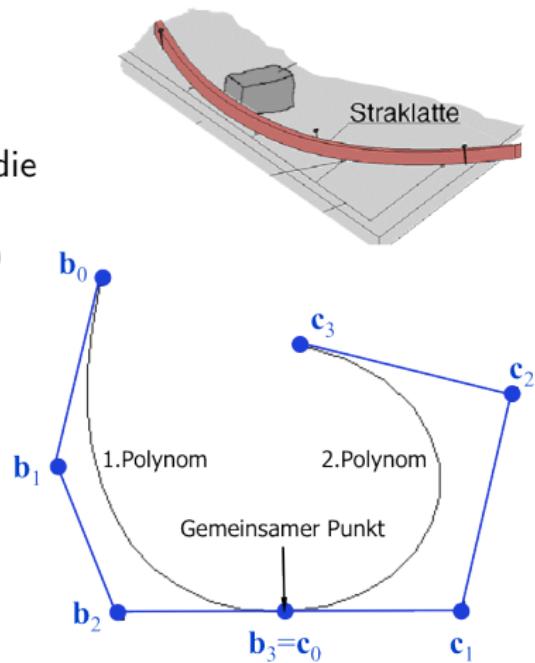
- Veränderung eines Kontrollpunkts verändert die gesamte Kurve



# Spline / Straklatte

Ein *Spline* vom Grad  $n$  ist eine stetige Kurve, die stückweise aus Polynomen vom Grad  $n$  zusammengesetzt ist

- Der Begriff Spline stammt aus dem Schiffsbau
  - Eine lange dünne Latte (Straklatte), die an einzelnen Punkten fixiert ist, biegt sich wie ein kubischer Spline (Grad 3)
- An den Punkten, an denen zwei Polynome zusammen treffen können verschiedene Bedingungen für den Übergang (die Glattheit) gelten
  - Stetig
  - Differenzierbar
  - Stetig differenzierbar
  - ...



# Splinekurven in Bézier-Form

- Stückweise aus  $N + 1$  Bézier-Kurven zusammengesetzte Kurve, definiert über den Parameterintervallen  $[t_0, t_1], [t_1, t_2], \dots, [t_N, t_{N+1}]$
- Kurvensegmente haben alle den gleichen (Polynom-)Grad, d.h.

$$\mathbf{x}_j(t) = \sum_{i=0}^n B_{i[t_j, t_{j+1}]}^n(t) \mathbf{p}_{i,j}, \quad j = 0, \dots, N$$

- In der Praxis meist kubische Bézier-Kurven (d.h.  $n=3$ )
- In den Anschlusspunkten  $\mathbf{x}_j(t_{j+1}), j = 0, \dots, N - 1$  ist die Kurve  $k$ -mal stetig differenzierbar (man spricht von  $C^k$ -Stetigkeit)
  - In der Praxis meist
    - $k = 1$ : Die ersten Ableitungen der Kurvensegmente stimmen überein ( $C^1$ -Stetigkeit)
    - $k = 2$ : Die zweiten Ableitungen der Kurvensegmente stimmen überein ( $C^2$ -Stetigkeit)

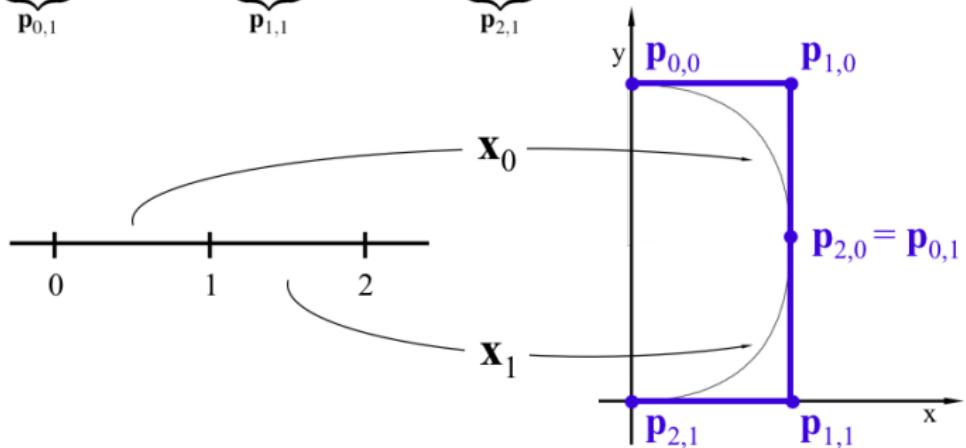


# Beispiel

Bézier-Splinekurve mit den beiden quadratischen Bézier-Kurven

$$\mathbf{x}_0(t) = B_{0[0,1]}^2(t) \underbrace{\binom{0}{2}_{\mathbf{p}_{0,0}}} + B_{1[0,1]}^2(t) \underbrace{\binom{1}{2}_{\mathbf{p}_{1,0}}} + B_{2[0,1]}^2(t) \underbrace{\binom{1}{1}_{\mathbf{p}_{2,0}}}, \quad t \in [0, 1]$$

$$\mathbf{x}_1(t) = B_{0[1,2]}^2(t) \underbrace{\binom{1}{1}_{\mathbf{p}_{0,1}}} + B_{1[1,2]}^2(t) \underbrace{\binom{1}{0}_{\mathbf{p}_{1,1}}} + B_{2[1,2]}^2(t) \underbrace{\binom{0}{0}_{\mathbf{p}_{2,1}}}, \quad t \in [1, 2]$$



