

2. EVC Test - Computergrafik

Vorwort

Diese Stoffsammlung/Zusammenfassung enthält den Stoff des 2. Teils, 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 auf dem 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.

Das ist nicht der vollständige Stoff des 2. Tests

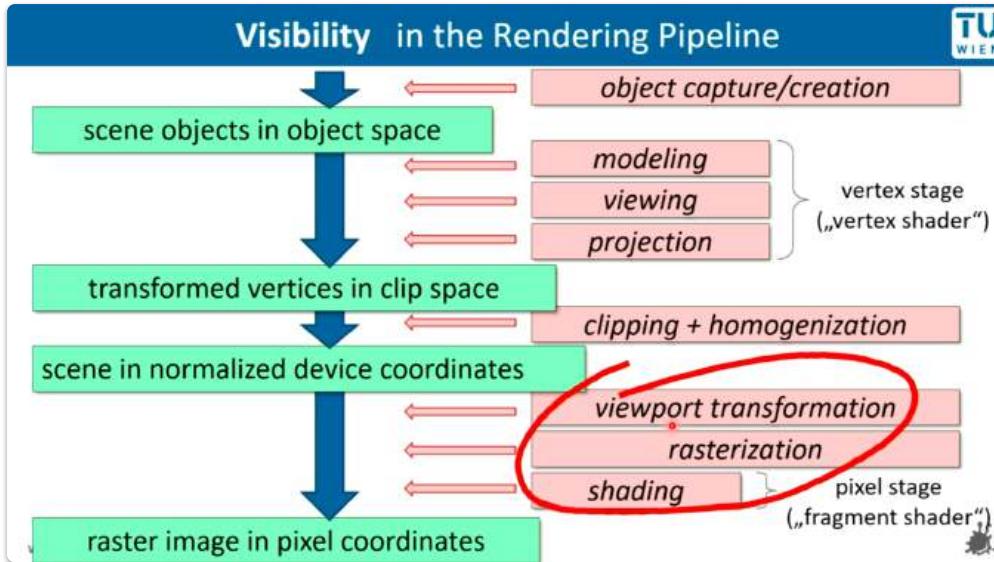
Da genau so wie in den letzten Jahren, zum 2. Test nicht nur Stoff von der 2. Hälfte des Semesters sondern auch Themen vom 1. Teststoff benötigt werden, sollte man sowohl die Zusammenfassung vom 1. Test, als auch den vom 2. Test lernen. In dieser Datei wird nur der Stoff vom 2. Teil von EVC behandelt!

Inhalt

- 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

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.

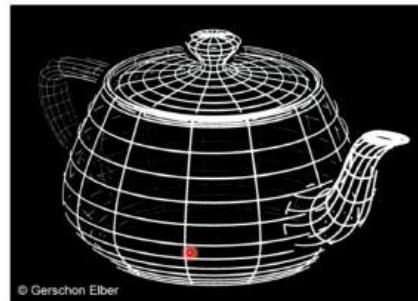
3. Berücksichtigung von Transparenz

- 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

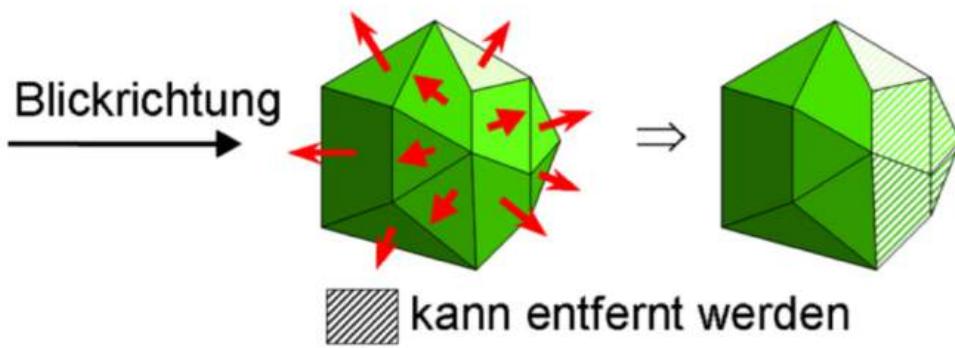
- **Backface Culling** ist kein vollständiges Sichtbarkeitsverfahren, sondern dient als Optimierungsschritt.
 - **Durchschnittliche Reduktion:** Etwa 50% der Polygone werden entfernt.

3. Berechnungsverfahren

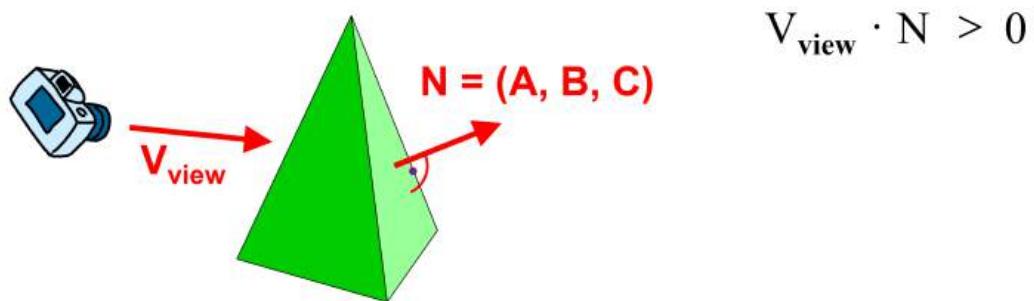
- **Orthographische Projektion:**
 - **Berechnung:** Das Skalarprodukt des Blickrichtungsvektors V_{view} und der Oberflächennormale N wird berechnet.
 - **Formel:** $V_{view} \cdot N > 0 \Rightarrow$ Polygon ist unsichtbar.
- **Perspektivische Projektion:**
 - **Berechnung:** Der Blickpunkt (x, y, z)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

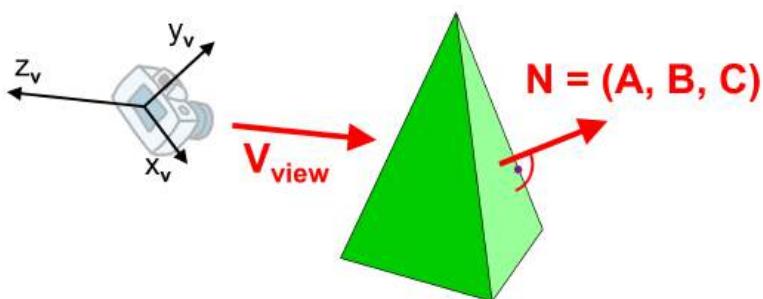


object description in viewing coordinates $\Rightarrow V_{\text{view}} = (0, 0, V_z)$

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

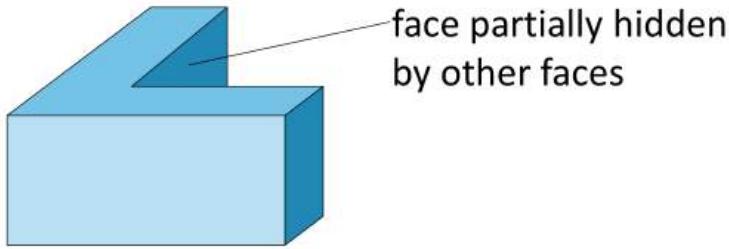
sufficient condition: if $C \leq 0$ then back-face

↑
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: pink;">.8 .7</td><td></td></tr> <tr><td></td><td></td><td style="background-color: pink;">.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!

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

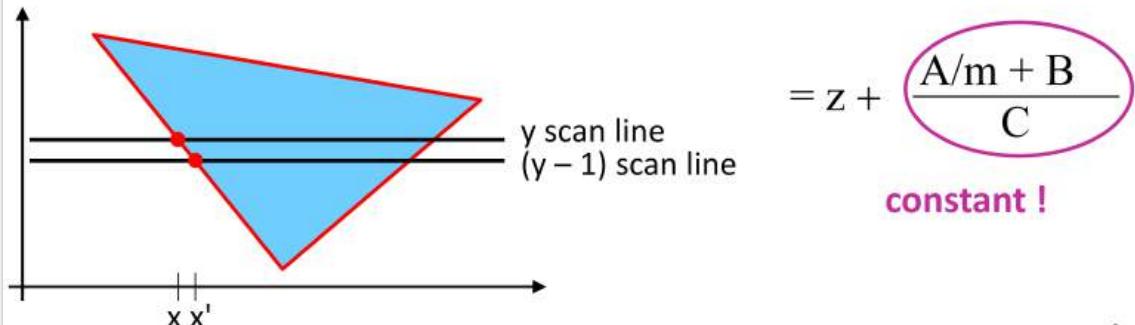
$$z'' = \frac{-Ax - B(y-1) - D}{C} = z + \frac{B}{C}$$

Vorteile

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

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

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



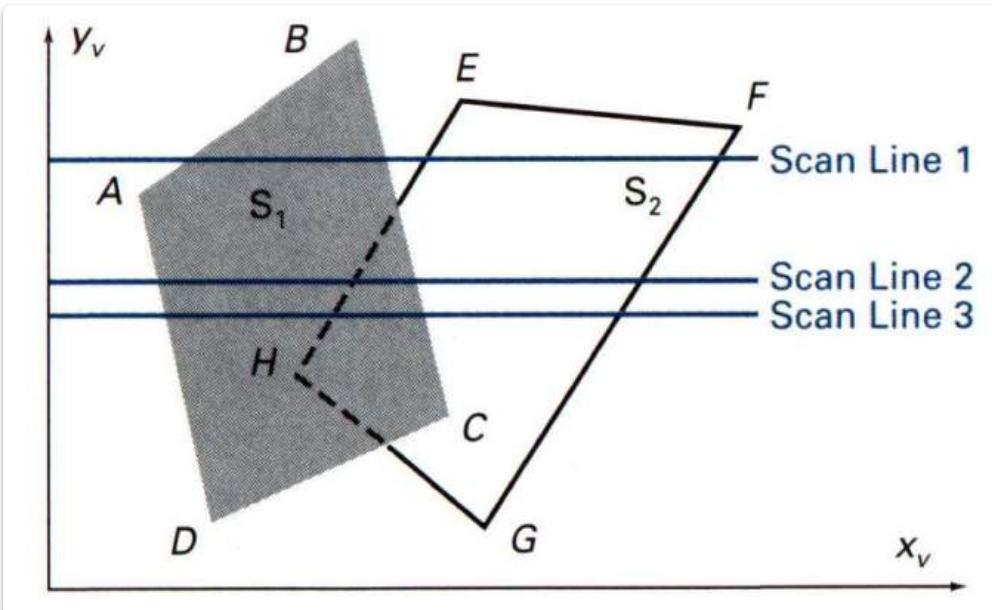
Anmerkung

- **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)

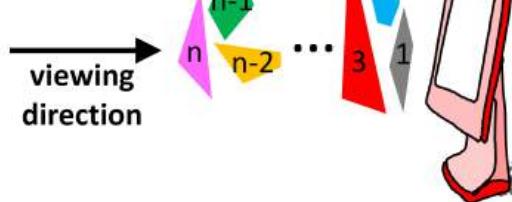
- (1) "approximate"-sorting using smallest z -value (greatest depth)
- (2) fine-tuning to get correct depth order

surfaces scan converted in order

sorting both in image and object space

scan conversion in image space

also called "painter's algorithm"



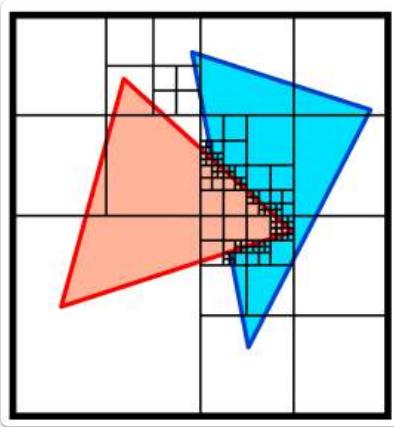
Werner Purgathofer

23

Area-Subdivision Methode

[EVC_Skriptum_CG](#), p.33

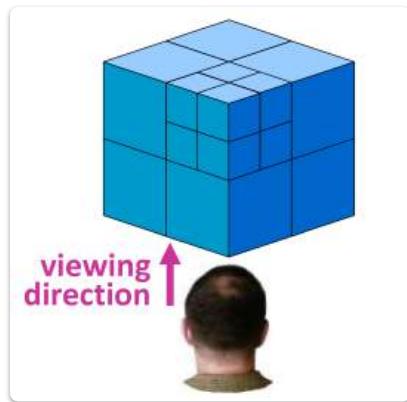
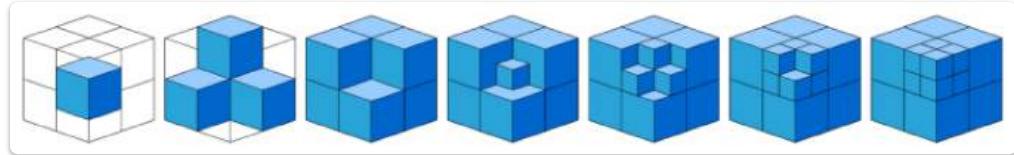
- **Grundidee:** Divide-and-Conquer-Ansatz für das Sichtbarkeitsproblem.
- **Analogie:** Ähnlich der Quadtree-Repräsentation von Bildern.
- **Vorgehensweise:**
 1. **Einfache Fälle:** Sichtbarkeitsproblem wird in grober Auflösung gelöst.
 2. **Komplizierte Fälle:**
 - Unterteilung der aktuellen Bildfläche 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

