

# Computergrafik

Prof. Dr. Christian Pape

Hochschule Karlsruhe

# Inhalt

## Einführung

Mathematische und geometrische  
Grundlagen

Bildrepräsentation

Raytracing

Transformationsmatrizen

Projektionen

Graphik-Pipeline

OpenGL

Texturabbildungen

Schattierung von Oberflächen

Prüfung

# Prüfungsleistungen

- ▶ Computergrafik: 2 SWS Vorlesung + 2 SWS Labor
- ▶ Aktuelle Prüfungsordnungen:
  - ▶ MINB (PO5): Pflichtfach, Modulprüfung mit Computer-Vision, 120 min, Labor.
  - ▶ INFB (PO7): Pflichtfach, Modulprüfung, 90 min, Labor.  
(Computer-Vision ist Wahlfach).
- ▶ Zukünftige Prüfungsordnungen:
  - ▶ MINB (PO6): Pflichtfach, Modulprüfung mit Computer-Vision, 120 min
  - ▶ INFB (PO8): Wahlfach, 90 min Klausur und Labor.  
(Computer-Vision ist Wahlfach).

- ▶ Drei Aufgaben teilweise aufgeteilt auf mehrere Übungsblätter.
  1. Darstellung einer bestehenden Asteroids-Implementierung ändern.
  2. Einen rudimentären Raytracer implementieren.
  3. Darstellung der Asteroids-Implementierung drei-dimensional implementieren mit Kamerabewegung.
- ▶ C++-20 (g++), SDL2 für Rastergrafik, SDL2\_Mixer für Sound, GoogleTest (gtest) für automatisierte Tests.
- ▶ Buildprozess mit cmake und make.
- ▶ Verwendung von IDEs auf eigene Gefahr.
- ▶ EU07: Ubuntu. Aufgabenblätter sind aus Unix-Sicht beschrieben.
- ▶ Windows MinGW verwenden.
- ▶ Aufgaben sind nicht auf Macs getestet.
- ▶ Aufgaben und Quelltextrahmen befinden sich in Ilias.

# Labor

## Anmeldung Labor

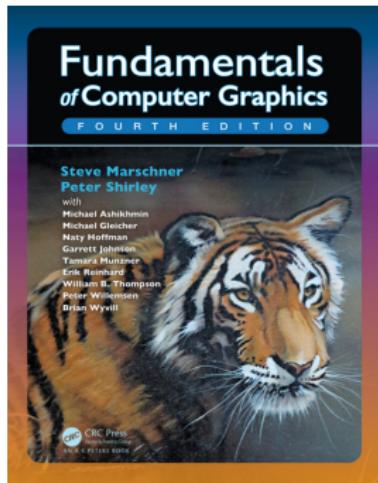
- ▶ Anmeldung zu Übungsgruppen via Ilias.
- ▶ Start nach der Vorlesung.
- ▶ Begrenzung auf 30 Personen pro Gruppe (Warteliste mit automatischem Aufrücken).
- ▶ Betreuung: Niklas Oesterle.
- ▶ Spätmöglichste Abgabetermine siehe Ilias, bei den Übungsgruppen.
- ▶ Lösung muss vor Abgabe in Ilias hochgeladen werden.
- ▶ Lösung muss Betreuer präsentiert und erläutert werden. Nur Hochladen reicht nicht.
- ▶ Keine Gruppenarbeit. Keine Quelltexte austauschen.

# Vorlesung

- ▶ Iliasgruppe beitreten.
- ▶ Folien werden kurz vor der Vorlesung aktualisiert.
- ▶ Ein Teil der Inhalte ist als Tafelanschrieb vorhanden.
- ▶ Mitschreiben oder abfotografieren. Bei letzterem bitte keine Personen mit ablichten.
- ▶ Zusätzliche Übungsaufgaben gibt es derzeit nicht.

# Vorlesung

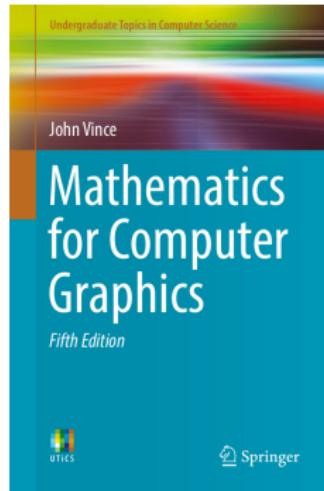
## Literatur



- ▶ S. Marchner, P. Shirley. *Fundamentals of Computer Graphics*. O'Reilly Verlag.
- ▶ Insbesondere: Kapitel 1–8 (Kern des Kern)
- ▶ Mathematische Grundlagen, Rastergrafik, Raytracing, Transformationen, Projektionen, Graphik-Pipeline.
- ▶ Teile der Vorlesung weichen in der Darstellung ab.
- ▶ Später noch vertiefende und weiterführende Inhalte.

# Vorlesung

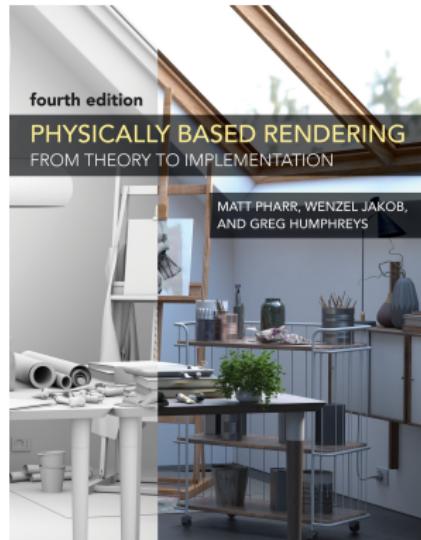
## Literatur



- ▶ J. Vince. *Mathematics for Computer Graphics*. Springer-Verlag.
- ▶ Grundlagen und Vertiefung der mathematischen Grundlagen.
- ▶ Ggf. verwenden, um mathematische Inhalte noch auf andere Weise erläutert zu bekommen.

# Vorlesung

## Literatur



- ▶ Open Book als HTML.
- ▶ <https://pbrt.org/>
- ▶ Printversion erhältlich.
- ▶ Implementierungsnahe Darstellung anhand eines zugehörigen existierenden Raytracers.

# Was ist Computergrafik?

## Engere Definition

- ▶ *Modellierung* und *Erzeugung* (rendering) von (animierten) *Bildern* (images).
- ▶ Modellierung (Modellraum, world space): Mathematische Modell, bestehend aus Punkten, Flächen, Linien, ...
- ▶ Rendering: Bilderzeugungsverfahren wie **Raytracing**, Radiosity, **Rasterverfahren**, 3D-Druck, ...
- ▶ „Bilder“ (Bildraum, screen space): **Pixelgrafiken**, Vektorgrafiken, Druck auf Papier, 3D-Erzeugnisse wie Spritzdruck oder CNC-Fräsen.

# Was ist Computergrafik?

## Einordnung und Abgrenzung

- ▶ Teil der Informatik (*Wissenschaft der automatisierten Verarbeitung von Daten mit Hilfe von Digitalrechnern*).
- ▶ *Graphische Datenverarbeitung* ist Oberbegriff
  - ▶ **Computergrafik:** Vom Modellraum zum Bildraum
  - ▶ Computervision: Vom Bildraum zum Modellraum
  - ▶ Digital Image Processing: Verarbeitungen im Bildraum, Filter, Farbanpassungen, Dithering, ...
  - ▶ Computational Geometry (grafisch-geometrisch Algorithmen): Algorithmen und Datenstrukturen, bei denen verarbeitete Daten grafische Primitive sind.
  - ▶ ...
- ▶ Beispiel Sonografie (Ultraschall): Aus Ultraschall 3D-Modelldaten erzeugen ist Computervision, die erzeugten Modelldaten rendern ist Computergrafik.
- ▶ Viele inhaltliche Überschneidungen.

# Anwendungen

## Auswahl

- ▶ Computerspiele
- ▶ Filmindustrie: Spezialeffekte, animierte Filme (Cartoons)
- ▶ Industrie (CAD/CAM): Konstruktion von Maschinen, Autos, Spielfiguren.
- ▶ Visualisierung von Daten: Medizin, Data-Mining
- ▶ Simulationen: (Animierte) Darstellung simulierter Prozesse, Flugsimulatoren
- ▶ Abbildung zeigt Flugsimulator DASA (Wikipedia).



# Inhalt

Einführung

**Mathematische und geometrische  
Grundlagen**

Bildrepräsentation

Raytracing

Transformationsmatrizen

Projektionen

Graphik-Pipeline

OpenGL

Texturabbildungen

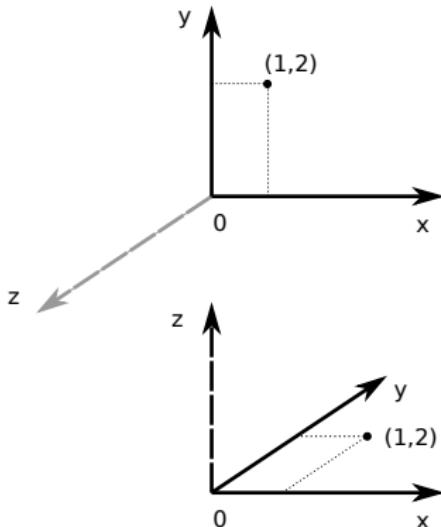
Schattierung von Oberflächen

Prüfung

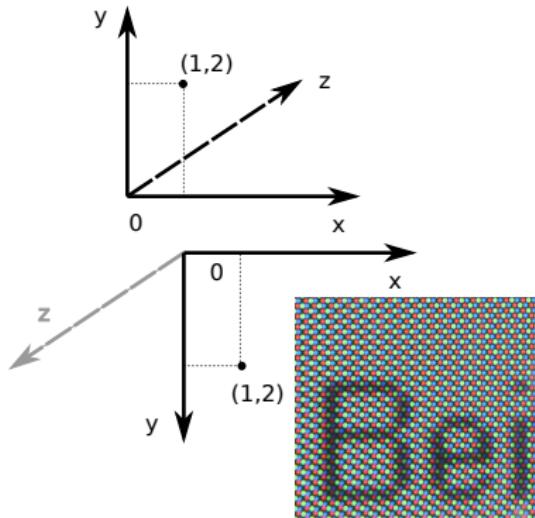
## Globales Koordinatensystem

## Euklidischer Raum

- ▶ Rechts-händisch
  - ▶ Unten:  $90^\circ$  um x-Achse nach hinten gedreht



- ▶ Links-händisch
  - ▶ Unten:  $180^\circ$  um x-Achse nach vorne gedreht



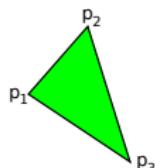
(Lochmaske Kathodenstrahlröhre, Wikipedia)

- Graue z-Achse weglassen: 2-dimensionaler Fall

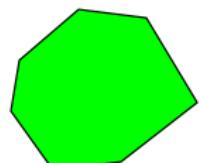
# Polygon

**Definition** Ein **Polygon** ist eine Fläche, die durch eine Folge von  $k > 2$  Punkten  $p_1, \dots, p_k$  definiert ist mit  $p_i \in \mathbb{R}$ . Die Punkte bilden einen geschlossenen Streckenzug.

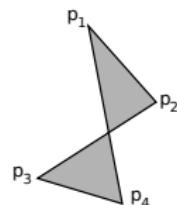
- Wir betrachten nur **planare** Polygone: Die Fläche liegt in einer Ebene.
- Wir betrachten nur **einfache** Polygone, deren Kanten sich nicht überschneiden (überschlagen).



Dreick



konvex



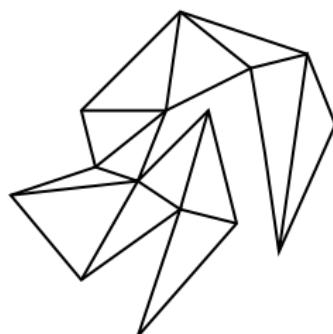
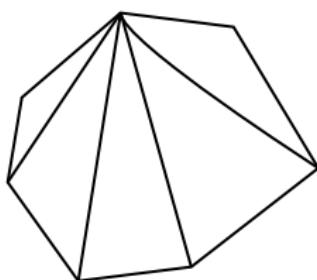
Überschlagend



einfach

# Triangulierung

**Satz** Jedes einfache Polygon mit  $k$  Punkten lässt sich in  $k - 2$  Dreiecke zerlegen.



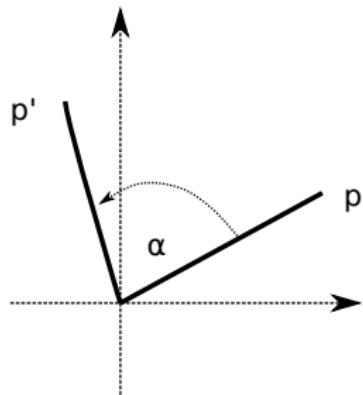
**Definition** Ein **Ohr** ist ein aus drei aufeinander folgende Punkte gebildetes (inneres) Dreieck eines Polygon, welches von keiner Kante überschnitten wird.

**Satz** Jedes einfache Polygon mit mehr als drei Punkten besitzt mindestens zwei Ohren (Zwei-Ohren-Theorem).

**Algorithmus** Solange ein Ohr suchen und abschneiden, bis nur noch ein Dreieck vom Polygon übriggeblieben ist.

# Sinus und Cosinus

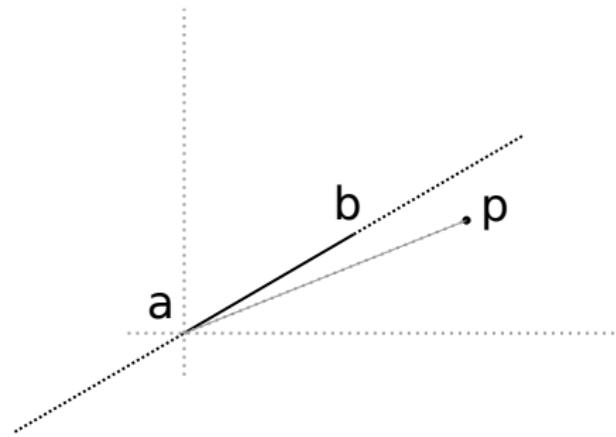
- ▶  $\cos \alpha$  im Einheitskreis ist die Länge der Projektion auf die x-Achse des Radius 1 um den Winkel  $\alpha$ .
- ▶  $\sin \alpha$  analog zur y-Achse.
- ▶ Für beliebigen Radius  $r > 0$  hat die Projektion die Länge  $r \cdot \cos \alpha$ .
- ▶  $\sin \alpha \geq 0$  für  $0^\circ \leq \alpha \leq 180^\circ$ .
- ▶  $\cos \alpha \geq 0$  für  $-90^\circ \leq \alpha \leq 90^\circ$
- ▶  $\sin^2 \alpha + \cos^2 \alpha = 1$   
(Trigonometrischer Phytagoras)
- ▶  $\sin(\alpha \pm \beta) = \sin \alpha \cdot \cos \beta \pm \cos \alpha \cdot \sin \beta$   
(Additionstheorem(e))



# Lage eines Punkts zu einer Strecke

Gegeben: Strecke von  $a$  nach  $b$ , ein Punkt  $p$

Gesucht: Liegt  $p$  auf der Strecke  $\overline{ab}$ , links davon oder rechts?



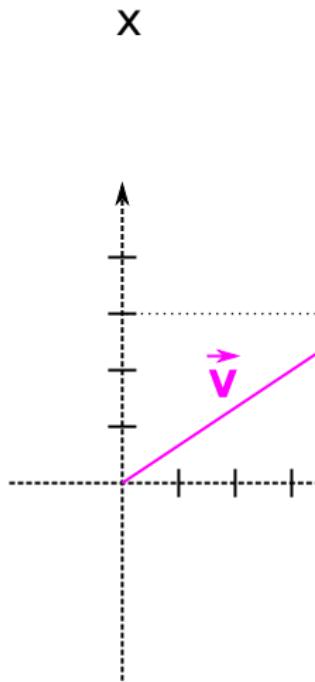
- ▶ Zwei Winkel  $\alpha$  und  $\beta$  finden, so dass Winkeldifferenz in diesen drei Fällen 0 (auf der Strecke), positiv (links) oder negativ (rechts) ist.
- ▶ Additionstheorem für sin verwenden, um Lösung zu berechnen.
- ▶ Anwendung: Punkt liegt in einem Dreieck.

# Vektor

## Definition

- ▶ Ein n-dimensionaler **Vektor**  $\vec{v}$  hat eine *Länge* und eine *Richtung*.
- ▶ Repräsentation durch kartesische Koordinaten in  $\mathbb{R}^n$ .
- ▶ In der Computergrafik meist  $2 \leq n \leq 4$ .
- ▶ Alternative Repräsentation durch Polarkoordinaten (für uns nicht von Bedeutung).
- ▶  $\vec{v} = (x_1, x_2, \dots, x_n)$

4,5



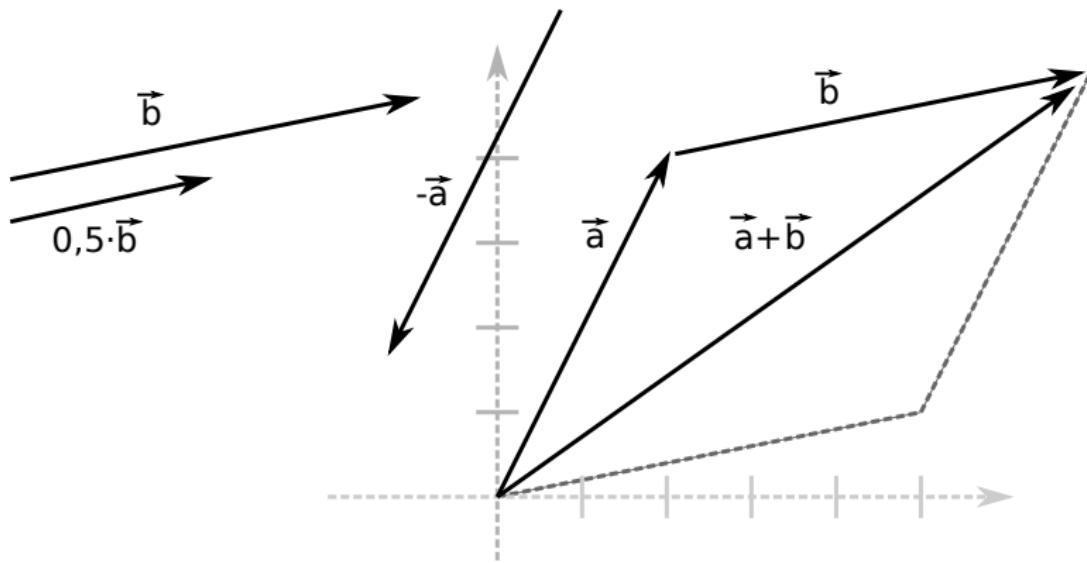
- ▶ Länge  $|\vec{v}| = \sqrt{x_1^2 + \dots + x_n^2}$  (euklidische Distanz).
- ▶ Richtungswinkel  $\alpha$  (meist Bogenmaß)
- ▶ In der Regel  $\alpha \in [-\pi, \pi]$  bzw.  $\alpha \in [-180, 180]$

# Vektorraum

- ▶ **Vektoraddition** und -subtraktion komponentenweise:
- ▶  $\vec{a} + \vec{b} := (a_{x_1} + b_{x_1}, \dots, a_{x_n} + b_{x_n})$ .
- ▶  $\vec{a} - \vec{b} := (a_{x_1} - b_{x_1}, \dots, a_{x_n} - b_{x_n})$ .
- ▶ Vektoraddition bildet abelsche additive Gruppe über  $\mathbb{R}^n$ . Nullvektor  $\vec{0}$  ist neutrales Element,  $-\vec{a} := \vec{0} - \vec{a}$  ist inverses Element.
- ▶ **Skalarmultiplikation** mit  $s \in \mathbb{R}$ :  $s \cdot \vec{a} := (s \cdot a_{x_1}, \dots, s \cdot a_{x_n})$
- ▶ Skalarmultiplikation bildet mit Vektoraddition einen **Skalkörper** über  $\mathbb{R}$ :
  - ▶  $s \cdot (\vec{a} + \vec{b}) = s \cdot \vec{a} + s \cdot \vec{b}$
  - ▶  $(s + t) \cdot \vec{a} = s \cdot \vec{a} + t \cdot \vec{a}, t \in \mathbb{R}$
  - ▶  $(s \cdot t) \cdot \vec{a} = s \cdot (t \cdot \vec{a})$
  - ▶  $1 \cdot \vec{a} = \vec{a}$
- ▶ Diese algebraische Struktur heißt **Vektorraum**.

# Vektoraddition

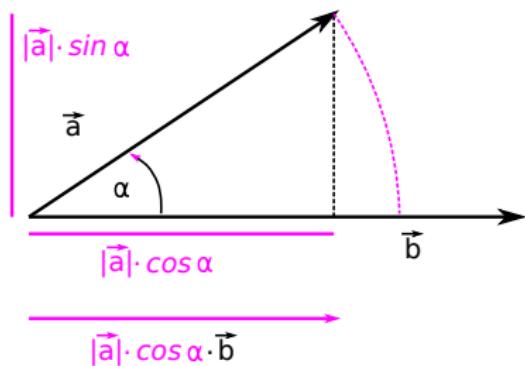
## Geometrische Anschauung



# Skalarprodukt

## Definition

- ▶ Gegeben seien zwei Vektoren  $\vec{a}$  und  $\vec{b}$ .
- ▶  $\vec{a} \cdot \vec{b} := a_{x_1} \cdot b_{x_1} + \dots + a_{x_n} \cdot b_{x_n} \in \mathbb{R}$  heißt **Skalarprodukt**.
- ▶ Es gelten folgende Eigenschaften:
  - ▶  $\vec{a} \cdot \vec{b} = \vec{b} \cdot \vec{a}$
  - ▶  $\vec{a} \cdot (\vec{b} + \vec{c}) = (\vec{a} \cdot \vec{c}) + (\vec{b} \cdot \vec{c})$
  - ▶  $s \cdot (\vec{a} \cdot \vec{b}) = \vec{a} \cdot (s \cdot \vec{b})$
  - ▶  $\vec{a} \cdot \vec{b} = \cos \alpha |\vec{a}| \cdot |\vec{b}|$
- ▶ Anwendungen: Bei orthogonalen Vektoren ist das Skalarprodukt 0, Berechnung von  $\cos \alpha$ .

