

cg_full

Vorwort

Diese Stoffsammlung/Zusammenfassung enthält den Stoff, der in der EVC Vorlesung der TU Wien im Sommersemester 2025 vorgetragen wurde, der auch in den jeweiligen Skripten und Slides zu finden ist. Die Struktur dieser Zusammenfassung basiert demnach auch der des Skriptums.

Disclaimer

Vieles der Zusammenfassung wurde mit AI generiert, basiert allerdings nur auf Inhalten der Unterlagen. Die Stellen die mit AI generiert wurden, wurden von mir überprüft und mit den Unterlagen verglichen, aber auch ich kann Fehler machen.

Demnach, falls sich irgendwo Fehler befinden oder es Verbesserungsvorschläge gibt, bitte an [@xmozz](#) auf Discord wenden.

Inhalt

- 1. Einführung
- 2. Graphikpipeline und Objektrepräsentationen
- 3. Transformationen
- 4. Farbe
- 5. Rasterisierung
- 6. Viewing
- 7. Clipping und Antialiasing
- 8. Sichtbarkeitsverfahren
- 9. Beleuchtung und Schattierung
- 10. Ray-Tracing
- 11. Globale Beleuchtung und Texturen
- 12. Kurven und Flächen
- 13. Computer Animation
- 14. Machine Learning für 3D Graphics

1. Einführung in die Computergraphik

Quellen: [EVC_Skriptum\(CG, p.5\)](#), [EVC_Skriptum\(CG, p.6\)](#)

Definition: Computergraphik ist ein Teilgebiet der Informatik, das sich mit der künstlichen Erzeugung und Manipulation von Bildern beschäftigt, inklusive der digitalen Repräsentation, Erzeugung und Manipulation der zugehörigen Daten.

Anwendungsbereiche: Die Notwendigkeit künstlicher Bilder wächst mit der Computerisierung vieler Lebensbereiche. Typische Anwendungen:

- **Entertainment:**

- **Computerspiele:** Größter Markt, treibende Kraft in der Forschung.
- **Filmindustrie:** Erzeugung unmöglicher Szenen, Nachbearbeitung, Ergänzung von Inhalten; Verschmelzung mit klassischer Filmproduktion und Animation.

- **Computer Aided Design (CAD):**

- Industrielle Produktentwicklung und visuelle Inspektion (Geräte, Behältnisse, Sportartikel, Schmuck, Autos, Flugzeuge, Fenster).
- Architektur: Virtuelle Begehung vor Bau.
- Straßen- und Landschaftsplanung: Visuelle Vorwegnahme und Planung.

- **Werbung:**

- Langjährige Nutzung von Computergraphik.
- Finanzstarke Spotindustrie (Kurzanimationen, Manipulationen).
- Marketing mit interaktiven visuellen Methoden -> Imagevorteil.

- **Simulatoren:**

- Training in teuren oder gefährlichen Technologien (Flugzeugpiloten, Raumfahrer, Autosimulation).
- Simulation von Gefahren- und Katastrophensituationen zur Vorbereitung und Schulung.
- Nutzung von wahrnehmungsbasiertem Rendering (Berücksichtigung der Augenwahrnehmung).

- **Kulturerbe:**

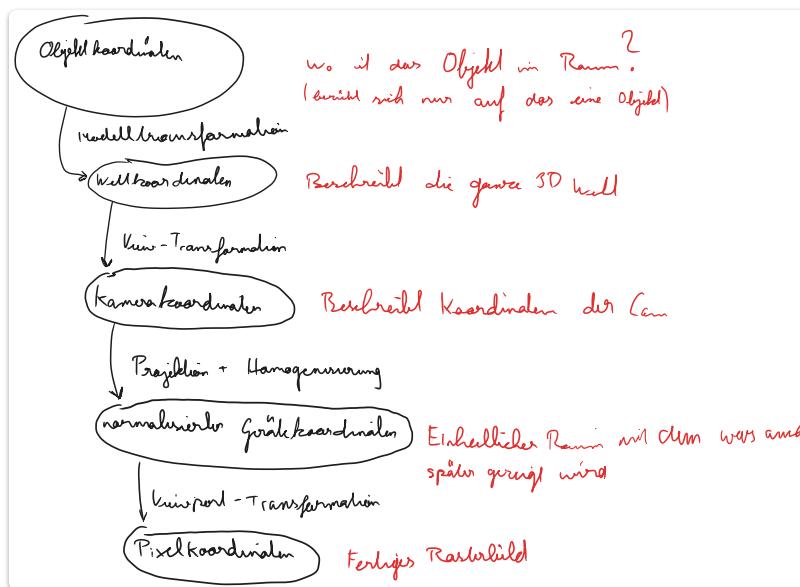
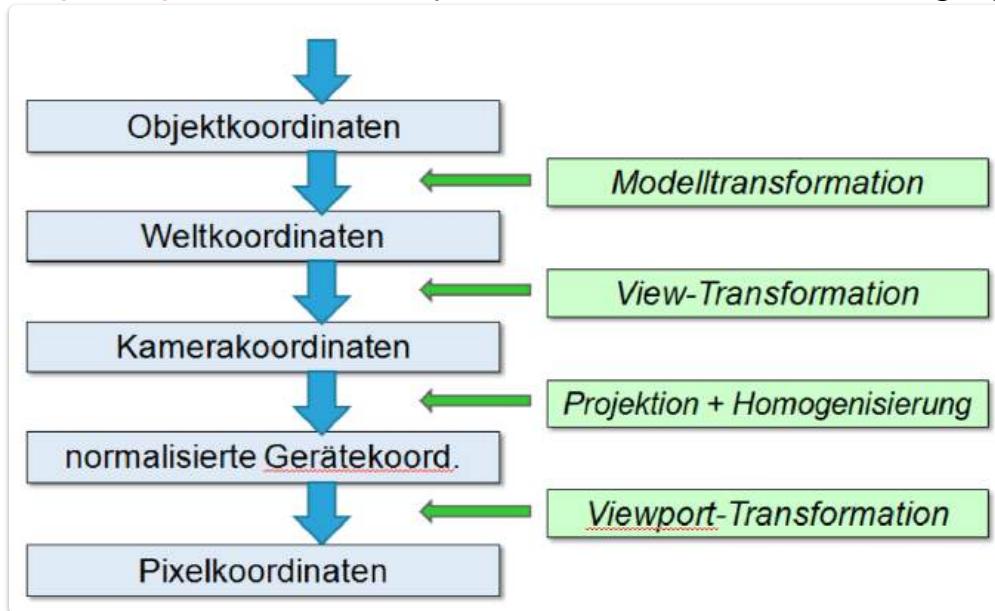
- Virtuelle Bewahrung/Wiederherstellung von geschädigten Dingen und Bauwerken.
- Unterstützung von Museen und Bildung (z.B. Geschichtsunterricht).

- **Wissenschaft:**

- **Visualisierung:** Finden von Strukturen und Informationen in unübersichtlichen/großen Datenmengen (explorativ und routinemäßig, z.B. Medizinische Bildgebung).

Komponenten von Computergraphik-Software: Ein Graphiksystem umfasst viele Schritte von der Datenmodellierung bis zur Bilddarstellung.

- **Graphik-Pipeline:** Kette von Operationen und Daten zur Bilderzeugung.



- [2. Graphikpipeline und Objektrepräsentationen](#)
- **Graphik-Primitive:** Einfache geometrische Grundformen (Linien, Kreise, Rechtecke) zur geräteunabhängigen Darstellung. [2. Graphikpipeline und Objektrepräsentationen](#)
- **Rasterisierung:** Umwandlung von Primitiven in Pixelinformationen. [5. Rasterisierung](#)
- **Objektmodellierung:** Kombination geometrischer Primitiven (inkl. Freiformflächen) und Speicherung in geeigneten geometrischen Datenstrukturen. [2. Graphikpipeline und Objektrepräsentationen](#)
- **Platzierung:** Anordnung der Objekte im Weltkoordinatensystem.
- **Projektion:** Festlegung der Ansicht durch Kameraparameter.
- **Geometrische Transformationen:** Homogene Matrizen für Platzierung und Projektion. [3. Transformationen](#)

- **Clipping:** Entfernen von Bildteilen außerhalb des Betrachtungsfensters. [7. Clipping und Antialiasing](#)
- **Sichtbarkeitsberechnung:** Entfernen verdeckter Bildinformationen. [8. Sichtbarkeitsverfahren](#)
- **Beleuchtungsmodelle:** Einfache Schattierung bis realistisches Ray-Tracing und globale Beleuchtung. [8. Sichtbarkeitsverfahren](#)
- **Texturen:** Zusätzliche Oberflächeninformationen, kombinierbar mit lokalen geometrischen Strukturen.
- **Anti-Aliasing:** Verfahren zur Reduktion des Rastereindrucks bei der Bildausgabe. [7. Clipping und Antialiasing](#)

2. Graphikpipeline und Objektrepräsentationen

Graphikpipeline:

Quelle: [EVC_Skriptum_CG](#), p.7

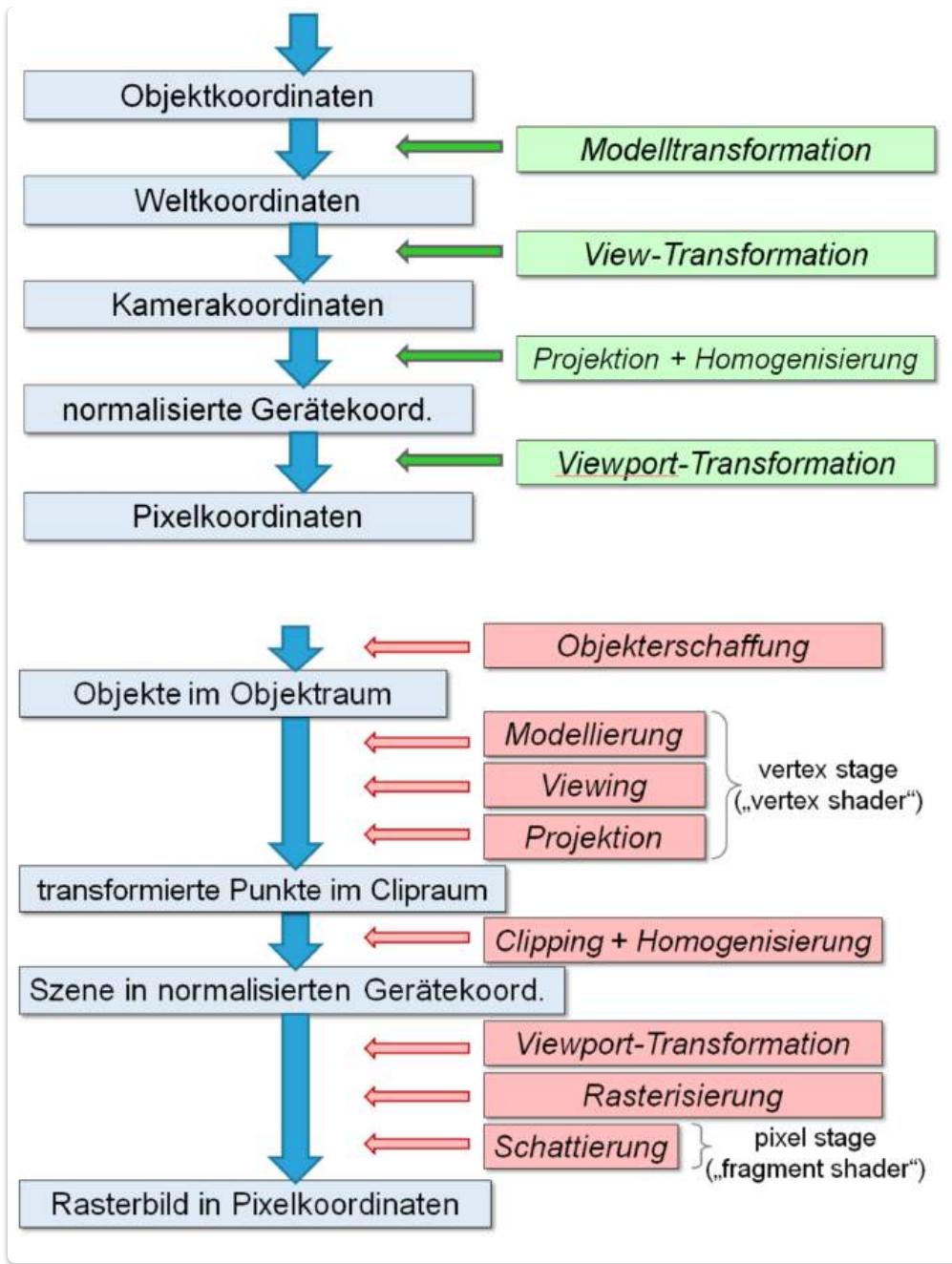
Informationen werden in aufeinanderfolgenden Schritten in ein Bild transformiert. Diese Abfolge wird als **Graphikpipeline** bezeichnet und kann je nach Fokus auch **Viewing-Pipeline**, **Transformationspipeline** oder **Rendering-Pipeline** genannt werden.

Schritte der Graphikpipeline:

1. **Objekt- und Szenenbeschreibung:** Die Objekte und ihre Anordnung in der Szene werden auf irgendeine Weise definiert. Die Objekterschaffung kann durch **Modellierung** oder **Scanning** erfolgen.
2. **Festlegung der Blickrichtung etc.:** Die Kameraparameter und die gewünschte Perspektive werden bestimmt.
3. **Projektion der Objekte:** Die 3D-Objekte werden auf eine 2D-Ebene projiziert.
4. **Umwandlung in Rasterpunkte:** Das projizierte Ergebnis wird in Pixel umgewandelt, um auf einem Bildschirm dargestellt zu werden.

Zusätzliche Informationen:

- **Vertex Shader und Fragment Shader:** Dies sind programmierbare Teile von Graphikkarten, die in der Pipeline eine Rolle spielen.
- **Grundprinzip:** Es existieren viele ähnliche Darstellungen der Graphikpipeline, die alle dasselbe grundlegende Prinzip beschreiben.
- **Einordnung weiterer Kapitel:** Die meisten weiteren Themen in diesem Skriptum zur Graphik können direkt in dieses Schema der Graphikpipeline eingeordnet werden.

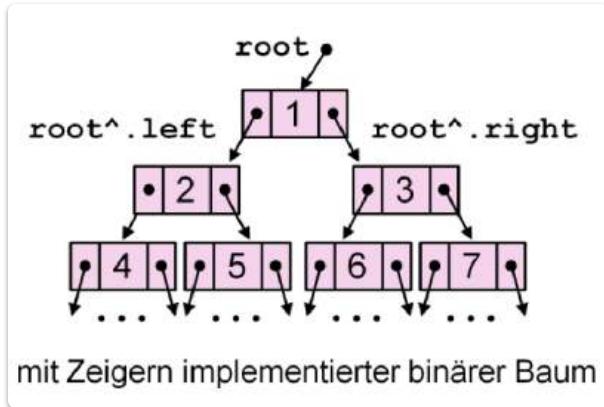
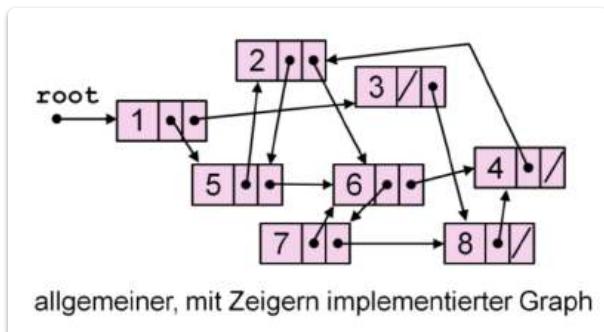


Einschub: Graphen und Bäume

Quelle: [EVC_Skriptum_CG](#), p.7

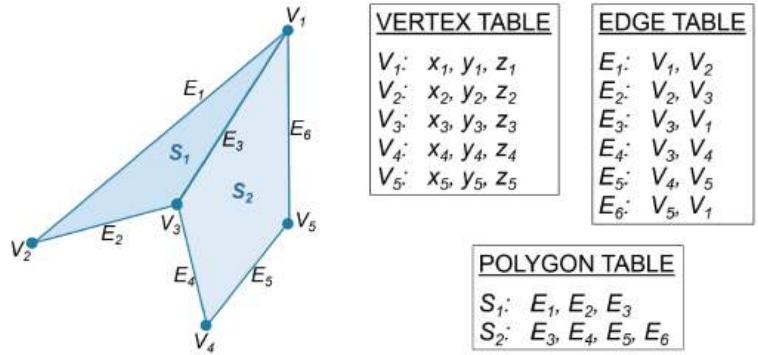
- **Repräsentation:** Ähnlich verketteten Listen können Graphen und speziell Bäume durch **zeigerverkettete Strukturen** dargestellt werden.
- **Knotenstruktur:** Einzelne Knoten benötigen ausreichend **Zeigerkomponenten**, um die gewünschte Struktur abzubilden (z.B. zwei Zeiger für binäre Bäume).
- **Operationen:** Einfügen und Entfernen von Knoten analog zu verketteten Listen.
- **Bearbeitungsreihenfolge:** Steuerung durch Aufrufe geeigneter Nachfolger.
- **Rekursive Abarbeitung binärer Bäume (Beispiele):**
 - **Pre-order:** Wurzel → linker Nachfolger → rechter Nachfolger (Beispielhafte Ausgabe: 1245367)

- **In-order:** linker Nachfolger → Wurzel → rechter Nachfolger (Beispielhafte Ausgabe: 4251637)
- **Post-order:** linker Nachfolger → rechter Nachfolger → Wurzel (→4526731)



Polygon-Listen (B-Reps)

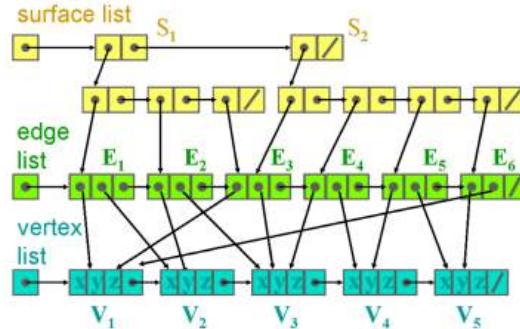
Dreidimensionale Objekte werden meist durch Polygonlisten repräsentiert (oft Dreiecke). Eine Menge von Polygone, die die Oberfläche eines Objektes beschreibt, nennt man *Boundary-Representation* („B-Rep“). Datenstrukturen für B-Reps enthalten neben geometrischer Information auch Attribute (Eigenschaften). Die Geometrie besteht aus Punktlisten, Kantenlisten, Flächenlisten und muss auf Konsistenz und Vollständigkeit überprüft werden.



Dieselbe Struktur lässt sich auch mit Zeigerlisten darstellen (siehe Abbildung rechts).

Die gesamte Koordinateninformation befindet sich in den **Punktknoten** (**V** steht für **Vertex**). Wenn ein Punkt an einen neuen Ort transformiert wird, so reicht es, die Koordinaten dieses Punktes zu verändern. Die Topologie bleibt dabei erhalten. Die lineare Verkettung der Kanten und Punkte erleichtert die Bearbeitung (z.B. alle Kanten einmal zeichnen, oder alle Punkte verschieben).

Das Beispiel links zeigt für eine ganz einfache Situation mit 2 Polygone, wie die Punktliste (**Vertex Table**), Kantenliste (**Edge Table**) und die Flächenliste (**Surface Table**) sich gegenseitig referenzieren. Nur in der **Punktliste** ist die tatsächliche geometrische Information enthalten, die anderen Listen beschreiben lediglich die Topologie.



Wichtig!

- Das heißt der einzige Ort, wo Tatsächlich Koordinaten gespeichert sind, ist die Vertex Tabelle
- Die anderen Tabellen sind jeweils nur Referenzen auf sich gegenseitig und die VT
- Somit ist das anpassen verschiedener Attribute wie Position einfach durchzusetzen, da man nur in der jeweiligen Tabelle was ändern muss und sich der Rest anpasst.
- Außerdem hatten wir hier dann noch einen kleinen Wiederholung von [Vektoren und ihre Produkte \(Aus EVC\)](#) mehr dazu auch noch hier [7. Vektoren](#) und [8. Matrizen](#)

Wichtige Begriffe

Quelle: [EVC_Skriptum\(CG\)](#), p.8

- **Polygonfläche:**
 - Repräsentation beinhaltet die **Trägerebene** (definiert durch die Gleichung $Ax + By + Cz + D = 0$) und die zugehörigen **Eckpunkte** (V_1 bis V_n).
 - Aus den Ebenenparametern (A, B, C, D) lässt sich direkt der **Normalvektor der Ebene** (A, B, C) ableiten.
- **Backface:** Die Rückseite eines Polygons, die ins Innere des Objekts zeigt.
- **Frontface:** Die Vorderseite eines Polygons, die die Außenseite des Objekts bildet.

Wenn man ein rechtshändiges Koordinatensystem vorausschickt und die Eckpunkte jedes Polygons (von vorne betrachtet) im mathematisch positiven Sinn (also gegen den Uhrzeigersinn) anordnet, dann gilt für einen Punkt (x, y, z) :

- wenn $Ax + By + Cz + D = 0$ dann liegt der Punkt **auf** der Ebene
- wenn $Ax + By + Cz + D < 0$ dann liegt der Punkt **hinter** der Ebene
- wenn $Ax + By + Cz + D > 0$ dann liegt der Punkt **vor** der Ebene

- **Berechnung des nach außen gerichteten Normalvektors:** Aus drei aufeinanderfolgenden Eckpunkten (V_1, V_2, V_3) eines Polygons (bei rechtshändigem Koordinatensystem und gegen den Uhrzeigersinn angeordnet) kann der nach außen gerichtete Normalvektor \mathbf{N} berechnet werden durch die Formel: $N = (V_2 - V_1) \times (V_3 - V_1)$
- **Dreiecke als Polygone:** Sehr häufig verwendet, da sie Algorithmen vereinfachen (z.B. sind Dreiecke immer eben).
- **Dreiecks-Mesh:** Eine Datenstruktur, die ausschließlich aus Dreiecken besteht.
- **Dreiecks-Strip:** Eine lineare Abfolge von Dreiecken, bei der jedes weitere Dreieck durch Angabe nur eines neuen Punktes definiert wird (die vorherige Kante wird beibehalten).

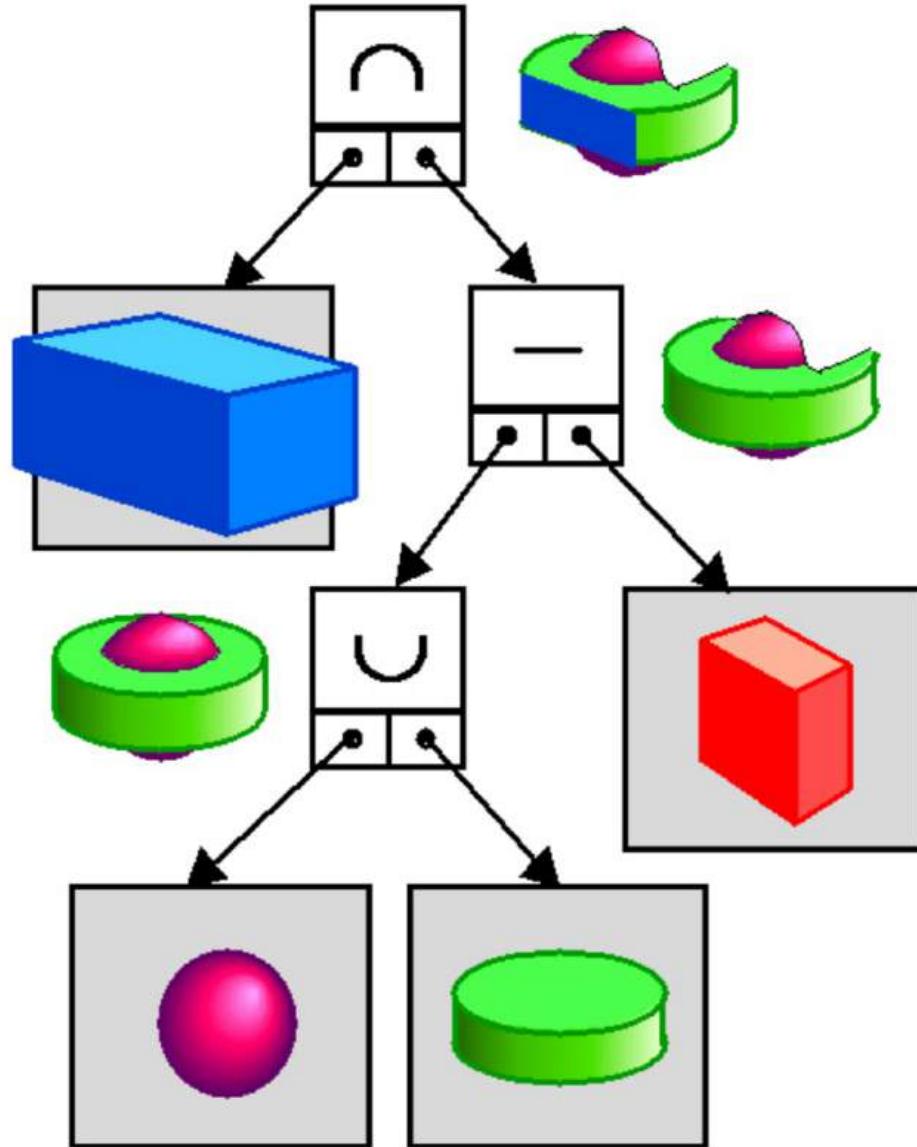
Constructive Solid Geometry (CSG)

Quellen:

- [EVC_Skriptum_CG, p.8](#)
- [EVC_Skriptum_CG, p.9](#)
- **Konstruktion:** Objekte werden durch **Mengenoperationen** (Vereinigung, Durchschnitt, Differenz) auf **dreedimensionale Primitive** (Kugel, Tetraeder, Würfel, Zylinder etc.) konstruiert.
- **Datenstruktur:** Anordnung in einer **hierarchischen Datenstruktur**, dem sogenannten **CSG-Baum** (eigentlich ein kreisfreier Graph).
- **Konsistenz:** CSG-Objekte sind immer **konsistent** (keine Löcher, wohldefiniertes Inneres), da Primitive trivialerweise konsistent sind und die Mengenoperationen Konsistenz bewahren.
- **Knoteninformation:** Jeder Knoten im CSG-Baum enthält zusätzlich **Transformationen (Matrizen)**, die auf den darunterliegenden Teilbaum angewendet werden.
- **Vorteile:**
 - **Exakte Repräsentation:** Primitive behalten ihre exakte geometrische Form (z.B. eine perfekte Kugel).
 - **Geringer Speicherbedarf.**
 - **Einfache Transformationen.**
- **Nachteile:**
 - **Aufwändiges Rendering:** Komplizierte Berechnung von Bildern.
 - **Lösungsmöglichkeiten für Rendering:**

- Umwandlung der CSG-Struktur in eine **B-Rep-Repräsentation** und herkömmliches Rendering.
- Direkte Bilderstellung mittels **Ray-Casting** oder **Ray-Tracing**.

Hier ein Beispiel:

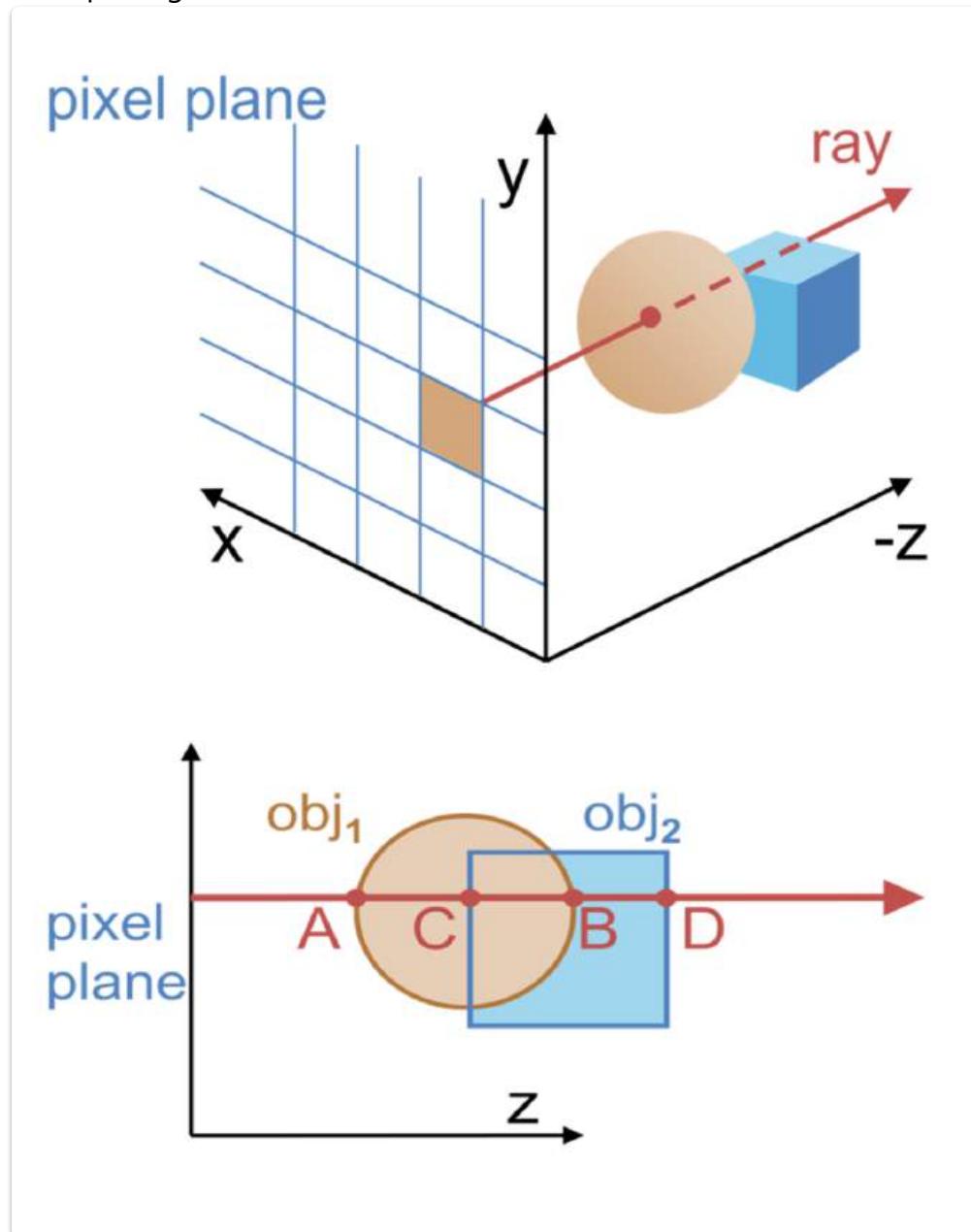


Ray-Casting von CSG-Objekten

EVC_Skriptum_CG, p.9

- **Methode:** Pixelweise Bildberechnung. Für jedes Pixel wird ein **Strahl (Ray)** in Blickrichtung ausgesendet und mit allen Objekten geschnitten. Der **vorderste Schnittpunkt** bestimmt das sichtbare Objekt und dessen Farbe für das Pixel.
- **Berechnung im CSG-Baum (rekursiv):**
 - **Endknoten (Primitive):** Berechnung aller Schnittpunkte ist einfach.
 - **Zwischenknoten (Operationen):** Verknüpfung der Schnittpunktlisten der beiden Nachfolger entsprechend dem Operator:

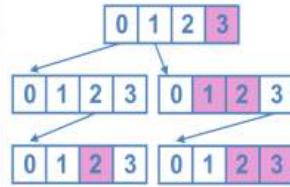
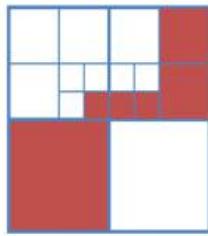
- **Vereinigung (Union):** Aus Listen (A,B) und (C,D) entsteht (A,D).
- **Durchschnitt (Intersection):** Aus Listen (A,B) und (C,D) entsteht (C,B).
- **Differenz (Difference):** Aus Listen (A,B) und (C,D) entsteht (A,C).
- **Wurzel:** Der **erste Punkt** der resultierenden verknüpften Schnittpunktliste wird ausgewählt.
- **Ray-Tracing:** Eine fortgeschrittenere Technik, die Ray-Casting als Teilmenge beinhaltet und später genauer behandelt wird.



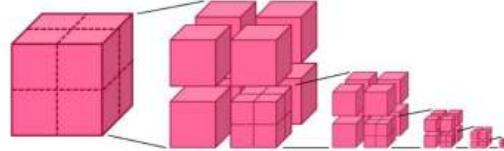
Quadtrees und Octrees

Ein **Quadtree** ist eine Datenstruktur, die zur Repräsentation beliebiger zweidimensionaler Strukturen geeignet ist. Der relevante Bereich wird überall dort in vier Viertel geteilt, wo die Information noch zu kompliziert ist um einfach abgelegt zu werden, andernfalls wird die einfache Information abgelegt. Jedem Bildbereich entspricht ein Knoten eines Baumes, in dem jeder Knoten (maximal) vier Nachfolger hat („Quadtree“).

Quadrant 0	Quadrant 1
Quadrant 3	Quadrant 2



Das nebenstehende Beispiel zeigt einen **einfachen Quadtree**, der eine zweifarbige einfache Graphik repräsentiert. Der Wurzelknoten entspricht dem ganzen Bild, die Knoten in der zweiten Reihe entsprechen den oberen zwei Vierteln des Bildes und die letzten zwei Knoten entsprechen den zwei Bereichen, die am feinsten aufgelöst sind.



Ein **Octree** ist die Erweiterung dieses Konzeptes auf **drei Dimensionen**. Ein beliebig geformtes Objekt (oder auch eine ganze Szene) innerhalb eines Würfels wird dadurch repräsentiert, dass „einfache“ Teilwürfel (leer

Grundprinzip

- Ein **Octree** ist die Erweiterung des Quadtrees auf **drei Dimensionen**.
- Der Raum (z.B. eine Szene oder ein Objekt) wird rekursiv in **acht Teilwürfel (Oktanten)** unterteilt.
- Jeder Knoten im Baum hat **acht Nachfolger**.
- Die Unterteilung wird fortgesetzt, bis:
 - der Teilwürfel **einfach** ist (komplett leer oder vollständig im Objekt),
 - oder eine festgelegte **Mindestgröße** erreicht wurde (z.B. ein Tausendstel der Gesamtausdehnung).

Aufbau des Baums

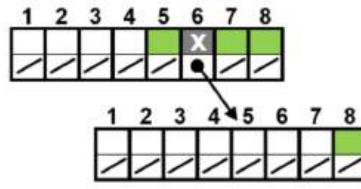
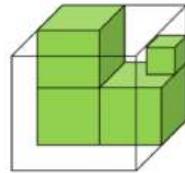
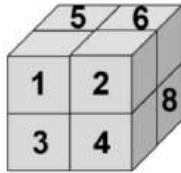
- **Einfache Teilwürfel** werden als **Blätter (Endknoten)** gespeichert.
- **Komplexe Teilwürfel** werden weiter in acht kleinere Würfel unterteilt.
- Die Struktur ergibt einen **rekursiven Baum**, in dem jeder Knoten bis zu acht Kinder hat.

Vorteile

- Kann **beliebige 3D-Formen** repräsentieren.
- **Schnelle Abfragen** möglich (z.B. Inhalt an einem bestimmten Punkt).
- **Mengenoperationen** sind einfach umzusetzen durch die rekursive Struktur.

Nachteile

- **Unpräzise Repräsentation** bei komplexen oder schrägen Oberflächen.
- **Hoher Speicherbedarf** bei feiner Auflösung.
- **Transformationen** (z.B. Drehungen) sind aufwändig, oft muss der Octree neu aufgebaut werden.



Linearisierung: X(EEEESX(EEEEEEES)SS)
E ... Empty, S ... Solid, X ... Mixed

Rendering (Darstellung)

- Ablauf bei Verwendung eines speicherbasierten Renderings:

```
if Knoten ist einfach
    then zeichne Knoten #d.h. tue nichts wenn Knoten leer ist
else rekursiver Aufruf der 8 Oktanten von hinten nach vorne
```

Quelle:

[EVC_Skriptum\(CG\)](#), p.10

Szenengraphen

Quelle: [EVC_Skriptum\(CG\)](#), p.10

Begriff:

- Objektorientierte Datenstruktur.
- Hierarchische Beschreibung der logischen und/oder räumlichen Anordnung von Elementen in 2D/3D-Szenen.
- Oberbegriff für verschiedene hierarchische Beschreibungsformen graphischer Objekte.

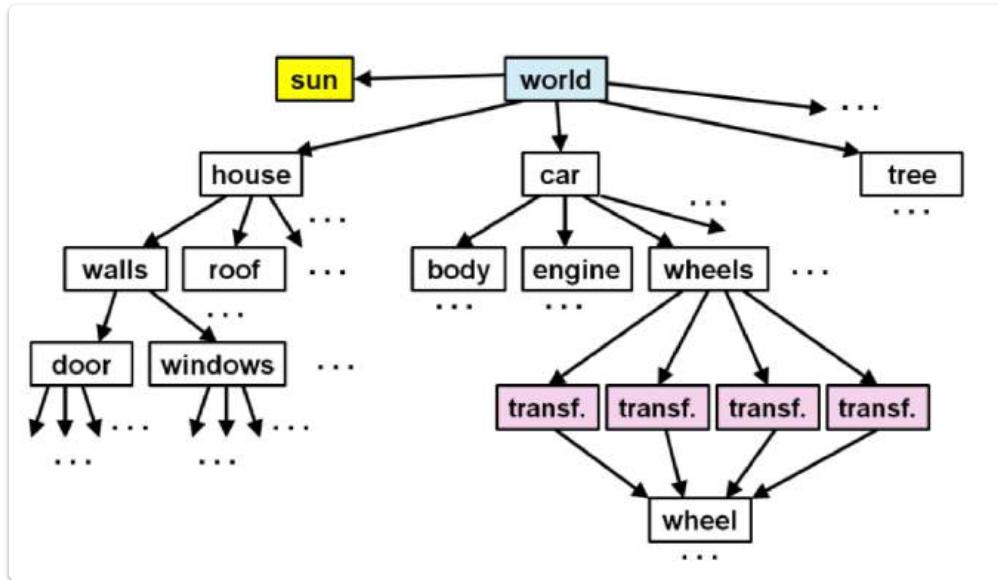
Struktur:

- Graphentheoretisch: Baumähnlicher gerichteter kreisfreier Graph.
- Wurzelknoten:** Repräsentiert die gesamte Szene.
- Zwischenknoten:** Beschreiben Teilszenen (Wurzeln von Teilbäumen).
- Endknoten:** Repräsentieren die einfachsten Objekte der Szene (unterschiedliche Repräsentationen möglich).

Beispiel (Stadt):

- Wurzel:** Stadt
- Zwischenknoten:** Haus, Fenster

- **Endknoten:** Fensterscheibe, Schraube



Vorteile/Konzepte:

- **Mehrfachnutzung:** Wiederkehrende Elemente können im Graphen mehrfach referenziert werden.
- **Transformationen:** Zwischenknoten enthalten Informationen für den gesamten Teilgraphen:
 - Material, Farbe
 - Position, Lage im Raum
 - Größe
 - Verzerrung
- **Matrixdarstellung:** Geometrische Transformationen können elegant in Matrizen gespeichert werden.

Relevanz:

- Zentrales Konzept in Systemen wie OpenSceneGraph, VRML und X3D.

Andere Objektrepräsentationen

- es gibt noch viele andere Objektrepräsentationen und Datenstrukturen
- Oft für sehr spezielle Anwendungen

Beispiele:

- **BSP-Bäume:**
 - Effiziente Kollisionserkennung.
 - Darstellung von komplexen Szenen.
- **Fraktale:**

- Naturnahe Formen (Landschaften, Wolken, Bäume).
- Kunst und Design.
- **Prozedurale Modelle:**
 - Generierung von Landschaften, Gebäuden, Pflanzen.
- **Partikelsysteme:**
 - Effekte wie Rauch, Feuer, Wasser, Explosionen.
- **Physikalisch basierte Modelle:**
 - Realistische Simulation von Objekten und Umgebungen.
- **3D-Volumendaten:**
 - Medizinische Bildgebung (CT, MRT).
 - Geologische Daten.

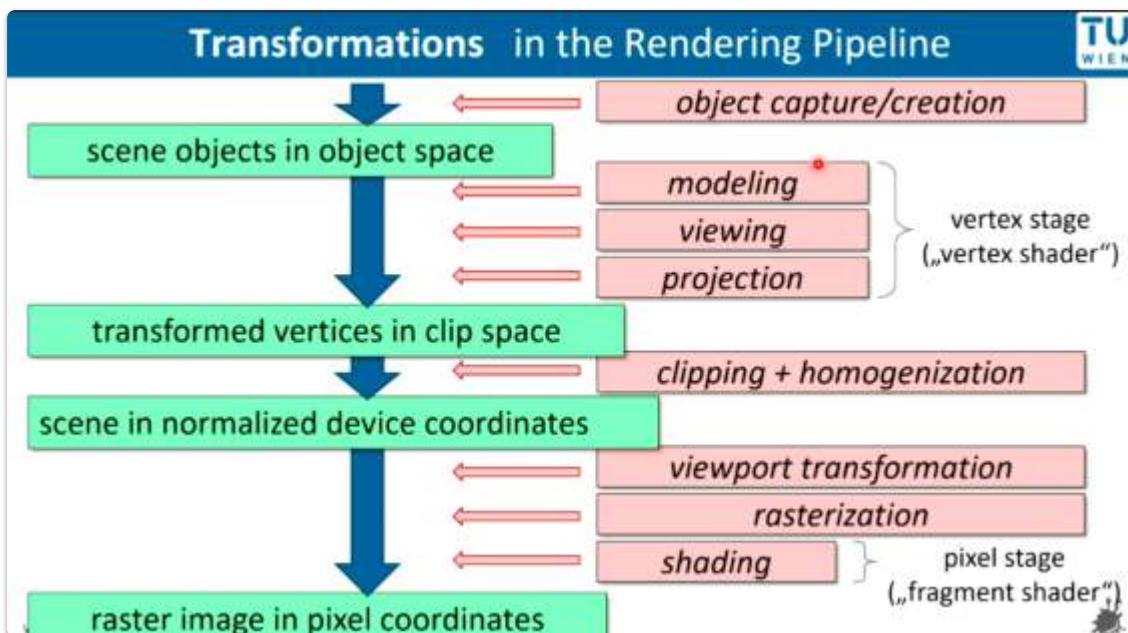
Quelle: [EVC_Skriptum_CG](#), p.10

3. Transformationen

Behandelt:

- Verschieben
 - Vergrößern
 - Verkleinern
 - Drehen
 - Spiegeln
 - usw...
- ...innerhalb oder zwischen Koordinatensystemen.

In der Rendering Pipeline wäre das hier:



Einfache 2D-Transformationen

Translation (Verschiebung)

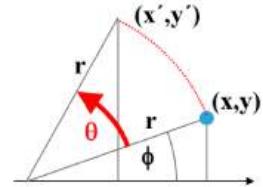
Das Verschieben eines Punktes (x, y) um den Vektor (t_x, t_y) liefert den transformierten Punkt:

$$(x', y') = (x + t_x, y + t_y)$$

Rotation (Drehung)

Durch das Drehen eines Objektes mit dem Winkel θ um den Koordinatenursprung kommt der Punkt (x, y) wegen $x = r \cdot \cos\theta$ und $y = r \cdot \sin\theta \Rightarrow x' = r \cdot \cos(\phi + \theta) = r \cdot \cos\phi \cdot \cos\theta - r \cdot \sin\phi \cdot \sin\theta = x \cdot \cos\theta - y \cdot \sin\theta$ (und y' analog) auf

$$(x', y') = (x \cdot \cos\theta - y \cdot \sin\theta, x \cdot \sin\theta + y \cdot \cos\theta)$$



zu liegen.

Skalierung (Vergrößerung oder Verkleinerung)

Beim Skalieren eines Objektes um den Faktor s um den Ursprung $(0, 0)$ wird ein Punkt (x, y) auf

$$(x', y') = (s \cdot x, s \cdot y)$$

abgebildet. Wenn in x- und y-Richtung unterschiedliche Skalierungsfaktoren s_x und s_y verwendet werden, dann erhält man

$$(x', y') = (s_x \cdot x, s_y \cdot y).$$

Reflexion (Spiegelung)

Die Spiegelung an einer Koordinatenachse ist ein Sonderfall der Skalierung mit $s_x = -1$ oder $s_y = -1$.

Alle anderen Transformationen können durch Hintereinanderausführen der beschriebenen einfachen Transformationen erreicht werden. Diese Abbildungen (mit Ausnahme der Translation) lassen sich auch durch Transformationsmatrizen darstellen. Dabei werden die Punkte als Vektoren dargestellt, um mit ihnen die Matrixoperationen durchführen zu können:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \quad (x', y') = (x + t_x, y + t_y) \quad \dots?$$

(a) Skalierung

(b) Rotation (gegen Uhrzeigersinn)

(c) Spiegelung um x-Achse

(d) Verschiebung

Homogene Koordinaten

Grundprinzip

- Ziel: Auch Translation soll in Matrixform darstellbar sein → **homogene Koordinaten**
- Erweiterung eines Punkts (x, y) zu (x, y, h) , meist mit $\mathbf{h} = 1$
- Rückrechnung in 2D:
 - $x = \frac{x'}{h}$
 - $y = \frac{y'}{h}$

Grundlegende Transformationen

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

(a) 2D-Skalierung

(b) 2D-Rotation

(c) 2D-Translation

Vorteil einheitlicher Matrixform

- Alle Transformationen können als Matrizen dargestellt und kombiniert werden
- Durch **Assoziativität** der Matrizen: Vorab-Multiplikation möglich → eine Gesamtmatrix M
→ effizientere Berechnung

Warum ist es vorteilhaft, alle Transformationen in einheitlicher Matrixschreibweise zu formulieren? Meist werden größere Teile (Objekte, Bilder) als Ganzes transformiert, d.h. auf jeden Punkt dieser Gebilde wird die gleiche Folge von Transformationen angewendet. Dies entspricht einer sequenziellen Multiplikation eines Punktes P mit Matrizen M_1, M_2, M_3, \dots : $P' = M_1 \cdot P, P'' = M_2 \cdot P', P''' = M_3 \cdot P'', \dots$ Nun kann man sich die Assoziativität der Matrixmultiplikation [also $(M_1 \cdot M_2) \cdot M_3 = M_1 \cdot (M_2 \cdot M_3)$] zunutze machen und den Rechenaufwand damit massiv reduzieren:

$$\text{Statt } P^{(n)} = M_n \cdot (M_{n-1} \cdot \dots \cdot (M_3 \cdot (M_2 \cdot (M_1 \cdot P)))) \dots \\ \text{schreibt man } P^{(n)} = (M_n \cdot M_{n-1} \cdot \dots \cdot M_3 \cdot M_2 \cdot M_1) \cdot P.$$

Nun kann man $M = (M_n \cdot M_{n-1} \cdot \dots \cdot M_3 \cdot M_2 \cdot M_1)$ vorher ausrechnen und diese eine Gesamtmatrix dann auf alle Punkte anwenden.

Kurznotation für Transformationen

- $T(tx, ty)$ = Translation um Vektor (tx, ty)
- $R(\theta)$ = Rotation um Winkel θ
- $S(s_x, s_y)$ = Skalierung in x- und y-Richtung

Inverse Transformationen

- $T^{-1}(tx, ty) = T(-tx, -ty)$
- $R^{-1}(\theta) = R(-\theta)$
- $S^{-1}(sx, sy) = S(1/sx, 1/sy)$

Komplexere Transformationen

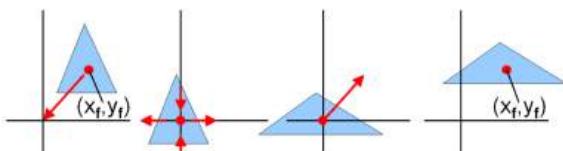
Skalierung um einen anderen Punkt als den Koordinatenursprung:

1. Schritt = Verschieben des Skalierungszentrums in den Koordinatenursprung: $T(-x_f, -y_f)$
2. Schritt = Skalieren des Objektes im Koordinatenursprung: $S(s_x, s_y)$
3. Schritt = Zurückverschieben des Objektes an die ursprüngliche Stelle: $T^{-1}(-x_f, -y_f) = T(x_f, y_f)$

Die allgemeine Matrix für die Skalierung mit (x_f, y_f) als Zentrum erhält man nun so:

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

Als weiteres Beispiel betrachten wir die



Spiegelung an einer beliebigen Achse $y = mx + b$:

1. Schritt = Verschieben, so dass die Achse durch den Koordinatenursprung geht: $T(0, -b)$
2. Schritt = Drehen, so dass die Achse z.B. mit der x-Achse zusammenfällt: $R(-\theta)$ [$m = \tan\theta$]
3. Schritt = Spiegeln an der x-Achse: $S(1, -1)$
4. Schritt = Zurückdrehen, so dass Achse ursprünglichen Winkel hat: $R^{-1}(-\theta) = R(\theta)$
5. Schritt = Zurückverschieben, so dass Achse an der ursprünglichen Stelle liegt: $T^{-1}(0, -b) = T(0, b)$

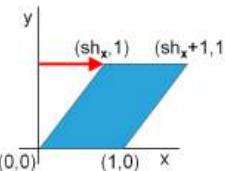
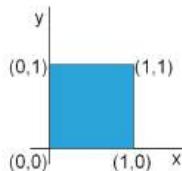
Die allgemeine Matrix für die Spiegelung an der Achse $y = mx + b$ erhält man nun so:

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

Scherung

Eine weitere wichtige Transformation ist die Scherung, sie hat im einfachsten Fall in x-Richtung mit fixierter x-Achse die Form:

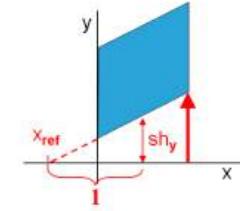
$$\begin{bmatrix} 1 & \text{sh}_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Etwas allgemeiner kann die Scherung auch an einer Parallelen zu einer Achse erfolgen, das sieht etwa in y-Richtung dann so aus:

Natürlich kann man jetzt auch wieder ganz leicht die allgemeine Matrix für eine Scherung ableiten, die nicht parallel zu einer der Koordinatenachsen erfolgt: zuerst Drehung in achsenparallele Lage, dann Scherung, und dann wieder zurückdrehen.

$$\begin{bmatrix} 1 & 0 & 0 \\ \text{sh}_y & 1 & -\text{sh}_y \cdot x_{\text{ref}} \\ 0 & 0 & 1 \end{bmatrix}$$



Viewing-Transformation

- Transformation von Weltkoordinaten in **Kamerakoordinaten** (auch Viewport-Koordinaten genannt)
- Kombination aus:
 - Ursprung verschieben
 - Rotation zur Achsenausrichtung
 - Skalierung der Achsen
- Kein Rückverschieben/-drehen nötig

Affine Transformationen

- Alle behandelten Transformationen = affine Transformationen
- Definition: Koordinaten werden über lineare Abbildung + Translation transformiert
- Eigenschaften:
 - Erhalten Kollinearität (3 Punkte auf Linie bleiben auf Linie)
 - Erhalten Verhältnis von Streckenlängen auf Geraden
 - Parallele Linien bleiben parallel
 - Endliche Punkte bleiben endlich
- Zusammensetbar aus: Skalierung, Rotation, Translation, Scherung, Spiegelung
- Transformationen mit nur Rotation, Translation, Spiegelung = längen- und winkelerhaltend

3D Transformationen

3D Transformationen

Alle 2D-Konzepte lassen sich leicht auf 3D erweitern. Man benötigt wieder eine **homogene Komponente**, so dass 4x4-Matrizen auf 4-dimensionalen Vektoren operieren. Später werden wir sehen, dass man auch Projektionen auf diese Art formulieren kann.

Hier sind einmal die **wichtigsten 3D-Transformationen**:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

(a) 3D-Skalierung

(b) 3D-Translation

(c) Spiegelung um yz-/xz-/xy-Ebene

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

(d) 3D-Rotation um x-Achse

(e) 3D-Rotation um y-Achse

(f) 3D-Rotation um z-Achse

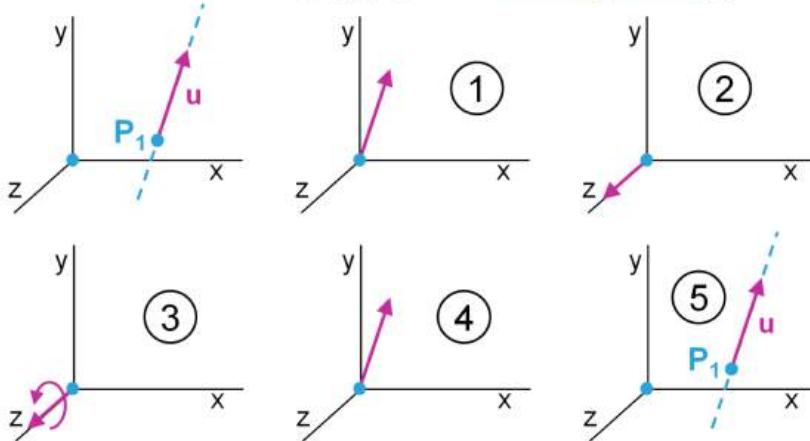
Die Namen für diese einfachen Transformationsmatrizen sehen nun so aus:

- $\mathbf{T}(t_x, t_y, t_z)$ = Translation um den Vektor (t_x, t_y, t_z)
- $\mathbf{R}_x(\theta)$ = Rotation um den Winkel θ um die x-Achse; y- und z-Achse analog
- $\mathbf{S}(s_x, s_y, s_z)$ = Skalierung um die Faktoren s_x, s_y und s_z .

Als Beispiel für eine komplexere Transformation wollen wir eine

Drehung um den Winkel θ um eine beliebige Achse im Raum

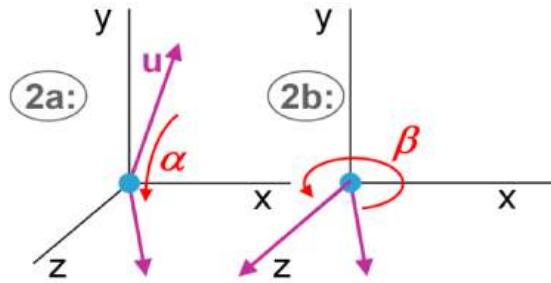
herleiten. Die Achse sei durch einen Punkt $P_1(x_1, y_1, z_1)$ und einen Richtungsvektor u gegeben.



1. Schritt = Punkt P_1 in den Koordinatenursprung verschieben: $T(-x_1, -y_1, -z_1)$
2. Schritt = Vektor u in die z-Achse drehen
 - (a) Vektor u um die x-Achse in die xz-Ebene drehen: $R_x(\alpha)$
Sei $u = (a, b, c)$, dann ist $u' = (0, b, c)$ die Projektion von u auf die yz-Ebene. Der Drehungswinkel α um die x-Achse ergibt sich aus $\cos\alpha = c/d$ mit $d = \sqrt{(b^2 + c^2)}$
 - (b) Vektor u um die y-Achse in die z-Achse drehen: $R_y(\beta)$
Der Drehungswinkel β um die y-Achse ergibt sich aus $\cos\beta = d$ (bzw. $\sin\beta = -a$)
3. Schritt = Drehung um θ um die z-Achse ausführen: $R_z(\theta)$
4. Schritt = Vektor u in die ursprüngliche Richtung zurückdrehen: zuerst $R_y(-\beta)$, dann $R_x(-\alpha)$
5. Schritt = Punkt P_1 an die ursprüngliche Position zurückverschieben: $T(x_1, y_1, z_1)$

Die resultierende Matrix berechnet sich also so:

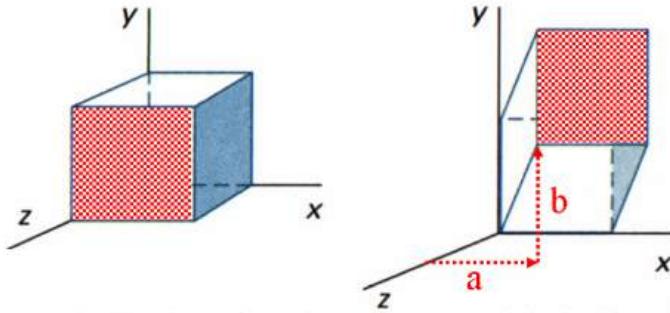
$$\begin{aligned} R(\theta) &= T^{-1}(-x_1, -y_1, -z_1) \cdot R_x - 1(\alpha) \cdot R_y - 1(\beta) \cdot R_z(\theta) \cdot R_y(\beta) \cdot R_x(\alpha) \cdot T(-x_1, -y_1, -z_1) = \\ &= T(x_1, y_1, z_1) \cdot R_x(-\alpha) \cdot R_y(-\beta) \cdot R_z(\theta) \cdot R_y(\beta) \cdot R_x(\alpha) \cdot T(-x_1,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



Die **Scherung in 3D** ist ebenfalls einfach darstellbar:

Eine Scherung parallel zur xy-Ebene um den Wert a in x-Richtung und den Wert b in y-Richtung erreicht man mittels

$$\begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Eine Scherung mit einer anderen Fixebene als einer der Koordinatenhauptebenen kann man sich leicht ableiten.

4. Farbe

Farbe ist eine der Grundesszenen der Computergraphik. Farbe richtig zu verstehen und zu handhaben ist ein Grundwerkzeug für Computergraphiker. Das häufig verwendete RGB-Farbmodell ist jedoch nicht in der Lage, alle Farben darzustellen, und auch sonst sehr approximativ. Viele Farbberechnungen werden meist nur näherungsweise gemacht (was oft reicht), und die exakte Farbenlehre ist sehr komplex

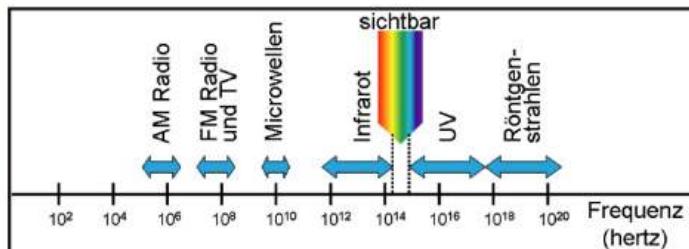
Ich habe zum Thema Farbe so gut wie keine PowerPoint slides eingebaut, bei Bedarf hier schauen: [EVC-CG04-Farbe_2025S_Slides.pdf](#)

Was ist Farbe?

Unser Auge kann elektromagnetische Strahlung im eher engen Frequenz-bereich zwischen etwa $3,8 \cdot 10^{14} \text{ Hz} (\approx 780 \text{ nm})$ und $7,8 \cdot 10^{14} \text{ Hz} (\approx 380 \text{ nm})$ erkennen. Dabei empfinden wir diese Strahlung als Licht. Dieser sichtbare Bereich ist von Mensch zu Mensch leicht unterschiedlich, und viele Tiere haben andere Grenzen. Andere Frequenzbereiche dienen anderen Zwecken (siehe Diagramm).

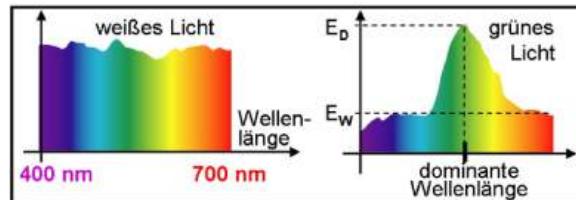
Unser Auge kann innerhalb des sichtbaren

Bereiches sogar unterscheiden, welche Frequenz die Strahlung hat, das empfinden wir dann als unterschiedliche Farben. Langwelligeres Licht (also niedrigere Frequenz) empfinden wir als rot, kurzwelligeres Licht (also höhere Frequenz) als blau bis violett. Dazwischen liegen alle Regenbogenfarben.



[zur Erinnerung: $c = \lambda \cdot f$, wobei c ... Lichtgeschwindigkeit, λ ... Wellenlänge, f ... Frequenz]

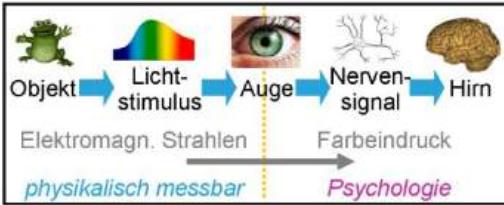
Tatsächlich kommt in der Natur aber höchst selten spektralreines Licht vor (das nur genau eine Wellenlänge hat), sondern meist sehen wir eine Mischung aus vielen Farben (Spektrum). Frequenzen mit mehr Energie bestimmen dann welche Farbe wir wahrnehmen, man spricht von dominanter Wellenlänge. Sind alle Anteile (ungefähr) gleich groß, so sehen wir ein farbloses Licht (also weiß oder grau). Wenn man mit E_D die Energie der dominanten Wellenlänge bezeichnet, und mit E_W die durchschnittliche Energie der anderen Wellenlängen, so nennt man $(E_D - E_W)/E_D$ die Reinheit (purity) einer Farbe. Die Helligkeit ergibt sich als die Fläche (Integral) unter der Spektralkurve.



Kolorimetrie

Kolorimetrie

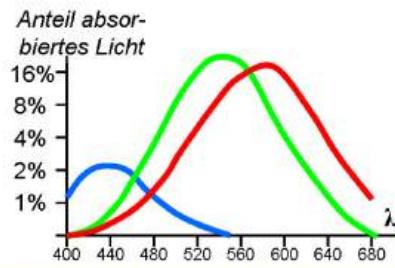
Die Kolorimetrie ist die Wissenschaft von der technischen Beschreibung von Farben. Man möchte also eine Farbe durch Zahlen, durch exakte Angaben beschreiben. Da aber eine Farbe ein empfundener Sinneseindruck ist, und keine physikalisch direkt messbare Größe, kann nur der visuelle Stimulus numerisch definiert werden (also das, was ein Mensch sieht), und zwar so dass



1. Stimuli mit den gleichen Spezifikationen unter gleichen Bedingungen gleich aussehen,
2. Stimuli die gleich aussehen die gleichen Spezifikationen haben,
3. die verwendeten Zahlen stetige Funktionen der physikalischen Parameter sind (d.h. kleine Änderungen der Zahlen bewirken kleine Änderungen der Farben und umgekehrt).

Kolorimetrie berücksichtigt also nur die *visuelle Unterscheidbarkeit* von elektromagnetischer Strahlung. Alle Spektren, die den gleichen Farbeindruck erzeugen, sind in diesem Sinn nicht unterscheidbar, bilden eine Äquivalenzklasse im Farbraum.

Die *Retina* des Auges, das ist die lichtempfindliche Schicht im hinteren inneren Bereich des Augapfels, enthält etwa 120 Millionen Stäbchen und Zapfen. Stäbchen können keine Farben unterscheiden, dafür sind sie sehr lichtempfindlich. Zapfen sind wesentlich weniger leicht aktivierbar, dafür gibt es drei verschiedene Arten, wobei jede Art in einem anderen Wellenlängenbereich empfindlich ist (die Empfindlichkeitskurven sind in der Graphik rechts abgebildet). Unser Farbempfinden setzt sich folglich aus der Kombination von drei getrennten „nicht-farbigen“ skalaren Signalen zusammen, daher bezeichnet man das menschliche Farbempfinden als *Tristimulus*. Die Empfindlichkeitskurven der drei Zapfenarten haben ihre Maxima bei Rot, Grün und Blau, es ist also durchaus angebracht, von Rot-, Grün und Blau-Zapfen zu sprechen. Erst das Gehirn mischt diese 3 Werte zu einer Farbe zusammen. Dies ist auch die Grundlage dafür, dass man dem Auge „alle“ Farben dadurch vorgaukeln kann, dass man eine Farbe aus nur 3 Grundfarben zusammensetzt. Wenn man kleine Lichtpunkte in rot, grün und blau nahe genug nebeneinander platziert, nehmen wir dies als einen Punkt in der so *additiv* gemischten Farbe wahr.



The Human Eye

Werner Purgathofer

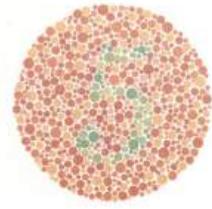
TU WIEN

retina contains:

- rods: black/white (grey)
- cones: color

Farbfehlsichtigkeit:

Bei manchen Menschen fehlt erbbedingt eine Zapfenart (oder sogar zwei) oder es sind die Empfindlichkeitskurven der Zapfen nicht ausreichend verschieden, dann fehlt die Fähigkeit, so viele verschiedene Farben wie die meisten zu unterscheiden. Man spricht von *Farbschwäche* oder *Farblindheit*. Die häufigste Art ist Rot-Grün-Blindheit, bei der die Rot- und Grün-Zapfen auf zu ähnliche Wellenlängen reagieren. Etwa 8% aller Männer sind zumindest geringfügig farbfehlsehig! Testbilder, in denen die Information nur erkennbar ist, wenn man bestimmte (z.B. rötliche und grünliche) Töne gleicher Helligkeit unterscheiden kann (siehe Bild), dienen zur Diagnose von Farbfehlsehigkeit. Da man im Leben auch mit reduziertem Farbsehen sehr gut zurecht kommt, wissen viele Leute gar nichts von ihrer Einschränkung.



Mögliche Beeinträchtigungen:

red/green blindness

→ red & green cones too similar

blue blindness

→ no blue cones

monochromatism

→ all cones missing

mehr zum Auge: [2. Bildaufnahme](#)

Farbmodelle:

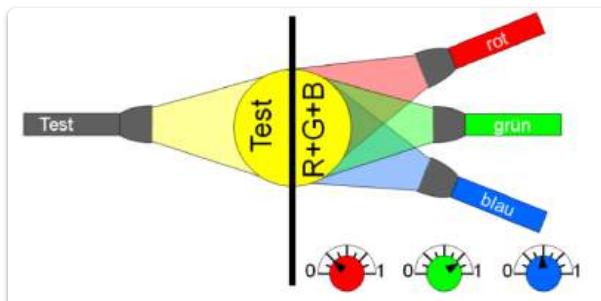
CIE 1931 XYZ-Farbmodell

[EVC_Skriptum_CG, p.16](#)

Grundlage: Tristimulus-Theorie (Farbwahrnehmung durch 3 Zapfentypen).

Experiment: Farbvergleich mit 3 Grundfarben (Rot, Grün, Blau) zur Erzeugung einer Testfarbe.

- **Problem:** Negative Farbanteile nötig (Grundfarbe muss zur Testfarbe addiert werden).



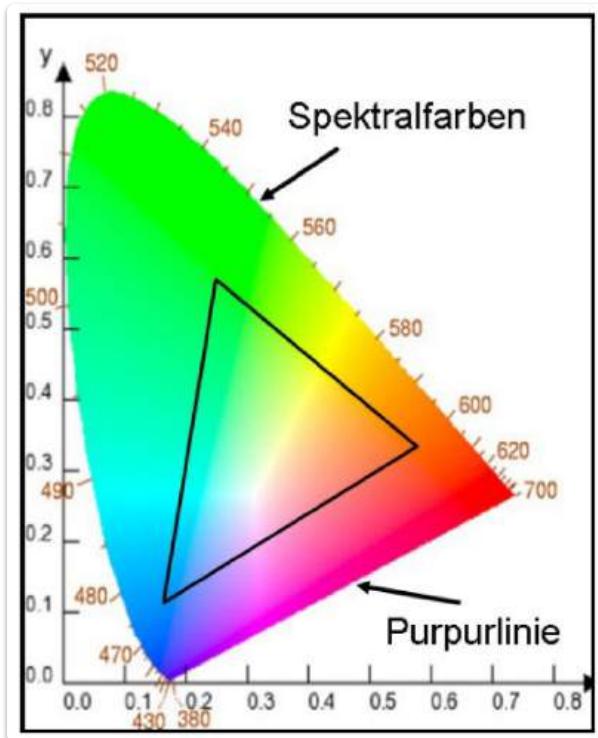
Transformation: Umwandlung in positive "imaginäre Grundfarben" X, Y, Z.