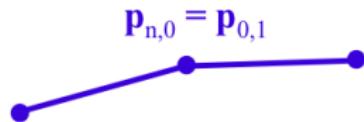
# $C^0$ und $C^1$ Übergangsbedingungen

Zwei Bézier-Kurven  $\mathbf{x}_0(t) = \sum_{i=0}^n B_{i[t_0, t_1]}^n(t) \mathbf{p}_{i,0}$ ,  $\mathbf{x}_1(t) = \sum_{i=0}^n B_{i[t_1, t_2]}^n(t) \mathbf{p}_{i,1}$

sind im Punkt  $\mathbf{x}_0(t_1) = \mathbf{x}_1(t_1)$

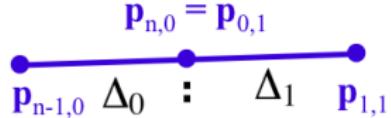
- *stetig ( $C^0$ -stetig)*, wenn gilt

$$\mathbf{p}_{n,0} = \mathbf{p}_{0,1}$$



- *stetig differenzierbar ( $C^1$ -stetig)*, wenn  $\frac{n}{\Delta_0}(\mathbf{p}_{n,0} - \mathbf{p}_{n-1,0}) = \frac{n}{\Delta_1}(\mathbf{p}_{1,1} - \mathbf{p}_{0,1})$  mit  $\Delta_i = t_{i+1} - t_i$ ,  $i = 0, 1$ , was sich mit  $\mathbf{p}_{n,0} = \mathbf{p}_{0,1}$  umformen lässt zu

$$\mathbf{p}_{n,0} = \frac{\Delta_0}{\Delta_0 + \Delta_1} \cdot \mathbf{p}_{1,1} + \frac{\Delta_1}{\Delta_0 + \Delta_1} \cdot \mathbf{p}_{n-1,0}$$



# $C^2$ Übergangsbedingungen

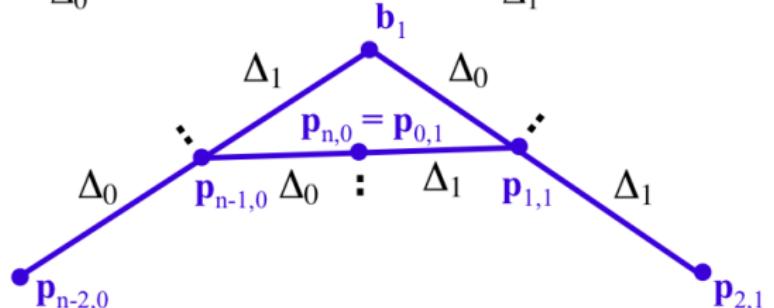
Zwei Bézier-Kurven  $\mathbf{x}_0(t) = \sum_{i=0}^n B_{i[t_0, t_1]}^n(t) \mathbf{p}_{i,0}$ ,  $\mathbf{x}_1(t) = \sum_{i=0}^n B_{i[t_1, t_2]}^n(t) \mathbf{p}_{i,1}$

sind im Punkt  $\mathbf{x}_0(t_1) = \mathbf{x}_1(t_1)$  2-mal stetig differenzierbar ( $C^2$ -stetig), wenn

$$\frac{n(n-1)}{\Delta_0^2} (\mathbf{p}_{n,0} - 2\mathbf{p}_{n-1,0} + \mathbf{p}_{n-2,0}) = \frac{n(n-1)}{\Delta_1^2} (\mathbf{p}_{2,1} - 2\mathbf{p}_{1,1} + \mathbf{p}_{0,1})$$

Mit  $\mathbf{p}_{n,0} = \mathbf{p}_{0,1}$  und der  $C^1$ -Bedingung folgt hieraus

$$\mathbf{p}_{n-1,0} + \frac{\Delta_1}{\Delta_0} (\mathbf{p}_{n-1,0} - \mathbf{p}_{n-2,0}) = \mathbf{p}_{1,1} + \frac{\Delta_0}{\Delta_1} (\mathbf{p}_{1,1} - \mathbf{p}_{2,1}) =: \mathbf{b}_1$$



# Beispiel

Über dem Parameterintervall  $[0, 1]$  ist die folgende Bézier-Kurve gegeben

$$\mathbf{x}_1(t) = B_0^3(t) \binom{0}{0} + B_1^3(t) \binom{0}{1} + B_2^3(t) \binom{1}{2} + B_3^3(t) \binom{2}{2}$$

Wie lauten die Kontrollpunkte  $\mathbf{p}_{i,2}$  einer zweiten, im Punkt  $(2, 2)^T$   $C^2$ –stetig anschließenden Bézier-Kurve  $\mathbf{x}_2$  über dem Parameterintervall  $[1, 2]$ ?