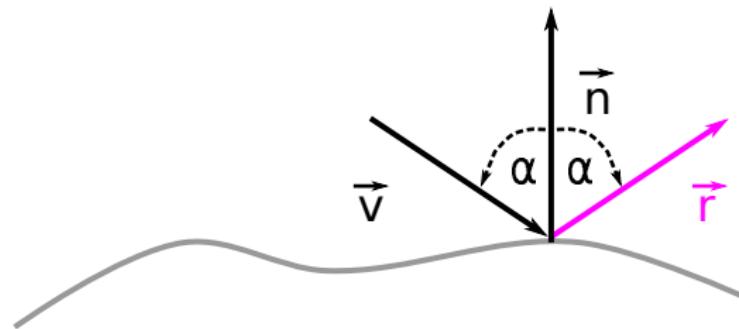


# Skalarprodukt

## Reflektionsvektor

Gegeben: Ein Vektor  $\vec{v}$  ein Vektor  $\vec{n}$  mit  $|\vec{n}| = 1$ , der senkrecht auf einer Oberfläche steht (Normalenvektor).

Gesucht: Reflektionsrichtung  $\vec{r}$  (Einfallswinkel gleich Ausfallwinkel)

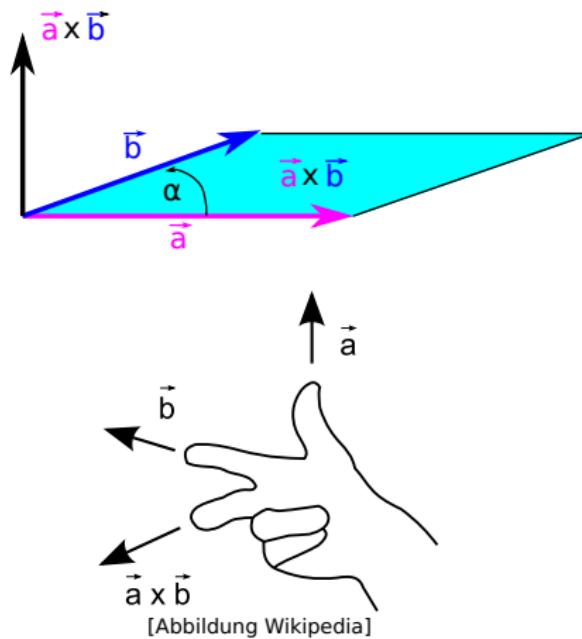


- ▶ Behauptung:  $\vec{r} = \vec{v} - 2(\vec{v} \cdot \vec{n}) \cdot \vec{n}$
- ▶ Anwendungen: Raytracing, Stoßgesetze

# Kreuzprodukt (Vektorprodukt)

## Definition

- ▶ Gegeben  $\vec{a}, \vec{b} \in \mathbb{R}^3$
- ▶  $\vec{a} \times \vec{b} := (a_2 \cdot b_3 - a_3 \cdot b_2, a_3 \cdot b_1 - a_1 \cdot b_3, a_1 \cdot b_2 - a_2 \cdot b_1) \in \mathbb{R}^3$  heißt **Kreuzprodukt** von  $\vec{a}$  und  $\vec{b}$ .
- ▶ Es gelten folgende Eigenschaften:
- ▶  $\vec{a} \times (\vec{b} + \vec{c}) = (\vec{a} \times \vec{b}) + (\vec{a} \times \vec{c})$
- ▶  $\vec{a} \times \vec{b} = -(\vec{b} \times \vec{a})$   
(Antikommutativ)
- ▶  $\vec{a} \times \vec{a} = 0$
- ▶  $|\vec{a} \times \vec{b}| = |\vec{a}| \cdot |\vec{b}| \sin \alpha.$
- ▶ **Drei-Finger-Regel** für rechte Hand von  $\vec{a} \times \vec{b}$
- ▶ Anwendungen: Schnittpunkt Halbstrahl mit Dreieck, Physik



# Lineare Interpolation von Punkten

Gegeben: Zwei Punkte  $\vec{a}$  und  $\vec{b}$  in  $\mathbb{R}^n$

Gesucht: Alle Punkte  $p(\alpha)$  auf der Geraden, die durch  $\vec{a}$  und  $\vec{b}$  gebildet werden,  $\alpha \in \mathbb{R}$ .

- ▶ In der Informatik:  $p = \text{lerp}(\vec{a}, \vec{b}, \alpha)$  (linear interpolation), z.B. `std::lerp`.

- ▶ Zusätzliche Eigenschaften:  $\text{lerp}(\vec{a}, \vec{b}, 0) = \vec{a}$  und  $\text{lerp}(\vec{a}, \vec{b}, 1) = \vec{b}$

Für  $0 \leq \alpha \leq 1$  liegen die Punkte auf der Strecke zwischen  $\vec{a}$  und  $\vec{b}$ .

- ▶ Lösung:  $\text{lerp}(\vec{a}, \vec{b}, \alpha) = \alpha \vec{a} + (1 - \alpha) \vec{b}$

- ▶ Anwendungen:  $\alpha$ -Blending,  $\gamma$ -Korrektur, Gerade in Parameterdarstellung (Physik: Berechnung von Schwerpunkten).

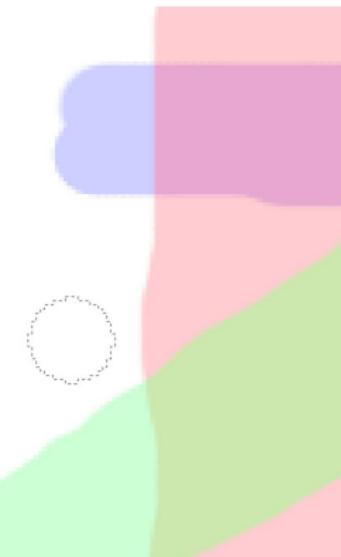
- ▶  $\alpha$ -Blending: Pinsel mit Farbe  $a$  auf farbigen Untergrund  $b$  malen. Dann soll die Untergrundfarbe zu 50 % ( $\alpha = 0.5$ ) durchscheinen. (Aquarell)

- ▶ Umkehrung:  $p$  gegeben,  $\alpha$  gesucht. Numerisch stabiles, robustes Verfahren?

# Lineare Interpolation von Punkten

## Alpha-Blending

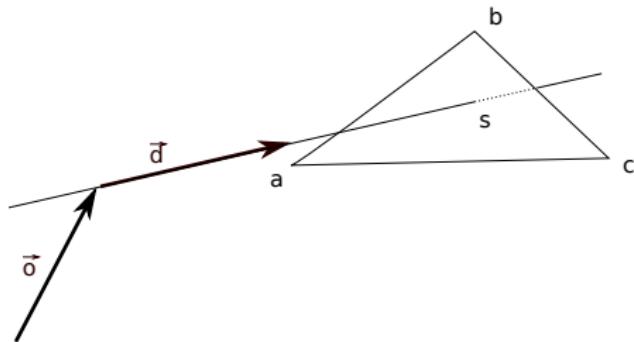
- ▶ GIMP Pinseleigenschaften
- ▶ Deckkraft =  $\alpha$ -Wert (in Prozent)



# Schnitt Halbstrahl mit Dreieck

Gegeben: Ein Dreieck im  $\mathbb{R}^3$  mit Punkten  $a, b$  und  $c$  und Gerade  $g(t) = \vec{o} + t \cdot \vec{d}$  (für  $t \geq 0$ ).

Gesucht: Schnittpunkt  $s$  von  $g(t)$  mit Dreieck.



- ▶ Ebenengleichung aufstellen. Beispielsweise Hessesche Normalform:  
 $\vec{n} \cdot \vec{v} = d$ .
- ▶  $g(t)$  einsetzen und nach  $t$  auflösen:  $t = \frac{\vec{n}\vec{a} - \vec{n}\vec{o}}{\vec{n}\vec{d}}$
- ▶ Liegt  $s$  im Dreieck? (bereits gelöst)
- ▶ Anwendung: Raytracing

# Strahlensätze

## Erste Strahlensatz

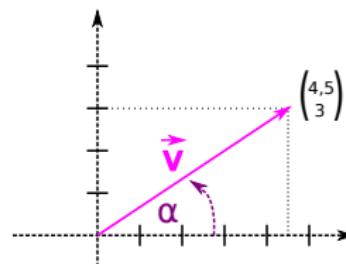
Prämissen: Ein **Scheitelpunkt**  $s$ ,

zwei Halbstrahlen, die von  $s$  ausgehen

und zwei parallele Strecken, welche die Halbstrahlen in den Punkten  $a, b$  und  $c, d$  schneiden (siehe Abbildung).

Dann gilt:

$$\frac{\overline{sa}}{\overline{sb}} = \frac{\overline{sc}}{\overline{sd}}$$

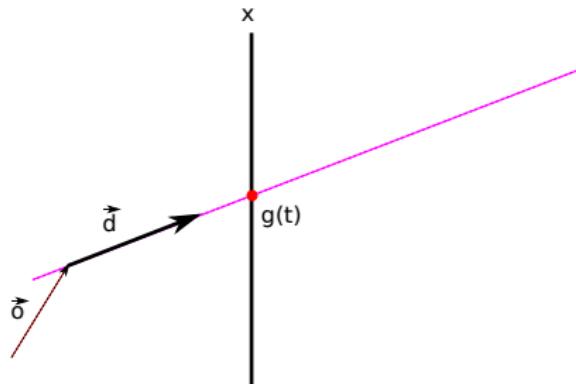


# Schnitt mit Strecken

## Achsenparallele Strecke

Gegeben: Gerade in Parameterdarstellung  $g(t) = \vec{o} + t \vec{d}$  und Koordinate  $x$  (jede Dimension möglich).

Gesucht: Lösung  $t$  für Schnittpunkt  $g(t)$  für Koordinate  $x$ .



- ▶ Behauptung:  $t = \frac{x - o_x}{d_x}$
- ▶ Lösung mit Strahlensatz
- ▶ Anwendung: Schnitt mit achsen-orientierten Rechteck (axis aligned bounding box, AABB)

# Matrizenrechnung

- Eine  $n \times m$ -**Matrix**  $A \in M^{n \times m}$  ist eine rechteckige Anordnung von Elementen einer Menge  $M$  in  $n$ -Zeilen und  $m$ -Spalten:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \dots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} = (a_{ij}), i = 1, \dots, n; j = 1, \dots, m$$

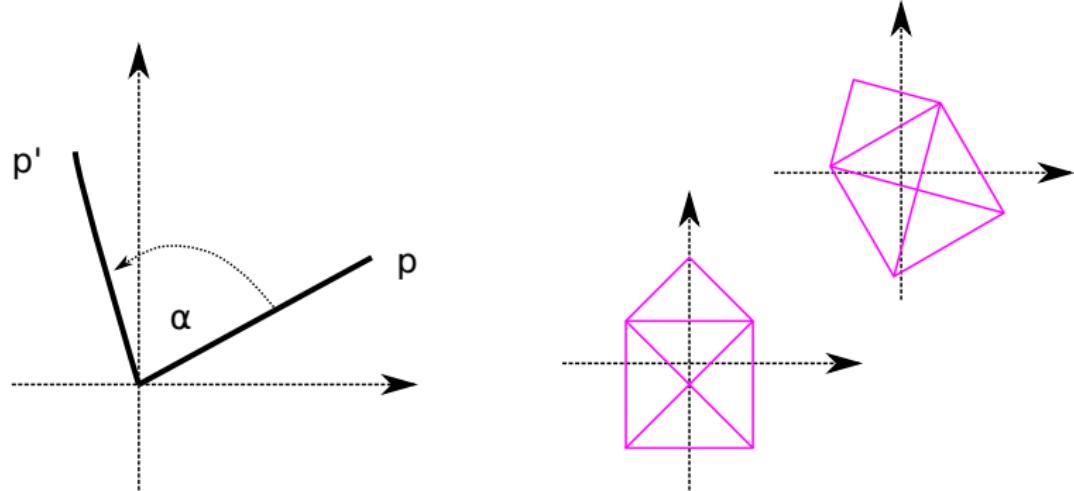
- Computergrafik meist quadratische Matrizen ( $n = m$ ) und  $n = 2, 3, 4$
- Addition elementweise:  $A + B := (a_{ij} + b_{ij})$ .
- Multiplikation:  $A \cdot B := c_{ij} = \sum_{k=1}^m a_{ik} \cdot b_{kj}$  (Zeile  $i$  mal Spalte  $j$ )
- Es gelten (bei passenden Matrizen) Assoziativ- und Distributivgesetze. Aber nicht kommutativ.
- Multiplikation  $n \times m$ -Matrix  $A$  mit Vektor  $v = (x_1, \dots, x_m)$ :  
 $A \cdot v := (\sum_{i=1}^m a_{1i} \cdot x_i, \dots, \sum_{i=1}^m a_{ni} x_i)$
- Anwendungen: Transformationen von Punkten und Objekten.

# Matrizenrechnung

## Drehmatrix

Gegeben: Punkt  $p \in \mathbb{R}^2$ , Winkel  $\alpha$ .

Gesucht: Drehung von  $p$  um den Winkel  $\alpha$  um den Ursprung herum.



- ▶ Drehung mit **Drehmatrix**  $R_\alpha$  repräsentieren.
- ▶ Drehung ist Multiplikation  $p' = R_\alpha \cdot p$ .

# Lineare Gleichungssystem (LGS)

## Cramersche Regel

Gegeben:  $m$  Lineare Gleichungen  $a_{1i} \cdot x_1 + \dots + a_{mi} \cdot x_n = b_i$  mit  $n$  Unbekannten  $x_i$  für  $i = 1, \dots, m$ . (Wertemenge  $\mathbb{R}$ )

Gesucht: Eine Belegung von  $x$ , welches die Gleichungen erfüllt. (Lösung des LGS)

- ▶ In Matrixform  $A \cdot x = b$
- ▶ mit  $x = (x_1, \dots, x_n)$ ,  $b = (b_1, \dots, b_m)$ ,  $A = ((a_{ij}))$ ,  $i = 1, \dots, m; j = 1, \dots, n$
- ▶ Cramersche Regel für **quadratische** LGS ( $n = m$ ):

$$x_i = \frac{\det(A_i)}{\det(A)}$$

wobei  $A_i$  aus  $A$  entsteht, indem die  $i$ -te Spalte durch  $b$  ersetzt wird.

- ▶ Beispiel:

$$\begin{array}{rcl} x &+& y &= 1 \\ x &-& y &= 2 \end{array}$$

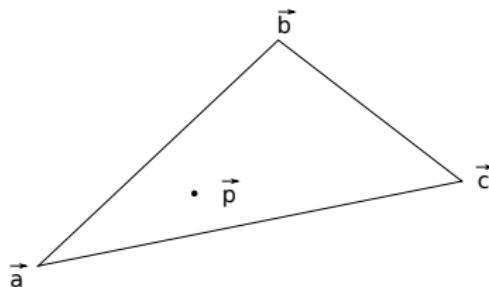
- ▶ Anwendungen: Lösungen von Schnittpunkt zweier Geraden

# Lineare Interpolation

## Barycentrische Koordinaten

Gegeben: Drei paarweise, verschiedene Punkte  $\vec{a}, \vec{b}, \vec{c}$  in  $\mathbb{R}^n$ ,  
Punkt  $\vec{p}$  in der durch die Punkte definierten Ebene.

Gesucht:  $u, v, w \in \mathbb{R}$  mit  $\vec{p} = u \cdot \vec{a} + v \cdot \vec{b} + w \cdot \vec{c}$   
**(Baryzentrische Koordinaten)**



- ▶ Zusätzliche Eigenschaften (analog lerp mit  $p = \alpha a + \beta b$ ):  
 $u + v + w = 1$   
 $0 \leq u, v, w \leq 1$ , wenn  $\vec{p}$  im Dreieck liegt.
- ▶ Folgerung:  $w = 1 - u - v$
- ▶ Anwendungen: Interpolation von Normalenvektoren, Schwerpunkt einer Fläche an einem Punkt.

# Lineare Interpolation

## Barycentrische Koordinaten

- Anwendungsbeispiel: Farbauswahl GIMP.
- Lineare Interpolation zwischen ausgewählter Farbe (hier Rot), Schwarz und Weiß durch Verschiebung eines Punkts.



# Schnitt Halbstrahl mit Dreieck

Verbesserung mit Barycentrische Koordinaten

Gegeben: Ein Dreieck im  $\mathbb{R}^3$  mit Punkten  $a, b$  und  $c$  und Gerade  $g(t) = \vec{o} + t \cdot \vec{d}$  (für  $t \geq 0$ ).

Gesucht: Schnittpunkt  $s$  von  $g(t)$  mit Dreieck.

- ▶ Wie bisher: Sehstrahl und Ebene auf Parallelität prüfen.
- ▶ Dann:  $u$  und  $v$  berechnen,  $t$  erst zum Schluss
- ▶ LGS aufstellen zur Lösung von  $t$ ,  $u$  und  $v$  via

$$\vec{o} + t \cdot \vec{d} = u \cdot \vec{a} + v \cdot \vec{b} + (1 - u - v) \cdot \vec{c}$$

- ▶ Mit Cramerschen-Regel lösen.
- ▶ Für  $v_1, v_2, v_3 \in \mathbb{R}^3$  gilt:  $\det(v_1|v_2|v_3) = v_1 \cdot (v_2 \times v_3)$  (Beweis durch Nachrechnen)
- ▶ Möller, Tomas und Trumbore, Ben: *Fast, Minimum Storage Ray-Triangle Intersection*. Journal of Graphics Tools. 2: 21–28.

# Inhalt

Einführung

Mathematische und geometrische  
Grundlagen

**Bildrepräsentation**

Raytracing

Transformationsmatrizen

Projektionen

Graphik-Pipeline

OpenGL

Texturabbildungen

Schattierung von Oberflächen

Prüfung

# Raster vs Vektorgraphik

- ▶ Rastergraphik
- ▶ Vektorgraphik

# Rastergraphik

- ▶ **Rastergraphik:** Bild wird durch eine Matrix von farbigen **Pixel** (pixel elements) repräsentiert.
- ▶ Anzahl Zeilen und Spalten der Matrix nennt man Auflösung.
- ▶ Pixel können quadratisch sein.
- ▶ Ausgabegeräte:
  - ▶ *Bildschirm:* Röhrenmonitor, LED, Plasma (selbstleuchtend), LCD (lichtdurchlässig), ...
  - ▶ Hardcopy: Tintenstrahldrucker, 3D-Drucker,  
...
- ▶ Eingabegeräte: Digitalkamera, Flachbettscanner, ...
- ▶ Farbbildschirme: Farbe ist in Rot, Grün und Blau-Anteil aufgeteilt (additive Farbmischung).

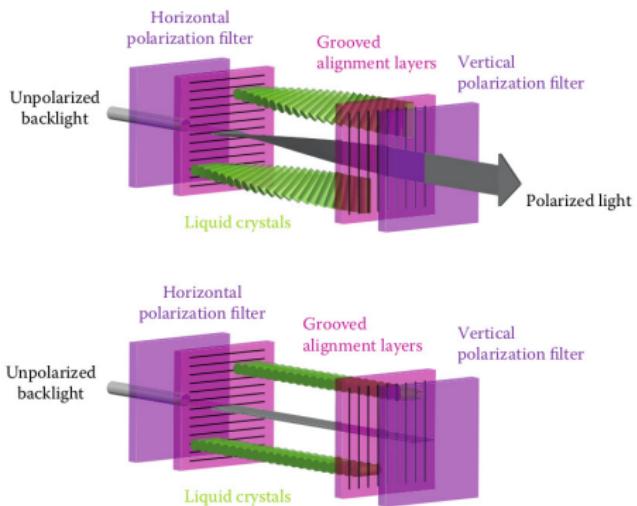


PLATO V Terminal  
with plasma display  
1981 (Wikipedia)

## Beispiel LC-Display

- ▶ LCD (liquid crystal displays)
  - ▶ Unpolarisierte weiße Hintergrundbeleuchtung
  - ▶ Flüssigkristalle zwischen zwei Dioden und gerillten Polarisationsfiltern  
  - ▶ Spannung zwischen Dioden ordnet Kristalle unterschiedlich an
  - ▶ Oben: Kristalle polarisieren Licht vertikal.
  - ▶ Unten: Horizontale Polarisierten bleibt erhalten.
  - ▶ Pixel besteht aus drei LC mit Farbfilter für Rot, Grün und Blau.

The diagram illustrates the internal structure of an LCD pixel. It shows two sets of parallel horizontal lines representing 'Horizontal polarization filter' layers. Between these filters are three colored rectangular blocks labeled 'Liquid crystal'. A green arrow labeled 'Unpolarized backlight' enters from the left and passes through the top polarization filter. The light then passes through the first liquid crystal layer, which rotates the polarization by 90 degrees, making it vertical. This vertically polarized light then passes through the second polarization filter. The second liquid crystal layer does not rotate the polarization again, so the light remains vertical. Finally, the third liquid crystal layer rotates the polarization back to horizontal, and the light passes through the bottom polarization filter, exiting as horizontally polarized light. The entire assembly is contained within a small rectangular frame representing a single pixel.



# Beispiel Tintenstrahldrucker

- ▶ Druckkopf besteht aus einer Matrix einzelner Düsen.
- ▶ Düse wird durch Druck gepresst: ein Farbkleck wird abgesondert (z.B. Piezokristall)
- ▶ Muster von Farbkleckse bestimmen Helligkeit.
- ▶ Pixel besteht aus drei *additiv* gemischten Farben: Cyan, Yellow, Magenta.
- ▶ Druckauflösung wird mit Pixeldichte beschrieben: DPI (dots per inch), PPI (pixel per inch).

U.S. Patent Feb. 8, 1983 Sheet 1 of 2 4,372,773

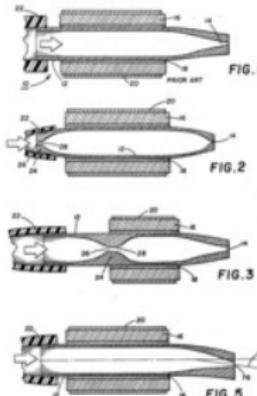


Abbildung Wikipedia

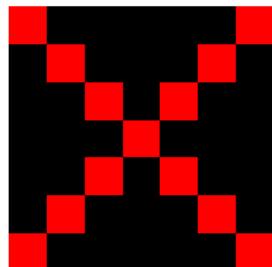
# Portable Anymap

- ▶ Binär- und textbasierte (ASCII), unkomprimierte Dateiformate für RGB-Bitmaps
- ▶ Teil von Netpbm: Sammlung 350 Kommandozeilen-orientierter Programme inklusive Konverter für ca. 100 Dateiformate.
- ▶ <https://netpbm.sourceforge.net>

Portable Bitmap	PBM	Monochrom	1 Bit
Portable Graymap	PGM	Graustufen	8/16 Bit
<b>Portable Pixmap</b>	PPM	RGB-Farben	24/48 Bit

- ▶ jeweils **Text-** und Binärversion.
- ▶ Portable Anymap: Erweiterung, die alle drei Formate in einem erweiterten Format inklusive Alphakanal vereint.