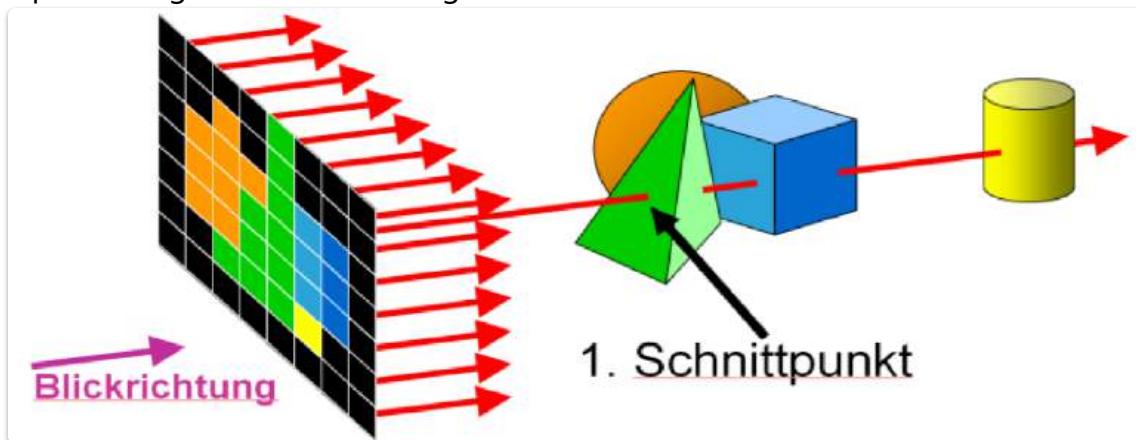


- **Von vorne nach hinten:**
 - Rekursives Durchlaufen der Octree-Knoten.
 - Zeichnen nur der sichtbaren Bereiche.
 - Notwendigkeit, sich bereits gezeichnete Bereiche zu merken, um Overdraw zu vermeiden.
- **Genereller Vorteil:** Das inhärente Wissen über die Tiefenordnung im Vergleich zu anderen Datenstrukturen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



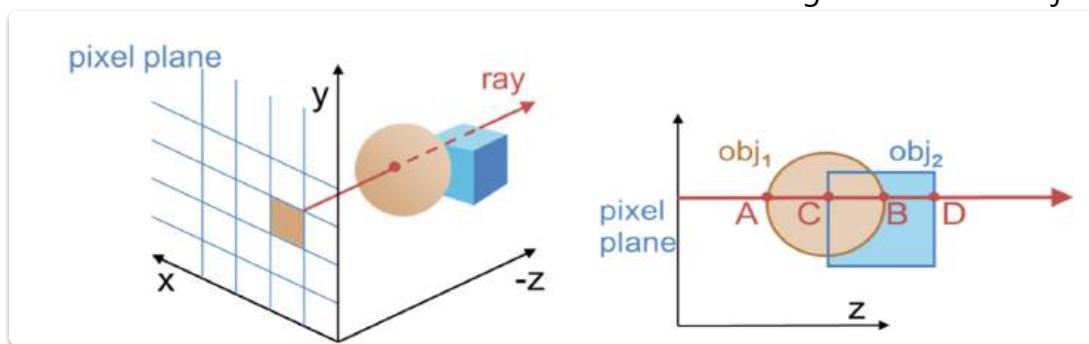
Ray-Casting von CSG-Objekten

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.

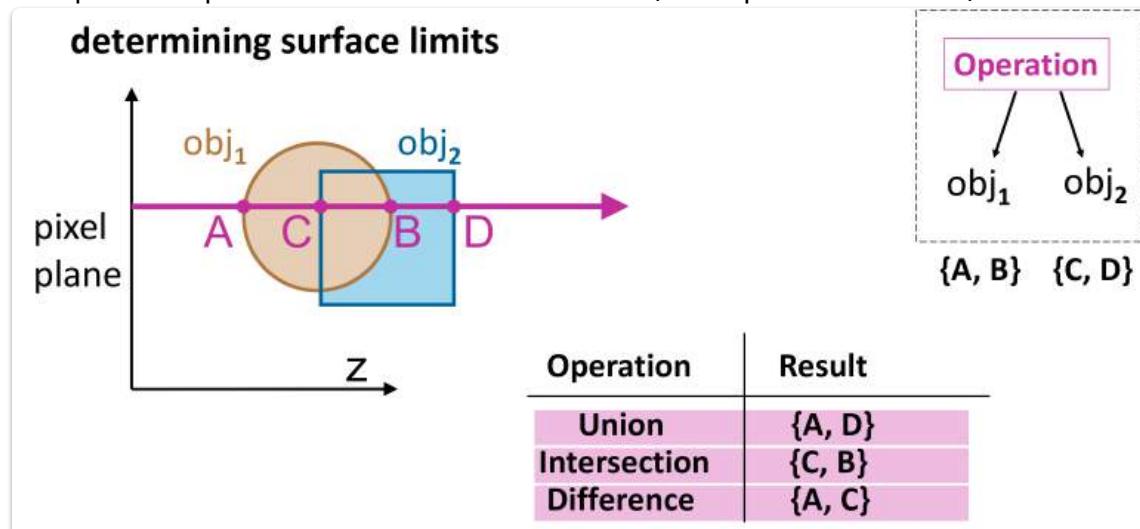
4. **Pixelfarbe:** Das Pixel erhält die Farbe des am vordersten geschnittenen Objekts.



- **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).



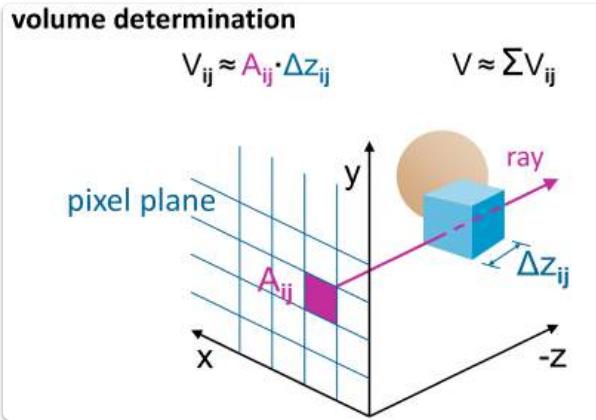
Ray-Casting ist eine vereinfachte Version von Ray-Tracing, mit dem noch viele weitere optische Effekte simuliert werden können. Dazu kommen wir in einem späteren Kapitel.

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

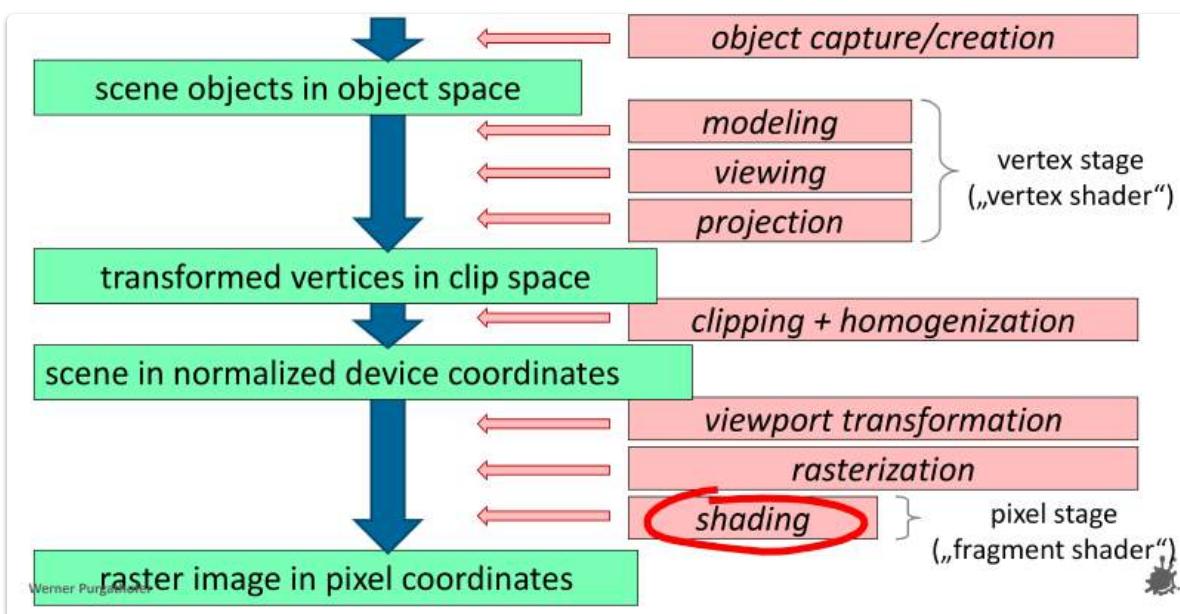
Bildraum-Verfahren:

- 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).



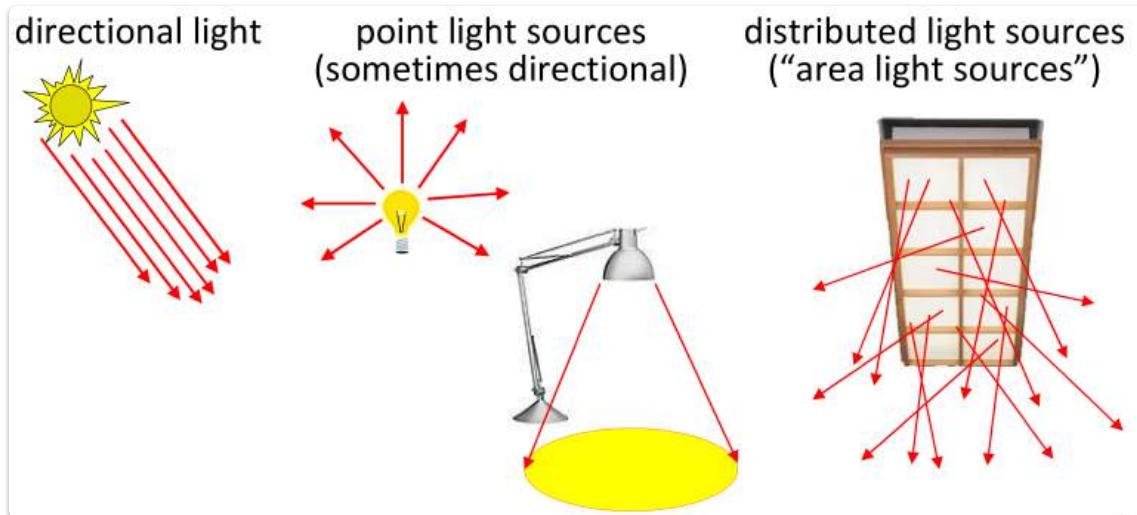
Lichtquellen und Oberflächen

Lichtquellen

EVC_Skriptum_CG, p.35

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

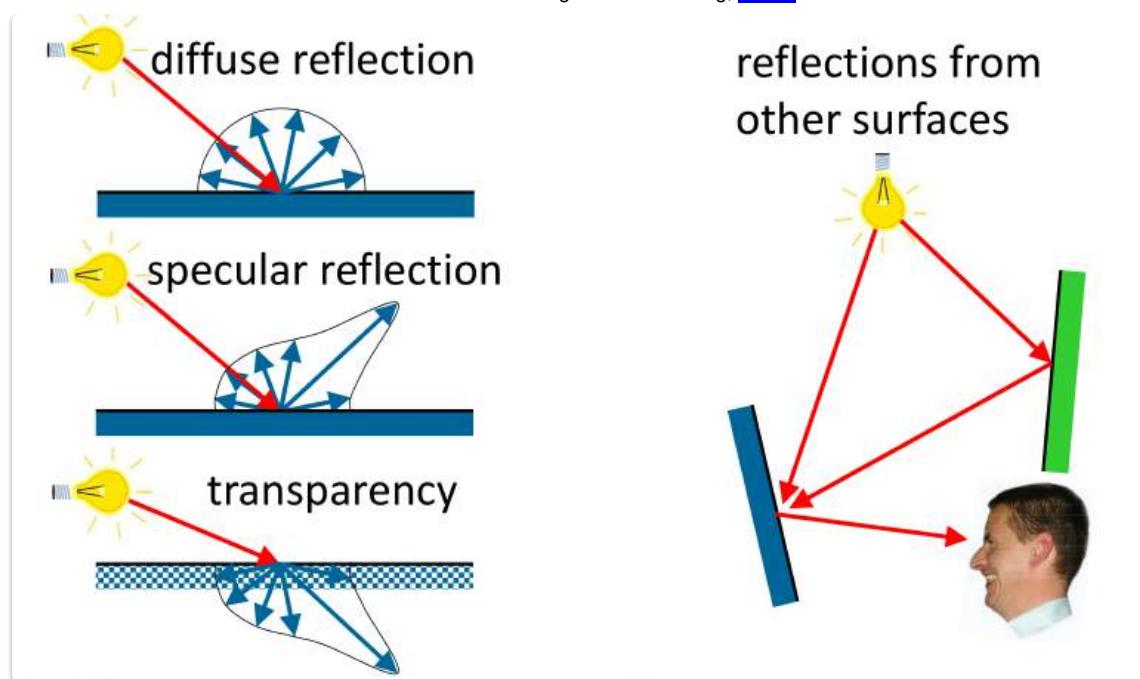
- Lichtrichtung (z.B. paralleles Sonnenlicht)
- Punktlichtquellen (strahlen Licht von einem einzelnen Punkt in alle Richtungen)
- Gerichtete Punktlichtquellen (kombinieren Punktlichtquelle mit einer kegelförmigen oder rechteckigen Lichtverteilung)
- Flächige Lichtquellen (emittieren Licht von einer ausgedehnten 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)

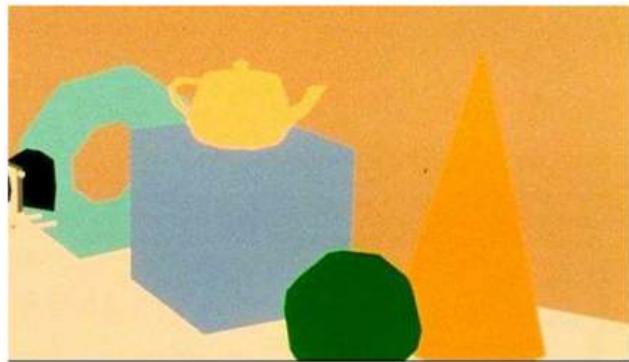
EVC_Skriptum_CG, p.35

- **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](#)

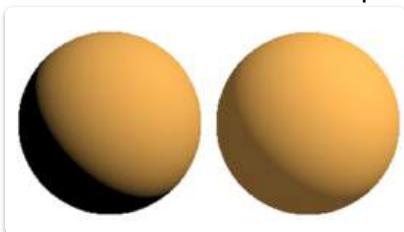
- **Kernaussage:** Die Helligkeit einer diffus reflektierenden Oberfläche ist proportional zum Kosinus des Winkels zwischen der Oberflächennormale und der Richtung zur Lichtquelle. Flacher Lichteinfall führt zu dunkleren Oberflächen.
- **Bedeutung:** Erzeugt den Eindruck räumlicher Form durch Helligkeitsvariationen.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