- **$C^0$ -Stetigkeit:**

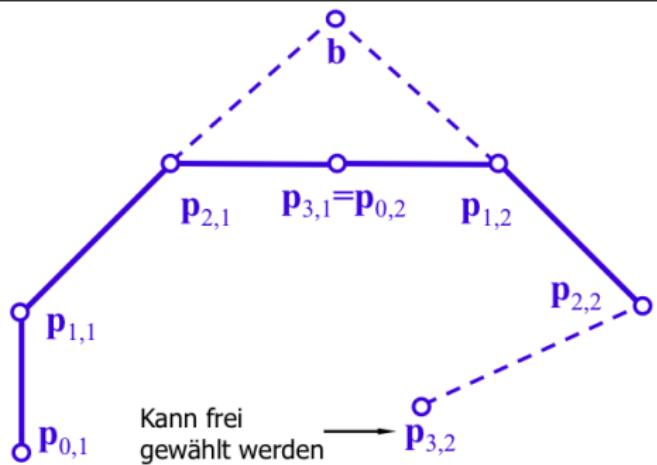
$$\mathbf{p}_{0,2} = (2, 2)^T$$

- **$C^1$ -Stetigkeit:**

$$\begin{aligned}\mathbf{p}_{1,2} &= 2\mathbf{p}_{3,1} - \mathbf{p}_{2,1} \\ &= 2(2, 2)^T - (1, 2)^T = (3, 2)^T\end{aligned}$$

- **$C^2$ -Stetigkeit:**

$$\begin{aligned}\mathbf{p}_{2,2} &= \mathbf{p}_{1,1} + 2(\mathbf{p}_{1,2} - \mathbf{p}_{2,1}) \\ &= (0, 1)^T + 2((3, 2)^T - (1, 2)^T) \\ &= (4, 1)^T\end{aligned}$$



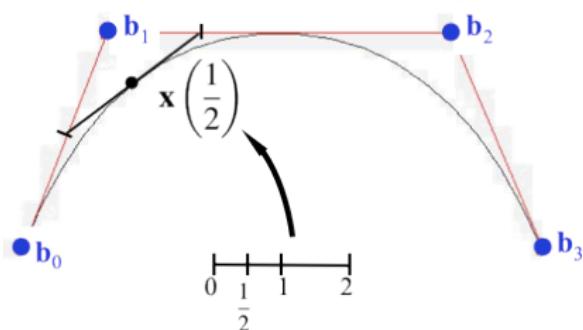
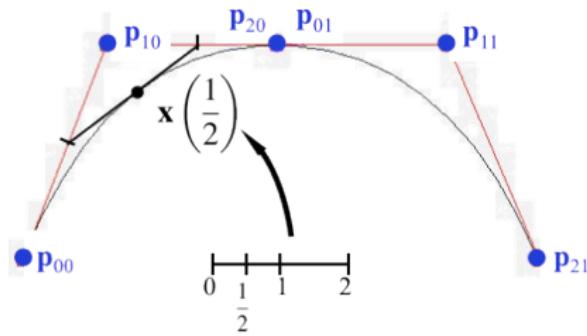
# Redundante Bézier-Spline-Kontrollpunkte (I)

Beobachtung:

Bei einer  $C^1$ -stetigen quadratischen Bézier-Spline-Kurve mit Kurvenstücken

$$\mathbf{x}_0(t) = \sum_{i=0}^2 B_i^2(t) \mathbf{p}_{i0}, \quad t \in [0, 1] \quad \text{und} \quad \mathbf{x}_1(t) = \sum_{i=0}^2 B_i^2(t) \mathbf{p}_{i1}, \quad t \in [1, 2]$$

sind die Kontrollpunkte  $\mathbf{p}_{20} = \mathbf{p}_{01}$  redundant.



Die gleiche Kurve kann daher anstatt durch  $\mathbf{p}_{00}, \mathbf{p}_{10}, \mathbf{p}_{20}, \mathbf{p}_{01}, \mathbf{p}_{11}, \mathbf{p}_{21}$  auch durch die Kontrollpunkte  $\mathbf{b}_0, \mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3$  beschrieben werden.