# Portable Pixmap



- ▶ Pro Datei ein Bild, max. 70 Zeichen pro Zeile
- ▶ Pro Farbeanteil ein Wert (ITU-R Recommendation BT.709)
- ▶ Pixel: Drei mit Leerzeichen getrennte Farbwerte
- ▶ Folge von mit Leerzeichen getrennte Pixel
- ▶ Linkshändisches Koordinatensystem

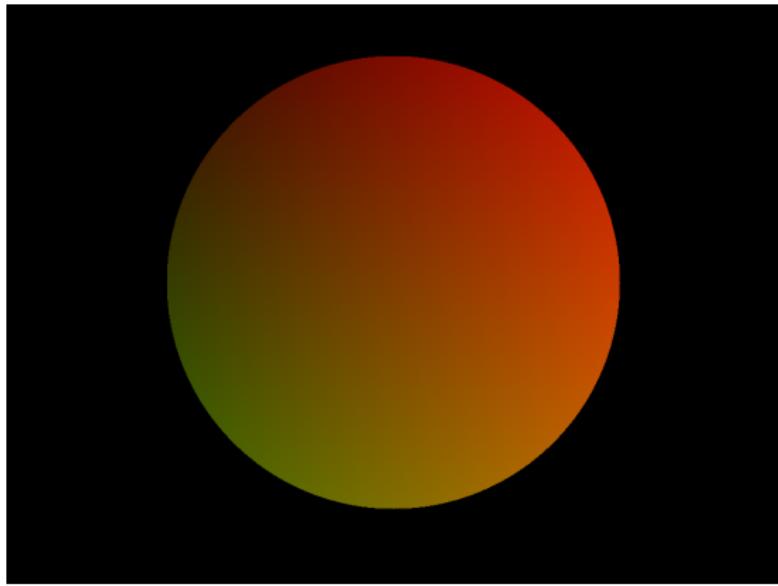
```
P3      # magic number: PPM Textformat
7 7      # 7 x 7 Raster
15      # max. Wert pro RGB-Farbanteil [0, 65536]
15 0 0    0 0 0    0 0 0    0 0 0    0 0 0    0 0 0    15 0 0
  0 0 0    15 0 0   0 0 0    0 0 0    0 0 0    15 0 0   0 0 0
  0 0 0    0 0 0    15 0 0   0 0 0    15 0 0   0 0 0    0 0 0
  0 0 0    0 0 0    0 0 0    15 0 0   0 0 0    0 0 0    0 0 0
  0 0 0    0 0 0    0 0 0    0 0 0    15 0 0   0 0 0    0 0 0
  0 0 0    0 0 0    0 0 0    0 0 0    0 0 0    15 0 0   0 0 0
  0 0 0    15 0 0   0 0 0    0 0 0    0 0 0    0 0 0    15 0 0
15 0 0    0 0 0    0 0 0    0 0 0    0 0 0    0 0 0    0 0 0
```

# Portable Pixmap

```
#include <iostream>

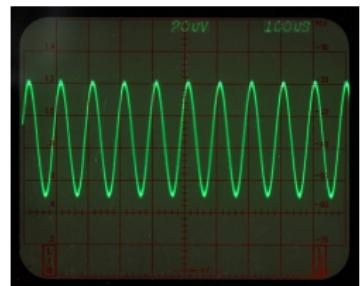
int main(void) {
    std::cout << "P3" << std::endl;
    std::cout << "1024 768" << std::endl;
    std::cout << "255" << std::endl;
    for (size_t y = 0; y < 768; y++) {
        for (size_t x = 0; x < 1024; x++) {
            if ( (x-512) * (x-512) + (y-368) * (y-368) <= 300 * 300 )
                std::cout << (x / 4) << " " << (y / 6) << " 0\t";
            else
                std::cout << "0 0 0\t";
        }
        std::cout << std::endl;
    }
}
```

# Portable Pixmap



# Vektorgrafik

- ▶ **Vektorgrafik:** Bild wird durch grafische Objekte (Kreis, Linie) und deren Eigenschaften (Farbe, Strickstärke) beschrieben.
- ▶ Geometrische Objekte werden innerhalb eines Koordinatensystems (2D/3D) angegeben.
- ▶ Vektorgrafiken besitzen keine rasterartige Auflösung.
- ▶ *Vektormonitor:* Röhre mit frei positionierbaren Elektronenstrahl (monochrom).
- ▶ Heute: Vektorgrafik wird auf Rastergrafik abgebildet.
- ▶ Asteroids: Electrohome G05-801/2 (19 Zoll). 4:3 Seitenverhältnis. Ansteuerung 1024x1024 Punkte.



Oszilloskop (Wikipedia)

# Portable Document Format (PDF)

- ▶ Ab 1992 entwickelt von Adobe, basierend auf PostScript, um Dokumente Geräte-unabhängig darzustellen.
- ▶ Keine Programmiersprache wie PostScript, erweitert PostScript um Dokumenten- und Dateistruktur, Hyperlinks, Eingabefelder.
- ▶ PDFs können als ASCII-Texte codiert werden.
- ▶ Acht Datentypen für Objekte und Beispielwerte:

Boolean true false

Number integer 123 -17 0 and real numbers 1.5 -0.5 4.

Strings (Hallo) () (a (b) c)

Names /Name1 /A;Various\*\*\*Characters?

Arrays [ 549 3.14 false (Ralph) [1 2 3] /SomeName ]

Dictionaries << /Key1 17 /Key2 (text) >>

Streams stream ...viele einzelne Zeichen... endstream (immer  
hinter einem Dictionary!)

null

# Portable Document Format

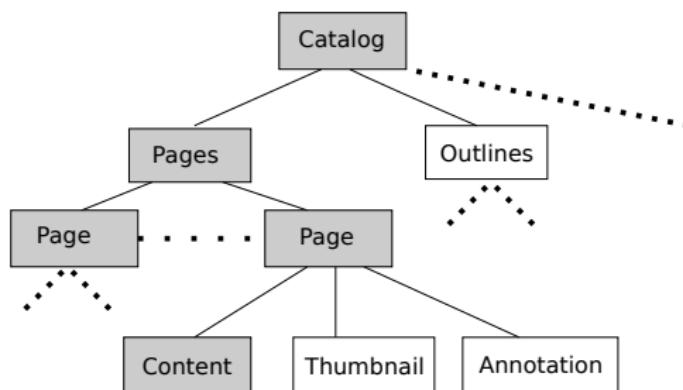
## Catalog

- ▶ Objekte können indirekt über einen *object identifier* referenziert werden.
- ▶ Definition: 1 0 obj (Hallo) endobj
- ▶ object identifier: besteht aus zwei ganzen Zahlen, erste positiv: *object number* gefolgt von *generation number* (für Ursprungsdokument immer 0)
- ▶ Referenzieren des Objekts über das Zahlenpaar gefolgt von R:  
[ 1 0 R (guten) (Tag) 1 0 ]
- ▶ Keine Reihenfolgeabhängigkeiten zwischen Objekten und Referenzen:

```
1 0 obj % minimale Definition des document catalogs
<< /Type /Catalog
    /Pages 2 0 R
>>
endobj
```

# Portable Document Format

- ▶ PDF ist hierarchisch aus Dictionaries aufgebaut.
- ▶ Grau: Notwendig Dictionaries.



# Portable Document Format

## Pages

- ▶ Pages enthält die darzustelenden Seiten (Page).
- ▶ Die Seiten werden als Array angegeben mit Verweisen auf jeden Seite.
- ▶ Anzahl Seiten muss angegeben werden.

```
2 0 obj
  << /Type /Pages
    /Kids [ 3 0 R ]
    /Count 1
>>
endobj
```

# Portable Document Format

## Page

- ▶ Page: Beschreibung einer Seite insbesondere Inhalt und Größe.
- ▶ MediaBox: Array [ llx lly urx ury ] von unterer linken und obere rechten Ecke des anzuseigendes Seitenbereichs (user space, Geräteunabhängig)
- ▶ Resources: Prozeduren, Fonts, ...
- ▶ ProcSet: Array vordefinierter, von der Seite verwendeter Funktionalitäten (Prozeduren). /PDF beinhaltet Basisfunktionalitäten.

```
3 0 obj
<< /Type /Page
    /Parent 2 0 R
    /MediaBox [ 0 0 612 792 ] % US-Letter
    /Contents 4 0 R
    /Resources << /ProcSet [ /PDF ] >>
>>
endobj
```

# Portable Document Format

## Grafikbefehle

- ▶ Content: Text, Grafiken, ...
- ▶ Length: Anzahl Zeichen des Streams.
- ▶ 150 250 m: An Koordinate (150, 250) beginnt ein Grafikpfad.
- ▶ 150 305 1: Eine Linie wird zur letzten Koordinate des Pfads definiert.
- ▶ S: Der definierte Pfad wird gezeichnet (Stroke).

```
4 0 obj
<< /Length 88 >>
stream
% Draw a black line segment, using the default line width
150 250 m
150 350 l
S
endstream
```

# Portable Document Format

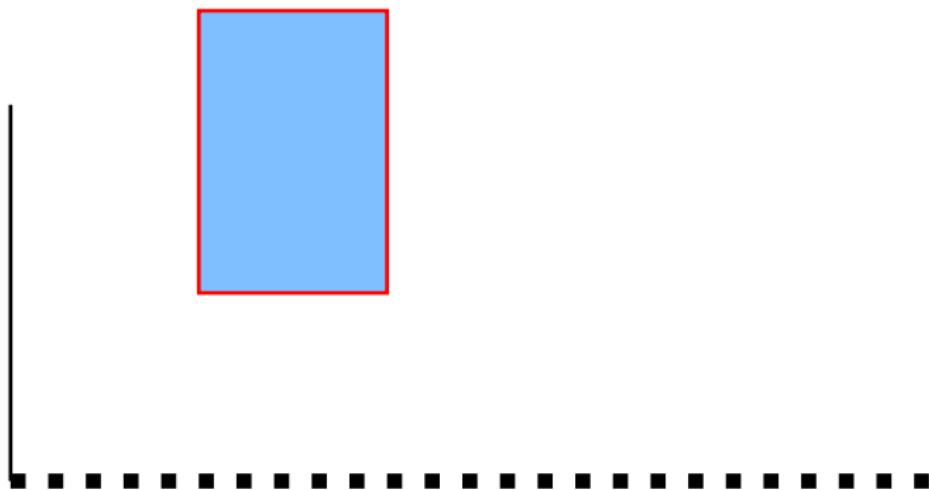
## Grafikbefehle

```
150 250 m % Draw a black line segment with default line width
150 350 l
S
4 w % Set line width to 4 points
[ 4 6 ] 0 d % Set dash pattern to 4 units on, 6 units off
150 250 m
400 250 l
S
[ ] 0 d % Reset dash pattern to a solid line
1 w % Reset line width to 1 unit
1.0 0.0 0.0 RG % Red for stroke color (upper case)
0.5 0.75 1.0 rg % Light blue for fill color
200 300 50 75 re % add rectangle to path
B % Fill and then stroke the path
```

# Portable Document Format

## Darstellung

- Mit pdf2ps nach Postscript konvertiert, anschließend mit ps2eps nach Encapsulated PostScript konvertiert, um es in LaTeX einzubinden.



# Portable Document Format

## Aufbau PDF

- ▶ PDF besteht aus vier aufeinander folgende Abschnitte.
- ▶ Header: Erste Zeile mit verwendeter PDF Version, z.B %PDF1.4
- ▶ Body: Alle Objekte, wie zuvor beschrieben.
- ▶ Cross-Reference-Table (xref): Index mit Position auf jedes indirekte Objekt in der Datei.
- ▶ Trailer: Position der Cross-Reference-Table und speziellen indirekten Objekten aus dem Body, vor allem der Wurzel.

# Portable Document Format

## Index

xref

0 5

0000000000 65535 f

0000000009 00000 n

0000000071 00000 n

0000000147 00000 n

0000000300 00000 n

- ▶ Start mit xref.
- ▶ Zwei Zahlen: Kleinste referenziert Objektnummer und Anzahl Einträge.
- ▶ Jeder Eintrag auf eine Zeile:
  - ▶ 10-stellige Byteposition des Objekts mit führenden Nullen,
  - ▶ 5-stellige Versionsnummer mit führenden Nullen,
  - ▶ ein Buchstabe n = frei, f = belegte,
  - ▶ Zeilenumbruch
- ▶ Byte-Position herausfinden: grep -b "2 0 obj" doc.pdf

# Portable Document Format

## Trailer

```
trailer
<< /Size 5
    /Root 1 0 R
>>
startxref
506
%%EOF
```

- ▶ Start mit trailer.
- ▶ Anzahl Objekte (wie in xref) und mindestens der Verweis auf die Wurzel des Dokuments.
- ▶ Nach startxref steht auf neuer Zeile die Byte-Position des Anfangs des Index.
- ▶ %%EOF
- ▶ Eine PDF-Datei wird vom Ende her verarbeitet.

# Portable Document Format

## Fehlerquellen

- ▶ PDF werden nicht von Menschen gelesen oder geschrieben.
- ▶ Editor kann bei UTF-8 zusätzliche Byte-Sequenz am Anfang setzen, was als Fehler interpretiert werden.
- ▶ Einzelne Zeichen werden vom Editor nicht richtig codiert gespeichert. Trat beim Konvertieren mit pdf2ps beim Bindestrich von %PDF-1.4 auf, noch einmal neu das Zeichen getippt und gespeichert.
- ▶ Bei Fehlern Hexdump der Datei ansehen, z.B. mit "hexdump -c doc.pdf"

# Inhalt

Einführung

Mathematische und geometrische  
Grundlagen

Bildrepräsentation

**Raytracing**

Transformationsmatrizen

Projektionen

Graphik-Pipeline

OpenGL

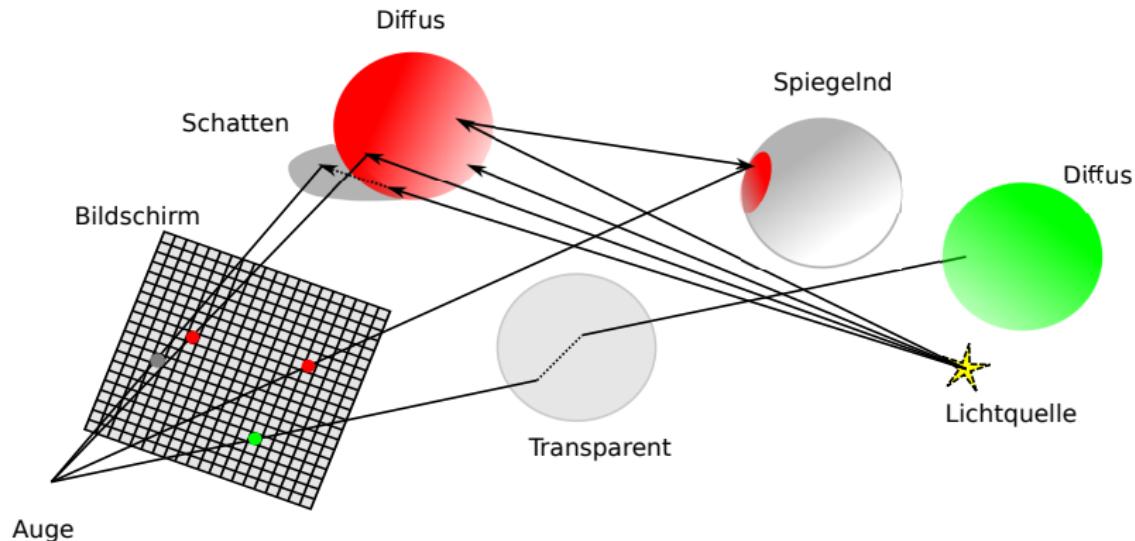
Texturabbildungen

Schattierung von Oberflächen

Prüfung

# Raytracing

## Prinzip

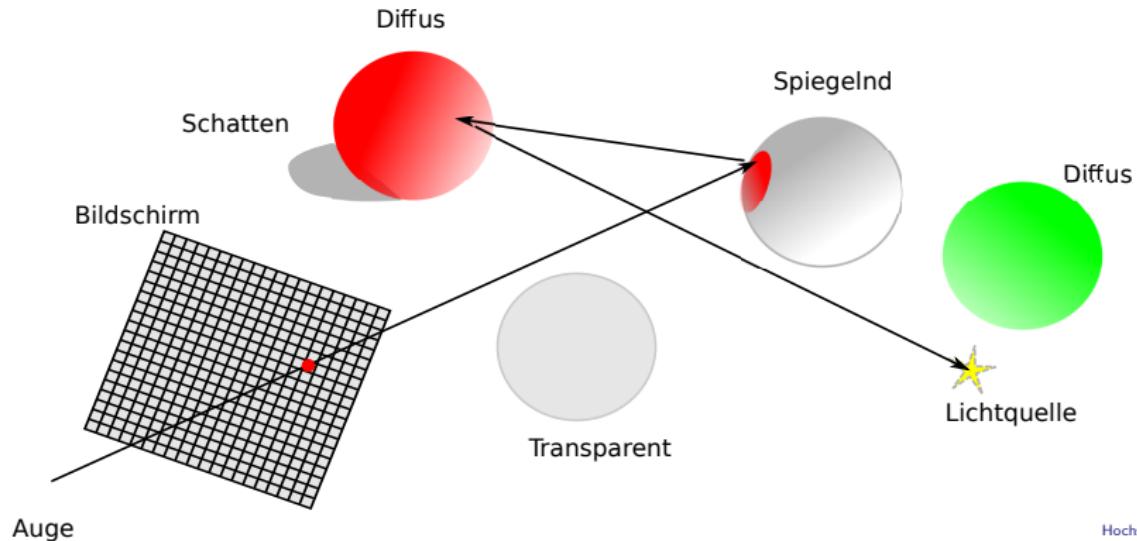


- ▶ Ausbreitung des Lichts im Raum von einer Lichtquelle zu einem Auge.
- ▶ Unendlich viele Lichtstrahlen. Nur wenige treffen ins Auge auf Netzhaut.
- ▶ Bildschirm ist nach vorne verlagerte „Netzhaut“.

# Raytracing

## Rückwärts-Strahlverfolgung

- ▶ Raytracing in Vorwärtsrichtung vom Licht nicht zielgerichtet.
- ▶ Licht rückwärts vom Auge durch Pixel des Bildschirms zum sichtbaren Objekt verfolgen (**Backwards-Raytracing**).
- ▶ Auftreffpunkt Sehstrahl mit sichtbaren Objekt bestimmen.
- ▶ Bei Spiegel: Reflexionsstrahl weiterverfolgen.
- ▶ Bei Transparenz: Refraktionstrahl weiterverfolgen.



# Raytracing

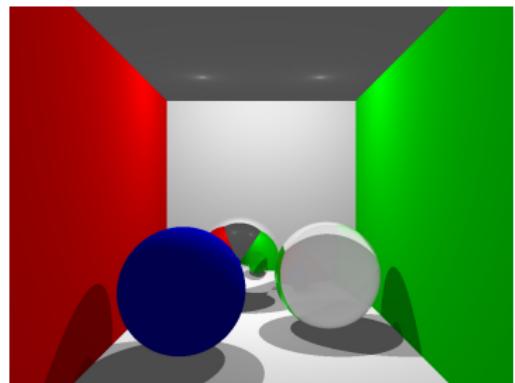
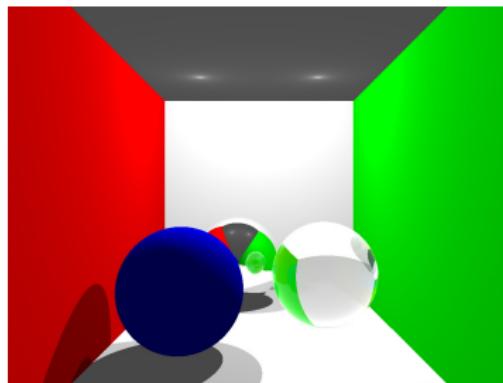
## Cornell-Box



- ▶ Wie nah an der Realität ist ein Bilderzeugungsverfahren (**renderer**, to render sth. visible)?
- ▶ **Cornell** university **box**, 1985: Test, um Güte des Radiosity-Verfahrens zu bestimmen.
- ▶ Nach vorne offenes „Zimmer“ mit einer Lichtquelle an der Decke, roten linken Wand, grüner rechten. Ansonsten weißen Wände.
- ▶ Gegenstände können hineingelegt werden.
- ▶ <http://hatchstudios.com/work/cornell-box-physical-model>

# Raytracing

## Cornell-Box



- ▶ Ohne physikalisches Objekt als Testfall möglich.
- ▶ Test lässt sich gegebenen Anforderungen anpassen.
- ▶ Mit Povray erzeugtes Bild (links), mit eigener Implementierung (rechts).
- ▶ Zwei punktförmige Lichtquellen, statt Flächenstrahler.
- ▶ Wände bestehen aus sehr sehr großen Kugeln.

# Raytracing

- ▶ Es wird über Pixel der Rastergraphik iteriert: **image-ordered rendering**.
- ▶ Für jedes Pixel muss der Sehstrahl erzeugt werden.
- ▶ Für jeden Sehstrahl muss der Schnittpunkt mit dem sichtbaren Objekt ermittelt werden.
- ▶ Das Pixel muss mit der Farben des sichtbaren Objekts gefärbt werden (**schattieren, shading**).
- ▶ Farbe kann in RGB-Anteile aufgeteilt und mit Gleitkommazahlen im Wertebereich [0; 1] codiert werden. Color oft ein 3D-float-Vector.
- ▶ Umrechnung `static_cast<int>(255 * c.get_red())`.

```
Screen screen(WIDTH, HEIGHT);
for (y = 0; y < HEIGHT; y++)
    for (x = 0; x < WIDTH; x++)
        Ray ray = ...
        Object object = find_nearest_object( ray )
        Color color = shade(object, ... )
        screen.set_pixel(x,y, color)
```

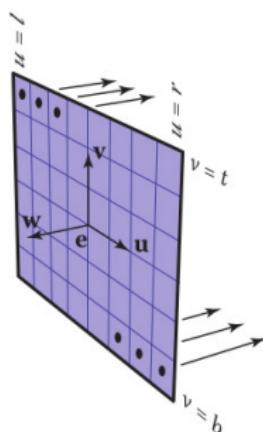
# Raytracing

Sehstrahl

Gegeben Kamera mit Augenpunkt  $e$  und lokalem orthogonalem Koordinatensystem  $\vec{u}$ ,  $\vec{v}$  und  $\vec{w}$  sowie Abstand  $d$  zum Bildschirm (Projektionsfläche). Koordinaten des Bildschirms ( $I, r, b, t$ ).