CIE-Diagramm (1931):

- Normierung auf Helligkeit 1.
- Projektion auf die XY-Ebene mit Koordinaten (x, y) .
- z ergibt sich aus $x + y + z = 1$.
- Vollständige Farbdefinition: (x, y, Y) , wobei Y die Helligkeit darstellt.

Eigenschaften des CIE-Diagramms:

- **U-förmige Außenkante:** Spektralreine Farben (monochromatisches Licht).
- **Purpurlinie:** Verbindet die Endpunkte der U-Form, enthält Komplementärfarben spektraler Farben (keine reine Wellenlänge).
- **Jeder Punkt:** Représentiert eine andere Farbe.
- **Linearkombination zweier Farben:** Liegt auf der geraden Linie zwischen den Farben.
- **Weißpunkt:** Liegt etwa in der Mitte.
- **Komplementärfarben:** Liegen an entgegengesetzten Enden einer Geraden durch den Weißpunkt.
- **RGB-Monitor-Farbraum:** Darstellbare Farben liegen innerhalb des Dreiecks, das durch die Rot-, Grün- und Blau-Punkte des Monitors aufgespannt wird.
- **Begrenzung:** Kein Monitor kann alle sichtbaren Farben darstellen, da keine drei realen Farben das gesamte CIE-Diagramm abdecken.

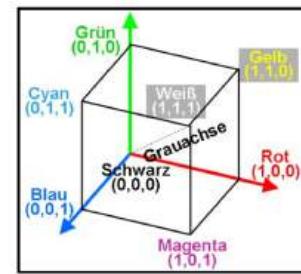


RGB Farbmodell

RGB-Farbmodell



Neben Farbräumen (eigentlich Farbraumbeschreibungen) wie dem CIE-Modell, die alle Farben zu beschreiben imstande sind, gibt es Farbräume zur Beschreibung der Farben eines Gerätes. Für Bildschirme wird fast immer



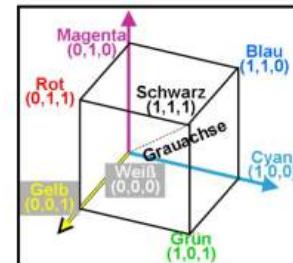
das RGB-Modell verwendet. Dabei wird ein Pixel aus drei kleinen Farb-punkten zusammengesetzt, deren Lichtsumme (*additive Farbmischung!* – siehe Skizze mit den Kreisen) einen Farbeindruck erzeugt. Je nach verwendeter Technologie und konkreten Materialien hat jeder Monitor geringfügig unterschiedliche Grundfarben, aus denen unterschiedliche Teilmengen aller Farben erzeugt werden können. Den Raum der Farben, die ein Gerät erzeugen kann, nennt man sein *Gamut*.

CMY Farbmodell

CMY-Farbmodell

Das *Mischen von farbiger Tinte auf einem Blatt Papier* unterliegt ganz anderen Regeln als die additive Farbmischung von Licht. Je mehr Tinte man verwendet, desto dunkler wird das Ergebnis, weil man ja eigentlich einen Filter vor das passiv reflektierende Papier bringt, daher spricht man von *subtraktiver Farbmischung* (siehe Skizze mit den Kreisen). Das CMY-Modell dazu ist das *Komplement des RGB-Raumes*. Für einfache Anwendungen gilt daher

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

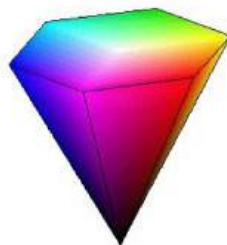


Vielfach kommt einem auch das *CMYK-Modell* unter. K steht dabei für Key, das entspricht der Farbe Schwarz. Beim Druck werden hierbei alle Grauanteile mit schwarzer Farbe extra gedruckt statt sie als Mischung gleicher Anteile von Cyan, Magenta und Yellow teurer und schlechter zu erzeugen.

HSV- und HLS- Farbmodelle

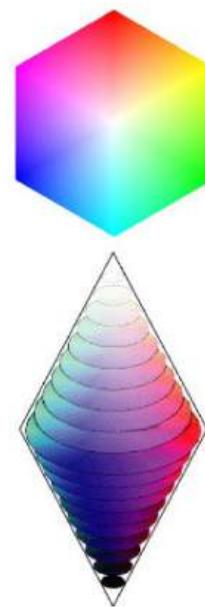
Neben den für Geräte sinnvollen Farbräumen gibt es noch Beschreibungen der Farben in einer Weise, die dem *menschlichen Benutzer* entgegen kommt. Wir können nur sehr schwer und mit viel Übung eine Zielfarbe aus den Komponenten R, G, B oder C, M, Y beschreiben. Unsere üblichen Beschreibungen von Farben setzen sich aus Qualitäten wie einem Farbwort, einer *Helligkeit* und einer *Farbreinheit* zusammen. Daher werden für das User-Interface zur Farbdefinition solche Farbsysteme verwendet, die in diesen 3 Dimensionen funktionieren. Dazu gehören *HLS*, *HSV*, *Munsell*, *RAL*, *NCS*, *Coloroid* und einige andere.

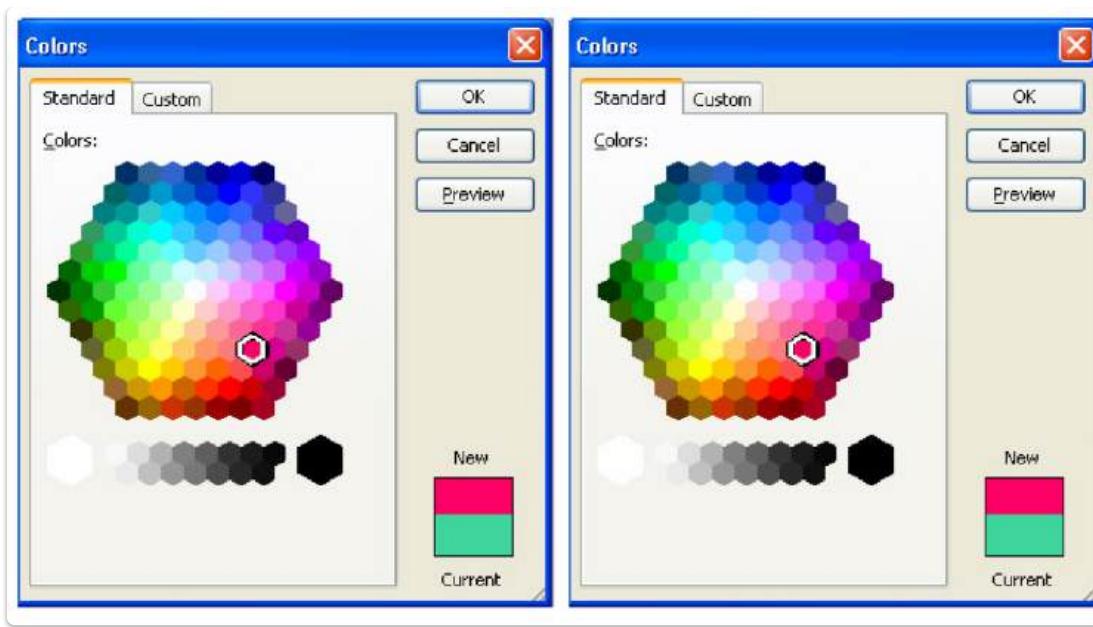
HSV steht für *Hue*, *Saturation* und *Value*. Hue heißt Bunnton oder Farbton, bezeichnet die Farbe entlang eines Farbkreises, der von Rot über Orange, Gelb, Grün, Cyan, Blau, Violet, Magenta wieder ins Rot geht. Wenn man den RGB-Würfel genau in Richtung seiner Grau-achse anschaut, so sieht man diesen Farbkreis als Grenze des entstehenden Sechsecks (Abbildung). Saturation heißt Sättigung und gibt an wie rein eine Farbe ist, wie stark sie sich also von Grau unterscheidet. Value heißt Wert und gibt so etwas wie die *Helligkeit* der Farbe an.



Je dunkler nun eine Farbe ist, desto weniger Abstufungen der Sättigung gibt es. Dadurch lassen sich alle Farben in einer Pyramide darstellen, deren Spitze schwarz ist und deren Grundfläche das Farb-Sechseck ist (Abb. links). Die Farbe wird in Grad entlang der Basiskante angegeben (Rot=0°, Grün=120°, Blau=240°), die Sättigung in Prozent des Abstandes von der Pyramidenachse und die Helligkeit als Prozent des Abstandes der Grundfläche von der Spitze. Ein mittelhelles gesättigtes Gelb hat damit den HSV-Wert (60, 1, 0.5).

Ganz ähnlich funktioniert das *HLS-System* (auch *HSL*), bei dem H=Hue, L=Lightness oder Luminance, S=Saturation heißen. Die Form des Modells ist jedoch diesmal ein Doppelkegel, der oben an der Spitze weiß ist und unten schwarz (Abb. rechts). Der Hintergrund ist die Annahme, dass Weiß viel heller ist als jede reine Farbe.





Meist hat man heutzutage in Desktopanwendungen mehrere verschiedene Farbmodelle zur Auswahl

Farbsymbolik

Quelle: [EVC_Skriptum_CG](#), p.18

- Farben spielen in unserem Leben eine große Rolle.
- Die Verwendung von Farben kann zwischen verschiedenen Kulturen divergieren
- Manche Bedeutungen beziehen sich nur auf ein gewisses Gebiet.

Sprachgebrauch

- **Grundvokabular:** Jede Sprache besitzt einen Kernbestand an Farbbezeichnungen.
- **Anzahl:** Variiert stark zwischen Sprachen (2 bis 20 grundlegende Termini).
- **Nuancen:** Zusätzlich existieren zahlreiche Bezeichnungen für Farbnuancen.
- **Deutsch:**
 - 6 bis 11 grundlegende Farbterme.
 - Ca. 150 bis 200 zusätzliche Bezeichnungen (z.B. oliv).
- **Andere Sprachen:**
 - **Italienisch:** Unterscheidung innerhalb einer Farbkategorie (z.B. *azzurro* für Himmelblau, *blu* für Dunkelblau).
 - **Ungarisch:** Unterscheidung innerhalb einer Farbkategorie (z.B. *piros* und *vörös* für Rot).
- **Fazit:** Die Kategorisierung und Benennung von Farben ist sprachabhängig und kann feiner oder gröber ausfallen.

Farbe in der Religion

- **Symbolkraft:** Farben besitzen in der Spiritualität oft symbolische und kulturell/religiös bedeutsame Inhalte.
- **Heilige Farbe:** In vielen Weltreligionen (außer dem Christentum) existiert eine heilige Farbe.
- **Islam:**
 - **Grün:** Lieblingsfarbe des Propheten Mohammed.
 - **Bedeutung:** Oft auf islamischen Staatsflaggen vertreten (z.B. Saudi-Arabien).

Farben in der Politik

- **Zuordnung:** Politische Strömungen und Parteien werden oft mit bestimmten Farben assoziiert.
- **Funktion:** Farben dienen als **einheitliches Erkennungsmerkmal**.
- **Beispiele:**
 - **Rot:** Marxismus-Leninismus, Sozialismus, Arbeiterbewegung.
 - **Grün:** Umweltorganisationen und -parteien.

Kennzeichnung durch Farben

- **Alleinstehendes Merkmal:** Farbe selbst transportiert Bedeutung ohne zusätzliche Erklärung.
- **Beispiele:**
 - **Wasserhähne:** Rot (warm), Blau (kalt).

Farben im Verkehr

- **Bewusste Information:** Farben in Verkehrszeichen, Lichtern und Ampeln sind codiert.
- **Rot/Weiß:** Verbots- und Gefahrenschilder.
- **Blau/Weiß:** Gebots- und Informationsschilder.
- **Ampel:** Rot (Stopp), Gelb (Vorsicht), Grün (Fahren).
- **Begrenzungslichter:** Rot (links vorbeifahren), Weiß (rechts vorbeifahren).

Farben in der Technik

- **Beschreibung von Teilen:** Farben kennzeichnen spezifische Komponenten.
- **Beispiel (Elektrik):** Phase, Nullleiter, Erdung (durch Drahtfarbe).

Farben in der Natur

- **Farbwirkung:** Natur nutzt Farben für verschiedene Zwecke.
- **Anlocken:** Buntes Balzgefieder/Schnäbel (Vögel).
- **Tarnung:** Anpassung an die Umgebung.

- **Warnung:** Abschreckung von Fressfeinden.

Assoziationen zu Farben (kulturabhängig)

- **Blau:** Himmel, Weite, Ferne, Sehnsucht, Phantasie.
- **Rot:** Blut, Krieg, Tod, Lebenskraft, Leidenschaft, Liebe, Zorn.
- **Grün:** Wiesen, Wälder, Natur, "grüner Daumen", Hoffnung, Zuversicht.
- **Gelb:** Sommer, Sonne, Lebensfreude, Licht, Gold, aber auch Neid, Geiz, Eifersucht, Egoismus, Verlogenheit.
- **Schwarz:** Tod, Ende, Leere, Trauer (in "weißen" Kulturen), Freude (in manchen "dunklen" Hautfarben-Kulturen).
- **Weiß:** Vollkommenheit (in "weißen" Kulturen), Trauer (in manchen "dunklen" Hautfarben-Kulturen), Freude (in "weißen" Kulturen).

5. Rasterisierung

EVC_Skriptum_CG, p.19

Was ist Rasterisierung?

- Prozess der **Umwandlung von Vektordaten** (z.B. Linien, Kurven) in **Pixel** zur Anzeige auf **bildschirmbasierten Ausgabegeräten**.
- Damit Grafiken angezeigt werden können, braucht es in der **Programmiersprache** entsprechende Befehle → sogenannte **Grafikprimitive**.

Grafikprimitive – die Bausteine der Darstellung

In 2D:

- **Punkte und Linien**
- **Polygone** (z.B. Dreiecke, Rechtecke)
- **Kreise, Ellipsen, Kurven** – auch in gefüllter Form
- **Bitmap-Operationen** (z.B. Bilder einfügen, verschieben)
- **Text** – Buchstaben, Zeichen, Zahlen

In 3D:

- **Dreiecke und andere Polygone** – Grundelemente für 3D-Modelle
- **Freiformflächen** – z.B. Bézier-Flächen, Splines

Zusätzliche Befehle zur Eigenschaftsdefinition

- Neben dem „Was zeichnen?“ braucht man auch „Wie?“
- Beispiele für **Eigenschaftsdefinitionen**:
 - **Farbe**
 - **Füllmuster**
 - **Textur** (Bild, das über Fläche gelegt wird)
 - **Materialeigenschaften** (für Beleuchtung, Glanz etc.)
 - **Transparenz**

ⓘ Merke:

Diese Einstellungen gelten **global**, d.h. sie beeinflussen **alle danach gezeichneten Objekte**, bis sie erneut geändert werden.

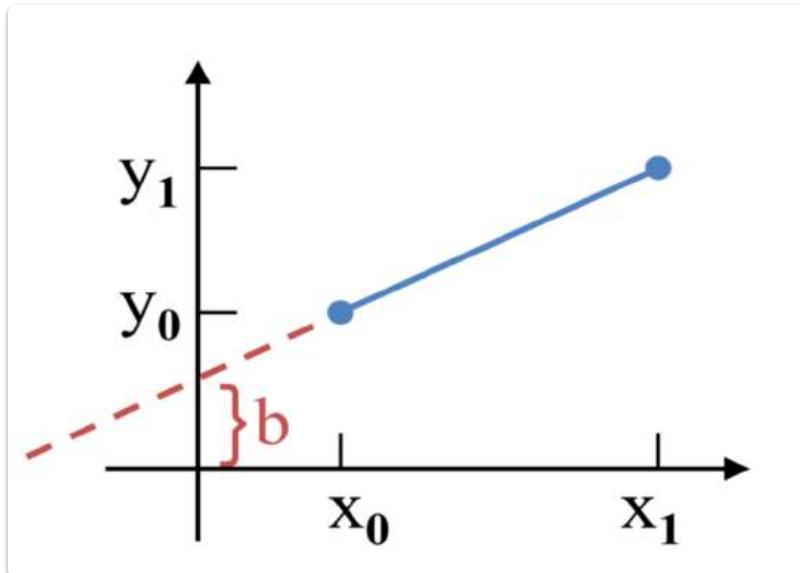
Linienalgorithmen

Linien werden in der Form $y = m * x + b$ angegeben wobei m den Anstieg beschreibt und $(0, b)$ den Schnittpunkt der y Achse.

Aus Endpunkten (x_0, y_0) und (x_1, y_1) kann man sich m und b berechnen:

$$m = \frac{(y_1 - y_0)}{(x_1 - x_0)}$$

$$b = y_0 - m * x_0$$



DDA-Verfahren

Der einfache DDA-Algorithmus für $|m| < 1$ zählt zu y_0 für jeden Schritt nach rechts ($x+ = 1$) den Wert m dazu und rundet das Ergebnis danach auf ganze Zahlen. Dadurch entsteht eine Linie, bei der für jeden x-Wert genau ein Pixel für die Linie erzeugt wird.

```

1 dx = x1 - x0; dy = y1 - y0;
2 m = dy / dx;
3
4 x = x0; y = y0;
5 setPixel (round(x), round(y));
6
7 for (k = 0; k < dx; k++) {
8     x += 1; y += m;
9     setPixel (round(x), round(y))
10 }
```

Für $|m| > 1$ werden x und y vertauscht, und das Verfahren wird in senkrechter Richtung durchgeführt. Auch der nachfolgende Bresenham-Algorithmus wird nur für $0 < m < 1$ dargestellt, die anderen Richtungen erhält man durch Spiegelung und durch Rotation um 90° .

DDA Line-Drawing Algorithm

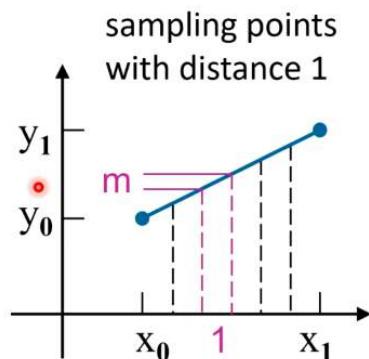


line equation: $y = m \cdot x + b$

$$\delta y = m \cdot \delta x \quad \text{for } |m| < 1$$

$$\left(\delta x = \frac{\delta y}{m} \quad \text{for } |m| > 1 \right)$$

DDA (digital differential analyzer):



$$\text{for } \delta x=1, |m|<1 : y_{k+1} = y_k + m$$

DDA – Based on Taylor Series Expansion



$$T_{f(x;a)} = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (x-a)^n = f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2 + \frac{f'''(a)}{6}(x-a)^3 + \dots$$

$$f(x+1) = g(f(x), f'(x), f''(x), \dots) \quad (x+1)^n = \sum_{k=0}^n \binom{n}{k} x^k$$

$$f(x) = x^2, \quad f'(x) = 2x, \quad f''(x) = 2, \quad f'''(x) = 0$$

$$f(x+1) = (x+1)^2 = x^2 + 2x + 1 = f(x) + f'(x) + 1$$

$$f'(x+1) = 2x + 2 = f'(x) + 2$$

Eduard Gröller



Bresenham-Verfahren

Man schaut ob man näher zum oberen oder unteren Pixel ist und setzt das dann so

Der **Bresenham-Algorithmus** erzeugt exakt dasselbe Ergebnis wie der einfache DDA, verwendet jedoch nur Integer-Arithmetik. Er ist dadurch schneller, leichter in Firm- oder Hardware zu implementieren, und überdies lässt er sich auch einfach für andere Kurven anpassen, z.B. Kreise, Ellipsen, Spline-Kurven usw.

Bei Linien lässt sich der nächste Punkt so berechnen:

$$y = m * (x_k + 1) + b$$

(lässt sich ganz einfach aus Linearen Funktionen erschließen)

Hier werden dann nicht die genauen y Werte berechnet sondern lediglich die Entscheidung getroffen, ob y_k oder y_{k+1} näher zum exakten y -Wert liegt.

Abstand zu y_k ist:

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

Abstand zu y_{k+1} ist:

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

Nun berechnet man sich die Differenz zwischen d_{lower} und d_{upper} :

$$d_{lower} - d_{upper}$$

- Wenn diese Differenz negativ ist, dann nimmt man den unteren Punkt (x_{k+1}, y_k)
- Wenn positiv den oberen (x_{k+1}, y_{k+1})

Optimierung durch Entscheidungsvariable

Um keine Fließkommaoperationen (Multiplikation/Division) durchführen zu müssen, wird eine **Entscheidungsvariable** eingeführt. Dazu setzt man:

$$m = \frac{\Delta y}{\Delta x} \quad \text{mit} \quad \Delta x = x_1 - x_0, \quad \Delta y = y_1 - y_0$$

Multipliziert man die obige Differenz mit Δx , ergibt sich:

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

Diese Entscheidungsvariable hat dasselbe Vorzeichen wie $d_{lower} - d_{upper}$, benötigt aber keine Division mehr.

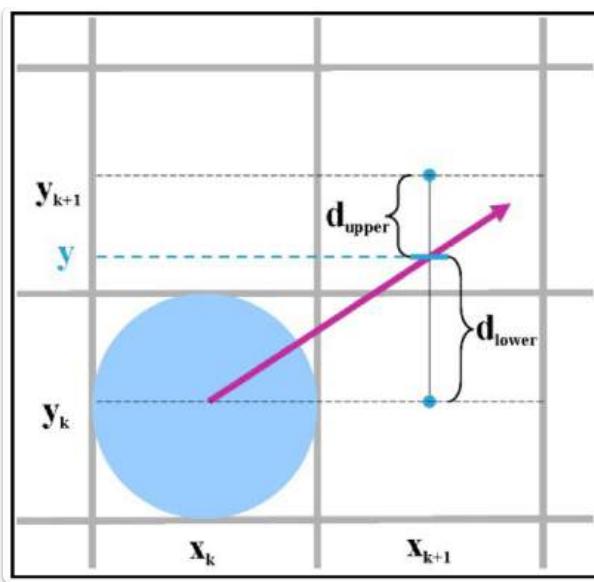
Rekursive Berechnung der nächsten Entscheidungsvariable:

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

Das bedeutet: Die neue Entscheidungsvariable lässt sich **einfach aus der vorherigen berechnen**, je nachdem, ob y erhöht wurde oder nicht – also ganz ohne Neuberechnung des exakten y -Werts.

Startwert:

$$p_0 = 2\Delta y - \Delta x$$



Attribute

Graphikprimitive können mit vielerlei Eigenschaften erzeugt werden, sogenannten Attributen.

Attribute von Punkten und Linien

Neben allgemein bekannten Eigenschaften von Linien, wie Strichdicke, Strichlierungsmuster, Farbe oder Pinseltyp, gibt es noch ein paar Attribute, die einem oft weniger bewusst sind. Dazu gehören etwa die Liniendenenden bei breiteren Linien sowie die Form von Ecken bei breiten Linien:



Weiters ist Antialiasing auch für Linien ein Thema, dazu werden etwas weiter unten Details gebracht.

Attribute von Text

[EVC_Skriptum_CG](#), p.21

Typische Eigenschaften von Text:

- **Font / Schriftart:**
z. B. *Courier, Helvetica, Times, Fraktal*
- **Stil:**
normal, fett, kursiv, unterstrichen, durchgestrichen etc.
- **Größe:**
in Punkten angegeben (z. B. 12pt)
- **Richtung:**
horizontal, vertikal, rotiert
- **Farbe:**
z. B. RGB-Werte oder vordefinierte Farbnamen

- **Bündigkeit:**
links, rechts, zentriert, Blocksatz

Serifen vs. serifenos

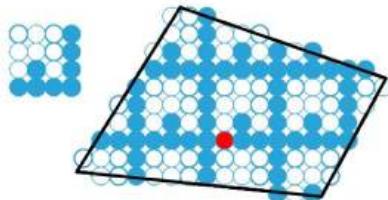
- **Fonts mit Serifen** (z.B. Times):
 - ✓ Besser geeignet für **Fließtext** – durch Serifen wird das Auge entlang der Zeile geführt.
- **Fonts ohne Serifen** (z.B. Helvetica):
 - ✓ Ideal für **plakativen Text**, Überschriften oder Bildschirmschirmdarstellung.



Attribute von (2D-) Polygonen und Flächen

Klarerweise sind die **Attribute des Randes von Flächen** dieselben wie die von Linien. Dazu kommt nun die Fläche selbst, die mit einer Füllung versehen werden kann. Muster werden dabei gewöhnlich durch repetitive Aneinanderreihung eines Grundmusters ausgehend von einem Referenzpunkt (auch Seed-Point genannt) erzeugt.

In vielen Anwendungen ist es auch notwendig, eine **Kombination** des neu gezeichneten Musters mit dem Hintergrund zu erzeugen. Hier gibt es viele Varianten, die oft auf logischen Verknüpfungen aufbauen: **AND**, **OR**, **XOR**. Das Mischen von Farben erfolgt meist durch Linearkombination der vorhandenen Hintergrundfarbe B mit der zu zeichnenden Vordergrundfarbe F : $P = t \cdot F + (1 - t) \cdot B$



Baryzentrische Koordinaten

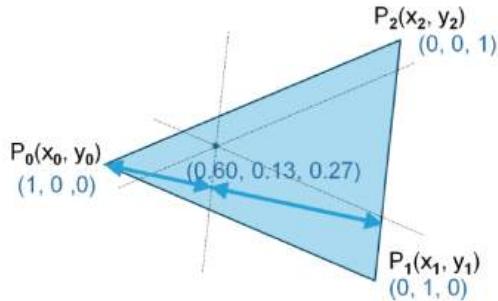
Baryzentrische Koordinaten sind eine Grundlage für die Interpolation von Pixeln in Dreiecken.

Dreiecke rasterisieren

Um Dreiecke zu füllen verwendet man oft **baryzentrische Koordinaten**. Jeder Punkt der Ebene wird dabei als gewichtetes Mittel der drei Eckpunkte des Dreiecks dargestellt:

$$P = \alpha P_0 + \beta P_1 + \gamma P_2$$

(α, β, γ) nennt man dann die baryzentrischen Koordinaten des Punktes P , wobei immer gilt: $\alpha + \beta + \gamma = 1$. Alle Punkte mit $(0 < \alpha < 1, 0 < \beta < 1, 0 < \gamma < 1)$ liegen innerhalb des Dreiecks; sobald einer dieser Werte negativ oder größer als 1 ist, liegt der Punkt außerhalb des Dreiecks.



Zum Füllen eines Dreiecks berechnet man für jedes Pixel einer (möglichst engen) Umgebung dessen baryzentrische Koordinaten und zeichnet alle für deren Mittelpunkt ($0 < \alpha < 1, 0 < \beta < 1, 0 < \gamma < 1$) gilt. Dabei kann man sehr einfach beliebige Eckpunktattribute (z.B. Farbe) in jedem Pixel mit (α, β, γ) gewichtet berechnen, dies entspricht einer linearen Interpolation dieser Werte.

Baryzentrische Koordinaten beschreiben einen Punkt P im Bezug auf die Eckpunkte eines Dreiecks P_0, P_1, P_2 :

$$P = \alpha P_0 + \beta P_1 + \gamma P_2$$

mit der Bedingung:

$$\alpha + \beta + \gamma = 1$$

Punkt liegt innerhalb des Dreiecks, wenn gilt:

$$0 < \alpha < 1, \quad 0 < \beta < 1, \quad 0 < \gamma < 1$$

Füllalgorithmus:

- Für jedes Pixel im Bounding-Box-Rechteck des Dreiecks:
 - Baryzentrische Koordinaten (α, β, γ) des Pixelmittelpunkts berechnen.
 - Liegt der Punkt **innerhalb** des Dreiecks? → **Pixel zeichnen**

Vorteil:

Mit α, β, γ lassen sich beliebige **Attribute (z. B. Farbe, Tiefe, Textur)** linear interpolieren:

$$Attribut(P) = \alpha \cdot Attribut(P_0) + \beta \cdot Attribut(P_1) + \gamma \cdot Attribut(P_2)$$

Beispiel der Baryzentrischen Koordinaten:

Angenommen, du hast ein Dreieck mit den Eckpunkten:

- A,
- B,
- C.

Dann sind die **baryzentrischen Koordinaten** für jeden Eckpunkt wie folgt:

Eckpunkt A:

→ (1, 0, 0)

Bedeutet: 100 % bei A, 0 % bei B, 0 % bei C → also exakt Punkt A.

Eckpunkt B:

→ (0, 1, 0)

Bedeutet: 100 % bei B → also exakt Punkt B.

Eckpunkt C:

→ (0, 0, 1)

Bedeutet: 100 % bei C → also exakt Punkt C.

Also:

- Die baryzentrischen Koordinaten (α, β, γ) eines Punkts P im Dreieck erfüllen immer:
 $P = \alpha A + \beta B + \gamma C$ mit $\alpha + \beta + \gamma = 1$
- Für Punkte **innerhalb** des Dreiecks sind **alle drei Werte positiv**.
- Für Punkte **auf einer Kante** ist **eine Koordinate = 0**.
- Für die **Eckpunkte** ist **zwei Koordinaten = 0**.

General Polygon Fill Algorithms

TU
WIEN

- *triangle rasterization*
- other polygons: what is inside?
- scan-line fill method
- flood fill method

barycentric
coordinates!

clean borders
between
adjacent triangles?

Werner Purgathofer 31

Berechnen der baryzentrischen Koordinaten

Sei $g_{12}(x, y) = a_{12}x + b_{12}y + c_{12} = 0$ die Trägergerade durch die Punkte P_1 und P_2 , dann berechnet sich α des Punktes $P(x_p, y_p)$ zu

$$\alpha = g_{12}(x_p, y_p) / g_{12}(x_0, y_0).$$

β und γ werden analog berechnet.

Um das doppelte Zeichnen der Kanten aneinander grenzender Dreiecke zu vermeiden, werden nur Pixel gezeichnet, deren Mittelpunkt innerhalb eines (exakten) Dreiecks liegen. Pixel genau auf einer Kante sind speziell zu behandeln, z.B. durch Regeln wie „Kanten unten und rechts werden gerendert, Kanten oben und links nicht“. Dadurch stellt man sicher, dass jedes Kantenpixel nur einmal behandelt wird.

Gegeben: ein Dreieck mit Eckpunkten $P_0 = (x_0, y_0)$, $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$

Und ein beliebiger Punkt $P = (x, y)$, z.B. der Mittelpunkt eines Pixels

Schritt 1: Trägergeraden der Dreiecksseiten

Zu jeder Seite des Dreiecks wird eine lineare Funktion $g_{ij}(x, y)$ bestimmt, die null ist, wenn der Punkt auf der Geraden zwischen P_i und P_j liegt:

$$g_{ij}(x, y) = a_{ij}x + b_{ij}y + c_{ij}$$

Diese Gerade ist die Kante gegenüber von Punkt P_k .

Zum Beispiel:

- $g_{12}(x, y)$: Gerade durch P_1 und $P_2 \rightarrow$ gegenüber von $P_0 \rightarrow$ liefert α
- $g_{20}(x, y)$: Gerade durch P_2 und $P_0 \rightarrow$ gegenüber von $P_1 \rightarrow$ liefert β
- $g_{01}(x, y)$: Gerade durch P_0 und $P_1 \rightarrow$ gegenüber von $P_2 \rightarrow$ liefert γ

Die Formel für $g_{ij}(x, y)$ ist:

$$g_{ij}(x, y) = (y_i - y_j)x + (x_j - x_i)y + (x_i y_j - x_j y_i)$$

Schritt 2: Baryzentrische Koordinate berechnen

Um z.B. α zu berechnen (also Anteil von P_0), setzt man den Punkt $P = (x, y)$ in g_{12} ein:

$$\alpha = \frac{g_{12}(x, y)}{g_{12}(x_0, y_0)}$$

Analog für β und γ :

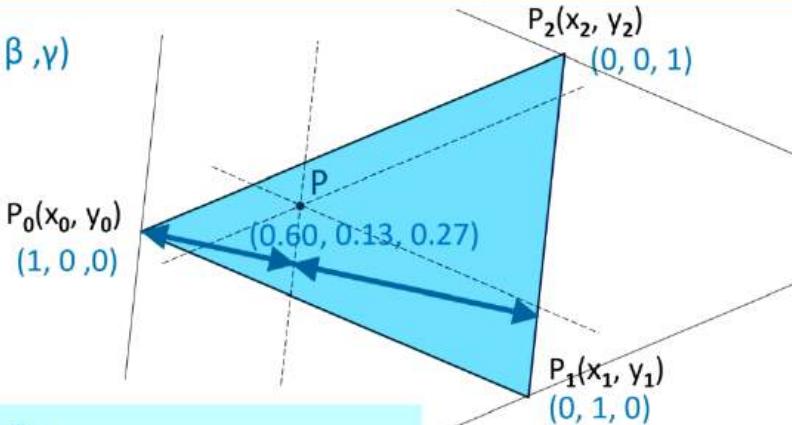
$$\beta = \frac{g_{20}(x, y)}{g_{20}(x_1, y_1)}, \quad \gamma = \frac{g_{01}(x, y)}{g_{01}(x_2, y_2)}$$

Der Nenner ist der jeweilige Wert, den die Gerade beim zugehörigen Eckpunkt liefert. Da die Punkte P_0, P_1, P_2 nicht auf ihren gegenüberliegenden Seiten** liegen, ist das kein Problem.

kompaktere Version hier: [5. FS - Rasterisierung > Barzentrische Koordinaten berechnen](#)

Triangles: Barycentric Coordinates

notation: (α, β, γ)



$$P = \alpha P_0 + \beta P_1 + \gamma P_2$$

$$\text{triangle} = \{P \mid \alpha + \beta + \gamma = 1, 0 < \alpha < 1, 0 < \beta < 1, 0 < \gamma < 1\}$$

Triangle Rasterization Algorithm



for all x

```

    for all y          /* use a bounding box! */
        {compute ( $\alpha, \beta, \gamma$ ) for (x,y) ;
         if ( $0 < \alpha < 1$ ) and ( $0 < \beta < 1$ ) and ( $0 < \gamma < 1$ )
         {
            draw pixel (x,y)
         }
    }
```

$$P = \alpha P_0 + \beta P_1 + \gamma P_2$$

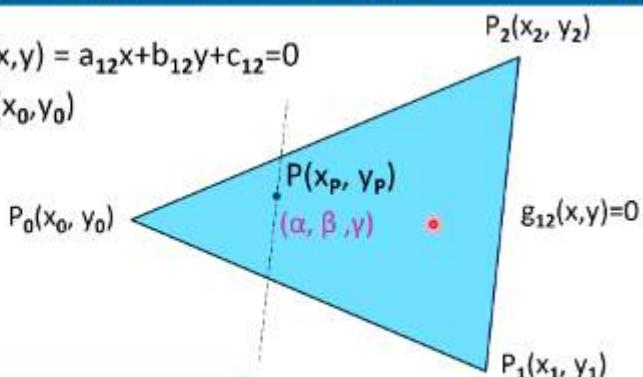
$$\text{triangle} = \{P \mid \alpha + \beta + \gamma = 1, 0 < \alpha < 1, 0 < \beta < 1, 0 < \gamma < 1\}$$

Computing (α, β, γ) for $P(x_p, y_p)$



line through P_1, P_2 : $g_{12}(x, y) = a_{12}x + b_{12}y + c_{12} = 0$

then $\alpha = g_{12}(x_p, y_p) / g_{12}(x_0, y_0)$



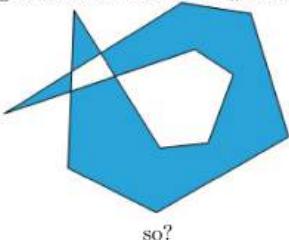
$$P = \alpha P_0 + \beta P_1 + \gamma P_2$$

$$\text{triangle} = \{P \mid \alpha + \beta + \gamma = 1, 0 < \alpha < 1, 0 < \beta < 1, 0 < \gamma < 1\}$$

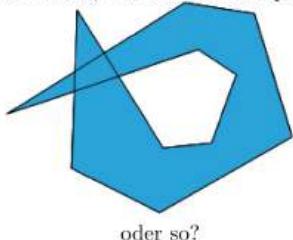


Was ist in einem Polygon innen?

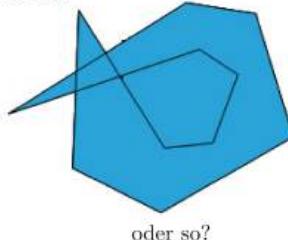
Bevor man mit dem Füllen von Flächen beginnt, muss man sich fragen, was denn zu füllen sei. Bei einer einfachen geschlossenen Kurve ist „innen“ leicht zu definieren, was aber bei komplizierteren Kurven?



so?

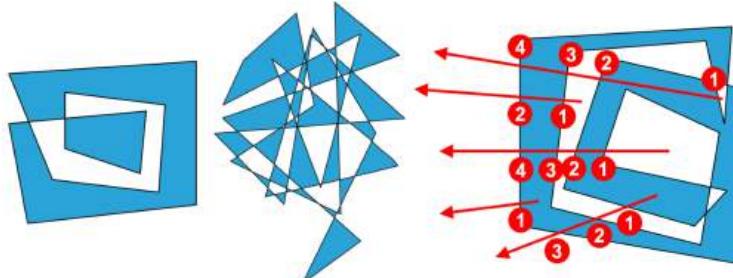


oder so?



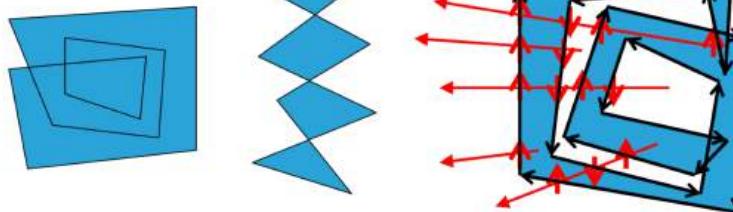
oder so?

Odd-Even-Rule Zieht man von einem Punkt aus einen beliebigen Halbstrahl, so ist der Punkt innerhalb, wenn die Zahl der Schnitte mit der Kurve ungerade ist, ansonsten ist der Punkt außerhalb (in Abb. oben links, sowie alle Bilder rechts). Jede Kante hat also eine Seite innen und die andere außen.



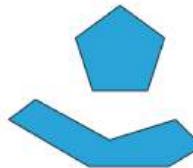
Nonzero-Winding-Number-Rule

Punkte sind außerhalb, wenn sich auf einem beliebigen Halbstrahl gleich viele im Uhrzeigersinn und gegen den Uhrzeigersinn verlaufende Kurvenkanten befinden, ansonsten innerhalb (in Abb. oben Mitte, sowie alle Bilder rechts).

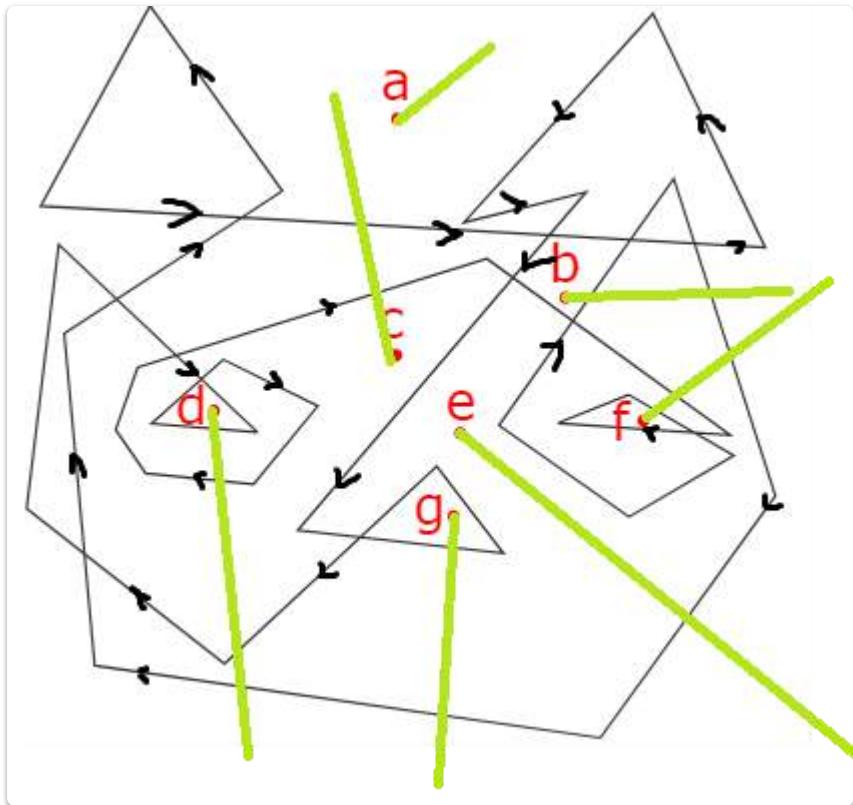


All-In-Rule Alles, was irgendwie umschlossen ist, ist innen. Wird selten verwendet, meist beim Pokern \odot (in Abb. oben rechts).

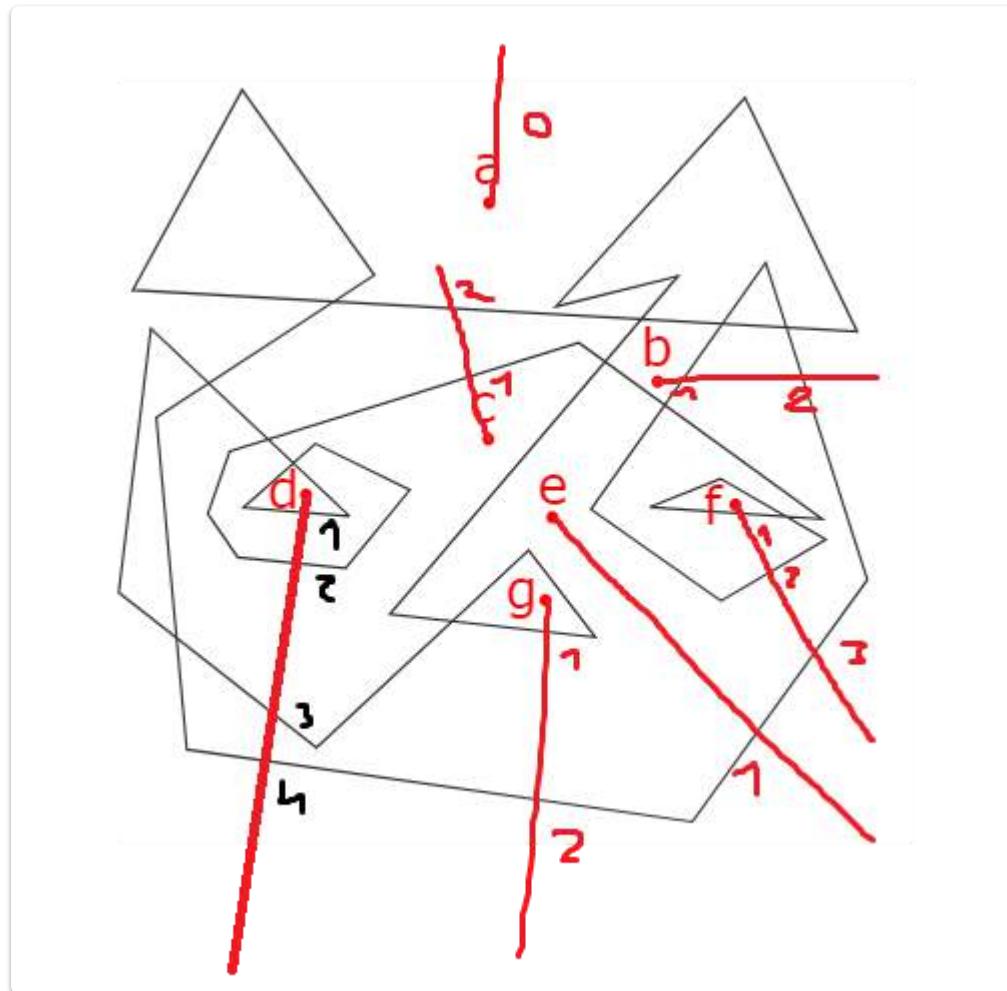
Ein Polygon heißt *konvex* wenn alle inneren Winkel kleiner als 180° sind (oberes Bild), andernfalls *konkav* (unteres Bild). Da konvexe Polygone viel weniger Sonderfälle erzeugen, sind viele Algorithmen für konvexe Polygone ausgelegt (oft sogar nur für Dreiecke). Daher braucht man auch Methoden um konkave Polygone in mehrere konvexe Polygone zu zerlegen (oft in Dreiecke).



35



Hier muss man Pfeile einzeichnen, wie man das Polynom als Oneliner zeichnen könnte, dann Linien von den Punkten nach draußen machen und schauen ob gleich viele in die eine Richtung wie in die andere Richtung gehen... Wenn nein dann innerhalb.



36.

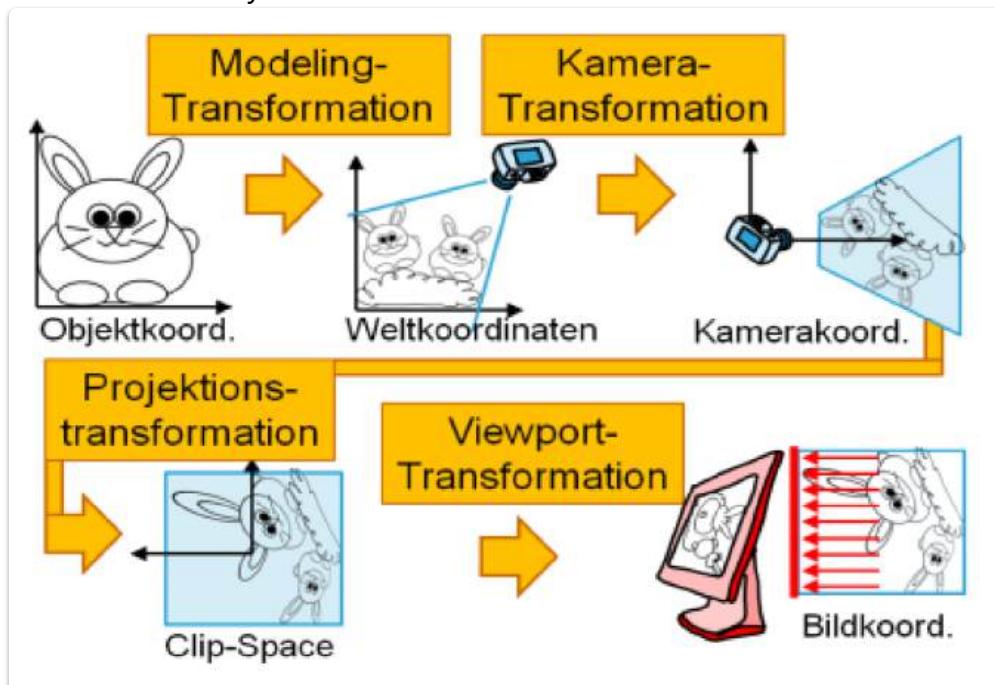
Hier muss man zählen wie viele Kanten man überqueren muss um eine Linie ganz nach

außen ziehen zu können. Wenn diese Anzahl ungerade ist, dann liegt der Punkt in dem Polygon drinnen. Wenn nicht außerhalb

6. Viewing

Viewing in der Graphik-Pipeline

- Szenenmodellierung erfolgt in **Weltkoordinaten**
- Objekte werden aus **lokalen Koordinatensystemen** über **geometrische Transformationen (Matrizen)** in Weltkoordinaten überführt
- Nach Festlegung der **Kameraparameter**:
 - Transformation der Koordinaten in **Kamerakoordinaten**
 - Anschließende **Projektion** der Kamerakoordinaten
- Ergebnis der Projektion liegt in einem **normierten Würfel** (meist mit Seitenlänge 2)
- Danach erfolgt die **Viewport-Transformation**:
 - Überführung der normierten Koordinaten in **Gerätekordinaten** des Ausgabemediums
- In der Geometrie existieren verschiedene Projektionen
- In der **Computergraphik** sind hauptsächlich zwei Projektionen relevant:
 - **Parallelprojektion**
 - **Perspektivische Projektion**
- Zuerst wird die **Parallelprojektion** angenommen
- Danach wird die **Integration der perspektivischen Projektion** in die Pipeline betrachtet
- Vorgehensweise erfolgt **von hinten nach vorne** (vom Endergebnis zurück), da dies einfacher zu analysieren ist

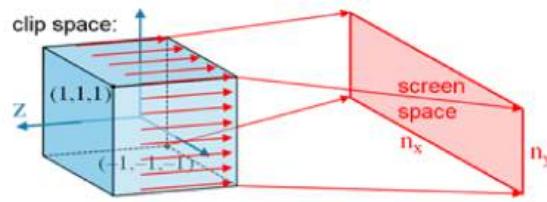


Viewport-Transformation

Das ist das was eben vom Clip-Space in die Bildkoordinaten des Ausgabegeräts umwandelt

Wir nehmen also an, dass die Szene bereits im Clip-Space vorhanden ist, d.h. alle relevanten Koordinaten befinden sich in einem achsenparallelen Würfel der Seitenlänge 2 mit Mittelpunkt (0,0,0). Wir wollen eine orthographische Abbildung (parallele Normalprojektion) mit Blickrichtung -z auf einen Bildschirm mit Abmessungen $n_x \times n_y$ (Pixel) durchführen. Es müssen also alle Punkte $(-1, -1, z)$ auf $(0, 0)$ abgebildet werden und alle Punkte $(1, 1, z)$ auf $(n_x \times n_y)$. Diese lineare Abbildung wird durch die Matrix Mvp bewerkstelligt:

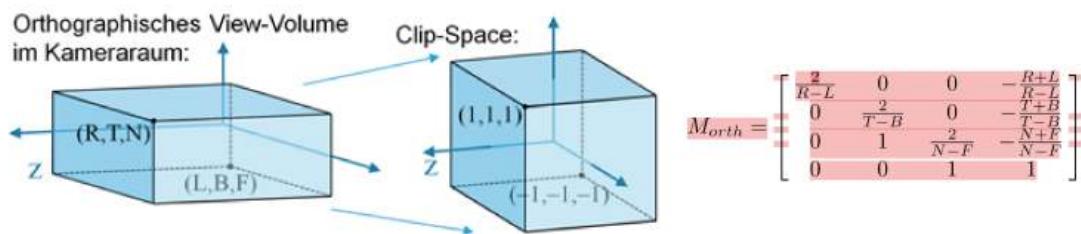
$$\begin{bmatrix} x_{\text{screen}} \\ y_{\text{screen}} \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



Deren Richtigkeit kann sofort nachgewiesen werden, indem man die Eckpunkte einsetzt. Die Matrix weist aber in den roten Zahlen noch eine Besonderheit auf: die z-Werte werden erhalten! Dies ist im Moment ohne Belang, wird aber bei späteren Schritten (vor allem bei der Berechnung der Sichtbarkeit) noch von großem Wert sein.

Projektionstransformation:

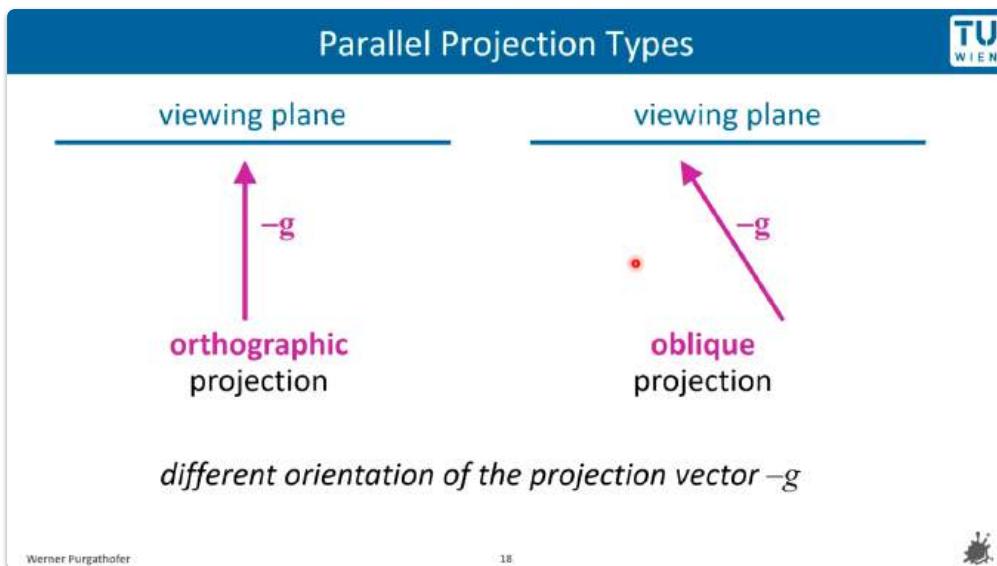
- Annahme: Orthographische Projektion
- Ziel: Vereinfachte Transformation durch achsenparallelen Quader
- Gegeben: Quader mit Grenzen:
 - L (Left)
 - R (Right)
 - B (Bottom)
 - T (Top)
 - N (Near)
 - F (Far)
- Transformation dieses Quaders in den normierten Würfel $[-1, 1]^3$
- Dabei gilt:
 - Punkt $(L, B, F) \rightarrow (-1, -1, -1)$
 - Punkt $(R, T, N) \rightarrow (1, 1, 1)$
- Umsetzung erfolgt über eine Transformationsmatrix



Bemerkung: Eine Parallelprojektion kann auch schräg auf eine Abbildungsebene erfolgen (zum Beispiel beim Schattenwurf), diese Variante wird hier nicht berücksichtigt.

Warum ist das wichtig?

- **Anzeige auf 2D-Bildschirmen:** Unsere Monitore und Bildschirme sind zweidimensional. Um 3D-Grafiken darauf darstellen zu können, müssen die 3D-Objekte in 2D projiziert werden.



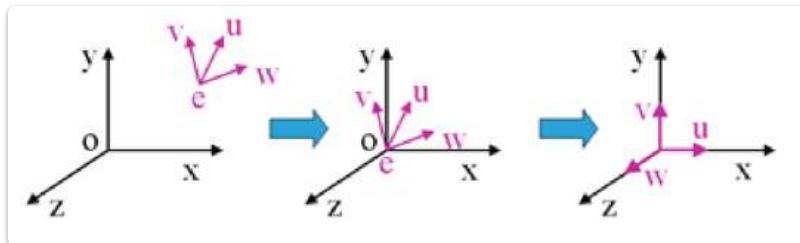
Kamera Transformation

- Beim Festlegen der Kamerawerte bestehen mehrere **Freiheitsgrade**:
 1. **Position der Kamera** im Raum
 2. **Blickrichtung** von dieser Position aus
 3. **Orientierung** der Kamera (Definition von „oben“)
 4. **Größe des Bildausschnittes** (analog zur Brennweite oder Zoomfaktor)



- Aus den ersten drei Werten ergibt sich das **Viewing-Koordinatensystem** mit den Achsen u, v, w
- Eigenschaften des Viewing-Koordinatensystems:
 - Die **uv-Ebene** ist **normal** zur Hauptblickrichtung
 - Die Blickrichtung verläuft entlang der **negativen w-Achse**
- In **Animationen** wird die Kameradefinition oft automatisch aus Bedingungen berechnet:
 - z.B. Kamerafahrt um ein Objekt
 - z.B. Kamerasteuerung in Flugsimulationen
 - Ziel: **Unkomplizierte Erzeugung gewünschter Effekte**
- Umwandlung von **Weltkoordinaten** in **Viewingkoordinaten** erfolgt durch eine Kette von **einfachen Transformationen**:

- **Translation**, um Koordinatenursprünge aufeinander abzustimmen
- **Drei Rotationen**, um Koordinatenachsen zur Deckung zu bringen:
 - Zwei Drehungen zur Ausrichtung der ersten Achse
 - Eine weitere Drehung für die zweite Achse
 - Die dritte Achse ergibt sich automatisch korrekt
- Zusammensetzung dieser Transformationen in eine **Transformationsmatrix**:
 $M_{WC \rightarrow VC} = R_z \cdot R_y \cdot R_x \cdot T$
- Zur vollständigen **Projektionsbeschreibung** werden zusätzlich die **Grenzen des darzustellenden Bereichs** benötigt
- Mehr zu Transformations findet man hier: [3. Transformationen](#)



Ausgehend von der Kameraposition geht man prinzipiell folgendermaßen vor, um das Viewing-Koordinatensystem festzulegen:

1. Wahl einer Kameraposition (auch Augpunkt oder Viewing-Punkt genannt).
2. Wahl einer Blickrichtung, die negative Blickrichtung ergibt die w-Achse.
3. Wahl einer Richtung t „nach oben“; aus dieser lassen sich dann die u- und v-Achsen berechnen.
4. Da die Abbildungsebene normal auf die Blickrichtung liegt, ergibt das Vektorprodukt $t \times w$ die Richtung der u-Achse.
5. Berechnung der v-Achse als Vektorprodukt der w- und u-Achsen: $v = w \times u$.
6. Die Wahl von minimalen und maximalen u-, v- und w-Werten zur Eingrenzung des Ausschnittes der Szene, der abgebildet wird: L(eft), R(ight), B(ottom), T(op), N(ear), F(ar).

Hier nochmal die Formeln:

e ... eye position

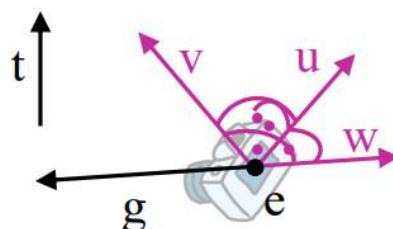
g ... gaze direction (positive w-axis points to the viewer)

t ... view-up vector

$$w = -\frac{g}{|g|}$$

$$u = \frac{t \times w}{|t \times w|}$$

$$v = w \times u$$



Orthographisches Viewing

Für die orthographische Projektion (**Kamera bildet parallel ab**) haben wir nun alle Schritte durch Matrizen beschrieben, diese können wir wie bei den geometrischen Transformationen zu einer einzigen Matrix zusammensetzen (**multiplizieren**), die dann die gesamte Viewing-Transformation durchführt:

$$\begin{bmatrix} x_{\text{screen}} \\ y_{\text{screen}} \\ z \\ 1 \end{bmatrix} = (\mathbf{M}_{\text{vp}} * \mathbf{M}_{\text{orth}} * \mathbf{M}_{\text{cam}}) * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Man beachte: die rechteste Matrix wird zuerst mit dem Punkt (x,y,z) multipliziert. Wenn man das Assoziativgesetz anwendet, dann kann man aber auch zuerst die 3 Matrizen miteinander multiplizieren, und kann damit alle Punkte nur mit dieser einen Ergebnismatrix direkt von Weltkoordinaten in Gerätekordinaten transformieren!

Viewing: Camera + Projection + Viewport



$$\begin{bmatrix} x_{\text{screen}} \\ y_{\text{screen}} \\ z \\ 1 \end{bmatrix} = \mathbf{M}_{\text{vp}} \cdot \mathbf{M}_{\text{orth}} \cdot \mathbf{M}_{\text{cam}} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

pixels on the screen

world coordinates

viewport transformation

projection transformation

camera transformation

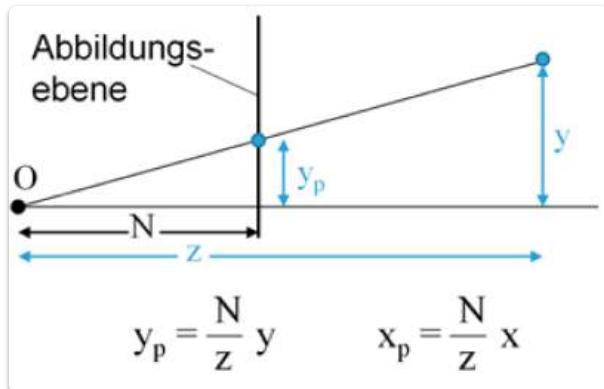
Perspektive

- Perspektivische Projektion ist **keine affine Transformation**:
 - Affine Eigenschaften wie Parallelität bleiben **nicht** erhalten
 - Kann **nicht** mit einer 3×3 -Matrix dargestellt werden
- Lösung: Verwendung von **homogenen Koordinaten**
 - Einziger Fall, in dem die **homogene Komponente $h \neq 1$**
 - Erfordert einen **Divisionsschritt** am Ende der Transformation: Koordinaten werden durch h geteilt
- Grundlagen der perspektivischen Transformation:
 - O : **Projektionszentrum**
 - **Blickrichtung**: entlang der **negativen z -Achse**
 - **Abbildungsebene**: normal zur z -Achse im Abstand N (Near)
- Abbildung eines Punktes (x, y, z) auf die Ebene:

$$(x, y, z) \rightarrow \left(\frac{x \cdot N}{z}, \frac{y \cdot N}{z}, N \right)$$

- Das lässt sich durch eine Matrix P darstellen:

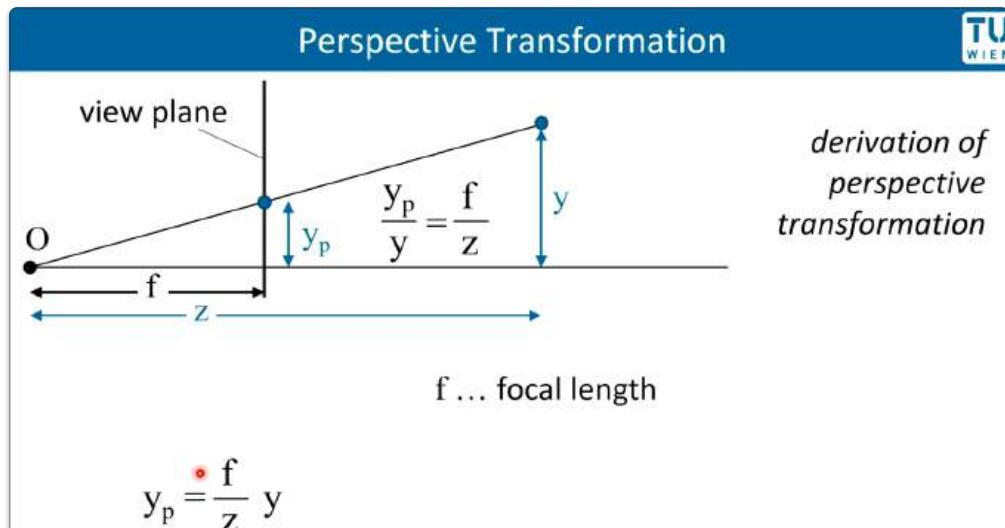
$$P = \begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 1 & N + F & -F * N \\ 0 & 0 & N & 1 \end{bmatrix}$$



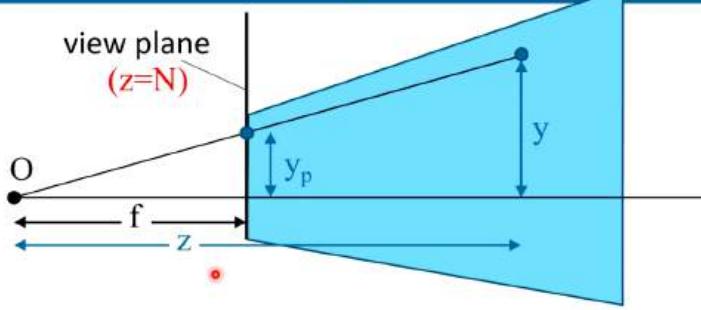
- Ein Punkt $(x, y, z, 1)$ wird mit der **Projektionsmatrix P** multipliziert
- Ergebnis der Multiplikation: $(x \cdot N, y \cdot N, z \cdot (n + F) - F \cdot N, z)$
- Danach erfolgt das **Homogenisieren** (Normalisieren): Division durch die letzte Komponente z
- Ergebnis nach Division:

$$\left(\frac{x \cdot N}{z}, \frac{y \cdot N}{z}, (N + F) - \frac{F \cdot N}{z}, 1 \right)$$

Hier der Inhalt der Folien zu dem:



Perspective Transformation



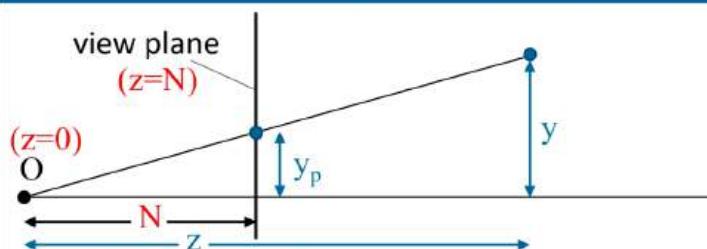
derivation of perspective transformation

$$y_p = \frac{f}{z} y$$

Werner Purgathofer

38

Perspective Transformation



derivation of perspective transformation

$$x_p = \frac{N}{z} x$$

analogous:

$$y_p = \frac{N}{z} y$$

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

Werner Purgathofer

38

Perspective Transformation

$$P = \begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & N+F & -F \cdot N \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \cdot N \\ y \cdot N \\ z \cdot (N+F) - F \cdot N \\ z \end{bmatrix}$$

derivation of perspective transformation

homogenization: divide by z

$$x_p = \frac{N}{z} x$$

$$y_p = \frac{N}{z} y$$

$$\leadsto \begin{bmatrix} x \cdot N/z \\ y \cdot N/z \\ (N+F) - F \cdot N/z \\ 1 \end{bmatrix}$$

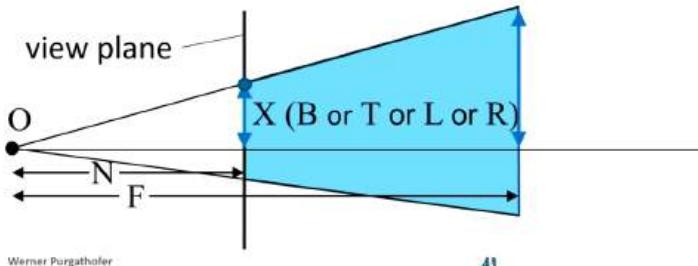
Werner Purgathofer

39

mit homogenization ist gemeint, dass man alles durch z dividiert, sodass die z-Koordinate = 1 ist.