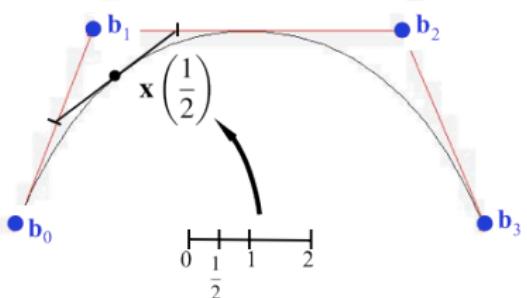
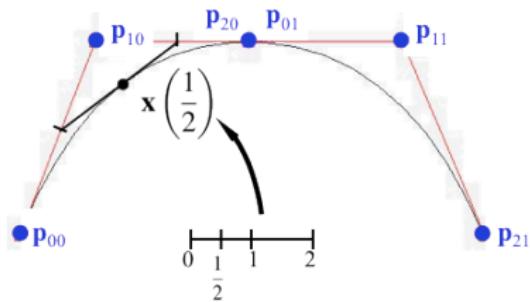
# Redundante Bézier-Spline-Kontrollpunkte (II)

Fragen:

- Kann eine Bézier-Spline-Kurve mit Hilfe von Basis-Funktionen  $N_i^k(t)$  in der Form

$$\mathbf{x}(t) = \sum_{i=0}^n N_i^k(t) \mathbf{b}_i$$

dargestellt werden?



- Gibt es einen de Casteljau ähnlichen Algorithmus zur Berechnung von Kurvenpunkten?

# Splinekurven in B-Spline-Form (B-Spline-Kurven)

B-Spline-Kurven (das „B“ steht für „Basis“) sind eine *komakte Bechreibung* von Spline-Kurven (stückweise zusammengesetzte Kurven)

- Die Parameterintervalle  $[t_0, t_1], [t_1, t_2], \dots, [t_N, t_{N+1}]$  werden beschrieben durch einen (*schwach monotonen*) Knotenvektor
- $$T = (t_0, t_1, \dots, t_{n-1}, t_n, t_{n+1}, \dots, t_{n+k}) \text{ mit } t_i \leq t_{i+1} \forall i \text{ und } t_1 < t_{n+k}$$
- Die Bernstein-Polynome  $B_{i_{[a,b]}}^n(t) = \binom{n}{i} \frac{(b-t)^{n-i}(t-a)^i}{(b-a)^n}$  werden durch *B-Spline-Funktionen*  $N_i^k(t)$  der *Ordnung k bzw. vom Grad k – 1* als Basis-Funktionen ersetzt

Eine B-Spline-Kurve ist somit festgelegt durch

- 1 einen (*schwach monotonen*) Knotenvektor  $T = (t_0, \dots, t_{n+k})$  und
- 2 die Ordnung  $k$  (definieren  $n$  Basis-Funktionen  $N_i^k(t)$  vom Grad  $k – 1$ )
- 3 sowie  $n + 1$  Kontrollpunkte  $\mathbf{b}_0, \dots, \mathbf{b}_n$

# (Normalisierte) B-Spline-Funktionen

Die (normalisierten) B-Spline-Funktionen  $N_i^k(t)$  der Ordnung  $k$  (vom Grad  $k - 1$ ) über einem schwach monotonen Knotenvektor  $T = (t_0, \dots, t_{n+k})$  sind rekursiv definiert (nach Cox und de Boor)

Für  $k = 1$  setzt man

$$N_i^1(t) = \begin{cases} 1 & \text{für } t_i \leq t < t_{i+1} \\ 0 & \text{sonst} \end{cases}$$

und für  $k > 1$

$$N_i^k(t) = \frac{t - t_i}{t_{i+k-1} - t_i} N_i^{k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} N_{i+1}^{k-1}(t), \quad i = 0, \dots, n$$

Dabei wird definiert

$$\frac{0}{0} := 0$$

# Beispiel für B-Spline-Funktionen

$T = (0, 1, 2, 3)$  und  $k = 2$  (Polynomgrad 1)

Aus  $k = 2$  und  $n + k = 3$  folgt  $n = 1$  und

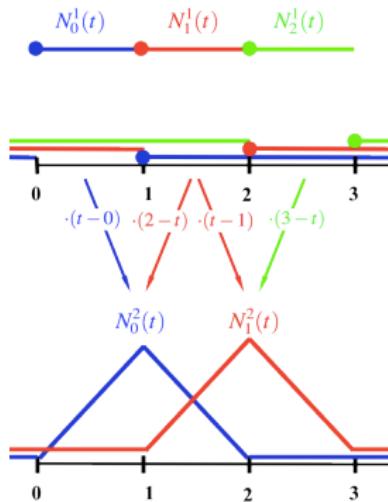
$$N_0^1(t) = \begin{cases} 1 & \text{für } 0 \leq t < 1 \\ 0 & \text{sonst} \end{cases}$$