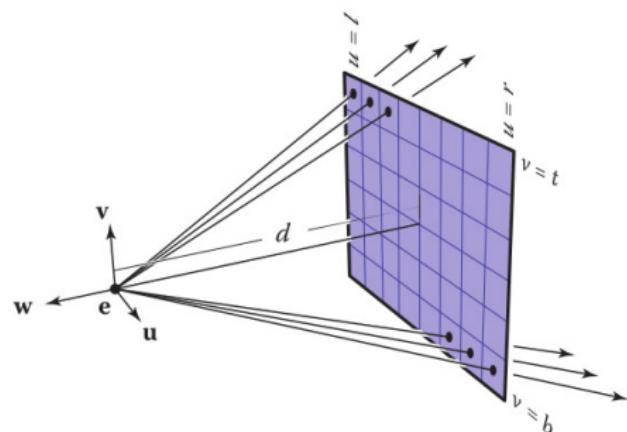
Auflösung  $n_x, n_y$ .

- ▶  $\vec{v}$  parallel zum Bild, zeigt nach oben.
- ▶  $-\vec{w}$  ist Blickrichtung.
- ▶  $\vec{u} = \vec{v} \times \vec{w}$ .



Parallel projection

same direction, different origins



Perspective projection

same origin, different directions

# Raytracing

## Sehstrahl

- ▶ Koordinate  $(p_u, p_v)$  des Pixels  $(i, j)$

$$p_u = l + (r - l)(i + 0,5)/n_x$$

$$p_v = b + (t - b)(j + 0,5)/n_y$$

- ▶ Parallelprojektion:  $ray.origin = e + p_u * \vec{u} + p_v * \vec{v}$  und  
 $ray.direction = -\vec{w}$
- ▶ Zentralprojektion:  $ray.origin = e$  und  
 $ray.direction = -d\vec{w} + p_u \vec{u} + p_v * \vec{v}$

```
Camera camera = ...
```

```
Ray ray = camera.get_ray(i, j)
```

# Raytracing

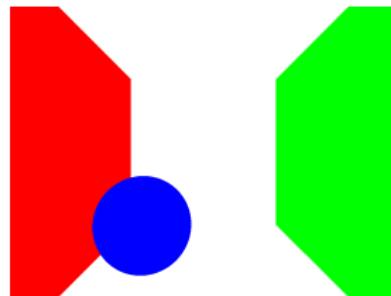
## Sichtbares Objekt

- ▶ Schnittpunktalgorithmus Sehstrahl  $ray.origin + t \cdot ray.direction$  mit Objekt nötig.
- ▶  $t > 0$ : Objekt potentiell sichtbar. Minimumssuche über alle Objekte

```
find_visible_object( ray )
Object visible_object = null;
minimal_t = INFINITY
for (Object o : get_objects() )
    intersects, t = o.intersects(ray, t, ... ))
    if (intersects and t > 0)
        visible_object = o
        minimal_t = t
return visible_object
```

# Raytracing

## Raycasting



- ▶ **Raycasting:** Nur Primärstrahlen werden berücksichtigt.
- ▶ Keine Reflexion oder Transmission möglich.
- ▶ Schattierung hier: Direkt Farbwert des Materials übernehmen, kein Licht berücksichtigen.

```
Scene scene = ...  
Screen screen(WIDTH, HEIGHT);  
Camera camera = ....  
for (y = 0; y < HEIGHT; y++)  
    for (x = 0; x < WIDTH; x++)  
        Ray ray = camera.get_ray(x,y)  
        Object object = scene.find_nearest_object( ray )  
        screen.set_pixel(x,y, object.get_material().get_color())
```

# Raytracing

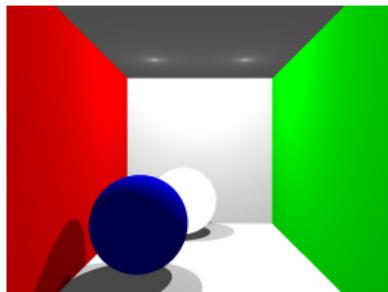
## Lambertian Beleuchtung

- Anteil **diffuser Reflexion** matter Oberflächen, um Objekten Plastizität zu verleihen.
- Johann Heinrich Lambert, 1760.

**Gegeben** Oberflächennormale  $\vec{n}$  eines Oberflächenpunkts, Blickrichtung  $\vec{v}$  zum „Auge“, Richtung zur Lichtquelle  $\vec{l}$ , dessen Lichtintensität/-farbe  $I$  und Anteil diffusen Lichts  $k_d$ .

**Gesucht** Intensität des Lichts (Farbe)  $L$ , die gesehen wird.

$$L = k_d \cdot I \max\{0, \cos \Theta\} = k_d \cdot I \max\{0, \vec{n} \cdot \vec{l}\}$$

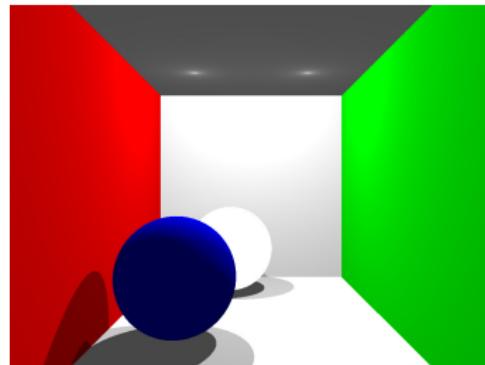


- Vektoren müssen normalisiert sein.
- $\Theta$  ist Winkel zwischen  $\vec{n}$  und  $\vec{l}$
- Intensität hängt nicht von Einfallswinkel des Sehstrahls ab.
- Schatten ließen sich nicht abschalten.

# Raytracing

## Schatten

- ▶ Bei  $n$  Lichtquellen  $\vec{l}_i$  müssen Anteile addiert werden.
- ▶ **Schatten** entsteht, wenn für eine Lichtquelle der Strahl vom betrachteten Punkt zur Lichtquelle durch ein Objekt blockiert wird.
- ▶ Anteil  $k_a$  **ambientes** Licht für Grundhelligkeit hinzufügen, damit völlig schattige Flächen nicht total schwarz werden.



- ▶  $n'$  Lichtquellen  $\vec{l}'_i$  berücksichtigen, die keinen Schatten verursachen:

$$L = k_a + k_d \sum_1^n \cdot \vec{l}'_i \max\{0, \vec{n} \cdot \vec{l}'_i\}$$

- ▶ Max. Farbwerte begrenzt: Überlauf durch Durchschnitt verhindern:

$$L = k_a + k_d \frac{1}{n} \sum_1^n \cdot \vec{l}'_i \max\{0, \vec{n} \cdot \vec{l}'_i\}$$

# Raytracing

## Schatten

### ► Vorsicht!

- ▶ Bei Gleitkommazahlen oder numerische Ungenauigkeiten der Schnittpunktberechnung, kann Schnittpunkt  $p$  hinter der Oberfläche sein.
- ▶ Schattenstrahl hat Schnittpunkt mit Oberfläche des Objekts oder Hinterseite anderer Objekte.
- ▶ Resultat sind oftmals dunkle Punkte auf der Oberfläche, die im Licht ist.
- ▶ **Schattenakne**
- ▶ Schattenstrahl mit  $p + \epsilon \cdot \vec{n}$  bilden.

### ► Optimierung

- ▶ Wahrscheinlichkeit hoch, dass bei einem Schatten, der nächste Schattenstrahl durch das zuvor blockierende Objekt wurde.
- ▶ **Schattenpuffer:** Objekte, die oft oder zeitlich vorher mit Schattenstrahl Schnittpunkt hatten, zuerst prüfen.
- ▶ Einfachste Schattenpuffer: Nur das Objekt, was zuletzt einen Schattenstrahl blockiert hat speichern und prüfen.

# Raytracing

## Schatten

```
Scene scene = ...
Screen screen(WIDTH, HEIGHT);
Camera camera = ....
for (y = 0; y < HEIGHT; y++)
    for (x = 0; x < WIDTH; x++)
        Ray ray = camera.get_ray(x,y)
        // normal, t-Param, intersection, object
        HitContext hit_context = null
        hit_context = scene.find_nearest_object(scene, ray)
        Color color = BACKGROUND_COLOR
        Light [] lights
        if (hit_context != null)
            lights = scene.find_light_sources( hit_context )
            n = scene.get_lights().size()
            color = lambertian(n, lights, hit_context)
            screen.set_pixel(x,y, object.get_material().get_color())
```

# Raytracing

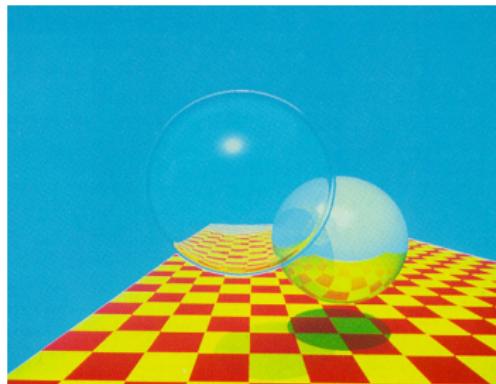
## Schatten

```
find_light_sources( hit_context )
    Light [] visible_lights
    for (Light light : lights  )
        Ray shadow_ray(hit_context.intersection,
                        light.position - hit_context.intersection)

    HitContext shadow_context = scene.find_nearest_object( shadow )
    if (shadow_context == null)
        visible_lights.add( light )
    return visible_lights
```

# Raytracing

## Whitted-Style



- ▶ Turner Whitted, 1980.
- ▶ Reflexionen und **Transmission**.
- ▶ Schatten.
- ▶ Strahlrückverfolgung kann rekursiv implementiert werden.
- ▶ Abbruch bei vorgegebener Tiefe oder wenn der Strahl ins „Leere“ (Horizont) trifft.
- ▶ Reflexion (schon behandelt)
- ▶ (Abb. Wikipedia)

# Raytracing

## Transmission / Refraktion

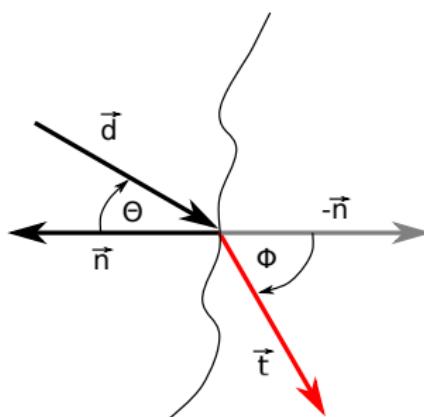
Gegeben: Zwei Einheitsvektoren  $\vec{d}$  und  $\vec{n}$ . Brechungskoeffizienten  $\eta_1$  und  $\eta_2$ .

Gesucht: Transmissionsvektor  $\vec{t}$ , der dem *Snellschen Gesetz* genügt.

- ▶  $\Theta$  sei der Winkel zwischen  $\vec{d}$  und  $\vec{n}$ ,  $\Phi$  der zwischen  $\vec{t}$  und  $\vec{n}$ .
- ▶  $\eta_1$  und  $\eta_2$  sind Materialeigenschaften, Vakuum hat 1 und Glas 1,52.
- ▶ **Snellsche Gesetz:**  
 $\eta_1 \sin \Theta = \eta_2 \sin \Phi$ .
- ▶ Es sei  $n := \frac{\eta_1}{\eta_2}$ .
- ▶ Lösung:

$$\vec{t} = n \cdot \left( \vec{d} - (\vec{d} \cdot \vec{n}) \cdot \vec{n} \right) - \vec{n} \sqrt{1 - n^2 \cdot (1 - (\vec{d} \cdot \vec{n})^2)}$$

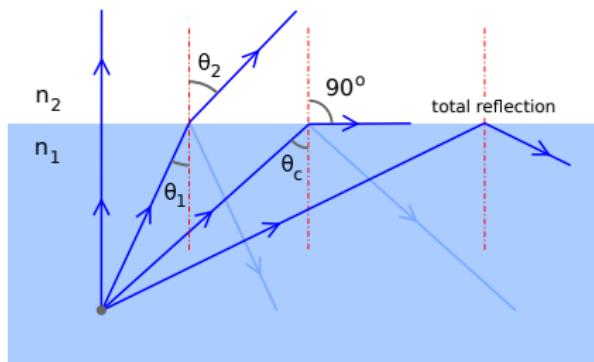
- ▶ Falls Term unter Wurzel kleiner Null: interne Reflexion



# Raytracing

## Transmission / Refraktion

- ▶ Interne Reflektion tritt nur bei Austritt von optisch dichterem Medium zu weniger dichtem auf (nur dann wird Term unterhalb Wurzel negativ, da  $\sin \Phi > 0$  für  $-90^\circ < \Phi < 90^\circ$  gilt).
- ▶ Beispiel: Unter Wasser zum Himmel schauen, dann gilt  $n = \frac{1,333}{1,02} > 1$ . Bei großem Winkel wird Term negativ.



(Abb. Wikipedia)

# Raytracing

## Schlicksche Approximation

- ▶ Auch durchlässige Materialien wie Glas reflektieren einen Teil des Lichts  $R(\Theta)$  abhängig vom Einfallswinkel und den Brechungskoeffizienten  $\eta_1$  und  $\eta_2$ .
- ▶ Genaue physikalische Modellierung rechenintensiv.
- ▶ Schlicksche Approximation:

$$R_0 := \left( \frac{\eta_1 - \eta_2}{\eta_1 + \eta_2} \right)^2$$

$$R(\Theta) := R_0 + (1 - R_0)(1 - \cos \Theta)^5$$

- ▶  $\cos \Theta = -\vec{d} \cdot \vec{n}$
- ▶ Verzweigende Rekursion, da  $1 - R(\Theta)$  Refraktionsanteil ist.
- ▶ Falls  $R_0$  zu klein, keine Reflexion.
- ▶ Falls  $R_0$  zu groß, keine Transmission.

# Raytracing

## Rekursiv

```
Scene scene = ...
Screen screen(WIDTH, HEIGHT);
Camera camera = ....
for (y = 0; y < HEIGHT; y++)
    for (x = 0; x < WIDTH; x++)
        Ray ray = camera.get_ray(x,y)
        Color color = trace(ray, scene, ...)
        screen.set_pixel(x,y, object.get_material().get_color())
```

# Raytracing

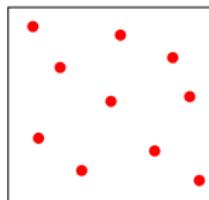
## Rekursiv

```
trace(scene, ray, d)
Color color = BLACK
Ray reflective
Ray refracted
if (depth != 0)
    HitContext hit = scene.find_nearest_object(scene, ray)
    Material material = hit.object.material
    if (material.is_reflective())
        reflective = ray.get_reflective( ... )
        color = color + K_R * trace(scene, reflective, d-1)
    if (material.is_transmissive())
        // schlicks approx R = R(Theta)
        if (refracted ray = refraction( ... ))
            color += K_T * (1-R) * trace(scene, refracted, d-1)
        if (reflective_ray = reflective( ... ))
            color += K_T * R(Theta) * trace(scene, reflective, d-1)
    ...
color += lambertian(n, lights, hit_context)
```

# Raytracing

## Abtastprobleme

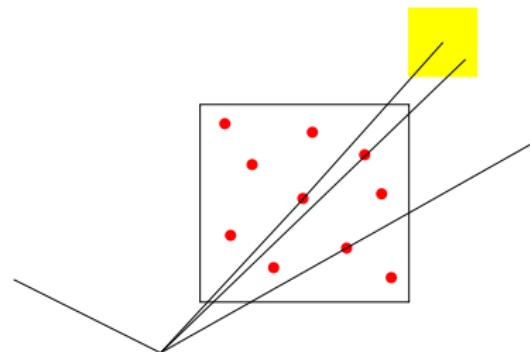
- ▶ Gleichmäßiges Raster führt zu Abtastproblemen (Alias-Effekten):
  1. Treppenstufen bei Übergängen zwischen Objekten. Insbesondere bei starkem Farbkontrast.
  2. Kleine, aber sichtbare Objekte werden vom Sehstrahl nicht getroffen (zum Raster gleichmäßig ausgerichtetes, sichtbares Netz).
- ▶ Lösung zu 1.: Auflösung erhöhen, z.B. horizontal und vertikal verdoppeln. Pixelwert als Durchschnitt aus vier Pixel berechnen.
- ▶ Reduziert nur Wahrscheinlichkeit von Problem 2.
- ▶ Lösung: Auflösung erhöhen und ungleichmäßiges Raster pro Pixel verwenden (**Stochastisches Raytracing, distributed raytracing**).
- ▶ Natur / Evolution: Rezeptoren auf Netzhaut sind zufällig verteilt.



# Raytracing

## Weiche Schatten

- ▶ Distributive Raytracing erhöht Rechenaufwand pro Pixel.
- ▶ Auch bei Schattenstrahlen anwenden.
- ▶ Bei Flächenstrahler (Sonne wirkt z.B. wie ein Flächenstrahler) ergeben sich auch weiche Schatten, da nicht jeder Schattenstrahl die Licht emittierende Fläche trifft.



# Inhalt

Einführung

Mathematische und geometrische  
Grundlagen

Bildrepräsentation

Raytracing

Transformationsmatrizen

Projektionen

Graphik-Pipeline

OpenGL

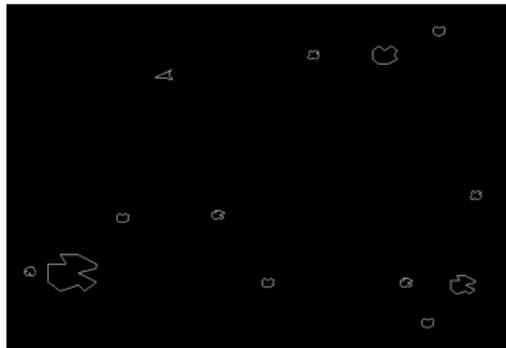
Texturabbildungen

Schattierung von Oberflächen

Prüfung

# Transformationsmatrizen

- ▶ Grafische Objekte werden oft verschoben (Translation), vergrößert (Skalierung), gedreht (Rotation), um vom **Objektraum** zum **Modelraum** transformiert zu werden.
- ▶ **Objektraum:** Modelliertes Objekt mit lokalem Koordinatensystem.
- ▶ Beispiel: Gleiches Objekt, z.B. Asteroid, an mehreren Stellen im Modellraum in unterschiedlicher Größe zeichnen und bewegen.
- ▶ Position von Vektoren (Eckpunkte Polygon) müssen geändert werden.
- ▶ Operationen werden mit Matrizen implementiert.



# Transformationsmatrizen

## Skalierung

**Gegeben** Objekt mit Koordinaten  $p_1, \dots, p_n$ , einen Skalierungsfaktor  $a \in \mathbb{R}$ .

**Gesucht** Neue Koordinaten,  $p'_i = a \cdot p_i$

- Asteroids: Mittlere Asteroid ist zweimal größer als kleiner, der Große viermal.

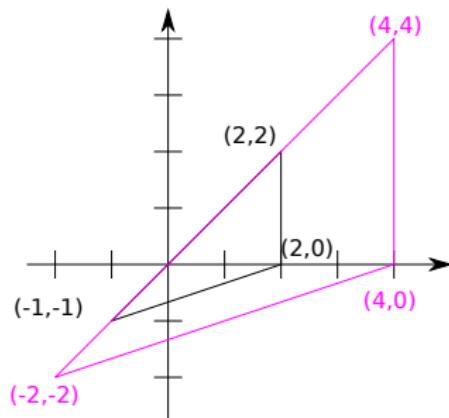
$$\begin{pmatrix} a & 0 \\ 0 & a \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \end{pmatrix} = \begin{pmatrix} a \cdot p_x & 0 \\ 0 & a \cdot p_y \end{pmatrix}$$

- Streckung an einer Raumachse:

$$\begin{pmatrix} a & 0 \\ 0 & 1 \end{pmatrix}$$

- In drei Dimensionen:

$$\begin{pmatrix} a_1 & 0 & 0 \\ 0 & a_2 & 0 \\ 0 & 0 & a_3 \end{pmatrix}$$



# Transformationsmatrizen

## Translation

**Gegeben** Objekt mit Koordinaten  $p_1, \dots, p_n$ , einen Richtungsvektor  $\vec{v}$ .

**Gesucht** Neue Koordinaten,  $p'_i = p_i + \vec{v}$

- ▶ Statt Vektoraddition mit Matrixmultiplikation  $p'_i = A \cdot p_i$
- ▶ Beispiel  $\mathbb{R}^2$ ,  $p = (p_x, p_y)$ ,  $v = (v_x, v_y)$
- ▶ Mit  $2 \times 2$ -Matrix keine (sinnvolle) Lösung möglich, aber mit  $3 \times 3$ -Matrix:

$$\begin{pmatrix} p_x + v_x \\ p_y + v_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & v_x \\ 0 & 1 & v_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

- ▶ Skalierung:

$$\begin{pmatrix} a \cdot p_x \\ a \cdot p_y \\ 1 \end{pmatrix} = \begin{pmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

- ▶ **Homogene Koordinaten.** Vektoren werden um eine Dimension (hier mit Wert 1) erweitert.

# Transformationsmatrizen

Umrechnung von homogenen zu kartesischen Koordinaten

**Gegeben** Homogene Koordinaten  $(x : y : z : a)$  mit  $a \in \mathbb{R}, a \neq 0$

**Gesucht** Kartesische Koordinaten  $(x', y', z')$

$$x' = \frac{x}{a}, y' = \frac{y}{a}, z' = \frac{z}{a}$$

- Bei  $a = 1$  nichts zu tun, aber  $a$  wird nicht immer 1 sein (später)!

# Transformationsmatrizen

Wieso so kompliziert?