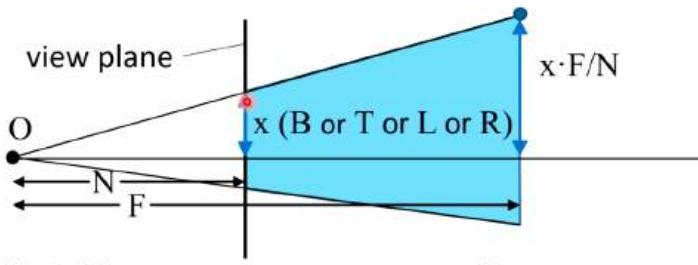
Example: (Right) Top Near Corner

$$\begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & N+F & -F \cdot N \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} R \\ T \\ N \\ 1 \end{bmatrix} = \begin{bmatrix} R \cdot N \\ T \cdot N \\ N \cdot (N+F) - F \cdot N \\ N \end{bmatrix} \rightsquigarrow \begin{bmatrix} R \\ T \\ N \\ 1 \end{bmatrix}$$



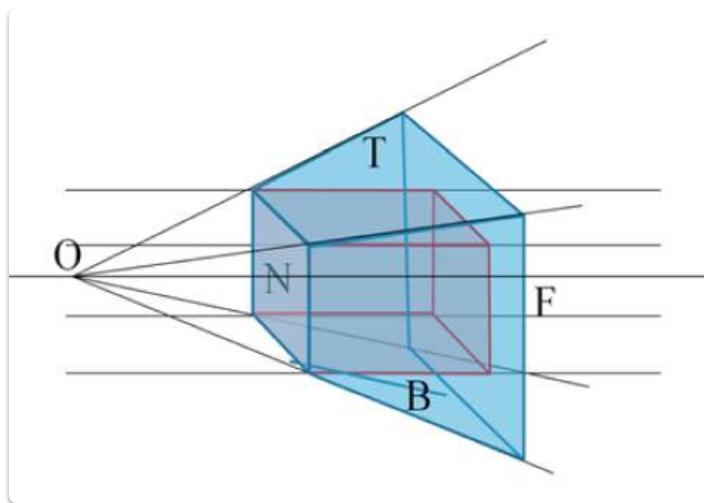
Example: (Left) Top Far Corner

$$\begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & N+F & -F \cdot N \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} L \cdot F/N \\ T \cdot F/N \\ F \\ 1 \end{bmatrix} = \begin{bmatrix} L \cdot F \\ T \cdot F \\ F \cdot (N+F) - F \cdot N \\ F \end{bmatrix} \rightsquigarrow \begin{bmatrix} L \\ T \\ F \\ 1 \end{bmatrix}$$



- Die perspektivische Projektion führt zu einer **Verzerrung des Szenebereichs**:
 - Dieser Bereich wird als „**View Frustum**“ bezeichnet
 - Das View Frustum ist ein **Pyramidenstumpf**, der auf einen **achsparallelen Quader** abgebildet wird
 - In diesem Quader liefert die **orthographische Projektion** dasselbe Bild wie die **perspektivische Projektion** im View Frustum
- Nach der Verzerrung kann die bereits erarbeitete **Parallelprojektion** angewendet werden, um die perspektivische Matrix M_{per} zu berechnen
- Alternative Methode:
 - **Einsetzen** der Projektionsmatrix P an der richtigen Stelle in der Gesamtviewingberechnung
 - Dadurch wird eine **Gesamtmatrix** erzeugt, die die Transformation von Modellkoordinaten (x, y, z) zu Gerätekordinaten (x_{screen}, y_{screen}) in einem Schritt ausführt

$$\begin{bmatrix} x_{screen} \\ y_{screen} \\ z \\ 1 \end{bmatrix} = M_{vp} * \underbrace{(M_{orth} * P * M_{cam} * M_{mod})}_{M_{per}} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



Clip Space vs. NDC Space

TU
WIEN

- Clip Space: $[-w, w] \times [-w, w] \times [-w, w]$, $w > 0$
- Normalized Device Coordinates (NDC): $[-1,1] \times [-1,1] \times [-1,1]$

Clip Space	NDC Space
Constraints $x_{\min} = -w$ $x_{\max} = w$ $y_{\min} = -w$ $y_{\max} = w$ $z_{\min} = -w$ $z_{\max} = w$ $w > 0$	$(x_{\min}/w, y_{\min}/w, z_{\min}/w)$ $(x_{\max}/w, y_{\max}/w, z_{\max}/w)$ $(x_{\min}/w, y_{\max}/w, z_{\min}/w)$ $(x_{\max}/w, y_{\max}/w, z_{\max}/w)$ $(x_{\min}/w, y_{\min}/w, z_{\max}/w)$ $(x_{\max}/w, y_{\min}/w, z_{\max}/w)$ $(x_{\min}/w, y_{\max}/w, z_{\max}/w)$ $(x_{\max}/w, y_{\max}/w, z_{\max}/w)$
Pre-perspective divide puts the region surviving clipping within $-w \leq x \leq w, -w \leq y \leq w, -w \leq z \leq w$	
Post-perspective divide puts the region surviving clipping within the $[-1,+1]^3$	

Image Source: Mark Kilgard, University of Texas, CS 354 Graphics Math (2012)

- **Homogenisierung:**

- Wenn eine **perspektivische Abbildung** beteiligt ist, muss das Ergebnis am Ende durch die homogene Komponente z' dividiert werden.
- Im **Clipraum** wird nicht nur das **Clipping** durchgeführt, sondern auch die **Homogenisierung**.
- Nach der Homogenisierung wird die **Viewport-Matrix** als letzter Schritt angewendet.

- **Wichtige Eigenschaften der Projektionstransformation:**

1. **Gerade Strecken bleiben gerade Strecken:**

- Um eine Strecke (z.B. Seite eines Polygons) abzubilden, reicht es, nur die beiden Endpunkte zu transformieren.

2. **Relative Ordnung der z-Werte bleibt erhalten:**

- Der Abstand der Punkte von der Kamera bleibt relativ zueinander erhalten, jedoch nicht die **Abstandswerte selbst**.
- Diese Eigenschaft ist wichtig für die **Sichtbarkeitsberechnung**:

$$z_1, z_2, N, F < 0$$

$$z_1 < z_2$$

$$\frac{1}{z_1} > \frac{1}{z_2}$$

$$| * (-F * N)(< 0)$$

$$-F * \frac{N}{z_1} < -F * \frac{N}{z_2}$$

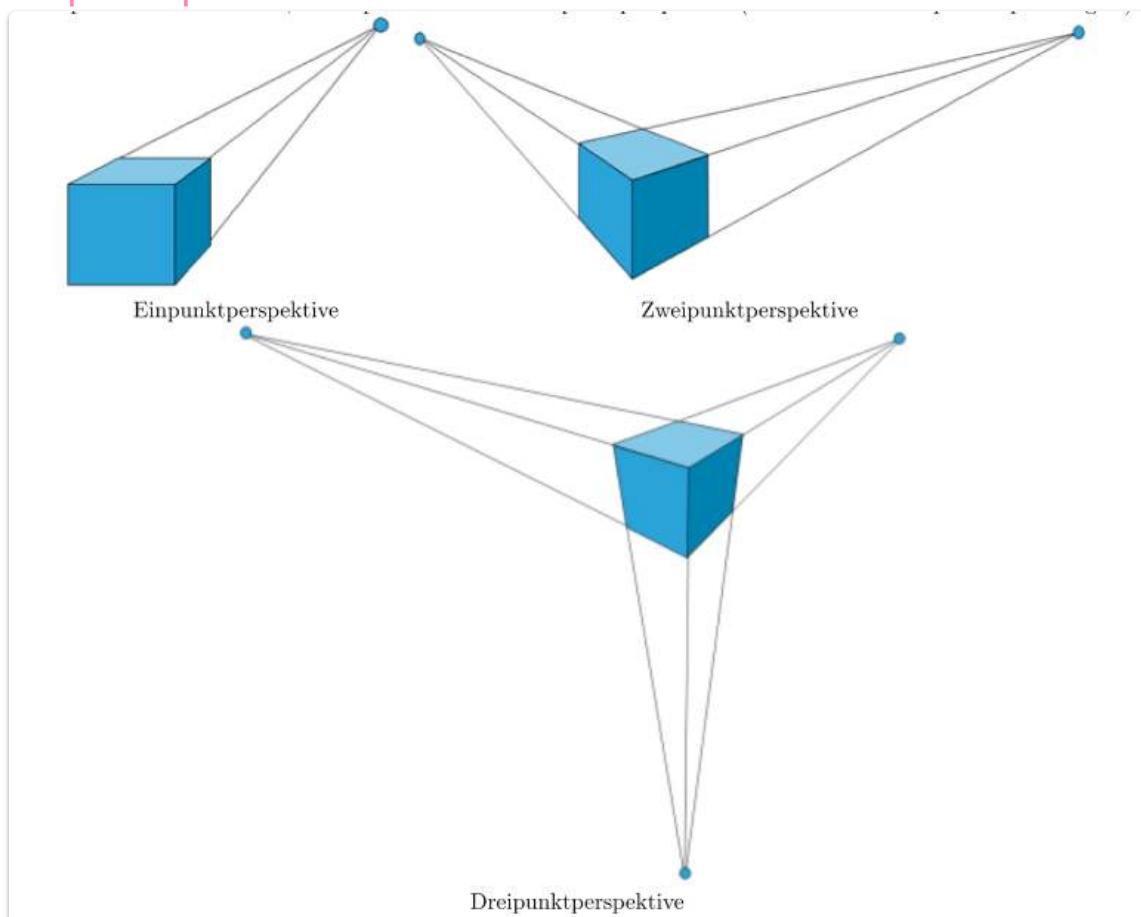
$$| + (N + F)$$

$$(N + F) - F * \frac{N}{z_1} < (N + F) - F * \frac{N}{z_2}$$

Fluchtpunkte:

Anzahl der Hauptfluchtpunkte hängt von der Lage der **Bildebene** zum **Koordinatensystem** ab:

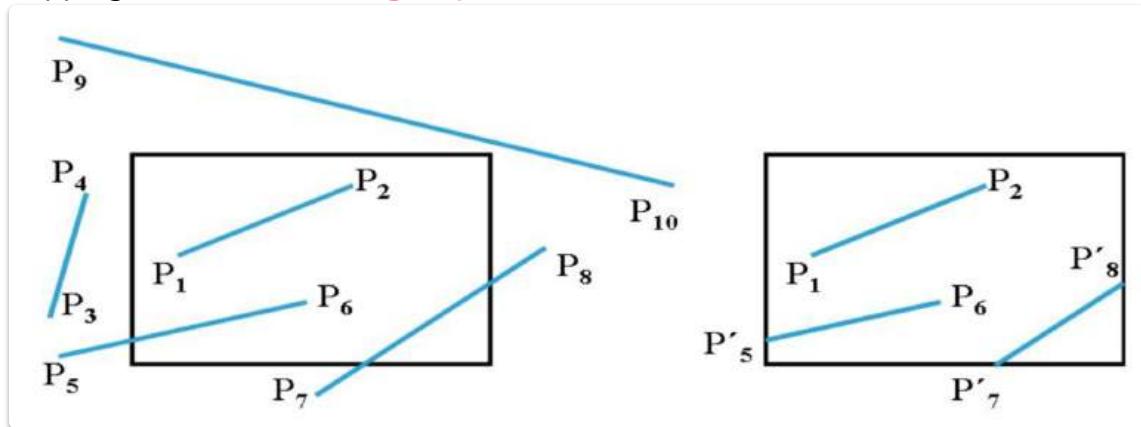
1. **Einpunktperspektive**: Zwei Achsen sind parallel zur Bildebene.
2. **Zweipunktperspektive**: Eine Achse ist parallel zur Bildebene.
3. **Dreipunktperspektive**: Keine Achse ist parallel zur Bildebene, es gibt **3 Hauptfluchtpunkte**.



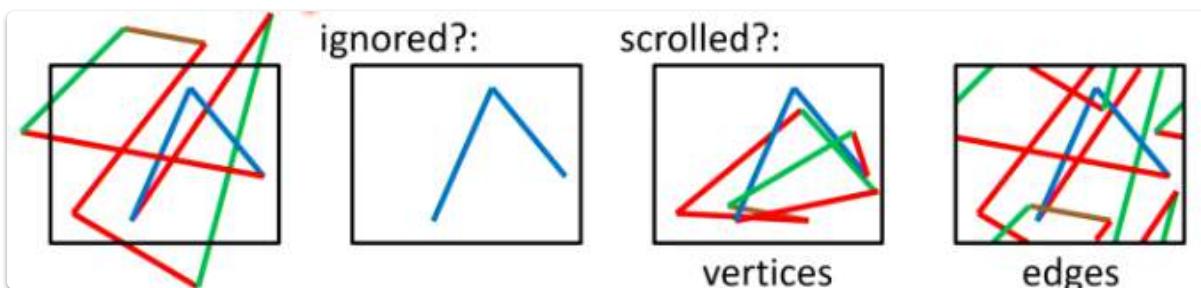
7. Clipping und Antialiasing

Line Clipping:

- Clipping bezeichnet das **Abschneiden von Bildteilen**, die außerhalb des Darstellungsfensters liegen.
- Clipping wird durchgeführt, um **unnötige nachfolgende Umformungen** von nicht sichtbaren Teilen zu vermeiden:
 1. Clipping in Weltkoordinaten:
 - Analytische Berechnung zum frühestmöglichen Zeitpunkt.
 2. Clipping in Clipkoordinaten:
 - Analytische Berechnung an **achsenparallelen Grenzen** (einfacher).
 3. Clipping bei der Rasterkonversion:
 - Clipping erfolgt innerhalb des Algorithmus, der ein **Grafikprimitiv** in **Punkte umwandelt**.
- Clipping ist eine sehr **häufige Operation**, daher muss es **einfach und schnell** sein.



Wichtige Fragen: Was soll abgeschnitten werden?



Clippen von Linien: Cohen-Sutherland-Verfahren

- Algorithmen zum Clippen von Linien nutzen die Tatsache aus, dass jede Linie in einem rechteckigen Fenster höchstens **einen sichtbaren Teil** besitzt.
- Wichtige Grundprinzipien der **Effizienz**:
 1. Häufige einfache Fälle früh eliminieren.

2. Teure Operationen wie Schnittpunkt-Berechnungen vermeiden.

- Ein einfaches Linieneckling könnte so aussehen:

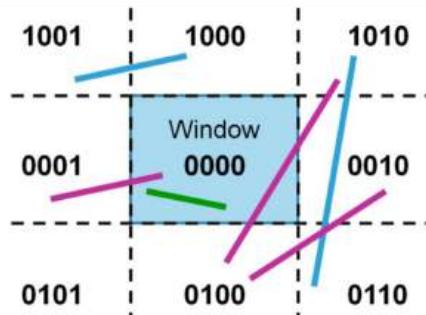
```

1 for endpoints (x0,y0), (xend,yend)
2 intersect parametric representation
3   x = x0 + u * (xend - x0)
4   y = y0 + u * (yend-y0)
5 with window borders:
6   intersection <= 0 < u < 1

```

Der Cohen-Sutherland-Algorithmus klassifiziert zuerst die Endpunkte einer Linie hinsichtlich ihrer Lage zum Clippingfenster: oben, unten, links, rechts, und codiert diese Information in 4 Bit. Nun kann man schnell überprüfen:

1. OR der beiden Codes = 0000 \Rightarrow Linie ganz sichtbar
2. AND der beiden Codes \neq 0000 \Rightarrow Linie ganz unsichtbar
3. andernfalls mit einer relevanten Fensterkante schneiden, und den weggescnittenen Punkt durch den Schnittpunkt ersetzen.
GOTO 1.



- Schnittpunktberechnungen mit vertikalen Fensterkanten:

- Für die linke Kante:

$$y = y_0 + m(x_{wmin} - x_0) \quad y = y_0 + m(x_{wmin} - x_0)$$

- Für die rechte Kante:

$$y = y_0 + m(x_{wmax} - x_0) \quad y = y_0 + m(x_{wmax} - x_0)$$

- Schnittpunktberechnungen mit horizontalen Fensterkanten:

- Für die untere Kante:

$$x = x_0 + \frac{(y_{wmin} - y_0)}{m} x = x_0 + m(y_{wmin} - y_0)$$

- Für die obere Kante:

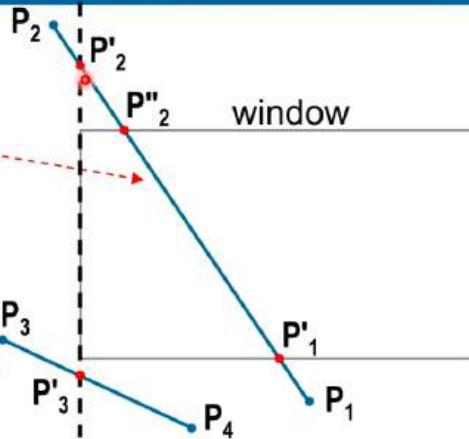
$$x = x_0 + \frac{(y_{wmax} - y_0)}{m} x = x_0 + m(y_{wmax} - y_0)$$

- Punkte, die genau auf den Fensterkanten liegen, gelten als innerhalb des Fensters.
- Es sind höchstens 4 Schleifendurchläufe erforderlich, da es höchstens 4 Schnittpunkte gibt.
- Effizienz:** Schnittpunktberechnungen werden nur durchgeführt, wenn sie wirklich notwendig sind.
- Clipping von Kreisen:**
 - Ähnliches Verfahren wie für Linien.
 - Kreise können beim Clipping in mehrere Teile zerfallen.

Cohen-Sutherland Line Clipping

passes through clipping window

intersects boundaries without entering clipping window



vertical: $y = y_0 + m(xw_{\min} - x_0)$,

horizontal: $x = x_0 + (yw_{\min} - y_0)/m$,

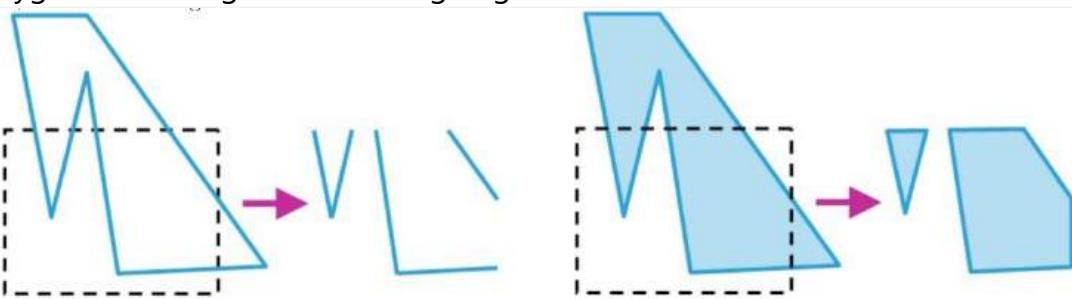
Werner Purgathofer

13



Polygon Clipping:

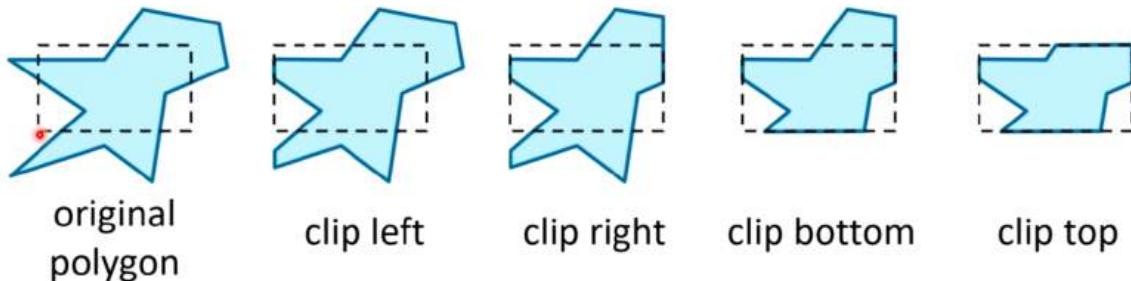
- **Polygon-Clipping** muss sicherstellen, dass nach dem Clipping ein **gültiges Polygon** entsteht, auch wenn der Clipping-Vorgang mehrere Teile erzeugt.
- Beispiel:
 - **Linien-Clipping-Algorithmus:**
 - Das Ergebnis zeigt ein **unvollständiges Polygon**, bei dem nicht mehr erkennbar ist, was innen und was außen liegt.
 - **Korrekte Polygon-Clipping-Verfahren:**
 - Das Polygon zerfällt in **mehrere Teile**, die alle korrekt **gefüllt** werden können.
- **Wichtig:** Auch wenn mehrere Teile entstehen, müssen alle Teile des resultierenden Polygons **korrekt** gefüllt und als gültige, sichtbare Bereiche behandelt werden.



Bei einfachem Linien Clipping könnten Linien zurückkommen deshalb gibt es extra Verfahren für das Polynomclipping:

Sutherland-Hodgman Polygon Clipping

processing polygon boundary as a whole against each window edge
 → output: list of vertices



clipping a polygon against successive window boundaries

Man zerlegt das hier in **4 Schritten** die meist rekursiv aufgerufen werden

Clippen von Dreiecken:

- **Geometrische Daten** bestehen in der Praxis häufig nur aus **Dreiecken**. Der **Renderingprozess** hat dabei kein Wissen mehr über den Zusammenhang der Dreiecke und behandelt diese als eine „**Triangle Soup**“ (Dreiecks-Suppe).
- **Wichtig:**
 - Beim Clipping von Dreiecken muss immer darauf geachtet werden, dass nur Dreiecke entstehen – keine anderen Primitives.
- Beim **Clippen eines Dreiecks** gegen eine Kante gibt es vier mögliche Fälle:
 1. In einigen Fällen kann auch ein **Viereck** entstehen.

Clipping of Triangles

often b-reps are “triangle soups”
 clipping a triangle → triangle(s)

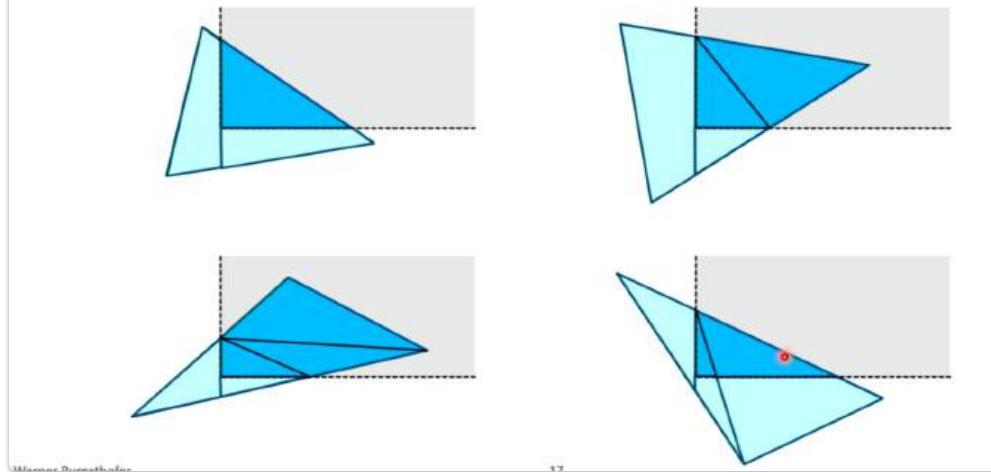
4 possible cases:

- (1) inside
- (2) outside
- (3) triangle
- (4) quadrilateral → 2 triangles

(inside)

2. Das Viereck muss sofort in **zwei Dreiecke** zerteilt werden, um die Weiterverarbeitung zu ermöglichen.

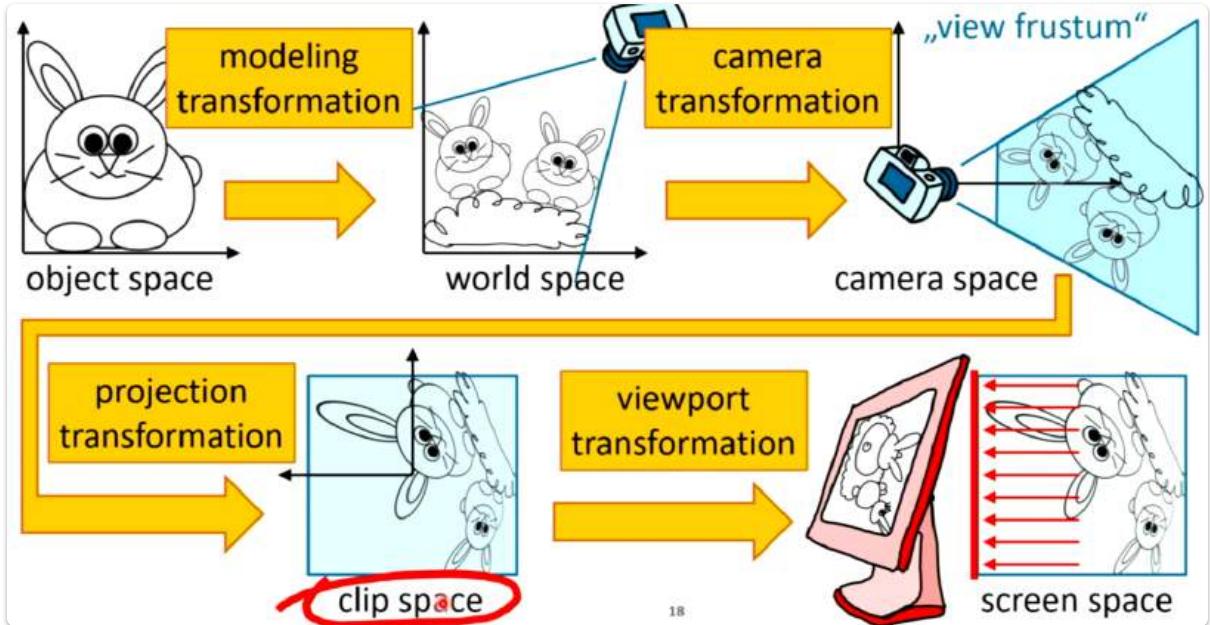
corner cases need no extra handling!



- An den **Ecken des Clip-Fensters** kann es vorkommen, dass mehr Dreiecke erzeugt werden, als tatsächlich notwendig wären (siehe Beispiel links), aber dies wird durch die **Einfachheit des Algorithmus** mehr als kompensiert.

Clipping in Clipkoordinaten:

- Clip-Space:**
 - Die Begrenzungsflächen des **View-Frustums** sind achsenparallel (d.h., die Grenzen sind bei $x = \pm 1, y = \pm 1, z = \pm 1$).
 - Dies vereinfacht die Feststellung, ob ein Punkt **innerhalb oder außerhalb** des Frustums liegt, da es nur einen **einfachen Vergleich** zwischen zwei Zahlen erfordert.



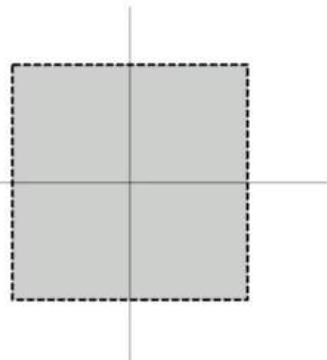
- Clipping vor der Homogenisierung:**
 - Um zu vermeiden, dass Punkte hinter dem Kamerapunkt projiziert werden, wird das Clipping schon **vor der Homogenisierung** der Punktkoordinaten durchgeführt.
 - Dabei wird an den **Ebenen $x = \pm h, y = \pm h, z = \pm h$** geclipt (was ebenso einfach ist).
- Vorteile:**
 - Punkte, die hinter dem Kamerapunkt liegen, werden **nicht projiziert**.

2. **Ersparnis der Homogenisierungsdivision** für Punkte, die außerhalb des Clipbereiches liegen.

clipping against $x = \pm 1, y = \pm 1, z = \pm 1$

(x, y, z) inside?

→ only compare one value per border!



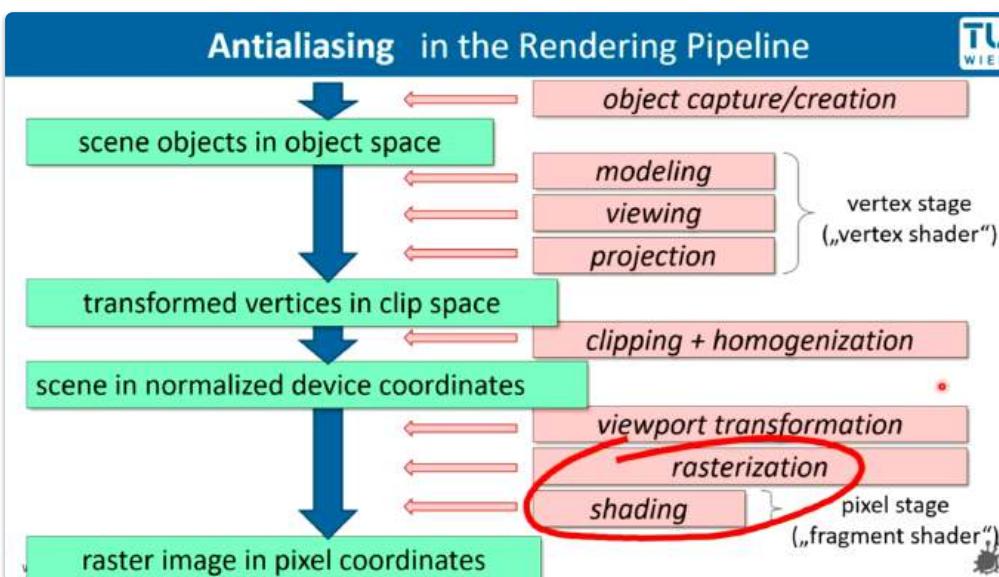
is done *before* homogenization:

clipping against $x = \pm h, y = \pm h, z = \pm h$

clips points that are behind the camera!

reduces homogenization divisions

Aliasing und Antialiasing:



Aliasing

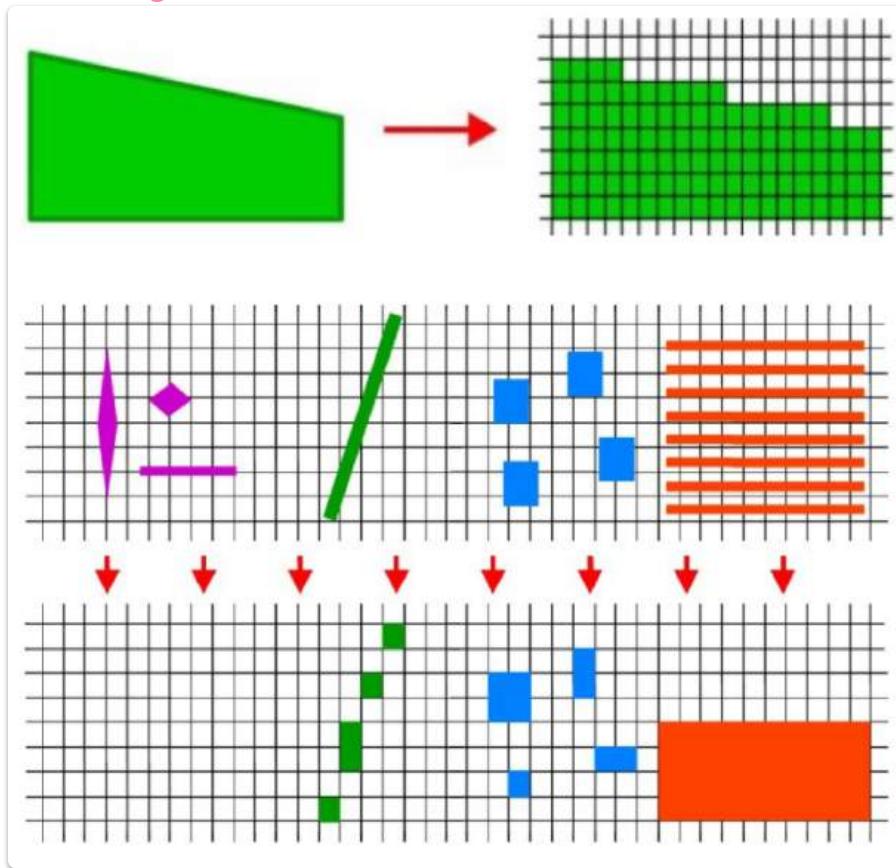
- **Aliasing-Effekte** sind Fehler, die bei der **Umwandlung (Diskretisierung)** von analogen in digitale Informationen auftreten.
- **Ursachen für sichtbare Aliasing-Effekte:**
 1. Zu geringe **Auflösung**
 2. Zu wenige **verfügbare Farben**
 3. Zu wenige **Bilder pro Sekunde** (Frames per second)
 4. **Geometrische Fehler**
 5. **Numerische Fehler**
- Ein Beispiel für **Aliasing**: Ein Pixel hat nur einen Wert, stellt aber tatsächlich eine **kleine Fläche** dar, was zu Unschönheiten in Rasterbildern führen kann.

Antialiasing

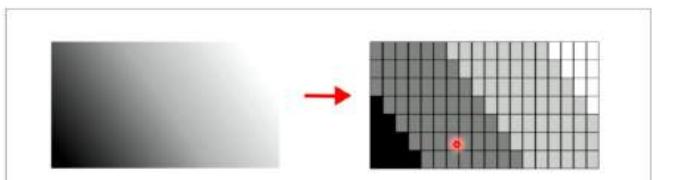
- **Antialiasing** bezeichnet Methoden zur **Reduktion** unerwünschter Aliasing-Artefakte.
- Verbesserung der **Hardware** ist meist unrealistisch, daher kommen hauptsächlich **Software-Methoden** zum Einsatz.
- Der Fokus liegt oft auf **Anti-Aliasing** zur Behandlung des **Auflösungsproblems**.

Bekannte Aliasing-Effekte neben dem Treppeneffekt

1. **Verschwinden kleiner Objekte**
2. **Unterbrochene, schmale Objekte**
3. **Unterschiedliche Größen gleicher Objekte**
4. **Zerstörung feiner Texturen**.



Begriff Aliasing kommt von Alias: Das was angezeigt wird, zeigt sich an etwas anderem...

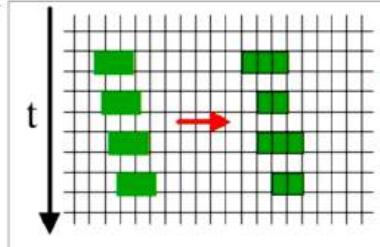


artificial color borders can appear

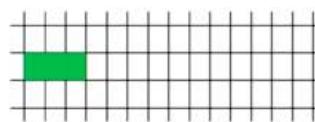
Aliasing kann auch in Animationen vorkommen:

- jumping images

- "worming"



•



- backwards rotating wheels



je nach dem welchen Pixel Mittelpunkt man anschaut bekommt man verschiedene Ergebnisse. Außerdem bekommt man dann einen Worming Effekt (also das Objekt wird immer größer und dann wieder kleiner). Ein anderer Effekt wäre, dass sich zum Beispiel in Filmen es so aussieht als würden sich die Räder rückwärts bewegen.

Antialiasing von Linien:

Ursache von Aliasing

- **Aliasing** entsteht durch **ungenügend feine Abtastung** des wahren Bildes, was zu Fehlern in der Rekonstruktion führt.
- **Nyquist-Shannon-Abtasttheorem:**
 - **Theoretische Grundlage:** Eine Information kann nur korrekt rekonstruiert werden, wenn die **Abtastfrequenz (sampling rate)** mindestens doppelt so hoch ist wie die höchste zu übertragende **Informationsfrequenz**.
 - Diese Grenze wird als **Nyquist-Limit** bezeichnet. ([2. Bildaufnahme](#))

Beispiel und Fehlerbehebung

- **Beispiel:** Eine zu grobe Abtastrate eines Signals führt zu einer **falschen Rekonstruktion** (z.B. eine Kurve wird zu einem Polygonzug, siehe Abbildung).
- Fehler können reduziert werden durch:
 1. **Vorfilterung des Signals.**
 2. **Nachbearbeitung des fertigen Bildes** (jedoch unterliegt diese der Vorfilterung und ist weniger effektiv).

Antialiasing für Linien

- **Pixel, die von einer Linie weiter durchkreuzt werden,** sollen mehr Linienfarbe erhalten als Pixel, die nur leicht gestreift werden.
- **Prozess:**

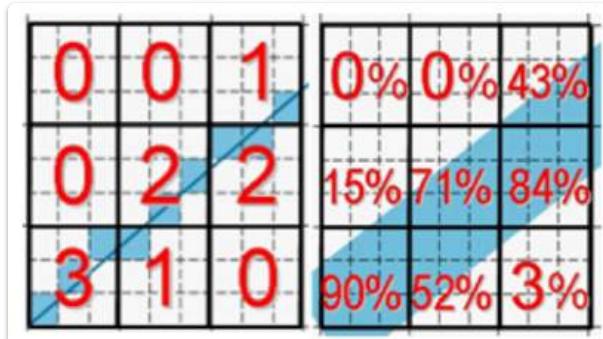
1. **Unterteilung jedes Pixels in Subpixel:** Für jedes Pixel wird gezählt, wie viele Subpixel von der Linie durchkreuzt werden.
2. **Intensitätswahl:** Die Intensität der Linienfarbe wird **proportional zur Anzahl** der Subpixel gewählt, die von der Linie durchquert werden.

Breitere Linien

- Für **breitere Linien** wird der **Prozentsatz der Überdeckung** des Pixels durch die Linie berechnet und die Intensität der Linienfarbe entsprechend angepasst.

Weighted Oversampling

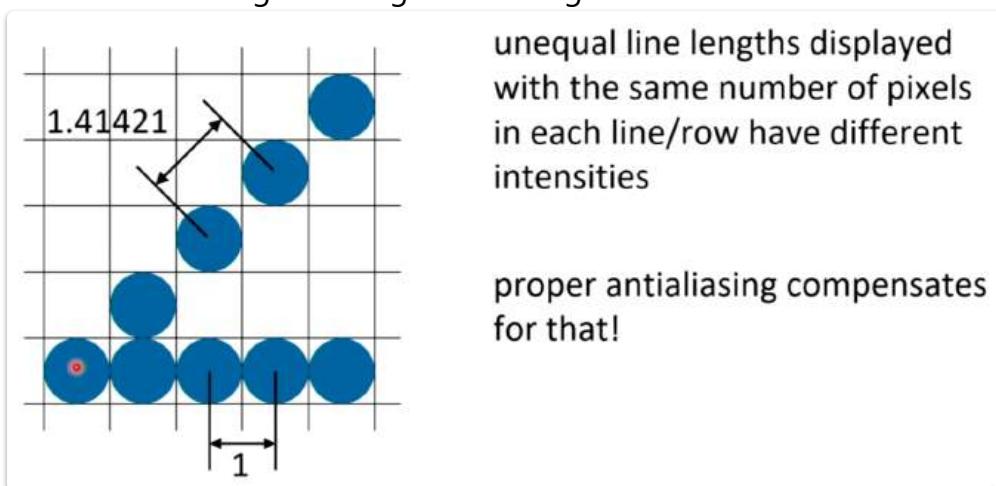
- Basierend auf der Erkenntnis, dass die **Mitte eines Pixels wichtiger** ist als der Rand, werden in einigen Fällen die **Subpixel in der Mitte stärker gewichtet** als die am Rand.
- Diese Technik wird als „**weighted oversampling**“ bezeichnet.



Hier haben wir ein 3 mal 3 Pixelfeld welches dann am Bildschirm angezeigt wird, und die kleinen Pixel im Pixel sind die Subpixel, welche gerendert werden wollen. Man kann allerdings nur die großen Pixel an- und abschalten.

Andere Effekte von Antialiasing:

Verschiedene Längen trotz gleicher Pixelgröße:



Das kann man mit Area Boundaries behoben:

The diagram illustrates the process of adjusting pixel intensities along an area boundary. On the left, a grid shows a diagonal boundary line passing through a cluster of blue circles. The axes are labeled x , y , $x + 1$, and $y + 1$. Below this, two small images show the effect of supersampling: the first shows a rough yellow-to-black transition, while the second shows a smoother one. On the right, a box labeled "alternative 1: supersampling" contains a diagram of a "Subdivided Pixel Area". It shows a black diagonal line labeled "Surface Boundary" crossing a grid. Two horizontal red lines labeled "Scan Line 1" and "Scan Line 2" intersect the grid. The grid cells are shaded in blue, representing the subdivided pixels.

Antialiasing von Polygonkanten:

1. Ziel des Antialiasings

- **Ziel:** Reduktion der aliasing-bedingten Treppeneffekte an den Kanten von Polygonen.



2. Methoden zur Berechnung des Antialiasings

- **Alternativen:**
 - **Supersampling:** Mehrere Proben pro Pixel werden verwendet.
 - **Überdeckungsgradberechnung:** Der Überdeckungsgrad eines Pixels durch das Polygon wird direkt berechnet.

3. Berechnung des Überdeckungsgrads

- **Rasterkonversion:** Der Überdeckungsgrad wird während der Rasterkonversion berechnet, also beim Erzeugen der Randlinie und Füllung des Polygons.
- **Scanlinien-Füllverfahren:**

- Bei der Berechnung der Endpunkte der Scanlinien (Spans) im Füllverfahren fallen genug Informationen an, um den Überdeckungsgrad fast kostenfrei zu berechnen.

4. Verwendung der Entscheidungsvariable aus dem Bresenham-Algorithmus

- Bresenham-Linien-Algorithmus:**

(siehe [5. Rasterisierung](#))

- Die Entscheidungsvariable p_k gibt an, welches Pixel als nächstes gezeichnet wird.
- Diese Variable kann so umgewandelt werden, dass ihr Wert den Überdeckungsgrad des letzten Pixels darstellt.
- Transformation:**
 - $p' = y - y_{mid}$, wobei:

$$y_{mid} = \frac{y_k + y_{k+1}}{2}$$

- Das Vorzeichen von p' hat die gleiche Bedeutung wie das Vorzeichen von p_k .

- Berechnung des Überdeckungsgrads:**

- $p = p' + (1 - m)$, wobei m der Steigungsfaktor ist.
- Der Wert von p liegt im Bereich $0 \leq p \leq 1$ und entspricht dem Überdeckungsgrad an der Stelle x_k .

Antialiasing Area Boundaries (2)

alternative 2: similar to Bresenham algorithm

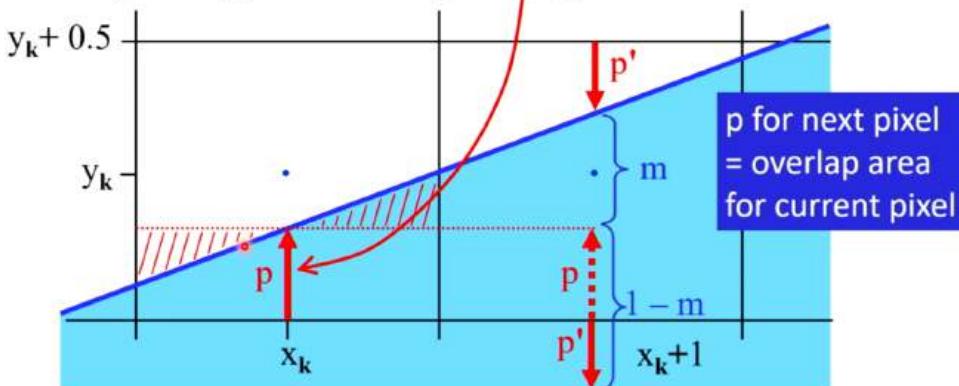
$$p' = y - y_{mid} = [m(x_k + 1) + b] - (y_k + 0.5)$$

$p' < 0 \Rightarrow y \text{ closer to } y_k$
 $p' > 0 \Rightarrow y \text{ closer to } y_{k+1}$

$p = p' + (1 - m) :$
 $p < (1 - m) \Rightarrow y \text{ closer to } y_k$
 $p > (1 - m) \Rightarrow y \text{ closer to } y_{k+1}$

(and $p \in [0,1]$)

$$\begin{aligned} p &= p' + (1 - m) = [m(x_k + 1) + b] - (y_k + 0.5) + (1 - m) = \\ &= mx_k + b - y_k + 0.5 = mx_k + b - (y_k - 0.5) \end{aligned}$$



5. Effizienz

- **Inkrementelle Berechnung:** Das Antialiasing lässt sich sehr schnell und inkrementell berechnen, was zu einer effizienten Verarbeitung führt.

6. Anpassungen für andere Winkel

- **Drehungen und Spiegelungen:** Für andere Winkel werden Drehungen um 90° und/oder Spiegelungen des Verfahrens verwendet.



Abtastung und Fouriertransformation:

1. Fouriertransformation und Frequenzraum

- **Fouriertransformation:** Beschreibt ein Signal im Ortsraum (z.B. eine Scanlinie in einem Bild) als Summe von Sinusschwingungen im Frequenzraum.
 - **Sinusschwingung:** Durch **Frequenz**, **Phase** und **Amplitude** beschrieben.
 - **Spektrum:** Im Frequenzraum wird ein Signal durch sein Spektrum spezifiziert, also Phase und Amplitude in Abhängigkeit von den Frequenzen ω .
 - **Euler-Identität:** $e^{ix} = \cos(x) + i \sin(x)$, mit der Phase und Amplitude effizient durch imaginäre Zahlen beschrieben werden.
- **Inverse Fouriertransformation:** Wandelt das Spektrum im Frequenzraum zurück in das Signal im Ortsraum.

2. Faltung

- **Faltung:** Kombiniert zwei Funktionen und ergibt das integralgewichtete Summenprodukt der beiden.
 - **Formel:** $f_1 * f_2(x) = \int_{-\infty}^{\infty} f_1(\tau) f_2(x - \tau) d\tau$
- **Faltungstheorem:**
 - Multiplikation zweier Funktionen im Ortsraum entspricht der Faltung ihrer Spektren im Frequenzraum:

$$f_1 f_2 = F_1 * F_2$$
 - Faltung im Ortsraum entspricht der Multiplikation der Spektren im Frequenzraum:

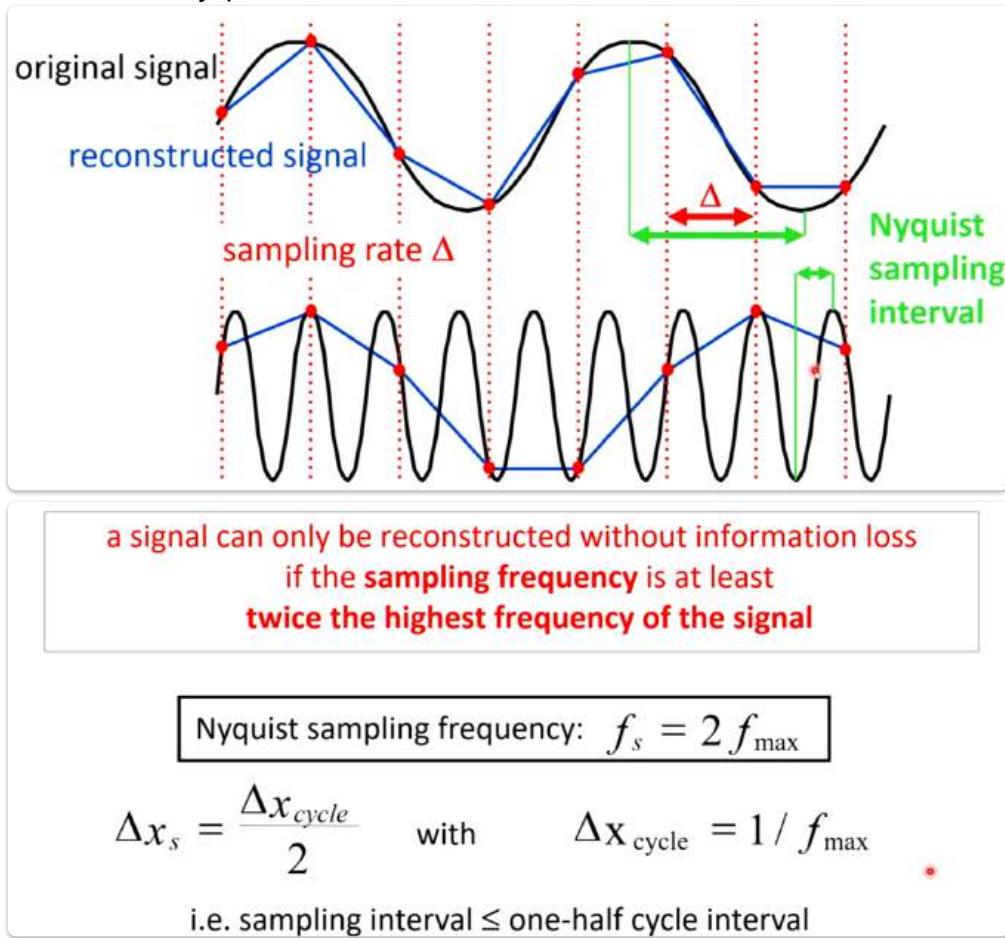
$$f_1 * f_2 = F_1 F_2$$

3. Abtasten und Diskretisierung

- **Abtasten im Ortsraum:** Multiplikation des Signals mit einer Kammfunktion comb_T .
 - **Frequenzraum:** Entspricht einer Faltung mit der Kammfunktion $\text{comb}_{1/T}$.
 - Die Zahnabstände in comb_T und $\text{comb}_{1/T}$ sind invers proportional zueinander.
- **Faltung mit der Kammfunktion:**
 - Führt zu einer **Replikation des Spektrums** im Frequenzraum (Schattenspektra).

4. Nyquist-Limit und Aliasing

- **Nyquist-Limit:** Bei zu niedriger Abtastfrequenz (Abtastintervall T zu groß) sind die Zähne der Kammfunktion comb_T im Ortsraum zu weit auseinander und die Replikationen im Frequenzraum (durch $\text{comb}_{1/T}$) zu nah beieinander.
 - Dies führt zu **Aliasing**, da die Schattenspektra sich überlappen und eine fehlerfreie Rekonstruktion unmöglich wird.
- weiteres zu Nyquist: [2. Bildaufnahme](#)



5. Rekonstruktion und Schattenspektra

- **Exakte Rekonstruktion:** Um die durch die Diskretisierung entstandenen Schattenspektra zu entfernen, wird das Spektrum der diskretisierten Funktion im Frequenzraum mit einer **Rechteckfunktion** multipliziert.
 - Das ursprüngliche Spektrum bleibt übrig.
- **Rekonstruktion im Ortsraum:**

- **Faltung mit Sinc-Funktion:** Die exakte Rekonstruktion erfolgt durch Faltung im Ortsraum mit der **Sinc-Funktion**: $\text{Sinc}(x) = \frac{\sin(x)}{x}$

6. Praktische Rekonstruktion

- Da die Sinc-Funktion über einen unendlichen Bereich nicht null ist, wird für eine praktikable Rekonstruktion:
 - **Rechteckfunktion** (Nächster-Nachbar-Interpolation) oder
 - **Dreiecksfunktion** (lineare Interpolation) verwendet.

Fourier Transform

TU WIEN

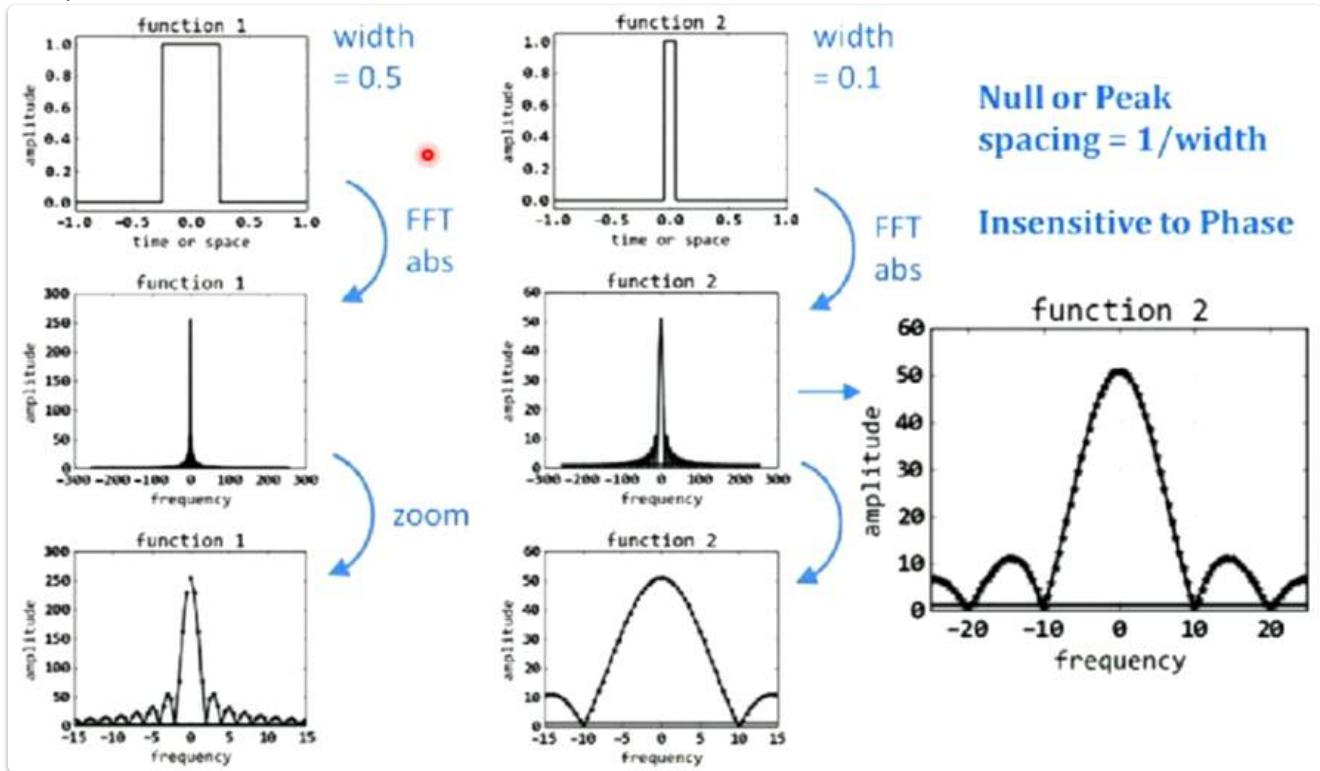
■ Link between spatial $f(x)$ and frequency $F(\omega)$ domain

$$F(\omega) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i \omega x} dx$$

$$f(x) = \int_{-\infty}^{\infty} F(\omega) e^{2\pi i \omega x} d\omega$$

$$e^{ix} = \cos x + i \sin x$$

Beispiel zur Fourier Transformation:



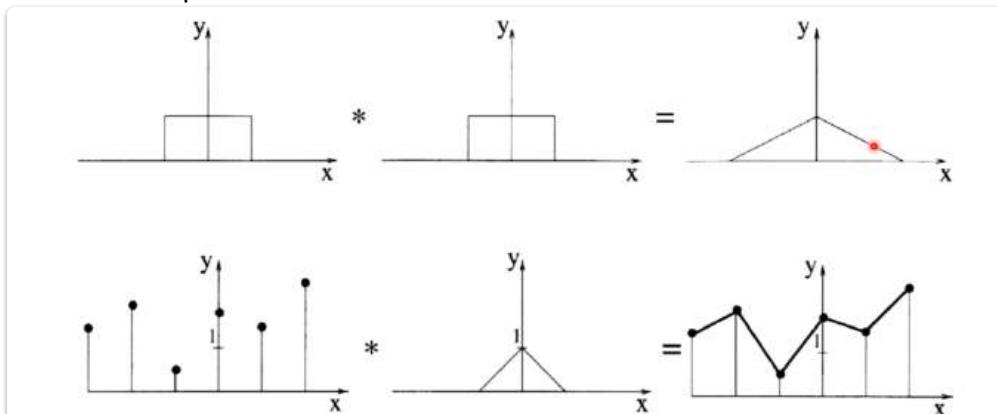
- The spectrum of the convolution of two functions is equivalent to the product of the transforms of both input signals, and vice versa
- Convolution $f_1 * f_2(x) = \int_{\mathbb{R}} f_1(\tau) f_2(x - \tau) d\tau$

Convolution theorem $f_1 * f_2 \equiv F_1 F_2$

$$F_1 * F_2 \equiv f_1 f_2$$

"Wenn ich eine Faltung im einen Raum mache (Orts oder Frequenzraum), ist es im anderen Raum dann zu multiplizieren"

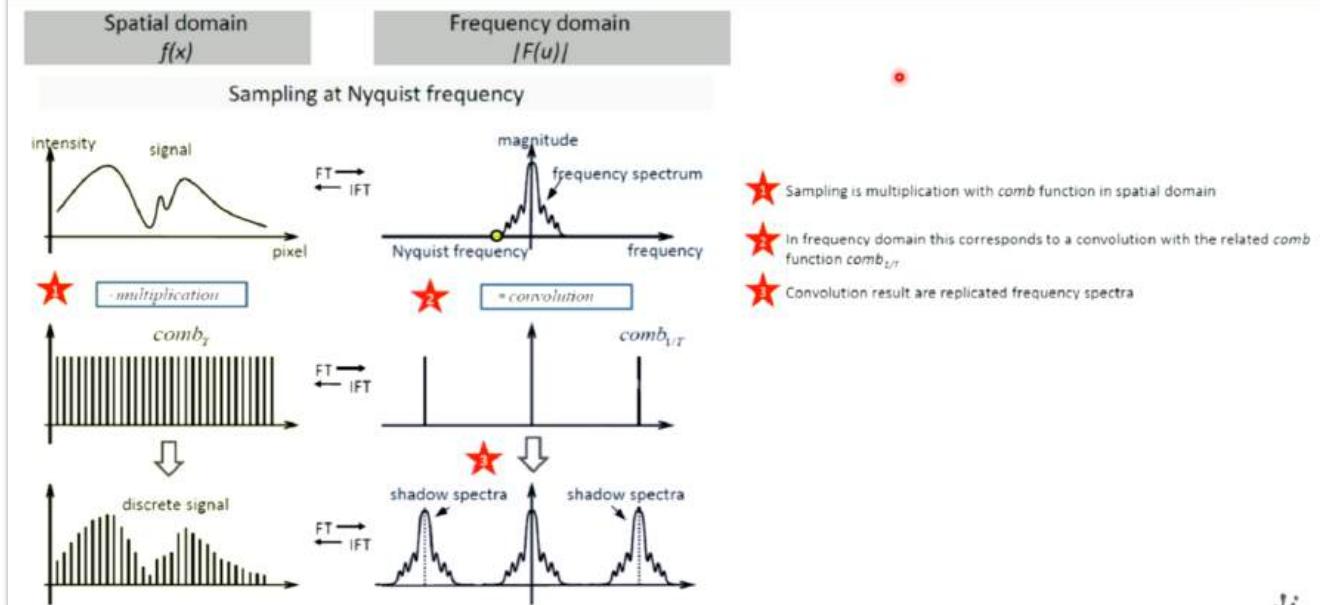
Hier ein Beispiel zu dem Convolution theorem: (Mehr dazu siehe: [7. Globale Operationen](#))



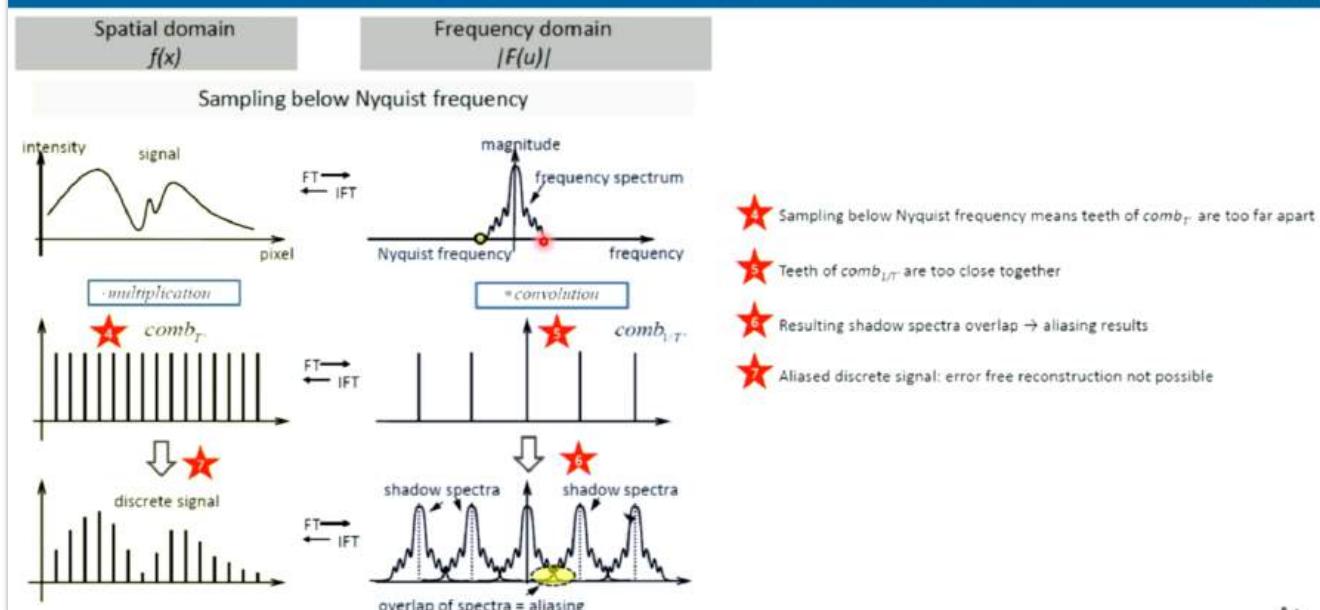
damit das 2. geht, muss der Abstand der Samplepunkten genau doppelt so groß sein wegen Nyquist (das bitte nochmal Fact-checken, er hat schnell gesprochen)

Cheatsheet für alles was Sampling betrifft:

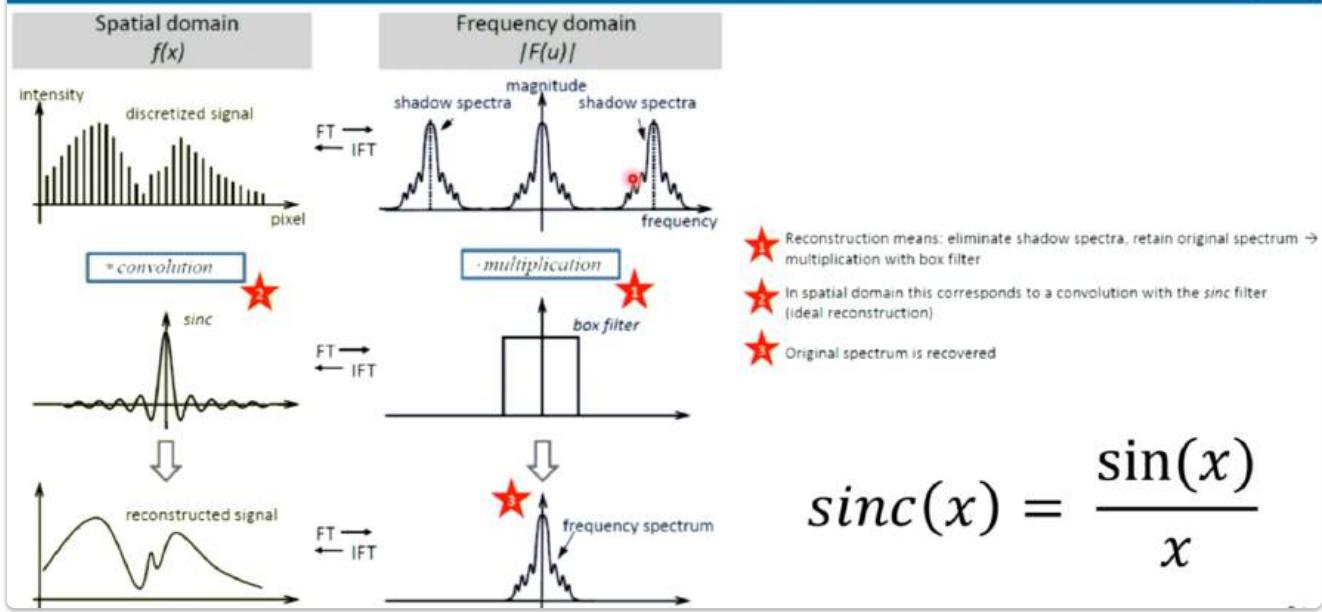
Sampling at the Nyquist Frequency



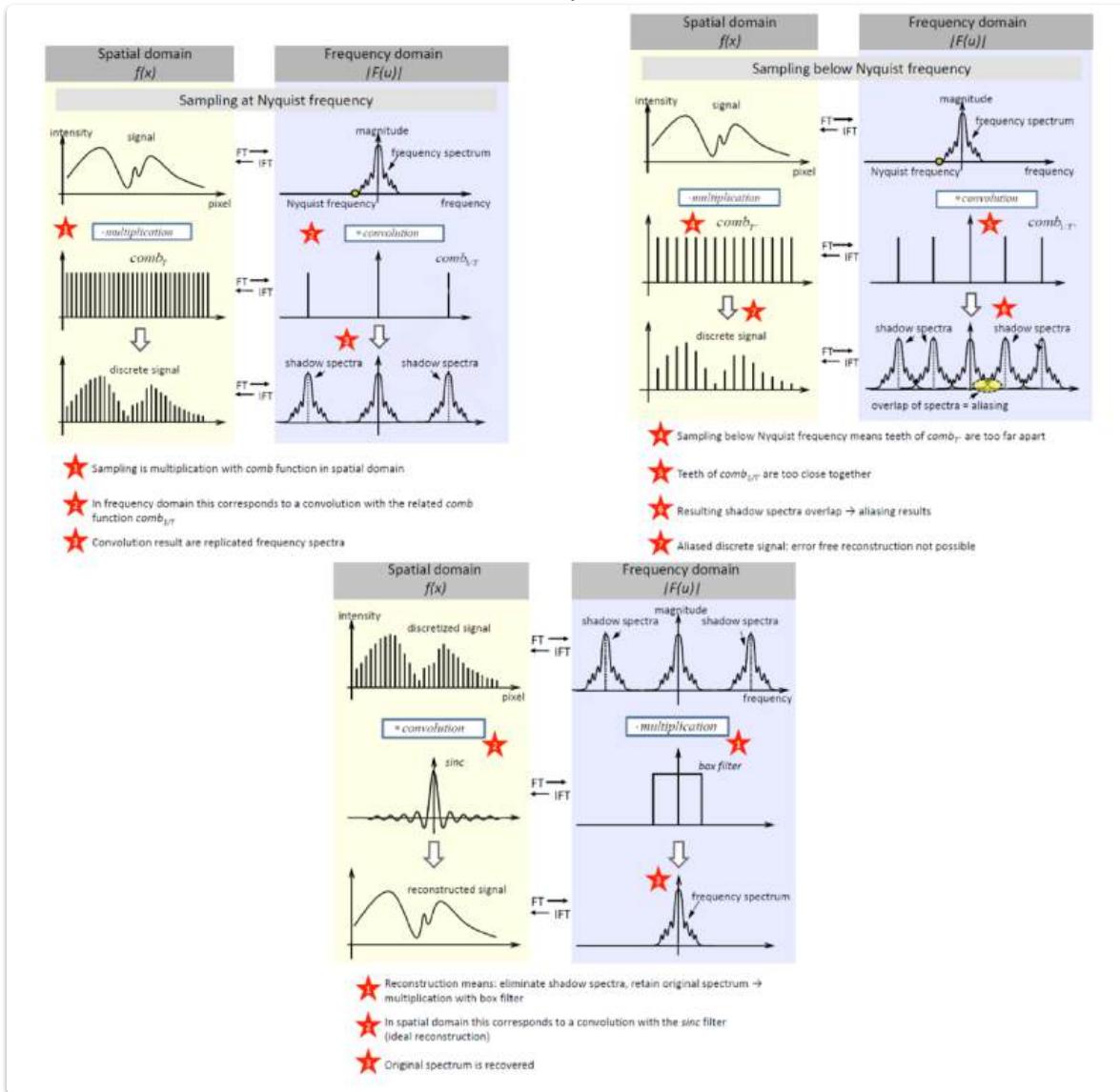
Sampling Below the Nyquist Frequency



Reconstruction

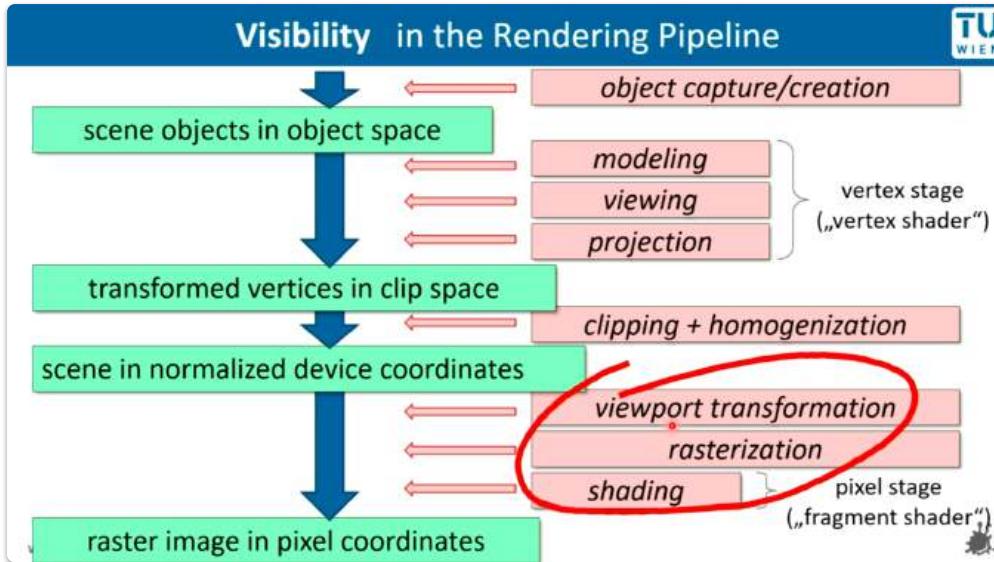


hier nochmal hochauflösend aus dem Skriptum:



8. Sichtbarkeitsverfahren

Das Sichtbarkeitsverfahren kommt hier in dem Bereich der Rendering Pipeline vor:



1. Ziel von Sichtbarkeitsverfahren

- **Ziel:** Korrekte und glaubwürdige Darstellung von Szenen, indem unsichtbare Teile der Objekte weggelassen werden.
 - **Unsichtbare Teile:** Rückseiten von Objekten und Teile, die von anderen Objekten verdeckt werden.
- **Begriff:** Hidden-Line- oder Hidden-Surface Eliminierung.