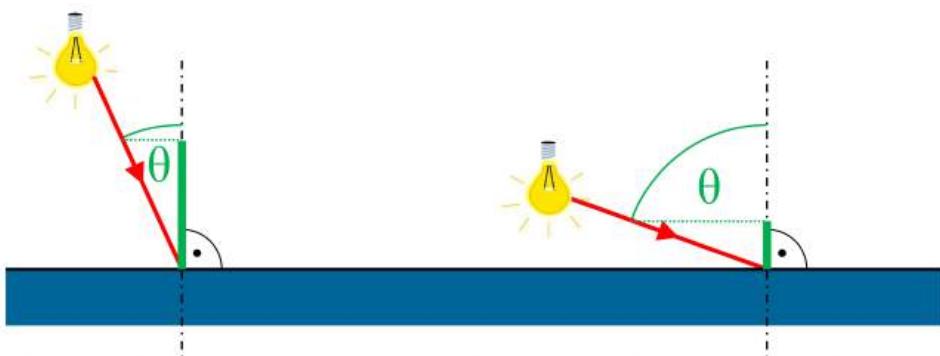
$$L = k_d \cdot I \cdot \cos \theta$$

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

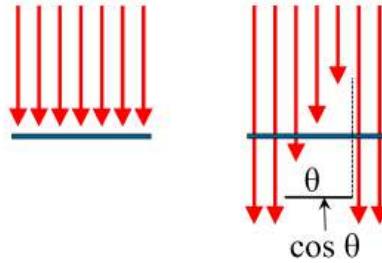
L ... pixel color

k_d ... diffuse coefficient

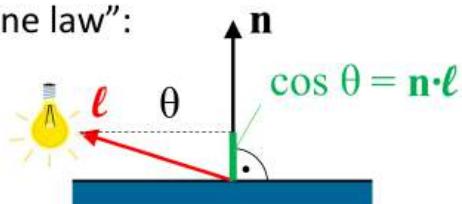
$$L = k_d \cdot I \cdot \cos \theta$$

for ideal diffuse reflectors (Lambertian reflectors)

brightness depends on
orientation of the surface:



"Lambert's cosine law":

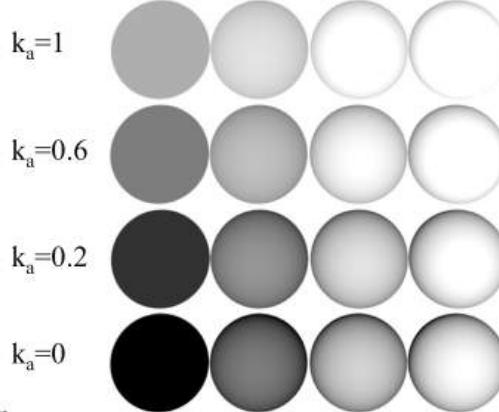


$$L_{\text{diff}} = k_d \cdot I \cdot (\mathbf{n} \cdot \mathbf{l})$$

total diffuse reflection:

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

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

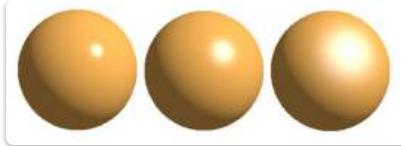


(sometimes extra k_a
for ambient light)

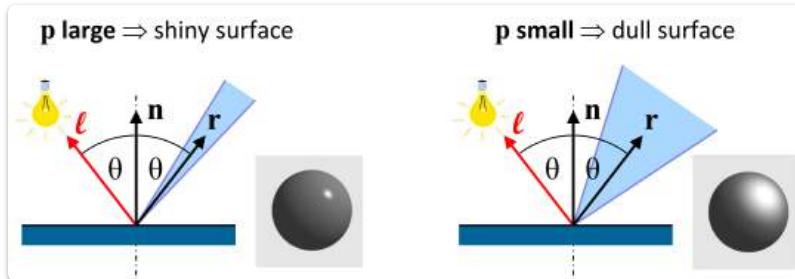
Werner Purgathofer

Glanzpunkte (Specular Highlights)

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



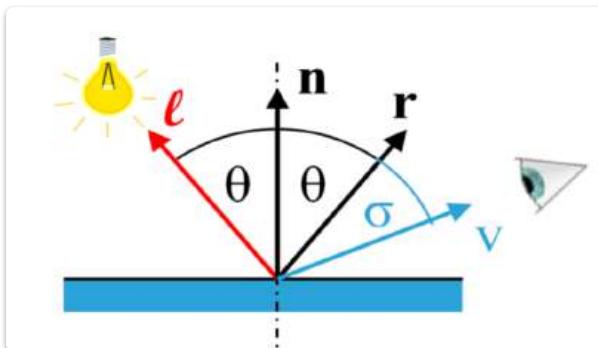
- Da die exakte spiegelnde Reflexion äußerst kompliziert zu berechnen ist, behilft man sich mit einer einfachen Funktion, die einen ähnlichen Verlauf hat wie das Highlight: $\cos^p(\alpha)$.
- Mit dem freien Parameter p lässt sich dabei die „Poliertheit“ der Oberfläche steuern:
 - Je größer p ist, desto 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).



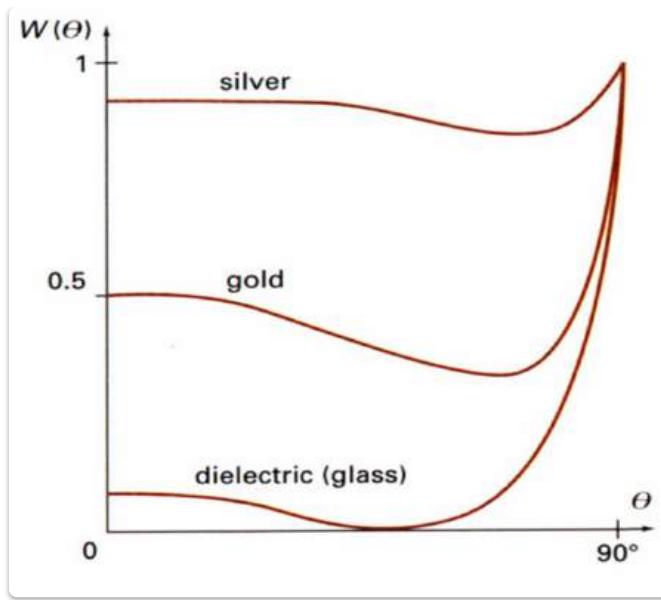
- Um diesen Effekt im richtigen Ausmaß zur Beleuchtung hinzufügen zu können, wird noch ein weiterer Faktor eingeführt, der spiegelnde Reflexionskoeffizient k_s .
- Der Glanz berechnet sich dann nach diesem sogenannten **Phong-Beleuchtungsmodell** so:

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

- 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.

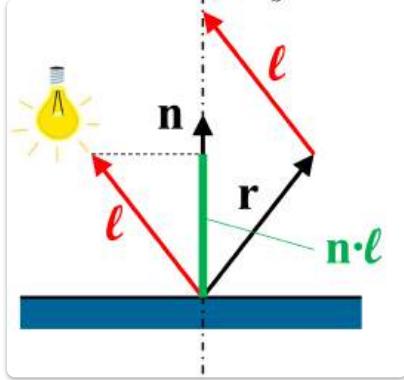


- Das Bild zeigt die Abhängigkeit dieser Funktion $W(\theta)$ vom Winkel zwischen Lichteinfall und Normale auf der Oberfläche für drei verschiedene Materialien (Silber, Gold, dielektrisches Material).



- 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:

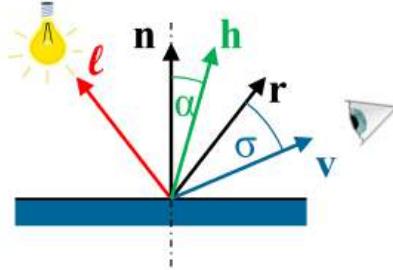
$$r = (2n \cdot l)n - l$$



- 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}

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



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

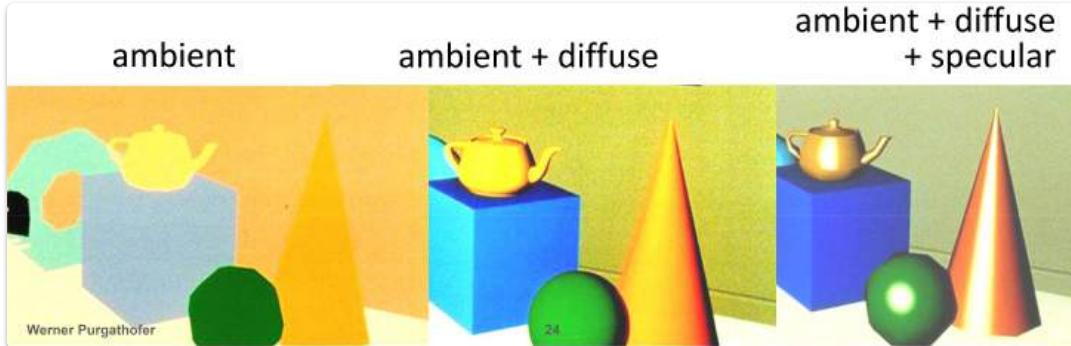
this revised model is called
“Blinn-Phong Shading”

- 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.

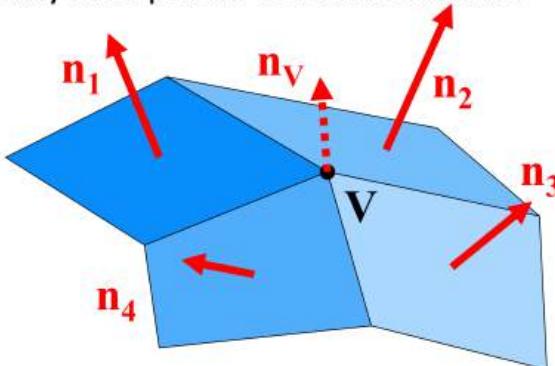


Konkret geht das so:

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.

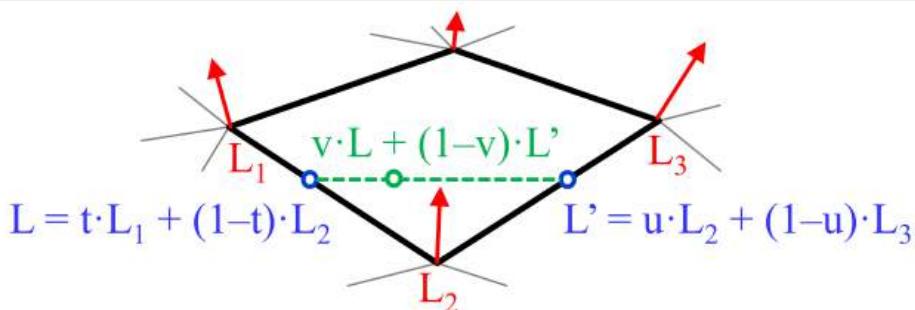
intensity-interpolation:

- 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).

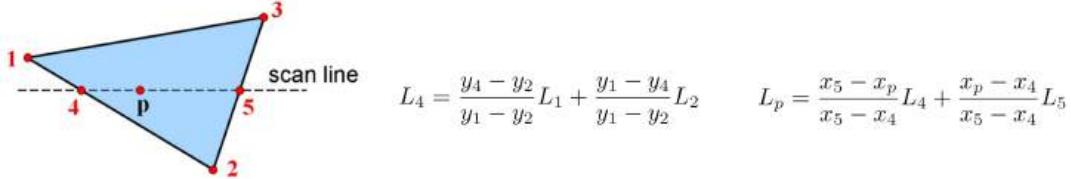


- find normal vectors at corners and calculate shading (intensities) there: L_i
- interpolate intensities along edges linearly: L, L'
- interpolate intensities along scanlines linearly: L_p

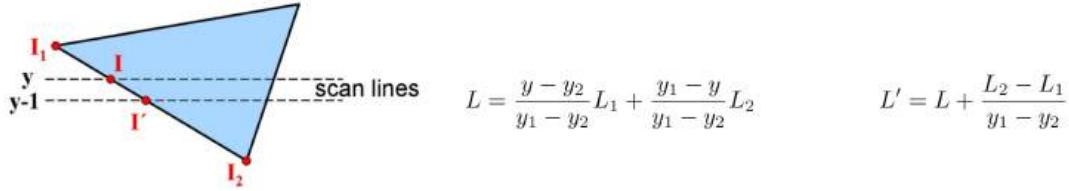
Dennoch verbleiben **Fehlerquellen**. So wird die Silhouette natürlich nicht verändert, dadurch verbleiben störende Polygonkanten sichtbar:



Einfache lineare Interpolation zur Berechnung eines Pixelwertes L_p :



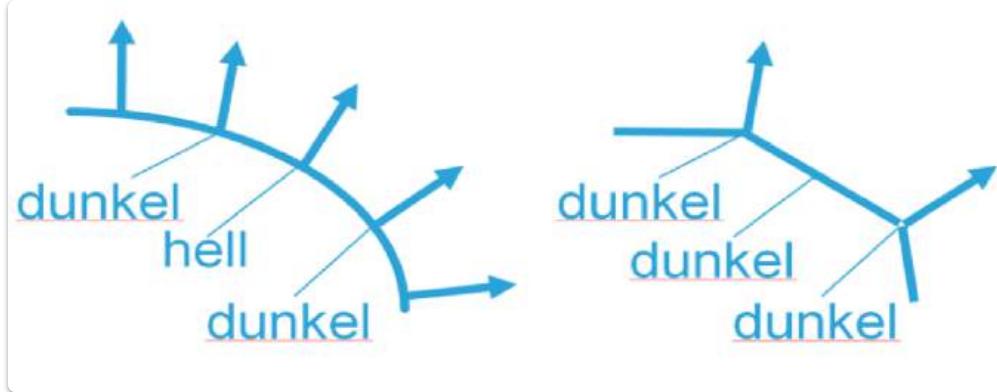
Die *lineare (!) Interpolation* der Intensitäten kann natürlich wieder inkrementell erfolgen, z.B.:



Probleme bei Gouraud-Schattierung (Glanzpunkte)

[EVC_Skriptum_CG, p.37](#)

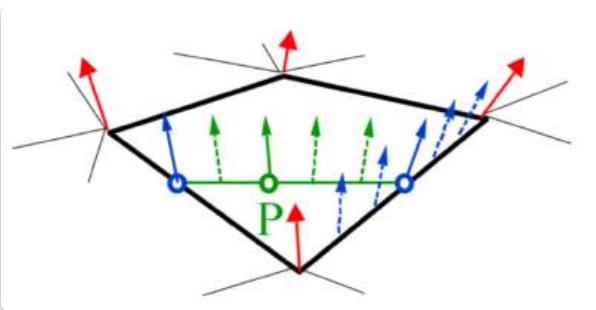
- Zufällige Interpolationsergebnisse bei Glanzpunkten möglich.
- Abhängig davon, ob Eckennormale zufällig Glanzpunkt erzeugt.
- Störend bei bewegten Objekten (Glanzpunkt "wandert").