- ▶ Uniforme Repräsentation gebräuchlicher Operationen, keine Spezialbehandlung nötig (Code einfacher).
- ▶ Bei Grafikkarten wird parallel in Hardware multipliziert (kein Performanceverlust).
- ▶ CPU berechnet für eine Reihe von Transformationen auf ein Objekt bestehend aus vielen Punkten nur eine Matrix.
- ▶ Nicht transformierte Objekte verbleibt im Speicher der Grafikkarte (Flaschenhals zwischen CPU und Grafikkarte)
- ▶ 3D-APIs basieren zwangsweise auf Transformationsmatrizen.
- ▶ Schreibweise homogene Koordinaten meist  $(x_1 : x_2 : 1)$  statt  $(x_1, x_2, 1)$

# Transformationsmatrizen

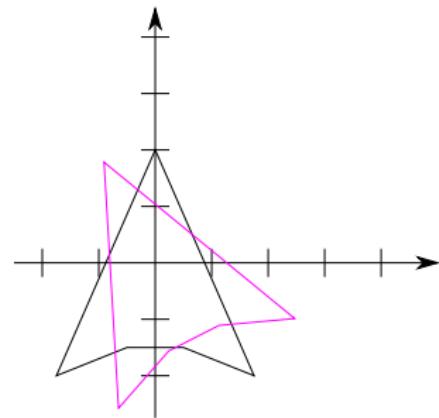
## Drehung

- ▶ Drehung eines Objekts um einen Winkel  $\alpha$  in einer Ebene .
- ▶ Asteroids: Raumschiff wird um seinen Mittelpunkt (gegen den Uhrzeigersinn) vom Spieler in der xy-Ebene gedreht.
- ▶ Drehung in xy-Ebene, yz-Ebene und xz-Ebene.

$$\begin{pmatrix} \cos \alpha & -\sin \alpha & 0 & 0 \\ \sin \alpha & \cos \alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha & 0 \\ 0 & \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} \cos \alpha & 0 & -\sin \alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin \alpha & 0 & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



# Transformationsmatrizen

Reihenfolge der Operationen wichtig

- ▶ Matrizenmultiplikation ist nicht kommutativ!
- ▶ Reihenfolge hängt von der Anwendung ab.
- ▶ Mitte des Raumschiffs liegt im Nullpunkt liegen.
- ▶ Erst Skalieren (Mitte ändert sich nicht), Drehen und dann Verschieben.
- ▶ Größe verdoppeln, um  $45^\circ$  drehen und dann Richtung  $(12,5 : -5,75 : 1)$  verschieben.

$$\begin{pmatrix} 1 & 0 & 12,5 \\ 0 & 1 & -5,75 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0,707 & -0,707 & 0 \\ 0,707 & -0,707 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- ▶ Matrix einmal ausmultiplizieren und auf jeden Punkt anwenden.
- ▶ Anzahl Additionen und Multiplikationen sinkt gegenüber direkt implementierten Operationen mit der Anzahl der Operationen.

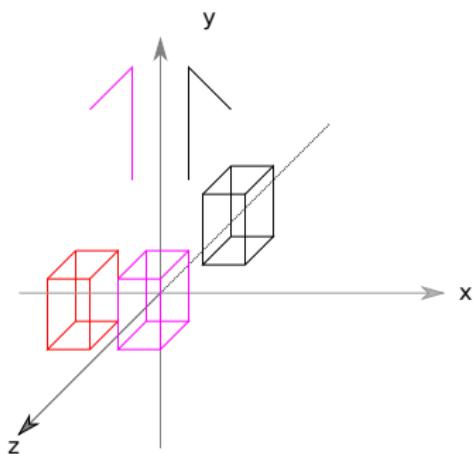
# Transformationsmatrizen

## Spiegelung

- ▶ Spiegelung ist Sonderfall einer Skalierung mit negativen Faktor.
- ▶ Spiegelung an y-Achse bzw. xy-Ebene:

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

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



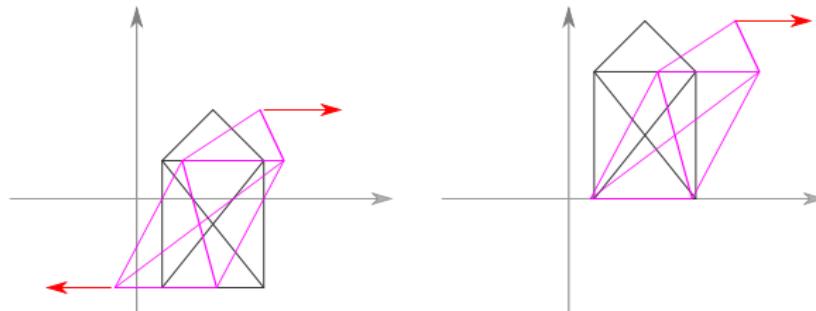
- ▶ Wie transformiert man den roten Kasten aus dem schwarzen?

# Transformationsmatrizen

## Scherung

- ▶ Zwei parallele Kräfte wirken in entgegengesetzter Richtung auf einen Körper aus und verformen ihn.
- ▶ Beispiel (rechts): x-Koordinate wird bei größerer y-Koordinate weiter nach rechts verschoben.  $x' = x + m \cdot y$ .

$$\begin{pmatrix} 1 & 0,2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + 0,2 \cdot p_y \\ p_y \\ 1 \end{pmatrix}$$

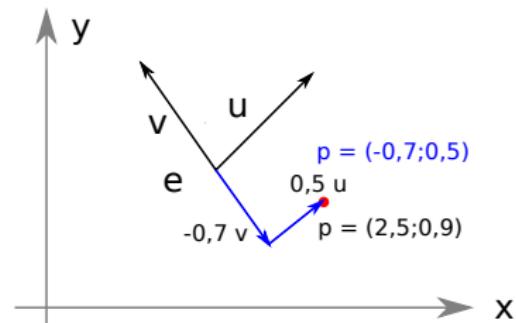


- ▶ Baum „wieg“ sich im Wind mit linear änderndem  $m \in [-0,5; 0,5]$

# Transformationsmatrizen

## Lokale Koordinaten

- ▶ Jede  $n$ -linear unabhängigen Vektoren  $\vec{u}, \vec{v}, \dots$  und ein Punkt  $e$  spannen einen  $n$ -dimensionalen Vektorraum auf.
- ▶ Jeder Punkt  $p$  kann mit  $p = e + p_u \cdot \vec{u} + p_v \cdot \vec{v}$  repräsentiert werden.
- ▶  $(p_u, p_v)$  sind lokale Koordinaten.
- ▶  $\vec{u}, \vec{v}$  und  $e$  bilden ein **lokales Koordinatensystem**.
- ▶ Von lokalen zu globalen Koordinaten (frame-to-canonical):



$$\begin{pmatrix} u_x & v_x & e_x \\ u_y & v_y & e_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_u \\ p_v \\ 1 \end{pmatrix} = e + p_u \cdot \vec{u} + p_v \cdot \vec{v}$$

- ▶ Von globalen zu lokalen Koordinaten (canonical-to-frame):

$$\begin{pmatrix} u & v & e \\ 0 & 0 & 1 \end{pmatrix}^T$$

# Inhalt

Einführung

Mathematische und geometrische  
Grundlagen

Bildrepräsentation

Raytracing

Transformationsmatrizen

Projektionen

Graphik-Pipeline

OpenGL

Texturabbildungen

Schattierung von Oberflächen

Prüfung

# Projektionen

- Bisher: Objekte wurden vom Objektraum in den Modelraum (world space) verschoben, skaliert, gedreht, (vervielfacht), ...

**Gegeben:** Objekte im  $\mathbb{R}^3$  Modelraum

**Gesucht:** Gesucht ist  $\mathbb{R}^2$  Bild im Bildraum (image space, screen space) mit Hilfe einer View-Transformation.

- Raytracing: Bild wurde mit durch Iterieren über die Pixel erzeugt (**pixel order rendering**).
- Im Folgenden: Bild wird durch Iterieren über die Objekte erzeugt (**object order rendering**).
- **Viewing-Transformation:** Matrix, die Punkte im Modelraum auf Pixel im Bildraum transformiert.
  1. Kamera-/Augen-Transformation
  2. Projektions-Transformation
  3. Viewport-/Fenstertransformation

# Projektionen

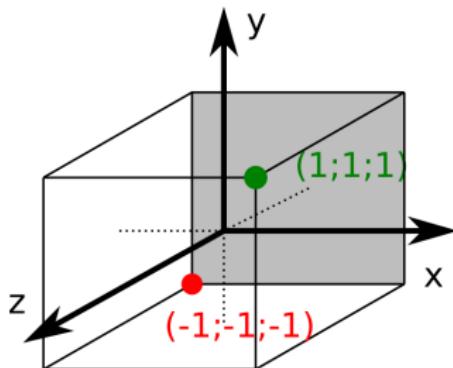
## Viewport-Transformation

Gegeben: Punkte des  $\mathbb{R}^3$  Modellraums in einem Quader

$[-1; 1] \times [-1; 1] \times [-1; 1]$ . Sichtrichtung in negativer z-Richtung.

Gesucht: Parallelprojektion auf hintere Fläche  $[r; l] \times [t; b]$  im  $\mathbb{R}^2$  Bildraum.

- ▶  $n_x$  horizontale,  $n_y$  vertikale Pixel.
- ▶ Vorkommaanteil soll (fast) Pixelkoordinate entsprechen.



$$\begin{pmatrix} x_{screen} \\ y_{screen} \\ -1 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{n_x}{2} & 0 & \frac{n_x-1}{2} & 0 \\ 0 & \frac{n_y}{2} & \frac{n_y-1}{2} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_{canonical} \\ y_{canonical} \\ z_{canonical} \\ 1 \end{pmatrix}$$

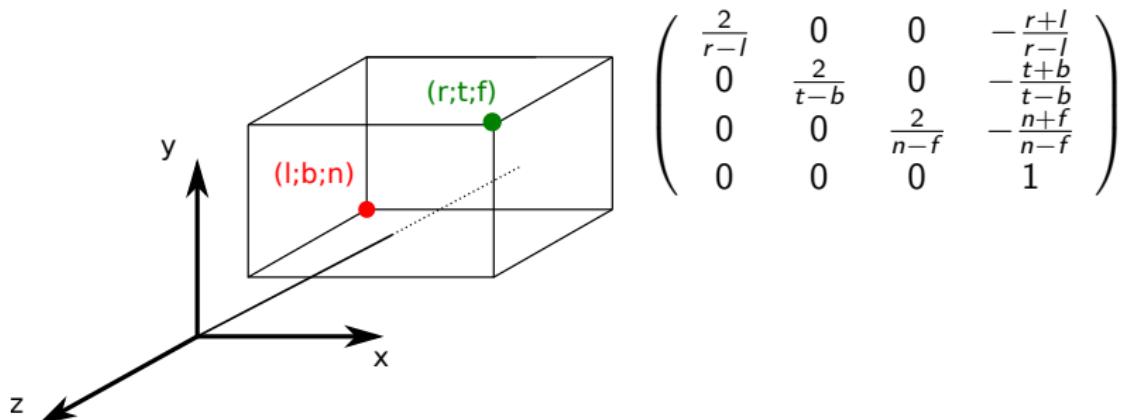
# Projektionen

## Kanonische Sicht

Gegeben: Punkte des  $\mathbb{R}^3$  Modelraums in einem Quader  $[l, r] \times [b, t] \times [f, n]$ .  
Augenpunkt im Nullpunkt, Sichtrichtung in negativer z-Richtung.

Gesucht: Zentralprojektion auf hintere Fläche des Quaders im  $\mathbb{R}^2$  Bildraum.

- Es gilt  $r > l, t > b$  und  $n > f$ .



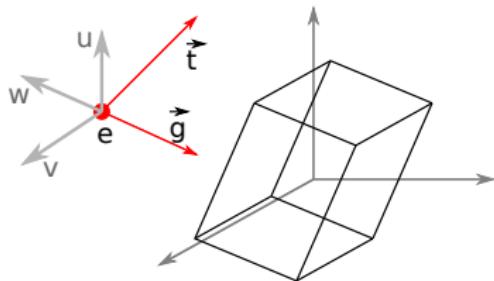
# Projektionen

## Kamera-Transformation

**Gegeben:** Punkte des  $\mathbb{R}^3$  Modelraums . Eine Kamera mit Augenpunkt  $e$ , einer Blickrichtung  $\vec{g}$  und einen Vektor  $\vec{t}$  (zeigt nach „oben“)

**Gesucht:** Verschiebung und Drehung der Punkt, so dass vom globalen Nullpunkt Richtung negativer z-Achse alle Punkte liegen.

- $\vec{t}, \vec{g}$  und  $e$  bilden ein (nicht notwendigerweise orthogonales) lokales Koordinatensystem.



$$\vec{w} := \frac{-\vec{g}}{|\vec{g}|}, \vec{u} := \frac{\vec{t} \times \vec{w}}{|\vec{t} \times \vec{w}|}, \vec{v} := \vec{w} \times \vec{u}$$

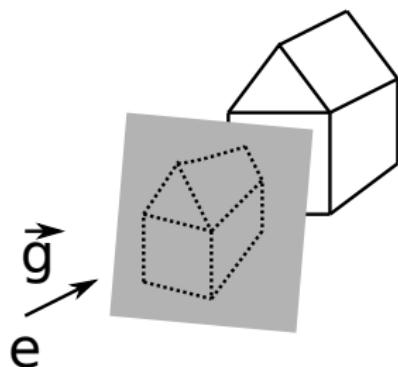
$$\begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

# Projektionen

## Zentralprojektion

**Gegeben:** Punkte des  $\mathbb{R}^3$  Modelraums . Eine Kamera mit Augenpunkt  $e$ , einer Blickrichtung  $\vec{g}$  und einen Vektor  $\vec{t}$  (zeigt nach „oben“) und Abstand  $d$  zu einer Projektionsfläche.

**Gesucht:** Perspektivische Projektion aller Punkte von  $e$  auf die Fläche entlang Blickrichtung.



$$\begin{pmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

- ▶ Strahlensatz anwenden: Es muss durch  $p_z$  geteilt werden.
- ▶ Mit linearen Transformationsmatrizen nicht möglich.
- ▶ Lösung:  $w$ -Komponente auf  $p_z$  setzen, beim Zeichnen durch  $w$  teilen.

# Projektionen

## Zentralprojektion

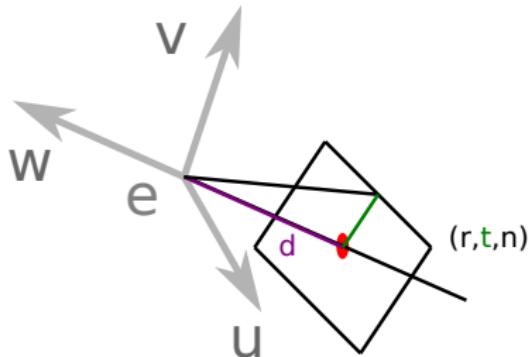
- ▶ Tiefeninformation geht verloren, wird aber später benötigt.
- ▶ z-Koordinaten nicht linear auf eine monotone Funktion abbilden.
- ▶  $p_z \mapsto (n + f) \cdot p_z - f \cdot n$
- ▶ Hat die Eigenschaft:  $p_z = n$  und  $p_z = t$  jeweils auf  $n$  und  $t$  abgebildet werden.

$$\begin{pmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

# Projektionen

## Sichtfeld

- ▶ **Sichtfeld** (field of view): Bildwinkel  $\Theta$  eines optischen Geräts oder Sensors, innerhalb dessen Aufzeichnungen stattfinden können.
- ▶ Beim Menschen **Gesichtsfeld** (visual field): Optische Wahrnehmung des Auges ohne dieses zu bewegen. Ca.  $107^\circ$  horizontal (links und rechts),  $60^\circ$ - $70^\circ$  vertikal nach oben,  $70^\circ$ - $80^\circ$  nach unten.
- ▶ Beim Sichtfeld werden Augenbewegungen mit zugelassen.
- ▶ Alternative Angabe der Zentralprojektion mit Sichtfeld möglich.



- ▶ Annahme:  $l = -r$ ,  $b = -t$  (Bildmitte wird anvisiert),  $n_x$  gegeben. Pixel quadratisch.
- ▶  $d = -n$  ist normalerweise gegebener Abstand.
- ▶ Vertikaler FoV:  $\tan \frac{\Theta}{2} = \frac{t}{|n|}$
- ▶ Alternativen: Horizontaler oder diagonaler FoV.

# Inhalt

Einführung

Mathematische und geometrische  
Grundlagen

Bildrepräsentation

Raytracing

Transformationsmatrizen

Projektionen

Graphik-Pipeline

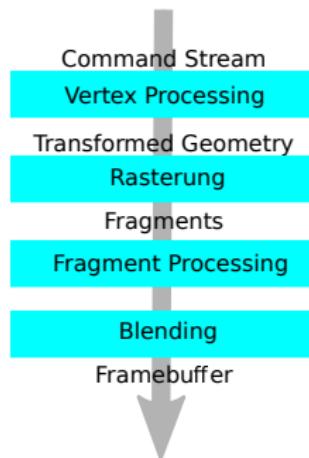
OpenGL

Texturabbildungen

Schattierung von Oberflächen

Prüfung

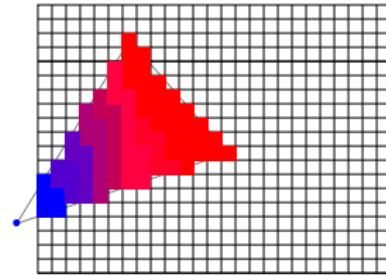
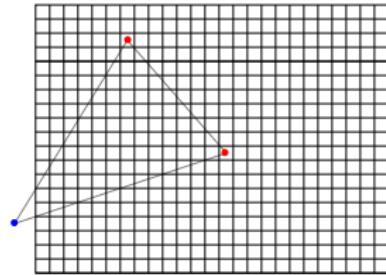
# Graphik-Pipeline



- ▶ Vertex-Processing: Grafische Primitive, die durch Punkte definiert sind, werden verarbeitet, z.B. mit Transformationsmatrizen.
- ▶ Rasterung: Primitive werden in einzelne Fragmente zerlegt. Für jedes Pixel ein Fragment.
- ▶ Fragment-Processing: Fragmente werden geprüft, z.B. Einfärbung.
- ▶ Blending: Farben werden kombiniert, z.B. Fragment-Farbe mit Hintergrund kombiniert.

# Graphik-Pipeline

- ▶ Koordinaten des 3D-Modell (Punkte) können mit den bisherigen Projektionen einschließlich Viewport-Transformation den Bildkoordinaten zugeordnet werden (links).
- ▶ Jeder Koordinaten können Farb- oder andere Materialeigenschaften wie Oberflächennormalen zugeordnet sein.
- ▶ Es fehlt die **Rasterung** der Objekte (Fläche) und Einfärbung der sichtbaren Pixel (rechts).



# Rasterung von Strecken

## Midpoint-Algorithmus

**Gegeben:** Eine Strecke in der x/y-Ebene mit Punkten

$x = (x_0, y_0), y = (x_1, y_1) \in \mathbb{R}^2$  in Bildkoordinaten. Pixel sind quadratisch.

**Gesucht:** Alle Pixel, die von der Strecke überdeckt werden, so dass ein lückenloser Verlauf entsteht.

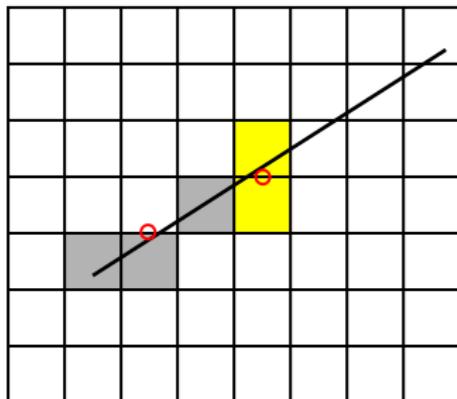
- ▶ Annahme  $x_0 \leq x_1$  (Ansonsten Punkte vertauschen).
- ▶ Steigung  $m$  der Strecke ist  $m = \frac{y_1 - y_0}{x_1 - x_0}$
- ▶ Implizite Repräsentation der Strecke:

$$f(x, y) = (y_0 - y_1) \cdot x + (x_1 - x_0) \cdot y + x_0 \cdot y_1 - x_1 \cdot y_0 = 0$$

- ▶ Wir betrachten  $m \in [0, 1)$
- ▶ Ziel: dünnste Strecke zeichnen, so dass keine Lücken entstehen.

# Rasterung von Strecken

## Midpoint-Algorithmus



- ▶ Pixel  $(x, y)$  sei zuletzt gezeichnetes Pixel (Mitte des Pixels).
- ▶  $(x + 1, y)$  oder  $(x + 1, y + 1)$  auswählen (gelbe Pixel)?
- ▶ Mittleren Punkt auf der horizontalen Strecken dazwischen betrachten:  
 $(x + 1, y + 0, 5)$
- ▶ Liegt der Punkt unterhalb der Strecke: oberes Pixel. Ansonsten unteres Pixel.
- ▶ Start:  $x_0$  und  $y_0$  zur nächsten ganzen Zahl runden.

---

```
1  y = y0
2  for x = x0 to x1 do
3      draw(x, y)
4      if (f(x + 1, y + 0.5) < 0) then
5          y = y + 1
```

---

# Rasterung von Strecken

## Midpoint-Algorithmus Verbesserung

---

```
1  y = y0
2  for x = x0 to x1 do
3      draw(x, y)
4      if (f(x + 1, y + 0.5) < 0) then
5          y = y + 1
```

---

- ▶ Berechnungen vom vorherigen Pixel wiederverwenden.
- ▶  $f(x+1, y) = (y_0 - y_1) \cdot (x+1) + (x_0 - x_1) \cdot y + y_1 \cdot y_0 - x_1 \cdot y_0 = f(x, y) + (y_0 - y_1)$
- ▶  $f(x+1, y+1) = f(x, y) + (y_0 - y_1) + (x_0 - x_1)$

---

```
1  y = y0
2  d = f(x0 + 1, y0 + 0.5)
3  for x = x0 to x1 do
4      draw(x, y)
5      if d < 0
6          y = y + 1
7          d = d + (x1 - x0) + (y0 - y1)
8      else
9          d = d + (y0 - y1)
```

---

# Rasterung von Strecken

## Midpoint-Algorithmus

- ▶ Farbe zwischen Anfangs- und Endpunkt der Strecke kann linear interpoliert werden.
- ▶ Pitteway, M. L. V., 1967: Algorithm for drawing ellipses or hyperbolae with a digital plotter. The Computer Journal. [2]
- ▶ Van Aken, Jerry; Novak, Mark, 1985: Curve-Drawing Algorithms for Raster Displays. Association for Computing Machinery. [3]
- ▶ Bresenham, J. E., 1965. Algorithm for Computer Control of a Digital Plotter. IBM Systems Journal, 4(1). [1]

# Rasterung von Dreiecken

**Gegeben:** Dreick mit drei Eckpunkten  $p_0 = (x_0, y_0)$ ,  $p_1 = (x_1, y_1)$  und  $p_2 = (x_2, y_2)$  in der Bildebene.