2. Ansätze in der Sichtbarkeitsberechnung

- **Objektraum-Methoden:**
 - **Vorgehen:** Vergleichen der Lage der Objekte miteinander.
 - **Ziel:** Nur die vorderen (sichtbaren) Teile der Objekte werden gezeichnet.
- **Bildraum-Methoden:**
 - **Vorgehen:** Für jeden Bildteil wird separat berechnet, was an dieser Stelle sichtbar ist.

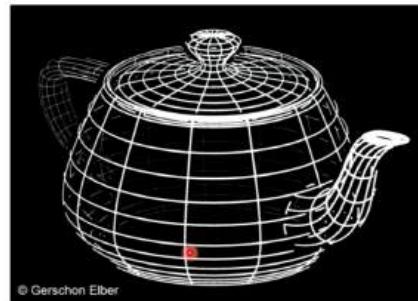
3. Berücksichtigung von Transparenz

- Die Erläuterungen zu den Sichtbarkeitsverfahren berücksichtigen **nicht** transparente Objekte.

Depth Cueing

3D Display: Depth Cueing + Visibility

- only visible lines
- intensity decreases with increasing distance



Da geht's darum bei Objekten unsichtbare Polygone anders darzustellen

Backface Detection (Backface Culling)

1. Ziel von Backface Culling

- **Ziel:** Elimination von Polygonen, die sicher nicht sichtbar sind, um den Aufwand nachfolgender Verarbeitungsschritte zu reduzieren.
 - **Nicht sichtbare Polygone:** Polygone, deren Oberflächennormale vom Betrachter weg zeigen.

2. Funktionsweise

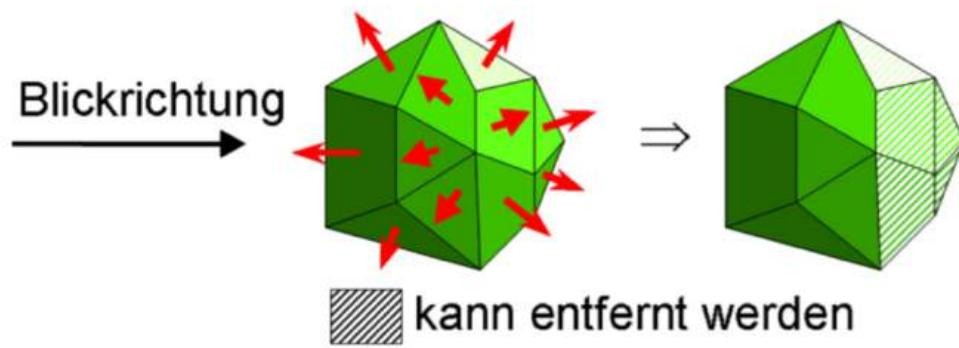
- **Backface Culling** ist kein vollständiges Sichtbarkeitsverfahren, sondern dient als Optimierungsschritt.
 - **Durchschnittliche Reduktion:** Etwa 50% der Polygone werden entfernt.

3. Berechnungsverfahren

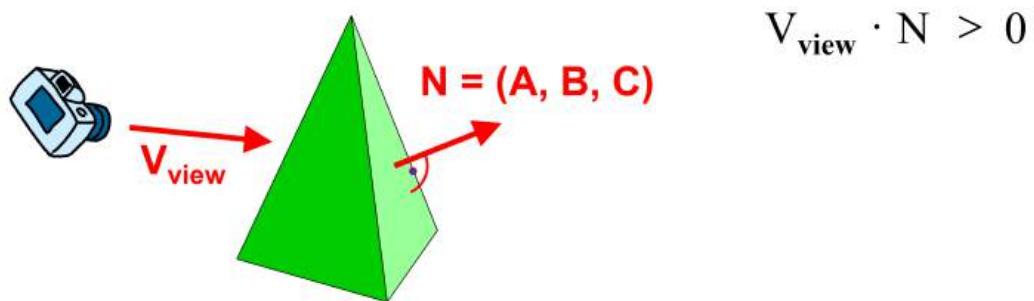
- **Orthographische Projektion:**
 - **Berechnung:** Das Skalarprodukt des Blickrichtungsvektors V_{view} und der Oberflächennormale N wird berechnet.
 - **Formel:** $V_{view} \cdot N > 0 \Rightarrow$ Polygon ist unsichtbar.
- **Perspektivische Projektion:**
 - **Berechnung:** Der Blickpunkt (x, y, z) wird in die Ebenengleichung eingesetzt.
 - **Formel:** $Ax + By + Cz + D < 0 \Rightarrow$ Polygon ist unsichtbar.

4. Annahmen

- Die gleichen Annahmen wie bei der Verwendung von „Polygonlisten“ werden zugrunde gelegt.



eliminating back faces of closed polyhedra
 view point (x, y, z) “inside” a polygon surface if
 $Ax + By + Cz + D < 0$
 or polygon with normal $N=(A, B, C)$ is a back face if

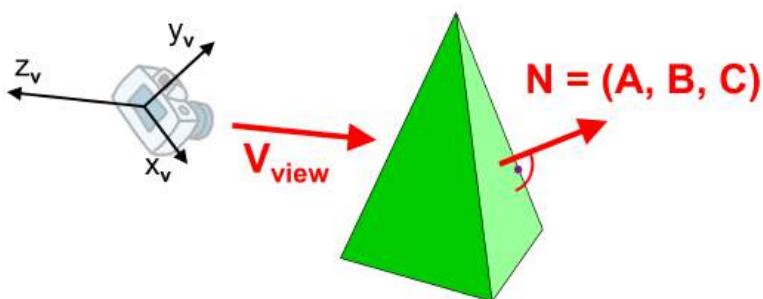


object description in viewing coordinates $\Rightarrow V_{\text{view}} = (0, 0, V_z)$

$$V_{\text{view}} \cdot N = V_z C$$

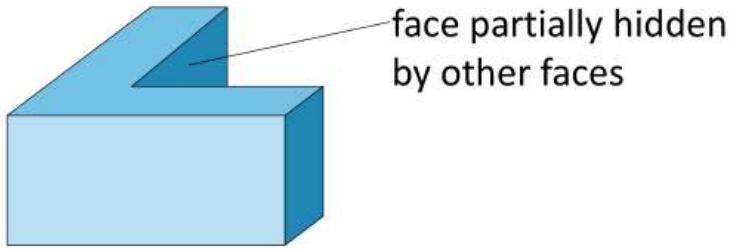
sufficient condition: if $C \leq 0$ then back-face

↑
negative !



Der Vektor hat in x und y Koordinaten 0, 0 und nur die Z Koordinate ist relevant.

complete visibility test only for non-overlapping convex polyhedra



... is a preprocessing step for other objects:

about **??** of surfaces are eliminated

Depth Buffer Verfahren (Z Puffer Verfahren)

[EVC_Skriptum_CG](#), p.32

Problemstellung: Sichtbarkeitsproblem

- Ziel: Bestimmung, welche Objekte in einer 3D-Szene für den Betrachter sichtbar sind und welche verdeckt werden.
- Lösung des Z-Puffer-Algorithmus erfolgt pixelweise für eine gegebene Bildauflösung.

Kernidee des Z-Puffer-Algorithmus

- **Zusätzlicher Speicher:** Neben dem Framebuffer (Farbinformation pro Pixel) wird ein Z-Puffer (oder Tiefenpuffer) benötigt.
- **Z-Wert pro Pixel:** Der Z-Puffer speichert für jedes Pixel die Tiefeninformation (z-Koordinate) des bisher gezeichneten Objekts.
- **Blickrichtung:** Normalerweise in z-Richtung, daher Speicherung des einzelnen z-Wertes ausreichend.

Speicherbereiche

- **Framebuffer:** Speichert die Farbinformation jedes Pixels.
- **Z-Puffer (Depth Buffer):** Speichert den z-Wert (Tiefe) jedes Pixels.

Ablauf des Algorithmus

```

for all (x,y)                      // Initialisierung des Hintergrundes
    depthBuff(x,y) = -1             // größtmögliche Entfernung
    frameBuff(x,y) = backgroundColor
for each polygon P                  // Schleife über alle Polygone
    for each position (x,y) on polygon P
        calculate depth z
        if z > depthBuff(x,y) then
    
```

```

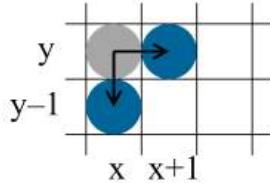
depthBuff(x,y) = z
frameBuff(x,y) = surfColor(x,y)
else
    // else nichts !

```

polygons with corresponding z-values image depth-buffer	<table border="1" style="border-collapse: collapse; width: 100px; height: 100px;"> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td style="background-color: blue;">.3 .3</td><td></td></tr> <tr><td></td><td></td><td style="background-color: blue;">.3 .3</td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> </table> <table border="1" style="border-collapse: collapse; width: 100px; height: 100px;"> <tr><td></td><td></td><td style="background-color: magenta;">.8 .7</td><td></td></tr> <tr><td></td><td></td><td style="background-color: magenta;">.7 .6</td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> </table> <table border="1" style="border-collapse: collapse; width: 100px; height: 100px;"> <tr><td></td><td></td><td style="background-color: green;">.6 .5 .4</td><td></td></tr> <tr><td></td><td></td><td style="background-color: green;">.6 .5 .4</td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> </table>							.3 .3				.3 .3								.8 .7				.7 .6								.6 .5 .4				.6 .5 .4						<table border="1" style="border-collapse: collapse; width: 100px; height: 100px;"> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> </table> <table border="1" style="border-collapse: collapse; width: 100px; height: 100px;"> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> </table> <table border="1" style="border-collapse: collapse; width: 100px; height: 100px;"> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> </table>																																																
		.3 .3																																																																																								
		.3 .3																																																																																								
		.8 .7																																																																																								
		.7 .6																																																																																								
		.6 .5 .4																																																																																								
		.6 .5 .4																																																																																								

Effizienz

- **Inkrementelle Berechnung:** Für ebene Polygone lassen sich die z-Werte natürlich wieder inkrementell effizienter berechnen.



depth at (x, y) :

depth at $(x+1, y)$:

depth at $(x, y-1)$:

$$Ax + By + Cz + D = 0$$

$$z = \frac{-Ax - By - D}{C}$$

constants!

$$z' = \frac{-A(x+1) - By - D}{C} = z - \frac{A}{C}$$

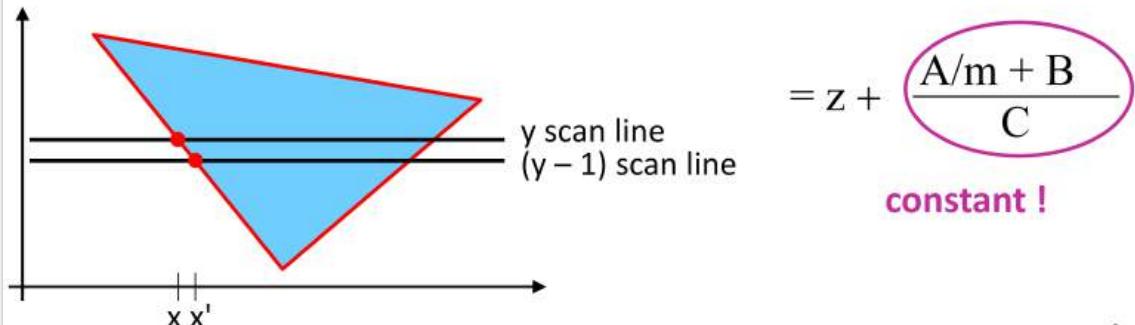
$$z'' = \frac{-Ax - B(y-1) - D}{C} = z + \frac{B}{C}$$

Vorteile

- **Keine Sortierung der Objekte notwendig:** Polygone können in beliebiger Reihenfolge gezeichnet werden.

$$z = \frac{-Ax-By-D}{C}$$

$$y' = y - 1 \quad \Rightarrow \quad z' = \frac{-A(x-1/m)-B(y-1)-D}{C}$$



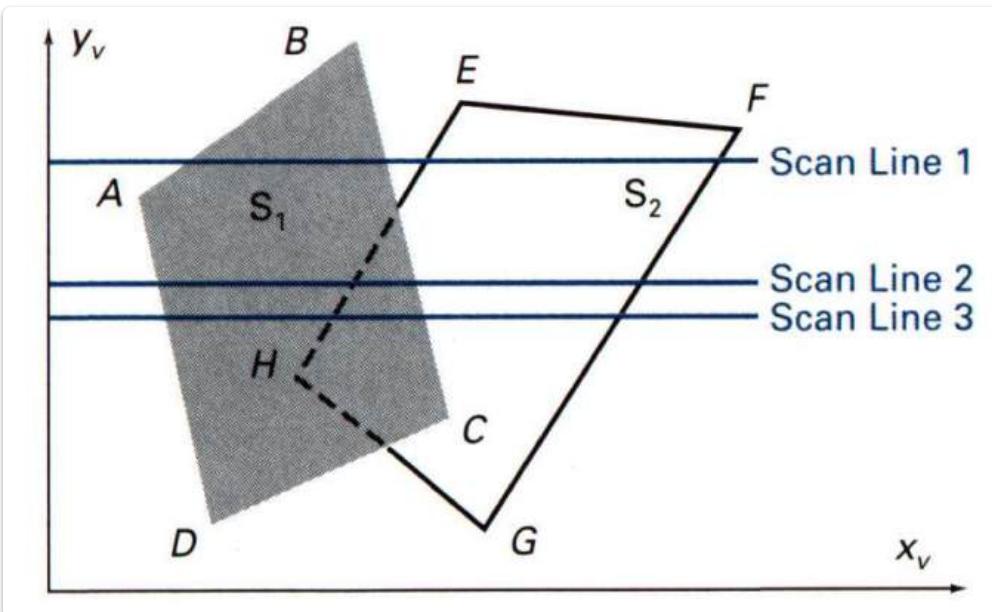
Anmerkung

- **Viewport-Transformation:** Oft wird z mit -1 multipliziert bevor das z -Puffer-Verfahren angewendet wird.

Scanline-Methode

EVC_Skriptum_CG, p.32

- **Ziel:** Korrekte Sichtbarkeitsberechnung pixelzeilenweise.
- **Ablauf:** Zeilenweises Vorgehen (z.B. von oben nach unten, y fallend).
- **Vorteil:** Nutzt die Kohärenz zwischen aufeinanderfolgenden Scanlines (geringe Änderung des Sichtbarkeitsverhaltens).



Depth-Sorting-Methode (Painter's Algorithm)

EVC_Skriptum_CG, p.33

- **Grundprinzip:**

- Sortiere alle Polygone nach ihrer Tiefenlage (von hinten nach vorne).
- Zeichne die Polygone in dieser sortierten Reihenfolge.
- **Logik:** Spätere (weiter vorne liegende) Polygone übermalen frühere (weiter hinten liegende) und erzeugen so korrekte Sichtbarkeit.
- **Hauptaufwand:** Sortierung der Polygone.
- **Herausforderung:** Sicherstellen, dass kein Polygon ein anderes verdeckt, das in der Sortierreihenfolge später kommt (weiter vorne liegt).
- **Vorgehensweise:**
 1. **Grobsortierung:** Schnelle erste Sortierung der Polygone.
 2. **Überprüfung:** Testen, ob die Sortierung korrekt ist (keine fehlerhaften Verdeckungen).
 3. **Umsortierung (ggf.):** Bei Bedarf Anpassung der Reihenfolge, um korrekte Sichtbarkeit zu gewährleisten.

surfaces sorted in order of decreasing depth (viewing in $-z$ -direction)

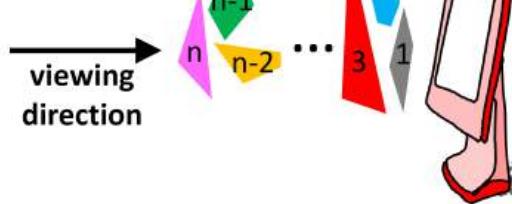
- (1) "approximate"-sorting using smallest z -value (greatest depth)
- (2) fine-tuning to get correct depth order

surfaces scan converted in order

sorting both in image and object space

scan conversion in image space

also called "painter's algorithm"



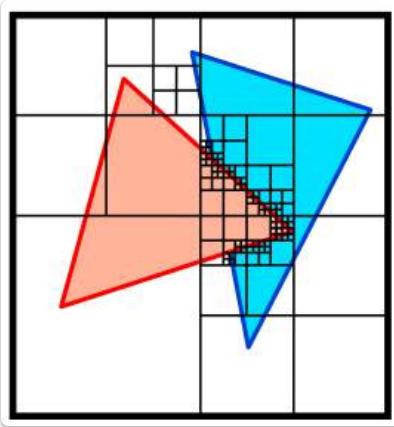
Werner Purgathofer

23

Area-Subdivision Methode

[EVC_Skriptum_CG](#), p.33

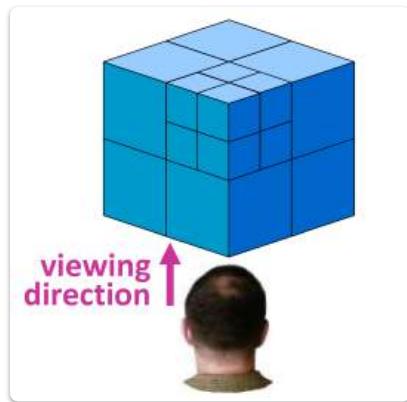
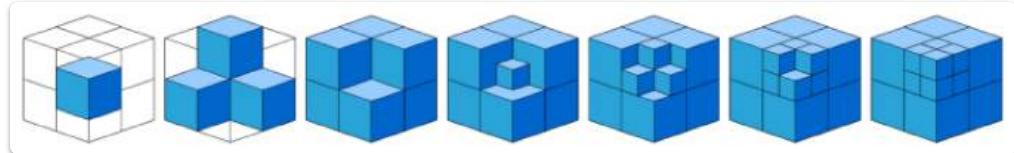
- **Grundidee:** Divide-and-Conquer-Ansatz für das Sichtbarkeitsproblem.
- **Analogie:** Ähnlich der Quadtree-Repräsentation von Bildern.
- **Vorgehensweise:**
 1. **Einfache Fälle:** Sichtbarkeitsproblem wird in grober Auflösung gelöst.
 2. **Komplizierte Fälle:**
 - Unterteilung der aktuellen Bildfläche in vier gleich große Viertel.
 - Rekursive Anwendung der Methode auf jede der vier Teilstufen.
- **Garantie:** Die rekursive Unterteilung bis zur maximalen Bildauflösung stellt eine pixelgenaue Lösung des Sichtbarkeitsproblems sicher.



Octree-Methode

EVC_Skriptum_CG, p.33

- **Datenstruktur:** Repräsentation der Szene als Octree (raumorientierter Baum).
- **Vorteil der Datenstruktur:** Implizites Wissen über die Tiefenordnung (was vorne und hinten ist) für jede Blickrichtung.
- **Rendering-Strategien:**
 - **Von hinten nach vorne:**
 - Rekursives Durchlaufen der Octree-Knoten.
 - Rendern der Teilwürfel in der Reihenfolge: *entferntester* -> 3 nächstnäheren -> nächste 3 -> vorderster.
 - Beispiel für frontale Blickrichtung

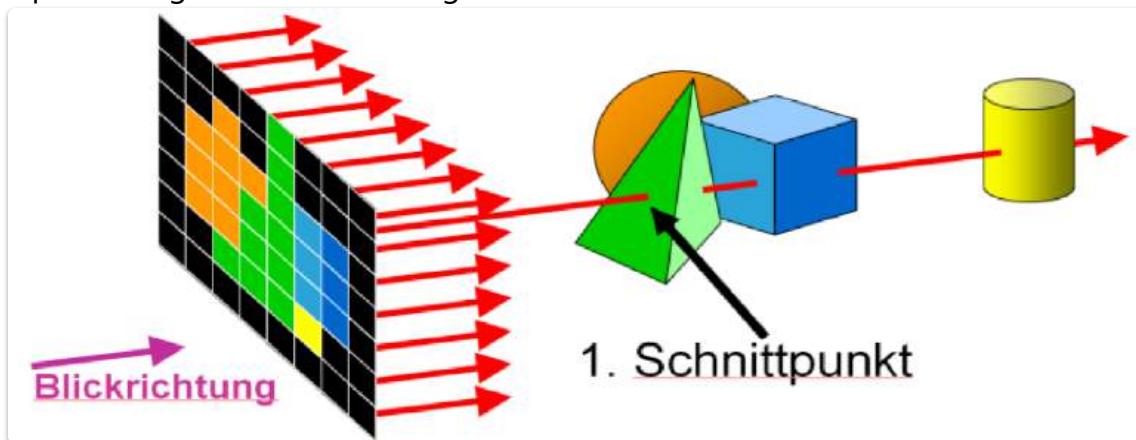


- **Von vorne nach hinten:**
 - Rekursives Durchlaufen der Octree-Knoten.
 - Zeichnen nur der sichtbaren Bereiche.
 - Notwendigkeit, sich bereits gezeichnete Bereiche zu merken, um Overdraw zu vermeiden.
- **Genereller Vorteil:** Das inhärente Wissen über die Tiefenordnung im Vergleich zu anderen Datenstrukturen vereinfacht die Sichtbarkeitsbestimmung.

Ray-Casting

EVC_Skriptum_CG, p.33, 10. Ray-Tracing

- **Grundprinzip:** Berechnung der Sichtbarkeit für jedes Pixel einzeln.
- **Ablauf:**
 1. **Blickstrahl:** Vom Pixel in Blickrichtung wird eine gerade Linie (Blickstrahl) in die Szene projiziert.
 2. **Schnittpunktberechnung:** Der Blickstrahl wird mit allen Objekten/Polygonen in der Szene geschnitten.
 3. **Nächster Schnittpunkt:** Aus der Menge der Schnittpunkte wird derjenige ausgewählt, der am nächsten zum Betrachter liegt.
 4. **Pixelfarbe:** Die Farbe der Oberfläche am nächsten Schnittpunkt bestimmt die Farbe des betrachteten Pixels.
- **Vorteile:**
 - Ermöglicht das Rendern verschiedenster Oberflächen (nicht nur Polygone), sofern der Schnitt mit einer Geraden berechenbar ist (z.B. Freiformflächen).
 - **Benötigte Information (für Schattierung):** Oberflächennormale am Auftreffpunkt des Strahls.
- **Nachteile:**
 - **Hoher Rechenaufwand:** Für jedes Pixel müssen Schnittpunktberechnungen mit potenziell vielen Objekten durchgeführt werden (Millionen von Pixeln und möglicherweise tausende bis Millionen von Objekten).
 - **Notwendigkeit:** Effiziente Implementierung der Schnittpunkttests und weitere Optimierungen zur Reduzierung des Rechenaufwands sind unerlässlich.



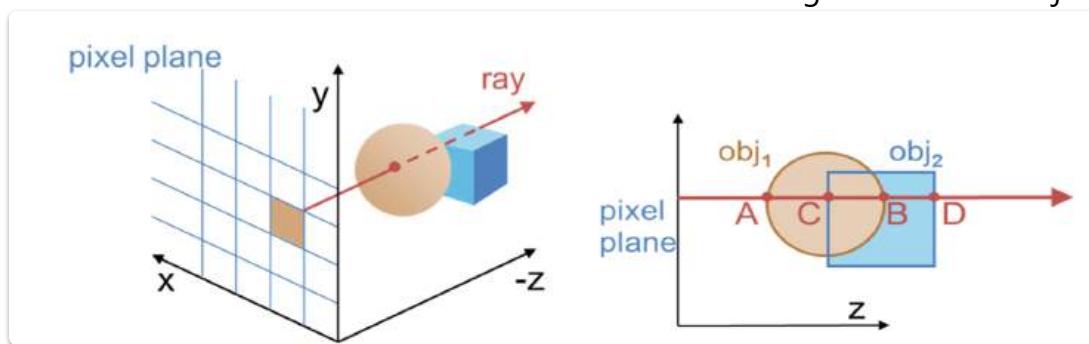
Ray-Casting von CSG-Objekten

EVC_Skriptum_CG, p.33

- **Gebräuchliche Methode:** Pixelweise Berechnung des Bildes durch Ray-Casting.
- **Ablauf pro Pixel:**
 1. **Ray-Erzeugung:** Ein Blickstrahl (Ray) wird in Blickrichtung "ausgeworfen".
 2. **Schnitt mit allen Objekten:** Der Ray wird mit allen Objekten der Szene geschnitten.

3. **Vorderster Schnittpunkt:** Der Schnittpunkt, der am nächsten zum Betrachter liegt, bestimmt das sichtbare Objekt.

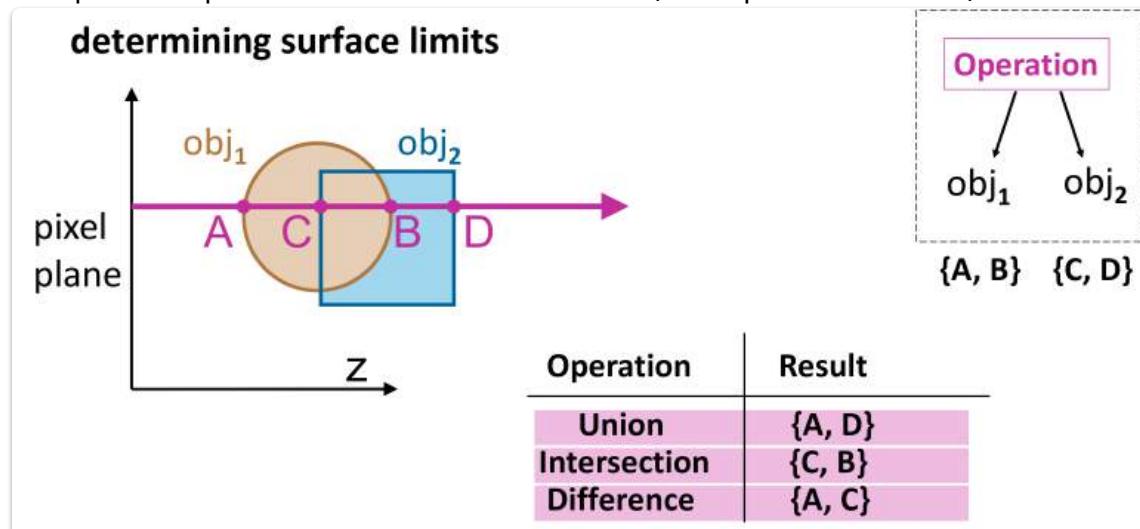
4. **Pixelfarbe:** Das Pixel erhält die Farbe des am vordersten geschnittenen Objekts.



- **Rekursive Berechnung bei CSG-Bäumen:**

- **Endknoten (Primitive Objekte):** Direkte und einfache Berechnung aller Ray-Objekt-Schnittpunkte.
- **Zwischenknoten (Boolesche Operationen):**
 - Die Schnittpunktlisten der beiden Kindknoten werden entsprechend dem Booleschen Operator verknüpft:
 - **Vereinigung (Union):** Kombination der Schnittpunktlisten (Beispiel: A, D).
 - **Durchschnitt (Intersection):** Schnittmenge der Schnittpunktlisten (Beispiel: C, B).
 - **Differenz (Subtraction):** Schnittpunkte des ersten Objekts, die nicht im zweiten Objekt liegen (Beispiel: A, C).
- **Wurzelknoten:** Der erste Punkt (der dem Betrachter am nächsten liegt) der resultierenden verknüpften Schnittpunktliste wird ausgewählt.

- **Relation zu Ray-Tracing:** Ray-Casting ist eine vereinfachte Version von Ray-Tracing, das komplexere optische Effekte simulieren kann (wird später behandelt).



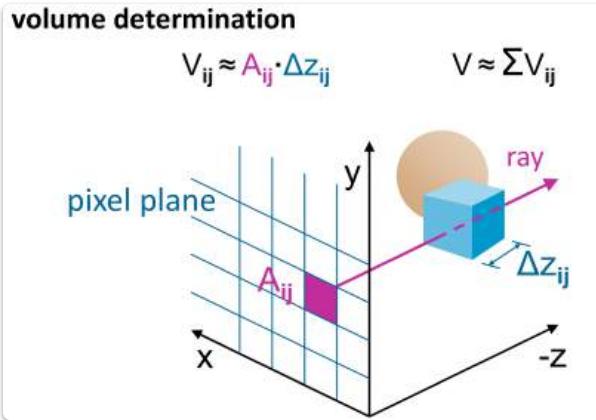
Ray-Casting ist eine vereinfachte Version von Ray-Tracing, mit dem noch viele weitere optische Effekte simuliert werden können. Dazu kommen wir in einem späteren Kapitel.

Ray-Casting = für jedes Pixel der Darstellungsfläche:

- erzeuge eine Gerade durch das Pixel in Blickrichtung („Blickstrahl“)
- schneide den Blickstrahl mit allen Objekten
- wähle aus der Schnittpunktliste den zum Betrachter nächsten Punkt
- färbe das Pixel mit der Farbe der Oberfläche dieses Punktes

Ergänzung von den Slides

Man kann das auch verwenden um das Volumen zu berechnen



Klassifizierung der Verfahren

[EVC_Skriptum_CG, p.34](#)

Wir wollen nun noch überlegen, welche Verfahren im Objektraum arbeiten und welche im Bildraum. Dies ist nicht immer ganz eindeutig klassifizierbar, aber im Großen und Ganzen gilt:

Objektraum-Verfahren:

- Backface Detection
- Depth Sorting
- Octree-Methode

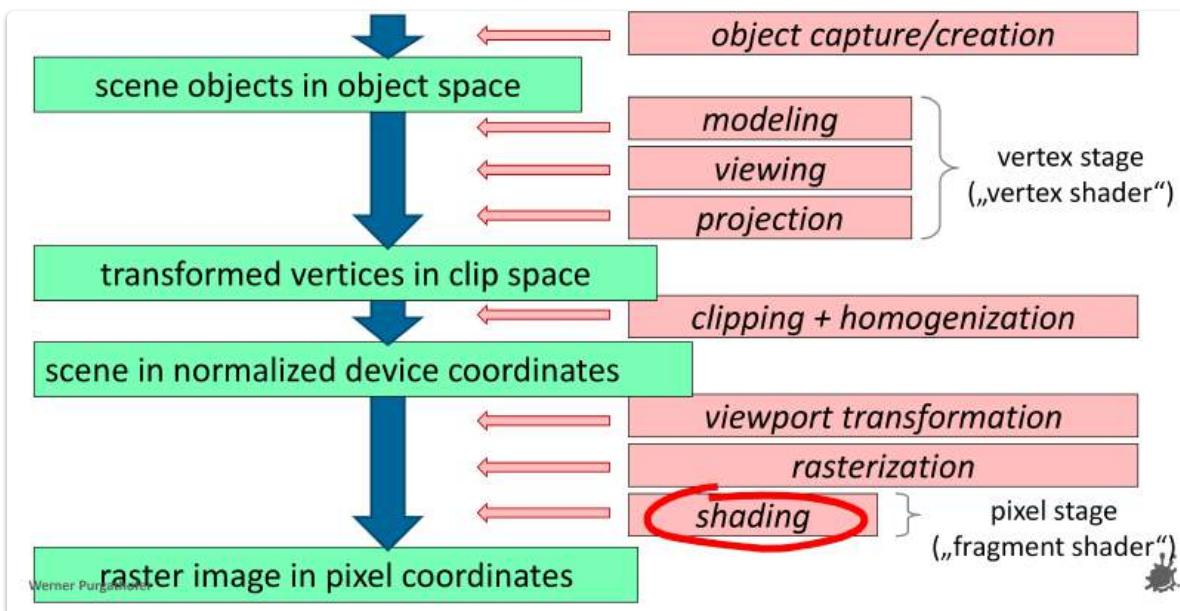
Bildraum-Verfahren:

- Z-Puffer
- Scanline-Methode
- Area Subdivision
- Ray Casting

9. Beleuchtung und Schattierung

EVC_Skriptum_CG, p.35

- **Definition:** Ein Beleuchtungsmodell (oder Schattierungsmodell, Illumination/Lighting/Shading Model) berechnet die wahrgenommene Farbe/Helligkeit eines Objekts für den Betrachter basierend auf:
 - Lichtverhältnissen in der Szene.
 - Oberflächeneigenschaften des Objekts.
- **Ziel:** Bestimmung der Farbe, die das entsprechende Pixel im Bild erhalten soll.
- **Bedeutung:** Zusammen mit der perspektivischen Projektion der wichtigste Faktor für realistisch aussehende Computergraphik-Bilder.
- **Vereinfachung:** Die folgenden Betrachtungen und Formeln konzentrieren sich zunächst auf die **Helligkeit** der Beleuchtung.
- **Farbbehandlung:** Um Farben zu berücksichtigen, müssen die Berechnungen für verschiedene Wellenlängen durchgeführt werden (im einfachsten Fall für Rot, Grün und Blau).



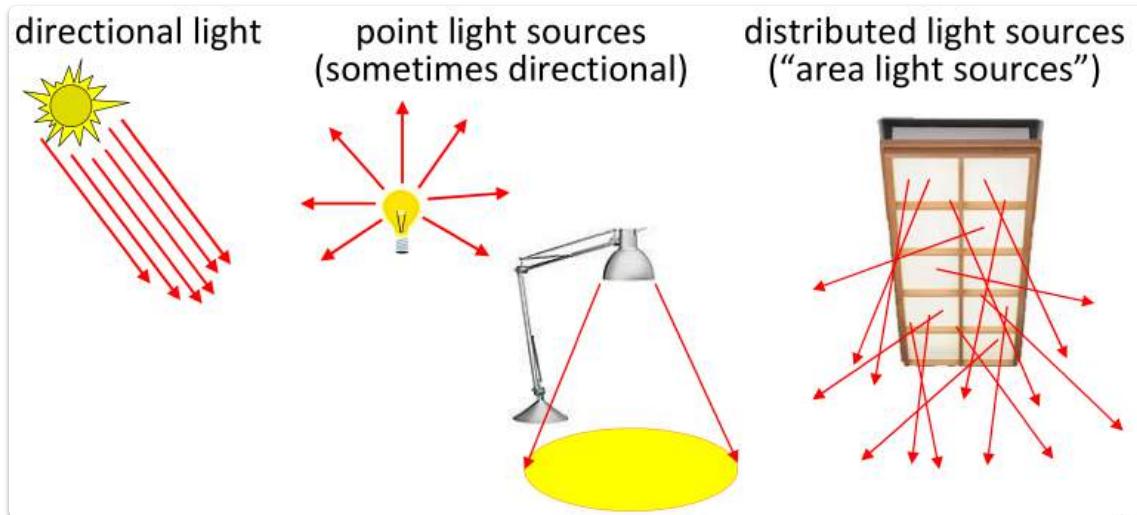
Lichtquellen und Oberflächen

Lichtquellen

EVC_Skriptum_CG, p.35

- **Notwendigkeit:** Voraussetzung zur Berechnung von Beleuchtungseffekten in einer Szene.
- **Merkmale von Lichtquellen:**
 - **Form:**

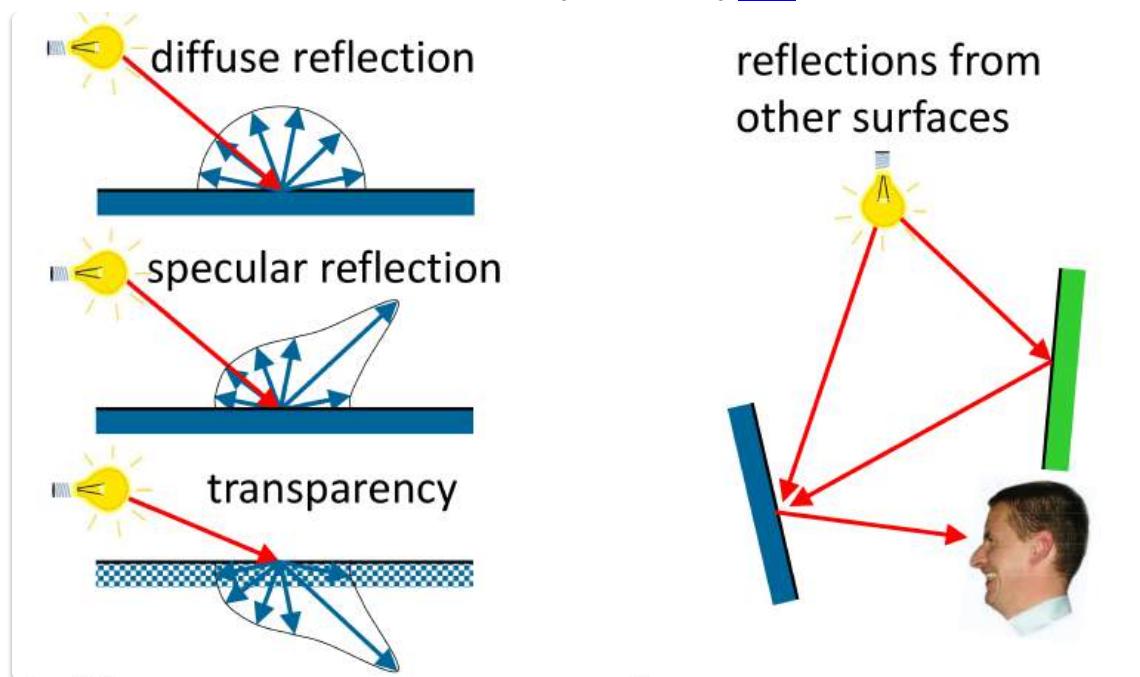
- Lichtrichtung (z.B. paralleles Sonnenlicht)
- Punktlichtquellen (strahlen Licht von einem einzelnen Punkt in alle Richtungen)
- Gerichtete Punktlichtquellen (kombinieren Punktlichtquelle mit einer kegelförmigen oder rechteckigen Lichtverteilung)
- Flächige Lichtquellen (emittieren Licht von einer ausgedehnten Oberfläche)
- ... (weitere Formen sind möglich)
- **Eigenschaften:**
 - Helligkeit (Intensität des Lichts)
 - Farbe (Spektrale Zusammensetzung des Lichts)
 - Entfernung (bei einigen Lichtquellen relevant für die Intensitätsabnahme)
 - ... (weitere Eigenschaften können definiert werden)



Objektoberflächen

EVC_Skriptum_CG, p.35

- **Wechselwirkung mit einfallendem Licht:** Oberflächen können Licht auf verschiedene Weisen beeinflussen:
 - **Diffuse Reflexion:** Licht wird in alle Richtungen gleichmäßig reflektiert (Beispiele: Papier, Kreide).
 - **Spiegelnde Reflexion:** Licht wird bevorzugt in die Spiegelungsrichtung reflektiert (Beispiele: Lack, Metall).
 - **Transparenz:** Licht durchdringt die Oberfläche und tritt auf der anderen Seite wieder aus (Beispiele: Glas, Wasser).
- **Realität:** Die meisten Oberflächen weisen eine Kombination dieser Eigenschaften auf.
- **Indirekte Beleuchtung:** Es ist wichtig zu beachten, dass Licht nicht nur direkt von Lichtquellen auf Oberflächen trifft, sondern auch von anderen Oberflächen reflektiert wird und somit zur Beleuchtung beiträgt.



Ein einfaches Beleuchtungsmodell

EVC_Skriptum_CG, p.35

- **Hintergrund:** Die physikalisch genaue Simulation von Licht und seiner Interaktion mit Oberflächen ist sehr aufwendig.
- **Ansatz in der Praxis:** Verwendung vereinfachter, empirischer Beleuchtungsmodelle.
- **Grundstruktur (ungefähre Darstellung):** (Die genaue Struktur wird in den folgenden Abschnitten detaillierter erläutert.)
- **Ziel:** Eine visuell plausible Beleuchtung mit überschaubarem Rechenaufwand zu erzielen.

Hintergrundlicht (Ambientes Licht)

EVC_Skriptum_CG, p.35

- **Realitätsbezug:** Objekte strahlen einen Teil des auf sie treffenden Lichts ab, wodurch es auch in Bereichen ohne direkte Beleuchtung nicht vollständig dunkel ist.
- **Definition:** Dieses überall vorhandene, indirekte Basislicht wird als **ambientes Licht** oder **Hintergrundlicht** bezeichnet.
- **Implementierung in einfachen Beleuchtungsmodellen:** Ein konstanter Helligkeitswert (I_a) wird zu jeder Beleuchtungsberechnung addiert, um diesen globalen Lichtanteil zu approximieren.

- ambient light (background light) I_a

- constant over a surface
- independent of viewing direction
- diffuse-reflection coefficient k_d ($0 \leq k_d \leq 1$)
- approximation of global diffuse lighting effects

$$L_{\text{ambdiff}} = k_d I_a$$



Lambert'sches Gesetz (Diffuse Reflexion)

[EVC_Skriptum_CG, p.35](#), [EVC_Skriptum_CG, p.36](#)

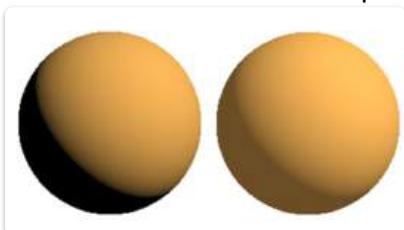
- **Kernaussage:** Die Helligkeit einer diffus reflektierenden Oberfläche ist proportional zum Kosinus des Winkels zwischen der Oberflächennormale und der Richtung zur Lichtquelle. Flacher Lichteinfall führt zu dunkleren Oberflächen.
- **Bedeutung:** Erzeugt den Eindruck räumlicher Form durch Helligkeitsvariationen.
- **Formel für die resultierende Helligkeit (L) durch diffuse Reflexion:**

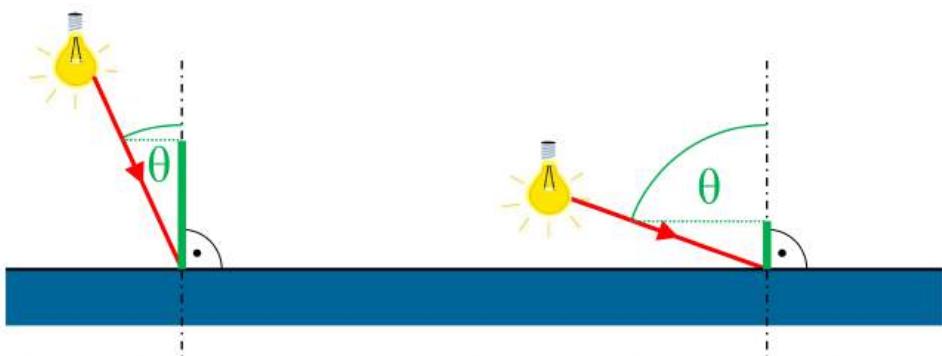
$$L = k_d \cdot I \cdot \cos \theta$$

oder in Vektorform:

$$L = k_d \cdot I \cdot (\mathbf{n} \cdot \mathbf{l})$$

- I : Helligkeit der relevanten Lichtquelle.
- k_d : Diffuser Reflexionskoeffizient der Oberfläche ($0 \leq k_d \leq 1$), gibt den Anteil des einfallenden Lichts an, der diffus reflektiert wird.
- θ : Winkel zwischen der Oberflächennormale (\mathbf{n}) und der Richtung zur Lichtquelle (\mathbf{l}).
- $\mathbf{n} \cdot \mathbf{l}$: Skalarprodukt zwischen dem Normalenvektor und dem Lichtrichtungsvektor.
- **Kombination mit ambientem Licht:**
 - Gesamte Helligkeit = Beitrag durch diffuse Reflexion + Beitrag durch ambientes Licht (I_a).
 - Führt bereits zu einer ansprechenden Darstellung (siehe Beispiel der Kugeln)





$$L = I \cdot \cos \theta$$

when considering
the material:

I ... light source intensity

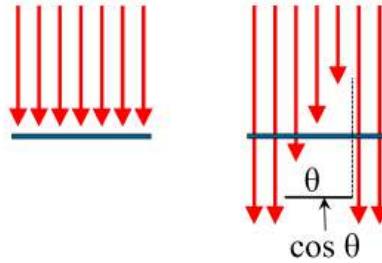
L ... pixel color

k_d ... diffuse coefficient

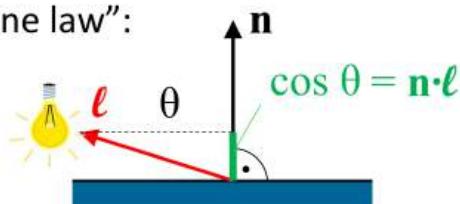
$$L = k_d \cdot I \cdot \cos \theta$$

for ideal diffuse reflectors (Lambertian reflectors)

brightness depends on
orientation of the surface:



"Lambert's cosine law":

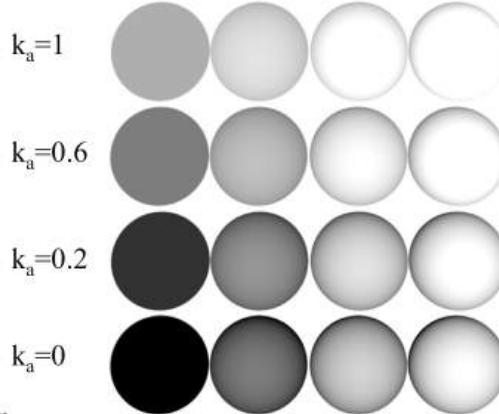


$$L_{\text{diff}} = k_d \cdot I \cdot (\mathbf{n} \cdot \mathbf{l})$$

total diffuse reflection:

$$L_{\text{diff}} = k_a I_a + k_d I(\mathbf{n} \cdot \mathbf{l})$$

$$k_d=0 \quad k_d=0.3 \quad k_d=0.7 \quad k_d=1$$

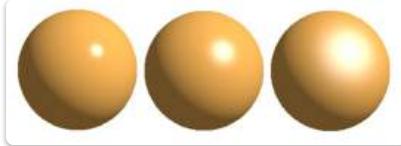


(sometimes extra k_a
for ambient light)

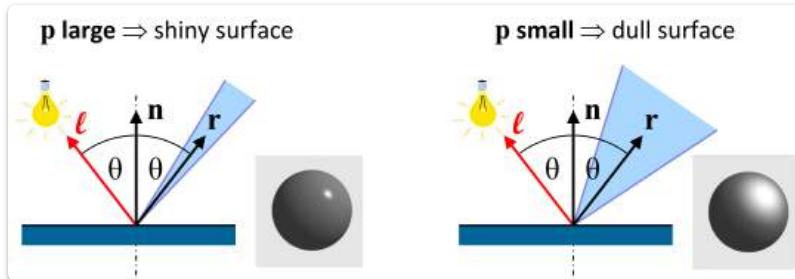
Werner Purgathofer

Glanzpunkte (Specular Highlights)

- Fast jede Oberfläche ist auch etwas spiegelnd. Wenn man diesen Aspekt nicht mitmodelliert, dann wirken alle Materialien gleich stumpf.



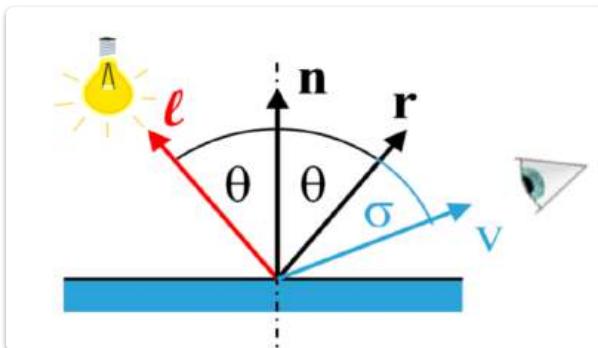
- Da die exakte spiegelnde Reflexion äußerst kompliziert zu berechnen ist, behilft man sich mit einer einfachen Funktion, die einen ähnlichen Verlauf hat wie das Highlight: $\cos^p(\alpha)$.
- Mit dem freien Parameter p lässt sich dabei die „Poliertheit“ der Oberfläche steuern:
 - Je größer p ist, desto kleiner wird der Glanzpunkt und desto glatter wirkt die Oberfläche** (linke Kugel im Bild).
 - Je kleiner p ist, desto matter wirkt die Oberfläche** (rechte Kugel im Bild).



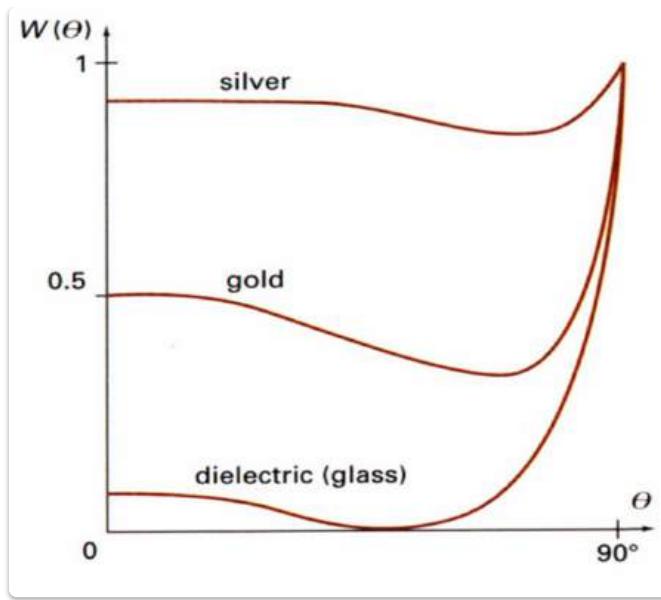
- Um diesen Effekt im richtigen Ausmaß zur Beleuchtung hinzufügen zu können, wird noch ein weiterer Faktor eingeführt, der spiegelnde Reflexionskoeffizient k_s .
- Der Glanz berechnet sich dann nach diesem sogenannten **Phong-Beleuchtungsmodell** so:

$$I_{spec} = k_s * I_L * \cos^p(\alpha)$$

- I_L : Intensität des Lichts.
- α : Winkel zwischen dem exakten Reflexionsstrahl r und der Richtung zum Auge v .
- Etwas näher an der Wahrheit ist die Verwendung des **Fresnel'schen Reflexionsgesetzes**, das beschreibt, dass der Spiegelungsgrad auch vom Lichteinfallswinkel θ abhängt.
- Also ist der Koeffizient k_s eigentlich eine Funktion $W(\theta)$ der Lichteinfallsrichtung l .
- Für die meisten Materialien ist dieser Wert aber fast konstant. Daher wird auf diesen Aufwand verzichtet, wenn man nicht gerade ein Material darstellen will, bei dem der Effekt auffällt.

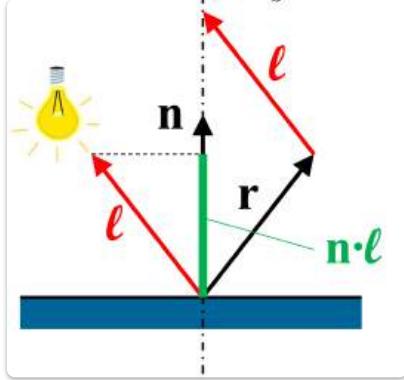


- Das Bild zeigt die Abhängigkeit dieser Funktion $W(\theta)$ vom Winkel zwischen Lichteinfall und Normale auf der Oberfläche für drei verschiedene Materialien (Silber, Gold, dielektrisches Material).



- Bei der Berechnung des Reflexionsvektors r muss man noch bedenken, dass es sich hier um **Vektoren im 3D-Raum** handelt, wobei l (Lichtrichtung), n (Oberflächennormale) und r in einer Ebene liegen müssen und alle Länge eins haben sollen.
 - r ergibt sich zu:

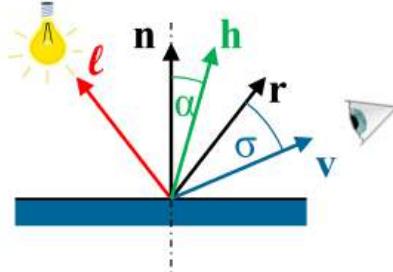
$$r = (2n \cdot l)n - l$$



- Weil die Glanzfunktion sowieso nur eine grobe Näherung ist, verwendet man auch häufig eine einfachere Formel, in der $r \cdot v$ (Winkel zwischen Reflexionsrichtung und Blickrichtung) durch $n \cdot h$ ersetzt wird.
 - h : Halbierungsvektor zwischen l und v .

simplified Phong model with halfway vector \mathbf{h}

$$L_{\text{spec}} = k_s \cdot I \cdot (v \cdot r)^p \quad \rightarrow \quad L_{\text{spec}} = k_s \cdot I \cdot (n \cdot h)^p$$



$$\mathbf{h} = \frac{\ell + \mathbf{v}}{\|\ell + \mathbf{v}\|}$$

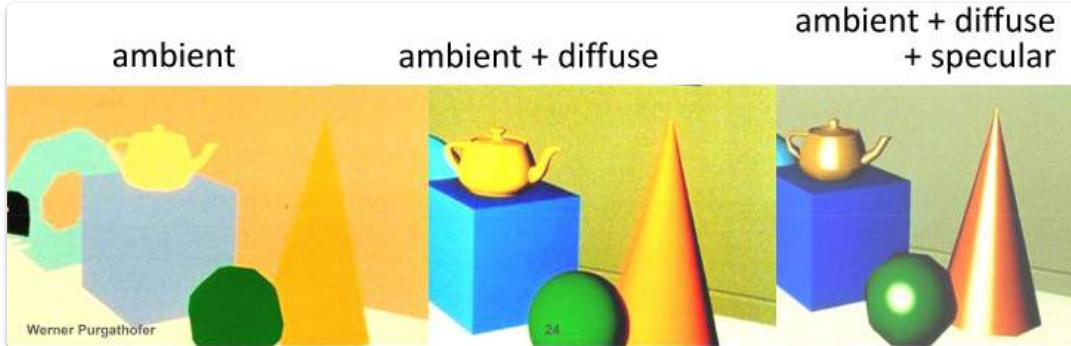
this revised model is called
“Blinn-Phong Shading”

- Der Winkel zwischen n und h ist oft sehr ähnlich dem Winkel zwischen r und v .

- Das resultierende Modell nennt man **Blinn-Phong-Beleuchtungsmodell**.
- Wenn wir alle bisherigen Komponenten zusammensetzen, erhalten wir ein einfaches komplettes Beleuchtungsmodell:

$$L = k_a * I_a + \sum_{i=1,\dots,N} (k_d * I_i * (n \cdot l_i) + k_s * I_i * (n \cdot h_i)^p)$$

- L : Gesamte Beleuchtung.
- k_a : Ambienter Reflexionskoeffizient.
- I_a : Intensität des ambienten Lichts.
- N : Anzahl der Lichtquellen.
- k_d : Diffuser Reflexionskoeffizient.
- I_i : Intensität der i -ten Lichtquelle.
- n : Oberflächennormale.
- l_i : Richtung zur i -ten Lichtquelle.
- k_s : Spekularer Reflexionskoeffizient.
- h_i : Halbierungsvektor zwischen l_i und der Blickrichtung v .
- p : Glanz-Exponent (Polierheit).



- Es gibt noch viele weitere Aspekte, die man berücksichtigen muss, um der Realität näher zu kommen, aber diese werden hier nicht näher beschrieben: Farbverschiebungen in Abhängigkeit der Blickrichtung, Einfluss der Entfernung der Lichtquelle, anisotrope Oberflächen und Lichtquellen, Transparenz, atmosphärische Effekte, Schatten und so weiter.

Schattierung von Polygonen

Flat-Shading

[EVC_Skriptum_CG, p.37](#)

- Beim Schattieren eines Polygons hat klarerweise jeder Punkt die gleichen Oberflächeneigenschaften, vor allem auch den gleichen Normalvektor.
- Beim einfachen Ausfüllen jedes Polygons mit einer Farbe werden die Grenzen zwischen den Polygone deutlich störend erkennbar.
- Der sogenannte **Mach-Band-Effekt**, ein kantenverstärkender Mechanismus des Auges, macht das Problem dabei noch ärger als es ist.

- Dieser Effekt lässt uns Kanten die dunklere Seite dunkler wahrnehmen als sie ist, und die hellere Seite heller als sie ist.
- Die einfachste Lösung dieses Problems ist das Interpolieren der Schattierung zwischen den Polygonen. Dazu sind zwei Verfahren üblich: **Gouraud-Schattierung** und **Phong-Schattierung**.



Gouraud-Schattierung

[EVC_Skriptum_CG](#), p.37

Die Gouraud-Schattierung interpoliert die berechneten Helligkeitswerte über die Polygonflächen. Dazu werden an den Eckpunkten der Polygone Helligkeitswerte berechnet und von diesen aus durch lineare Interpolation jedes Polygon gefüllt.

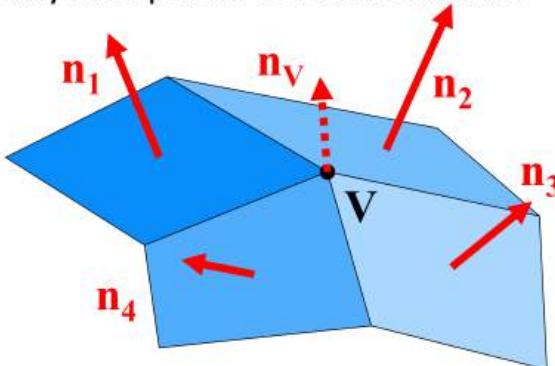


Konkret geht das so:

1. **Berechnung der Eckennormalen:** An jedem Eckpunkt wird eine Normale als Mittelwert der Normalen aller angrenzenden Polygone berechnet. Dies ist natürlich nur ein Näherungswert der Normale der echten zugrundeliegenden Fläche.
2. **Berechnung der Eckpunktintensitäten:** Aus den Eigenschaften der Oberfläche, der Normale (der gemittelten Eckennormalen) und der Lichteinfallsrichtung wird für jeden Eckpunkt ein Helligkeitswert („Schattierung“) berechnet. Beachte, dass dadurch angrenzende Polygone an diesen Eckpunkten alle die gleichen Werte erhalten.

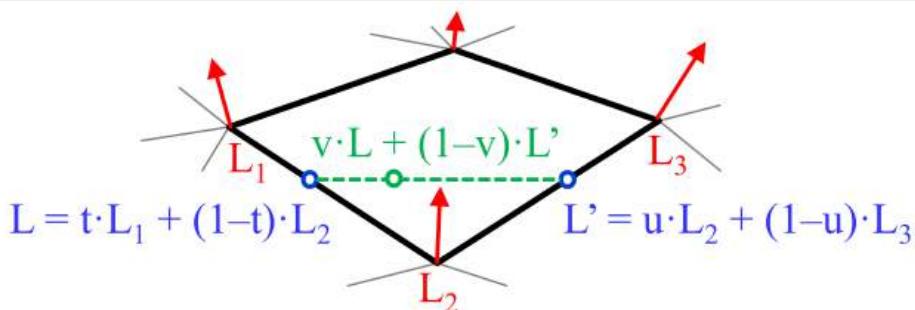
intensity-interpolation:

- determine average unit normal vector at each polygon vertex
- apply illumination model to each vertex
- linearly interpolate vertex intensities



$$\mathbf{n}_v = \frac{\sum_{k=1}^N \mathbf{n}_k}{\left\| \sum_{k=1}^N \mathbf{n}_k \right\|}$$

- Interpolation entlang der Polygonkanten:** Entlang der Polygonkanten werden die Helligkeitswerte linear interpoliert, d.h. es wird für jeden Schnittpunkt mit einer Scanline ein Wert ermittelt. Beachte, dass dadurch für aneinander grenzende Polygone entlang der gemeinsamen Kante die gleichen Werte entstehen.
- Interpolation entlang der Scanlines:** Entlang jeder Scanline wird von der linken bis zur rechten Polygongrenze wieder linear interpoliert. Dadurch haben nebeneinander liegende Pixel immer eine sehr ähnliche Helligkeit und es kommt zu keinen sichtbaren Kanten (im Idealfall).

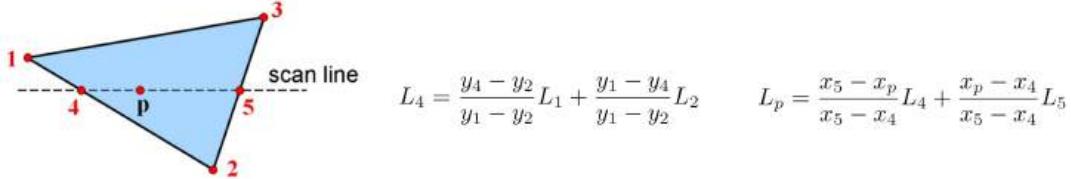


- find normal vectors at corners and calculate shading (intensities) there: L_i
- interpolate intensities along edges linearly: L, L'
- interpolate intensities along scanlines linearly: L_p

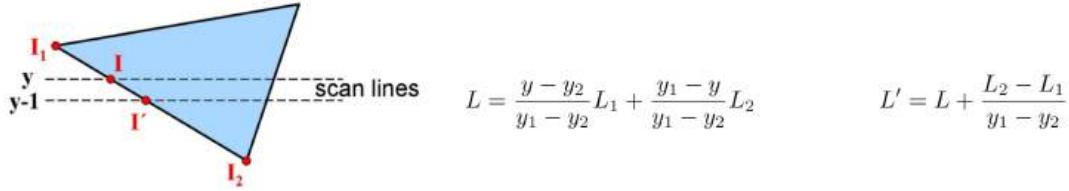
Dennoch verbleiben **Fehlerquellen**. So wird die Silhouette natürlich nicht verändert, dadurch verbleiben störende Polygonkanten sichtbar:



Einfache lineare Interpolation zur Berechnung eines Pixelwertes L_p :



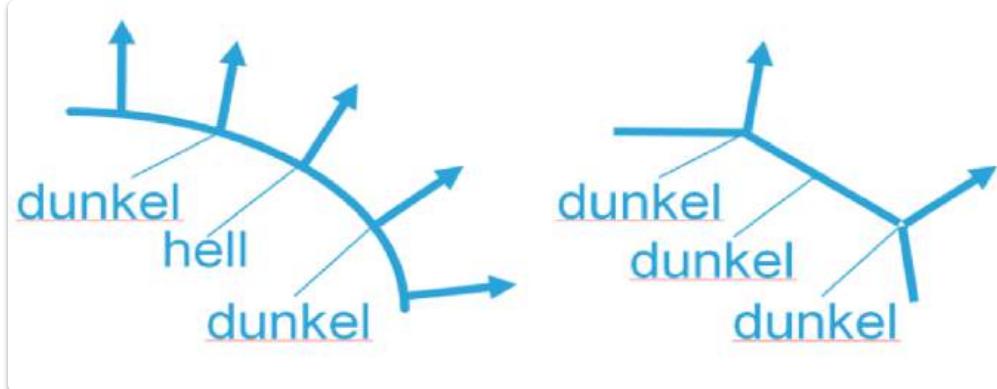
Die *lineare (!) Interpolation der Intensitäten* kann natürlich wieder inkrementell erfolgen, z.B.:



Probleme bei Gouraud-Schattierung (Glanzpunkte)

[EVC_Skriptum_CG, p.37](#)

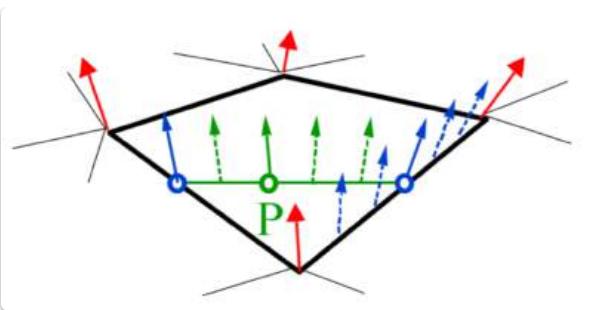
- Zufällige Interpolationsergebnisse bei Glanzpunkten möglich.
- Abhängig davon, ob Eckennormale zufällig Glanzpunkt erzeugt.
- Störend bei bewegten Objekten (Glanzpunkt "wandert").



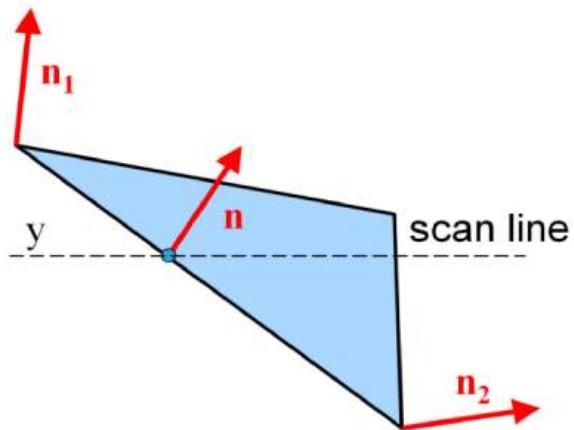
Phong-Schattierung

[EVC_Skriptum_CG, p.38](#)

- Alternative zu Gouraud-Schattierung für konsistenter Glanzpunkte.
- **Ablauf:**
 1. Normalen an Polygon-Eckpunkten berechnen.
 2. Diese Normalen entlang Polygonkanten interpolieren.
 3. Entlang Scanlines interpolierte Normalen weiter interpolieren (pro Pixel).
 4. Pro Pixel mit interpolierter Normalen Helligkeit nach Beleuchtungsmodell berechnen.
- **Unterschied:** Wir drehen Punkt 2 und 3 um. Also ich interpoliere die Vektoren und schattiere dann.
- **Vorteil:** Konsistenter Glanzpunkte.
- **Nachteil:** Höherer Aufwand (Beleuchtung pro Pixel).



Normalvektorinterpolation:



$$\mathbf{n} = \frac{y - y_2}{y_1 - y_2} \mathbf{n}_1 + \frac{y_1 - y}{y_1 - y_2} \mathbf{n}_2$$

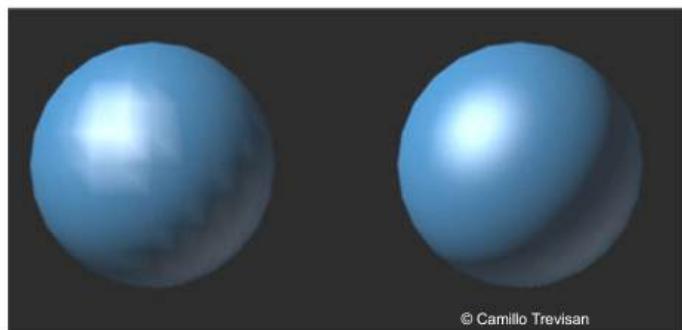
ZUR BEACHTUNG:

- Phong-Beleuchtungsmodell und Phong-Schattierung sind zwei unabhängige Konzepte!

Vergleich zwischen Gouraud und Phong:

comparison to Gouraud shading

- better highlights
- less Mach banding
- more costly
- wrong silhouette stays!