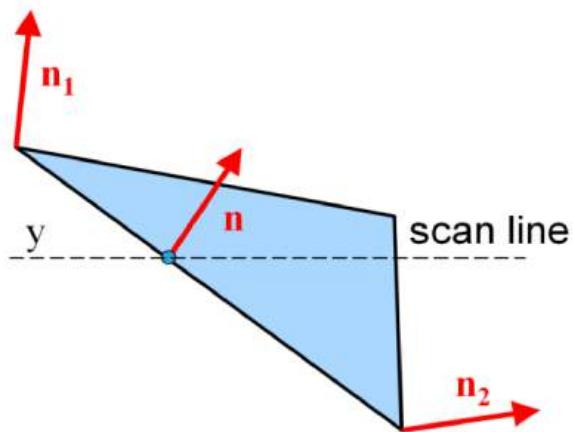
Phong-Schattierung

[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:



$$n = \frac{y - y_2}{y_1 - y_2} n_1 + \frac{y_1 - y}{y_1 - y_2} n_2$$

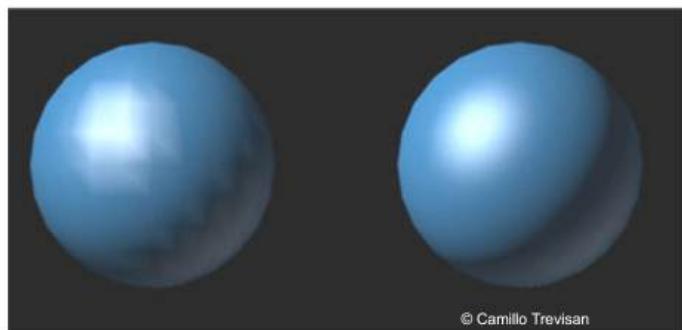
ZUR BEACHTUNG:

- 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!



© Camillo Trevisan

10. Ray-Tracing

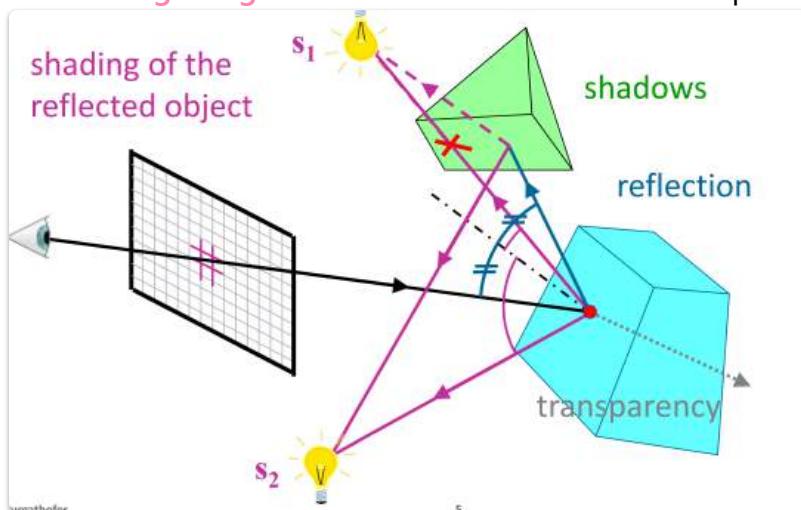
EVC_Skriptum_CG, p.39

- **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

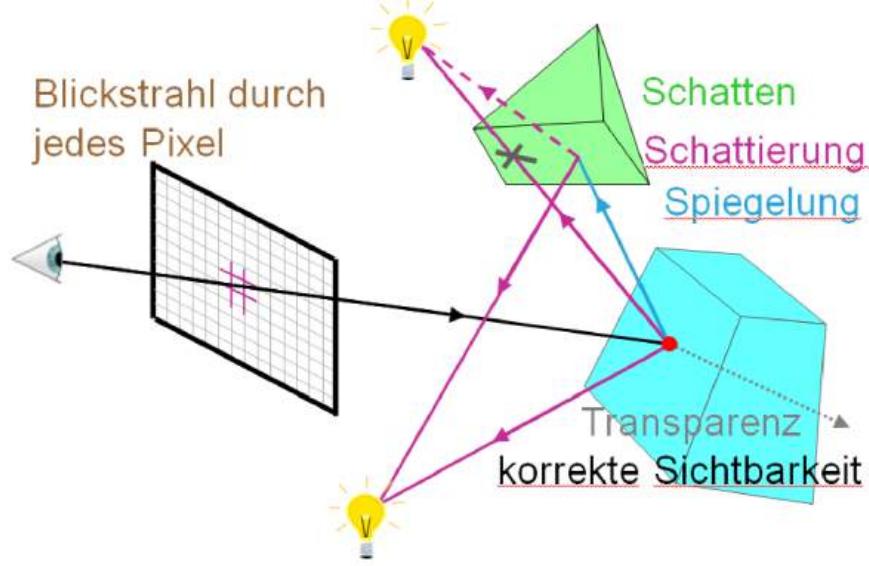
- **Blickstrahl (Primärstrahl):** Durch jeden Bildpunkt legen.
- **Schnitt:** Blickstrahl mit allen Oberflächen der Szene schneiden.
- **Nächster Schnittpunkt:** Denjenigen auswählen, der am nächsten zum Bild liegt.
- **Schattierung:** Schattierung dieses Objektpunktes (aus Blickrichtung) als Pixelwert.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



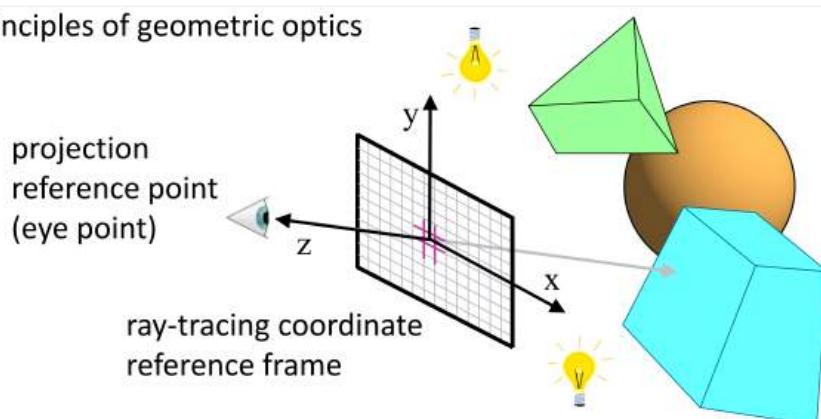
Schatten

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.



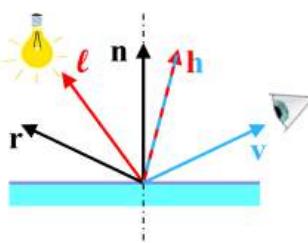
principles of geometric optics



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

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

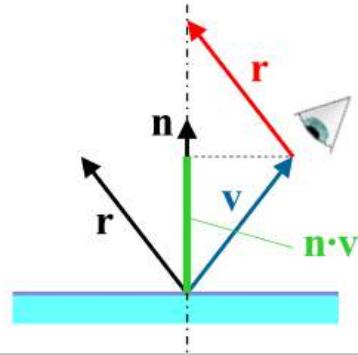
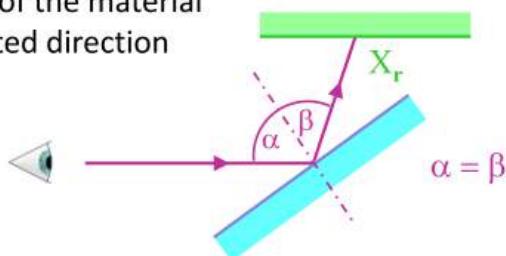
[EVC_Skriptum_CG, p.39](#)

- **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



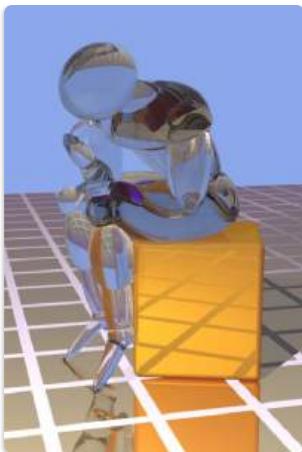
$$\mathbf{r} + \mathbf{v} = 2(\mathbf{n} \cdot \mathbf{v})\mathbf{n}$$

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

Transparenz

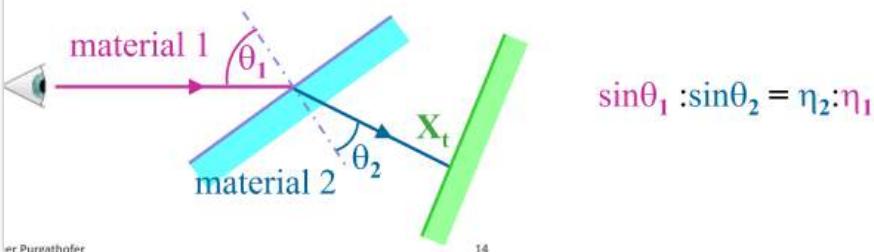
EVC_Skriptum_CG, p.39, p.40

- **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.



$$I_t = k_t \cdot X_t$$

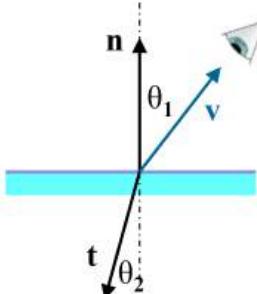
I_t ... illumination caused by transparency
 k_t ... transparency coefficient of the material
 X_t ... shading in the transparency direction



calculation of transparency ray

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

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



Rekursion

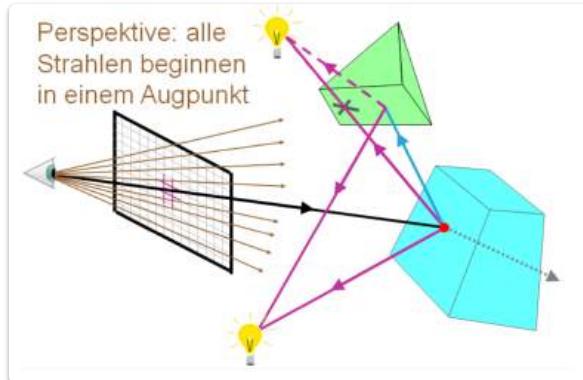
EVC_Skriptum_CG, p.40

- **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 .

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

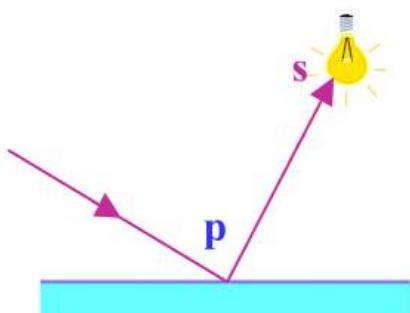
- n : Oberflächennormale.
- v : Richtung des einfallenden Strahls.
- $|r| = 1$ (garantiert durch Berechnung).

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

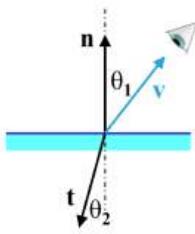
$$\text{ray} = \mathbf{p} + t \cdot (\mathbf{s} - \mathbf{p})$$

p ... intersection point

s ... light source position



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

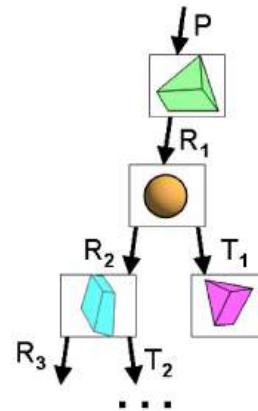
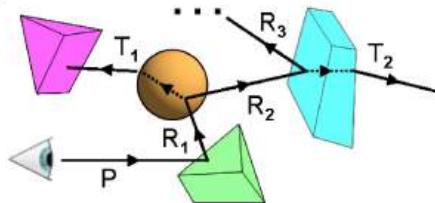


Transparenzstrahlen sind $p + t \cdot t$, wobei t sich aus dem Snellius'schen Brechungsgesetz $\sin \theta_1 : \sin \theta_2 = \eta_2 : \eta_1$ berechnet (η_i ist der Brechungsindex des Materials i):

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

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:

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

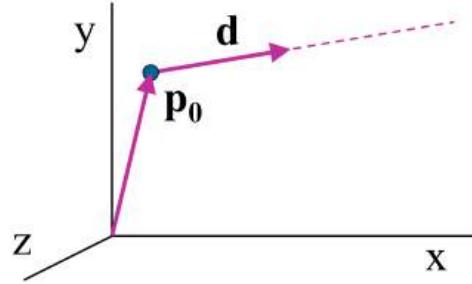
for primary rays:

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

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

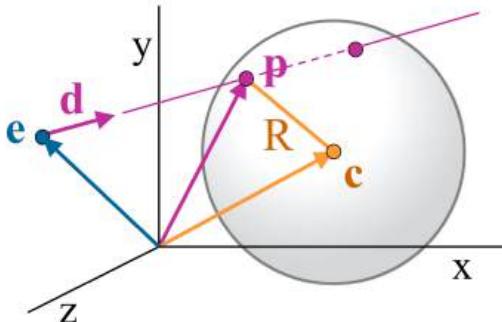
$$|\mathbf{p} - \mathbf{c}|^2 - R^2 = 0$$

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

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

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

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



if discriminant is negative \Rightarrow no intersections

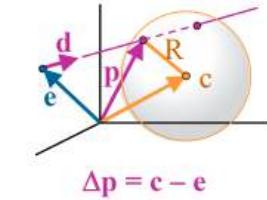
$$t = \mathbf{d} \cdot \Delta \mathbf{p} \pm \sqrt{(\mathbf{d} \cdot \Delta \mathbf{p})^2 - |\Delta \mathbf{p}|^2 + R^2}$$