**Gesucht:** Rasterung des Dreiecks, so dass keine Lücken entstehen.

- ▶ Den Eckpunkten seien Farbwerte  $c_0, c_1, c_2$  zugeordnet.
- ▶ Interpolation der Farbwerte für ein gerastertes Pixel mit Baryzentrischen Koordinaten.
- ▶ Oder: Bestimmung der sichtbaren Farbe in Eckpunkt mit z.B. Lambertschen Shading mit Normalen der Eckpunkte und dann Interpolation mit Baryzentrischen Koordinaten (Gouraud Shading).
- ▶ Problem: Dreiecke, die Eckpunkte teilen, müssen ohne Lücke dazwischen gerastert werden.
- ▶ Einfachste Lösung: Jedes Pixel, das im Dreick liegt, wird gezeichnet.

# Rasterung von Dreiecken

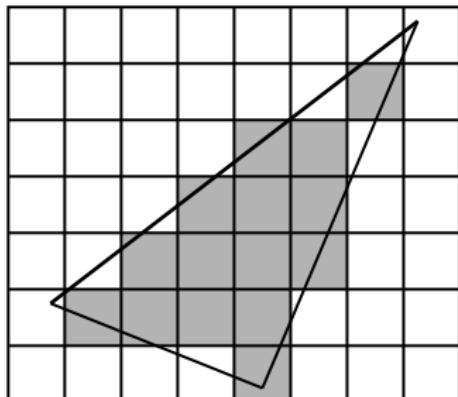
---

```
1  for all x do
2      for all y do
3          compute (u, v, w) for (x, y)
4          if (u ∈ [0, 1] and v ∈ [0, 1] and w ∈ [0, 1]) then
5              c = u · c0 + v · c1 + w · c2
6              drawpixel (x, y) with color c
```

---

- ▶ Wie Pixel  $(x, y)$  auswählen?
- ▶ Brute-Force: Bounding-Box des Dreiecks berechnen und über alle Pixel der Box iterieren.

- ▶  $x_{max} = \lceil \min\{x_0, x_1, x_2\} \rceil$
- ▶  $x_{min} = \lfloor \min\{x_0, x_1, x_2\} \rfloor$
- ▶  $y_{max} = \lceil \min\{y_0, y_1, y_2\} \rceil$
- ▶  $y_{min} = \lfloor \min\{y_0, y_1, y_2\} \rfloor$



# Rasterung von Dreiecken

## Verbesserung

- ▶ Seien  $f_{01}, f_{12}, f_{20}$  jeweils die implizite Repräsentation der Strecken  $\overline{p_0 p_1}, \overline{p_1 p_2}$  und  $\overline{p_2 p_0}$ .
- ▶ Dann gilt  $u = \frac{f_{12}(x,y)}{f_{12}(x_0,y_0)}$ ,  $v = \frac{f_{20}(x,y)}{f_{20}(x_1,y_1)}$ ,  $w = \frac{f_{01}(x,y)}{f_{01}(x_2,y_2)}$

---

```
1  for y = ymin to ymax do
2      for x = xmin to xmax do
3          u = f12(x, y) / f12(x0, y0)
4          v = f20(x, y) / f20(x1, y1)
5          w = f01(x, y) / f01(x2, y2)
6          if (u > 0 and v > 0 and w > 0) then
7              c = u · c0 + v · c1 + w · c2
8              drawpixel (x, y) with color c
```

---

- ▶ Algorithmus kann wie zuvor auch inkrementell verbessert werden.
- ▶ Degenerierte Dreiecke (Strecke) führt zu Division durch 0.

# Tiefenpuffer (z-Buffer)

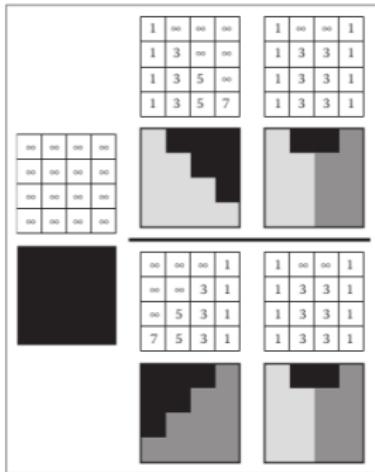
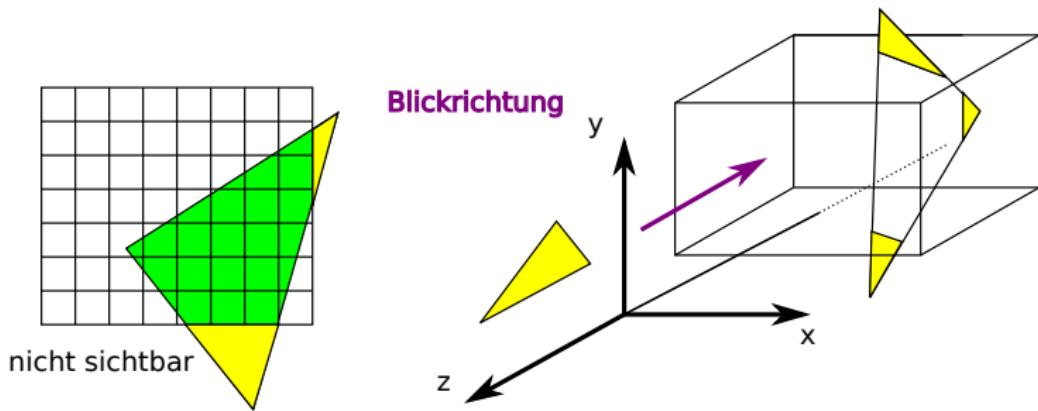


Abb. Fundamentals of Computer Graphics

- Gerastertes Pixel wird nur in Framebuffer (Bildspeicher) geschrieben, wenn zugehöriger Punkt dem Betrachter näher ist.
- Lösung: Zweiten Speicher, der Tiefeninformation eines Pixel (z-Koordinate des zugehörigen Punkts) speichert. **Tiefenpuffer**.
- Wertebereich z-Koordinate ist  $[-1, 1]$ .
- Pixel wird nur gesetzt, wenn z-Koordinate größer als bisherige ist (größer, da wir in negativer z-Richtung blicken!)
- In der Praxis werden nicht negative ganze Zahlen statt Gleitkommazahlen verwendet (Zahlenwerte negieren, ggf. um Eins verschieben, Exponent konstant setzen und Mantisse als ganze Zahl verwenden).

# Clipping

- ▶ Nur Dreiecke, die im Sichtbereich sind, sollen gezeichnet werden.
- ▶ **Clipping:** Abschneiden von Objektteilen an (rechteckigen) Bildausschnitten.
- ▶ Clipping in nicht-homogenen Weltkoordinaten ( $\mathbb{R}^3$ ) gegenüber den sechs Ebenen des Quaders. Also vor Kamera-Transformation.
- ▶ Oder nach kanonischer Form (OpenGL).



# Clipping

## Schnitt Strecke mit Ebene

**Gegeben:** Strecke von  $a$  nach  $b$ . Ebenengleichung  $\vec{n} \cdot \vec{p} + D = 0$ . ( $D = -d$  in  $\vec{n} \cdot \vec{p} = d$ )

**Gesucht:** Punkte auf Strecke, der in der Ebene liegt (falls vorhanden).

1. Prüfe, ob  $a$  und  $b$  sich auf verschiedenen Seiten der Ebene befinden (mit impliziter Form wie bei Midpoint-Algorithmus):

$$\operatorname{sgn}(\vec{n} \cdot a + D) \neq \operatorname{sgn}(\vec{n} \cdot b + D)$$

2. Parameterdarstellung:  $p(t) = a + t \cdot (b - a)$  in Ebenengleichung einsetzen und nach  $t$  auflösen:

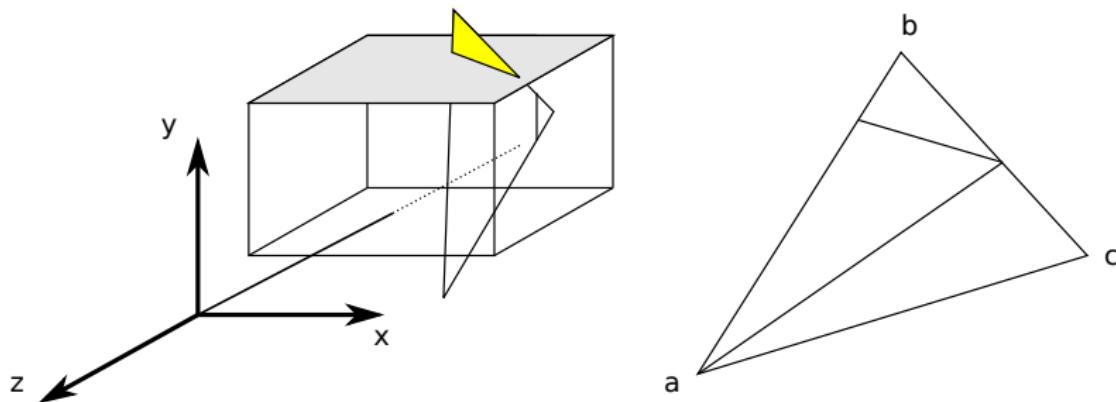
$$t = \frac{\vec{n} \cdot a + D}{\vec{n} \cdot (a - b)}$$

- Das gleiche für den die Strecke vom außen liegenden Punkt zum anderen Dreieckspunkt.

# Clipping

## Dreieck zerteilen

- ▶ Dreieck zerteilen, die innere mit den restlichen Ebenen clippen.
- ▶ Das Dreieck zerfällt immer in drei Dreiecke.
- ▶ Farben und Normalen in den Eckpunkten?



# Inhalt

Einführung

Mathematische und geometrische  
Grundlagen

Bildrepräsentation

Raytracing

Transformationsmatrizen

Projektionen

Graphik-Pipeline

OpenGL

Texturabbildungen

Schattierung von Oberflächen

Prüfung

# OpenGL

## Geschichte

OpenGL (Open Graphics Library) ist eine plattform-unabhängige, performante 2D/3D-Grafik-API-Spezifikation.

- 1980 Iris GL (Integrated Raster Imaging System Graphics Library), Silikon Graphics. Für Iris-Workstation (MC68000, MIPS).
  - 1992 OpenGL 1.0 (Mark Segal, Kurz Akeley). Formalisierung von Iris GL. Plattformunabhängig. Standardisierung durch Architectural Review Board (ARB).
  - 2004 OpenGL 2.0. Einführung von Shader-Sprachen (GLSL).
  - 2008 OpenGL 3.0. Teile der API als veraltet markiert und (3.1) entfernt. Kernfunktionalitäten (3.2), OpenCL (3.3).
  - 2017 OpenGL 4.0 – 4.6. Binäre Shader-Programme. 64-Bit Gleitkommazahlen, Texturkompression, teilweise DX11-Emulation.
- Aktuelle Hardware unterstützt mindestens Kernfunktionen von OpenGL 3.3.

# OpenGL

## Resourcen

- ▶ Khronos-Group heute verantwortlich für Standardisierung.
- ▶ <https://registry.khronos.org/OpenGL/specs/gl/glspec46.core.pdf>
- ▶ Direkte Seite für OpenGL: <https://www.opengl.org/>.
- ▶ Alternativen: Direct-3D (Microsoft), Vulcan (low-level, von AMD zur Khronos-Group), Metal (low-level, Apple), OpenGL ES (embedded / mobile) Teilmenge von OpenGL.
- ▶ OpenGL ist praktisch Standard bei plattform-unabhängigen Grafik-Programmen.
- ▶ Online-Tutorials (auf OpenGL 3.x+ achten):  
<https://www.learnopengl.com>, [www.opengl-tutorial.org](http://www.opengl-tutorial.org), [ogldev.org](http://ogldev.org) (Videos).

# OpenGL

## Resourcen

- ▶ OpenGL ist nur eine Spezifikation, nah an C, keine Implementierung.
- ▶ Zugriff auf plattformabhängige API-Einsprungadressen mit **GLEW** (lädt auch Treiber, DLL, je nach Betriebssystem).
- ▶ **OpenGL-Header-Dateien** Open-Source. Kann üblicherweise über Paketmanager installiert werden.
- ▶ OpenGL benötigt Zugriff auf Hardware, insbesondere Framebuffer (Context).
- ▶ OpenGL besitzt keine Funktionen, um Fenster, UI oder ähnliches zu bauen.
- ▶ **SDL2** / **GLFW** /**SFML** bieten Funktionen, um Context für OpenGL zu erzeugen.

# OpenGL

## Erstes Programm

1. Fenster mit SDL2 erzeugen. `SDL_WINDOW_OPENGL` wichtig.
2. OpenGL-Context mit SDL2 für das Fenster erzeugen.
3. GLEW initialisieren.
4. OpenGL-API-Befehle nutzen.

---

```
1  SDL_Window * window = nullptr;
2  if( SDL_Init( SDL_INIT_VIDEO ) < 0 ) {
3      std :: cout << "SDL_Error: " << SDL_GetError() << std :: endl ;
4      return EXIT_FAILURE;
5  }
6  window = SDL_CreateWindow( "Hello" , SDL_WINDOWPOS_UNDEFINED ,
7      SDL_WINDOWPOS_UNDEFINED , window_width , window_height ,
8      SDL_WINDOW_OPENGL | SDL_WINDOW_SHOWN );
9  if( window == nullptr ) {
10     std :: cout << "SDL_Error: " << SDL_GetError() << std :: endl ;
11     return EXIT_FAILURE;
12 }
```

---

# OpenGL

## Erstes Programm

1. Fenster mit SDL2 erzeugen. `SDL_WINDOW_OPENGL` wichtig.
2. **OpenGL-Context mit SDL2** für das Fenster erzeugen.
3. GLEW initialisieren.
4. OpenGL-API-Befehle nutzen.

---

```
1  SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
2  SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 3);
3  SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK,
4                      SDL_GL_CONTEXT_PROFILE_CORE);
5  SDL_GL_SetAttribute(SDL_GL_CONTEXT_FLAGS,
6                      SDL_GL_CONTEXT_FORWARD_COMPATIBLE_FLAG );
7  SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 24);
8  SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
9
10 SDL_GLContext context = SDL_GL_CreateContext(window);
```

---

# OpenGL

## Erstes Programm

1. Fenster mit SDL2 erzeugen. `SDL_WINDOW_OPENGL` wichtig.
2. OpenGL-Context mit SDL2 für das Fenster erzeugen.
3. **GLEW initialisieren.**
4. OpenGL-API-Befehle nutzen.

---

```
1 GLenum err = glewInit(); //  call after OpenGL render context is created
2 if (GLEW_OK != err) {
3     std::cerr << glewGetErrorString(err) << std::endl;
4 }
```

---

# OpenGL

## Erstes Programm

1. Fenster mit SDL2 erzeugen. `SDL_WINDOW_OPENGL` wichtig.
2. OpenGL-Context mit SDL2 für das Fenster erzeugen.
3. GLEW initialisieren.
4. **OpenGL-API-Befehle nutzen.**

---

```
1  SDL_GL_SetSwapInterval(1);
2  // vsync
3
4  glClearColor ( 1.0, 0.0, 0.0, 1.0 ); // R G B A
5  glClear ( GL_COLOR_BUFFER_BIT );
6  SDL_GL_SwapWindow(window);
7  SDL_Delay(2000);
8
9  glClearColor ( 0.0, 1.0, 0.0, 1.0 );
10 glClear ( GL_COLOR_BUFFER_BIT );
11 SDL_GL_SwapWindow(window);
12 SDL_Delay(2000);
```

---

# OpenGL

## Erstes Programm

- ▶ Am Ende des Programms Ressourcen freigeben.

---

```
1 SDL_GL_DeleteContext(context);
2 SDL_DestroyWindow(window);
3 SDL_Quit();
```

---

# OpenGL

Übersetzen

- ▶ g++ -lSDL2 -lgl -lglew background.cc (Unix, Groß/Kleinschreibung kann anders sein).
- ▶ g++ -lSDL2 -lopengl32 -lglew32 background.cc (MinGW).
- ▶ Reihenfolge teilweise wichtig.
- ▶ Am besten wieder CMake verwenden.

---

```
1 target_link_libraries(background SDL2 GL GLEW)
```

---

# OpenGL

## Dreieck zeichnen

- ▶ Ab OpenGL 3.x müssen Shader verwendet werden. Je nach Implementierung kann es Default-Shader geben. In der Regel bleibt der Bildschirm schwarz ohne eigene Shader.
- ▶ Erst einmal Beispiel ohne die Shader.
- ▶ OpenGL benötigt Koordinaten in kanonischer Form.
- ▶ Geometrische Primitive von OpenGL werden als Folge von Vertices angeben.
- ▶  $\mathbb{R}^n$  für  $n = 2, 3, 4$ . Homogene Koordinaten möglich.
- ▶ Hier  $n = 3$ .
- ▶ Daten müssen zur API (GPU) transferiert werden. GPU hat eigenen Speicher.

---

```
1 float vertices [] = {  
2     -0.5f, -0.5f, 0.0f, // left  
3     0.5f, -0.5f, 0.0f, // right  
4     0.0f, 0.5f, 0.0f } // top
```

---

# OpenGL

## Vertex Buffer Object (VBO)

- ▶ Mit einem Vertex-Buffer-Object (VBO) kann die GPU große Mengen an Daten speichern.
- ▶ OpenGL-API verwaltet VBOs. Verteile für Zugriff Ids (positive ganze Zahl).

---

```
1 GLuint vbo; // unsigned int
2 glGenBuffers(1, &vbo); // schreibt 1 Id eines VBO in vbo
3 GLunit vbos[5];
4 glGenBuffers(5, vbos); // schreibt 5 Ids hintereinander in vbos
5
6 glBindBuffer(GL_ARRAY_BUFFER, vbo); // aktiviert den VBO
7 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
8                 vertices, GL_STATIC_DRAW); // Daten transferieren
```

---

# OpenGL

## Vertex Buffer Object (VBO)

---

```
1 glBindBuffer(GL_ARRAY_BUFFER, vbo); // aktiviert den VBO
2 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
3               vertices, GL_STATIC_DRAW); // Daten transferieren
4
5 glBindBuffer(GL_ARRAY_BUFFER, 0); // deaktiviert aktuellen VBO
```

---

- ▶ GL\_STATIC\_DRAW: Daten im Buffer ändern sich später nicht.
- ▶ GL\_DYNAMIC\_DRAW: Daten im Buffer werden häufiger geändert.
- ▶ Nur ein VBO kann aktiv sein.
- ▶ Befehle, die VBOs ändern, beziehen sich auf aktiven VBO.

# OpenGL

## Vertex Array Object (VAO)

- ▶ Mit einem Vertex-Array-Object (VAO) werden mehrere VBO verwaltet und dessen Speicherlayout beschrieben.
- ▶ Das Speicherlayout wird in den Shadern verwendet. Es ist lediglich eine ganze positive Zahl.

---

```
1  GLuint vao;
2  glGenVertexArrays(1, &vao);
3
4  glBindVertexArray(vao); // aktivieren
5  // folgender Befehl bezieht sich auf aktuellen VBO und VAO
6  glVertexAttribPointer(0, // index des attributes (layout-id)
7                        3, // Anzahl floats eines Attributs (3D-Punkt)
8                        GL_FLOAT, // Datentyp der Werte eines Attribut
9                        GL_FALSE, // Daten sind nicht normalisiert
10                       3 * sizeof(float), // Abstand (stride) zwischen Attributen in bytes
11                       (void*)0); // offset zum ersten Attribut in bytes
12  glEnableVertexAttribArray(0); // aktivieren
```

---

# OpenGL

## Zeichnen

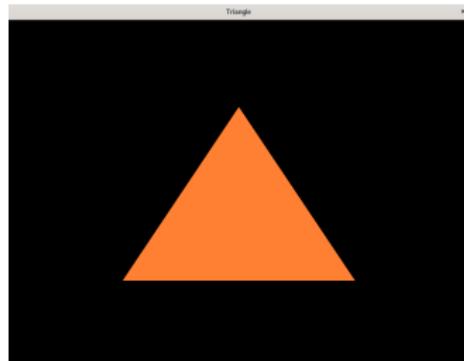
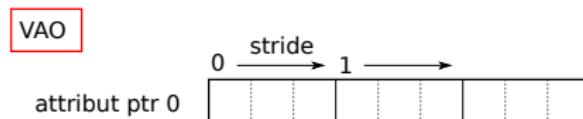
- Shaderprogramm(e) müssen aktiv sein (später)

---

```
1 // aktiver VAO wird gezeichnet:  
2 glDrawArrays(GL_TRIANGLES, // Dreick zeichnen  
3                 0, // Start index im Array  
4                 3); // Anzahl Attribute (drei Vertices)
```

---

- Es werden drei (3) Attribute indiziert. Startindex 0, Also 0, 1, 2



# OpenGL Shader (GLSL)

- ▶ GLSL (OpenGL Shader Language). An C orientierte Programmiersprache, um auf GPUs eigene Programme (Shader) ablaufen zu lassen.
- ▶ **Vertexshader:** Greift auf Geometriedaten via den Vertex-Array-Pointer auf die Vertex-Buffer zu. Es können z.B. Transformationen von Welt- auf kanonische Koordinaten wie Matrix-Multiplikation durchgeführt werden.
- ▶ Tessellationshader: Flächen können in kleinere Flächen unterteilt werden.
- ▶ Geometryshader: Aus gegebenen Primitiven z.B. Dreick, können neue erzeugt werden.
- ▶ **Fragmentshader:** Für ein Fragment die Farbe bestimmen.