10. Ray-Tracing

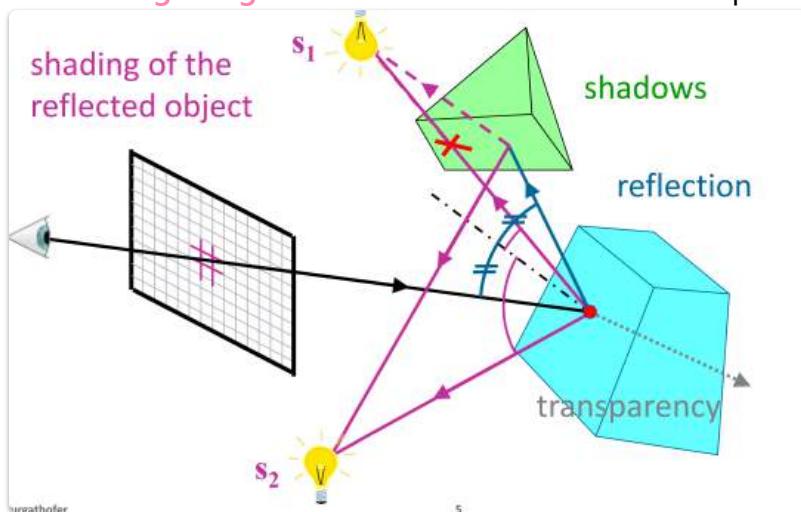
EVC_Skriptum_CG, p.39

- **Ray:** Strahl.
- **to trace:** Eine Spur verfolgen.
- **Ray-Tracing:** "Verfolgen von Strahlspuren".
- **Lichtstrahlen:** Durchlaufen in **verkehrter Richtung** (vom Auge zur Lichtquelle).
- **Aufbauend auf Ray-Casting:** Mächtige Methode zur Simulation wichtiger optischer Effekte:
 - Schattierung
 - Schatten
 - Spiegelbilder
 - Lichtbrechung
- **Einfachheit des Verfahrens:** Ermöglicht Darstellung komplexer Objekte:
 - Freiformflächen
 - Fraktale Oberflächen
 - Mathematische Funktionen aller Art
 - usw.

Das Ray-Tracing Prinzip

EVC_Skriptum_CG, p.39

- **Basisidee:** Licht, das auf einen Bildpunkt trifft, in umgekehrter Richtung verfolgen.
- **Ziel:** Untersuchen, woher das Licht kommt.
- **Schlussfolgerung:** Daraus das Aussehen dieses Bildpunktes (Pixels) bestimmen.



Korrekte Sichtbarkeit und Schattierung

EVC_Skriptum_CG, p.39

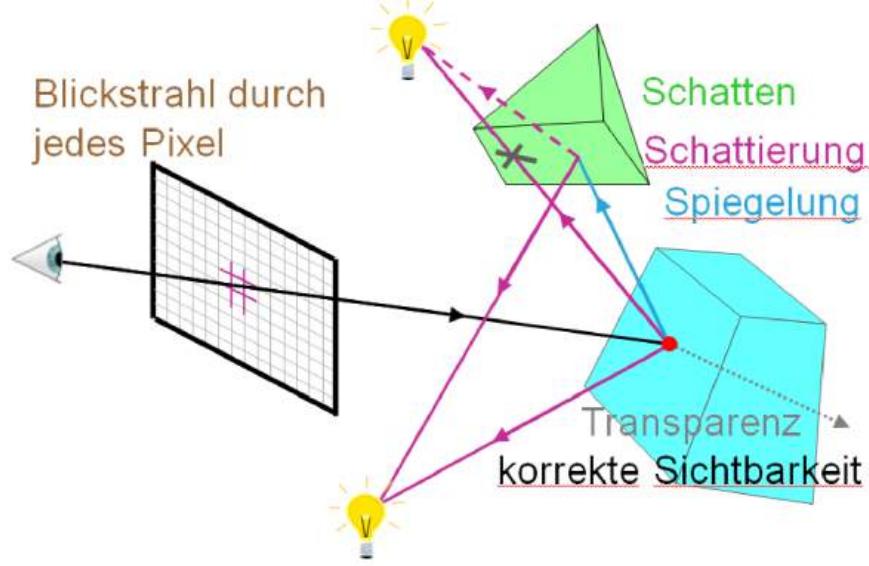
- **Blickstrahl (Primärstrahl):** Durch jeden Bildpunkt legen.
- **Schnitt:** Blickstrahl mit allen Oberflächen der Szene schneiden.
- **Nächster Schnittpunkt:** Denjenigen auswählen, der am nächsten zum Bild liegt.
- **Schattierung:** Schattierung dieses Objektpunktes (aus Blickrichtung) als Pixelwert.
- **Wiederholung:** Für alle Bildpunkte (Pixel).
- **Ergebnis:** Abbildung der Szene mit korrekter Sichtbarkeit.
- **Schattierungsmodell:** Beliebig wählbar (z.B. Phong-Modell).



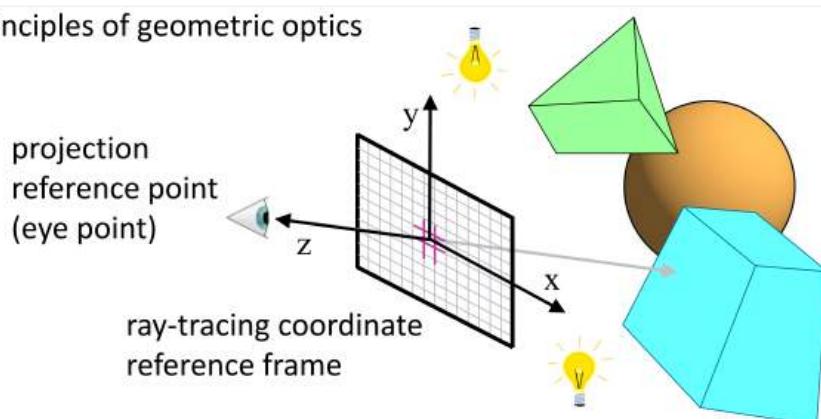
Schatten

EVC_Skriptum_CG, p.39

- **Schattierungsberechnung:** Benötigt Oberflächennormale und Richtungen zu allen Lichtquellen.
- **Direkter Lichteinfluss:** Nur wenn Lichteinfall nicht durch andere Objekte verdeckt ist.
- **Schattenfühler (Sekundärstrahl):** Vom zu schattierenden Punkt zur Lichtquelle legen.
- **Schnittprüfung:** Schattenfühler mit allen Objekten der Szene schneiden.
- **Schattenwurf:** Lichtquelle nicht berücksichtigen, wenn Schnittpunkt zwischen Objekt und Lichtquelle besteht.
- **Ergebnis:** Objektteile im Schatten erhalten weniger Lichteinfluss → automatischer Schattenwurf.



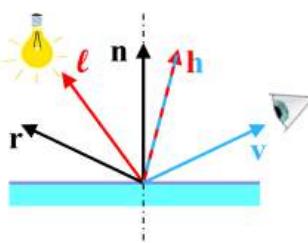
principles of geometric optics



$$\text{primary ray} = \text{eye point} + t \cdot (\text{pixel} - \text{eye point})$$

ambient light $k_a I_a$ shadow ray along ℓ

$$I_d = k_a I_a + k_d(\mathbf{n} \cdot \ell) + k_s(\mathbf{h} \cdot \mathbf{n})^p$$

diffuse reflection $k_d(\mathbf{n} \cdot \ell)$ specular reflection $k_s(\mathbf{h} \cdot \mathbf{n})^p$ 

Spiegelbilder

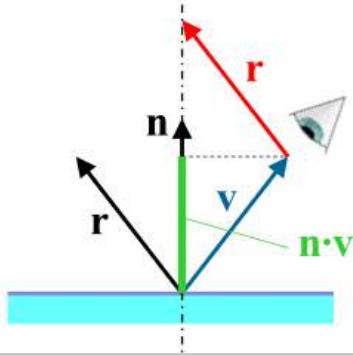
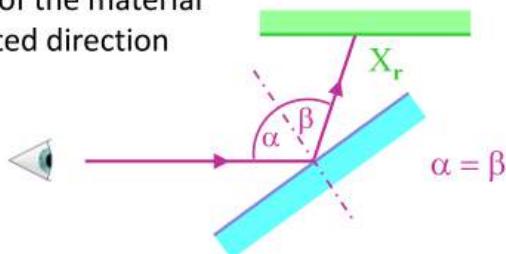
[EVC_Skriptum_CG, p.39](#)

- **Spiegelndes Objekt getroffen:** Nicht das Objekt selbst sichtbar, sondern das, was in Spiegelungsrichtung sichtbar ist.
- **Reflexionsgesetz:** Einfallswinkel = Ausfallwinkel (Symmetrie).
- **Reflexionsstrahl (Sekundärstrahl):** Blickstrahl an der Oberfläche spiegeln und in Spiegelungsrichtung verfolgen.

- **Schnitt:** Reflexionsstrahl mit allen Objekten schneiden.
- **Nächster Schnittpunkt:** Auswählen.
- **Farbe/Schattierung:** Schattierung dieses weiteren Auftreffpunktes (aus Richtung des Reflexionsstrahls) ist die Farbe, die der ursprüngliche Blickstrahl sieht.
- **Lokale Berechnung:** Reflexionsverhalten wird lokal berechnet → einfache Erzeugung gekrümmter Spiegel.

$$I_r = k_r \cdot X_r$$

I_r ... illumination caused by reflection
 k_r ... reflection coefficient of the material
 X_r ... shading in the reflected direction



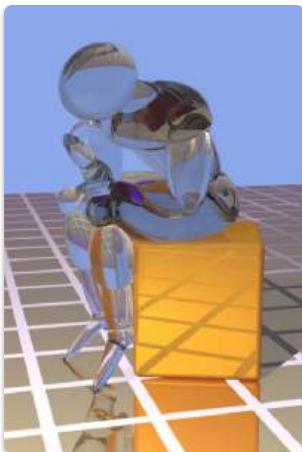
$$r + v = 2(\mathbf{n} \cdot \mathbf{v})\mathbf{n}$$

$$r = 2(\mathbf{n} \cdot \mathbf{v})\mathbf{n} - v$$

Transparenz

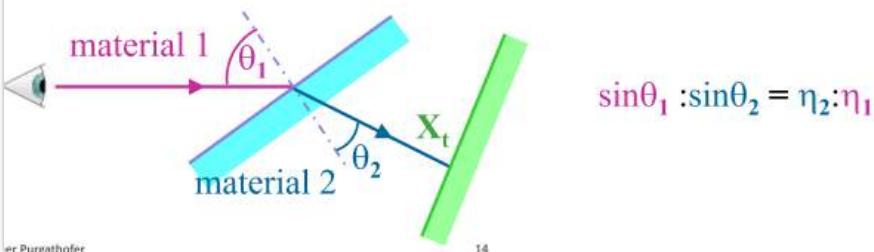
EVC_Skriptum_CG, p.39, p.40

- **Transparentes Objekt getroffen:** Man sieht, was der durch das Objekt verlaufende Transparenzstrahl trifft.
- **Transparenzstrahl (Sekundärstrahl):** Vom Auftreffpunkt durch das transparente Objekt verfolgen.
- **Brechungsgesetz:** Richtung des Transparenzstrahls so legen, dass das Material das Licht bricht.
- **Schnitt:** Transparenzstrahl mit allen Objekten schneiden.
- **Nächster Schnittpunkt:** Auswählen.
- **Farbe/Schattierung:** Schattierung dieses weiteren Auftreffpunktes (aus Richtung des Transparenzstrahls) ist, was der ursprüngliche Blickstrahl sieht.



$$I_t = k_t \cdot X_t$$

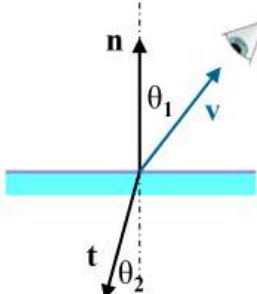
I_t ... illumination caused by transparency
 k_t ... transparency coefficient of the material
 X_t ... shading in the transparency direction



calculation of transparency ray

$$\frac{\sin \theta_2}{\sin \theta_1} = \frac{n_1}{n_2} \quad \sin \theta_2 = \frac{n_1}{n_2} \sin \theta_1$$

$$t = -\frac{n_1}{n_2} v - (\cos \theta_2 - \frac{n_1}{n_2} \cos \theta_1) n$$



Rekursion

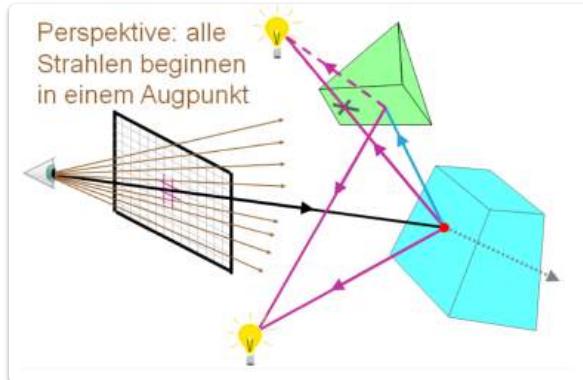
EVC_Skriptum_CG, p.40

- **Gleichwertigkeit der Strahlen:** Jeder Strahl (außer Schattenfühler) ist gleichwertig (Primär- oder Sekundärstrahl).
- **Aktion am Auftreffpunkt:** Unabhängig davon, ob Primär- oder Sekundärstrahl.
- **Ermöglicht:**
 - Mehrfachspiegelungen.
 - Spiegelungen hinter transparentem Material.
 - Usw.

Perspektive

EVC_Skriptum_CG, p.40

- **Erzeugung primärer Blickstrahlen:** Bestimmt die Abbildung der Szene auf die Bildebene.
- **Parallele Strahlen (normal zur Bildebene):** Orthogonale Parallelprojektion.
- **Strahlen von fiktivem Augpunkt:** Perspektivische Projektion (natürliche Entstehung ohne Mehraufwand).



Ray-Tracing Implementierung

[EVC_Skriptum_CG](#), p.40

Einen Ray-Tracer zu schreiben ist also ganz einfach. Man braucht eine Funktion, die eine Gerade mit allen Objekten schneidet und den vordersten Schnittpunkt zurückliefert.

Ray-Tracing Pseudocode:

```

FOR alle Pixel  $p_0$  DO
  1. lege Blickstrahl vom Auge  $e$  aus durch  $p_0$ ,
     schneide mit allen Objekten und wähle den nähesten Schnittpunkt  $p$ 
  2. FOR alle Lichtquellen  $s$  DO
      schneide Schattenführer  $p \rightarrow s$  mit allen Objekten
      IF kein Schnittpunkt zwischen  $p$ ,  $s$  THEN Schattierung += Einfluss von  $s$ 
  3. IF Oberfläche von  $p$  ist spiegelnd
      THEN verfolge Sekundärstrahl; Schattierung += Einfluss der Reflexion
  4. IF Oberfläche von  $p$  ist transparent
      THEN verfolge Sekundärstrahl; Schattierung += Einfluss der Transparenz
    
```

Viewing-Koordinatensystem und Strahldarstellung

- **Viewing-Koordinatensystem (Standard):**
 - xy-Ebene = Bildebene.
 - Hauptblickrichtung = negative z-Achse.
- **Strahlen (parametrisierte Form):**

$$p(t) = p_0 + t \cdot d$$

- p_0 : Startpunkt.
- t : Parameter.
- d : Richtungsvektor.
- **Primärstrahlen:**

- e : Augpunkt.
- p_0 : Pixelkoordinate.
- Schattenfühler:

$$p + t \cdot (s - p)$$

- p : Oberflächenpunkt.
- s : Lichtquellenposition.
- Reflexionsstrahlen:

$$p + t \cdot r$$

- r : Reflexionsrichtung des Blickstrahls v .

$$r = (2n \cdot v)n - v \text{ (aus Reflexionsgesetz)}$$

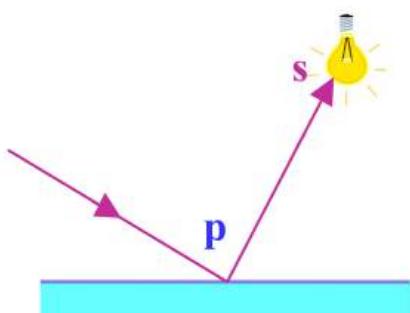
- n : Oberflächennormale.
- v : Richtung des einfallenden Strahls.
- $|r| = 1$ (garantiert durch Berechnung).

ray = intersection point + $t \cdot$ vector to light source

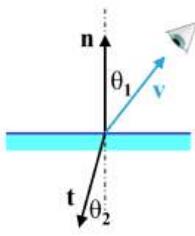
$$\text{ray} = \mathbf{p} + t \cdot (\mathbf{s} - \mathbf{p})$$

p ... intersection point

s ... light source position



a light source influences the result only if
there is no intersection with $0 < t < 1$

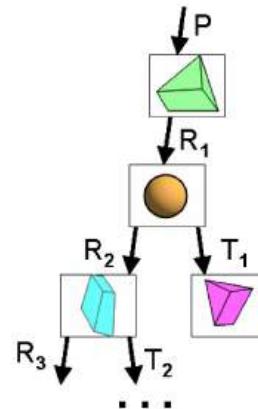
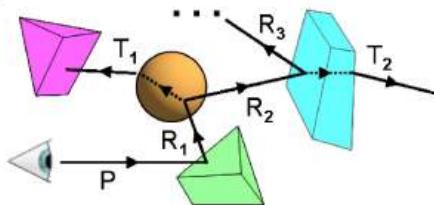


Transparenzstrahlen sind $p + t \cdot t$, wobei t sich aus dem Snellius'schen Brechungsgesetz $\sin \theta_1 : \sin \theta_2 = \eta_2 : \eta_1$ berechnet (η_i ist der Brechungsindex des Materials i):

$$t = -\frac{\eta_1}{\eta_2}v - (\cos \theta_2 - \frac{\eta_1}{\eta_2} \cos \theta_1)n$$

Auch der Vektor t hat wieder die Länge 1 (siehe linke Skizze).

Wenn man nun solcherart die Lichtstrahlen in verkehrter Richtung „verfolgt“, so entsteht eine **rekursive Aufruffolge** (siehe Skizze unten), die einem Strahlenbaum entspricht (rechte Skizze). Normalerweise wird dieser Baum aber nicht in dieser Form gespeichert, sondern ist nur eine symbolische Darstellung der Rekursionsaufruffolge.



Schnitte zwischen Strahlen und Objekten (Ray-Tracing)

[EVC_Skriptum_CG, p.41](#)

- **Bedingungen für darzustellende Objekte:**
 - Schnittpunkt mit Gerade muss berechenbar sein.
 - Oberflächennormale am Schnittpunkt muss bekannt sein.
 - Materialeigenschaften am Schnittpunkt müssen vorhanden sein.
- **Erfüllung der Bedingungen:**
 - BReps (Boundary Representations): Einfach.
 - CSG-Bäume (Constructive Solid Geometry): Durch rekursive Evaluation.
 - Viele andere Datenformate (z.B. Freiformflächen).
- **Notwendigkeit:** Funktionen zur Schnittberechnung mit Strahl für jede Primitivart.
- **Beispiele (folgen):**
 - Kugel
 - Polygon

Schnitt Strahl-Kugel

[EVC_Skriptum_CG, p.41](#)

Kugelgleichung: $|p - c|^2 - R^2 = 0$

In diese setzt man den Strahl ein: $|(e + td) - c|^2 - R^2 = 0$

Dann wird zur besseren Lesbarkeit Δp eingeführt: $\Delta p = c - e$

Und man erhält eine quadratische Gleichung in t : $t^2 - 2(d \cdot \Delta p)t + (|\Delta p|^2 - R^2) = 0$

Die 2 Lösungen entsprechen den beiden Schnittpunkten mit der Kugel:

$$t = d \cdot \Delta p \pm \sqrt{(d \cdot \Delta p)^2 - |\Delta p|^2 + R^2}$$

In Fällen, wo $R^2 \ll |\Delta p|^2$ ist (das ist durchaus häufig), entstehen in dieser Formel Rundungsfehler. Um diese zu vermeiden, kann man $d^2 = 1$ ausnutzen und die Formel umformen, so dass Rundungsfehler unwahrscheinlicher werden:

$$t = d \cdot \Delta p \pm \sqrt{|\Delta p|^2 - (d \cdot \Delta p)^2 - R^2}$$

ray equation:

$$\mathbf{p}(t) = \mathbf{p}_0 + t \cdot \mathbf{d}$$

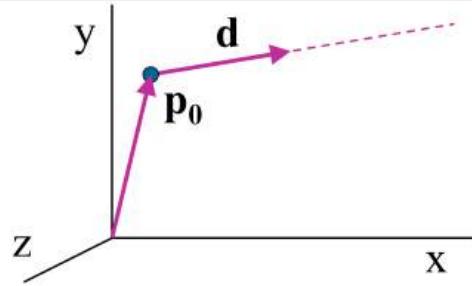
for primary rays:

$$\mathbf{d} = \frac{\mathbf{p}_0 - \mathbf{e}}{|\mathbf{p}_0 - \mathbf{e}|}$$

for secondary rays:

$$\mathbf{d} = \mathbf{r}$$

$$\mathbf{d} = \mathbf{t}$$



\mathbf{p}_0 ... initial-position vector
 \mathbf{d} ... unit direction vector
 \mathbf{e} ... eye-point vector
 \mathbf{r} ... reflection vector
 \mathbf{t} ... transparency vector

parametric ray equation
 inserted into sphere equation

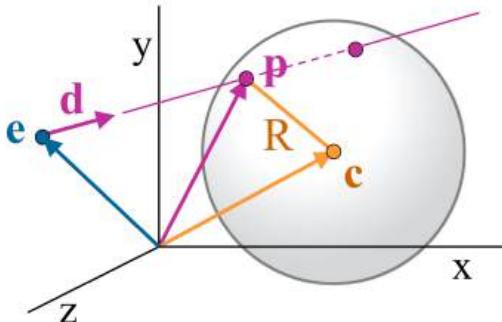
$$|\mathbf{p} - \mathbf{c}|^2 - R^2 = 0$$

$$|(\mathbf{e} + t\mathbf{d}) - \mathbf{c}|^2 - R^2 = 0$$

$$\Delta \mathbf{p} = \mathbf{c} - \mathbf{e}$$

$$t^2 - 2(\mathbf{d} \cdot \Delta \mathbf{p})t + (|\Delta \mathbf{p}|^2 - R^2) = 0 \quad (\mathbf{d}^2 = 1)$$

$$t = \mathbf{d} \cdot \Delta \mathbf{p} \pm \sqrt{(\mathbf{d} \cdot \Delta \mathbf{p})^2 - |\Delta \mathbf{p}|^2 + R^2}$$



if discriminant is negative \Rightarrow no intersections

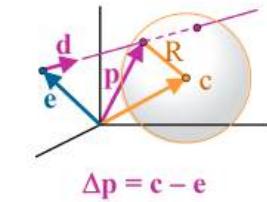
$$t = \mathbf{d} \cdot \Delta \mathbf{p} \pm \sqrt{(\mathbf{d} \cdot \Delta \mathbf{p})^2 - |\Delta \mathbf{p}|^2 + R^2}$$

$$\Rightarrow t = \mathbf{d} \cdot \Delta \mathbf{p} \pm \sqrt{R^2 - |\Delta \mathbf{p} - (\mathbf{d} \cdot \Delta \mathbf{p})\mathbf{d}|^2}$$

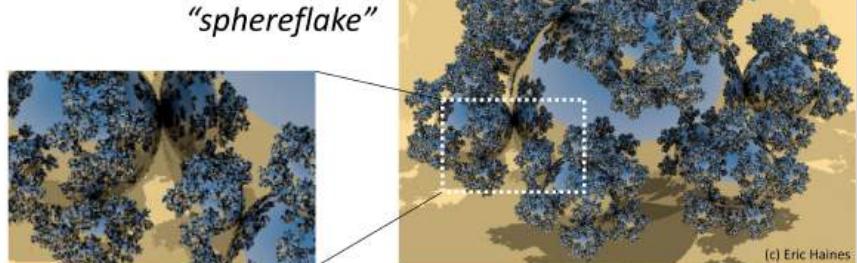
$$|\Delta \mathbf{p} - (\mathbf{d} \cdot \Delta \mathbf{p})\mathbf{d}|^2 = |\Delta \mathbf{p}|^2 - 2\mathbf{d}^2|\Delta \mathbf{p}|^2 + (\mathbf{d} \cdot \Delta \mathbf{p})^2\mathbf{d}^2$$

$$\mathbf{d}^2 = 1 \quad \mathbf{d}^2 = 1$$

(to avoid roundoff errors
when $R^2 \ll |\Delta \mathbf{p}|^2$)

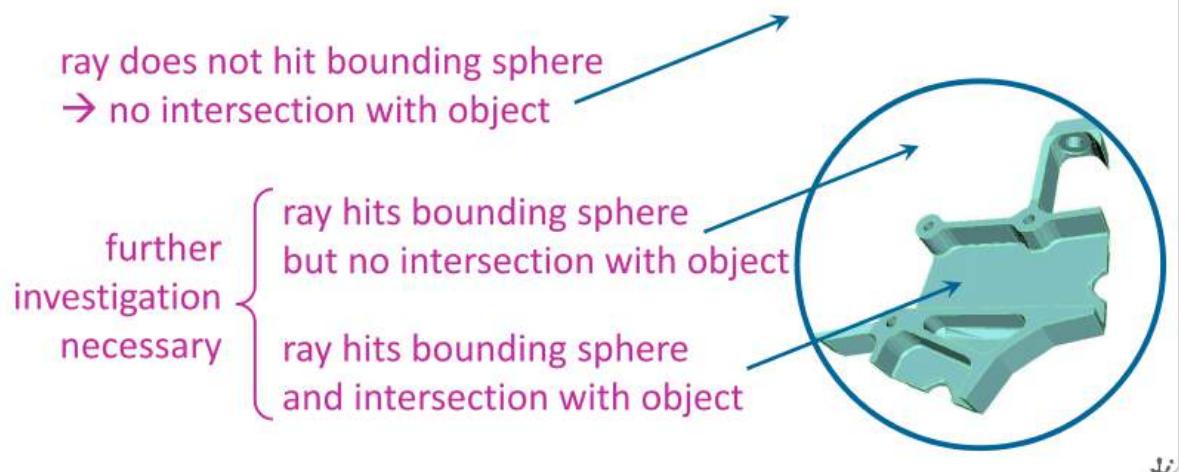


Werner Purgathofer



(c) Eric Haines

use **bounding sphere** to eliminate easy cases



Schnitt Strahl-Polygon

[EVC_Skriptum_CG](#), p.41, p.42

- **Ziel:** Schnittpunkt eines Strahls mit einem Polygon bestimmen.
- **Schritt 1: Backface Detection**
 - Überprüfen, ob das Polygon in die richtige Richtung schaut. (siehe Backface Detection)
- **Schritt 2: Strahlengleichung**
 - Strahl definiert als: $\mathbf{p} = \mathbf{p}_0 + t \cdot \mathbf{d}$
 - \mathbf{p} : Punkt auf dem Strahl
 - \mathbf{p}_0 : Ursprung des Strahls
 - \mathbf{d} : Richtung des Strahls
 - t : Parameter entlang des Strahls
- **Schritt 3: Ebenengleichung des Polygons**
 - Form: $Ax + By + Cz + D = 0$

- Kann auch in Normalenform geschrieben werden: $\mathbf{n} \cdot \mathbf{p} = -D$
 - $\mathbf{n} = (A, B, C)$: Normalenvektor der Ebene
- **Schritt 4: Schnittpunkt mit der Ebene berechnen**
 - Setze die Strahlengleichung in die Ebenengleichung ein:
 - $\mathbf{n} \cdot (\mathbf{p}_0 + t \cdot \mathbf{d}) = -D$
 - $\mathbf{n} \cdot \mathbf{p}_0 + t(\mathbf{n} \cdot \mathbf{d}) = -D$
 - Löse nach t auf:
 - $t(\mathbf{n} \cdot \mathbf{d}) = -(D + \mathbf{n} \cdot \mathbf{p}_0)$
 - $t = -\frac{D + \mathbf{n} \cdot \mathbf{p}_0}{\mathbf{n} \cdot \mathbf{d}}$
- **Schritt 5: Überprüfung des Schnittpunkts**
 - Liegt der Schnittpunkt innerhalb des Polygons?
 - Oder neben dem Polygon?
 - Kann durch Projektion auf eine Hauptebene in 2D überprüft werden.

1. use bounding sphere to eliminate easy cases

2. locate front faces $\mathbf{d} \cdot \mathbf{n} < 0$

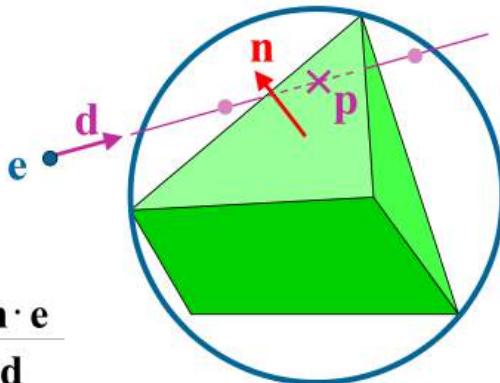
3. solve plane equation

$$Ax + By + Cz + D = 0$$

$$\mathbf{n} = (A, B, C)$$

$$\mathbf{n} \cdot \mathbf{p} = -D$$

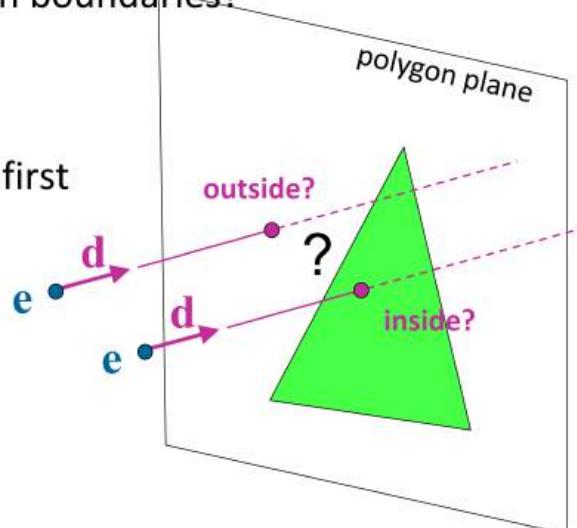
$$\mathbf{n} \cdot (\mathbf{e} + t\mathbf{d}) = -D \quad \Rightarrow \quad t = -\frac{D + \mathbf{n} \cdot \mathbf{e}}{\mathbf{n} \cdot \mathbf{d}}$$



4. intersection point inside polygon boundaries?

perform inside-outside test

→ smallest t to an inside point is first intersection point of polyhedron



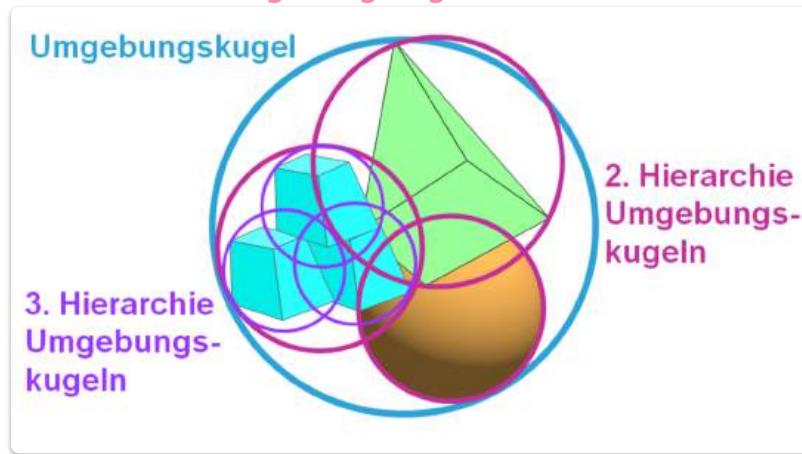
Ray-Tracing Beschleunigung

- **Problem:** Ray-Tracing ist rechenintensiv.
 - Beispiel: Szene mit 1000 Polygonen auf 1000x1000 Pixel Fläche $\Rightarrow 10^9$ Schnittberechnungen (nur für Primärstrahlen, ohne Optimierungen).
- **Notwendigkeit:** Signifikante Beschleunigung des Verfahrens ist erforderlich.
- **Wichtigste Methode:** Reduktion der Anzahl notwendiger Schnittberechnungen.
 - Ansatz: Ausnutzung von Kohärenz.

Objektumgebungen

EVC_Skriptum_CG, p.42

- **Problem:** Direkter Schnitt komplexer Objekte mit Strahlen ist ineffizient.
- **Idee:** Umgebungen (Bounding Volumes) für Objekte definieren, um schnelle Vorabtests durchzuführen.
- **Umgebungskugeln (Bounding Spheres)**
 - Einfache geometrische Form (Kugel).
 - Umschließt das gesamte Objekt.
 - **Vorteil:** Schnelle Schnittprüfung mit dem Strahl.
 - **Funktionsweise:**
 - Trifft der Strahl die Umgebungskugel?
 - **Nein:** Dann trifft er auch nicht das eingeschlossene Objekt \Rightarrow keine detaillierte Schnittberechnung notwendig (Performancegewinn).
 - **Ja:** Detaillierte Schnittprüfung mit dem Objekt erforderlich.
- **Hierarchische Umgebungskugeln**



- Komplexe Objekte können hierarchisch in kleinere Teillojekte mit eigenen Umgebungskugeln unterteilt werden.
- **Vorteil:** Verfeinerte Tests ermöglichen früheres Ausschließen von irrelevanten Teilen der Szene.
- **Prinzip:**
 1. Große Umgebungskugel für das gesamte Objekt.
 2. Unterteilung in kleinere Gruppen mit eigenen Umgebungskugeln (2. Hierarchie).
 3. Weiter Unterteilung bis zu einfachen Objekten (3. Hierarchie).

- **Komplexität:**

- Ohne Umgebungskugeln: $O(n)$ Schnittversuche (wobei n die Anzahl der Objekte/Teile ist).
- Mit hierarchischen Umgebungskugeln: Reduktion auf etwa $O(\log n)$.

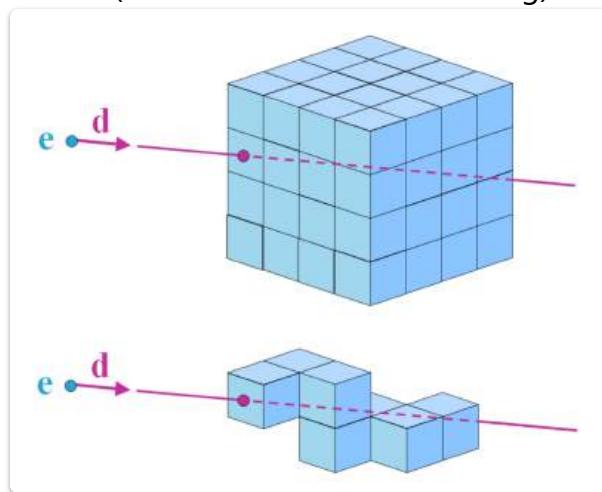
- **Alternative Objektumgebungen:**

- Statt Umgebungskugeln können auch andere Formen verwendet werden, z.B. Umgebungsquader (Bounding Boxes).
- **Abwägung:** Mehraufwand für komplexere Formen vs. genauere Umschließung (engeres Anliegen) und potenzieller Gewinn bei der Schnittprüfung.

Raumteilungs-Methoden

EVC_Skriptum_CG, p.42

- **Alternative zu Objektumgebungen:** Aufteilung des gesamten Raumes, in dem sich die Szene befindet.
 - Unabhängig von der Objektverteilung.
 - **Methoden:**
 - Regelmäßiges Raster (Array von Würfeln/Voxeln).
 - Octree (hierarchische Raumauftteilung).



- **Vorteile:**

- Man muss nur die Objekte in den Teilräumen betrachten, durch die der Strahl geht.
- Schnelle Berechnung des nächsten Teilraums auf dem Strahlpfad.
- Sobald ein Schnittpunkt innerhalb eines Teilwürfels gefunden wurde, kann die Suche beendet werden.

- **Algorithmen:** Ähnlich dem 3D-Bresenham-Verfahren zur schnellen Durchquerung der Teilräume.

- **Vorgehensweise für einzelne Teilwürfel:**

- Strahlgleichung: $\mathbf{p}(t) = \mathbf{p}_0 + t \cdot \mathbf{d}$
- Eintrittspunkt \mathbf{p}_{in} des Strahls in den Teilwürfel.
- Normalenvektoren der Würfelflächen sind $(\pm 1, 0, 0), (0, \pm 1, 0), (0, 0, \pm 1)$.

- Für drei Flächen mit $\mathbf{d} \cdot \mathbf{n} > 0$ (d.h. der Strahl bewegt sich auf die Fläche zu), bestimmt man den Schnittpunkt mit dem Strahl und wählt den vordersten Schnittpunkt (kleinstes t) aus.
- Diese Methode funktioniert auch, wenn die Würfel unterschiedlich groß sind (z.B. in einem Octree).
- Berechnung des Austrittspunkts und des zugehörigen t -Wertes:
 - Austrittspunkt: $\mathbf{p}_{out} = \mathbf{p}_{in} + t_k \mathbf{d}$
 - Ebene der Austrittsfläche: $\mathbf{n}_k \cdot \mathbf{p} = -D_k$
 - Da \mathbf{p}_{out} auf der Ebene liegt: $\mathbf{n}_k \cdot \mathbf{p}_{out} = -D_k$
 - Einsetzen der Gleichung für \mathbf{p}_{out} : $\mathbf{n}_k \cdot (\mathbf{p}_{in} + t_k \mathbf{d}) = -D_k$
 - Auflösen nach t_k :
 - $\mathbf{n}_k \cdot \mathbf{p}_{in} + t_k(\mathbf{n}_k \cdot \mathbf{d}) = -D_k$
 - $t_k(\mathbf{n}_k \cdot \mathbf{d}) = -D_k - \mathbf{n}_k \cdot \mathbf{p}_{in}$
 - $t_k = \frac{-D_k - \mathbf{n}_k \cdot \mathbf{p}_{in}}{\mathbf{n}_k \cdot \mathbf{d}}$

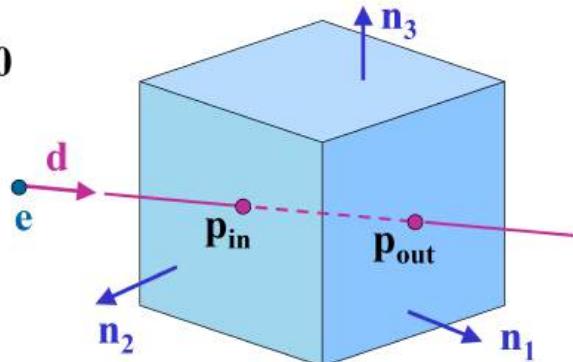
Slides:

ray direction \mathbf{d} & ray entry position \mathbf{p}_{in}

→ potential exit faces $\mathbf{d} \cdot \mathbf{n}_k > 0$

→ normal vectors

$$\mathbf{n}_k = \begin{cases} (\pm 1, 0, 0) \\ (0, \pm 1, 0) \\ (0, 0, \pm 1) \end{cases}$$



→ check signs of components of \mathbf{d}

calculation of exit positions, select smallest t_k

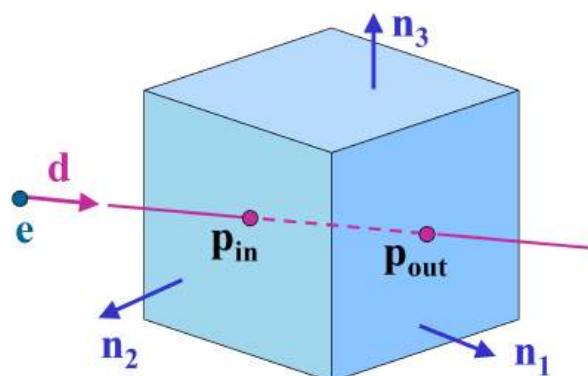
$$\mathbf{p}_{out,k} = \mathbf{p}_{in} + t_k \mathbf{d}$$

$$\mathbf{n}_k \cdot \mathbf{p}_{out,k} = -D_k$$

$$t_k = \frac{-D_k - \mathbf{n}_k \cdot \mathbf{p}_{in}}{\mathbf{n}_k \cdot \mathbf{d}}$$

example: $\mathbf{n}_k = (1, 0, 0)$

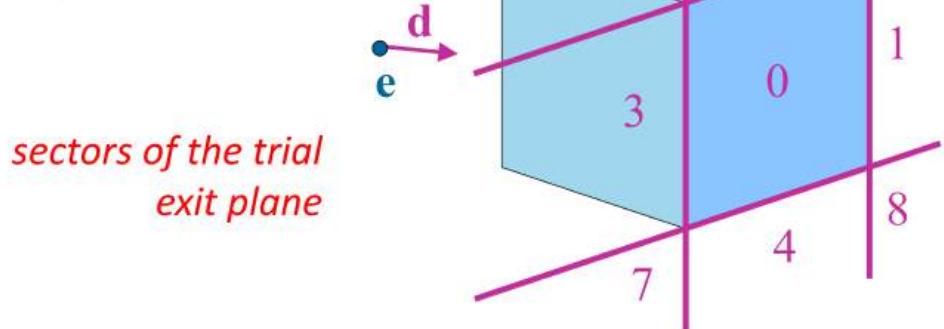
$$x_k = -D_k \Rightarrow t_k = \frac{x_k - x_0}{x_d}$$



variation: trial exit plane

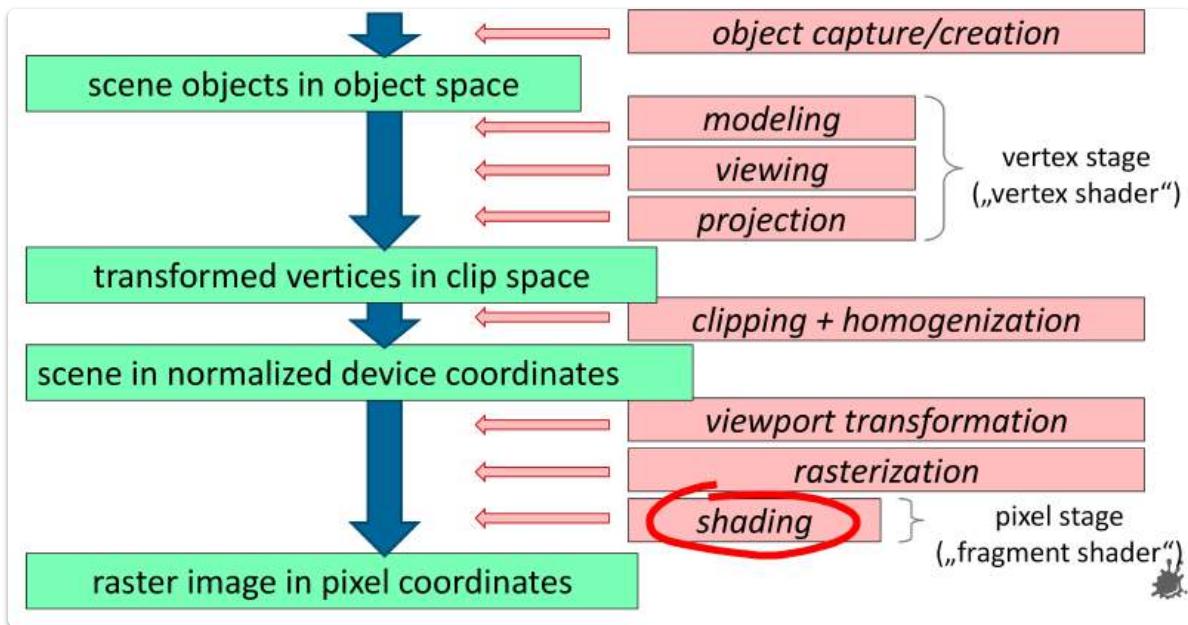
perpendicular to largest component of \mathbf{d}

- (a) exit point in 0 \Rightarrow done
- (b) $\{1, 2, 3, 4\} \Rightarrow$ side clear
- (c) $\{5, 6, 7, 8\} \Rightarrow$ extra calc.



11. Globale Beleuchtung und Texturen

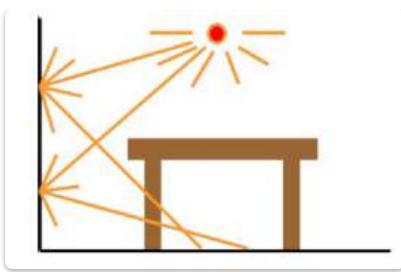
Rendering Pipeline



Radiosity

EVC_Skriptum_CG, p.43

- **Ursprung**: Wärmeübertragung.
- **Modellierung**: Lichtausbreitung unter Beachtung des Energiegleichgewichts in einem geschlossenen System.
- **Beschreibung**: Physikalischer Vorgang der Ausbreitung von Licht in einer diffus reflektierenden Umgebung.
- **Ziel**: Berechnung der Helligkeit aller Flächen einer Szene unter Berücksichtigung der gegenseitigen Beeinflussung.
- **Effekt**: Auch Flächen, die nicht direkt beleuchtet sind, erhalten eine gewisse Helligkeit (indirekte Beleuchtung).
- **Sekundäre Lichtquellen**: Jeder beleuchtete Gegenstand wirkt als sekundäre Lichtquelle und strahlt Licht in die Umgebung ab.
- **Bildgenerierung**:
 - Zuerst wird die Lichtausbreitung im Raum berechnet, **ohne** dass die Kameraposition bekannt ist.
 - Vereinfachende Annahme: Der Beobachter beeinflusst die Ausbreitung des Lichts nicht.
 - **Vorteil**: Objekte können dann aus verschiedenen Richtungen dargestellt werden, ohne dass die Lichtausbreitung jedes Mal neu berechnet werden muss.



Die Radiosity Gleichung

EVC_Skriptum_CG, p.43

- **Szenenbeschreibung:** Besteht aus n ebenen Polygonen, beim Radiosity-Verfahren als Patches bezeichnet.
- **Patch-Eigenschaften:**
 - Jedes Patch P_i ist homogen.
 - Perfekt diffuse Oberfläche (Licht wird in alle Richtungen gleichmäßig abgestrahlt).
 - Lichtquellen sind ebenfalls Patches.
- **Radiosity B_i von Patch P_i :**
 - Gesamte abgestrahlte Energie pro Flächeneinheit.
 - Summe aus Eigenemission und reflektierter Leistung pro Flächeneinheit.
 - Lichtenergiedichte ist proportional zur wahrgenommenen Helligkeit.
- **Vereinfachende Annahme:** Radiosity ist für alle Positionen auf einem Patch gleich.
- **Die Radiosity Gleichung:**

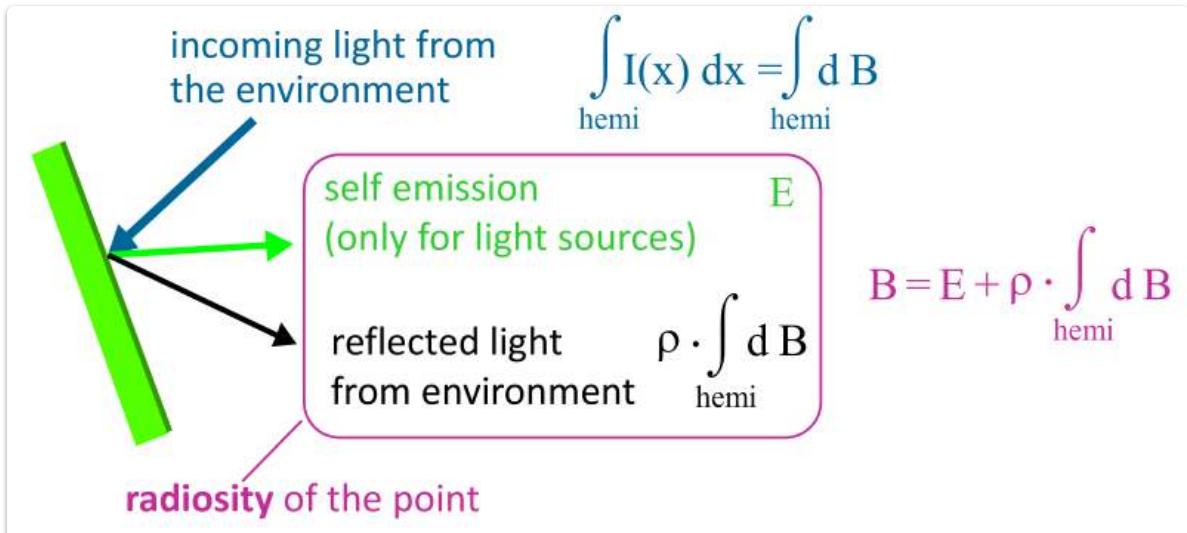
$$B_i = E_i + \rho_i \sum_{j=1}^n B_j F_{ij}$$

- B_i : Radiosity des Patches i .
- E_i : Eigenemission des Patches i .
- ρ_i : Diffuser Reflexionskoeffizient der Oberfläche von Patch i (gibt an, wie viel % des einfallenden Lichts diffus reflektiert wird, auch Albedo genannt).
- n : Anzahl der Patches in der Szene.
- B_j : Radiosity aller anderen Patches j .
- F_{ij} : Formfaktor (view factor) zwischen Patch i und Patch j .
 - Gibt an, welcher Anteil der Radiosity von Patch j auf Patch i wirkt (ist gleich dem Anteil der Radiosity von i , die auf j trifft, aufgrund des Reziprozitätsgesetzes).
 - Geometrische Größe, unabhängig von Lichtquellen oder Radiositywerten.
- **Gleichungssystem:** Die Radiosity-Formeln für n Patches ergeben ein lineares Gleichungssystem mit n Unbekannten B_i . (kann nur iterativ gelöst werden)
- **Matrix-Form des Gleichungssystems:**

$$B_i - \rho_i \sum_{j \neq i} B_j F_{ij} = E_i$$

$$\begin{pmatrix} 1 & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1 & \cdots & -\rho_2 F_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \cdots & 1 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix}$$

Herleitung der Gleichung



to calculate the light influence between surfaces

Radiosity = total light leaving a surface point

$$B = E + \rho \cdot \int_{hemi} d B$$

B ... radiosity hemi ... half space over point

E ... self emission ρ ... reflection coefficient

“radiosity = self emission + reflection property · sum of all incoming light”

- diffuse interreflections in a scene
- radiant energy transfers
- conservation of energy, closed environments
- subdivision of scene into *patches* with constant radiosity B_i

$$B = E + \rho \cdot \int_{\text{hemi}} d B$$

the scene is discretized into **n "patches"** (plane polygons) P_i , for each of these patches a constant radiosity B_i is assumed:

$$B = E + \rho \cdot \int_{\text{hemi}} d B \quad \Rightarrow \quad B_i = E_i + \rho_i \cdot \sum_{j=1}^n B_j \cdot F_{ij}$$

ρ_i diffuse reflection coefficient of patch **i**

F_{ij} “form factor”: describes what % of the influence on patch **i** comes from patch **j**;
= geometric size !

$$B_i = E_i + \rho_i \sum_{j \neq i} B_j F_{ij}$$

B_i ... radiosity of patch **i**

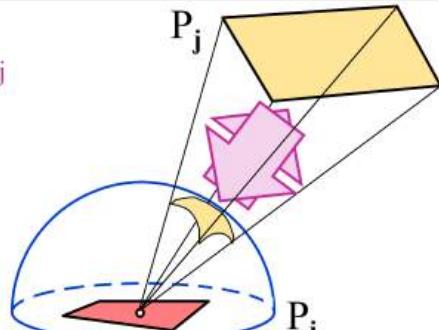
E_i ... self-emission of patch **i**

$\sum B_j F_{ij}$... contribution of other patches

F_{ij} ... form factor, defines

- contribution of B_i on patch **j** - which is equal to
- contribution of patch **j** to B_i

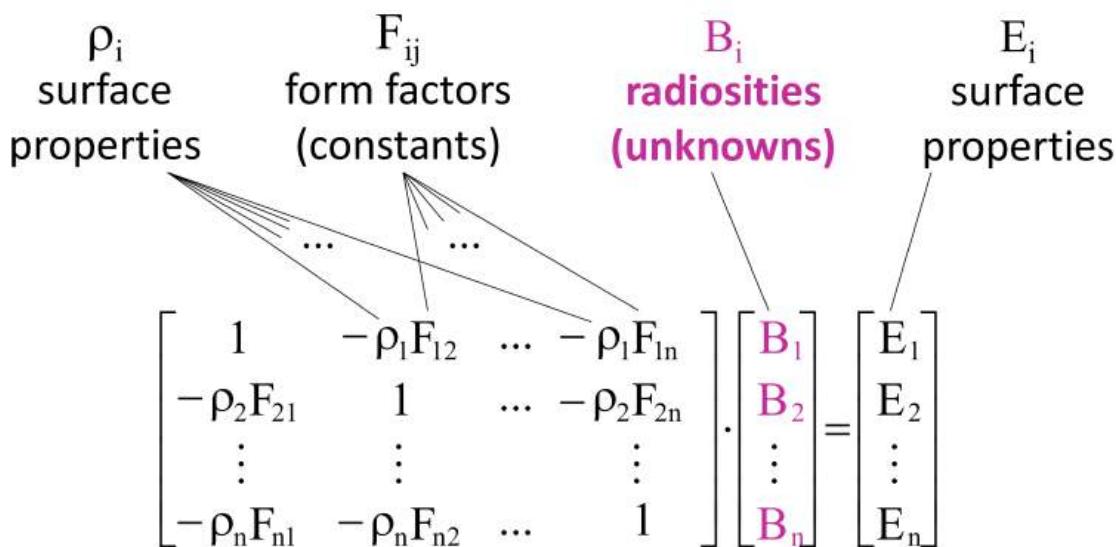
ρ_i ... reflectivity coefficient of patch **i** (“*albedo*”)



$$B_i = E_i + \rho_i \sum_{j \neq i} B_j F_{ij}$$

$$B_i - \rho_i \sum_{j \neq i} B_j F_{ij} = E_i$$

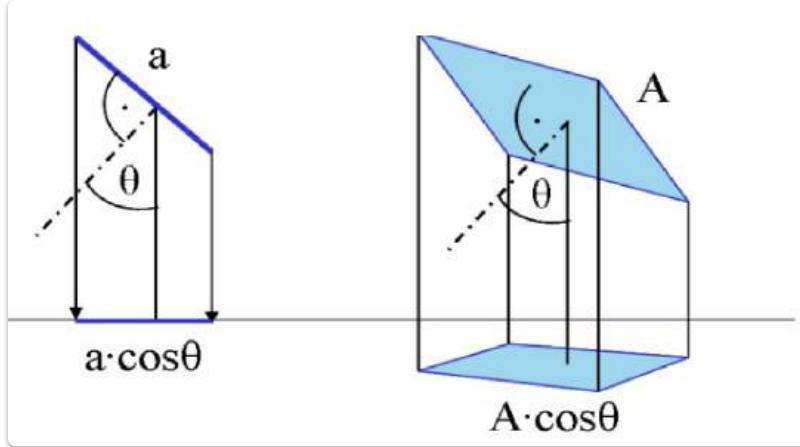
$$\begin{bmatrix} 1 & -\rho_1 F_{12} & \dots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1 & \dots & -\rho_2 F_{2n} \\ \vdots & \vdots & & \vdots \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \dots & 1 \end{bmatrix} \cdot \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix}$$



Berechnung der Formfaktoren

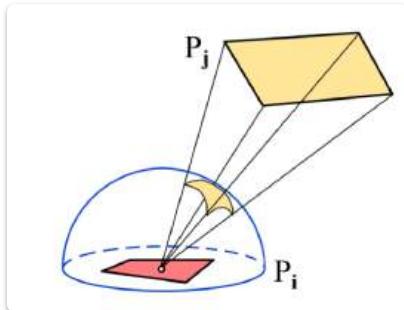
[EVC_Skriptum_CG, p.43, p.44](#)

- **Geometrischer Zusammenhang:** Die Fläche der Normalprojektion einer Fläche A auf eine andere Fläche verkleinert sich um den Kosinus des Winkels θ zwischen den Flächen: $A \cdot \cos \theta$.



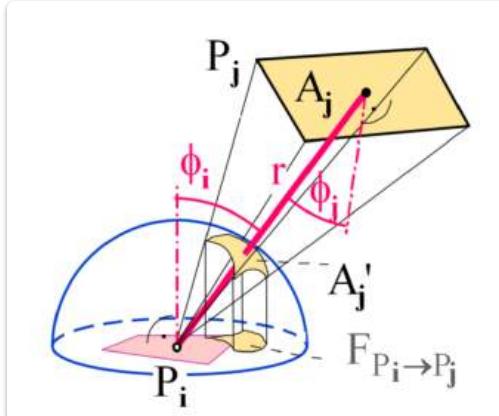
- **Definition des Formfaktors F_{ij} :**

- Anteil der vom Patch P_j ausgehenden Strahlungsenergie, die den Patch P_i trifft.
- Oder: Wie viel Prozent der Energie von P_j trifft auf P_i ?
- Reziprok: Gibt auch an, wie viel Prozent der auf P_i eingehenden Energie von P_j kommt.



- **Herleitung der Formel für F_{ij} (vereinfachte Annahme: Patches klein im Verhältnis zum Abstand r):**

1. Betrachte Patch P_i mit Fläche A_i .
2. Stelle dir über P_i eine Halbkugel (Hemisphäre) mit Radius 1 vor, auf die P_j projiziert wird.
3. Die projizierte Fläche A'_j hat die Größe $A_j \cos \phi_j$, wobei ϕ_j der Winkel zwischen der Normalen von P_j und der Verbindungsgeraden zwischen den Patches ist.



4. Energie, die unter einem Winkel ϕ_i auf P_i einfällt, ist proportional zu $\cos \phi_i$. Multiplikation mit $\cos \phi_i$ (entspricht einer Projektion auf die Grundfläche der Halbkugel) liefert den korrekten Anteil des Einflusses von Patch P_j auf P_i .
5. Normierung durch die Größe der Grundfläche der Halbkugel ($1^2\pi = \pi$).

- **Formel für den Formfaktor F_{ij} :**

$$F_{ij} = \frac{\cos \phi_i \cos \phi_j A_j}{\pi r^2}$$

- **Wichtige Voraussetzung:** Diese Formel gilt unter der Annahme, dass sich keine Hindernisse zwischen den beiden Patches befinden und das Licht ungehindert von P_j nach P_i gelangen kann. Korrekte Formfaktoren müssen auch die gegenseitige Sichtbarkeit berücksichtigen.
- ϕ_i : Winkel zwischen der Normalen von P_i und der Verbindungsgeraden.
- ϕ_j : Winkel zwischen der Normalen von P_j und der Verbindungsgeraden.
- A_j : Fläche des Patches P_j .
- r : Abstand zwischen den Patches.
- **Reziprozitätsprinzip:** Die Abhängigkeit der Formfaktoren zwischen zwei Patches setzt in Beziehung:

$$A_i F_{ij} = A_j F_{ji}$$

form factor F_{ij} : contribution of patch P_j to B_i = contribution of B_i to patch P_j

form factor properties:

■ conservation of energy

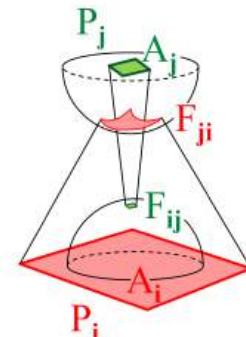
$$\sum_{j=1}^n F_{ij} = 1$$

■ uniform light reflection

$$A_i F_{ij} = A_j F_{ji}$$

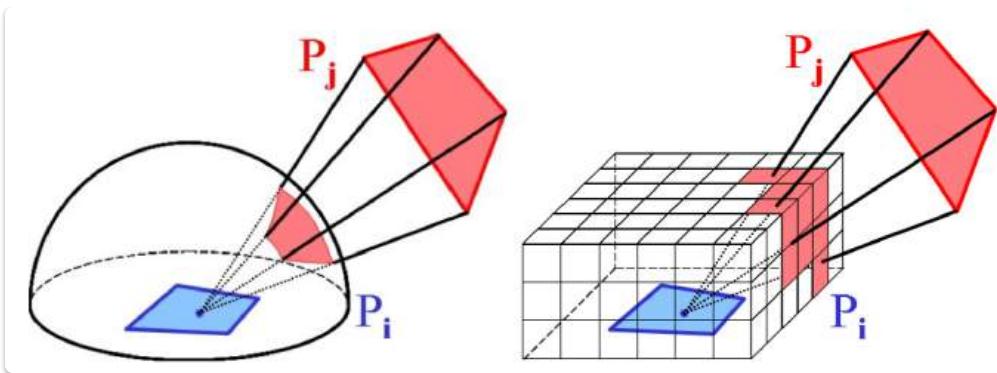
■ no self-incidence

$$F_{ii} = 0$$



Praktische Berechnung:

- **Hemicube-Methode**
 - Abschätzung der Formfaktoren statt einer Halbkugel über P_i wird ein Halbwürfel (Hemicube) verwendet.
 - Die gesamte Szene wird auf diesen Hemicube abgebildet (z-Buffer-Technologie).
 - Die Oberfläche des Halbwürfels wird in Pixel aufgeteilt.
 - Alle anderen Patches werden von P_i aus mit dem Würfelmittelpunkt als Projektionszentrum auf diese Pixel abgebildet.
 - Für jedes Pixel wird sein Formfaktor bestimmt.
 - Für jedes Patch wird die Summe der Formfaktoranteile seiner Pixel gebildet.
- **Ray-Tracing-Verfahren:** Alternative Methode zur Berechnung von Formfaktoren.



Fortschreitende Verfeinerung (Progressive Refinement)

EVC_Skriptum_CG, p.44

$$B_i = E_i + \rho_i \sum_{j \neq i} B_j F_{ij}$$

"shooting" → select brightest patch P_i and distribute its radiosity B_i

$$B_{j \text{ due to } B_i} = \rho_j B_i F_{ij} \frac{A_i}{A_j}$$

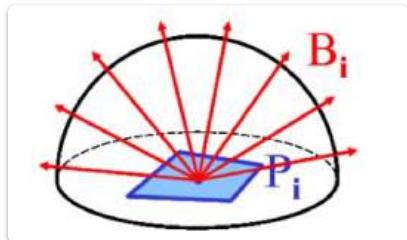
- **Ziel:** Iterative Lösung des Radiosity-Gleichungssystems.
- **Grundidee ("Shooting"):** Man wählt das jeweils hellste Patch aus und "schießt" (verteilt) dessen noch nicht abgestrahlte Energie auf alle anderen Patches.
- **Vorteil:**
 - In jedem Schritt werden alle Patches etwas besser beleuchtet.
 - Das Verfahren konvergiert schnell, da immer die Energie der hellsten Patches zuerst verteilt wird.
- **Radiosity-Anteil von P_i , der von P_j verursacht wird ($B_{(i \text{ von } B_j)}$):**
 - Dies beschreibt, wie viel Radiosity von Patch j zu Patch i beiträgt.
 - Es gilt: $B_{(i \text{ von } B_j)} = \rho_i B_j F_{ij}$
 - **Symmetrie:** Der Einfluss von P_i auf P_j ist symmetrisch zum Einfluss von P_j auf P_i .
 - Es gilt auch: $B_{(j \text{ von } B_i)} = \rho_j B_i F_{ji}$
- **Verknüpfung mit dem Reziprozitätsprinzip:**
 - Aus dem Reziprozitätsprinzip $A_i F_{ij} = A_j F_{ji}$ folgt, dass $F_{ij} = F_{ji} \frac{A_j}{A_i}$.
 - Setzt man dies in die Gleichung für $B_{(i \text{ von } B_j)}$ ein:

$$B_{(i \text{ von } B_j)} = \rho_i B_j F_{ij} = \rho_i B_j F_{ji} \frac{A_j}{A_i}$$
 - Diese Beziehung ermöglicht es, den Formfaktor F_{ij} direkt zu nutzen, auch wenn die Berechnung des Beitrags von P_j zu P_i erfolgt.
- **Speicherhaltung pro Patch:**

- **Gesammelte Radiosity (B_i):** Der bisher beste Schätzwert für die Radiosity des Patches.
- **Noch nicht verschossene Radiosity (ΔB_i):** Die Energie, die dieses Patch noch abgeben muss und die als Basis für die Auswahl des nächsten "hellsten" Patches dient.
- **Initialisierung:**

- Am Anfang werden B_i und ΔB_i mit der Eigenemission E_i des Patches initialisiert:

$$B_i = \Delta B_i = E_i \text{ für alle Patches } i.$$



Iterationsschritt des Progressive Refinement Algorithmus

p.45

```

select patch i with highest  $A_i * \Delta B_i$ 
FOR selected patch i {
    set up hemicube
    calculate form factors  $F_{ij}$ 
}
FOR each patch j {
     $\Delta rad := \rho_j * \Delta B_i * F_{ij} * A_i / A_j$ 
     $\Delta B_j := \Delta B_j + \Delta rad$ 
     $B_j := B_j + \Delta rad$ 
}
 $\Delta B_i := 0$ 

```

- **Bezeichnung:** Dieses Verfahren wird auch als "Progressive Refinement" (schrittweise Verfeinerung) bezeichnet.

Verschiedene Beispiele: [EVC-CG11-Globale Beleuchtung+Texturen_2025S_Slides](#), p.19

Aspekte von Radiosity

[EVC_Skriptum\(CG\)](#), p.45

- **Blickpunktunabhängigkeit:** Radiosity ist eine Methode zur Berechnung der Helligkeit von einzelnen diffusen Patches, die unabhängig von der Kameraposition (Blickpunkt) ist.
 - **Vorteil:** Die Lichtausbreitung muss nur einmal berechnet werden, danach können Ansichten aus beliebigen Blickpunkten generiert werden, ohne die Beleuchtung neu zu berechnen.

- **Nachfolgender Rendering-Schritt:** Nach der Radiosity-Berechnung ist meist noch ein zusätzlicher Rendering-Schritt notwendig, um das finale Bild zu erzeugen.
 - Oft wird hierfür ein einfaches Polygon-Verfahren mit Gouraud-Schattierung verwendet, um die berechneten Helligkeiten darzustellen.
- **Kombination mit Ray-Tracing:** Die durch Radiosity gewonnenen diffusen Schattierungswerte können als Basiswerte für ein Ray-Tracing-Verfahren verwendet werden.
 - **Vorteil:** Dadurch lassen sich zusätzlich Effekte wie Spiegelungen und scharfe Schatten, die Radiosity allein nicht modelliert, gut darstellen.

radiosity is viewpoint-independent
needs a rendering step to display

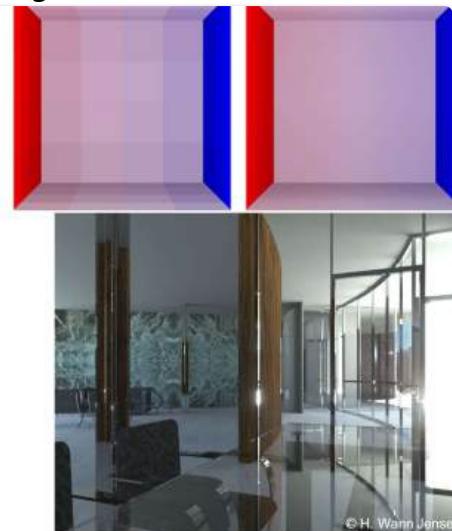
- polygon rendering
- Gouraud shading
- ray-tracing
- ...

combination with ray-tracing enables

- reflections
- shadows
- ...

Werner Purgathofer

23.