$$N_1^1(t) = \begin{cases} 1 & \text{für } 1 \leq t < 2 \\ 0 & \text{sonst} \end{cases}$$

$$N_2^1(t) = \begin{cases} 1 & \text{für } 2 \leq t < 3 \\ 0 & \text{sonst} \end{cases}$$

$$N_0^2(t) = tN_0^1(t) + (2-t)N_1^1(t) = \begin{cases} t & \text{für } 0 \leq t < 1 \\ (2-t) & \text{für } 1 \leq t < 2 \\ 0 & \text{sonst} \end{cases}$$

$$N_1^2(t) = (t-1)N_1^1(t) + (3-t)N_2^1(t) = \begin{cases} (t-1) & \text{für } 1 \leq t < 2 \\ (3-t) & \text{für } 2 \leq t < 3 \\ 0 & \text{sonst} \end{cases}$$



# Eigenschaften der B-Spline-Funktionen

Die B-Spline-Funktionen besitzen ähnliche Eigenschaften wie die Bernstein-Polynome. Die wichtigsten sind

- *Positivität und lokaler Träger:*

$$N_i^k(t) \geq 0 \quad \forall t \in [t_0, t_{n+k}]$$

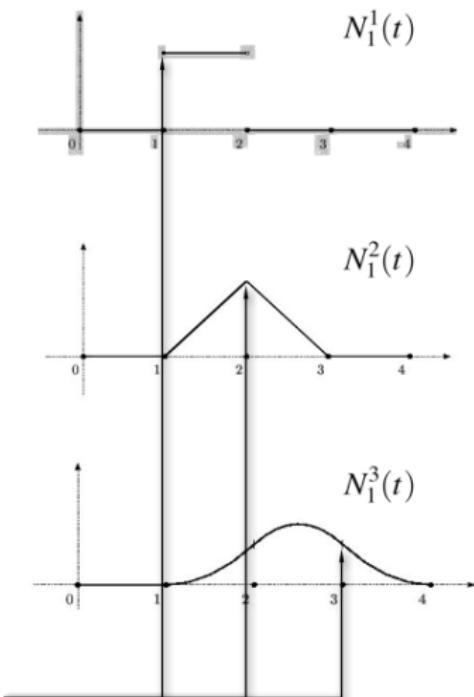
$$N_i^k(t) = 0 \quad \forall t \notin [t_i, t_{i+k}]$$

- *Teilung der Eins:*

$$\sum_{i=0}^n N_i^k(t) = 1 \quad \forall t \in [t_{k-1}, t_{n+1}]$$

- *Glattheit:*

Für einfache Knoten ( $t_{i-1} \neq t_i \neq t_{i+1}$ ) ist  $N_i^k(t)$  an den Knoten  $t_i, \dots, t_{i+k}$  jeweils  $C^{k-2}$ -stetig

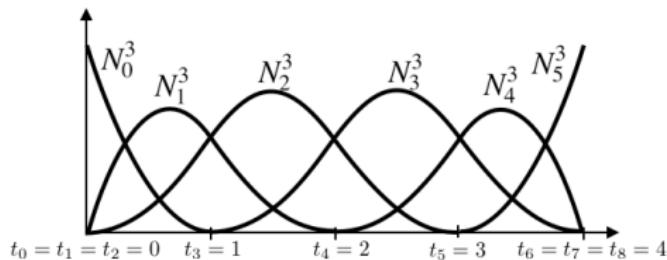
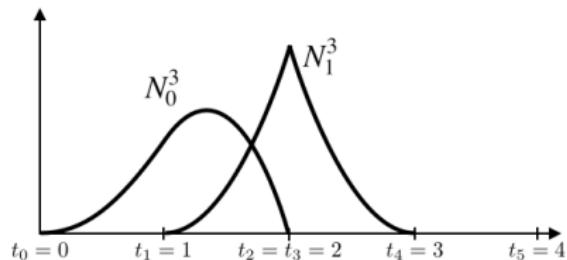


# Mehrfachknoten

Im Knotenvektor  $T = (t_0, \dots, t_{n+k})$  können Knoten mehrfach vorkommen

Haben / Knoten den gleichen Wert, d.h.  $t_i = \dots = t_{i+(l-1)}$ , so reduziert sich die Stetigkeit der B-Spline-Funktionen  $N_i^k(t)$  an dieser Stelle von  $C^{k-2}$  auf  $C^{k-(l-1)}$ .