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

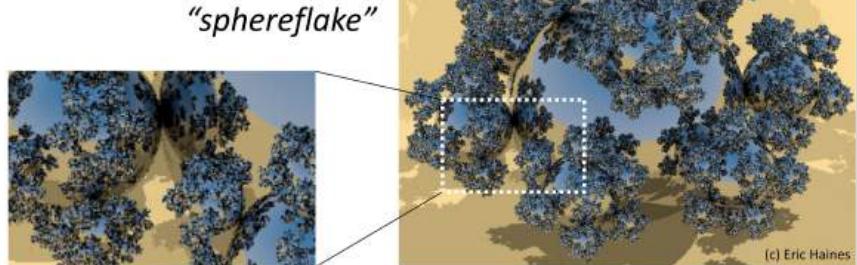
$$|\Delta \mathbf{p} - (\mathbf{d} \cdot \Delta \mathbf{p})\mathbf{d}|^2 = |\Delta \mathbf{p}|^2 - 2\mathbf{d}^2|\Delta \mathbf{p}|^2 + (\mathbf{d} \cdot \Delta \mathbf{p})^2\mathbf{d}^2$$

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

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

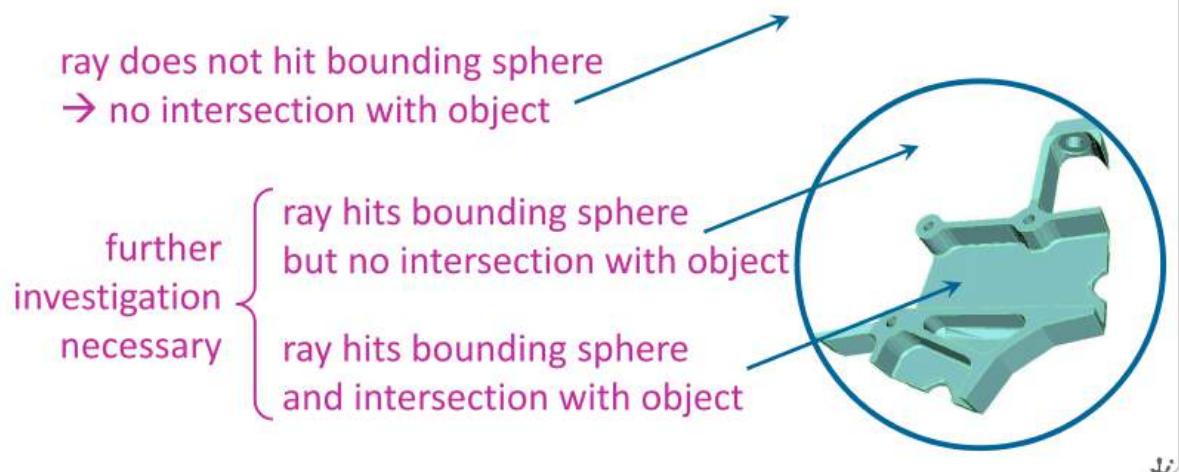


Werner Purgathofer



(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 $\mathbf{d} \cdot \mathbf{n} < 0$

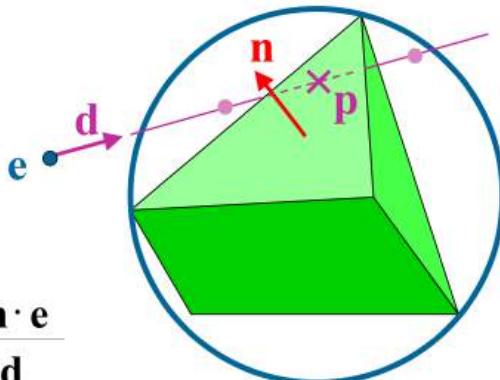
3. solve plane equation

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

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

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

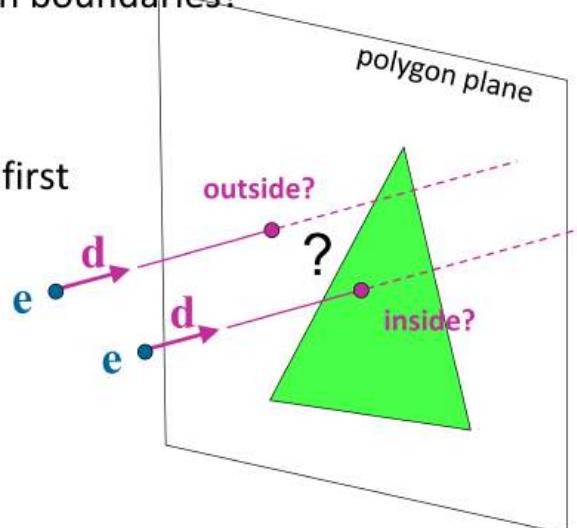
$$\mathbf{n} \cdot (\mathbf{e} + t\mathbf{d}) = -D \quad \Rightarrow \quad t = -\frac{D + \mathbf{n} \cdot \mathbf{e}}{\mathbf{n} \cdot \mathbf{d}}$$



4. intersection point inside polygon boundaries?

perform inside-outside test

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



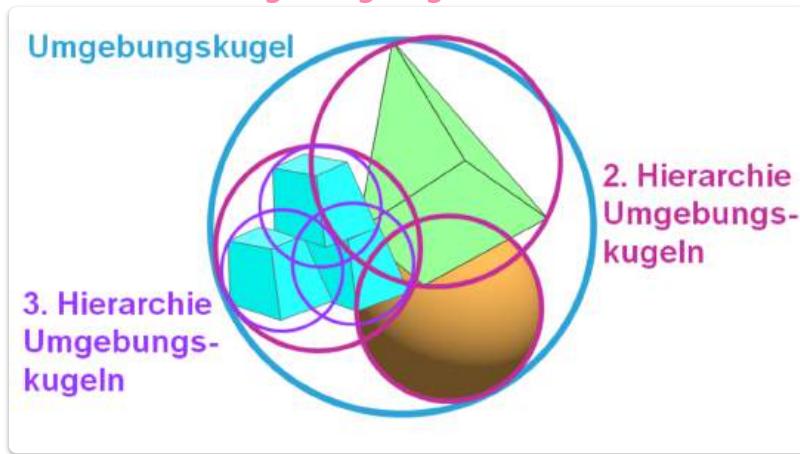
Ray-Tracing Beschleunigung

- **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 $O(\log n)$.

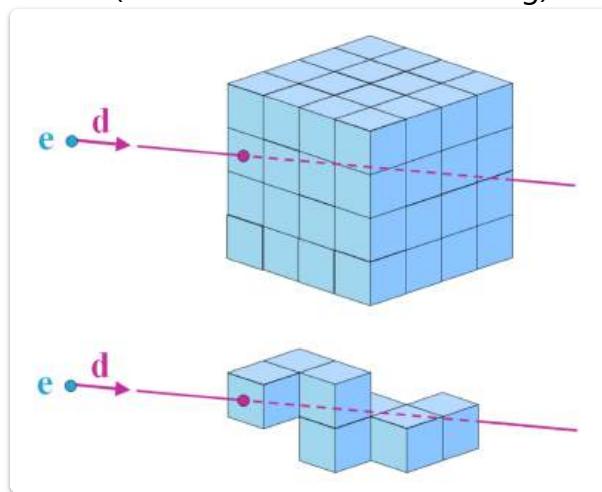
- **Alternative Objektumgebungen:**

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

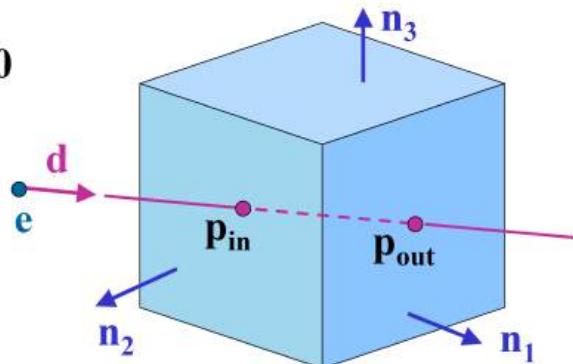
Slides:

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

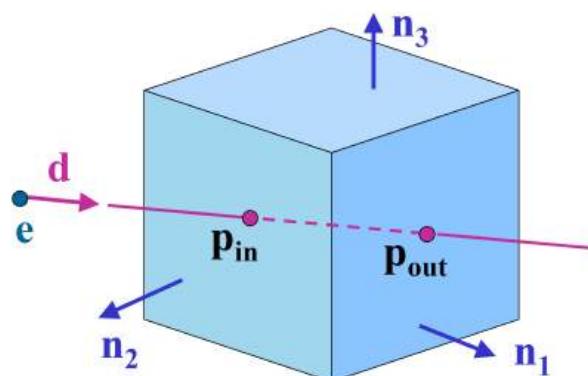
$$\mathbf{p}_{out,k} = \mathbf{p}_{in} + t_k \mathbf{d}$$

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

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

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

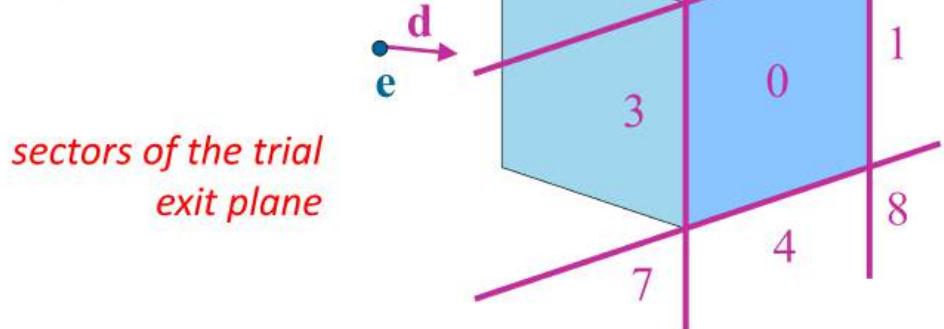
$$x_k = -D_k \Rightarrow t_k = \frac{x_k - x_0}{x_d}$$



variation: trial exit plane

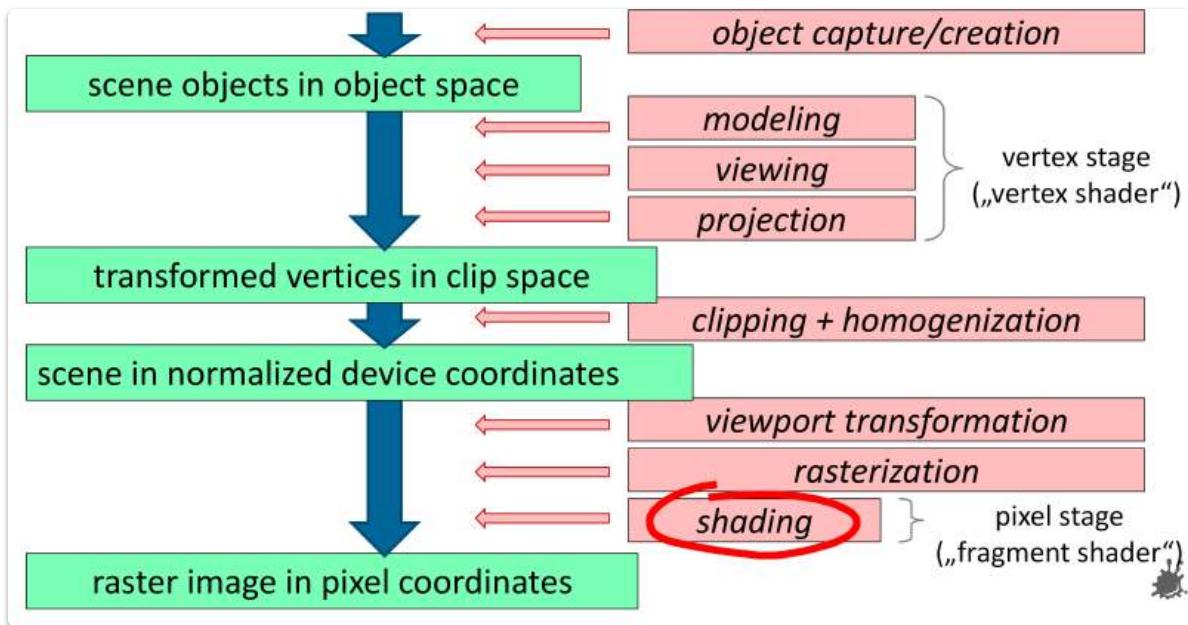
perpendicular to largest component of \mathbf{d}

- (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

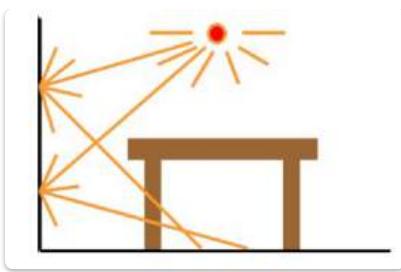
Rendering Pipeline



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:**

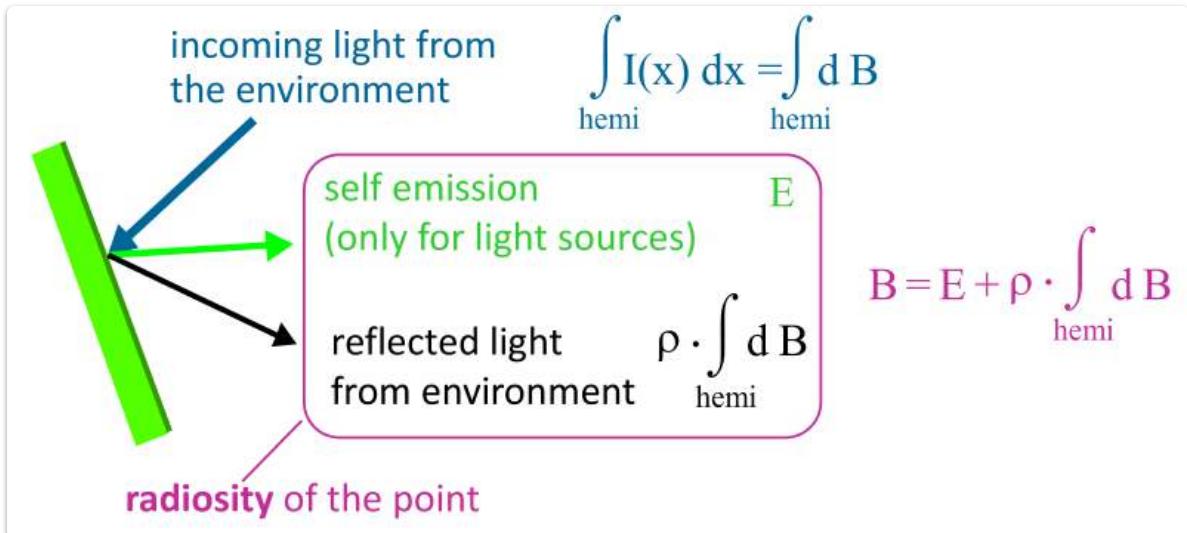
$$B_i = E_i + \rho_i \sum_{j=1}^n B_j F_{ij}$$