# Vertex-Shader

- ▶ Erste Zeile verlangt GLSL 3.3 Kernfunktionalitäten.
- ▶ Unsere Vertices sind schon in kanonischer Form.
- ▶ Es ist fast nichts zu tun, außer ein homogenen Vektor zu erzeugen.
- ▶ Mit `layout (location = 0)` wird auf Daten des Attribut-Pointers 0 zugegriffen.
- ▶ Das Attribut (ein 3d-Vertice) befindet sich in `aPos`.
- ▶ `gl_Position`: vordefinierter Ausgabewert des Vertice.
- ▶ `vec4()`: Erzeugt einen 4-dim-Vektor. Mehrfach überladen.

---

```
1 char * vertexShaderSource = "#version 330 core\n2 layout (location = 0) in vec3 aPos;\n3 void main()\n4 {\n5     gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n6 }";
```

---

# Fragment-Shader

- ▶ Erste Zeile verlangt GLSL 3.3 Kernfunktionalitäten.
- ▶ Berechnete Farbe soll in `FragColor` gespeichert werden (RGBA).
- ▶ Konstante Farbe direkt im Shader zuweisen.

---

```
1 char * fragmentShaderSource = "#version 330 core\n"
2 out vec4 FragColor;\n
3 void main()\n{
4     FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n
5 };"
```

---

# Übersetzen der Shader

- ▶ Beide Shader müssen in plattformabhängige Befehle der GPU übersetzt werden.
- ▶ Shader erzeugen (auf GPU).
- ▶ Source zu Shader hinzufügen.
- ▶ Shader übersetzen.

---

```
1 GLunit vertexShader = glCreateShader(GL_VERTEX_SHADER);
2 glShaderSource(vertexShader ,
3     1,                      // nur ein String (char *)
4     &vertexShaderSource, // Pointer auf char * (array of strings)
5     NULL); // Pointer auf Array mit length jedes Strings
6     // NULL = alle strings sind NULL-terminated
7 glCompileShader(vertexShader);
8
9 GLunit fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
10 glShaderSource(fragmentShader , 1, &fragmentShaderSource , NULL );
11 glCompileShader(fragmentShader);
```

---

# Übersetzen der Shader

- ▶ Beide Shader gehören zusammen und müssen zu einem ausführbaren Programm gebunden werden (linking)

---

```
1 GLunit shaderProgram = glCreateProgram();  
2 glAttachShader(shaderProgram, vertexShader);  
3 glAttachShader(shaderProgram, fragmentShader);  
4 glLinkProgram(shaderProgram);
```

---

# Compilerfehler prüfen

- ▶ Nach jedem Übersetzen mögliche Compilerfehler ausgeben.
- ▶ Analog für fragmentShader.

---

```
1 int success;
2 char infoLog[512];
3 glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
4 if (!success)
5 {
6     glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
7     std::cout << infoLog << std::endl;
8 }
```

---

# Linkerfehler prüfen

- ▶ Link-Status abfragen.
- ▶ Anschließend die Shader-Resourcen freigeben (nur das ausführbare Programm wird benötigt).

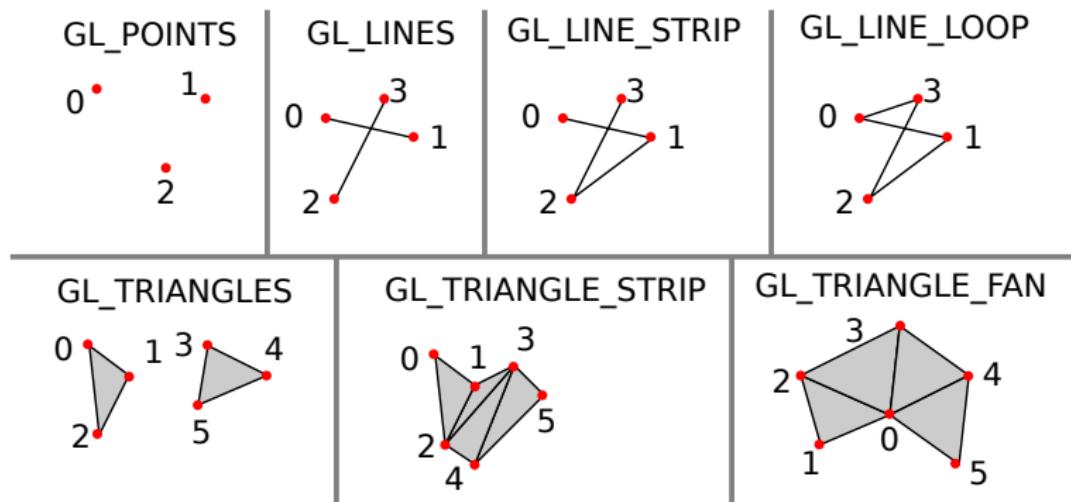
---

```
1 glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
2 if (!success) {
3     glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
4     std::cout << infoLog << std::endl;
5 }
6
7 glDeleteShader(vertexShader);
8 glDeleteShader(fragmentShader);
```

---

# Draw-Mode

- ▶ Erste Parameter `glDrawArrays` bestimmt, was für Primitive gezeichnet werden und welche Attribute dazu verwendet werden.
- ▶  $i = 0, 1, 2, \dots$  sei der Index der aufgezählten Attribute.
- ▶ VAO beschreibt das Speicherlayout der Daten – nicht was sie bedeuten.



# Datentypen GLSL

Datentyp	Werte
bool	true, false
uint	32-Bit unsigned
int	32-Bit 2er-Komplement
float, double	IEEE-754 single, double precision
vec2, ..., vec4	Vektor aus 2,..,4 floats
mat2, ..., mat4	2x2-, ..., 4x4-Matrix mit floats
bvecn, bmatn	Boolescher Vektor bzw. Matrix n-te Dimension
ivecn, imatn	integer
uvecn, umatn	unsigned integer
dvecn, dmatn	double

- ▶ double erst ab OpenGL 4.0

# Datentypen GLSL

---

```
1 const float a = 1.0;
2 vec3 v3 = vec3(0.0, -1.0, 1.5);
3 v3.y = a + 2.5; // x,y,w
4 v3.yw = vec2(3.0, 1.0); // swizzling
5 v3.xyzw = v3.wyxz; // nicht ausprobiert
6 v3.xx = v3.yy; // nicht erlaubt
7 vec4 v4 = vec4(v3, 0.0);
8 mat3 m3;
9 m3[0] = v3; // erste _Spalte_ auf v3 setzen
10 mat2 m2 = m2(1.0, 0.0, -1.5, 3.0);
11 mat4 m4 = m4(v3, 0.0, v3, 1.0, v3, 0.0);
12 mat4 m4 = m4(1.0); // Diagonalmatrix
```

---

# Datentypen GLSL

struct / array

---

```
1 struct Triangle2 {  
2     vec2 vertices[3];  
3 };  
4  
5 Triangle2 t = Triangle( vec2(-0.5, -0.5),  
6                         vec2(0.0, 0.5),  
7                         vec2(0.5, 0.5) );  
8  
9 Triangle2 triangle[2] = {t, t};
```

---

- ▶ Es gibt kein union.

# Anweisungen / Funktionen

- ▶ Funktionen können ähnlich C deklariert werden. `main()` muss vorhanden sein.
- ▶ `in`: Eingabeveriable.
- ▶ `out`: Ausgabeveriable.
- ▶ `inout`: Ein-/Ausgabeveriable.
- ▶ Es gibt für jede Shaderart vordefinierte Ein- und Ausgabevervariablen.

---

```
1  in vec3 normal; // Eingabe aus vorangehenden Shader
2  in vec3 color;
3  out vec4 outcolor;
4
5  void shade(in vec3 normal, in vec3 light, in vec3 color,
6              out vec3 outColor) {
7      outColor = vec4(color * (0.3
8                      + 0.7 * max(0.0, dot(normal, normalize(light))))) , 1.0);
9  }
10
11 void main() {
12     shade(normal, vec(0.0, 1.0, 0.5), color, color);
13     outcolor = vec4(color, 1.0);
14 }
```

---

# Datentypen GLSL

## Kontrollanweisungen

*GLSL uses the standard C/C++ set of statements. It has selection statements (if-else and switch-case), iteration statements (for, while, and do-while), and Jump statements (break, continue, and return). These statements work essentially as C++ defines them (you can declare variables in a for statement, for example), though there are some limitations. For example, you can declare variables in if conditions in C++, but not in GLSL.*

*Notice that goto is absent from the list of jump statements. GLSL has no goto construct.*

[https://www.khronos.org/opengl/wiki/Core\\_Language\\_\(GLSL\)  
#Control\\_flow](https://www.khronos.org/opengl/wiki/Core_Language_(GLSL)#Control_flow)

- ▶ Keine Rekursion möglich.

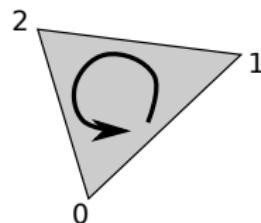
## Zwei Dreiecke zeichnen

- ▶ Dreiecke sollen übereinander liegen. Sichtbarkeit muss bestimmt werden.
- ▶ Tiefenpuffertest muss angeschaltet werden.
- ▶ Tiefenpufferinhalt muss gelöscht werden.
- ▶ Nur Dreiecke deren Vorderfläche sichtbar ist, sollen gezeichnet werden (**Backface-Culling**).

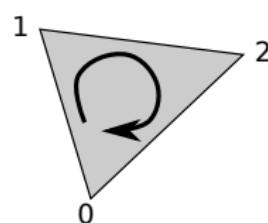
```
glEnable(GL_DEPTH_TEST); // Tiefentest anschalten  
glDepthFunc(GL_GREATER); // Standard ist GL_LESSER  
  
glEnable(GL_CULL_FACE); // Backface culling an  
glFrontFace(GL_CCW); // definiert Vorderfläche  
glCullFace(GL_BACK);  
glClearDepth(-1.0); // Standard 1.0  
  
glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );  
  
// Zeichenbefehle  
  
glDisable( .. ); // Ausschalten möglich
```

# Backface-Culling

- ▶ Blick ist auf die Fläche des Dreicks gerichtet.
- ▶ Ist GL\_CCW eingeschaltet, dann sind nur Flächen sichtbar, deren Punkte gegen dem Uhrzeigersinn in Blickrichtung orientiert sind.
- ▶ Dadurch ist eine sichtbare Vorderfläche definiert.
- ▶ Bei GL\_CW ist es umgekehrt.



counter clockwise

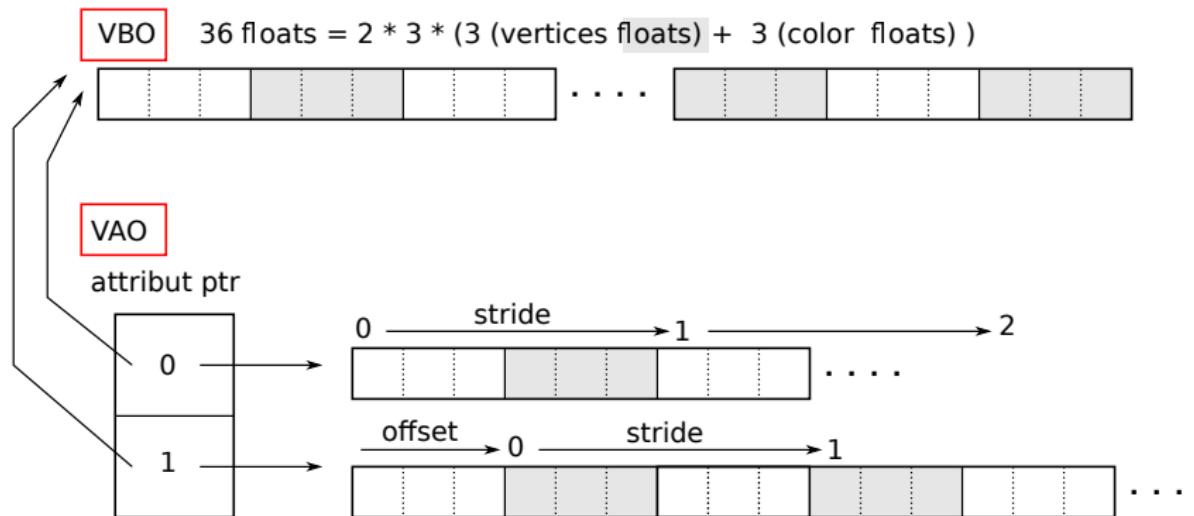


clockwise

- ▶ Es ist normalerweise nicht sinnvoll in einer Anwendung beide Varianten zu mischen.
- ▶ Beim Export eines Modellierungswerkzeugs, als auch beim Import in die Anwendung muss auf die Orientierung geachtet werden.
- ▶ Beim Import ist es sinnvoll, die Orientierung der Dreickspunkte ändern zu können.

# Zwei Dreiecke zeichnen

- Farbe soll pro Vertice definiert werden.



# Zwei Dreiecke zeichnen

---

```
1 float vertices [] = {  
2     -0.5f, -0.5f, 0.0f, // left  1  
3     1.0,    0.0f, 0.0f, // red  
4     0.5f, -0.5f, 0.0f, // right 1  
5     1.0f,   0.0f, 0.0f, // red  
6     0.0f,   0.5f, 0.0f, // top   1  
7     1.0f,   0.0f, 0.0f, // red  
8     -0.3f, -0.3f, -0.5f, // left  2  
9     0.0f,   1.0f, 0.0f, // green  
10    0.7f,   -0.3f, -0.5f, // right 2  
11    0.0f,   1.0f, 0.0f, // green  
12    0.0f,   0.7f, -0.5f, // top   2  
13    0.0f,   0.0f, 1.0f // blue  
14};
```

---

# Zwei Dreiecke zeichnen

---

```
1  GLuint vbo, vao;
2
3  glGenVertexArrays(1, &vao);
4  glGenBuffers(1, &vbo);
5
6  glBindBuffer(GL_ARRAY_BUFFER, vbo);
7  glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
8               GL_STATIC_DRAW);
9
10 glBindVertexArray(vao);
11
12 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
13                       6 * sizeof(float), (void *)0);
14 glEnableVertexAttribArray(0);
15
16 glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
17                       6 * sizeof(float), (void *) (3 * sizeof(float)) );
18 glEnableVertexAttribArray(1);
19
20 glClearColor(0.0, 0.0, 0.0, 1.0);
21 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
22
23 glUseProgram(shaderProgram);
24
25 glDrawArrays(GL_TRIANGLES, 0, 6); // sechs vertices zeichnen
```

---

# Inhalt

Einführung

Mathematische und geometrische  
Grundlagen

Bildrepräsentation

Raytracing

Transformationsmatrizen

Projektionen

Graphik-Pipeline

OpenGL

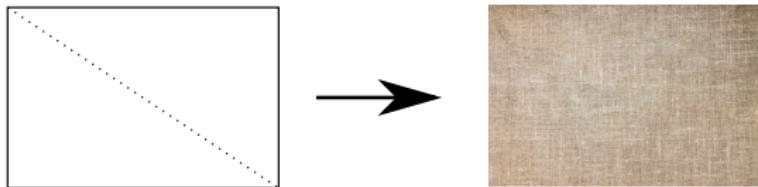
Texturabbildungen

Schattierung von Oberflächen

Prüfung

# Texturabbildungen

- ▶ Bisherige Oberflächen sind strukturlos, glatt und einfarbig.
- ▶ Raufasertapeten, Holzoberflächen, Haut, Leder, Kunststoffe, Textilien habe Strukturen oder Gebrauchsspuren.
- ▶ Diese Strukturen zu modellieren führt zu Objekten, die aus extrem vielen und sehr kleinen geometrischen Primitiven bestehen: Zu speicher- und rechenintensiv.
- ▶ Flächen statt mit einer Farbe mit den Farbpunkten einer Pixelgrafik einfärben: **Textur**.



<https://pixabay.com/>

# Texturabbildungen

- ▶ Texturen sind praktisch immer rechteckig.
- ▶ **Texturraum:** lokale Koordinatensystem der Textur.
- ▶ Texturkoordinaten sind meist auf  $[0, 1]^2$  normalisiert.
- ▶ Sie werden in der Regel mit  $u$  und  $v$  bezeichnet.
- ▶ Berechnung  $(u, v)$  zu Textur-Pixel  $(i, j) \in \mathbb{N}^2$  (texture element, **texel**).
- ▶ Ähnlich zu Viewport-Transformation. Auf links-/rechtshändische Koordinatensystem achten.

---

```
1 Color get_texture_color(Texture t, float u, float v) {  
2     unsigned int i = round( u * t.width - 0.5 );  
3     unsigned int j = round( v * t.height - 0.5 );  
4     return t.get_pixel_color(i, j);  
5 }
```

---

# Texturabbildungen

---

```
1 Color get_surface_color(Surface s, Point p, Texture t) {  
2     Vector normal s.get_normal();  
3     u, v = s.get_texture_coordinate(p);  
4     Color diffuse = get_texture_color(t, u, v);  
5     // apply shading algorithm  
6     // return resulting color  
7 }
```

---

- ▶ Statt diffuser Farbe, Punkt auf einer Oberfläche S mit Texturfarbe einer Textur mit Texturkoordinaten T schattieren.
- ▶ `get_texture_coordinate()` heißt **Textur-Koordinaten-Funktion** (texture coordinate function).
- ▶ Im folgenden  $\Phi$  genannt.
- ▶ Ähnlich Viewtransformation (Parallel-, Perspektivische oder ähnliche Transformation).
- ▶  $\Phi$  vielfältiger und pro Objekt wählbar.

$$\begin{array}{ccc} \Phi : & S & \rightarrow T \\ & (x, y, z) & \mapsto (u, v) \end{array}$$

# Texturabbildungen

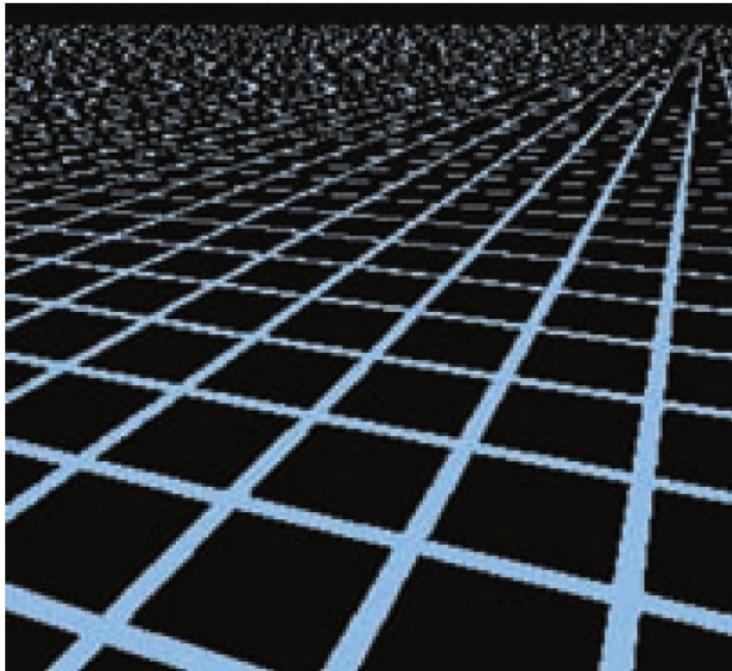
- ▶ Bei Beispiel mit Stofftextur auf Rechteck: Annahme kanonische Form liegt vor (entlang der Z-Achse blicken, oder z ist konstant)

$$\begin{array}{ccc} \Phi : & S & \rightarrow T \\ & (x, y, z) & \mapsto (a \cdot x, b \cdot y) \end{array}$$

- ▶ Für geeignete  $a, b$  (Breite/Höhe Rechtecks berücksichtigen)
- ▶ `get_texture_color()` nimmt Farbe des Texels, welches am nächsten zu  $(u, v)$  liegt.
- ▶ Abtastprobleme treten auf (alias)

# Texturabbildungen

## Abstraktprobleme



# Texturabbildungen

## Gewünschte Eigenschaften

- ▶ Bijektiv: Jeder Punkt der Textur soll höchstens (oder genau) einmal vorkommen.
- ▶ Geringe Verzerrungen der Größe: Links original, rechts vertikal länger



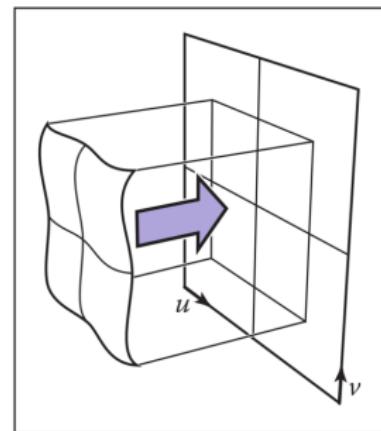
- ▶ Geringe Verzerrungen der Form: Scherung in x-Richtung.
- ▶ Gleichmäßige Übergänge an Kanten: Zwei Rechtecke nebeneinander.



# Texturabbildungen

## Planare Abbildung

- ▶ Parallelprojektion einer Textur auf eine Oberfläche.
- ▶  $M_t$  sei Projektion von Kamera- bis kanonischen Raum.
- ▶ Keine perspektivische Transformation.
- ▶ Blickrichtung  $M_t \cdot (x : y : z : 1)^T$  ist entlang z-Achse.
- ▶  $(u : v : * : 1)^T := M_t \cdot (x : y : z : 1)^T$
- ▶ Wenig Verzerrung bei flachen Oberflächen.



# Texturabbildungen

## Planare Abbildung

- ▶ Textur mit „Koordinaten“ zu Testen verwenden.
- ▶ Planare Abbildung auf Würfel mit abgerundeten Ecken und Rändern.
- ▶ Verzerrungen bei den stark gewölbten Rändern.
- ▶ Abbildungen ist bijektiv.