## Beispiele



# Beispiel für B-Spline-Funktionen mit Mehrfachknoten

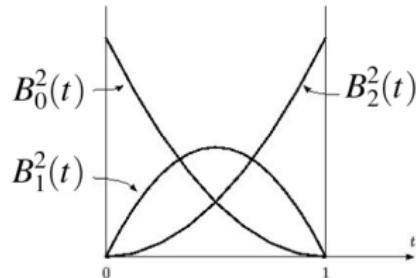
$T = (0, 0, 0, 1, 1, 1)$  und  $k = 3$  (Polynomgrad 2)

Aus  $k = 3$  und  $n + k = 5$  folgt  $n = 2$  und

$$N_0^3(t) = \begin{cases} (1-t)^2 = B_0^2(t) & \text{für } 0 \leq t < 1 \\ 0 & \text{sonst} \end{cases}$$

$$N_1^3(t) = \begin{cases} 2t(1-t) = B_1^2(t) & \text{für } 0 \leq t < 1 \\ 0 & \text{sonst} \end{cases}$$

$$N_2^3(t) = \begin{cases} t^2 = B_2^2(t) & \text{für } 0 \leq t < 1 \\ 0 & \text{sonst} \end{cases}$$



Allgemein gilt:

Die B-Spline-Funktionen  $N_i^k(t)$  der Ordnung  $k$  über dem Knotenvektor  $T = (\underbrace{0, \dots, 0}_{k-\text{mal}}, \underbrace{1, \dots, 1}_{k-\text{mal}})$  entsprechen den Bernstein-Polynomen  $B_i^{k-1}(t)$

# B-Spline-Kurven

Gegeben seien

- 1 ein (schwach monotoner) Knotenvektor  $T = (t_0, \dots, t_{n+k})$
- 2 die Kontrollpunkte (de-Boor-Punkte)  $\mathbf{b}_0, \dots, \mathbf{b}_n$  ( $n \geq k - 1$ )

Die Kurve

$$\mathbf{x}(t) = \sum_{i=0}^n \mathbf{b}_i N_i^k(t), \quad t \in [t_{k-1}, t_{n+1}]$$

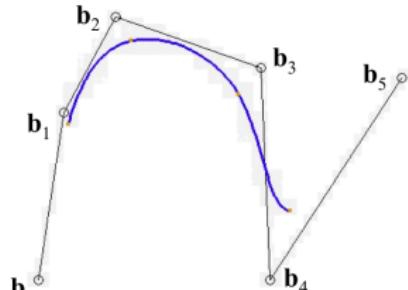
heißt *B-Spline-Kurve der Ordnung k über dem Knotenvektor (Träger) T*

Beispiel:

Knotenvektor  $T = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$

und Kontrollpunkte  $\mathbf{b}_0, \dots, \mathbf{b}_5$  liefern

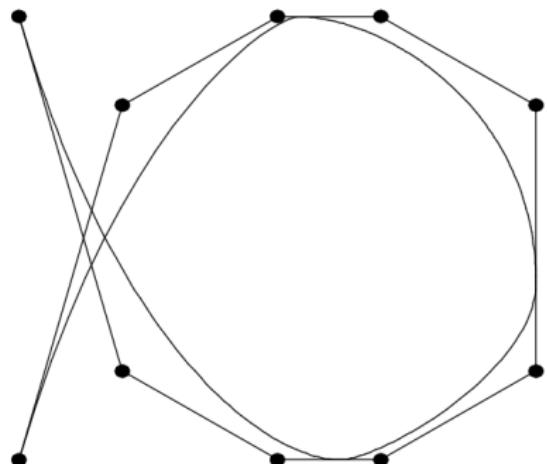
B-Spline-Kurve der Ordnung  $k = 4$  über  
dem Parameterintervall  $[3, 6]$  (jeweils eine  
kubische Kurve über  $[3, 4], [4, 5]$  und  $[5, 6]$ )



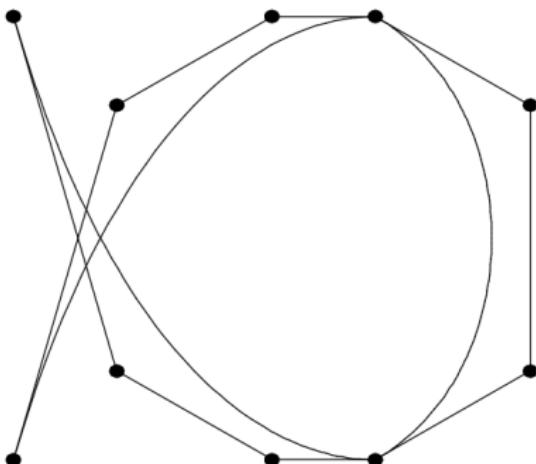
# Eigenschaften von B-Spline-Kurven (I)