- 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**

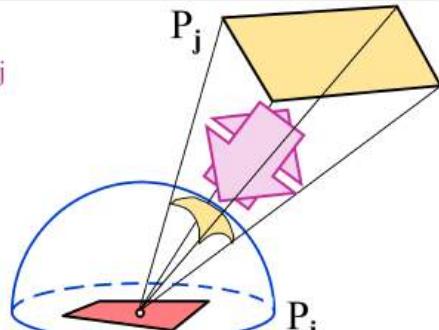
E_i ... self-emission of patch **i**

$\sum B_j F_{ij}$... 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

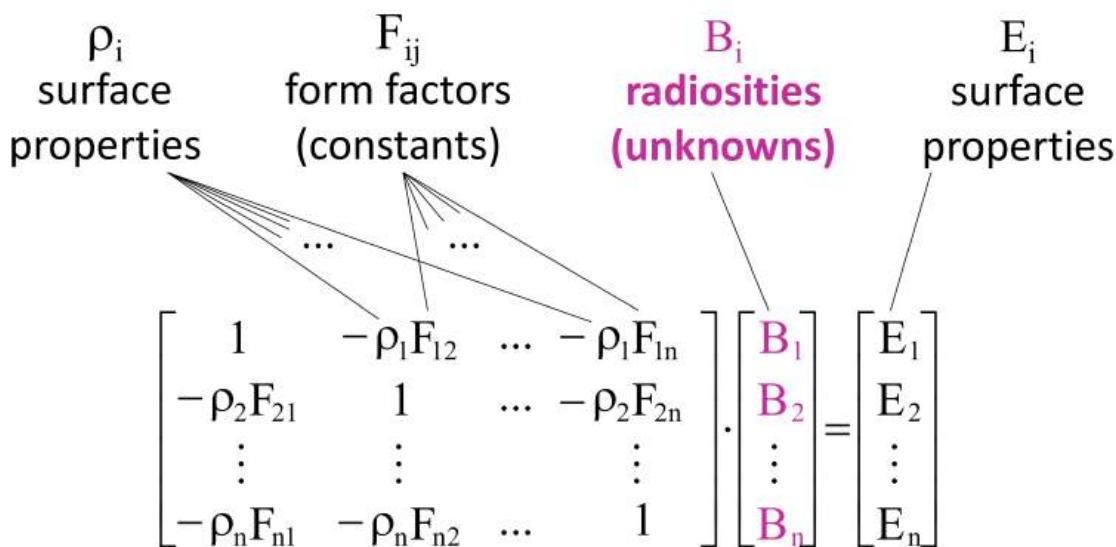
ρ_i ... reflectivity coefficient of patch **i** (“*albedo*”)



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

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

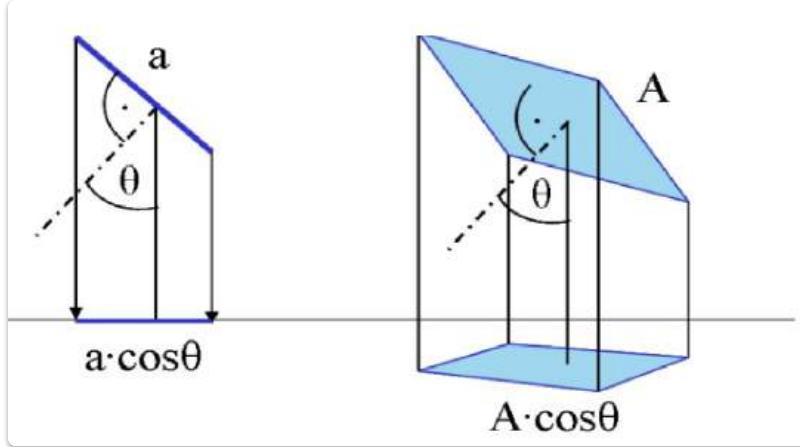
$$\begin{bmatrix} 1 & -\rho_1 F_{12} & \dots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1 & \dots & -\rho_2 F_{2n} \\ \vdots & \vdots & & \vdots \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \dots & 1 \end{bmatrix} \cdot \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots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



Berechnung der Formfaktoren

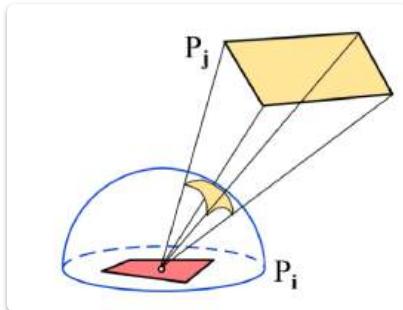
[EVC_Skriptum_CG, p.43, p.44](#)

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



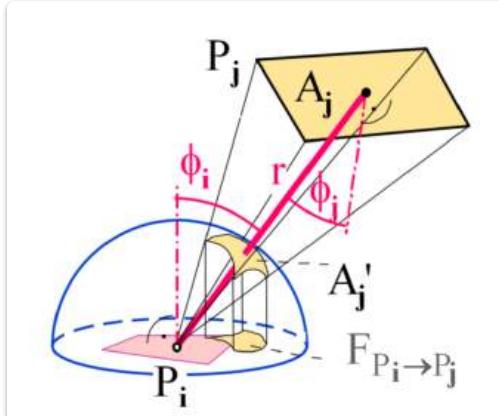
- **Definition des Formfaktors F_{ij} :**

- 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

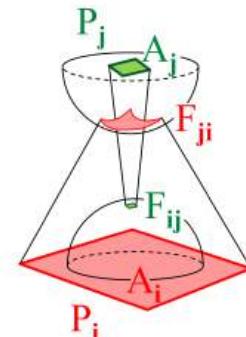
$$\sum_{j=1}^n F_{ij} = 1$$

■ uniform light reflection

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

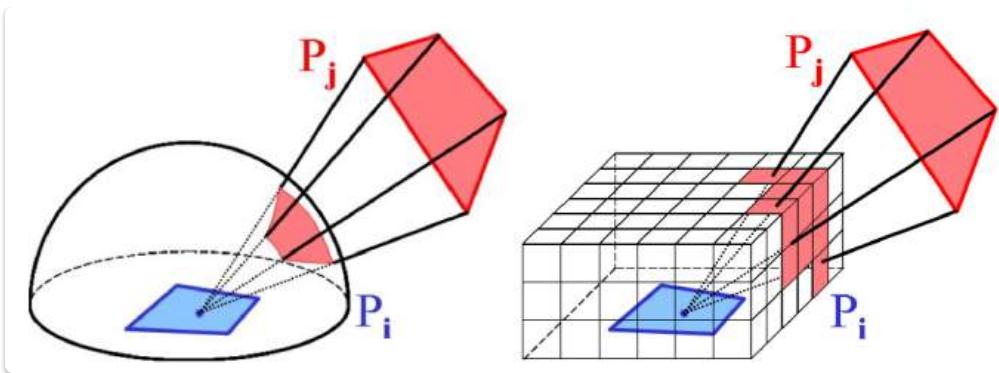
■ no self-incidence

$$F_{ii} = 0$$



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 Formfaktanteile 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

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

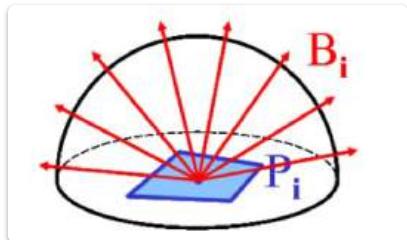
- **Ziel:** Iterative Lösung des Radiosity-Gleichungssystems.
- **Grundidee ("Shooting"):** Man wählt das jeweils hellste Patch aus und "schießt" (verteilt) dessen noch nicht abgestrahlte Energie auf alle anderen Patches.
- **Vorteil:**
 - In jedem Schritt werden alle Patches etwas besser beleuchtet.
 - 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

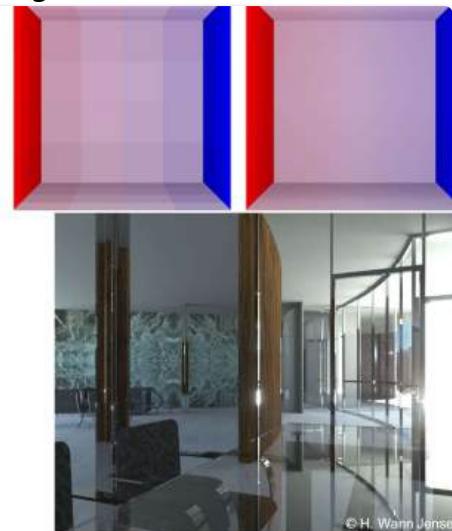
- polygon rendering
- Gouraud shading
- ray-tracing
- ...

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.

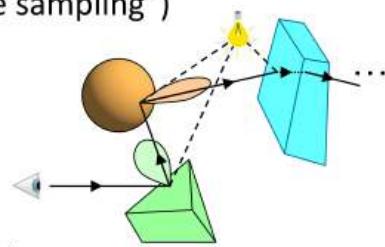
also called Monte Carlo ray tracing

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:**

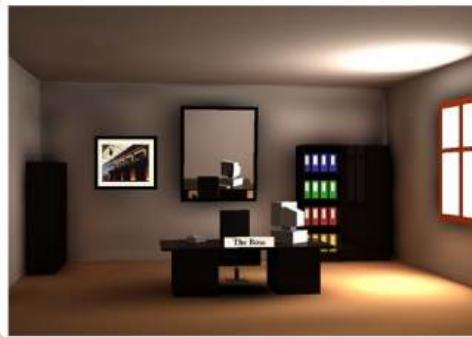
1. **"Vorwärts"-Verfolgung von Lichtstrahlen:** Im Gegensatz zum Ray-Tracing, das Strahlen von der Kamera verfolgt, werden beim Photon Mapping Lichtstrahlen (genannt "Photonen") von den Lichtquellen ausgesendet und in der Szene verfolgt. Dies ist eine "Vorwärts"-Richtung.
2. **Speichern der Lichtwirkung 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.

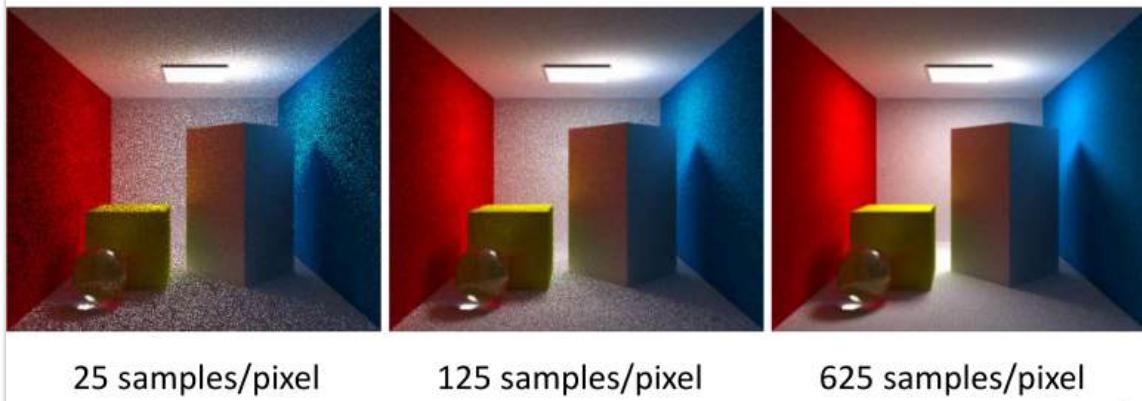


Optische Eigenschaften natürlicher Oberflächen: Mapping-Methoden

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.

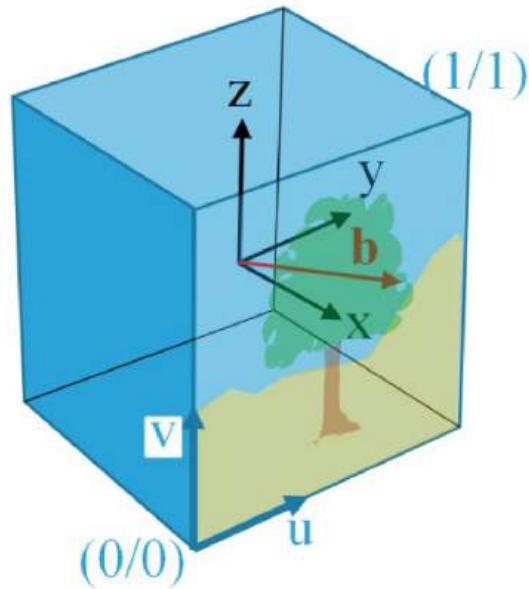
→ trace light rays from light source(s) and store illumination on objects



Environment Mapping

[EVC_Skriptum_CG, p.46](#)

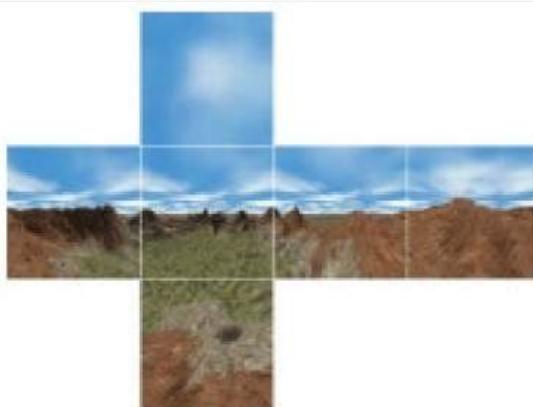
- **Definition:** Environment Mapping (Umgebungsabbildung) bezeichnet die Technik, wie die Umgebung oder Umwelt eines Objekts dessen Aussehen beeinflusst, insbesondere durch Reflexionen.
- **Effekt:** Je nach Oberflächeneigenschaften des Objekts wird die Umgebung auf verschiedene Weisen im Erscheinungsbild 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.