09	19	29	39	49	59	69	79	89	99
08	18	28	38	48	58	68	78	88	98
07	17	27	37	47	57	67	77	87	97
06	16	26	36	46	56	66	76	86	96
05	15	25	35	45	55	65	75	85	95
04	14	24	34	44	54	64	74	84	94
03	13	23	33	43	53	63	73	83	93
02	12	22	32	42	52	62	72	82	92
01	11	21	31	41	51	61	71	81	91
00	10	20	30	40	50	60	70	80	90

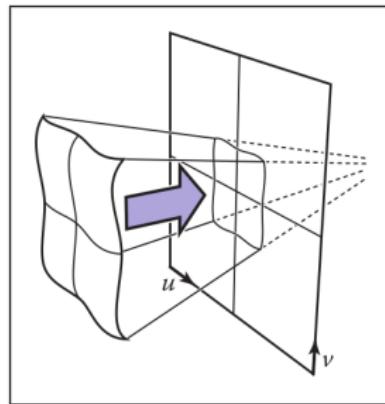


## Texturabbildungen

## Perspektivische Abbildung

- ▶ Perspektivische Transformation  $P_t$  (inklusive Kamera etc.).
  - ▶ Koordinate  $w$  darf nicht ignoriert werden.
  - ▶  $(u' : v' : * : w)^T := P_t \cdot (x : y : z : w)^T$

$$\Phi : u, v \mapsto \frac{u'}{w}, \frac{v'}{w}$$



# Texturabbildungen

## Zylinder

- ▶ Kanonischer Raum.
- ▶ Zylinder mit Radius 1 und Höhe 2 im Nullpunkt zentriert.

$$u = \frac{1}{2\pi}(\pi + \text{atan2}(y, x))$$

$$v = \frac{1}{2}(z + 1)$$

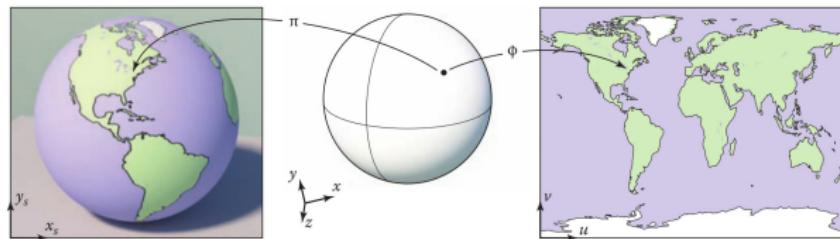
# Texturabbildungen

## Kugel

- ▶ Kanonischer Raum.
- ▶ Kugel mit Radius 1 im Nullpunkt zentriert.

$$u = \frac{1}{2\pi}(\pi + \text{atan2}(y, x))$$

$$v = \frac{1}{\pi}(\pi - \text{acos}(z/|x|))$$

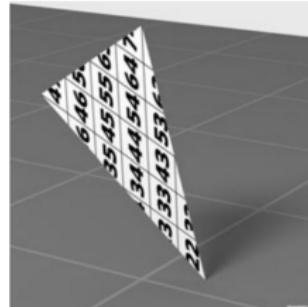


# Texturabbildungen

## Dreieck

- ▶ Texturkoordinaten pro Dreieckseckpunkt speichern (Zuordnung im Modellierungstool).
- ▶ Interpolation Texturkoordinaten für gerasterte Punkte mit Baryzentrischen Koordinaten.

09	19	29	39	49	59	69	79	89	99
08	18	28	38	48	58	68	78	88	98
07	17	27	37	47	57	67	77	87	97
06	16	26	36	46	56	66	76	86	96
05	15	25	35	45	55	65	75	85	95
04	14	24	34	44	54	64	74	84	94
03	13	23	33	43	53	63	73	83	93
02	12	22	32	42	52	62	72	82	92
01	11	21	31	41	51	61	71	81	91
00	10	20	30	40	50	60	70	80	90



- ▶  $(0, 2; 0, 2)$  für linken unteren Punkt etc.

# Texturabbildungen

## Dreiecksnetze / Polyeder

- ▶ „Auffalten“ von Polyedern auf Textur ergibt kontinuierliche Übergänge.
- ▶ Nicht notwendigerweise injektiv.
- ▶ Verzerrungen wie bei planarer Abbildung möglich, wenn Texturkoordinaten nicht gleiche Form haben wie Dreiecke der Oberfläche.
- ▶ Modellierungswerkzeuge unterstützen Auffalten auf Rechtecke. Diese können dann texturiert werden.

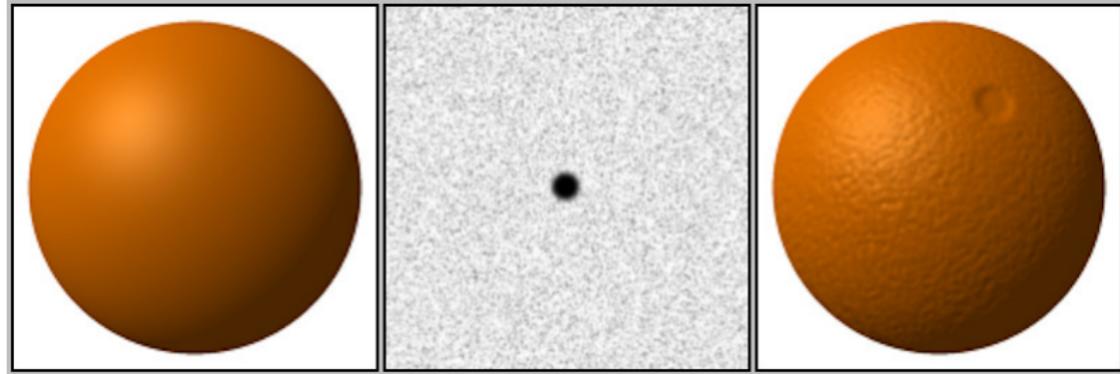
09	19	29	39	49	59	69	79	89	99
08	18	28	38	48	58	68	78	88	98
07	17	27	37	47	57	67	77	87	97
06	16	26	36	46	56	66	76	86	96
05	15	25	35	45	55	65	75	85	95
04	14	24	34	44	54	64	74	84	94
03	13	23	33	43	53	63	73	83	93
02	12	22	32	42	52	62	72	82	92
01	11	21	31	41	51	61	71	81	91
00	10	20	30	40	50	60	70	80	90



# Texturabbildungen

## Bumpmapping

- ▶ Statt Farbenpixel, Normalenvektoren oder Höhen (height map) speichern.
- ▶ Abweichungen von Oberflächen simulieren.
- ▶ Beim Shading zu Oberflächennormalen addieren ergibt eine Helligkeit in der Oberfläche, die den Normalen der Bumpmap entspricht.
- ▶ Ränder weiterhin glatt.
- ▶ Displacement-Mapping: Punkte der Oberfläche werden verschoben (tesselation shader).



# Texturabbildungen

## OpenGL

- ▶ OpenGL besitzt keine Funktionen, um Texturen in den Hauptspeicher zu laden.
- ▶ Hier: Simple OpenGL Image Library (SOIL) verwendet.
- ▶ Unterstützt unter anderem: BMP, PNG, JPG, TGA Formate.

---

```
1 #include <SOIL/SOIL.h>
2
3 GLuint texWidth, texHeight, channels;
4 unsigned char *texImage = SOIL_load_image ("jute.jpg",
5         &texWidth,
6         &texHeight,
7         &channels,
8         SOIL_LOAD_RGB);
9 if (texImage == NULL) {
10     std::cout << "Can not load image." << std::endl;
11     return EXIT_FAILURE;
12 }
```

---

# Texturabbildungen

## OpenGL

- Wie bei VBOs oder VAOs muss ein Texturbezeichner erzeugt und gebunden werden.
- Mit `glTexImage2D` wird OpenGL-Textur definiert
- `glActiveTexture(GL_TEXTURE0)` erlaubt Zugriff in Fragment-Shader.

```
1 glActiveTexture(GL_TEXTURE0); // select texture unit 0
2
3 glGenTextures(1, &texture);
4 glBindTexture(GL_TEXTURE_2D, texture);
5
6 glTexImage2D( GL_TEXTURE_2D,      // a 2D texture (1D/3D exists as well)
7               0,                  // level of detail
8               // (0 = use whole image, no midmaps)
9               GL_RGB,             // pixel format used by GPU
10              texWidth,           // number of pixel in width
11              texHeight,          // number of pixel in height
12              0,                  // must be 0 (border)
13              GL_RGB,             // pixel format in image
14              GL_UNSIGNED_BYTE, // data type of pixel
15              texImage);         // pointer to image
16
17 SOIL_free_image_data(texImage);
```

# Texturabbildungen

## OpenGL

- ▶ Mit `glTexParameterI()` können verschiedenste Texturparameter definiert werden.
- ▶ `GL_TEXTURE_WARP_S`, `GL_TEXTURE_WARP_T`: Beschreibt, wie mit Texturkoordinaten ausserhalb  $[0, 1]^2$  umgegangen werden soll.
- ▶ `GL_REPEAT`: Textur wird kontinuierlich fortgeführt.
- ▶ `GL_MIRRORED_REPEAT`: Fortführung, aber gespiegelt.
- ▶ `GL_CLAMP_TO_EDGE`: Randfarbe der Textur wird fortgeführt.
- ▶ `GL_CLAMP_TO_BORDER`: Spezielle Farbe wird verwendet.
- ▶ `GL_MIRROR_CLAMP_TO_EDGE`: Eine Spiegelung des Image, dann Randfarbe.

---

```
1 glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
2 glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
3
4 glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
5 glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

---

# Texturabbildungen

## OpenGL

- ▶ Filterung der Farbe, da Auflösung des Image nicht der des gerasterten Pixel entspricht.
- ▶ GL\_TEXTURE\_MIN\_FILTER: Pixel wird zu einem Texel gemappt.
- ▶ GL\_TEXTURE\_MAG\_FILTER: Pixel wird zu mehreren Texel gemappt.
- ▶ GL\_NEAREST: Texel, das Punkt am nächsten ist.
- ▶ GL\_LINEAR: Texel wird linear aus Texel in Umgebung des Punktes interpoliert.

---

```
1 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);  
2 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

---

# Texturabbildungen

## OpenGL

- ▶ Filterung der Farbe, da Auflösung des Image nicht der des gerasterten Pixel entspricht.
- ▶ GL\_TEXTURE\_MIN\_FILTER: Pixel wird zu einem Texel gemappt.
- ▶ GL\_TEXTURE\_MAG\_FILTER: Pixel wird zu mehreren Texel gemappt.
- ▶ GL\_NEAREST: Texel, das Punkt am nächsten ist.
- ▶ GL\_LINEAR: Texel wird linear aus Texel in Umgebung des Punktes interpoliert.

---

```
1 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);  
2 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

---

# Texturabbildungen

## OpenGL Vertex-Shader

- ▶ 2D-Texturkoordinate muss geholt und weitergegeben werden.

```
1 #version 330 core
2 layout (location = 0) in vec3 position;
3 layout (location = 1) in vec2 texture_coord;
4 out vec2 texture_coord_out;
5 void main() {
6     texture_coord_out = texture_coord;
7     gl_Position = vec4(position, 1.0);
8 }
```

# Texturabbildungen

## OpenGL Fragment-Shader

- ▶ 2D-Texturkoordinate muss geholt und weitergegeben werden.
- ▶ Uniform braucht in unserem Beispiel nicht gesetzt werden, da einer der mindestens 16 Texture-Einheiten (`glActiveTexture`) genutzt wird.
- ▶ Berechnung der Farbe mit `texture()`-Funktion.

---

```
1 #version 330 core
2 in vec2 texture_coord_out;
3 out vec4 out_color;
4 uniform sampler2D texture_data;
5 void main() {
6     out_color = texture(texture_data, texture_coord_out);
7 }
```

---

## Texturabbildungen

## OpenGL



# Inhalt

Einführung

Mathematische und geometrische  
Grundlagen

Bildrepräsentation

Raytracing

Transformationsmatrizen

Projektionen

Graphik-Pipeline

OpenGL

Texturabbildungen

Schattierung von Oberflächen

Prüfung

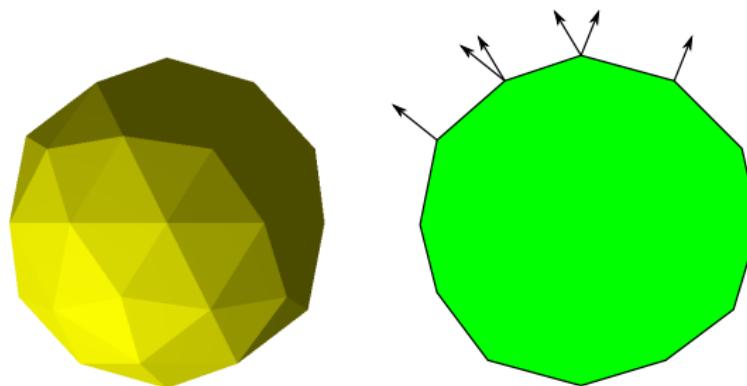
# Schattierung

- ▶ Bisher: Matte Oberflächen. Lambertian-Shading und Gouraud-Shading.
- ▶ Raytracing: Globale Beleuchtung mit Reflexion, Refraktion, Schatten.
- ▶ Objekt-basiertes Rendern: Spiegelungen und Beleuchtung einzelner Objekte ohne globale Beleuchtung. Shader-Algorithmen austauschbar.

# Schattierung

## Flat-Shading

- ▶ Matte Oberfläche. Lambertian-Beleuchtung.
- ▶ Eine senkrechte Normale für jede Fläche.
- ▶ Resultierende Farbe aller Punkte der Fläche gleich.
- ▶ Schnell, unrealistisch.
- ▶ Kanten werden hervorgehoben.



# Schattierung

## Lambertian-Shading

- ▶ Lichtquelle aus einer Richtung  $\vec{I}$ .
- ▶ Intensität (Farbe)  $I$  des Lichts.
- ▶ Für diffuse Beleuchtung  $k_d$  einer matten Oberfläche.

$$L = k_d \cdot I \max\{0, \cos \Theta\} = k_d \cdot I \max\{0, \vec{n} \cdot \vec{I}\}$$

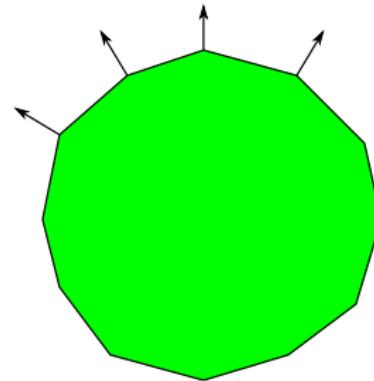
- ▶ Konstante, globale ambiente Beleuchtung  $k_a$  wird hinzugefügt.
- ▶ Beide Teile werden gewichtet.
- ▶ Gewichtung kann Teil der Faktoren  $k_a$  und  $k_d$  sein.

$$L = k_a + k_d \cdot I \max\{0, \vec{n} \cdot \vec{I}\}$$

# Schattierung

## Gouraud-Shading

- ▶ Eine Normale für jeden Punkt.
- ▶ Normale üblicherweise interpoliert aus umgebenen Flächennormalen.
- ▶ Farbe der Fläche wird aus Eckpunkten linear interpoliert.
- ▶ Fläche wirkt glatt.
- ▶ Realistisch bei kleinen Flächen.



# Schattierung

## Gouraud-Shading GLSL

- ▶ Farbe wird im Vertexshader berechnen.
- ▶ Eingehende Werte in Fragmentshader werden standardmäßig interpoliert.
- ▶ Fragment-Shader gibt nur eingehende Farbe weiter.

---

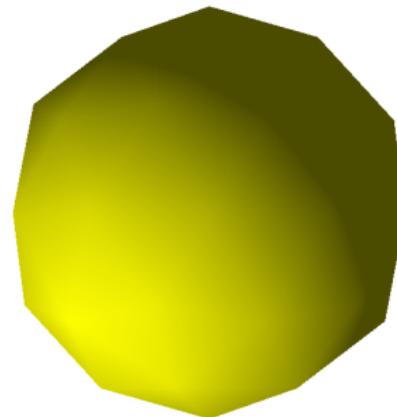
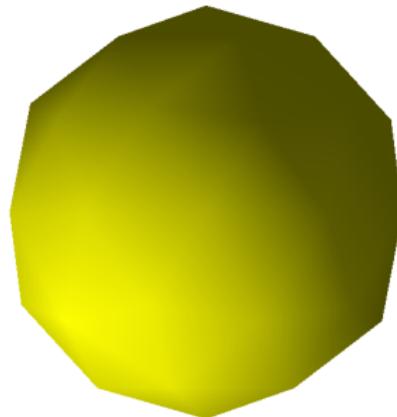
```
1 #version 330 core
2 layout (location = 0) in vec3 position;
3 layout (location = 1) in vec3 incolor;
4 layout (location = 2) in vec3 innormal;
5 out vec3 color;
6 vec4 normal;
7 vec4 light_direction;
8 uniform mat4 model;
9 void main()
10 {
11     gl_Position = model * vec4(position, 1.0);
12     light_direction = vec4(normalize(
13         vec3(-10.0, -10.0, -10.0) - position), 0.0);
14     normal = normalize(model * vec4(innormal, 0.0));
15     color = vec3(incolor * (0.3
16                         + 0.7 * max(0.0, dot(normal, light_direction))));
```

---

# Schattierung

## Phong-Shading

- ▶ Normale eines Punkts der Fläche wird aus Normalen der Eckpunkte interpoliert.
- ▶ Farbe der Fläche wird aus Eckpunkten linear interpoliert und mit interpolierten Normalen schattiert.
- ▶ Fläche wirkt glatter.
- ▶ Langsamer. Realistischer bei großen Flächen.
- ▶ Links Gouraud, rechts Phong.



# Schattierung

## Phong-Shading GLSL

- Vertex-Shader reicht Normale weiter. *Alle* Eingaben werden im Fragment-Shader interpoliert.

---

```
1 // Rest wie zuvor
2 out vec4 normal;
3 out vec3 pos;
4 void main()
{
    gl_Position = model * vec4(position, 1.0);
    pos = gl_position.xyz;
    normal = normalize(model * vec4(innormal, 0.0));
    color = incolor;
}
```

---

---

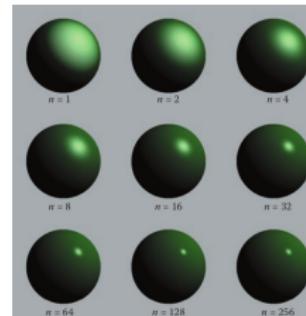
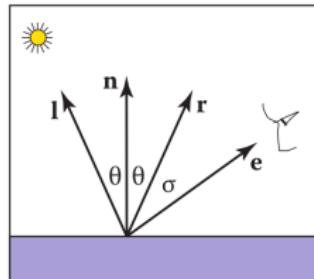
```
1 in vec3 pos;
2 in vec4 normal;
3 in vec3 color;
4 vec4 light_dir;
5 out vec4 outColor;
6 void main () {
    light_dir = vec4(normalize(vec3(-10.0, -10.0, -10.0) - pos), 0.0);
    outColor = vec4(color * (0.3 +
        0.7 * max(0.0, dot(normal, light_dir))), 1.0);
}
```

---

# Beleuchtungsmodell

## Phong-Beleuchtung

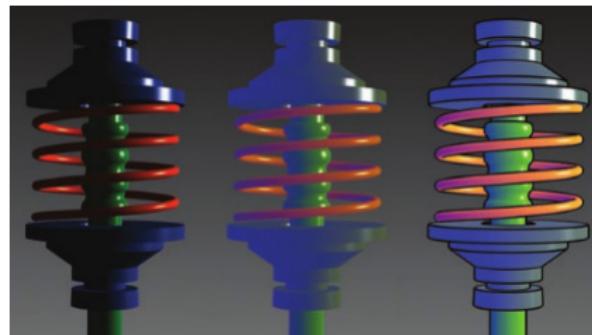
- ▶ Lokales heuristisches Beleuchtungsmodell.
- ▶ Glanzlicht: Je näher der Betrachter dem Reflektionsstrahl kommt, desto heller ist die Fläche.
- ▶ Heller, wenn Winkel  $\sigma$  kleiner wird. Nicht-linear.
- ▶  $c = k_s \cdot \max(0, r \cdot e)^p$
- ▶  $k_s$  ist Anteil spiegelnder Reflexion.
- ▶  $p$  heißt **Phong-Exponent**.



# Artistic Shading

## Gooch-Shading

- ▶ **Artistic-Shading:** Nicht-realistische Malstile menschlicher Künstler nachahmen.
- ▶ Cartoon, Pointilismus, Expressionismus, ...
- ▶ **Gooch-Shading** (auch Kalt-zu-Warm Shading): Kalte und Warme Farbe werden linear interpoliert, abhängig davon, wie stark die Fläche dem Licht zugewandt ist.
- ▶ Links normal, Mitte Gooch mit Blau (kalt) und Orange (warm), Rechts mit Silhouette.



# Artistic Shading

## Gooch-Shading

- ▶  $c_w$  warm,  $c_c$  kalt.
- ▶  $N$  sei Normale,  $L$  normalisierte Richtung zur Lichtquelle.
- ▶ Anteil warmes Licht  $k_w = \frac{1+N \cdot L}{2}$  (ähnlich Lambert) liegt im Bereich  $[0, 1]$ .

$$c = k_w \cdot c_w + (1 - k_w) \cdot c_c$$

# Inhalt

Einführung

Mathematische und geometrische  
Grundlagen

Bildrepräsentation

Raytracing

Transformationsmatrizen

Projektionen

Graphik-Pipeline

OpenGL

Texturabbildungen

Schattierung von Oberflächen

Prüfung

# Prüfung

- ▶ MINB: Computervision/Computergrafik 120 min gemeinsame Modulprüfung.
- ▶ INFB: Computervision-Wahlfach 60 min
- ▶ INFB: Computergrafik 90 min. Mit kurzer Pause im Anschluß Computervision-Wahlfach.
- ▶ 50 Prozent der Punkte zum Bestehen bei Computergrafik Wahlfach.
- ▶ Keine Hilfsmittel. Kein Taschenrechner.
- ▶ Prüfungsinhalt ist die Vorlesung mit den Übungsaufgaben.

# Prüfung Computergrafik

## Aufgaben

- ▶ Ca. 25 Prozent Wissensabfragen. Erläutern, Erklären, Beschreiben Sie ...
- ▶ Ca. 25 Prozent Rechenaufgaben, z.B. u-v-Parameter bestimmen.
- ▶ Algorithmen ausführen (es gab nur wenige).
- ▶ Objekt-Transformationen und View-Transformationen aufstellen.
- ▶ Kleine Modifikationen von gegebenen Code oder API-Befehlen möglich (OpenGL, Raytracing).
- ▶ Mindestens 50 Prozent der Prüfungsaufgaben werden ähnlich zu manchen der Übungsaufgaben sein.

-  BRESENHAM, J. E.: *Algorithm for computer control of a digital plotter.*  
IBM Systems Journal, 4(1):25–30, 1965.
-  PITTEWAY, M. L. V.: *Algorithm for drawing ellipses or hyperbolae with a digital plotter.*  
The Computer Journal, 10(3):282–289, 01 1967.
-  VAN AKEN, JERRY und MARK NOVAK: *Curve-Drawing Algorithms for Raster Displays.*  
ACM Trans. Graph., 4(2):147–169, apr 1985.