© H. Wann Jensen

Erweiterungen des Radiosity-Prinzips

[EVC_Skriptum_CG](#), p.45

Das vorgestellte Radiosity-Prinzip ist erweiterbar, um Effizienz und Qualität zu verbessern:

- **Hierarchische Strukturierung von Patches:**
 - **Ziel:** Reduzierung der Anzahl der zu berechnenden Patches.
 - **Methode:** Patches können hierarchisch strukturiert werden. Das bedeutet, dass entfernte Patches nicht einzeln behandelt werden müssen, sondern als Gruppen zusammengefasst werden können. Dies reduziert den Rechenaufwand erheblich.
- **Discontinuity-Meshing:**
 - **Problem:** An Stellen, an denen sich die Beleuchtung abrupt ändert (z.B. Schattenkanten), können zu große Patches eine falsche "Verschmierung" der Beleuchtung verursachen. Umgekehrt können zu kleine Patches den Rechenaufwand unnötig erhöhen.
 - **Lösung:** Discontinuity-Meshing passt die Größe der Patches an die Beleuchtungssituation an. In Bereichen mit starken Helligkeitsunterschieden (z.B. Schattenübergängen) werden kleinere Patches verwendet, um eine präzisere Darstellung zu ermöglichen. In Bereichen mit gleichmäßiger Beleuchtung können größere Patches verwendet werden.

Path Tracing

[EVC_Skriptum_CG, p.45](#)

- **Grundlage:** Path Tracing ist eine Erweiterung des [Ray-Tracing Verfahrens](#).
- **Alternativname:** Wird auch "*Monte Carlo Ray-Tracing*" genannt.
- **Kernprinzip:**
 - Beim klassischen Ray-Tracing werden an jedem Auftreffpunkt Sekundärstrahlen in alle relevanten Richtungen gelegt (z.B. Reflexion, Brechung, Schatten).
 - Beim Path Tracing wird stattdessen an jedem Auftreffpunkt **zufällig nur eine Richtung** entsprechend einer gültigen Verteilungsfunktion ausgewählt.
 - **Intuition:** Es wird ein "Lichtpfad" (Path) von der Kamera bis zu einer Lichtquelle oder ins Unendliche verfolgt, wobei an jeder Oberfläche nur eine zufällige Richtung für die Weiterverfolgung gewählt wird.
- **Vorteile:**
 - **Inklusion komplexer Lichtverhältnisse:** Ermöglicht die realistische Simulation von Szenen, in denen viele verschiedene Lichtrichtungen relevant sind.
 - **Diffuse Reflexion:** Kann diffuse Reflexionen sehr gut modellieren (im Gegensatz zu klassischem Ray-Tracing, das eher auf spiegelnde oder brechende Oberflächen spezialisiert ist).
 - **Ausgedehnte Lichtquellen:** Kommt besser mit ausgedehnten (nicht punktförmigen) Lichtquellen zurecht.
- **Herausforderungen & Lösungen:**
 - **Rauschen im Ergebnisbild:** Da pro Pixel nur eine zufällige Richtung verfolgt wird, führt dies zu Rauschen im Ergebnis.
 - **Lösung:** Um starkes Rauschen zu reduzieren, müssen pro Pixel **viele Strahlen** (Pfade) berechnet und gemittelt werden. Dies ist rechenintensiv.
- **Mathematischer Hintergrund:**
 - Grundsätzlich entspricht das Vorgehen des Path Tracings der [Monte-Carlo-Integration](#) eines mehrdimensionalen Integrals.
 - Dieses Integral wird als "Rendering Equation" bezeichnet und beschreibt die vollständige Lichtausbreitung im Raum.
 - **Rendering Equation (vereinfachtes Verständnis):** Eine komplexe mathematische Gleichung, die die gesamte Beleuchtung eines Punktes in einer Szene beschreibt, indem sie alle eingehenden Lichtstrahlen und deren Wechselwirkungen mit der Oberfläche berücksichtigt. Monte Carlo Integration ist eine Methode, um solche komplexen Integrale durch zufällige Stichproben zu approximieren.

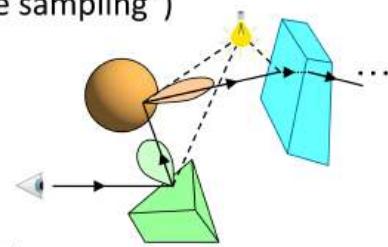
also called Monte Carlo ray tracing

ray directions selected randomly according to

distribution functions ("importance sampling")

uses Monte Carlo integration to solve

$$B = E + \rho \cdot \int_{\text{hemi}} d B$$



B ... radiosity **hemi** ... half space over point

E ... self emission **ρ** ... reflection coefficient

- Qualitätsverbesserung:

- Die Verwendung von **Quasi-Zufallszahlen** (anstelle von Pseudo-Zufallszahlen) reduziert die Varianz (d.h., das Rauschen) im Ergebnisbild merklich.
- **Quasi-Zufallszahlen:** Sind keine echten Zufallszahlen, sondern Sequenzen, die eine gleichmäßige Verteilung im Definitionsbereich aufweisen als Pseudo-Zufallszahlen. Dies hilft, die Stichproben für die Monte-Carlo-Integration besser über den gesamten Raum zu verteilen und somit das Ergebnis genauer zu machen.

Photon Mapping

[EVC_Skriptum_CG, p.46](#)

- **Grundidee:** Eine Methode zur Berechnung globaler Beleuchtungseffekte, die besonders gut mit komplexen Lichtinteraktionen wie Spiegelungen und Kaustiken (Lichtbündel, die durch Brechung oder Reflexion erzeugt werden) umgehen kann.

- **Vorgehensweise:**

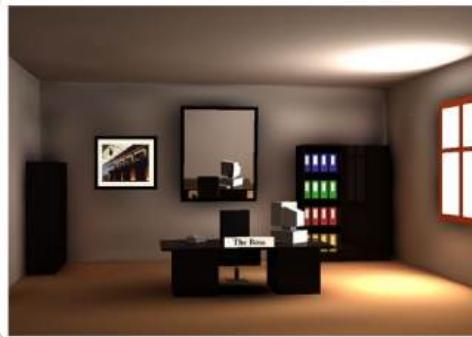
1. **"Vorwärts"-Verfolgung von Lichtstrahlen:** Im Gegensatz zum Ray-Tracing, das Strahlen von der Kamera verfolgt, werden beim Photon Mapping Lichtstrahlen (genannt "Photonen") von den Lichtquellen ausgesendet und in der Szene verfolgt. Dies ist eine "Vorwärts"-Richtung.
2. **Speichern der Lichtwirkung an Auftreffpunkten:** Wenn ein Photon auf eine Oberfläche trifft, wird die Wirkung des Lichts (z.B. Energie, Farbe) an diesem Auftreffpunkt gespeichert. Diese gespeicherten Punkte werden als "Photonen" in einer "Photon Map" abgelegt.
3. **Interpolation für das Aussehen des Objekts:** Später, während des Renderings, werden diese gespeicherten Photonen genutzt. An einem Punkt, für den die Beleuchtung berechnet werden soll, werden die Informationen von mehreren nahegelegenen Photonen interpoliert, um das Aussehen des Objekts zu bestimmen.

- **Vorteile:**

- **Korrekte Berechnung komplexer Lichtsituationen:** Ermöglicht die akkurate Simulation von:
 - **Spiegelungen der Lichtquellen:** Wie sich Lichtquellen in spiegelnden

Oberflächen abbilden.

- **Kaustiken:** Die Fokussierung von Licht durch Brechung (z.B. durch Glas) oder Reflexion (z.B. durch eine gewölbte, spiegelnde Oberfläche), die zu hellen Lichtmustern auf anderen Oberflächen führt.
- **Kombination mit anderen Verfahren:**
 - Eine Kombination aus **Path Tracing** und **Photon Mapping** kann nahezu alle Lichteffekte in einem Bild integrieren und so sehr realistische Renderings erzeugen. Path Tracing ist gut für direkte und diffuse Beleuchtung, während Photon Mapping seine Stärken bei Kaustiken und komplexen spiegelnden/refraktiven Pfaden hat.

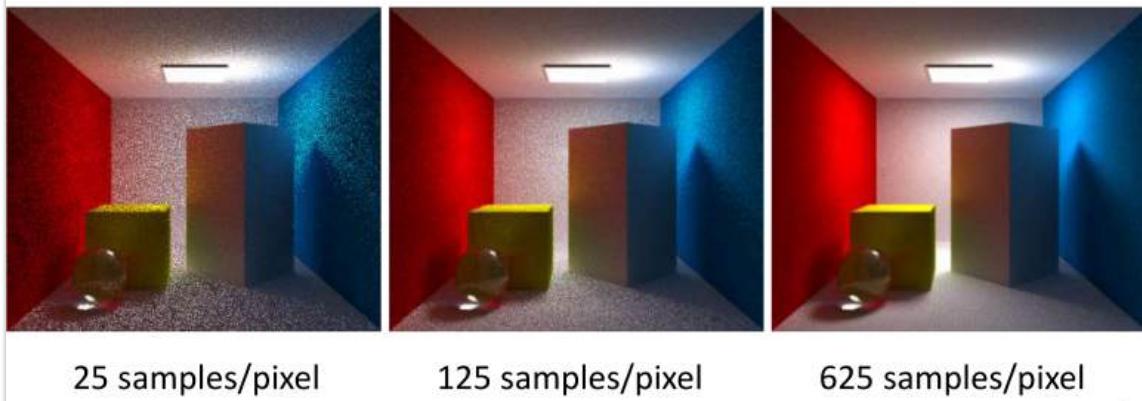


Optische Eigenschaften natürlicher Oberflächen: Mapping-Methoden

EVC_Skriptum_CG, p.46

- **Problem:** Natürliche Oberflächen weisen diverse Unregelmäßigkeiten auf, die entweder durch die Umgebung, eine variable Färbung der Oberfläche oder durch Oberflächenunebenheiten (Schattierung) verursacht werden.
- **Simulationsmethoden ("Mapping"):** Für diese drei Ursachen lassen sich die Effekte mit speziellen "Mapping"-Methoden simulieren:
 - **Environment Mapping:** Simuliert die Reflexion der Umgebung auf einer Oberfläche. (Beispiel: Eine glänzende Kugel spiegelt die Umgebung wider, ohne dass die Umgebung geometrisch modelliert werden muss.)
 - **Texture Mapping:** Bringt ein 2D-Bild (Textur) auf eine 3D-Oberfläche auf, um deren Farbe und Muster zu definieren. (Beispiel: Eine Ziegelmauertextur auf einer Wand.)
 - **Bump Mapping:** Simuliert Oberflächenunebenheiten, indem es die Normalenvektoren der Oberfläche modifiziert. Dies beeinflusst die Schattierung und erzeugt den Eindruck von Unebenheiten, ohne die Geometrie tatsächlich zu verändern. (Beispiel: Eine grobe Steintextur, die Risse und Vertiefungen simuliert.)
- **"Mapping" Bedeutung:** Im Kontext der Computergrafik bedeutet "Mapping" das Abbilden von Informationen (wie Farben, Helligkeiten, Normalen) von einem Raum (oft 2D-Bild) auf eine 3D-Oberfläche.

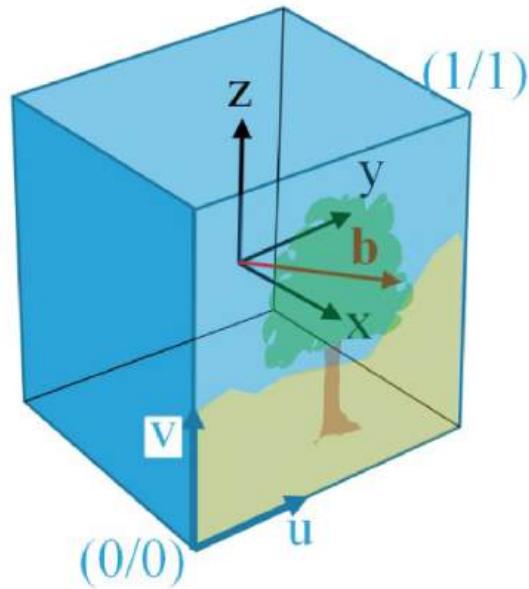
→ trace light rays from light source(s) and store illumination on objects



Environment Mapping

[EVC_Skriptum_CG, p.46](#)

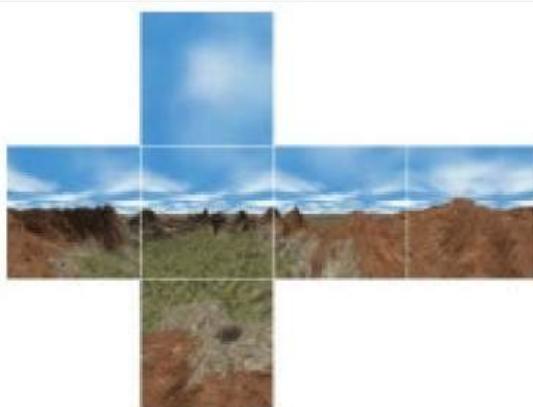
- **Definition:** Environment Mapping (Umgebungsabbildung) bezeichnet die Technik, wie die Umgebung oder Umwelt eines Objekts dessen Aussehen beeinflusst, insbesondere durch Reflexionen.
- **Effekt:** Je nach Oberflächeneigenschaften des Objekts wird die Umgebung auf verschiedene Weisen im Erscheinungsbild des Objekts widergespiegelt.
- **Anwendung bei verschiedenen Oberflächen:**
 - **Perfekt spiegelnde Oberflächen:** Für diese Oberflächen würde Ray-Tracing das exakte Spiegelbild der Umgebung erzeugen. Environment Mapping kann dies annähernd simulieren.
 - **Nicht perfekt diffuse Oberflächen:** Für diese Oberflächen, die einen gewissen Glanz aufweisen, kann man mit dem Phong-Modell (oder ähnlichen Reflexionsmodellen) einen Glanz-Effekt erzeugen, der die Spiegelung von Lichtquellen annähert.
 - **Mehr oder weniger spiegelnde Oberflächen:** Reflektieren ihre ganze Umgebung mehr oder weniger scharf. Die Genauigkeit der Spiegelung wird dabei reduziert, d.h. sie ist nicht exakt.
- **Motivation für Environment Mapping:**
 - Um ein effizientes Rendering von Objekten in einer komplexen Umgebung zu ermöglichen, ohne die gesamte Umgebung vollständig und exakt modellieren zu müssen.
 - **Vorgehen:** Die Umgebung wird in einem Vorverarbeitungsschritt von einem zentralen Punkt aus (z.B. dem Mittelpunkt des Objekts oder der darzustellenden Szene) als Bild (oder Satz von Bildern) produziert.



- **Art der Umgebungsinformation:**

- Es spielt keine Rolle, ob die Umgebungsbilder berechnet oder fotografiert wurden.
- **Wichtig:** Die Umgebung ist im Verhältnis zu den Objekten, die man darstellen will, sehr groß.
- **Annahme:** Bei der Darstellung eines Objekts wird für jeden Oberflächenpunkt näherungsweise angenommen, dass er sich im Zentrum dieser Umgebung befindet. Dies ist eine Vereinfachung, die für Objekte, die klein im Vergleich zur Umgebung sind, gute Ergebnisse liefert.

- **Vorteil:** Man kann allein aus der Richtung des Reflexionsstrahls bestimmen, welcher Umgebungschnitt getroffen wird (z.B. mit Polarkoordinaten), und erspart sich aufwändiges Ray-Tracing für die Umgebungsinformation.

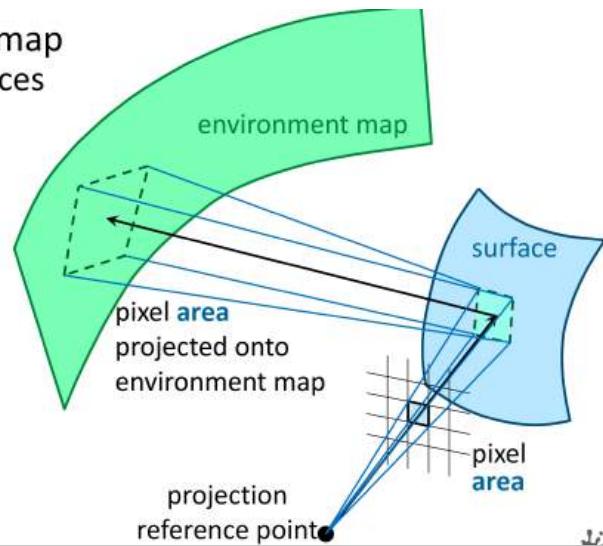


information in the environment map

- intensity values for light sources
- sky
- background objects

pixel area

- projected onto surface
- reflected onto environment map



Cube Map als Environment Map

EVC_Skriptum_CG, p.46

- **Häufigste Methode:** Häufig wird ein achsenparalleler Würfel als Environment Map verwendet, der als **Cube Map** bezeichnet wird.
- **Struktur der Cube Map:** Jede Seite dieses Würfels ist ein Bild der Größe $u \times v = [0, 1] \times [0, 1]$.
- **Effiziente Bildwertabfrage:** Um den Bildwert für eine bestimmte Reflexionsrichtung R_S effizient zu bestimmen, kann man die x, y, z Koordinaten des Reflexionsvektors nutzen.
 - **Beispiel (für die Seite $+x$):** Wenn der dominante Anteil des Reflexionsvektors die x -Komponente ist, wird die Textur der $+x$ -Seite verwendet. Die u, v -Koordinaten werden dann aus den verbleibenden Komponenten y und z abgeleitet:
 - $u = (y_R + x_R)/(2x_R)$
 - $v = (z_R + x_R)/(2x_R)$
 - Hierbei ist $R_S = (x_R, y_R, z_R)$ der Reflexionsvektor.
- **Reflexionsrichtung R_S :** Die Reflexionsrichtung erhält man nach dem gleichen Prinzip wie schon beim Ray-Tracing und bei den Schattierungsverfahren abgeleitet:
 - $R_S = (2n \cdot v)n - v$
 - v : Vektor vom Betrachter zum Oberflächenpunkt.
 - n : Oberflächennormale am Punkt.
 - **Intuitive Erklärung:** Dies ist die klassische Formel für die Reflexion eines Vektors an einer Oberfläche. Der Term $(2n \cdot v)n$ ist der Anteil des Vektors v , der parallel zur Normalen n ist, gespiegelt an der Oberfläche. Davon wird der ursprüngliche Vektor v subtrahiert, um die reine Reflexionsrichtung zu erhalten.



(1/1) find (u, v) from the direction vector $\mathbf{r}(x_r, y_r, z_r)$:

if $x_r > |y_r|$ and $x_r > |z_r|$ then "front face"

$$u = (y_r + x_r)/2x_r$$

$$v = (z_r + x_r)/2x_r$$

top view

analogous formulas for the other 5 faces
 $(-x > |y| \wedge -x > |z|, \quad y > |x| \wedge y > |z|, \quad -y > |x| \wedge -y > |z|, \quad z > |x| \wedge z > |y|, \quad -z > |x| \wedge -z > |y|)$

(1/1) calculation of the direction vector \mathbf{r} :

for a pixel:
 viewing direction \mathbf{v}
 and normal vector $\mathbf{n} \Rightarrow \mathbf{r} + \mathbf{v} = (2\mathbf{n} \cdot \mathbf{v})\mathbf{n}$

$$\mathbf{r} = (2\mathbf{n} \cdot \mathbf{v})\mathbf{n} - \mathbf{v}$$

Texture Mapping

EVC Skriptum CG, p.47

- Problem:** Viele Oberflächen in der realen Welt sind nicht einfarbig, sondern weisen komplexe Muster, Farben und Details auf (z.B. Holzmaserung, Bilder, Schrift, Verschmutzung, Kleidung, Marmor).
- Lösung:** Um solche Muster auf Oberflächen darzustellen, auch wenn die grundlegende geometrische Modellierung grob ist, wird die Technik des **Texture Mapping** verwendet.
- Was ist eine Textur?** Ein Muster, das auf eine Oberfläche aufgebracht wird. Beispiele: Fenster auf einer Hauswand, Wolken am Himmel, Gesichter, Knöpfe, Reißverschlüsse, Pflastersteine.

- **Zweischrittiger Prozess des Texture Mapping:**

1. **Textur-Objekt Transformation (Modellierungsschritt):**

- **Definition:** Zuerst muss definiert werden, welche Textur (oft ein 2D-Bild) auf welche Oberfläche des 3D-Objekts aufgebracht werden soll.
- **Parameter:** Dabei werden Orientierung, Skalierung, Wiederholung (Tiling) und andere Parameter festgelegt.
- **Raum:** Hier findet eine Transformation von **Textur-Raum (u,v) Array Koordinaten** (die 2D-Koordinaten innerhalb der Textur) in **Objekt-Raum (u',v')** **Flächen-Parameter** (Parameter, die einen Punkt auf der 3D-Oberfläche des Objekts eindeutig identifizieren) statt.
- **Bedeutung:** Dieser Schritt ist eigentlich Teil der Modellierung, bei der die Oberflächen ein visuelles Aussehen erhalten.

2. **Viewing- & Projektions-Transformation (Rendering-Schritt):**

Ziel: Die Textur muss korrekt auf das Abbild des Objekts im finalen Bild gerendert werden.

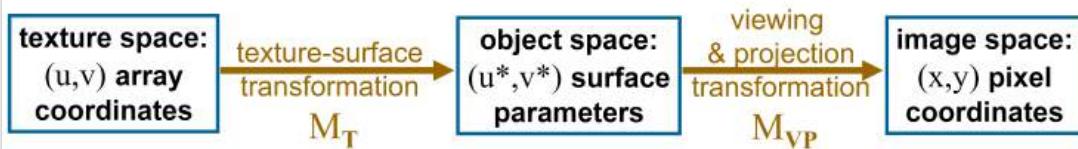
Raum: Dies beinhaltet die Transformation von den **Objekt-Raum (u',v')** **Flächen-Parametern** in **Bild-Raum (x,y) Pixel Koordinaten**.

* **Bedeutung:** Hier wird die Textur tatsächlich auf das gerenderte Objekt projiziert und dargestellt.

texture mapping

forward: texture scanning $(u,v) \rightarrow (x,y)$

backward: inverse scanning $(x,y) \rightarrow (u,v)$

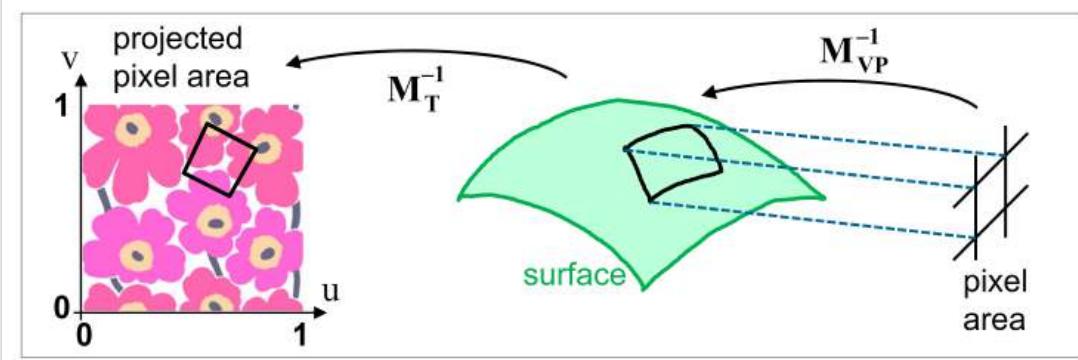


texture-surface transformation

$$\begin{aligned} u^* &= u^*(u,v) = a_{u^*}u + b_{u^*}v + c_{u^*} \\ v^* &= v^*(u,v) = a_{v^*}u + b_{v^*}v + c_{v^*} \end{aligned}$$

projecting pixel areas to texture space =

= inverse scanning $(x,y) \rightarrow (u,v)$



Erzeugung einer Textur

[EVC_Skriptum_CG, p.47](#)

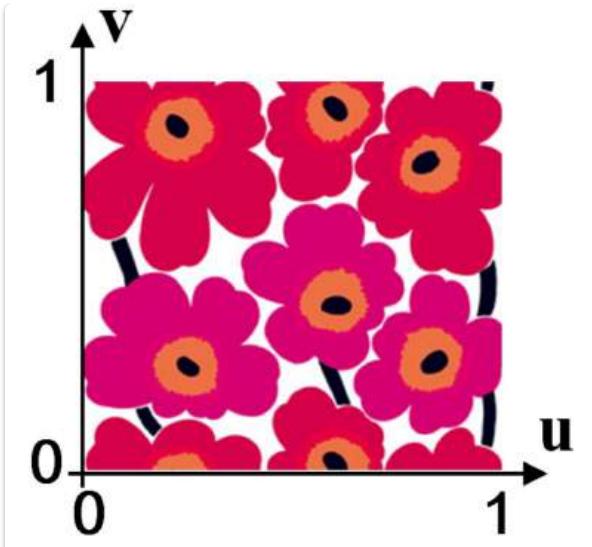
- **Grundsatz:** Die Herkunft einer Textur ist zweitrangig; wichtig ist nur, dass sie an allen benötigten Stellen definiert und abrufbar ist.
- **Standardverfahren:** Meist wird eine Textur in einem Vorverarbeitungsschritt als **Pixel-Array** (also ein Raster von Farbwerten) erstellt und später im Rendering-Prozess darauf zugegriffen.
- **Mögliche Quellen für Texturen:**
 - **Fotografien:** Man kann Fotos von realen Oberflächen verwenden.
 - **Scans:** Gescanntes Material.
 - **Programmgenerierte Texturen:** Texturen, die durch ein Programm erzeugt werden, bis hin zu Zufallsraten (Noise).
 - **Datenbanken:** Für häufig verwendete Texturen (z.B. Holzmaserung, Gras, Sand, Marmor, Kopfsteinpflaster, Stoffstrukturen) können Datenbanken angelegt werden.
- **Prozedurale Texturierung:**
 - **Definition:** Wenn Texturen aus einer **mathematischen Funktion** gewonnen werden, spricht man von "Procedural Texturing".
 - **Vorteile von Prozeduralen Texturen:**
 - **Kein Speicherplatz für Bilder:** Die Textur wird zur Laufzeit berechnet, nicht gespeichert.
 - **Unendliche Detailtiefe:** Können beliebig hoch aufgelöst werden, ohne an Qualität zu verlieren oder Pixelartefakte zu zeigen (im Gegensatz zu Bitmap-Texturen).
 - **Parameterisierbarkeit:** Oft lassen sich Parameter der Funktion ändern, um Variationen der Textur zu erzeugen.



Textur Objekt Transformation

[EVC_Skriptum_CG, p.47](#)

- **Ausgangssituation:**
 - Die Textur liegt normalerweise in einem **2D-Koordinatensystem** vor, dessen Achsen mit (u, v) bezeichnet werden. Diese (u, v) -Koordinaten sind die Texturkoordinaten.
 - Die Oberfläche des 3D-Objekts, auf die die Textur aufgebracht werden soll, hat ebenfalls eine **parametrische Darstellung**, die mit (u^*, v^*) bezeichnet wird. Diese (u^*, v^*) -Koordinaten sind die Oberflächenparameter des Objekts.

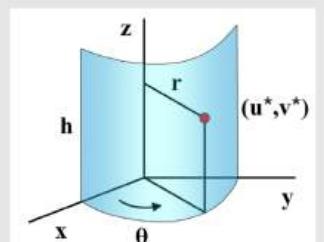


- **Ziel der Transformation:** Für jeden Punkt auf der 3D-Objektoberfläche (identifiziert durch seine (u^*, v^*) -Parameter) soll die zugehörige Farbe aus der Textur ermittelt werden. Das bedeutet, wir brauchen eine Abbildung von den Objekt-Oberflächenparametern zurück zu den Texturkoordinaten, oder umgekehrt, eine Abbildung von den Texturkoordinaten zu den Objekt-Oberflächenparametern.
- **Biliniere Funktion für die Textur-Objekt-Transformation:**
 - Eine häufig verwendete Methode, um eine Textur auf eine Oberfläche aufzubringen, ist eine bilinare Funktion. Diese Funktion bildet die Texturkoordinaten (u, v) auf die Oberflächenparameter (u^*, v^*) ab:
 - $u^* = u^*(u, v) = a_u u + b_u v + c_u$
 - $v^* = v^*(u, v) = a_v u + b_v v + c_v$
 - **Erklärung:** Diese linearen Gleichungen (bilinear, wenn man die unabhängigen Variablen u, v und die Konstanten $a_u, b_u, c_u, a_v, b_v, c_v$ betrachtet) definieren, wie ein Punkt in der 2D-Textur (u, v) auf einen Punkt auf der 3D-Oberfläche (u^*, v^*) abgebildet wird. Die Koeffizienten (a, b, c) steuern dabei Skalierung, Rotation, Scherung und Translation der Textur auf der Oberfläche.
 - **Praktische Bedeutung:** Man kann für jeden Punkt der Objektoberfläche die zugehörige Farbe aus der Textur bestimmen.
- **Bezeichnung:** Diese Funktion wird als **Textur-Objekt-Transformation** bezeichnet und mit M_T denotiert.

Beispiel:

Eine Textur $t(u, v), 0 \leq u, v \leq 1$, soll auf einen Viertel-Zylinder mit Höhe h aufgetragen werden, dessen Mantel in z -Richtung mit v^* parametrisiert ist und entlang der Krümmung mit $u^* (= \theta)$. Um nun für ein Texturpixel $t(u, v)$ [auch Texel genannt] zu berechnen, an welche Stelle des Zylinders es zu liegen kommt, muss man die Abbildung M_T definieren, das könnte etwa sein:

$$u^* = u \cdot \pi/2, v^* = v \cdot h \quad (\text{so passt die Textur genau auf das Zylinderviertel}).$$



Viewing und Projektionstransformation

- Die Abbildung eines 3D-Modells auf eine Bildebene ist grundsätzlich eine einfache *Projektion* M_{VP} .
- Beim *Rasterscannen* von Flächen (dem Prozess des Füllens von Pixeln), um sicherzustellen, dass jedes Pixel *genau einmal* gefärbt wird (also keine Übermalungen oder Löcher), arbeitet man in umgekehrter Richtung:
 - Man bestimmt für jedes Pixel (x, y) auf der Bildebene, welcher Oberflächenpunkt dort gezeichnet wird.
 - Dazu werden die (u_*, v_*) -Koordinaten (Texturkoordinaten) der Fläche und daraus das *Texel* (Textur-Pixel) für dieses Pixel bestimmt.
 - Dafür werden die inversen Operatoren M_{VP}^{-1} und M_T^{-1} benötigt.

Beispiel (Fortsetzung):

Für eine beliebige **Projektion** reicht es, wenn wir von jedem Punkt die (x, y, z) -Koordinaten kennen. Im Falle des obigen Zylinders ergibt sich: $x = r \cdot \cos u_*$, $y = r \cdot \sin u_*$, $z = v_*$.

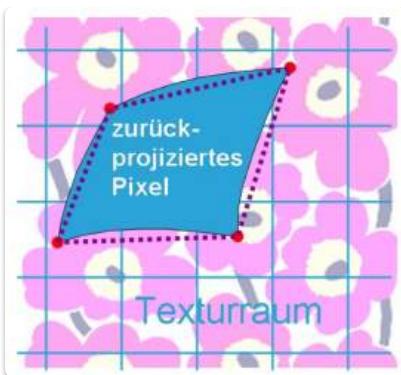
Nun wird das Problem von hinten angegangen:

- Für einen Bildpunkt P bestimmt man zuerst die Position (x, y, z) am Zylinder, die dort dargestellt wird (z.B. durch Ray-Casting).
- Für diesen Punkt muss man die Parameter der Oberfläche finden: $u_* = \cos^{-1}(x/r)$, $v_* = z$. Bis hier ist das also die inverse Transformation M_{VP}^{-1} .
- Jetzt muss für das Parameterpaar (u_*, v_*) noch die Textur gefunden werden, indem M_T invertiert wird: $u = 2u_* / \pi$, $v = v_* / h$ (das ist M_T^{-1})

Anti-Aliasing für Texturen

[EVC_Skriptum_CG, p.48](#)

- **Problem:** Texturen sind besonders anfällig für *Aliasing-Effekte*. Dies tritt verstärkt auf, wenn Texturmuster vergrößert (Verpixelung) oder verkleinert (Moiré-Effekte, Flackern) werden.
 - **Aliasing:** Unerwünschte Artefakte (wie Treppeneffekte, Flackern, Moiré-Muster), die entstehen, wenn ein kontinuierliches Signal (die Textur) mit einer zu geringen Abtastrate (den Pixeln) diskretisiert wird.
- **Korrekte, aber langsame Lösung:**
 - Man müsste den **Textur-Durchschnittswert** der Fläche berechnen, die von einer Rückprojektion des zu füllenden Pixels auf die Textur erzeugt wird.
 - **Rückprojektion:** Ein Pixel auf dem Bildschirm entspricht nicht einem einzelnen Punkt, sondern einer Fläche in der Textur. Man muss diese Fläche in der Textur bestimmen.
 - **Näherung:** Oft reicht es näherungsweise aus, das Viereck zu verwenden, das durch die gerade Verbindung der rückprojizierten Eckpunkte des Pixels entsteht.
 - **Nachteil:** Diese Methode ist sehr langsam.



- Optimierungen für Anti-Aliasing bei Texturen:

1. Mip-Mapping:

- **Prinzip:** Die Textur wird in **verschiedenen Größen (Auflösungen)** vorab berechnet und gespeichert (ein sogenannter "Mip-Map-Pyramide").
- **Anwendung:** Je nachdem, wie stark ein Objekt im Bild verkleinert erscheint, wird die passende Texturgröße aus der Mip-Map-Pyramide ausgewählt.
- **Qualitätsverbesserung:** Zwischen den verschiedenen Größenstufen kann dann linear interpoliert werden, um fließende Übergänge und eine bessere Qualität zu erzielen.
- **Vorteil:** Reduziert Aliasing bei Verkleinerung der Textur und verbessert die Cache-Kohärenz.

2. Summed-Area-Table-Methode (SAT):

- **Prinzip:** Man erstellt eine sogenannte **Summen-Textur (Summed-Area-Table)**. In jedem Punkt dieser Tabelle ist die Summe aller Textelwerte (Textur-Pixel) von einem Referenzpunkt bis zu diesem Punkt gespeichert.
- **Anwendung:** Durch einfache Differenzenbildung in dieser Summen-Textur kann man leicht den Durchschnittswert **beliebiger rechteckiger Bereiche** in der Originaltextur ermitteln.
- **Vorteil:** Ermöglicht die schnelle Berechnung von Durchschnittswerten für rechteckige Texturbereiche, was für die Filterung von Texturen beim Anti-Aliasing nützlich ist.
- **Einschränkung:** Bei nicht-rechteckigen rückprojizierten Pixeln kann die Summed-Area-Table nur eine Approximation liefern.

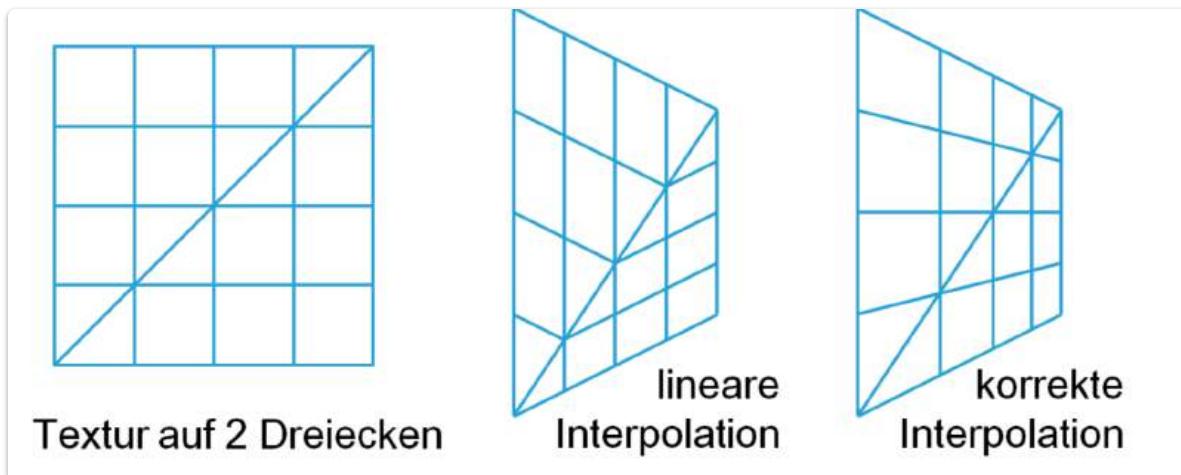
Texturen auf perspektivisch verzerrten Dreiecken

[EVC_Skriptum_CG, p.48](#)

Die Verwendung von **baryzentrischen Koordinaten** ermöglicht eine **lineare Interpolation der Eckpunktwerte** über die Dreiecksfläche.

Bei der **perspektivischen Transformation** geht diese Linearität der Oberflächenparameter jedoch verloren. Daher muss die Interpolation vor der Homogenisierung erfolgen, da sie dort nicht linear ist.

Anstatt die Farbe für einen Punkt $p(\alpha, \beta, \gamma)$ mit $\text{color}(x, y) = \alpha \cdot t_0 + \beta \cdot t_1 + \gamma \cdot t_2$ zu berechnen, werden die **Texturparameter u und v** mithilfe von **baryzentrischen Koordinaten ($\alpha_w, \beta_w, \gamma_w$)** vor der Perspektive ermittelt. Diese werden dann zur Berechnung des Texturwertes am Punkt $p(\alpha, \beta, \gamma) = p(x, y)$ verwendet.



- **Lineare Interpolation (nicht korrekt für perspektivisch verzerrte Dreiecke):** Wenn man die Texturparameter u und v direkt nach der perspektivischen Transformation linear interpoliert, entstehen Verzerrungen, wie im mittleren Bild gezeigt. Die Textur "verläuft" nicht gleichmäßig über die Fläche.
- **Korrekte Interpolation:** Um dies zu vermeiden, werden die Texturparameter u und v (oder genauer, die baryzentrischen Koordinaten $\alpha_w, \beta_w, \gamma_w$, die zu u, v führen) vor der perspektivischen Transformation berechnet. Dadurch bleibt die Linearität erhalten und die Textur wird korrekt auf das perspektivisch verzerrte Dreieck abgebildet.

Formeln für die korrekte Texturinterpolation:

Gegeben sind die baryzentrischen Koordinaten α, β, γ des Punkts $p(x, y)$ im perspektivisch transformierten Raum. Um die korrekten baryzentrischen Koordinaten $\alpha_w, \beta_w, \gamma_w$ im "World-Space" (vor der Projektion) zu finden, benötigen wir die homogenen Koordinaten und die w -Komponente (h_0, h_1, h_2) .

Die Formeln lauten:

$$d = h_1 h_2 + h_2 \beta(h_0 - h_1) + h_1 \gamma(h_0 - h_2)$$

$$\beta_w = h_0 h_2 \beta / d$$

$$\gamma_w = h_0 h_1 \gamma / d$$

$$\alpha_w = 1 - \beta_w - \gamma_w$$

- **Erklärung der Variablen:**

- h_0, h_1, h_2 : Dies sind wahrscheinlich die w -Komponenten (homogene Koordinaten) der Eckpunkte des Dreiecks vor der Division durch w (d.h., nach der Multiplikation mit der Projektionsmatrix, aber bevor die einzelnen Koordinaten durch w geteilt

werden). In der Praxis werden diese w -Werte oft als w_0, w_1, w_2 der Eckpunkte v_0, v_1, v_2 bezeichnet.

- d : Ein Hilfswert, der für die Berechnung von β_w und γ_w benötigt wird.
- $\alpha_w, \beta_w, \gamma_w$: Die korrigierten baryzentrischen Koordinaten im "World-Space".

Sobald wir die korrigierten baryzentrischen Koordinaten $\alpha_w, \beta_w, \gamma_w$ haben, können wir die Texturparameter u und v interpolieren:

$$u = \alpha_w u_0 + \beta_w u_1 + \gamma_w u_2$$

$$v = \alpha_w v_0 + \beta_w v_1 + \gamma_w v_2$$

- **Erklärung der Variablen:**

- $u_0, v_0, u_1, v_1, u_2, v_2$: Die Texturkoordinaten der drei Eckpunkte des Dreiecks.

Zuletzt wird die Farbe des Punktes mit den berechneten Texturparametern u und v bestimmt, indem der Texturwert aus einer Texturfunktion $t(u, v)$ abgefragt wird:

$$\text{color}(x, y) = t(u, v)$$

Solid Texturing

EVC_Skriptum_CG, p.49

Solid Texturing ist eine Methode, bei der eine Textur nicht als 2D-Bild, sondern als **3D-Volumen** definiert ist.

- **Wie es funktioniert:**
 - In einem **3-dimensionalen Parameterraum** wird für jeden Raumpunkt eine Farbe oder ein Texturwert definiert.
 - Wenn sich eine Oberfläche an einem bestimmten Raumpunkt befindet, wird die dort definierte Farbe/der Texturwert abgerufen und auf die Oberfläche angewendet.
- **Darstellung der Textur:**
 - Die 3D-Textur kann entweder als **mathematische Funktion** gegeben sein (z.B. eine Perlin Noise Funktion für Wolken oder Marmor).
 - Oder sie kann durch **Volumendaten** repräsentiert werden (ähnlich wie ein 3D-Gitter, bei dem jeder Voxel einen Farbwert speichert).
- **Wichtige Voraussetzung:** Man muss in der Lage sein, die Texturwerte für **jeden Raumpunkt** abzufragen.



Vorteile von Solid Texturing:

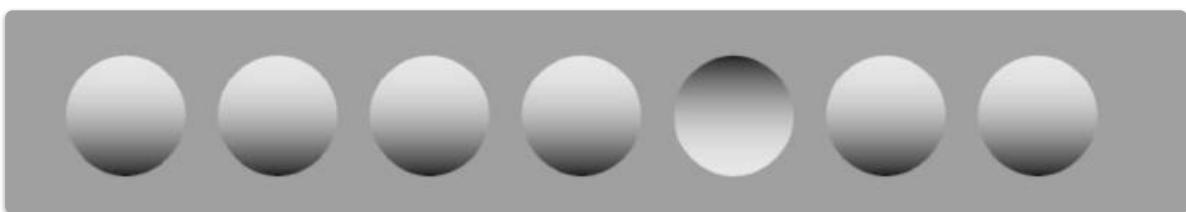
1. **Kohärentes Muster über Kanten hinweg:** Der größte Vorteil ist, dass das Texturmuster **nahtlos über alle Kanten** und zwischen verschiedenen Polygonen eines Objekts verläuft.
 - **Keine Zusammenfüngungsprobleme:** Bei traditionellem 2D-UV-Mapping kann es zu sichtbaren Nähten oder Verzerrungen an den Grenzen von UV-Segmenten kommen. Solid Texturing eliminiert dieses Problem, da die Textur "durch" das Objekt geht, als wäre es aus dem Texturvolumen geschnitten.
2. **Einfachere Abbildung der Textur auf das Objekt:** Die Zuordnung der Textur zu den Objektkoordinaten ist wesentlich einfacher zu handhaben, da sie direkt auf den 3D-Positionen des Objekts basiert und nicht erst komplexe 2D-UV-Koordinaten berechnet werden müssen.

Bump Mapping

[EVC_Skriptum_CG, p.49](#)

Viele Oberflächen in der realen Welt besitzen eine **geometrische Detailstruktur** (z.B. die Rinde eines Baumes, eine Münzoberfläche, Stoffgewebe, Putz an einer Wand). Solche feinen Details auf der Oberfläche als tatsächliche Geometrie (d.h. durch zusätzliche Polygone) zu modellieren, ist extrem aufwendig und erzeugt riesige Datenmengen.

Bump Mapping ist eine Technik, um diesen Aufwand signifikant zu reduzieren.

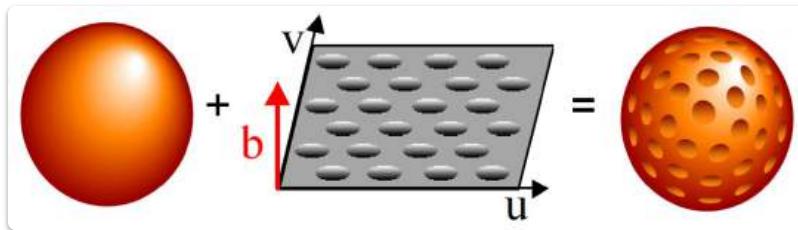


Betrachtet man die graue Leiste, scheint sie sechs Ausbuchtungen und eine Einbuchtung zu haben. Fühlt man die Oberfläche jedoch an, stellt man fest, dass sie vollkommen eben ist.

- **Warum sehen wir die Unebenheiten?**

- Der Eindruck von Unebenheiten entsteht allein durch die **Schattierung**. Unser Gehirn interpretiert die Helligkeits- und Schattenmuster als dreidimensionalen Raum.

Grundidee des Bump Mappings:



Die Grundidee des Bump Mappings besteht darin, den **Eindruck von Oberflächenunebenheiten (Bumps)** zu erwecken, ohne die tatsächliche Geometrie der Oberfläche zu verändern.

- Wie es funktioniert:**
 - Die geometrische Oberfläche bleibt unverändert (z.B. eine glatte Kugel oder Ebene).
 - Stattdessen wird der **Normalvektor** der Oberfläche (der für die Lichtberechnung verwendet wird) entsprechend der gewünschten Unebenheit **modifiziert**.
 - Diese Modifikation des Normalvektors beeinflusst, wie das Licht von der Oberfläche reflektiert wird, was wiederum die Schattierung verändert.
 - Dadurch entspricht die **Schattierung den "Bumps"**, obwohl geometrisch nichts an der Oberfläche verändert wurde.
- Vorteil:** Man kann mit sehr geringem Aufwand (nur durch die Anpassung der Normalen in der Fragment-Shader-Phase) den visuellen Eindruck von komplexen Details erzeugen, ohne die Anzahl der Polygone erhöhen zu müssen.

Bump-Mapping-Algorithmus

EVC_Skriptum_CG, p.49

Sei die Bump-Textur in Form eines Arrays von Höhenwerten $b(u, v)$ gegeben, das heißt also, dass die Position der Stelle $p(u, v)$ der Oberfläche, die durch das Parameterpaar (u, v) erzeugt wird, um $b(u, v)$ in Richtung des Normalvektors n an dieser Stelle verschoben erscheinen soll. n erhält man indem man das Kreuzprodukt zweier Tangentenvektoren auf die Länge 1 normiert:

$$n^* = p_u \times p_v, \quad n = n^*/|n^*|$$

Der verschobene Punkt $p'(u, v)$ ergibt sich dann zu: $p'(u, v) = p(u, v) + b(u, v) \cdot n$. Wir aber brauchen n' , also die Normale auf den verschobenen Punkt:

$$n' = p'_u \times p'_v$$

Nun gilt:

$$p'_u = \partial(p + b_n)/\partial u = p_u + b_u n + b n_u$$

und weil b sehr klein ist: $p'_u \approx p_u + b_u n$ analog gilt natürlich $p'_v \approx p_v + b v n$, sodass sich n' ergibt:

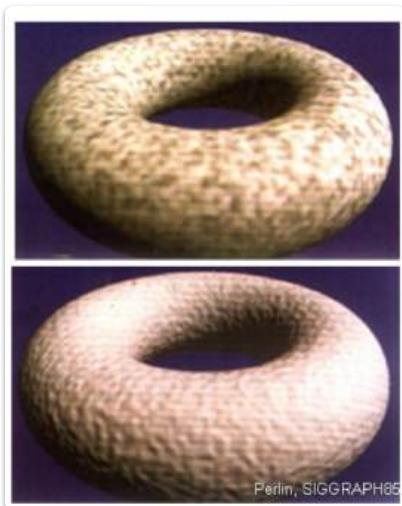
$$n' = p'_u \times p'_v = p_u \times p_v + b_v(p_u \times n) + b_u(n \times p_v) + b_u b_v(n \times n)$$

und aus $n \times n = 0$ folgt schließlich: $n' = n + b_v(p_u \times n) + b_u(n \times p_v)$.

Für die Berechnung des modifizierten Normalvektors benötigt man nicht den Höhenwert $b(u, v)$ direkt, sondern die **Ableitungen** von $b(u, v)$ nach u und v (also $\frac{\partial b}{\partial u}$ und $\frac{\partial b}{\partial v}$).

- **Praktische Anwendung:**

- In der Praxis sind die Parametrisierung der Oberfläche (UV-Koordinaten) und die der Bump-Textur häufig identisch.
- Das ermöglicht es, diese Ableitungen **vorzuberechnen** und direkt in der Bump-Map zu speichern, anstatt die eigentlichen Höhenwerte $b(u, v)$ zu speichern. Eine solche Map, die Ableitungen oder direkt die Normalen speichert, nennt man oft **Normal Map**.



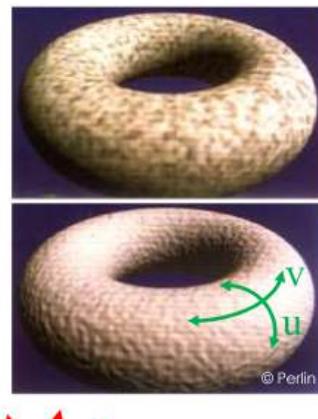
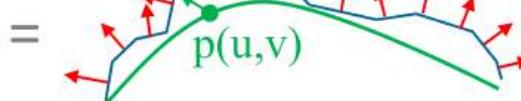
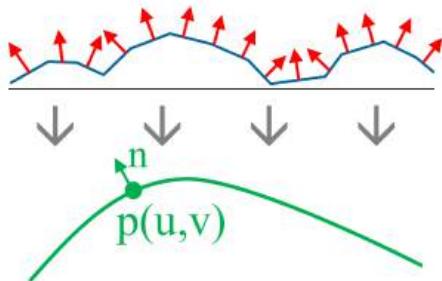
Beachte den Unterschied in der Donut-Abbildung:

- **Texture-Map (oben):** Zeigt die reinen Farb- oder Helligkeitsinformationen der Textur.
- **Bump-Map (unten):** Ist eine Darstellung, die die lokalen Unebenheiten kodiert.

surface roughness is simulated

perturbation function varies surface normal *locally*

bump map $b(u,v)$ derivative $b'(u,v)$



Der räumliche Eindruck der Oberfläche entsteht erst, wenn die **Schattierung eine Richtungsabhängigkeit bekommt**, d.h., wenn das Licht aus verschiedenen Richtungen unterschiedlich reflektiert wird, basierend auf den modifizierten Normalen.

Einschränkungen und Nachteile von Bump Mapping

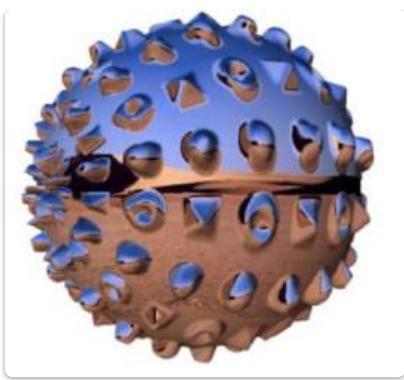
[EVC_Skriptum_CG](#), p.50

Bump Mapping ist ein **visueller Trick**, der die Schattierung verändert, aber die tatsächliche Geometrie nicht korrigiert. Dies führt zu **sichtbaren Fehlern**, die bei ausgeprägten "Bumps" deutlicher werden:

1. **Verzerrung bei flachen Winkeln:** Bei flachen Blickwinkel erscheint die simulierte Struktur stark verzerrt und unrealistisch.
2. **Glatte Silhouette:** Der **Umriss des Objekts bleibt glatt**, da die Geometrie unverändert ist, was bei Objekten mit echten Unebenheiten falsch aussieht.
3. **Glatte Schattenränder:** Bedingt durch die glatte Silhouette wirft das Objekt **Schatten mit glatten Rändern**, statt unregelmäßigen.
4. **Keine gegenseitigen Schatten der Bumps:** Die simulierten Unebenheiten **werfen keine Schatten aufeinander** oder auf die Oberfläche selbst, da sie keine echte Geometrie besitzen.
5. **Falsche Beleuchtung auf lichtabgewandter Seite:** Modifizierte Normalen können dazu führen, dass Oberflächenbereiche, die geometrisch im Schatten liegen sollten, **fälschlicherweise beleuchtet werden**.

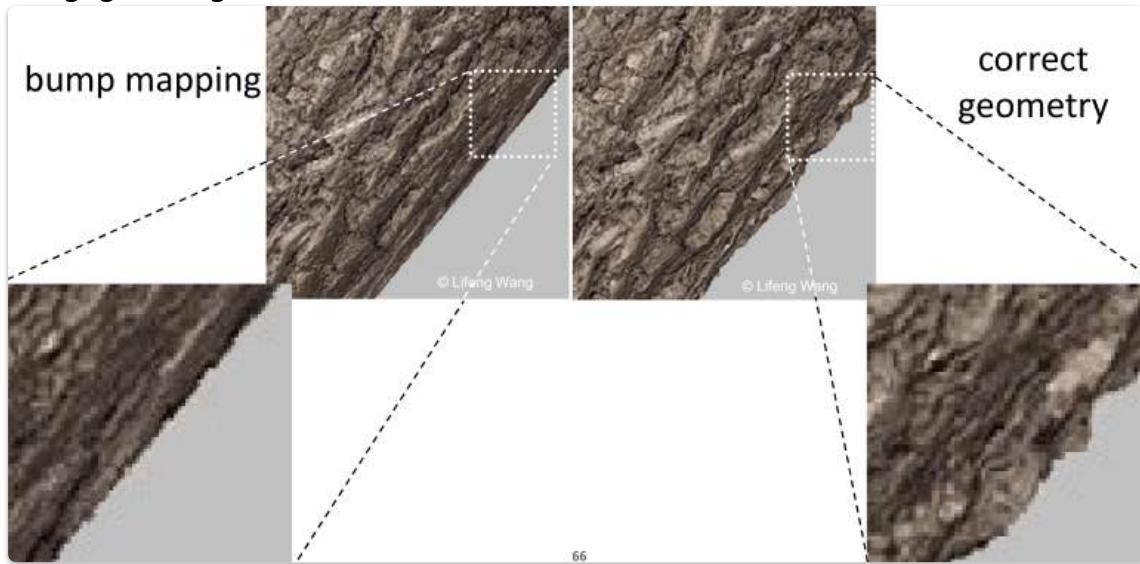
Displacement Mapping

Während Bump Mapping eine Vielzahl von Fehlern aufweist (wie in den vorherigen Notizen beschrieben), gibt es für jeden dieser Fehler mehr oder weniger aufwendige Hilfslösungen. Die **korrekteste Methode** zur Darstellung von Oberflächenunebenheiten ist jedoch, die **Oberfläche tatsächlich um die "Bump-Höhe" zu verändern**. Diese Methode wird als **Displacement Mapping** bezeichnet.



Funktionsweise und Vorteile:

- **Tatsächliche Geometrieverziehung:** Bei Displacement Mapping werden die Oberflächenpunkte tatsächlich verschoben. Das bedeutet, dass die geometrischen Koordinaten der Scheitelpunkte (Vertices) des Modells verändert werden, um die Unebenheiten zu repräsentieren.
- **Korrekte Silhouette:** Da die Geometrie physisch verändert wird, entsteht natürlich auch eine korrekte Silhouette des Objekts. Die Ränder des Objekts zeigen nun die tatsächlichen Unebenheiten und nicht mehr die glatte Form der ursprünglichen Geometrie.
- **Korrekte Schatten:** Die versetzte Geometrie wirft auch korrekte Schatten, einschließlich der gegenseitigen Schatten der Unebenheiten aufeinander.



Implementierung und Hardware-Unterstützung:

EVC_Skriptum_CG, p.50

- **Aufwand:** Displacement Mapping ist viel aufwendiger zu implementieren und rechenintensiver als Bump Mapping, da es eine tatsächliche Veränderung der Geometrie erfordert (oft durch Hinzufügen vieler neuer Polygone).
- **Hardware-Unterstützung:** Moderne Grafikkarten unterstützen Displacement Mapping jedoch. Dies geschieht typischerweise mittels einer zusätzlichen programmierbaren Hardware-Stufe in der Rendering-Pipeline, die oft als "Tessellation-Stage" bezeichnet wird.

- **Tessellation-Stage:** Diese Stufe befindet sich **zwischen dem Vertex- und dem Pixel-Stage**. Ihre Aufgabe ist es, die Dreiecke direkt auf der Hardware in **viele kleine Dreiecke zu unterteilen** (Tessellation). Jedes dieser neu erzeugten kleinen Dreiecke kann dann entsprechend einer **Displacement Map** verschoben werden, um die feinen Details zu erzeugen.

Zusammenfassend: Displacement Mapping bietet eine physikalisch korrektere Darstellung von Oberflächenstrukturen als Bump Mapping, indem es die Geometrie tatsächlich modifiziert. Dies ist zwar komplexer, wird aber durch moderne GPU-Hardware effizient unterstützt.

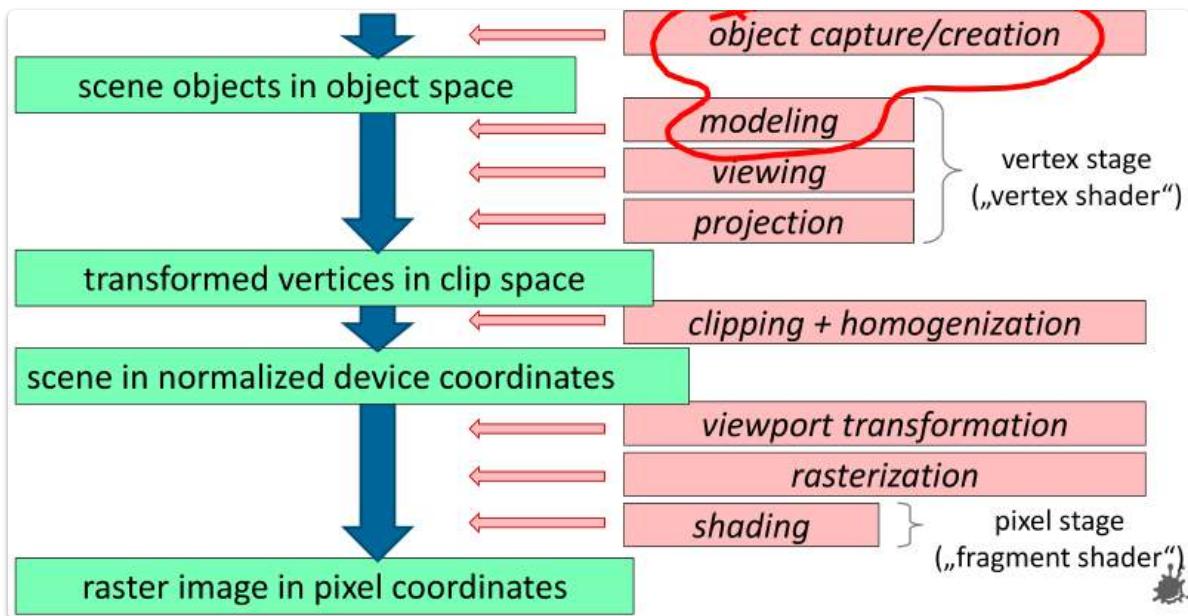
Kombination mehrerer Mappings

EVC_Skriptum_CG, p.50

- **Multitexturing** ist eine mächtige Methode, um *mehrere Mappings zu kombinieren*.
- Beispiele für kombinierbare Texturen sind:
 - Grundmuster (z.B. Holzmaserung)
 - Beleuchtung (z.B. Lichtkegel)
 - Verschmutzung
 - Unebenheiten
 - Fotos plus Annotationen (zusätzliche Informationen oder Markierungen auf einem Bild)



12. Kurven und Flächen

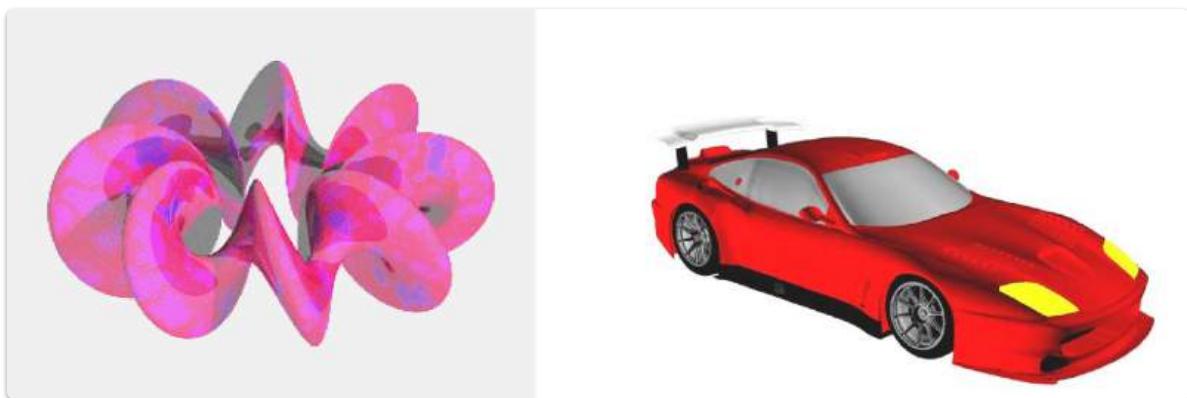


Freiformflächen (Modellierung und Darstellung)

[EVC_Skriptum_CG](#), p.51

Viele Anwendungen in der Computergrafik benötigen nicht nur elementare geometrische Formen, sondern auch die Möglichkeit, **beliebige allgemeine Flächen (Freiformflächen)** zu modellieren und darzustellen.

Die Prinzipien dafür werden oft zuerst anhand von Freiformkurven erläutert, da die zugrunde liegende Mathematik hier etwas einfacher ist. Die daraus entwickelten Verfahren lassen sich jedoch leicht auf Flächen erweitern.



Quadratische Flächen

[EVC_Skriptum_CG](#), p.51

Häufig verwendete Flächen können durch mathematische Formeln definiert werden. Man spricht hier von **analytischer Darstellung**. Es gibt drei Hauptarten der Definition:

1. Implizite Repräsentation:

- Bei dieser Darstellung liegen alle Punkte (x, y, z) , die eine bestimmte Formel erfüllen, auf der Oberfläche der Fläche.
- Beispiel Kugel:** $x^2 + y^2 + z^2 = r^2$.
- Intuition:** Man kann sich das vorstellen wie eine "Filterfunktion". Wenn man einen Punkt (x, y, z) in die Formel einsetzt und die Gleichung erfüllt ist, dann liegt der Punkt auf der Oberfläche. Wenn nicht, dann nicht.

2. Explizite Repräsentation:

- Eine Darstellung ist explizit, wenn ein Koordinatenwert (typischerweise z) von den anderen Koordinaten (x, y) abhängig dargestellt wird.
- Beispiel Kugel:** $z = \sqrt{r^2 - x^2 - y^2}$.
- Intuition:** Man kann sich das als eine "Funktion" vorstellen: Für jedes gegebene x und y berechnet man direkt das zugehörige z auf der Oberfläche. Dies funktioniert aber nur für Oberflächen, die für jedes (x, y) genau einen (oder eine begrenzte Anzahl von) z -Wert(e) haben.

3. Parametrische Repräsentation:

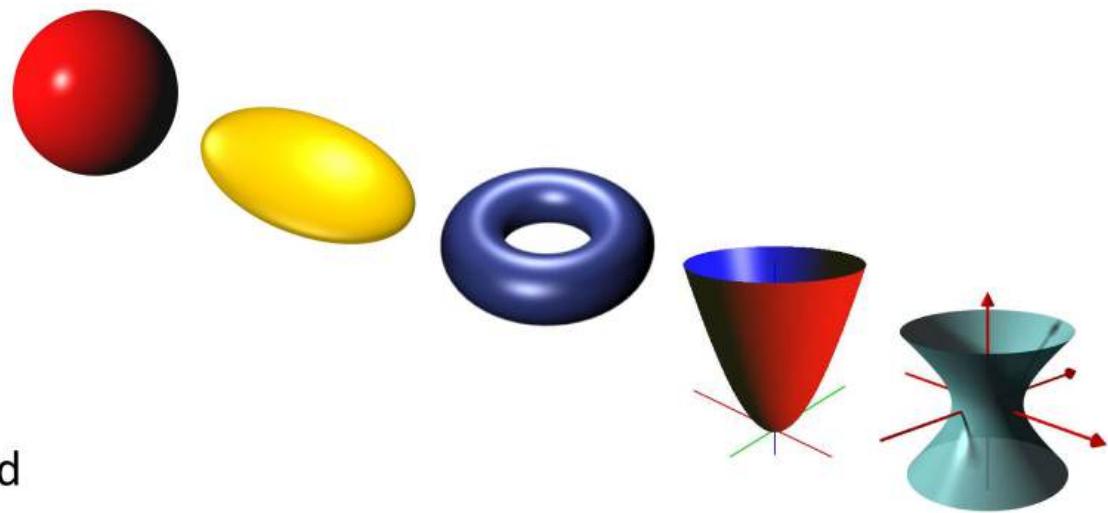
- Bei dieser Darstellung wird für jede Kombination von **Parameterwerten** (z.B. u, v oder ϕ, θ) ein Punkt auf der Oberfläche erzeugt. Die Koordinaten (x, y, z) sind Funktionen dieser Parameter.
- Beispiel Kugel:**
 - $x = r \cdot \cos \phi \cdot \cos \theta$
 - $y = r \cdot \cos \phi \cdot \sin \theta$
 - $z = r \cdot \sin \phi$
 - mit Parameterbereichen: $-\frac{\pi}{2} \leq \phi \leq \frac{\pi}{2}$ und $-\pi \leq \theta \leq \pi$.
- Intuition:** Hier hat man "Stellschrauben" (ϕ, θ) , die man verstellt. Jede Einstellung dieser Schrauben liefert einen Punkt auf der Oberfläche. Man "zeichnet" die Oberfläche, indem man diese Parameter über ihren Bereich variiert. Diese Methode ist sehr flexibel, da sie auch komplexe Formen darstellen kann, die bei expliziter Darstellung schwierig wären (z.B. eine geschlossene Kugel, bei der z nicht eindeutig von x, y abhängt).

• Weitere häufig verwendete quadratische Flächen:

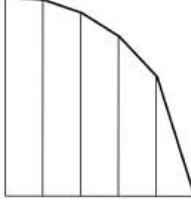
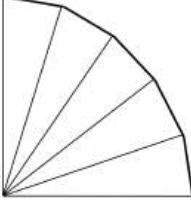
Ellipsoid: Eine gestreckte oder gestauchte Kugelform.

Torus: Eine Donut-Form.