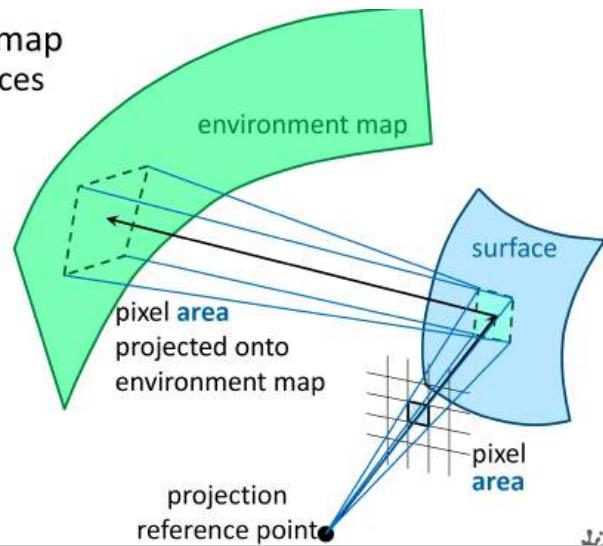


information in the environment map

- 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.

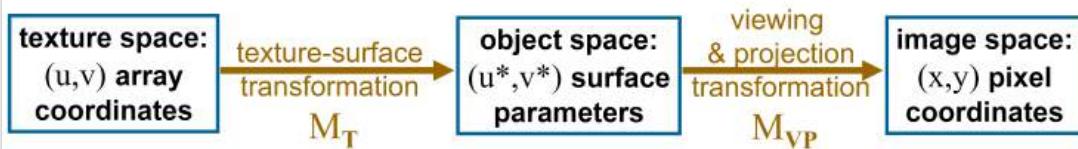
Raum: Dies beinhaltet die Transformation von den **Objekt-Raum (u',v')** **Flächen-Parametern** in **Bild-Raum (x,y) Pixel Koordinaten**.

* **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 $(x,y) \rightarrow (u,v)$

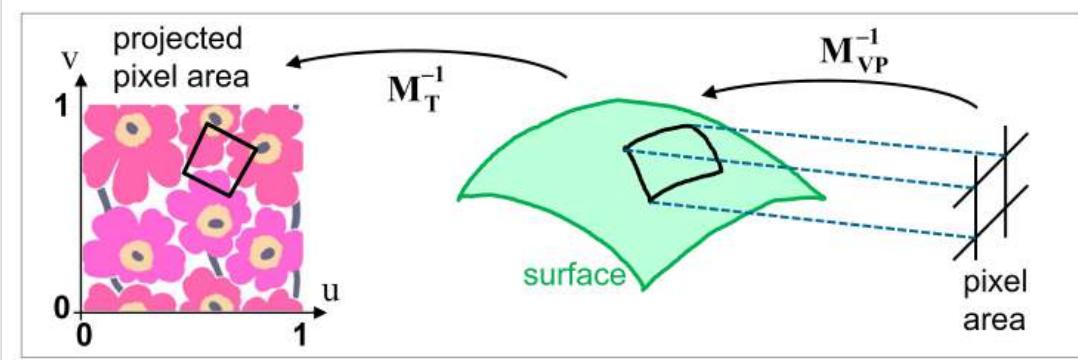


texture-surface transformation

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

projecting pixel areas to texture space =

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



Erzeugung einer Textur

[EVC_Skriptum_CG, p.47](#)

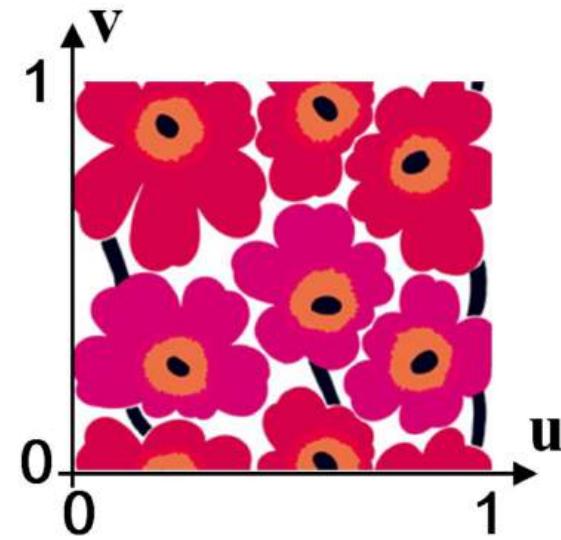
- **Grundsatz:** Die Herkunft einer Textur ist zweitrangig; wichtig ist nur, dass sie an allen benötigten Stellen definiert und abrufbar ist.
- **Standardverfahren:** Meist wird eine Textur in einem Vorverarbeitungsschritt als **Pixel-Array** (also ein Raster von Farbwerten) erstellt und später im Rendering-Prozess 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 Zufallswerten (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.

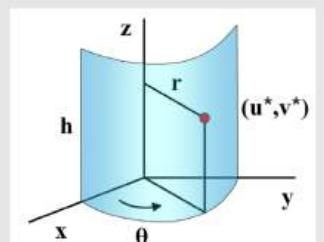


- **Ziel der Transformation:** Für jeden Punkt auf der 3D-Objektoberfläche (identifiziert durch seine (u^*, v^*) -Parameter) soll die zugehörige Farbe aus der Textur ermittelt werden. Das bedeutet, wir brauchen eine Abbildung von den Objekt-Oberflächenparametern zurück zu den Texturkoordinaten, oder 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: $x = r \cdot \cos u_*$, $y = r \cdot \sin u_*$, $z = v_*$.

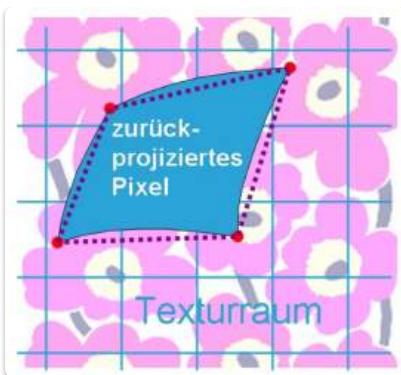
Nun wird das Problem von hinten angegangen:

- 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.

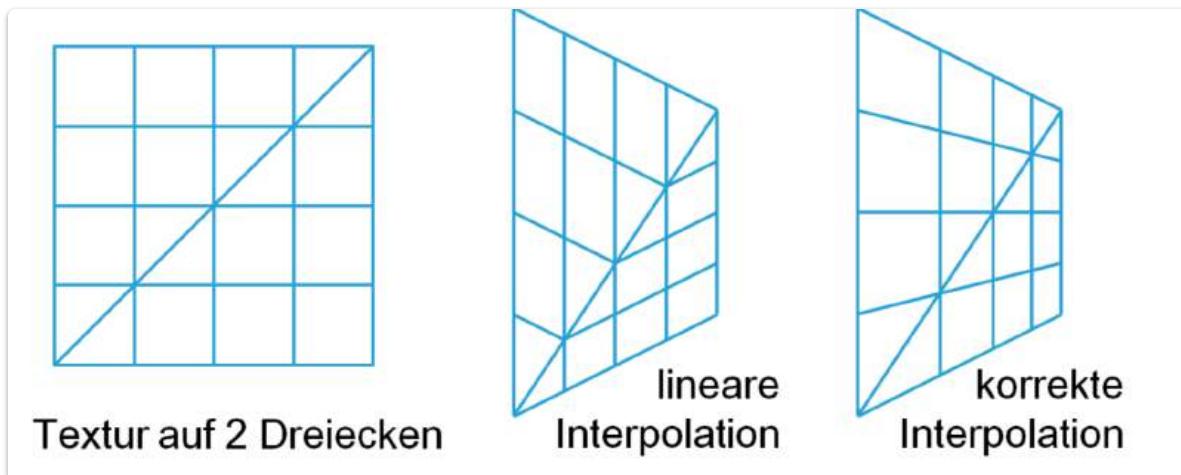
Texturen auf perspektivisch verzerrten Dreiecken

[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.

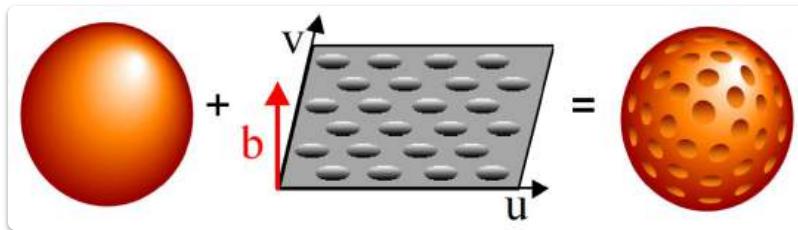


Betrachtet man die graue Leiste, scheint sie sechs Ausbuchtungen und eine Einbuchtung zu haben. Fühlt man die Oberfläche jedoch an, stellt man fest, dass sie vollkommen eben ist.

- **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:

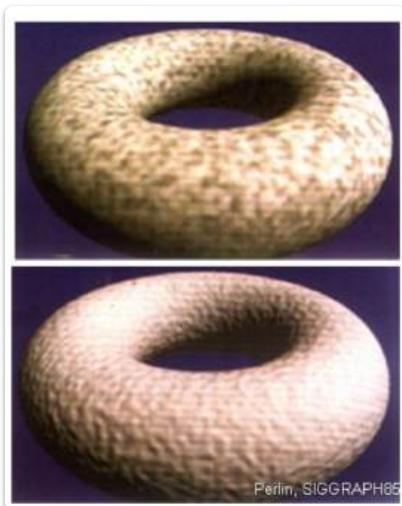
$$n' = p'_u \times p'_v = p_u \times p_v + b_v(p_u \times n) + b_u(n \times p_v) + b_u b_v(n \times n)$$

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**.



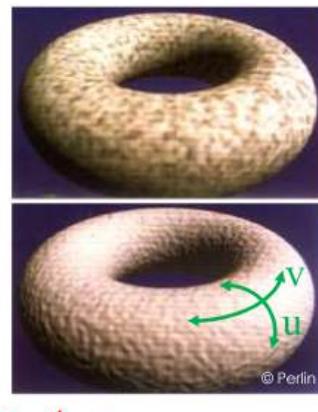
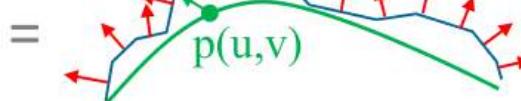
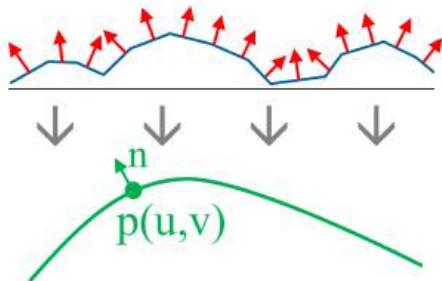
Beachte den Unterschied in der Donut-Abbildung:

- **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

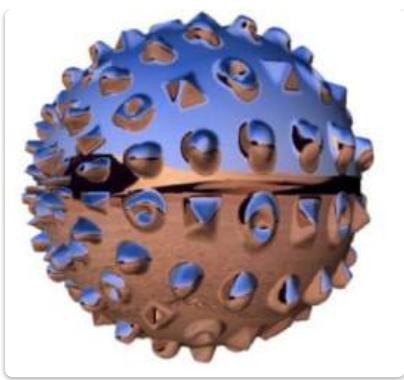
[EVC_Skriptum_CG](#), p.50

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**.

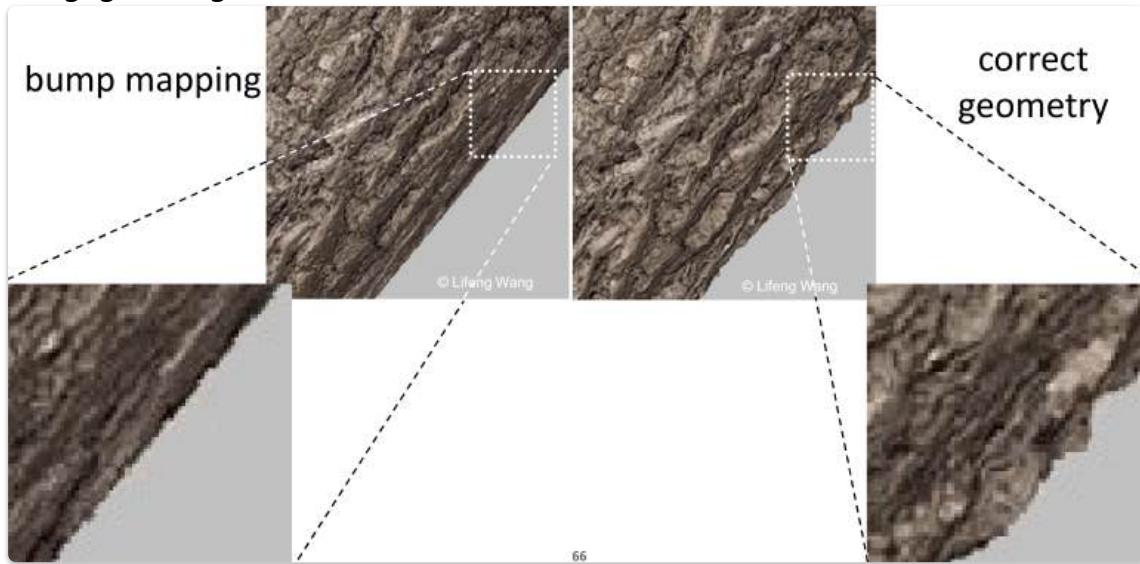
Displacement Mapping

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.