Hat der Knotenvektor einer B-Spline-Kurve der Ordnung  $k$

- $k$ -fache Randknoten, so interpoliert die B-Spline-Kurve den ersten und letzten Kontrollpunkt
- Mehrfachknoten im Inneren, so wird die B-Spline-Kurve „weniger glatt“



$$T = (0,0,0,0,1,1,2,2,3,3,4,4,4,4)$$

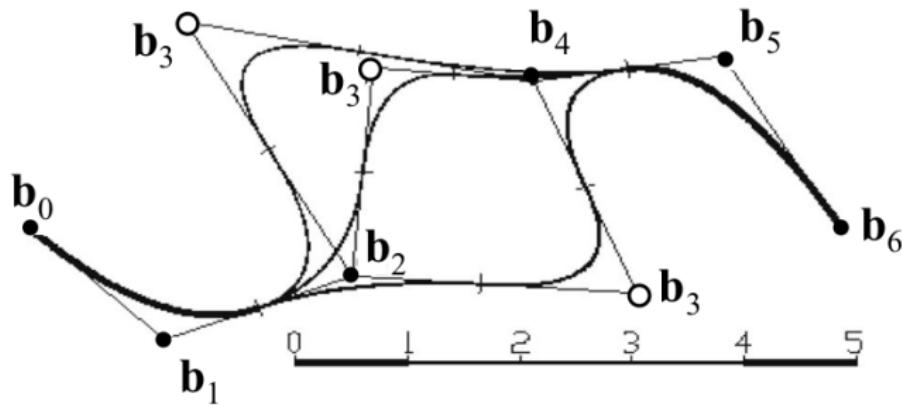


$$T = (0,0,0,0,1,1,1,2,2,2,3,3,3,3)$$

# Eigenschaften von B-Spline-Kurven (II)

Die Kontrollpunkte einer B-Spline-Kurve kontrollieren den lokalen Verlauf der Kurve

- Da  $N_i^k(t) = 0$  für  $t \notin [t_i, t_{i+k}]$  beeinflusst der Kontrollpunkt  $\mathbf{b}_i$  den Kurvenverlauf nur über dem Parameterbereich  $[t_i, t_{i+k}]$
- Umgekehrt wird die Form der Kurve über dem Parameterintervall  $[t_i, t_{i+1}]$  nur von den Kontrollpunkten  $\mathbf{b}_{i-(k-1)}, \dots, \mathbf{b}_i$  beeinflusst



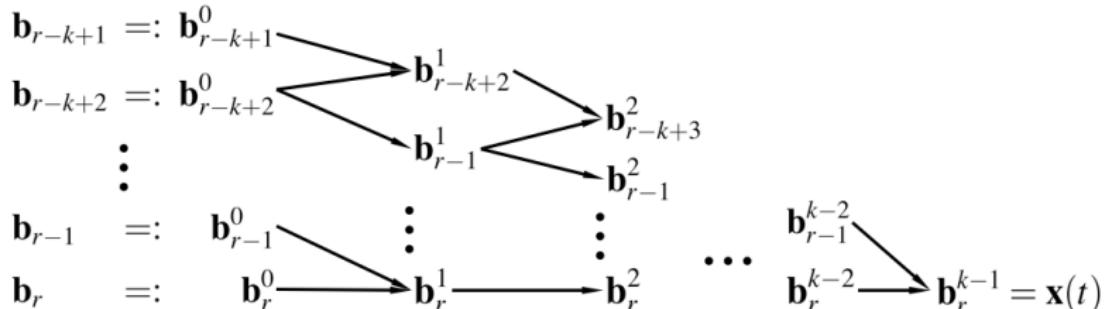
# Der de Boor-Algorithmus

Der de Boor-Algorithmus dient dem Auswerten einer B-Spline-Kurve am Parameterwert  $t$  (Verallgemeinerung des de Casteljau Algorithmus)

- 1 Finde  $r$  mit  $t \in [t_r, t_{r+1})$
  - 2 Für  $j = 1, \dots, k-1, i = r, \dots, r-k+1+j$  berechne

$$\mathbf{b}_i^j = (1 - \alpha_i^j) \mathbf{b}_{i-1}^{j-1} + \alpha_i^j \mathbf{b}_i^{j-1} \quad \text{mit} \quad \alpha_i^j = \frac{t - t_i}{t_{i-j+k} - t_i} \quad \text{und} \quad \mathbf{b}_i^0 := \mathbf{b}_j$$

Dann gilt  $\mathbf{x}(t) = \mathbf{b}_k^{k-1}$ . Rechenschema:



# Geometrische Interpretation des de Boor-Algorithmus

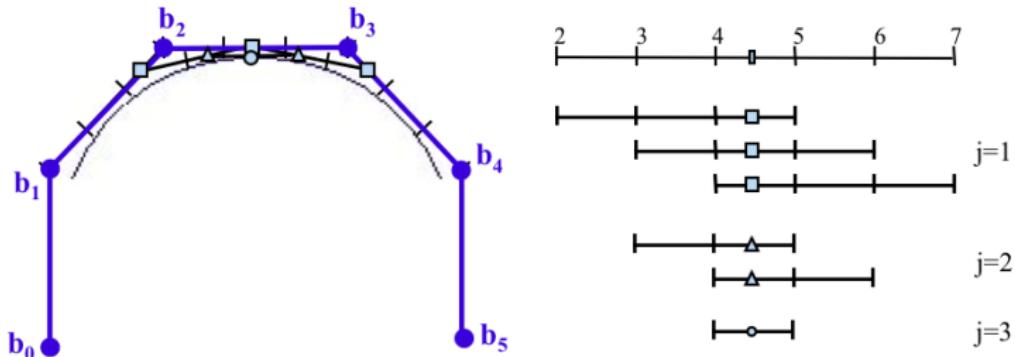
Der de Boor-Algorithmus besteht aus fortgesetzter linearer Interpolation mit unterschiedlichen Teilverhältnissen

Die Teilpunkte auf den Seiten des Kontrollpolygons stehen im gleichen Verhältnis wie der Parameterwert die Knoten im Knotenvektor teilt

Beispiel: Ermittle  $\mathbf{x}(t^*)$  mit  $t^* = 4.5$  der B-Spline-Kurve

$$\mathbf{x}(t) = \sum_{i=0}^5 \mathbf{b}_i N_i^4(t) \text{ über dem Knotenvektor } T = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$$

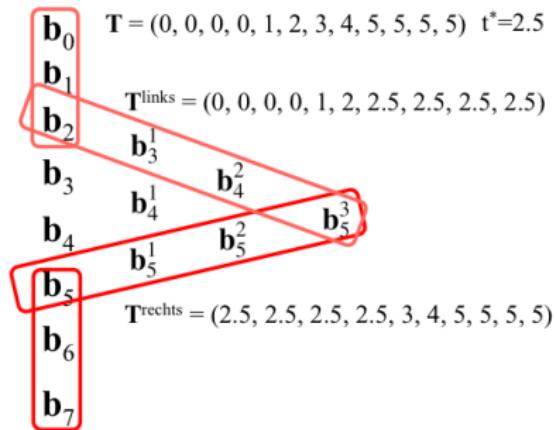
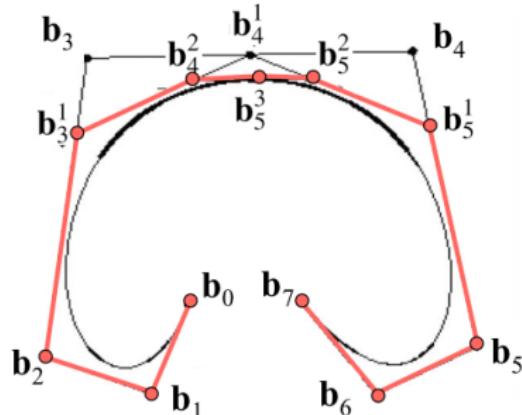
$t^* \in [t_4, t_5]$  also starte mit  $\mathbf{b}_{4+1} = \mathbf{b}_1$



# Zerlegung einer B-Spline-Kurve

Analog zum de Casteljau-Algorithmus kann der de Boor-Algorithmus zum Zerlegen einer B-Spline-Kurve in zwei Segmente verwendet werden

- Die *oberen Randelemente* und die *davor liegenden Kontrollpunkte* bilden die Kontrollpunkte des „linken“ B-Spline-Segments
- Die *unteren Randelemente* und die *darauf folgenden Kontrollpunkte* bilden die Kontrollpunkte des „rechten“ B-Spline-Segments



# NURBS-Kurven

Sind alle Knoten (bis auf die Randknoten) im Knotenvektor  $T$

- äquidistant (z.B.  $t_i = i$ ) so spricht man von *uniformen B-Spline-Kurven* (UBS-Kurven oder kurz UBS)
- nicht äquidistant so spricht man von *nicht-uniformen B-Spline-Kurven* (NUBS-Kurven oder kurz NUBS)

Ordnet man jedem Kontrollpunkt  $\mathbf{b}_i$  noch ein Gewicht  $w_i$  zu, so erhält man (analog zu den rationalen Bézier-Kurven) die *rationalen B-Spline-Kurven*

$$\mathbf{x}(t) = \frac{\sum_{i=0}^n w_i \mathbf{b}_i N_i^k(t)}{\sum_{i=0}^n w_i N_i^k(t)} \quad \text{bzw. homogen} \quad \mathbf{x}(t) = \sum_{i=0}^n \mathbf{B}_i N_i^k(t)$$

Rationale B-Spline-Kurven mit nicht-uniformem Knotenvektor heißen *nicht-uniforme rationale B-Spline-Kurven* (NURBS-Kurven)