* **Quadratics:** Dies ist ein allgemeiner Begriff für Flächen, die durch eine quadratische Gleichung (zweiten Grades) in x, y, z definiert sind. Dazu gehören Kugeln, Ellipsoide, Paraboloiden, Hyperboloiden usw.



d

$y = f(x)$ <i>axis dependent</i>	$x = f(u)$ $y = g(u)$ <i>axis independent</i>
<i>example:</i> $y = \sqrt{1-x^2}$ 	$x=\cos(u) \quad y=\sin(u)$ 

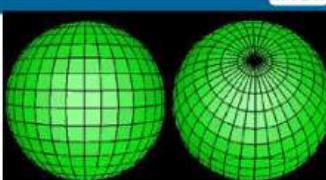
Mehr Formen

Hier nochmal mehr Informationen zu den Formen aus den Slides

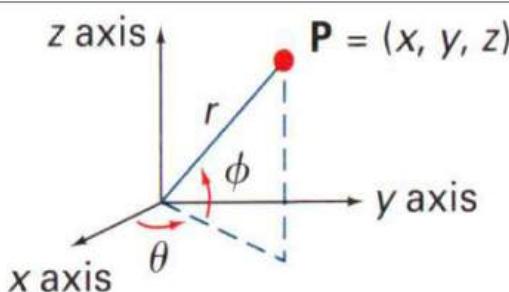
Quadric Surfaces: Sphere

- **implicit:** $x^2 + y^2 + z^2 = r^2$

- **parametric:** $x = r \cos \phi \cos \theta, \quad -\pi/2 \leq \phi \leq \pi/2$
 $y = r \cos \phi \sin \theta, \quad -\pi \leq \theta \leq \pi$
 $z = r \sin \phi$



parametric coordinate position
 (r, θ, ϕ) *on the surface of a*
sphere with radius r

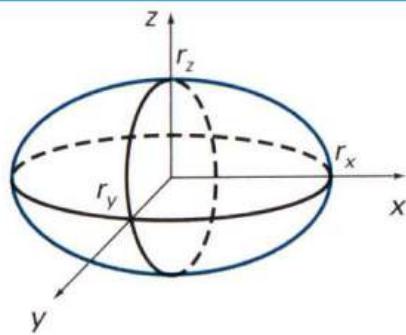


Werner Purgathofer

Quadric Surfaces: Ellipsoid

■ *implicit:*

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1$$



■ *parametric:*

$$x = r_x \cos \phi \cos \theta, \quad -\pi/2 \leq \phi \leq \pi/2$$

$$y = r_y \cos \phi \sin \theta, \quad -\pi \leq \theta \leq \pi$$

$$z = r_z \sin \phi$$

Quadric Surfaces: Torus

■ *implicit:*

$$(R - \sqrt{x^2 + y^2})^2 + z^2 = r^2$$

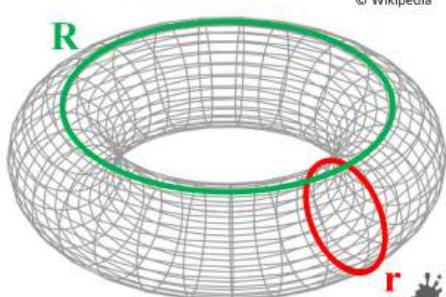


■ *parametric:*

$$x = (R + r \cos \phi) \cos \theta, \quad -\pi \leq \phi \leq \pi$$

$$y = (R + r \cos \phi) \sin \theta, \quad -\pi \leq \theta \leq \pi$$

$$z = r \sin \phi$$



Werner Purgathofer

9

Free Form Surfaces

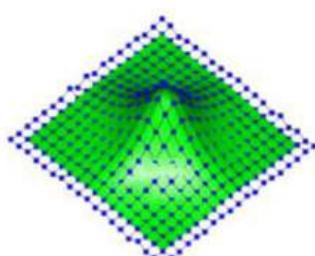
can be represented by

huge number of points (or polygons)

- + arbitrary shapes possible
- large memory requirements
- changes cause much work
- corners after scaling!
- modeling?

mathematical functions

- only for some shape categories
- + marginal memory requirements
- + changes are rather simple
- + definition arbitrarily exact
- modeling!



$$x = f(u,v)$$

$$y = g(u,v)$$

$$z = h(u,v)$$

Kurven

[EVC_Skriptum_CG, p.51](#)

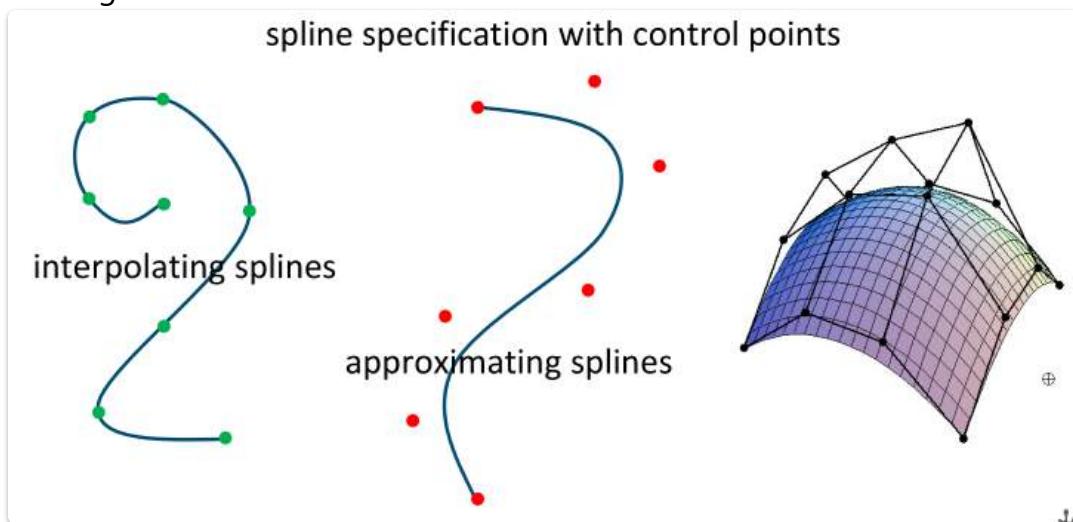
Kurven in der Computergrafik können auf zwei Arten definiert werden:

1. **Analytisch (Formelbasiert):** Die Kurve wird mathematisch durch eine Formel beschrieben.
Diese Methode ist oft weniger intuitiv für den Benutzer.
2. **Punktbasiert (Stütz-/Kontrollpunkte):** Die Kurve wird durch eine Reihe von Punkten geformt, die ihren Verlauf bestimmen. Dies ist die gängigere und benutzerfreundlichere Methode.

Unterscheidende Merkmale von Kurventypen:

Verschiedene Kurvenmodelle können anhand ihrer Eigenschaften klassifiziert werden:

- **Interpolation vs. Approximation:**
 - **Interpolierend:** Die Kurve geht direkt durch alle vorgegebenen Stützpunkte.
 - **Approximierend:** Die Kurve nähert sich den Kontrollpunkten an, berührt sie aber in der Regel nicht direkt.



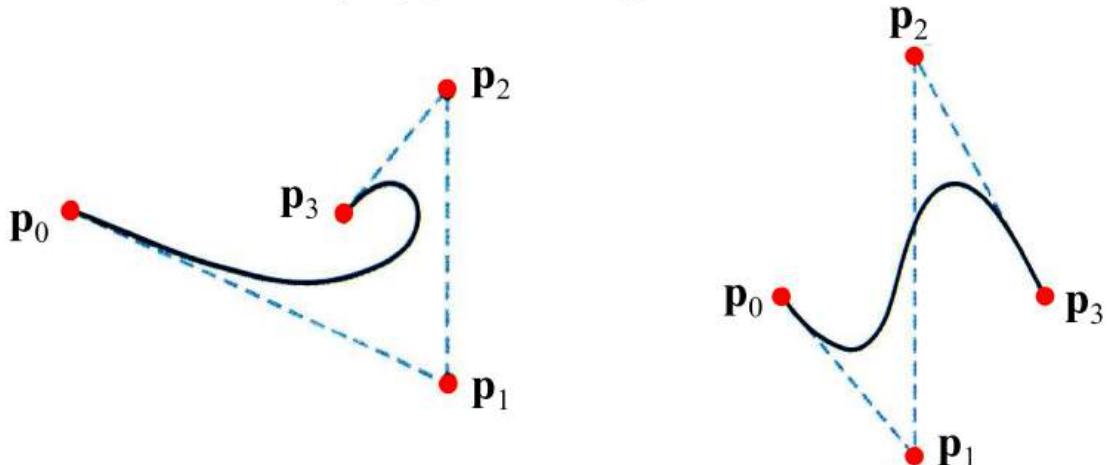
- **Stetigkeit an Verbindungsstellen:** Beschreibt die "Glätte" der Übergänge zwischen einzelnen Kurvensegmenten (z.B. keine Knicke oder abrupte Krümmungswechsel).
- **Einflussbereich von Punkten:**
 - **Globaler Einfluss:** Eine Änderung an einem Punkt wirkt sich auf die gesamte Kurve aus.
 - **Lokaler Einfluss:** Eine Änderung beeinflusst nur einen begrenzten Abschnitt der Kurve.
- **Abhängigkeit vom Koordinatensystem:**
 - **Achsenabhängig:** Die Kurvenform ändert sich, wenn das Koordinatensystem gedreht wird.
 - **Achsenunabhängig:** Die Kurvenform bleibt bei Rotation des Koordinatensystems erhalten.

- **Verhalten bei Richtungswechseln:** Tendenz zu "Dämpfung" (sanfte Übergänge) oder "Überschwingen" (die Kurve schwingt über den Kontrollpunkt hinaus, bevor sie sich annähert).
- **Morphologische Eigenschaften:** Dies umfasst Aspekte wie die möglichen Formen, die eine Kurve annehmen kann, ob sie doppelte Punkte (Schleifen) aufweisen kann oder ob sie sich zu einer geschlossenen Form (z.B. ein Kreis) schließen lässt.

Kurven, die mittels Stütz- oder Kontrollpunkten definiert werden, bezeichnet man allgemein als **Splines**. Im Folgenden werden gängige Spline-Verfahren vorgestellt.

(also called „Characteristic Polygon“)

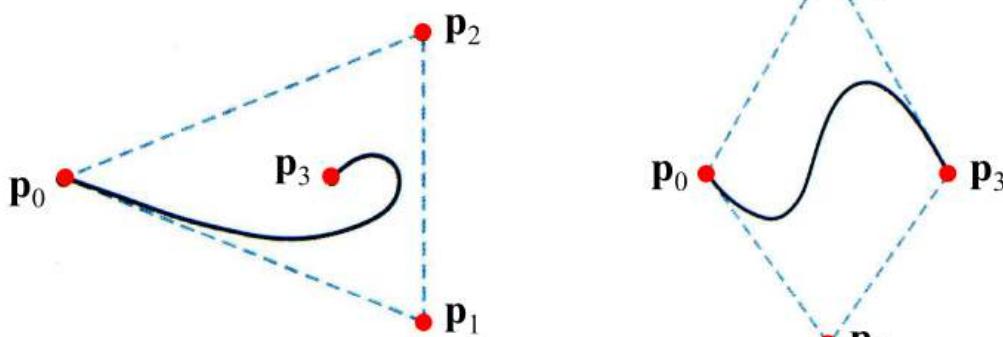
= polygon defining the curve



operations on splines:

- move, insert control points
- spline transformation by transforming all control points

convex hull property

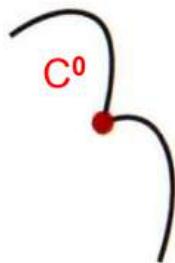


parametric continuity conditions (C^n)

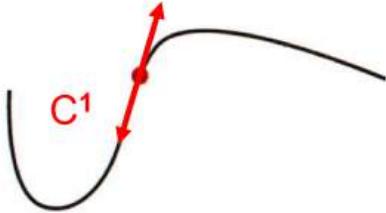
n = number of derivations at section joints that are equal

$$x = x(u) \quad y = y(u) \quad z = z(u) \quad u_{\min} < u < u_{\max}$$

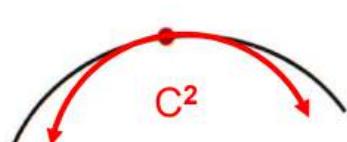
C^0 continuity



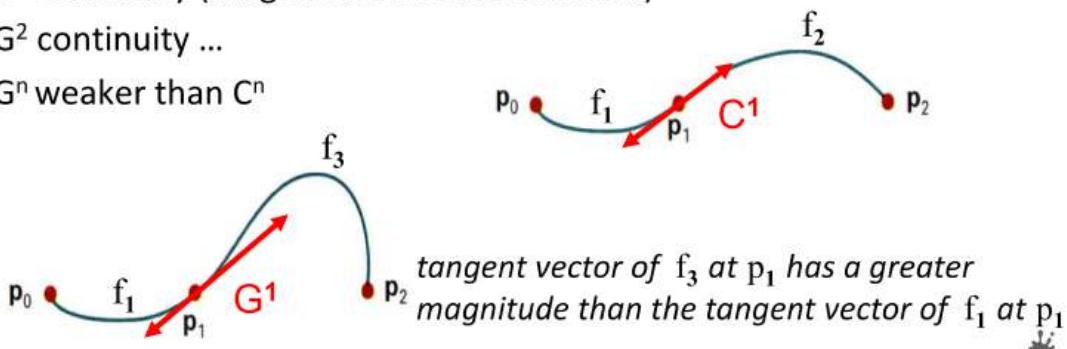
C^1 continuity



C^2 continuity

**geometric continuity conditions (G^n)**

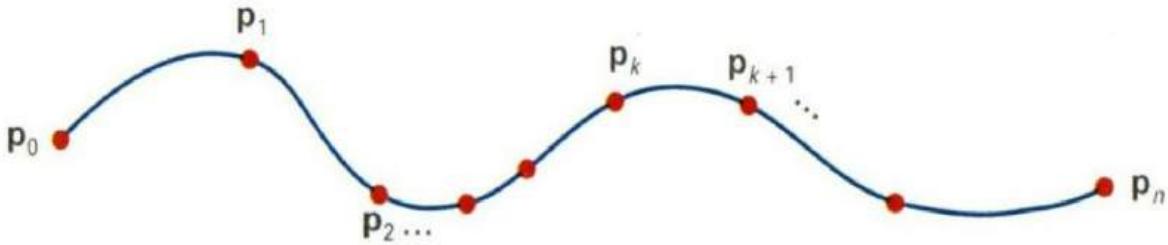
- derivations at joints have different magnitudes
- $G^0 (=C^0)$ continuity
- G^1 continuity (tangent vectors are collinear)
- G^2 continuity ...
- G^n weaker than C^n



Kubische Spline-Interpolation

[EVC_Skriptum_CG, p.52](#)

- Gegeben sind $n + 1$ Stützpunkte $p_i = (x_i, y_i(\dots), i)$, für $i = 0, \dots, n$.
- Eine Interpolationskurve, die zwischen je 2 Stützpunkten aus einem kubischen Polynom besteht, nennt man Kubischen Spline.
- Zwischen Stützpunkten p_k und p_{k+1} wird die Kurve durch einen Parameter u beschrieben:
 - $p_k(u) = a_k u^3 + b_k u^2 + c_k u + d_k$
 - Mit $k = 0, 1, 2, \dots, n - 1$ und $0 \leq u \leq 1$.
 - **Achtung:** a_k, b_k, c_k, d_k sind dabei Vektoren.



- **Berechnung eines Kurvenstücks:**

- Um ein Kurvenstück zwischen 2 Stützpunkten zu berechnen, benötigt man 4 Angabestücke.
- Die Definition an den Stützpunkten ist so gewählt, dass die kubischen Polynome sowohl C^1 -stetig (differenzierbar) als auch C^2 -stetig (zweifach differenzierbar, also gleiche Krümmung) verbunden sind.
- Man spricht dann von *natürlichen kubischen Splines*.
- Diese erhält man, indem man ein Gleichungssystem mit $4n$ Variablen löst.
- Dabei müssen an den Rändern 2 Nebenbedingungen vorgegeben werden, z.B. Krümmung am Anfang und am Ende ist null.
- **Nachteil kubischer Splines:** Jeder Stützpunkt hat einen Einfluss auf den gesamten Kurvenverlauf (globaler Einfluss).

Hermite-Interpolation

[EVC_Skriptum_CG, p.52](#)

- Die Hermite-Interpolation ist eine spezielle Form der kubischen Splines.
- **Besonderheit:** Neben den Stützpunkten p_k werden auch die *Ableitungen* Dp_k an den Stützpunkten vorgegeben.
- Das kubische Interpolationspolynom $p_k(u)$, $0 \leq u \leq 1$, zwischen den Punkten p_k und p_{k+1} lässt sich aus den 4 Bestimmungsstücken eindeutig berechnen:
 - $p_k(0) = p_k$
 - $p_k(1) = p_{k+1}$
 - $p'_k(0) = Dp_k$ (Ableitung am Startpunkt p_k)
 - $p'_k(1) = Dp_{k+1}$ (Ableitung am Endpunkt p_{k+1})
 - Dies gilt für $k = 0, \dots, n - 1$.
- Das Polynom $p_k(u) = a_k u^3 + b_k u^2 + c_k u + d_k$ lässt sich auch in Matrixschreibweise anschreiben.
- Die erste Ableitung dieser Kurve ist $p'_k(u) = 3a_k u^2 + 2b_k u + c_k$.
- Daraus kann man die 4 Bestimmungsstücke $p_k, p_{k+1}, Dp_k, Dp_{k+1}$ formulieren.

$$\mathbf{p}_k(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot \begin{bmatrix} \mathbf{a}_k \\ \mathbf{b}_k \\ \mathbf{c}_k \\ \mathbf{d}_k \end{bmatrix} \quad \mathbf{p}'_k(u) = [3u^2 \quad 2u \quad 1 \quad 0] \cdot \begin{bmatrix} \mathbf{a}_k \\ \mathbf{b}_k \\ \mathbf{c}_k \\ \mathbf{d}_k \end{bmatrix}$$

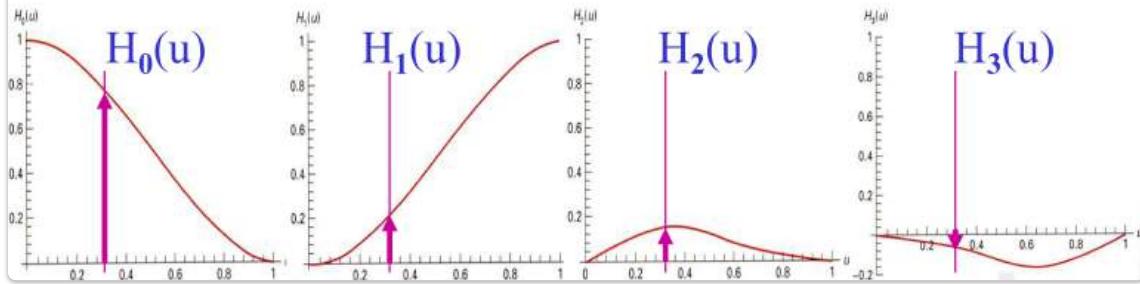
$$\begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{a}_k \\ \mathbf{b}_k \\ \mathbf{c}_k \\ \mathbf{d}_k \end{bmatrix}$$

Um die Koeffizientenvektoren $\mathbf{a}_k, \mathbf{b}_k, \mathbf{c}_k, \mathbf{d}_k$ von $\mathbf{a}_k u^3 + \mathbf{b}_k u^2 + \mathbf{c}_k u + \mathbf{d}_k$ zu berechnen, invertiert man diese Matrix. Die resultierende Matrix heißt Hermite-Matrix \mathbf{M}_H :

$$\begin{bmatrix} \mathbf{a}_k \\ \mathbf{b}_k \\ \mathbf{c}_k \\ \mathbf{d}_k \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}^{-1} \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix}$$

$$\mathbf{p}_k(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot \mathbf{M}_H \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix}$$

$H_k(u)$ blending functions:

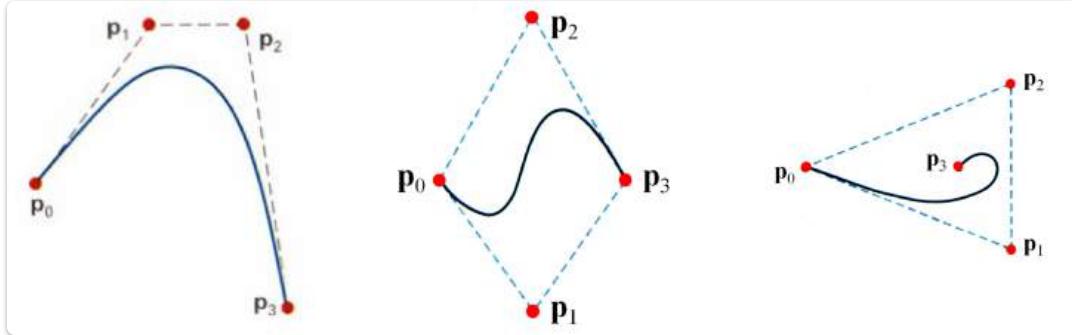


Bézier-Kurven

[EVC_Skriptum_CG, p.52, p.53](#)

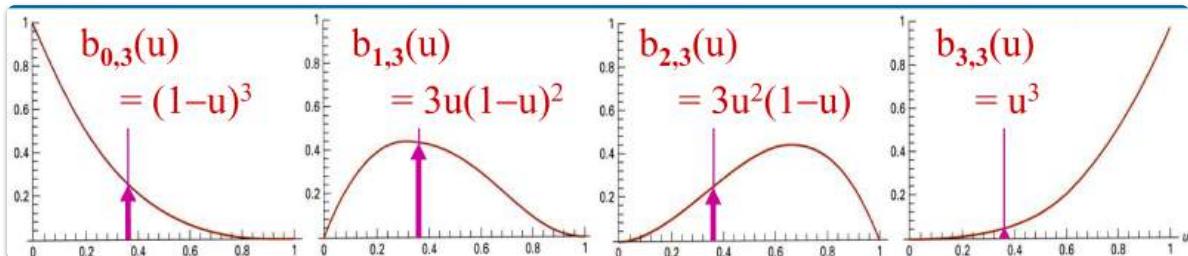
- Von Pierre Bézier um 1960 für die Beschreibung von Autokarosserien bei Renault beschrieben.
- Eine *approximierende* Kurve, die sogenannte *Bernstein-Polynome* $b_{k,n}$ als Gewichtsfunktionen für die Kontrollpunkte verwendet.
- Jeder Kurvenpunkt ist das gewichtete Mittel aller Kontrollpunkte:
 - $p(u) = \sum_{k=0}^n p_k b_{k,n}(u)$, mit $0 \leq u \leq 1$
 - Wobei $b_{k,n}(u) = \binom{n}{k} u^k (1-u)^{n-k}$ die Bernstein-Polynome sind.
- Die Gewichtsfunktionen $b_{k,n}(u)$ sind über einen Parameterbereich u im Bereich von 0 bis 1 definiert.
- Sie hängen von zwei Werten ab:
 - n : Anzahl der Stützpunkte der Kurve (genau genommen gibt es $n+1$ Stützpunkte).
 - k : gibt an, für welchen Stützpunkt die Gewichtsfunktion verwendet wird.
- Jeder dieser Werte (außer an den Rändern $u=0$ und $u=1$) ist größer als Null.
- Da die Summe der $b_{k,n}(u)$ über alle k überall 1 ist, ist jeder Punkt der Kurve $p(u)$ somit ein gewichteter Mittelwert aller Stützpunkte.
- **Beispiel für 4 Kontrollpunkte (also $n = 3$):**

- $p(u) = (1-u)^3 \cdot p_0 + 3u(1-u)^2 \cdot p_1 + 3u^2(1-u) \cdot p_2 + u^3 \cdot p_3$



Eigenschaften von Bézier-Kurven:

- Bei $n+1$ Kontrollpunkten ist $p(u)$ vom Grad n .
- Jeder Kontrollpunkt "zieht" die Kurve wie mit einem Gummiband an.
- Globaler Einfluss:** Gewichtsfunktion fast überall > 0 . (Eine Änderung an einem Kontrollpunkt beeinflusst die gesamte Kurve.)
- p_0 und p_n (Start- und Endpunkt) liegen auf der Kurve.
- Die Tangenten in p_0 und p_n sind die Verbindung zu den nächsten Punkten p_1 und p_{n-1} .
- Die Kurve liegt zur Gänze in der *konvexen Hülle* der Kontrollpunkte (die konvexe Hülle ist das kleinste konvexe Polygon, das alle Kontrollpunkte enthält).



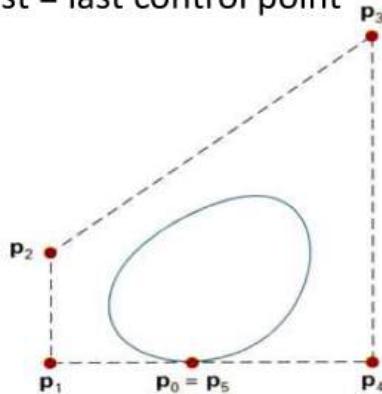
- Einige dieser Eigenschaften erkennt man aus der Form der Bernstein-Polynome $b_{k,n}$.

Weiteres aus den Slides Slides:

Bézier Curves Design Techniques (1)

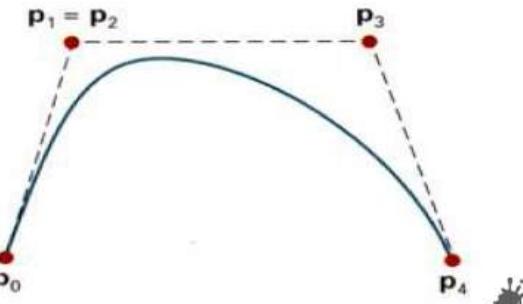
a ***closed Bézier curve***

generated by setting:
first = last control point



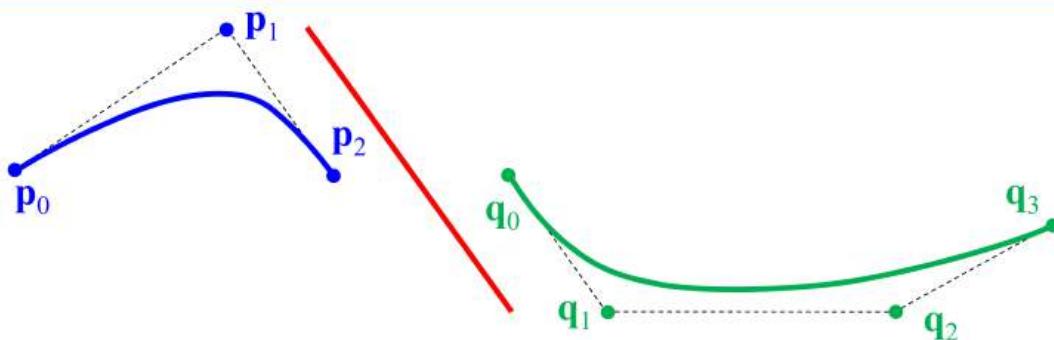
Werner Purgathofer

a Bézier curve can be made to pass closer to a given coordinate position by assigning ***multiple control points*** to that position



30

Bézier Curves Design Techniques (2)



piecewise approximation curve formed with 2 Bézier sections.
0-order and 1st-order continuity (C^0 , C^1 or G^0 , G^1) are attained by setting $q_0 = p_2$ and by making p_1 , p_2 , and q_1 collinear.

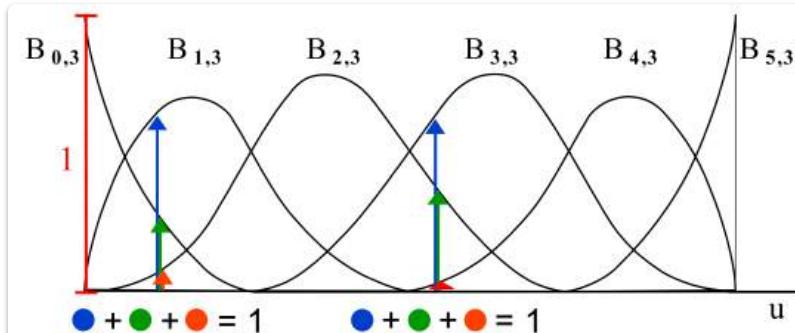
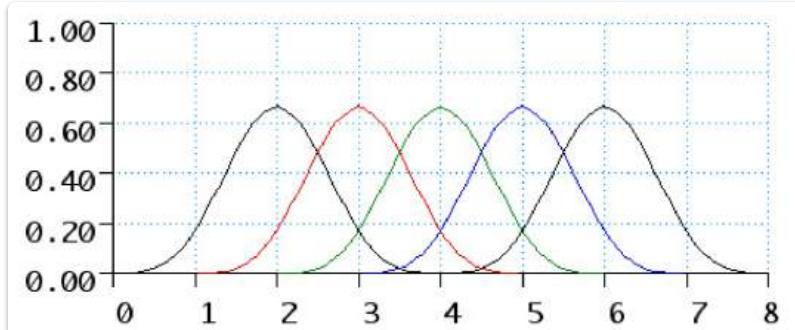
Cubic Bézier Curve Matrix Notation

$$\mathbf{p}(u) = (1-u)^3 \cdot \mathbf{p}_0 + 3u(1-u)^2 \cdot \mathbf{p}_1 + 3u^2(1-u) \cdot \mathbf{p}_2 + u^3 \cdot \mathbf{p}_3$$

$$\mathbf{p}(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot M_{\text{Bez}} \cdot \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix} \quad \text{with} \quad M_{\text{Bez}} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

B-Spline-Kurven

- Der Hauptnachteil der Bézierkurven ist der globale Einfluss der Kontrollpunkte auf die ganze Kurve. Dies hat zwei Hauptnachteile:
 - Jede Veränderung der Kontrollpunkte (Einfügen, Entfernen, Verschieben) verändert das Aussehen der Kurve an allen Stellen.
 - Die Rechenzeit für große Kontrollpunktmenge ist höher.
- Die Ursache liegt in der Form der Gewichtsfunktionen.
- Die sogenannten **B-Splines** sind ebenso wie die Bézier-Splines approximierende Kurven.
- Jedoch sind die Bernstein-Polynome durch **B-Spline-Polynome $B_{k,d}$** ersetzt.
- Diese beschränken die Anzahl der Kontrollpunkte, die einen Kurvenpunkt beeinflussen, auf d .
- Die Berechnung der $B_{k,d}$ ist etwas komplexer und erfolgt rekursiv.
- Für das Verständnis reicht es allerdings, die Form der B-Spline-Polynome zu sehen.**
- Man erkennt, dass jede Gewichtskurve nur in einem begrenzten Bereich ungleich null ist, so dass jeder Punkt über weite Bereiche keinen Einfluss auf die Kurve hat (im Gegensatz zu Bézier-Kurven).



- Eine wichtige Eigenschaft der B-Spline-Gewichtsfunktionen ist die Tatsache, dass (wie bei den Bernstein-Polynomen) für jeden Kurvenpunkt ihre Summe genau 1 ist:

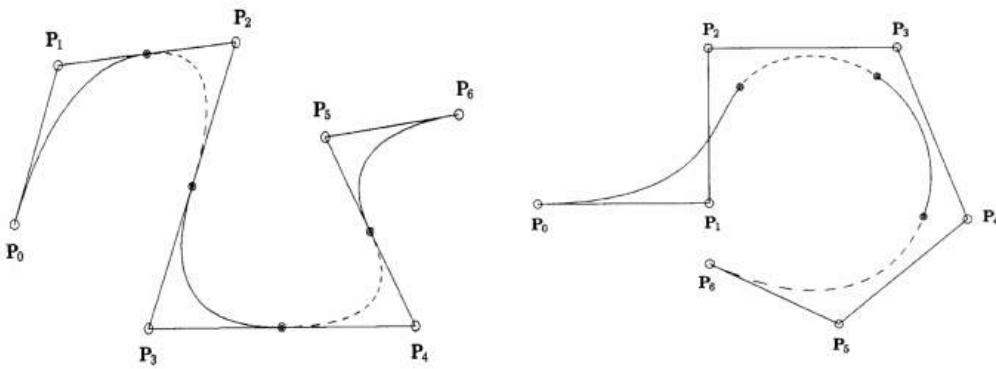
$$\sum_{k=0}^n B_{k,d}(u) = 1$$

- Jeder B-Spline-Kurvenpunkt ist somit ein **gewichteter Mittelwert** aus den Kontrollpunkten.

Beispiele für B-Spline-Kurven

EVC_Skriptum_CG, p.54

- Beispiele für B-Spline-Kurven mit $d = 3$ (links) und $d = 4$ (rechts):**



- Wenn man $d = n + 1$ wählt, erhält man Bézier-Kurven. Diese sind also ein Spezialfall der B-Splines.

Influence of d



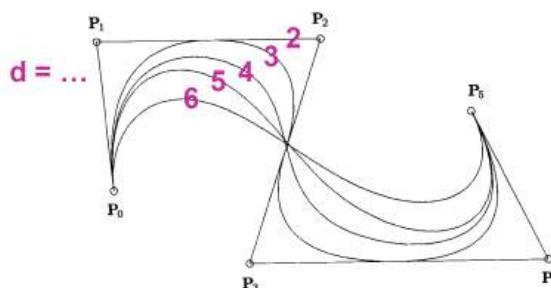
d describes, how many control points influence any point on the curve

$d = 2$ linear

$d = 3$ quadratic

$d = 4$ cubic

...



for $d=n+1$ you get Bézier curves!

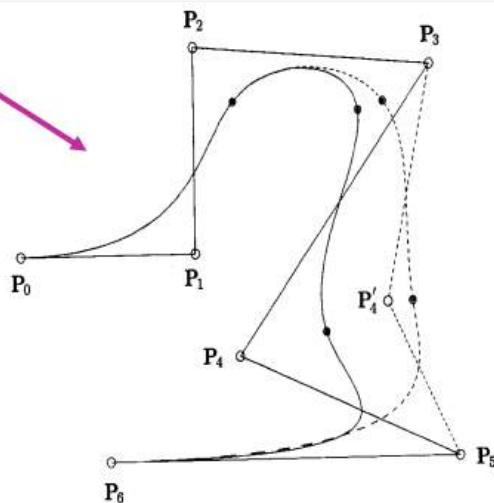
Hauptunterschiede zu Bézierkurven:

EVC_Skriptum_CG, p.54

- **Lokaler Einfluss der Kontrollpunkte** (Änderungen wirken sich nur auf einen begrenzten Bereich der Kurve aus, nicht global).
- **Aufwand linear zur Anzahl der Kontrollpunkte** (statt quadratisch bei Bézierkurven, was die Berechnung bei vielen Kontrollpunkten effizienter macht).

control points have local influence

effort is linearly dependent on n ,
therefore splitting of huge point
sets not necessary

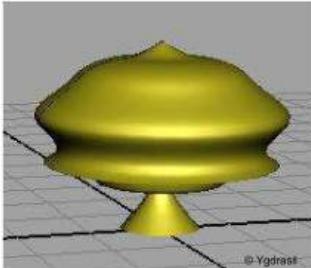


NURBS (Non-Uniform Rational B-Splines)

EVC_Skriptum_CG, p.54

- Die wichtigsten Erweiterungen dieser sogenannten *uniformen* B-Splines führen zu den **Non-Uniform Rational B-Splines**, besser bekannt als **NURBS**.
- Mit NURBS können auch *regelmäßige geometrische Formen* konsistent repräsentiert werden.
- Wichtiger Hinweis:** Alle beschriebenen Methoden (Splines) gelten gleichermaßen für Punkte im zweidimensionalen und im dreidimensionalen Raum.
- Grundsätzlich beschreiben also alle diese Splines **räumliche Kurven im dreidimensionalen Raum**.

further extension: **Non-Uniform Rational B-Splines = "NURBS"**
allow to combine freeform surfaces with regular surfaces



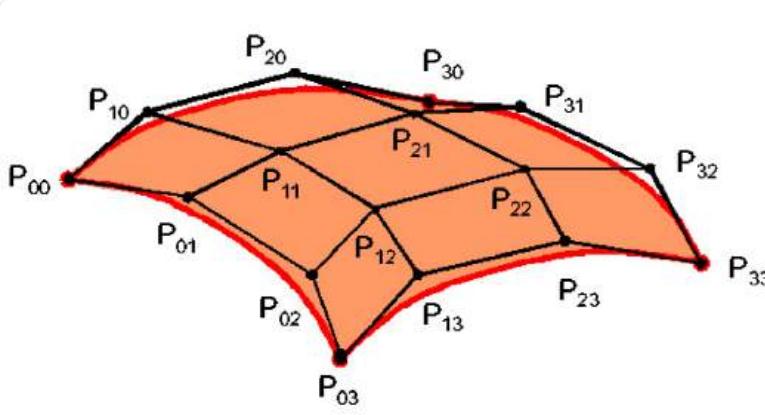
Freiformflächen -- Bézier- und B-Spline-Flächen

[EVC_Skriptum_CG, p.54](#)

- Bildet man über einem zweidimensionalen Punkteraster das kartesische Produkt zweier Kurvenscharen, so erhält man auf natürliche Weise **Freiformflächen**.
- Je nach zugrundeliegender Kurvenart erhält man verschiedene Flächen, z.B. Bézierflächen aus Bézierkurven, B-Splineflächen aus B-Splinekurven usw.
- Formel für Freiformflächen:**

$$p(u, v) = \sum_{j=0}^m \sum_{k=0}^n P_{j,k} b_{j,m}(v) b_{k,n}(u)$$

- Diese Formel zeigt, dass ein Punkt auf der Fläche $p(u, v)$ durch eine doppelte Summe über Kontrollpunkte $P_{j,k}$ und die Produkte von zwei Bernstein-Polynomen (oder anderen Gewichtsfunktionen) bestimmt wird.

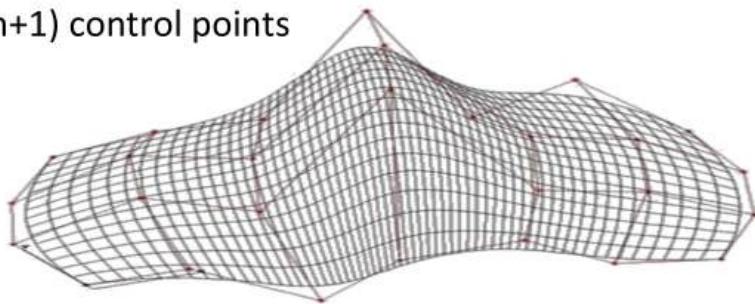


- Jedes Parameterpaar (u, v) führt zu einem Punkt der entstehenden Fläche.

- Die Randkurven der Fläche sind in diesem Beispiel (der Bézierfläche) wieder Bézierkurven.
- Auch die anderen Eigenschaften der Bézierkurven werden auf die Flächen übertragen.
- Analog zu diesen Bézierflächen erhält man B-Spline-Flächen, wenn man die B-Spline-Gewichtsfunktionen in die Formel einsetzt, oder etwa NURBS-Flächen für NURBS-Kurven.

$\mathbf{p}_{j,k}$: grid of $(m+1) \times (n+1)$ control points

just like for
Bezier surfaces!



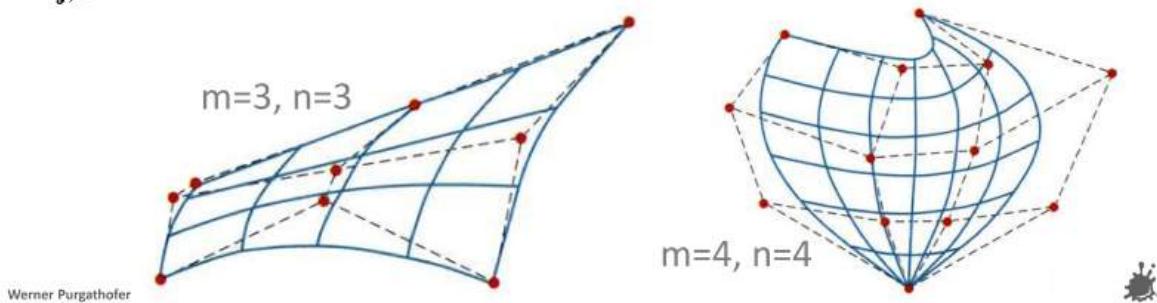
Bézier Surfaces Definition



Cartesian product of two Bézier curve bundles

$$\mathbf{p}(u, v) = \sum_{j=0}^m \sum_{k=0}^n \mathbf{p}_{j,k} b_{j,m}(v) b_{k,n}(u)$$

$\mathbf{p}_{j,k}$: grid of $(m+1) \times (n+1)$ control points

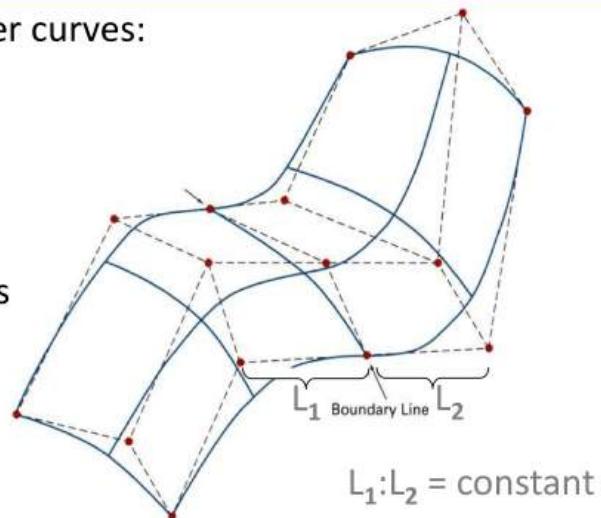


Bézier Surfaces Properties



have the same properties as Bézier curves:

- global influence
- interpolates corner points
- tangents at corner points
- convex hull property
- 1st-order continuity connections



Zeichnen von Freiformflächen:

- Um Freiformflächen zu zeichnen, können **Dreiecksnetze** aus den Flächen erzeugt werden, die wie B-Reps gerendert werden.
- Alternativ werden **Ray-Casting-Methoden** eingesetzt:
 - Dazu muss man eine Prozedur implementieren, die den Schnittpunkt einer Geraden mit der Fläche möglichst genau berechnet.
 - An dieser Stelle wird auch die Oberflächennormale zurückgeliefert (wichtig für Beleuchtungsberechnungen).

13. Computer Animation

Arten von Animation

Flipbook



https://youtu.be/p3q9MM_hM

2D Animation



<https://tenor.com/bV6ph.gif>

Zoetrope



Stop Motion



■ Artistic Control vs. Automation



Time consuming, but flexible

Visual Effects



3D Animation



Transformationen

[EVC_Skriptum_CG](#), p.55

- Eine affine Transformation, die sowohl eine Translation als auch eine Rotation beinhaltet, kann als homogene 4×4 Matrix dargestellt werden (siehe [3. Transformationen](#)):
 - $\begin{pmatrix} R & x \\ 0 & 1 \end{pmatrix}$, mit $R \in \mathbb{R}^{3 \times 3}$ als Rotationale Komponente und Translation $x \in \mathbb{R}^3$.

Translation

[EVC_Skriptum_CG, p.55](#)

- Angenommen, wir haben keine Rotation, dann kann ein Objekt mit der initialen Position $p^{(t_0)} \in \mathbb{R}^3$ zum Zeitpunkt t_0 zu einer Zielposition $p^{(t_1)}$ zum Zeitpunkt t_1 bewegt werden mittels:

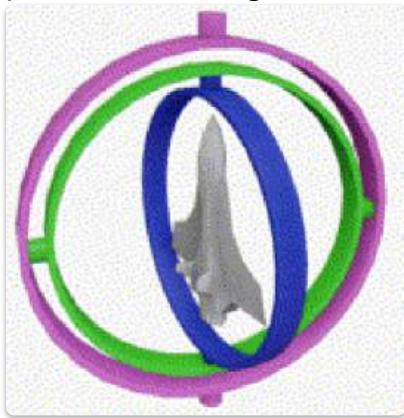
$$\mathbf{p}^{(t_1)} = \begin{bmatrix} 1 & \mathbf{x} \\ 0 & 1 \end{bmatrix} \mathbf{p}^{(t_0)}$$

- Die Position des Objekts zu einem beliebigen Zeitpunkt $t \in \mathbb{R}$, mit $t_0 < t < t_1$, kann zwischen der Start- und Endposition linear interpoliert werden:

$$p^{(t)} = p^{(t_0)} + (p^{(t_1)} - p^{(t_0)}) \frac{t - t_0}{t_1 - t_0}$$

[Rotation](#)[EVC_Skriptum_CG, p.55](#)

- Matrixdarstellungen von Rotationen** sind eine instabile Darstellung und bereiten Probleme, wie z.B. das *Gimbal Lock*. Normalerweise hat man drei Freiheitsgrade, wenn man ein Objekt im 3D-Raum rotiert.
- Verwendet man jedoch Matrizen für Rotationen, gibt es Konfigurationen, bei denen ein Objekt so gedreht wird, dass zwei Achsen parallel sind.
- Im Bild rechts zum Beispiel verklemmen sich der pinkfarbene und der grüne Kreis, und es macht keinen Unterschied mehr, ob man den inneren blauen Ring oder den äußeren pinkfarbenen Ring dreht. Somit haben wir einen Freiheitsgrad verloren.



- Eine Lösung besteht darin, **Quaternionen** zur Darstellung von Rotationen und *sphärische Interpolation* zu verwenden.
- Quaternionen sind eine Erweiterung der komplexen Zahlen und bestehen aus einer skalaren Komponente $s \in \mathbb{R}$ und einer Vektor-Komponente $\mathbf{v} \in \mathbb{R}^3$.
- Um eine beliebige Achse $\mathbf{n} \in \mathbb{R}^3$ um einen Winkel ϕ zu rotieren, kann diese Achsenwinkel-Darstellung einer Rotation in ein Quaternion q wie folgt umgewandelt werden:

- $$q = [s; \mathbf{v}] = [\cos \frac{\phi}{2}; \sin \frac{\phi}{2} \mathbf{n}]$$

(Hierbei ist n ein Einheitsvektor der Rotationsachse.)

- $q^{(t)} = \text{slerp}(q^{(t_0)}, q^{(t_1)}, t) = q^{(t_0)}(q^{(t_0)})^{-1}q^{(t_1)})^t$

(slerp steht für *spherical linear interpolation* und interpoliert zwischen zwei Quaternionen $q^{(t_0)}$ und $q^{(t_1)}$ über die Zeit t .)

- Um schließlich eine rotierte Position $P^{(t)}$ zu einem beliebigen Zeitpunkt t zu berechnen, wenden wir die interpolierte Rotation $q^{(t)}$ auf die Ausgangsposition $P^{(t_0)}$ an.
- Dazu bilden wir zuerst ein neues Quaternion, indem wir $P^{(t_0)}$ als Vektor-Komponente und Null als skalare Komponente nehmen.
- Das Ergebnis des Produkts mit der berechneten Rotation $q^{(t)}$ ist wiederum ein Quaternion mit Null als skalarer Komponente, und die Vektor-Komponente ist unsere gewünschte, rotierte Position $P^{(t)}$:

$$[0; P^{(t)}] = q^{(t)} \cdot [0; P^{(t_0)}] \cdot (q^{(t)})^{-1}$$

- Matrix representation of rotations is often unstable
 - Additional Problem: *Gimbal Lock*
- Best use **Quaternions**



$$\mathbf{q} = [s; \mathbf{v}] = [s \quad \underbrace{\begin{matrix} x & y & z \end{matrix}}_{\text{vector}}]$$

scalar

- Generalization of complex numbers
- Also written as:

$$\mathbf{q} = s + xi + yj + zk$$

with

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

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

■ **Quaternions:** $\mathbf{q} = [s; \mathbf{v}] = [s \quad x \quad y \quad z]$



■ Operations:

■ Addition: $\mathbf{q}_1 + \mathbf{q}_2 = [(s_1 + s_2) \quad (x_1 + x_2) \quad (y_1 + y_2) \quad (z_1 + z_2)]$

■ Multiplication: $\mathbf{q}_1 \cdot \mathbf{q}_2 = \begin{bmatrix} (s_1 \cdot s_2 - x_1 \cdot x_2 - y_1 \cdot y_2 - z_1 \cdot z_2) \\ (s_1 \cdot s_2 + x_1 \cdot x_2 + y_1 \cdot y_2 - z_1 \cdot z_2) \\ (s_1 \cdot s_2 - x_1 \cdot x_2 + y_1 \cdot y_2 + z_1 \cdot z_2) \\ (s_1 \cdot s_2 + x_1 \cdot x_2 - y_1 \cdot y_2 + z_1 \cdot z_2) \end{bmatrix}$

■ Inverse of normalized Quaternion: $\mathbf{q}^{-1} = \bar{\mathbf{q}} = [s \quad -x \quad -y \quad -z]$

- Can be computed from angle ϕ and normalized vector \mathbf{n} (*axis-angle representation*):

$$\mathbf{q} = \left[\cos \frac{\phi}{2}; \sin \frac{\phi}{2} \mathbf{n} \right]$$

■ Spherical Interpolation:

$$\mathbf{q}^{(t)} = \text{slerp}(\mathbf{q}^{(t_0)}, \mathbf{q}^{(t_1)}, t) = \mathbf{q}^{(t_0)} (\mathbf{q}^{(t_0)-1} \mathbf{q}^{(t_1)})^t$$

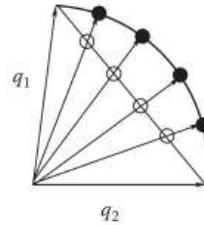


Figure taken from Shirley&Manzner:
Foundations of Computer Graphics, 5th Edition

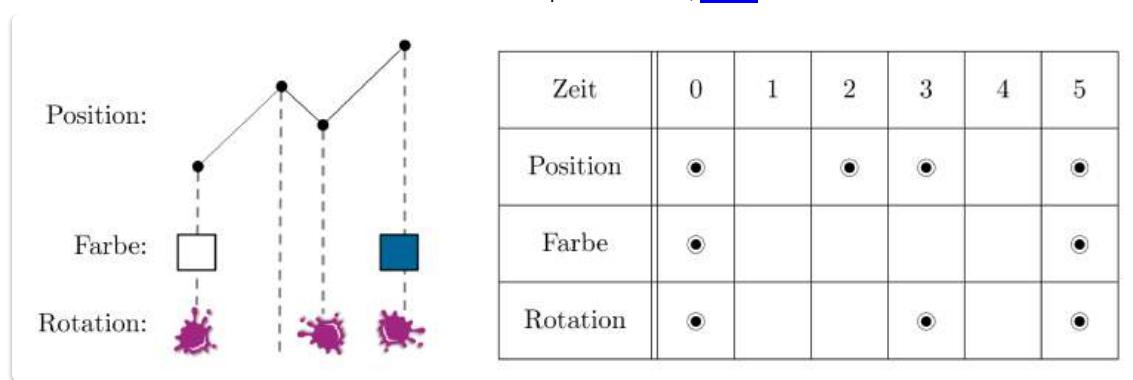
■ Apply a rotation to a point $\mathbf{p}^{(t)}$:

$$[0; \mathbf{p}^{(t)}] = \mathbf{q}^{(t)} \cdot [0; \mathbf{p}^{(t_0)}] \cdot \mathbf{q}^{(t)-1}$$

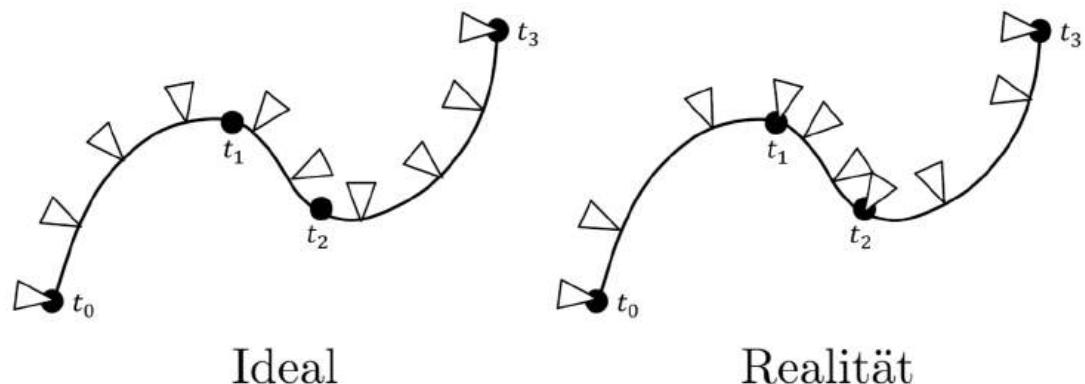
Keyframing

EVC_Skriptum_CG, p.55, p.56

- **Keyframing** ermöglicht komplexe Animationen mit einem Gleichgewicht zwischen Automatisierung und manueller Spezifikation.
- Szenenparameter werden nur zu bestimmten Zeitpunkten (*Keyframes*) festgelegt und sonst interpoliert.
- Parameteränderungen im Laufe der Zeit können in einer Tabelle kodiert werden.
- Die Zeitschritte bezeichnen wir als *Frames*.
- Jede Kombination aus einer *Zeit t* und einer Menge an *Szenenparametern f* zu diesem Zeitpunkt nennen wir einen *Keyframe k*.
- In unserem Fall besteht diese Menge von Szenenparametern *f* aus der Position *p*, Farbe *c* und einer Rotation, die durch ein Quaternion *q* dargestellt wird.
- Dies ergibt das Keyframe *k*: $(t_k, f^{(t_k)}) = (t_k, (p^{(t_k)}, c^{(t_k)}, q^{(t_k)}))$.
- Es kann auch nur eine *Teilmenge* der Szenenparameter angegeben werden, wie im Beispiel für Frame 2 und 3.



- Das Anpassen einer interpolierenden Kurve durch alle Keyframe-Positionen führt zu einer flüssigeren Bewegung anstelle einer linear Positionsänderung.
 - Je nach verwendeter Methode zur Konstruktion der Kurve können jedoch *plötzliche Sprünge* um die Keyframe-Positionen herum auftreten und die Geschwindigkeit, die das Objekt entlang der Kurve hat, kann *unregelmäßig* verlaufen (schlecht für eine gleichmäßige Kamerabewegung).
 - Idealerweise würden wir die Zeit gleichmäßig abtasten und erwarten, dass die resultierenden Punkte entlang der Kurve ebenfalls gleichmäßig verteilt sind.
 - Die meisten Kurvenberechnungsverfahren gruppieren jedoch die räumlichen Abtastpunkte nicht gleichmäßig.
 - **Problem:** Ungleichmäßige räumliche Verteilung der Abtastpunkte entlang der Kurve.
 - **Lösung für dieses Problem:** Die Zeit nicht gleichmäßig abtasten. Stattdessen bestimmen wir die *Länge der Kurve* (Bogenlänge) näherungsweise und teilen diese dann gleichmäßig ab. Dies bedeutet, dass wir die Kurve in Abschnitte gleicher Länge unterteilen.

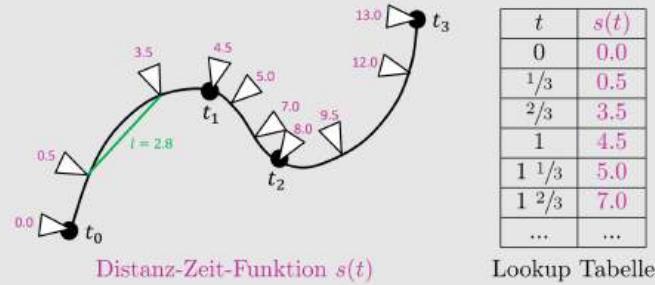


- Um dies zu erreichen, führen wir die **Distanz-Zeit-Funktion** $s(t) : \mathbb{R} \rightarrow \mathbb{R}$ ein.
 - Sie gibt im Grunde an, wie weit wir entlang einer Kurve bereits gereist sind, d.h. sie ordnet jedem Zeitpunkt die bis dahin zurückgelegte Entfernung zu.
 - Während $s(t)$ ursprünglich kontinuierlich ist, können wir sie diskretisieren und die Entfernung zwischen zwei Zeitpunkten durch euklidischen Distanz approximieren.
 - Somit approximieren wir die Form der Kurve mittels mehrerer linearer Abschnitte:
 - $s(t_i) \approx s(t_{i-1}) + \|P_i - P_{i-1}\|$ (Die Distanz zum Zeitpunkt t_i ist die Summe der vorherigen Distanz und der euklidischen Distanz zwischen dem aktuellen und vorherigen Punkt.)

- Wir können diese approximierten Werte unserer Distanz-Zeit-Funktion s in einer *Lookup-Zeit-Tabelle* speichern.
- Wenn wir dann $s(t)$ gleichmäßig abtasten, berechnen wir die Werte t , die wir benötigen, um unsere Kurve unter Verwendung dieser Lookup-Tabelle und linearer Interpolation auszuwerten.
- Durch nicht gleichmäßiges Abtasten der Zeit haben wir somit eine *ungefähr gleichmäßige Raumabtastung* erreicht.

Beispiel:

Im Folgenden nehmen wir an, dass wir eine Methode haben, um die **genaue Distanz-Zeit-Funktion** zu berechnen, was zu den **rosa Werten** im untenstehenden Bild führt. Ein Beispiel zur Berechnung einer Approximation von $s(t)$ zur Zeit $t = 0.6$ ist rechts, wobei die **euklidische Distanz** zwischen $\mathbf{p}_{0.3}$ und $\mathbf{p}_{0.6}$ verwendet wird, von der wir ebenfalls annehmen, dass sie gegeben ist.

Approximiere $s(t)$:

$$s(t_{0.6}) \approx s(t_{0.3}) + l = 0.5 + 2.8 = 3.3$$

Indem wir unsere Approximation von 3.3 mit dem richtigen Wert von 3.5 aus der Abbildung rechts vergleichen, stellen wir fest, dass wir einen Fehler 0.2 gemacht haben.

Als nächstes **tasten wir $s(t)$ gleichmäßig ab, z.B. bei 0.0, 1.0, 2.0, ... und berechnen mittels linear Interpolation die Werte t , an denen wir unsere Kurve auswerten wollen**. Zum Beispiel, wenn wir die Position eines Punktes entlang der Kurve nach der Distanz $s = 6.0$ berechnen wollen, benötigen wir dafür:

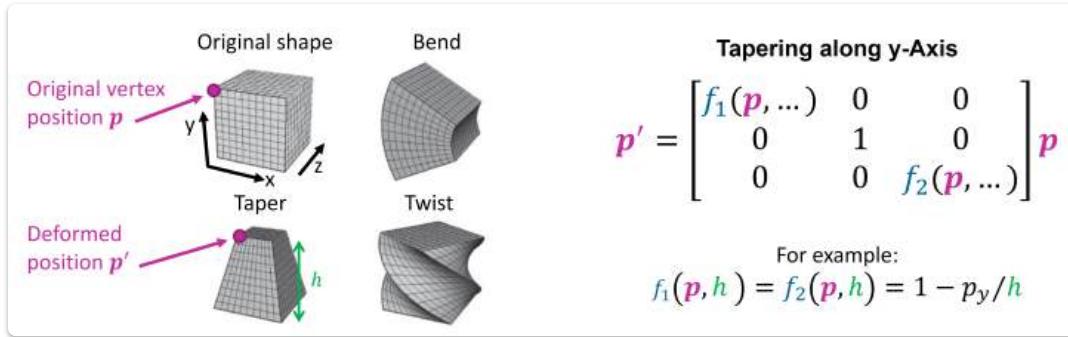
$$t = \frac{1.3 + 1.6}{2} = 2.5, \quad \text{da } 6.0 \text{ in der Mitte von } 5.0 \text{ und } 7.0 \text{ ist (siehe Tabelle).}$$

Deformationen

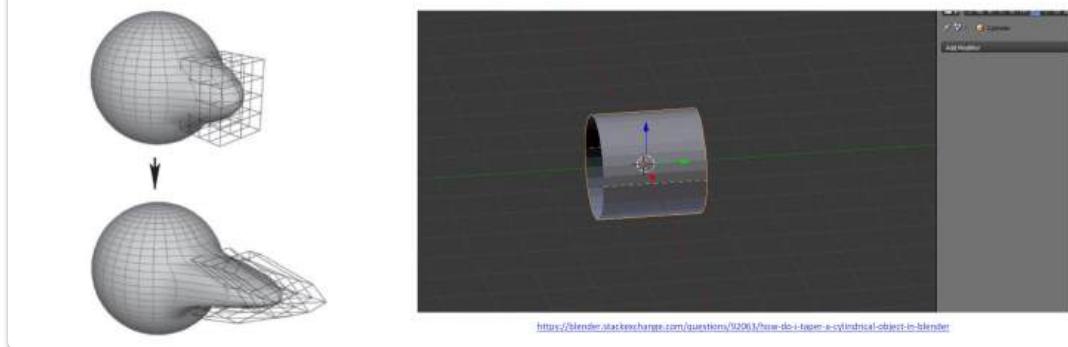
[EVC_Skriptum_CG, p.57](#)

- **Definition:** Eine Deformation kann als Funktion f definiert werden, die auf jede Vertex-Position p angewendet wird.
- Ziel ist es, die neue, deformierte Position $p' = f(p, \gamma)$ zu erhalten.
- γ sind optionale Parameter, die die Deformation steuern.
- **Einfache Deformationen:**
 - Können in Form von *nicht konstanten Matrizen* dargestellt werden.
 - Beispiele:
 - **Verjüngungs-/Taper-Operationen** (Verjüngung eines Objekts)
 - **Biege-/Bend-Operationen** (Biegen eines Objekts)
 - **Verdrehungs-/Twist-Operationen** (Verdrehen eines Objekts)
- **Komplexere Deformationen:**
 - Eine mögliche Lösung ist die Verwendung eines **Deformationskäfigs**.
 - **Vorteil:** Reduziert die Anzahl der manuell festzulegenden Vertex-Positionen erheblich.
 - **Funktionsweise:** Die Vertices (Eckpunkte) des Objekts sind an den Käfig "gebunden".

- **Effekt:** Immer wenn der Käfig bewegt wird, ändert sich auch die Position der Objekt-Vertices.



Free Form Deformation: Define f wrt. a deformation cage



Physikbasierende Simulation

[EVC_Skriptum_CG](#), p.57

Hierbei wird ein Objekt anhand physikalischer Gesetze simuliert und durch mehrere Punkte beschrieben, die über Bedingungen (engl. *Constraints*) verbunden sind.

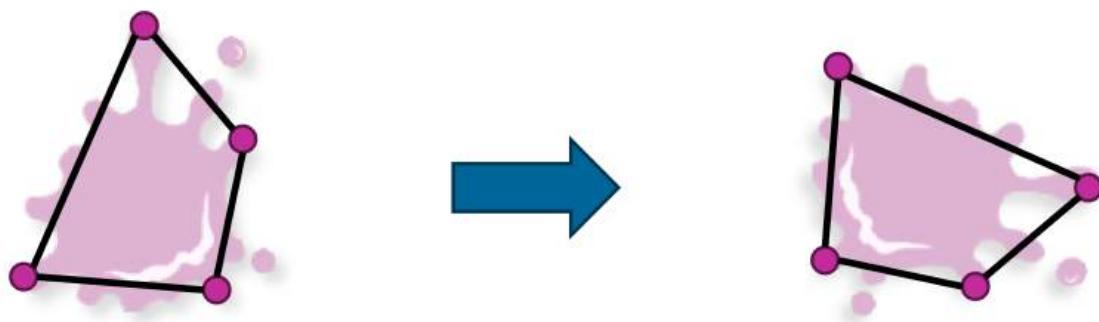
Punkt:

- **Position** $p \in \mathbb{R}^3, [m]$
- **Geschwindigkeit** $v = \frac{dp}{dt} \in \mathbb{R}^3, [m \cdot s^{-1}]$
- **Beschleunigung** $a = \frac{d^2p}{dt^2} \in \mathbb{R}^3, [m \cdot s^{-2}]$
- **Masse** $m \in \mathbb{R}, [kg]$

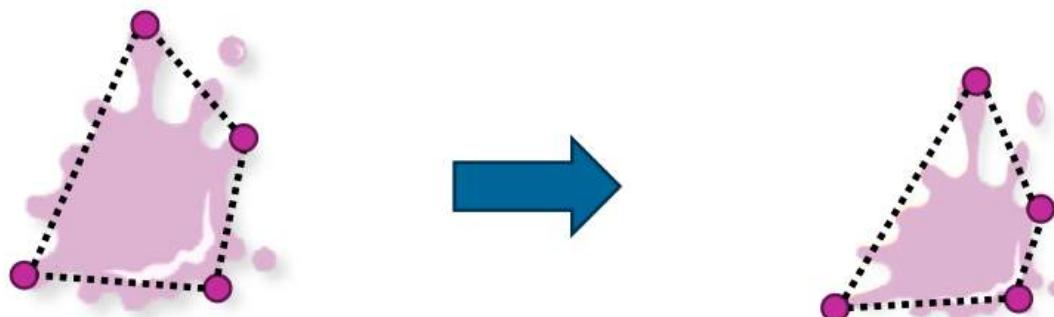
Einschränkung:

- **Starr:** Distanz/Volumen bleibt gleich, nur Transformationen (Holzwürfel, Stahlschwert) (engl. *rigid*)
- **Weich:** Distanz/Volumen variabel, Transformationen & Deformationen (Kleidung, weiches Gewebe) (engl. *soft*)

Unser Hauptinteresse besteht darin, die **Position** p zu simulieren. Dazu betrachten wir ebenfalls die Veränderung der Position über die Zeit, also die **Geschwindigkeit** eines Punktes.



- Distance stays the same
- Transformations only



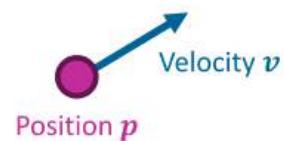
- Distance can change
- Transformations + Deformations

Bewegung bei konstanter Geschwindigkeit

Wenn die Geschwindigkeit v konstant ist (d.h. sie hängt nicht von der Zeit ab), können wir die zukünftige Position $\mathbf{p}^{(t+\Delta t)}$ eines Punktes nach einem beliebigen Zeitschritt Δt basierend auf der aktuellen Position $\mathbf{p}^{(t)}$ zur Zeit t wie folgt berechnen:

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \Delta t \cdot \mathbf{v}$$

- Represent by multiple points
- Velocity \mathbf{v} is the change of position over time



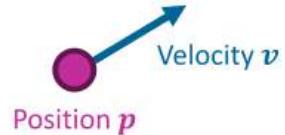
- If there is *constant* velocity, we can thus change the position via:

$$\mathbf{p}^{(t+1)} = \mathbf{p}^{(t)} + \frac{d\mathbf{p}^{(t)}}{dt} = \mathbf{p}^{(t)} + \mathbf{v}$$

Auf diese Weise erfolgt die Bewegung des Punktes mit konstanter Geschwindigkeit entlang einer geraden Linie (**Newtons erstes Gesetz**).

■ With *constant velocity*:

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \Delta t \cdot \mathbf{v}$$



Newton's 1st law

"A body *remains* at rest,
or in motion
at a constant speed
in a straight line,
unless acted upon by a **force**."

[https://commons.wikimedia.org/wiki/File:Portrait_of_Sir_Isaac_Newton,_1689_\(brightened\).jpg](https://commons.wikimedia.org/wiki/File:Portrait_of_Sir_Isaac_Newton,_1689_(brightened).jpg)

Änderung der Geschwindigkeit

Wir können die Geschwindigkeit ändern, indem wir eine Kraft $\mathbb{F} \in \mathbb{R}^3$, [$kg \cdot m \cdot s^{-2}$] wie zum Beispiel die Schwerkraft auf unseren Körper ausüben. Die Änderung der Geschwindigkeit wird als **Beschleunigung** a bezeichnet, die gemäß dem **zweiten Newtonschen Gesetz** berechnet werden kann:

$$\mathbb{F} = m \cdot a \Rightarrow a = \frac{\mathbb{F}}{m}$$

■ If there is *varying* velocity and a *constant* Force:

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \Delta t \cdot \mathbf{v}^{(t)} + \frac{1}{2}(\Delta t)^2 \mathbf{a}$$

$$\mathbf{v}^{(t+\Delta t)} = \mathbf{v}^{(t)} + \Delta t \mathbf{a}$$

$$\mathbf{a} = \frac{\mathbf{F}}{m}$$

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \int_t^{t+\Delta t} \mathbf{v}^{(t')} dt'$$

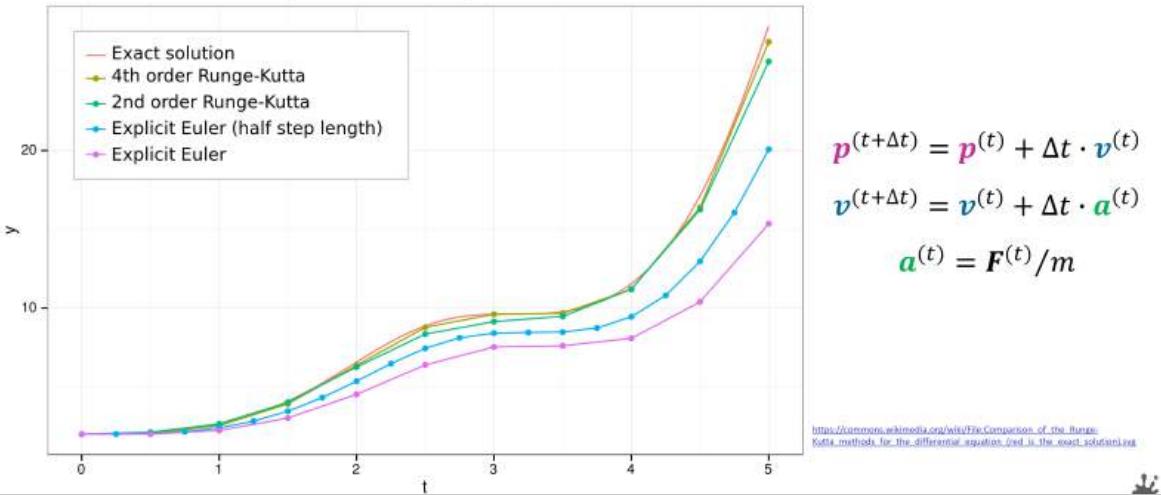
Hard to compute

⇒ use numerical integration methods!

$$\mathbf{a}^{(t)} = \mathbf{F}^{(t)}/m$$

Wenn sowohl eine sich ändernde Geschwindigkeit als auch eine sich ändernde Kraft annehmen, beinhaltet die Berechnung von $\mathbb{P}^{(t+\Delta t)}$ mehrere Integrale.

■ Numerical integration: **Explicit Euler**



- Large timesteps: Unstable
Small timesteps: more computations
- Better explicit integration scheme:
Runge-Kutta
 - More accurate, even with larger timesteps
 - But also involves more computations again
 - Often used in offline settings
or when higher accuracy is required

Numerische Integration

Explicit Euler:

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \Delta t \cdot \mathbf{v}^{(t)}$$

$$\mathbf{v}^{(t+\Delta t)} = \mathbf{v}^{(t)} + \Delta t \cdot \mathbf{a}^{(t)}$$

$$\mathbf{a}^{(t)} = \frac{\mathbf{F}^{(t)}}{m}$$

unstabil bei großen Zeitschritten
(hohe Abweichung/Oszillationen),
einfache Berechnung,
real-time

Implicit Euler:

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \Delta t \cdot \mathbf{v}^{(t+\Delta t)}$$

$$\mathbf{v}^{(t+\Delta t)} = \mathbf{v}^{(t)} + \Delta t \cdot \mathbf{a}^{(t+\Delta t)}$$

$$\mathbf{a}^{(t+\Delta t)} = \frac{\mathbf{F}^{(t+\Delta t)}}{m}$$

bedingungslos stabil,
aufwändige Berechnung
(Lösung eines Gleichungssystems),
offline

Semi-Implicit Euler:

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \Delta t \cdot \mathbf{v}^{(t+\Delta t)}$$

$$\mathbf{v}^{(t+\Delta t)} = \mathbf{v}^{(t)} + \Delta t \cdot \mathbf{a}^{(t)}$$

$$\mathbf{a}^{(t)} = \frac{\mathbf{F}^{(t)}}{m}$$

energiebewahrend,
einfache Berechnung,
real-time

```

1 //Expliziter Euler
2 Vec3 F = berechneKraft();
3 Vec3 beschleunigung = F * objekt.inverseMasse;
4 objekt.position += objekt.geschwindigkeit * deltaZeit;
5 objekt.geschwindigkeit += beschleunigung * deltaZeit;

1 //Semi-Impliziter Euler
2 Vec3 F = berechneKraft();
3 Vec3 beschleunigung = F * objekt.inverseMasse;
4 objekt.geschwindigkeit += beschleunigung * deltaZeit;
5 objekt.position += objekt.geschwindigkeit * deltaZeit;

```

■ Numerical integration: Explicit Euler

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \Delta t \cdot \mathbf{v}^{(t)}$$

$$\mathbf{v}^{(t+\Delta t)} = \mathbf{v}^{(t)} + \Delta t \cdot \mathbf{a}^{(t)}$$

$$\mathbf{a}^{(t)} = \mathbf{F}^{(t)}/m$$

```

Vec3 F = computeForce();
Vec3 acceleration = F * object.inverseMass;
object.position += object.velocity * deltaTime;
object.velocity += acceleration * deltaTime;

```

■ Numerical integration: Semi-Implicit Euler

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \Delta t \cdot \mathbf{v}^{(t+\Delta t)}$$

$$\mathbf{v}^{(t+\Delta t)} = \mathbf{v}^{(t)} + \Delta t \cdot \mathbf{a}^{(t)}$$

$$\mathbf{a}^{(t)} = \mathbf{F}^{(t)}/m$$

```

Vec3 F = computeForce();
Vec3 acceleration = F * object.inverseMass;
object.velocity += acceleration * deltaTime;
object.position += object.velocity * deltaTime;

```

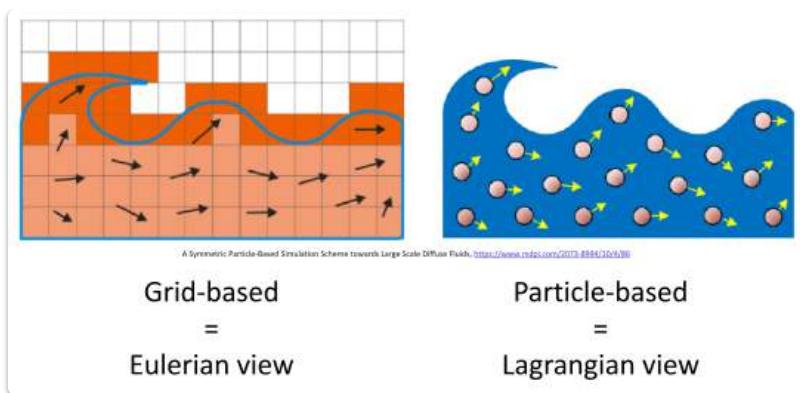
*Simple & more stable
Often used in practice*

Differentialgleichungen

[EVC_Skriptum_CG](#), p.58

Bisher haben wir uns sogenannte **partikelbasierte Simulationen** angesehen, bei denen eine Form durch mehrere Punkte approximiert wird. Dies wird auch als **Lagrange-Sichtweise** der Simulation bezeichnet.

Ein anderer Ansatz besteht darin, den Raum mit einem oft gleichmäßigen Gitter zu unterteilen, was dann als **Euler-Sichtweise** bezeichnet wird.



Letztendlich formulieren wir **Differentialgleichungen**, d.h. Gleichungen, die eine Funktion und ihre Ableitungen enthalten.

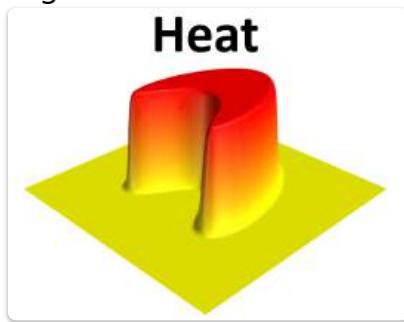
Arten von Differentialgleichungen

Gewöhnliche Differentialgleichungen (ODEs)

- Enthalten nur vollständige Ableitungen (geschrieben als $\frac{dp}{dx}$, $p'(x)$ oder manchmal \dot{p}).
- **Beispiel:** Zweites Newtonsches Gesetz $F(t, \mathbf{p}, \mathbf{v}) = m \frac{d^2\mathbf{p}^{(t)}}{dt^2}$.

Partielle Differentialgleichungen (PDEs)

- Komplexer als ODEs.
- Beschreiben mehrdimensionale Funktionen mit mehreren Parametern und deren partiellen Ableitungen (z.B. $\frac{\partial p}{\partial x}$ oder $\partial_x p$) bezüglich nur eines der Parameter.
- **Beispiel:** Die 2D-Wärmeleitungsgleichung (Bild 2), die beschreibt, wie sich Wärme entlang einer 2D-Oberfläche ausbreitet.
- **Weitere Anwendungen:** Akustische Wellen oder Fluidsimulationen (wobei die zugrunde liegende PDE die Navier-Stokes-Gleichung ist).



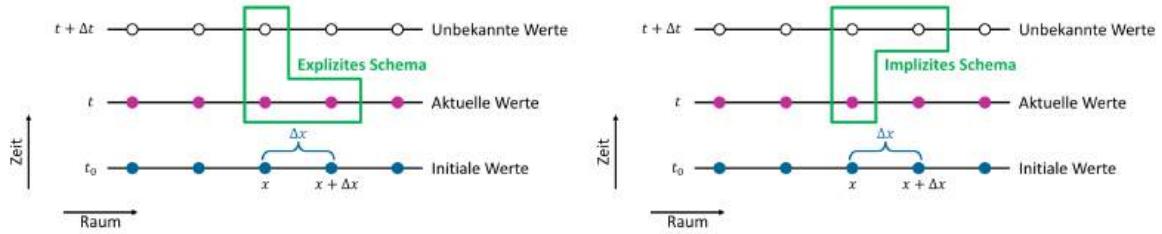
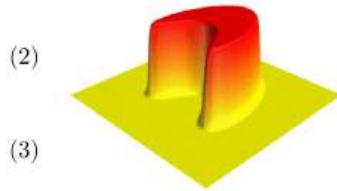
Numerische Berechnung

Eine Technik zur numerischen Berechnung von Ableitungen sind sogenannte **Finite-Differenzen-Verfahren** (Bild 3).

Die unbekannten Werte können wiederum mithilfe von **expliziten** oder **impliziten Verfahren** berechnet werden.

$$\frac{\partial h}{\partial t} = \alpha \left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} \right), \quad \text{with } h : \Omega \subset \mathbb{R}^2 \rightarrow \mathbb{R}$$

$$\frac{\partial u(x, t)}{\partial x} \approx \frac{u(x + \Delta x, t) - u(x, t)}{\Delta x}$$



Charakter Animation

EVC_Skriptum_CG, p.58

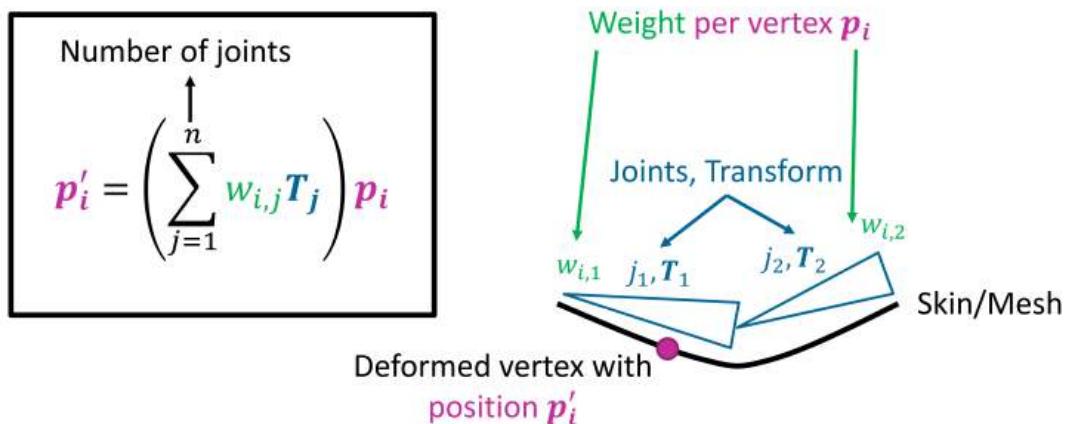
Linear Blend Skinning (LBS) ist eine einfache Methode zur Berechnung aktualisierter Vertex-Positionen basierend auf der Deformation eines zugrundeliegenden Skeletts.

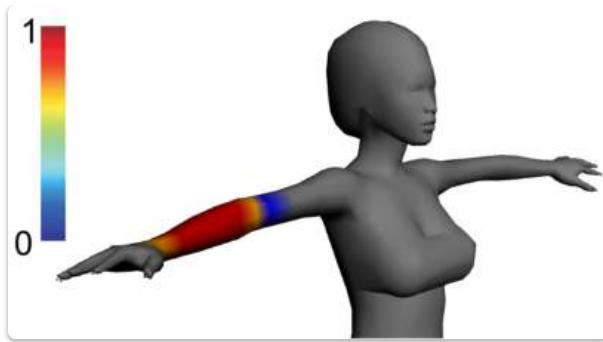
Funktionsweise von LBS

1. **Gelenk-Transformation:** Jedes Gelenk j wird eine Transformation \mathbb{T}_j zugewiesen (die auch von einer Hierarchie von Gelenken abhängen kann und zum Beispiel als Matrix $\mathbb{T}_j \in \mathbb{R}^{4 \times 4}$ dargestellt werden kann).
2. **Einfluss auf Vertices:** Jedes Gelenk j hat einen individuellen Einfluss auf jeden Vertex i , der durch das Gewicht $w_{i,j}$ erfasst wird.
3. **Deformation:** Um einen Vertex i von seiner ursprünglichen Position \mathbb{p}_i zu einer deformierten Position \mathbb{p}'_i zu bewegen, summiert man über die gewichteten Beiträge aller Gelenke (n ist die Anzahl der Gelenke):

$$\mathbb{p}'_i = \left(\sum_{j=1}^n w_{i,j} \mathbb{T}_j \right) \mathbb{p}_i$$

■ Linear Blend Skinning:

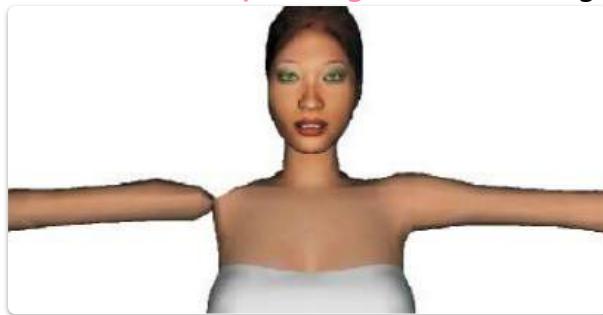




Probleme von LBS

LBS hat jedoch mehrere Probleme, wie zum Beispiel:

- Hoher manueller Aufwand.
- Das „Bonbonverpackungs“-Artefakt (engl. *candy wrapper effect*)



Alternativen zu LBS

Alternativ können **Ragdolls** basierend auf physikalischer Simulation oder **Motion-Capture-Techniken** verwendet werden:

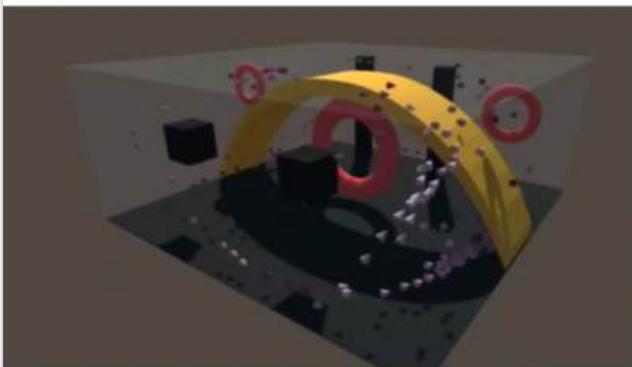
- **Motion-Capture:** Die Bewegungen realer Menschen werden direkt über Kameras und Marker auf den Schauspielern erfasst und dann auf einen virtuellen Charakter übertragen.

Prozedurale Techniken

[EVC_Skriptum_CG](#), p.58

Ähnlich wie physikbasierte Simulationen können **prozedurale Techniken** verwendet werden, um automatisch Animationen zu erstellen.

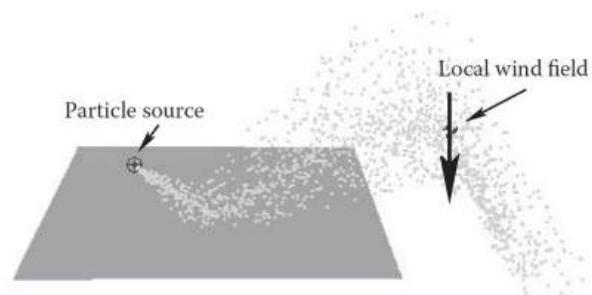
Swarm/Flock/Boids



Game of Live (Cellular automata)



Particle System + Wind Field



Merkmale

- Folgen einer bestimmten Reihe von Regeln, kombiniert mit Zufallsprinzipien.
- Ziel: plausible Animationen zu erstellen, jedoch nicht mit dem Ziel der physikalischen Korrektheit.

Beispiele

- **Game of Life:** Ein bekanntes Beispiel für einen zellulären Automaten.
- **Partikelsysteme:** Für Phänomene wie Regen, Schnee, Feuer.
- **Schwarmverhalten:** Erzeugung von plausiblem Schwarmverhalten (z.B. Vogelschwärme oder Fischschwärme).

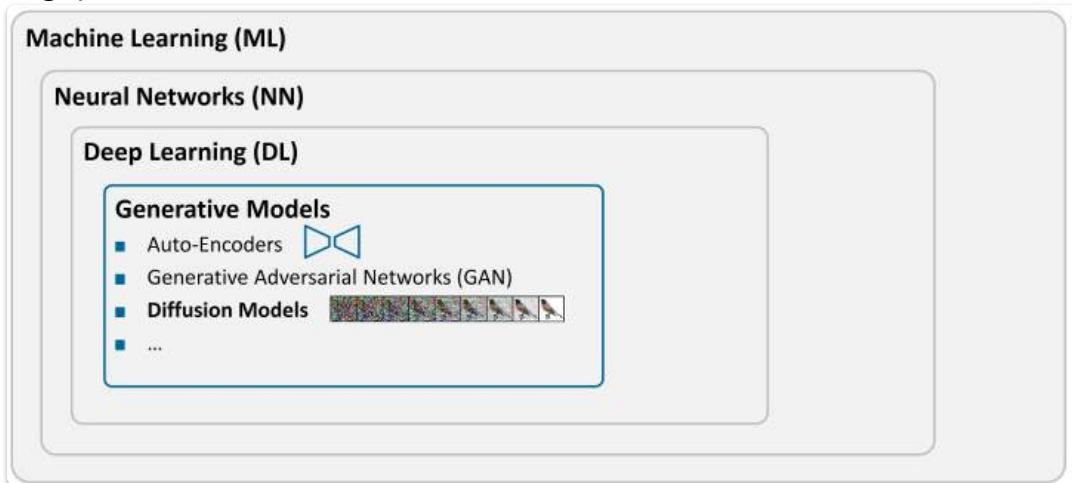
14. Machine Learning für 3D Graphics

Machine Learning und Neural Networks

EVC_Skriptum_CG, p.59

- **Maschinelles Lernen (ML):**

- Ein Zweig der *Künstlichen Intelligenz*.
- Konzentriert sich auf die Entwicklung von Algorithmen, die aus Daten lernen, um Vorhersagen oder Entscheidungen zu treffen.
- Dazu gehört die *Optimierung von Parametern* und Beziehungen innerhalb von Daten, um Modelle zu erstellen.
- *Grundlegendes Beispiel:* Lineare Regression, bei der Parameter an Datenpunkte angepasst werden.



- **Neurale Netze (NN):**

- Grundlegend für ML.
- Bestehen aus *miteinander verbundenen Knoten* (Neuronen).
- Verarbeiten Eingaben, um Ausgaben zu erzeugen.
- Funktionieren mittels *Multiplikation von Eingangsvektoren mit Gewichtsmatrizen* und Anwendung von *Aktivierungsfunktionen* (z.B. *ReLU* oder *Sigmoid*).
- Trotz ihrer Einfachheit sind diese Netze leistungsfähig genug für eine breite Palette an Aufgaben.

- **Deep Learning (DL):**

- Eine *Untergruppe von ML*.
- Verwendet Neurale Netze mit *vielen Parametern*, um komplexe Probleme zu lösen.
- Eignet sich hervorragend für Aufgaben wie:
 - *Bildklassifizierung* (Bilder in Kategorien sortieren).
 - *Segmentierung* (verschiedene Teile eines Bildes identifizieren).



- *Stilübertragungen* (Stil von Bildern ändern, Inhalt beibehalten).
- Synthesierung neuartiger Ansichten (neue 3D-Perspektiven aus begrenzten Daten).
- **Klassifizierer:**
 - Eine Anwendung des maschinellen Lernens.
 - Bei *hochdimensionalen Eingabedaten* (z.B. Bilder) werden diese in eine *niedrigdimensionale Ausgabe* (i.d.R. eine *Klassenbezeichnung*) umgewandelt.
 - Klassenbezeichnungen können zum Beispiel Kategorien wie „*Mensch*“, „*Hund*“ oder „*Vogel*“ sein.
 - Das Trapezsymbol in Klassifizierern steht für diese *Dimensionsreduktion* und zeigt, wie Rohdaten durch den Klassifizierungsprozess zu bestimmten Kategorien verdichtet werden.

Generative Modelle

[EVC_Skriptum\(CG\)_p.59](#)

- **Generative Modelle (GM):**
 - Eine Untergruppe des Deep Learning (DL).
 - Klasse von *statistischen Modellen*.
 - Dienen zur *Erzeugung neuer Dateninstanzen* (z.B. Bilder, Texte), die der ursprünglichen Trainingsverteilung ähneln.
 - Lernen die *Wahrscheinlichkeitsverteilung der Daten*, auf denen sie trainiert wurden.
 - Ermöglichen die Erzeugung neuer Datenpunkte mit *ähnlichen Merkmalen*.

Wichtige Arten von generativen Modellen:

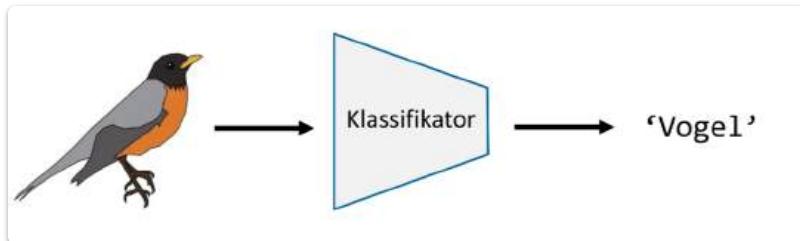
- **Auto-Encoder:**
 - Lernen, Eingabedaten in eine *kleinere Darstellung zu komprimieren*.
 - Rekonstruieren dann die Ausgabe anhand dieser Darstellung (dienen primär der Datenkompression und Merkmalsextraktion, können aber auch generativ eingesetzt werden).
- **Generative Adversarial Networks (GANs):**
 - Bestehen aus einem *Generator*, der Stichproben erzeugt.
 - Bestehen aus einem *Diskriminator*, der diese auf Echtheit auswertet (ein Wettbewerb zwischen Generator und Diskriminatator führt zu immer realistischeren Generierungen).

- **Diffusionsmodelle:**

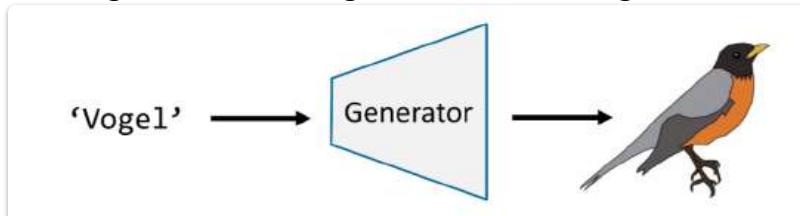
- Nutzen die Idee der *umgekehrten Diffusion* (d.h. Rauschentfernung).
- Erzeugen iterativ eine Ausgabe aus zufälligem Rauschen (beginnen mit Rauschen und entfernen es schrittweise, um ein klares Bild zu erhalten).

Konzeptionelle Vorstellung:

- **Klassifikator (rückwärts):** Ein Klassifikator würde eine hochdimensionale Eingabe (z.B. RGB-Bild) in eine niedrigdimensionale Ausgabe (eine Klassenbezeichnung wie 'Vogel') umwandeln.



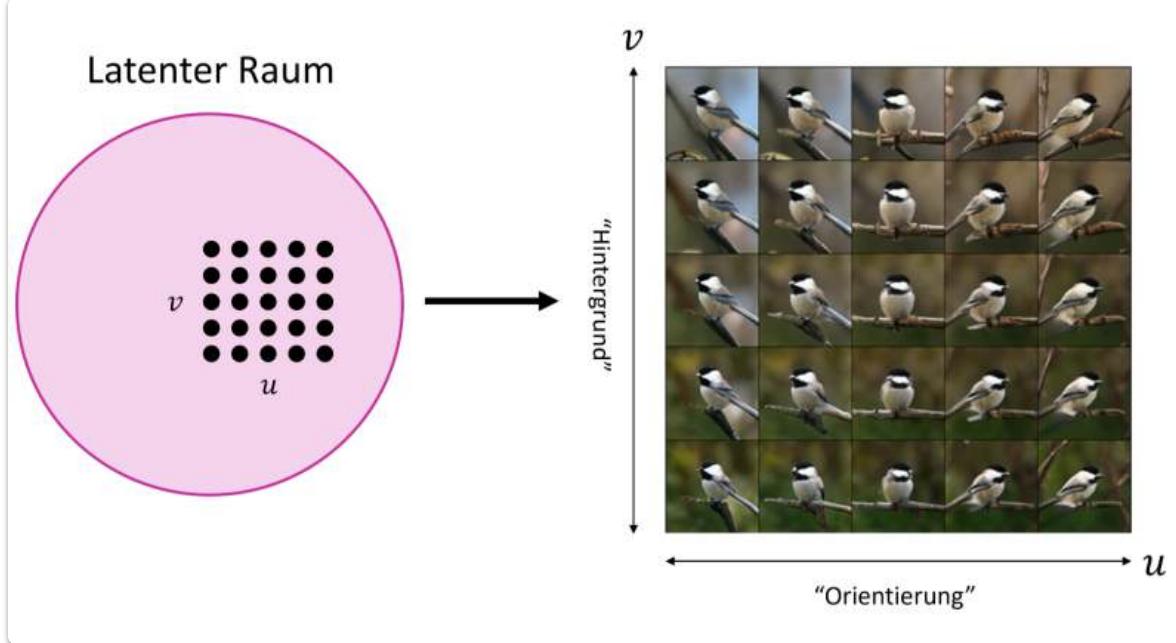
- **Generatives Modell:** Nimmt eine Eingabe mit *geringer Dimensionalität* (z.B. ein Texteingabe oder zufälliges Rauschen) und gibt ein *RGB-Bild* aus.



- Wenn man verschiedene Instanzen von Vögeln generieren und steuern möchte, welche Instanzen erzeugt werden, sind *zusätzliche Eingaben* in den Generator notwendig.
- Um *Variabilität* zu erreichen, wird oft *zufälliges Rauschen* verwendet. Dieses Rauschen dient dazu, eine Verteilung von zufälligen Eingaben auf eine gewünschte Verteilung von Ausgaben abzubilden.

Latenter Raum

EVC_Skriptum_CG, p.60



- Innerhalb des Modells sind die Daten in einem *hochdimensionalen, latenten Raum* enthalten.
- Dieser Raum umfasst Punkte, die durch *latente Variablen* definiert sind.
 - Diese Variablen kodieren die *Merkmale* unseres Vogels wie Farbe und Größe.
- Anfänglich sind diese Variablen *zufällig verteilt*.
- Um bestimmte Attribute der erzeugten Ausgabe besser zu kontrollieren, werden sie *bewusst manipuliert*.
- In einer zweidimensionalen Darstellung dieses Raums könnte beispielsweise:
 - Die Koordinate ' u ' die *Ausrichtung* des Vogels (links, vorne, rechts) vorgeben.
 - Die Koordinate ' v ' den *Hintergrundkontext* (blauer Himmel, grüne Blätter) bestimmen.
- **Variable Verschränkung (variable entanglement):** Latente Variablen sind jedoch nicht immer klar voneinander getrennt. Änderungen in einer Variablen können *unbeabsichtigt auf andere auswirken*.
 - *Beispiel:* Ein nach rechts gerichteter Vogel könnte einen braunen Hintergrund erscheinen lassen.

Generative Modelle für Bilderstellung

[EVC_Skriptum_CG, p.60](#)

- **Notwendigkeit zur Bilderstellung mit generativen Modellen:**
 - **Architektur-Auswahl:** Der erste Schritt ist die Auswahl einer Architektur, z.B. *GANs* oder *Diffusionsmodelle*.
 - **Training:**
 - Verwendet einen *großen Datensatz von Bildern*.
 - Manchmal auch mit *Beschriftungen* (Labels).
 - Ziel: Dem Modell beibringen, *neue Bilder akkurat zu erzeugen*.

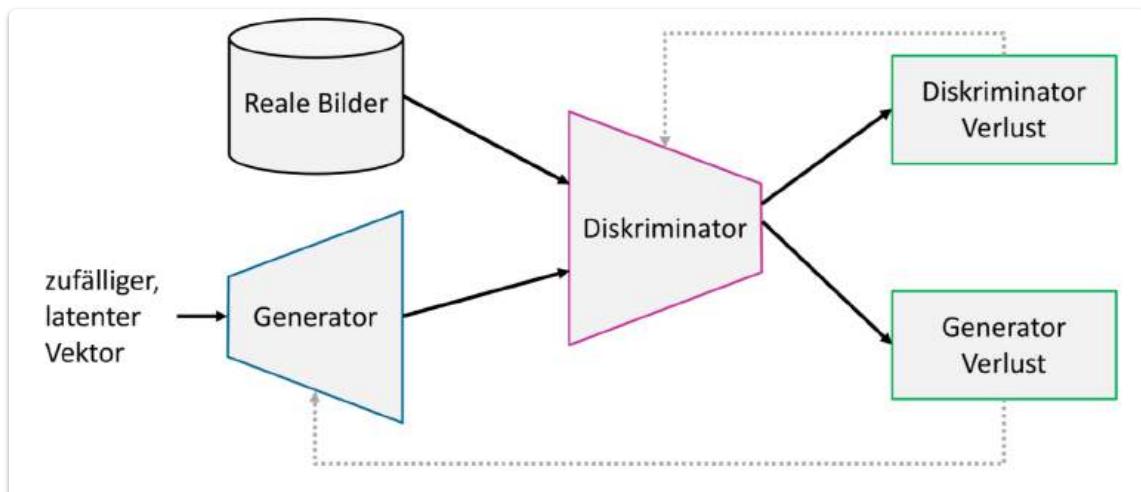
- **Inferenz:**

- Das Modell wendet das *Gelernte an*.
 - Erzeugt *neue Bilder* auf der Grundlage neuer Eingaben.
 - Ermöglicht die *kontrollierte Erzeugung spezifischer Merkmale* in der Ausgabe, wie Farbe und Größe.
-

Training eines Generative Adversarial Network (GAN)

EVC_Skriptum_CG, p.60

- Bei GANs liefern sich *zwei neuronale Netze*, der **Generator** und der **Diskriminator**, einen *Wettstreit* (Adversarial Process).
- **Aufgabe des Generators:**
 - Erzeugt Bilder aus *zufälligen, latenten Vektoren*.
 - Ziel: Die erzeugten Bilder sollen von realen Bildern *nicht zu unterscheiden* sein, um den Diskriminator zu "täuschen".
- **Aufgabe des Diskriminators:**
 - Unterscheidet zwischen den *erzeugten* Bildern (Fakes) und den *realen* Bildern.
- **Verlustfunktionen:**
 - Der Prozess beinhaltet *zwei Verlustfunktionen* (Loss Functions), die *gleichzeitig optimiert* werden.
 - Ziel:
 - Verbesserung der Fähigkeit des Generators zur Täuschung.
 - Verbesserung der Fähigkeit des Diskriminators zur Erkennung von Fälschungen.
 - **Entscheidend:** Die Verlustfunktionen müssen *differenzierbar* sein, um die *Backpropagation* (Anpassung der Gewichte im neuronalen Netz basierend auf dem Fehler) während des Trainings zu ermöglichen.
 - Dies verbessert die Lern- und Anpassungsfähigkeiten beider Netzwerke.



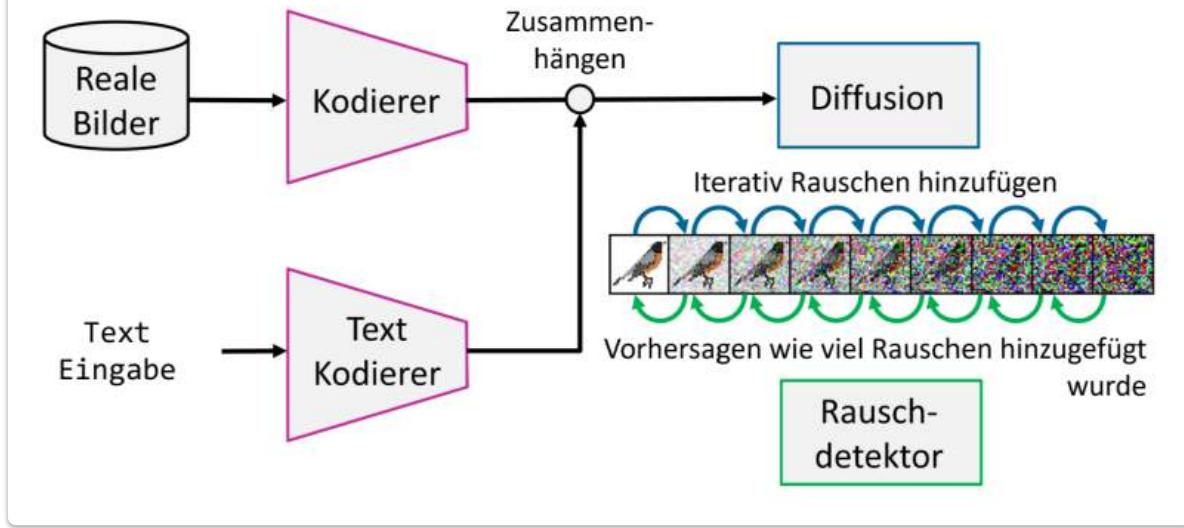
- **Reale Bilder** und **Zufälliger, latenter Vektor** sind die Eingaben.

- Der **Generator** erzeugt Bilder aus dem latenten Vektor.
 - Der **Diskriminator** erhält sowohl die realen Bilder als auch die vom Generator erzeugten Bilder.
 - Basierend auf der Ausgabe des Diskriminators werden der **Diskriminator Verlust** und der **Generator Verlust** berechnet.
-

Training eines Diffusionsmodells

EVC_Skriptum_CG, p.60

- Diffusionsmodelle werden trainiert, indem Bilder und Textaufforderungen (Prompts) in einem **gemeinsamen latenten Raum** kodiert werden.
- **Rauschhervorhebung:**
 - In den Bildern wird *iterativ Rauschen hinzugefügt*.
 - Typischerweise wird *gaußsches Rauschen* verwendet, aufgrund seiner probabilistischen Eigenschaften.
- **Lernen aus "verrauschten" Daten:**
 - Das hinzugefügte Rauschen ermöglicht es dem Modell, aus diesen verrauschten Daten zu lernen.
 - Dies geschieht mithilfe eines **Rauschdetektors**.
 - Der Rauschdetektor *sagt den Grad des Rauschens bei jedem Schritt vorher*.
- **Aufgabe des Rauschentferrners (Denoising):**
 - Das Rauschen soll anhand des *verrauschten Bildes* und der *zugehörigen Texteingabe* vorhergesagt werden.
 - Anschließend wird das vorhergesagte Rauschen *subtrahiert*.
- **Lernprozess und Ergebnis:**
 - Dieser Lernprozess stellt sicher, dass sich das Bild mit jeder Iteration einer *weniger verrauschten und genaueren Darstellung* annähert.



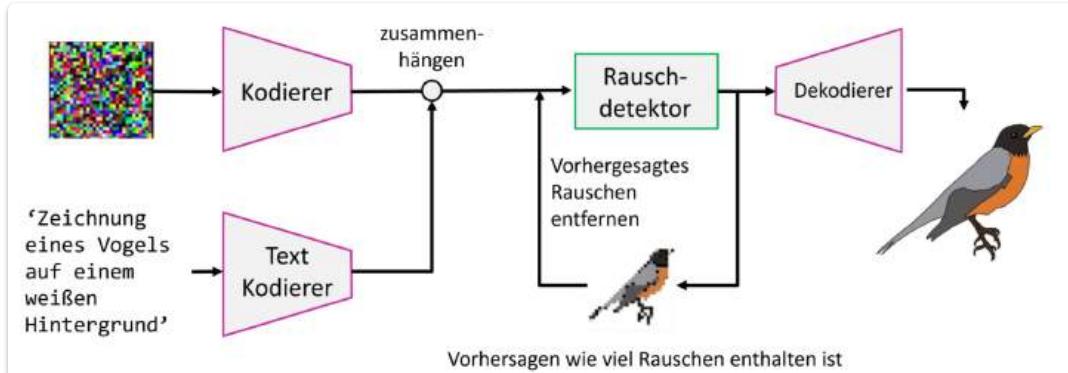
- **Reale Bilder** werden von einem **Kodierer** verarbeitet.
- **Text Eingabe** wird von einem **Text Kodierer** verarbeitet.
- Die Ausgaben beider Kodierer werden zusammengeführt ("Zusammenhängen").
- Dies fließt in die **Diffusion**, wo *iterativ Rauschen hinzugefügt* wird.
- Ein Beispielbild zeigt eine Reihe von immer stärker verrauschten Bildern.
- Der **Rauschdetektor** erhält die verrauschten Bilder und sagt vorher, wie viel Rauschen hinzugefügt wurde.

Inferenz in einem Diffusionsmodell

EVC_Skriptum_CG, p.61

- **Generierungsprozess (Inferenzphase):**
 - Beginnt mit einer **Textaufforderung** und einem Ausschnitt bestehend aus **zufälligem, gaußschem Rauschen**.
- **Kodierung und Verfeinerung:**
 - Beide (Textaufforderung und Rauschen) werden in den **latenten Raum** kodiert.
 - Sie werden durch eine Reihe von **Rauscherkennungs- und Subtraktionsschritten** kontinuierlich verfeinert.
- **Rauschdetektor:**
 - Bei jeder Iteration schätzt der **Rauschdetektor** die **Menge des vorhandenen Rauschens**.
 - Dieses Rauschen wird dann **subtrahiert**, um das Bild allmählich zu klären (denoising).
- **Entrauschungsprozess:**
 - Diese Abfolge wird so lange fortgesetzt, bis das Bild **ausreichend entrauscht** ist.
- **Dekodierung zum visuellen Bild:**

- Die resultierende latente Darstellung wird von einem **Dekodierer** wieder in ein *visuelles Bild dekodiert*.
- Der Dekodierer ist in der Regel als "Variational Auto-Encoder" (VAE) strukturiert.
- **Ergebnis und Fähigkeit:**
 - Die resultierenden 2D-Bilder, die aus verallgemeinerten Rausch- und Texteingaben generiert wurden, demonstrieren die Fähigkeit des Modells, *detaillierte und kontextgerechte Bilder aus einem hochdimensionalen latenten Raum zu erzeugen*.
- **Anwendung:**
 - Nun kann ein Werkzeug zur Generierung von *2D-Bildern anhand von Texteingaben* genutzt werden.
 - Für die Erstellung von *3D-Modellen und -Szenen* wird jedoch mehr benötigt.



- Ein Rauschbild (z.B. 'Zeichnung eines Vogels auf einem weißen Hintergrund') wird von einem **Kodierer** verarbeitet.
- Eine **Texteingabe** wird von einem **Text Kodierer** verarbeitet.
- Beide Ausgaben werden zusammengeführt ("zusammenhängen"). * Dies fließt in den **Rauschdetektor**, der das "Vorhergesagtes Rauschen entfernen" signalisiert und auch "Vorhersagen wie viel Rauschen enthalten ist".
- Anschließend geht es in den **Dekodierer**, der das endgültige, entrauschte Bild (z.B. einen Vogel) ausgibt.

Szenendarstellungen für Machine 3D-Learning

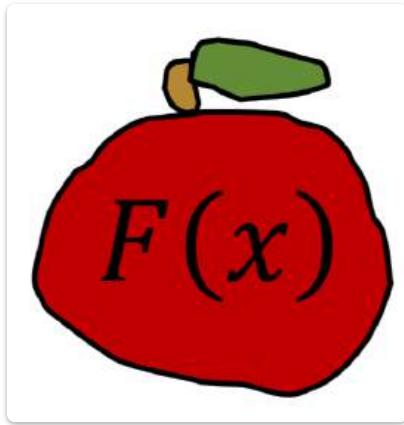
[EVC_Skriptum_CG, p.61](#)

- Ein reales Objekt kann nicht *detaillgenau* dargestellt werden, um jedes einzelne Detail zu erfassen.
- Es gibt verschiedene Darstellungen, die jeweils *Vor- und Nachteile* haben.
- Sie können in zwei Kategorien unterteilt werden: **implizit** und **explizit**.

Implizite Darstellungen

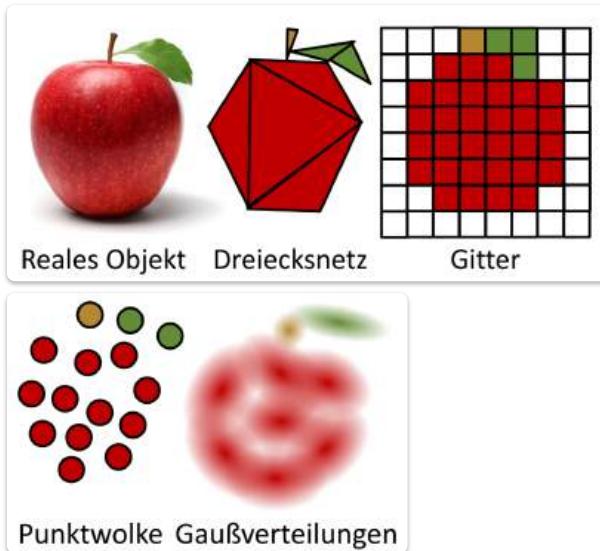
- Sind **Funktionen**, die für jeden Punkt im Raum einen Wert definieren.
- Beispiele:
 - Ob das Objekt vorhanden ist oder nicht.

- Wie weit die nächstgelegene Oberfläche entfernt ist (z.B. **Signed Distance Functions (SDFs)**).



Explizite Darstellungen

- Sind in der Regel **Mengen von diskreten Elementen**.
- Beispiele:
 - **Dreiecksnetz** (Mesh)
 - **Gitter** (Grid)
 - **Punktwolke** (Point Cloud)



Herausforderungen und Überlegungen

- **Beispiel Punktwolken:**
 - Beim Scannen eines realen Objekts erhält man einen *endlichen Satz von Punkten*.
 - Es gibt *keinen einfachen Weg*, um sicherzustellen, dass beim Rendern der Punkte *keine Löcher* bleiben, die verdecken, was sich hinter dem Objekt befindet.
 - Man könnte versuchen, die Löcher zu "schließen" und ein Dreiecksnetz zu konstruieren.
- **Training mit Dreiecksnetzen:**
 - Wichtig ist sicherzustellen, dass das Netz *nicht deformiert und ungültig* wird.
 - Dies kann passieren, wenn sich *Vertices (Eckpunkte)* frei bewegen können und sich dadurch *Flächen überschneiden*.

- **Fazit:** Es ist immer wichtig zu überlegen, welche Darstellung für die jeweilige Aufgabenstellung am *besten geeignet* ist.

Neural Radiance Fields (NeRFs)

[EVC_Skriptum_CG](#), p.61, p.62

- NeRFs bieten eine *implizite Darstellung räumlicher Daten*.
- Sie nutzen die Gewichte innerhalb eines neuronalen Netzes zur Modellierung von 3D-Umgebungen.
- Dieser Prozess wird dem **Deep Learning** zugeordnet und erfordert *erhebliche Rechenressourcen*.

Kernmechanismus eines NeRF

- Umfasst die **Abtastung entlang eines Strahls im 3D-Raum** (Ray Casting/Sampling).
- Diese Methode wird von einem **mehrschichtigen Perzepron** (engl. *multi-layer perceptron, MLP*) gesteuert.
- **Prozess:**
 1. Der Strahl durchquert die Szene.
 2. Das Modell bewertet die *Farbe und Dichte* an verschiedenen Punkten entlang des Pfades.
 3. Diese Werte werden *projiziert und gemischt*, um das endgültige Bild zu erzeugen.
 4. Durch diese Projektion werden die **volumetrischen Daten** (3D-Informationen im Raum) zu einem *einzelnen Farbwert pro abgetastetem Strahl* verdichtet.
 5. Dies resultiert in einer *sehr detaillierten Wiedergabe der Szene*.

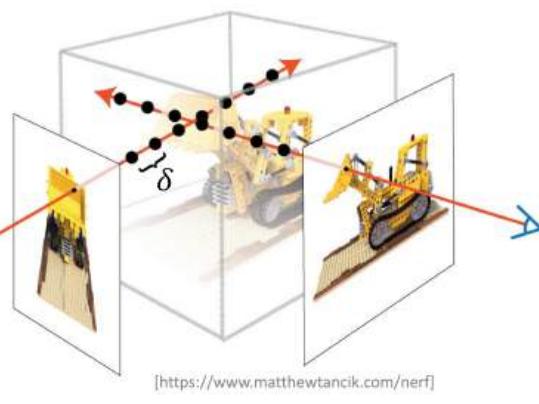
■ Each point in space contains:

- (View dependent) color – c
- Density (transparency) – σ

■ Use *raymarching* to render it

$$C(r) = \sum_{i=1}^N T_i (1 - \exp(-\sigma_i \delta_i)) c_i$$

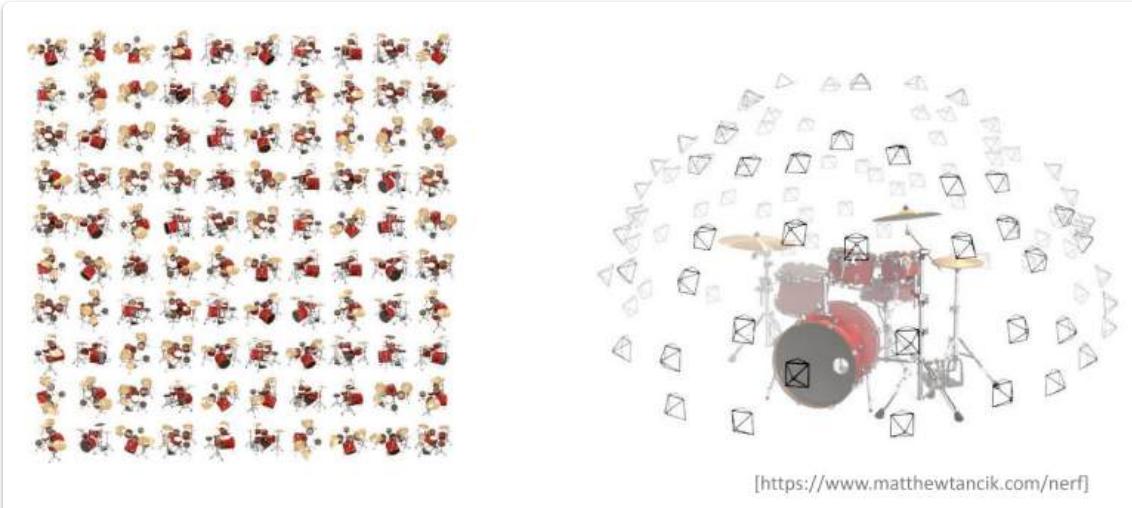
$$T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_j \delta_j\right)$$



Rekonstruktion einer realen Szene mit NeRFs

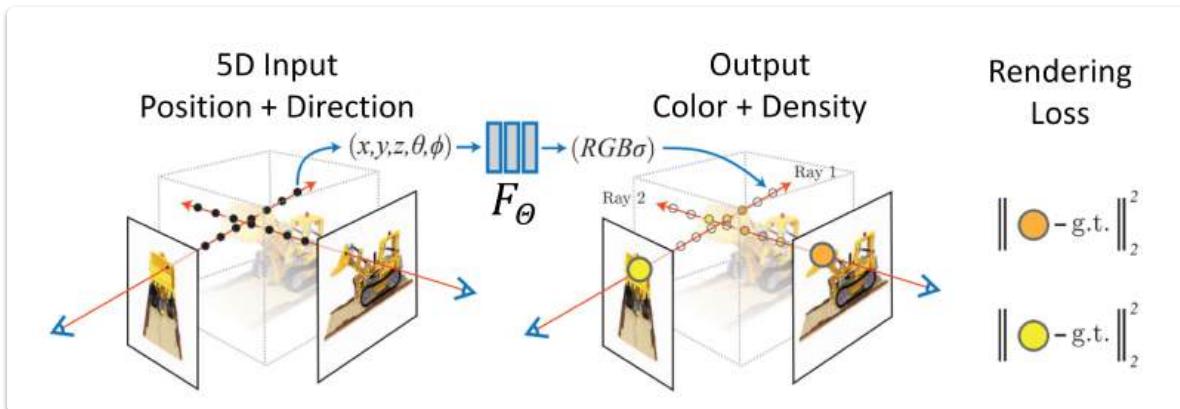
- **Basis:** Es werden *echte Fotografien* als Basis benötigt.
- **Kamerarekonstruktion:** Der *Structure-from-Motion-Algorithmus* (SfM) kann verwendet werden, um die entsprechenden Kamerapositionen zu erhalten, aus denen dann die Szene gerendert wird.

- **Optimierung:** Das gerenderte Bild wird mit dem entsprechenden Basisbild verglichen und das Netzwerk so optimiert, dass es mit diesen Bildern übereinstimmt.

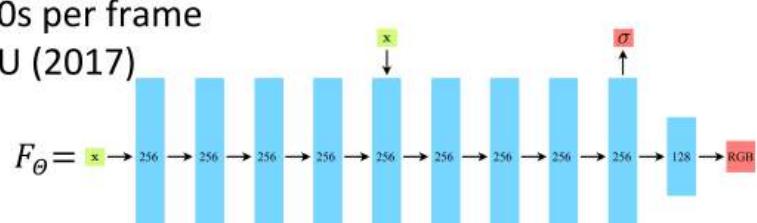


Herausforderungen und Flexibilität

- NeRFs basieren auf neuronalen Netzen, die **Blackboxen** sind.
- Das bedeutet, es ist sehr schwierig, einen Teil einer Szene **direkt zu ändern** (z.B. ein Objekt zu entfernen).
- Es kann notwendig sein, die Szene vor der Bearbeitung zunächst in eine **explizite Darstellung** (z.B. ein Dreiecksnetz) umzuwandeln.
- Dies kann unter Verwendung des **Marching-Cubes-Algorithmus** geschehen.
- **Fazit:** NeRFs haben sich als **sehr leistungsfähig** erwiesen und werden heutzutage **häufig verwendet**.

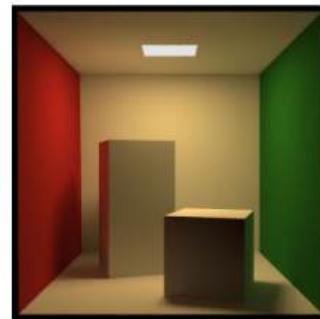


- Approx. 590 000 weights $\approx 3 \text{ MB}$
- 1-2 days to train and 30s per frame on an NVIDIA V100 GPU (2017)



- Remember that an explicit 1024^3 voxel grid requires **512 GB**

- The weights of a network represent its memory
- Given a network F_θ , the weights of F_θ may be underutilized or overutilized – not a lot of control
- Do we need a large NN to represent a simple scene?



- Is our NN large enough to capture all details?

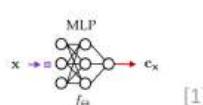


Gaussian Splatting (GS)

[EVC_Skriptum_CG, p.62](#)

Neural Radiance Fields

- Implicit



- Deep Learning



- Ray Marching



Gaussian Splatting

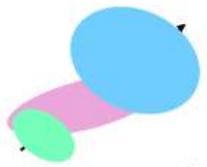
- Explicit



- Simple ML

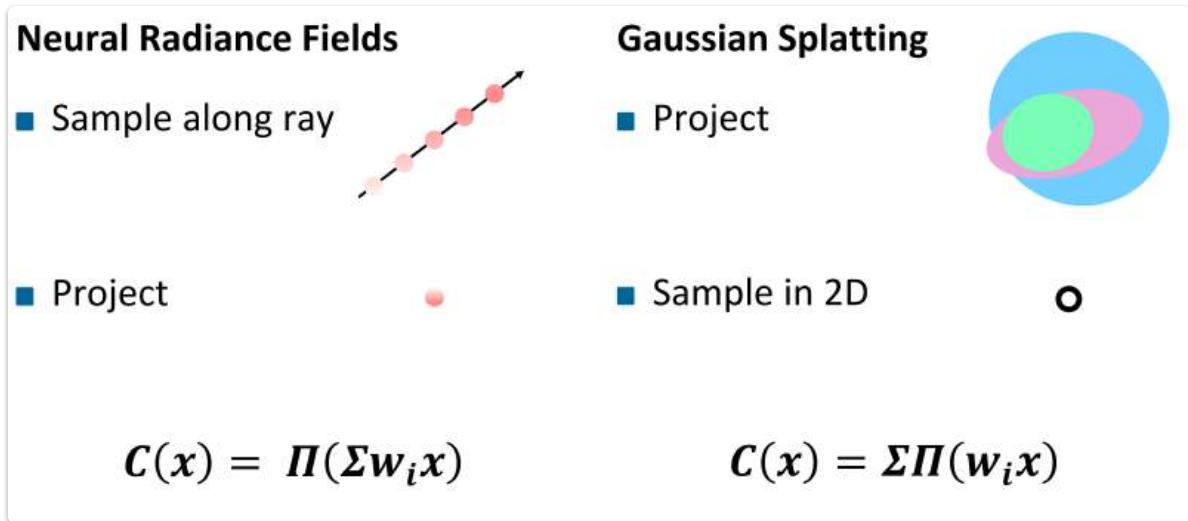


- Blending Gaussians



- Im Gegensatz zu NeRFs ist Gaussian Splatting (GS) eine **explizite Technik** zur Szenendarstellung.
- Es nutzt die Prinzipien des **Volumen Renderings**.

- Anstatt neuronale Netze als Blackboxen zu verwenden, betrachtet GS **Gaußsche Funktionen als primitive Formen**.



Definition einer Gaußglocke

- Jede Gaußglocke wird durch ihren **Mittelwert (Position)**, ihre **Kovarianzmatrix (Form)**, ihre **Deckkraft** und **Farbe** definiert.
- Dies ermöglicht eine *einfachere Analyse der maschinellen Lernverfahren* (z.B. stochastische Gradientenverfahren), um die *räumlichen Merkmale der Szene zu lernen*.

Normal distribution in 3D:

$$\mathbf{X} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma}), \quad \mathbf{X} = X_1, X_2, X_3$$

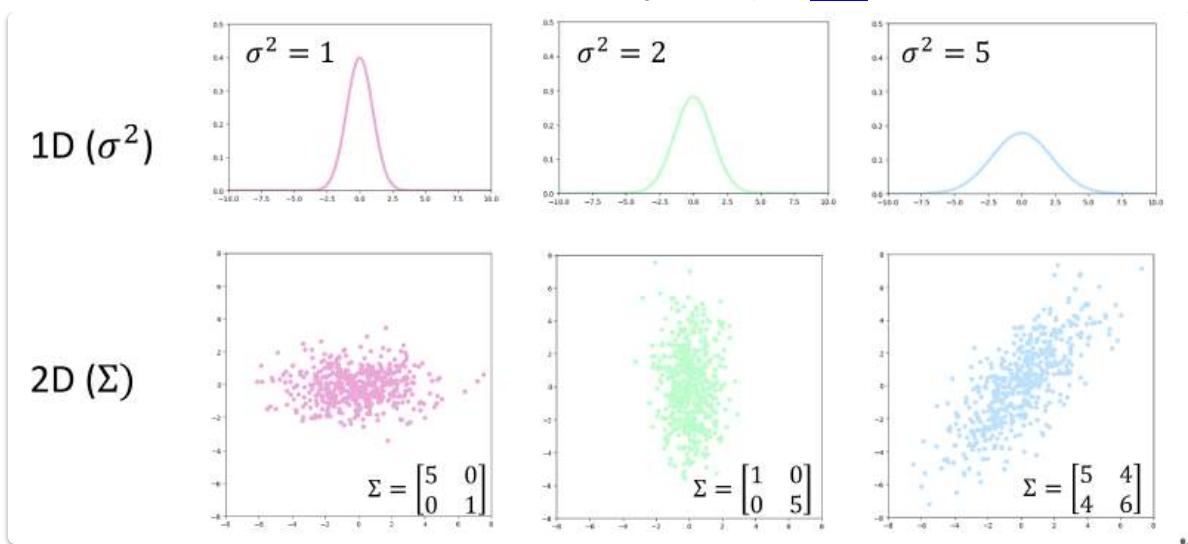
- Position \mathbf{X} , where \mathbf{X} is the vector $\boldsymbol{\mu} - \mathbf{x}$ (\mathbf{x} ... query point)
- Covariance matrix $\boldsymbol{\Sigma}$ (stretching/scaling)

$$G(\mathbf{X}) = e^{-\frac{1}{2}(\mathbf{X})^T \boldsymbol{\Sigma}^{-1} (\mathbf{X})}$$

- Additionally: Color (RGB → Spherical Harmonics) + transparency (α)

Rendering-Prozess bei GS

- Umfasst die **Projektion der 3D-Gaußglocke auf eine 2D-Ebene**.
- Anschließend erfolgt die **Rasterung** der projizierten Daten, um ein Bild zu generieren.
- Der Beitrag jeder Gaußglocke zum endgültigen Bild wird im 2D-Raum *sortiert und abgetastet*.
- Dies ermöglicht ein *effizientes Rendering* ohne den hohen Auswertungsaufwand von Deep-Learning-Architekturen.



Training von Gaussian Splatting

- **Initialisierung:** Das Training beginnt mit einer *anfänglichen Menge an Gaußglocken*.
- **Datengewinnung:** Diese Gaußglocken können mit dem **Structure-from-Motion-Algorithmus** (SfM) gewonnen werden.
 - SfM gibt einen Satz von *3D-Punkten* aus, die mit den realen Basisbildern übereinstimmen.
- **Anpassung der Gaußglocken:**
 - Anschließend werden diese Punkte direkt in *Gauß'sche Punkte umgewandelt*.
 - Ihre *Kovarianzmatrizen* werden so eingestellt, dass sie und ihre Nachbarn *Flächen ohne Löcher* bilden.
 - Idealerweise sollten sie eine *erhebliche Überlappung* bilden.
- **Detaillierung und Gradienten:**
 - Falls der anfängliche Satz an Gaußglocken nicht ausreicht, um alle Details in der Szene darzustellen, müssen diese entweder *geklont* oder *aufgeteilt* werden.
 - Dies kann über die *akkumulierte Größe ihrer Positionsgradienten* bestimmt werden.
 - Ein großer Gradient bedeutet, dass sich die Gaußglocke *schnell irgendwohin bewegen muss* oder dass sie an *mehreren verschiedenen Orten gleichzeitig befindlich* sein muss.
 - In diesen Fällen wird die Gaußglocke geklont oder aufgeteilt.
- **Ergebnis:** Auf diese Weise wird eine *qualitativ hochwertige 3D-Darstellung* mit mehr Gaußglocken und somit mehr Details erzielt, wo sie benötigt werden.

1 million Gaussians:



1 million Gaussians:
(as fully opaque ellipsoids)

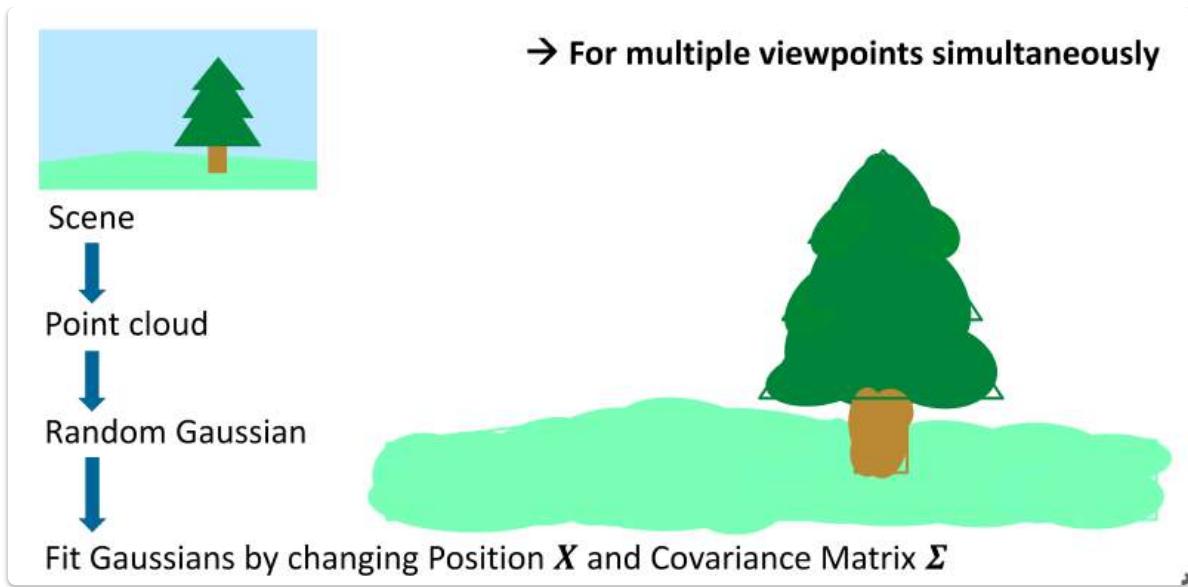


Generative Modelle für 3D Objects

[EVC_Skriptum_CG](#), p.62

- Bisher wurde die Erzeugung von 2D-Bildern und die Rekonstruktion realer Szenen behandelt.
- **Kernfrage:** Wie können diese beiden Verfahren kombiniert werden?
- **Ansatz:**
 - Der Prozess des *iterativen Änderns/Entrauschens von Bildern* (wie bei Diffusionsmodellen) kann genutzt werden, um neue Bilder zu erzeugen.
 - Gleichzeitig soll eine **3D-Objektdarstellung iterativ geändert** werden.
- **Ziel:** Es wird beschrieben, wie ein **Diffusionsmodell** und **3D Gaussian Splatting** kombiniert werden, um *neue 3D-Objekte zu synthetisieren*.

Ablauf der 3D-Objekt-Erstellung



1. Texteingabe als Startpunkt:

- Der Prozess beginnt mit einer **Texteingabe** (dem "Prompt"), die beschreibt, welches 3D-Objekt erstellt werden soll.

2. Initialisierung der Gaußschen Punkte:

- Ein erster Satz von **Gaußschen Punkten** wird erzeugt. Diese dienen als grundlegende Darstellung des 3D-Objekts.
- Die Erstellung kann **manuell** erfolgen.
- Alternativ kann ein **vortrainiertes Netzwerk** verwendet werden, das basierend auf der Texteingabe eine (anfänglich oft grobe und nicht optimierte) Punktwolke generiert.
- Diese Punkte werden anschließend mit einem Verfahren, das dem der Szenenrekonstruktion ähnelt, in **Gaußglocken** umgewandelt.

3. Rendering und Rauschhinzufügung:

- Die Gaußglocken werden aus **zufälligen Blickwinkeln** gerendert. Dieser Rendering-Prozess wird auch als "Splatting Renderer" bezeichnet.
- Die Ergebnisse sind **2D-Bilder** (Renderings).
- Da **Diffusionsmodelle** mit verrauschten Bildern arbeiten, wird diesen 2D-Renderings **künstlich Rauschen hinzugefügt**.

4. Anpassung durch Diffusionsmodell:

- Ein **Diffusionsmodell** analysiert die verrauschten 2D-Renderings.
- Es lernt, wie diese Bilder verändert werden müssen, um dem ursprünglichen Text-Prompt zu entsprechen (z.B. indem es Rauschen entfernt und Details hinzufügt, die zum Prompt passen).
- Basierend auf den vom Diffusionsmodell vorgeschlagenen Änderungen der 2D-Bilder wird die **zugrunde liegende 3D-Objektdarstellung** (die Gaußglocken selbst) entsprechend angepasst.

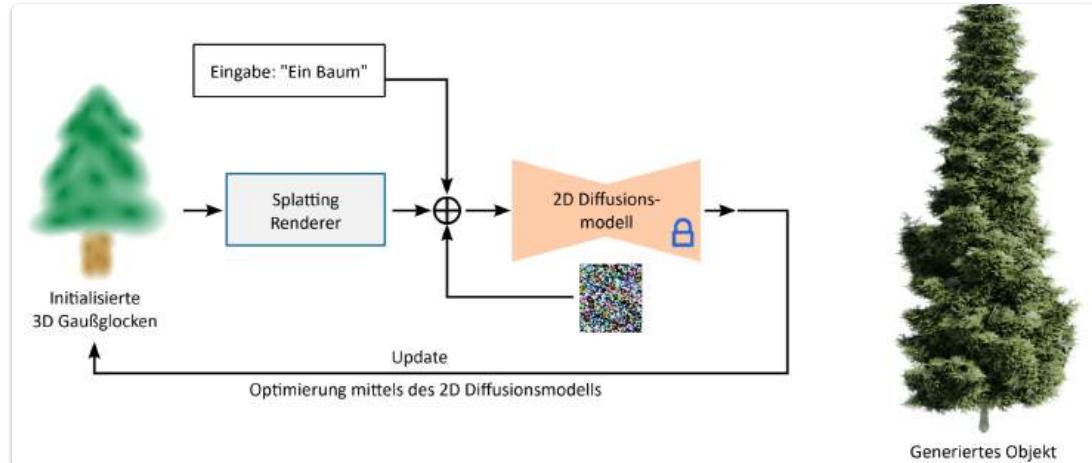
5. Verfeinerung und Detailverbesserung:

- Um die Detailtiefe des 3D-Objekts zu erhöhen, werden die Gaußglocken **aufgeteilt und geklont**.

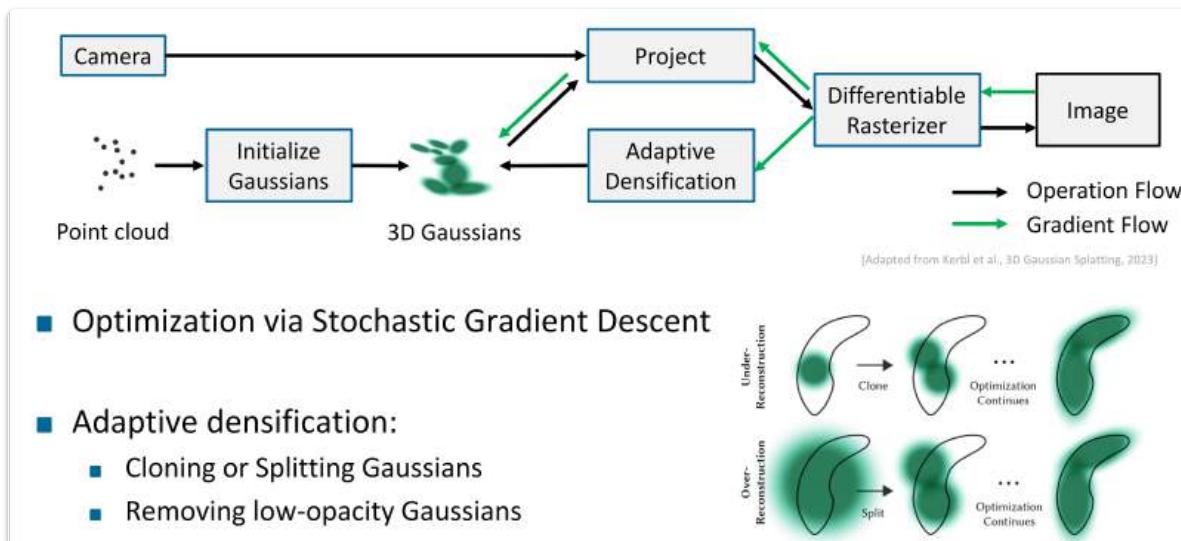
- Dieser Schritt erfolgt unter Berücksichtigung der **Größe ihres Gradienten** (ein Maß für die lokale Veränderung der Eigenschaften der Gaußglocke, was auf Bereiche hinweist, die mehr Details benötigen).

6. Automatisierte Erstellung:

- Nach **mehreren Hundert bis Tausend Iterationen** dieses Zyklus aus Rendering, Diffusionsmodell-Anpassung und Verfeinerung wird das neue 3D-Objekt **automatisch erstellt**.



Training im Detail:



- Optimization via Stochastic Gradient Descent
- Adaptive densification:
 - Cloning or Splitting Gaussians
 - Removing low-opacity Gaussians