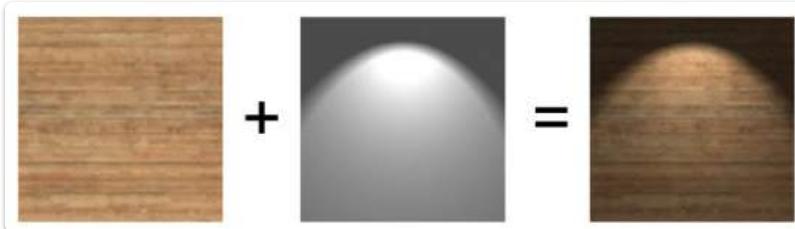
- **Tessellation-Stage:** Diese Stufe befindet sich **zwischen dem Vertex- und dem Pixel-Stage**. Ihre Aufgabe ist es, die Dreiecke direkt auf der Hardware in **viele kleine Dreiecke zu unterteilen** (Tessellation). Jedes dieser neu erzeugten kleinen Dreiecke kann dann entsprechend einer **Displacement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

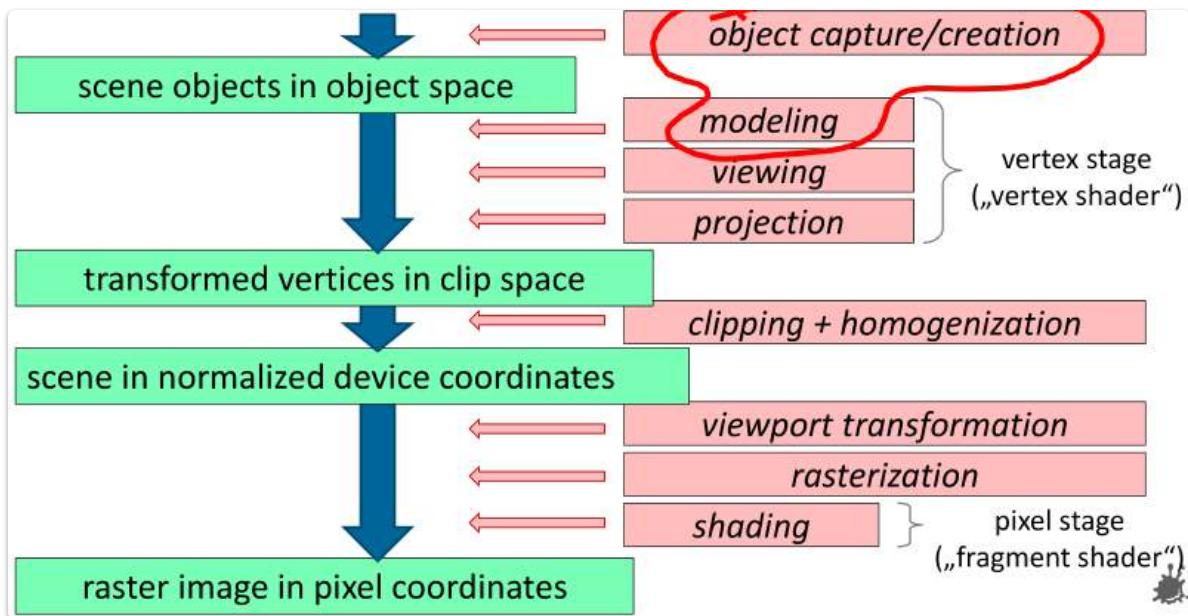
Kombination mehrerer Mappings

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

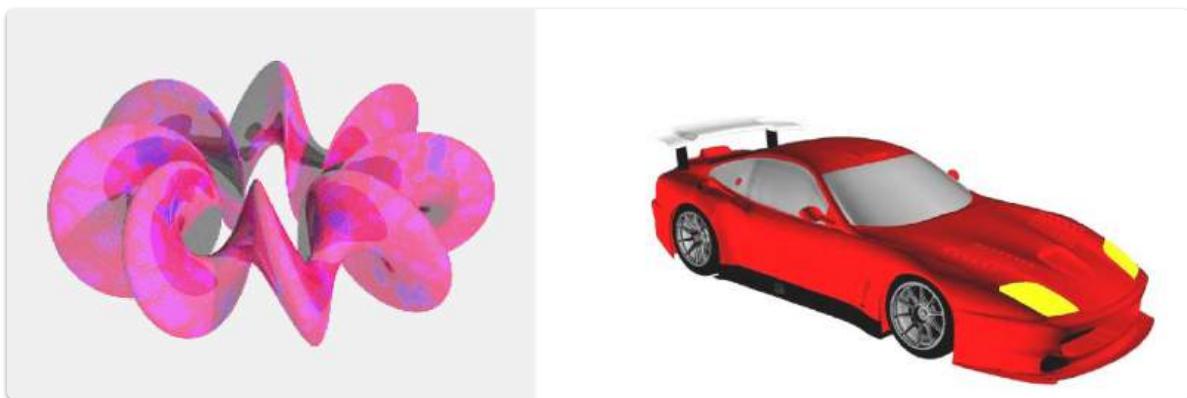


Freiformflächen (Modellierung und Darstellung)

[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:

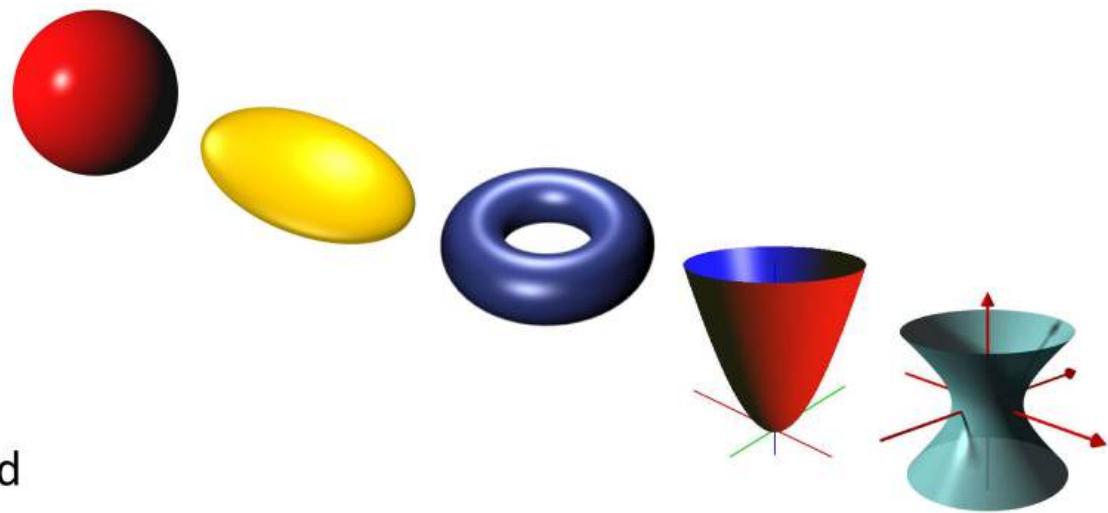
- Bei dieser Darstellung wird für jede Kombination von **Parameterwerten** (z.B. u, v oder ϕ, θ) ein Punkt auf der Oberfläche erzeugt. Die Koordinaten (x, y, z) sind Funktionen dieser Parameter.
- Beispiel Kugel:**
 - $x = r \cdot \cos \phi \cdot \cos \theta$
 - $y = r \cdot \cos \phi \cdot \sin \theta$
 - $z = r \cdot \sin \phi$
 - 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:

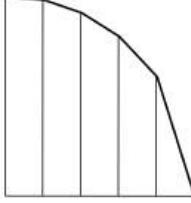
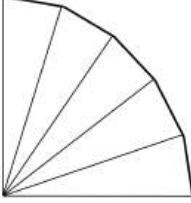
Ellipsoid: Eine gestreckte oder gestauchte Kugelform.

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> $y = \sqrt{1-x^2}$ 	$x=\cos(u) \quad y=\sin(u)$ 

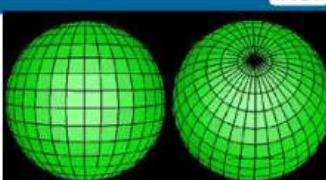
Mehr Formen

Hier nochmal mehr Informationen zu den Formen aus den Slides

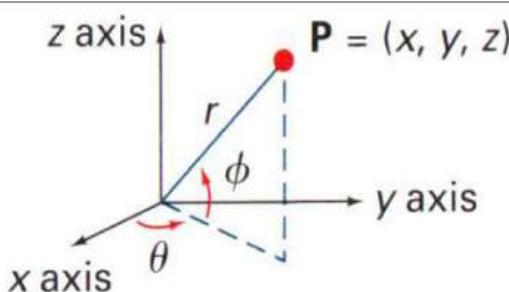
Quadric Surfaces: Sphere

- **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

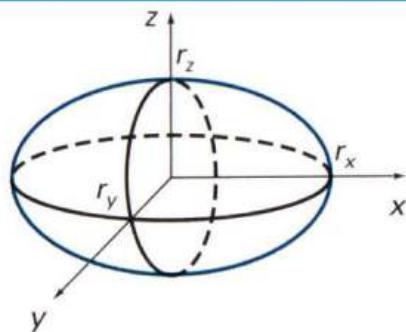


Werner Purgathofer

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:*

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

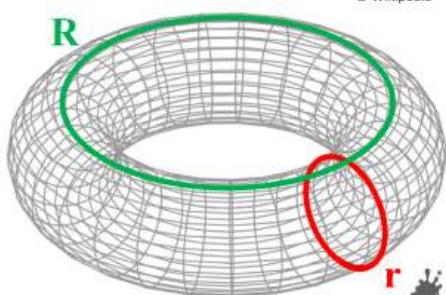


■ *parametric:*

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

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

$$z = r \sin \phi$$



Werner Purgathofer

9

Free Form Surfaces

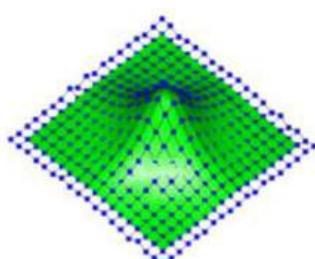
can be represented by

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](#)

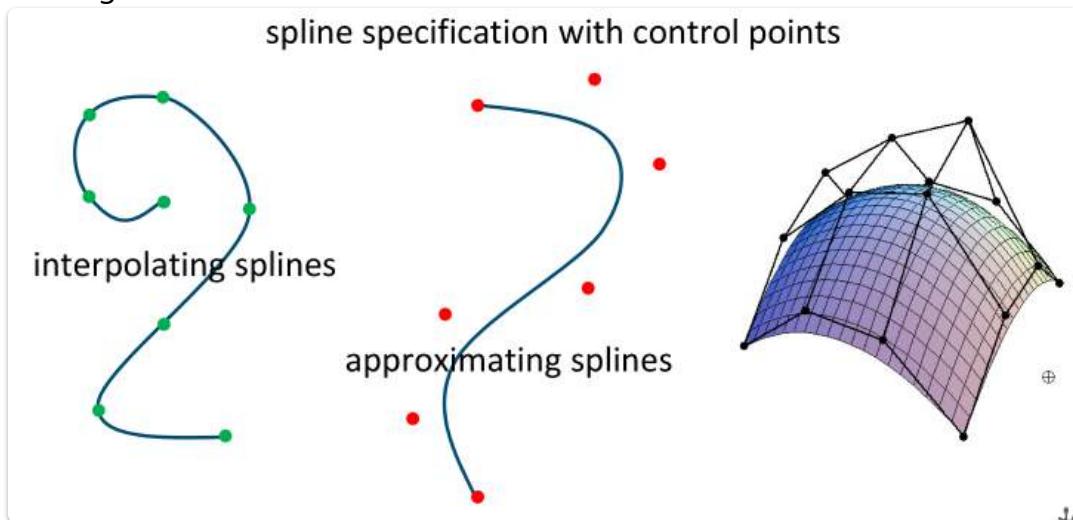
Kurven in der Computergrafik können auf zwei Arten definiert werden:

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.



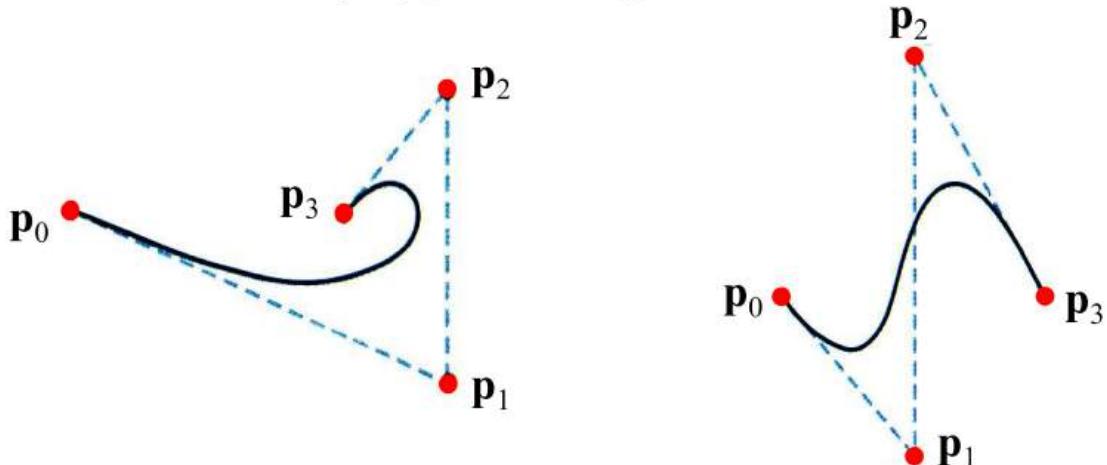
- **Stetigkeit an Verbindungsstellen:** Beschreibt die "Glätte" der Übergänge zwischen einzelnen Kurvensegmenten (z.B. keine Knicke oder abrupte Krümmungswechsel).
- **Einflussbereich von Punkten:**
 - **Globaler Einfluss:** Eine Änderung an einem Punkt wirkt sich auf die gesamte Kurve aus.
 - **Lokaler 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“)

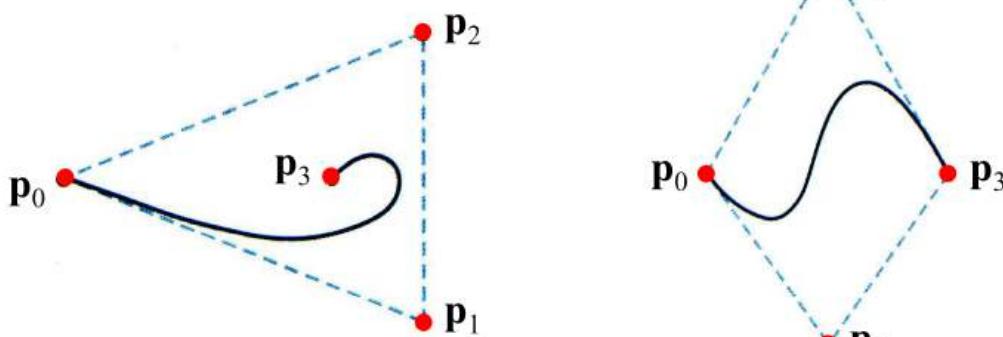
= polygon defining the curve



operations on splines:

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

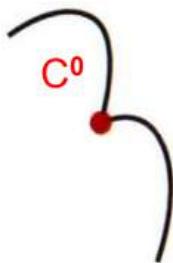
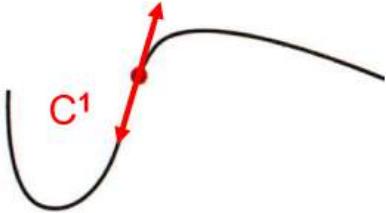
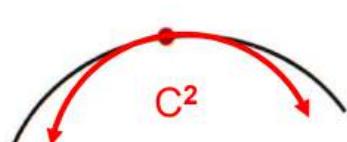
convex hull property



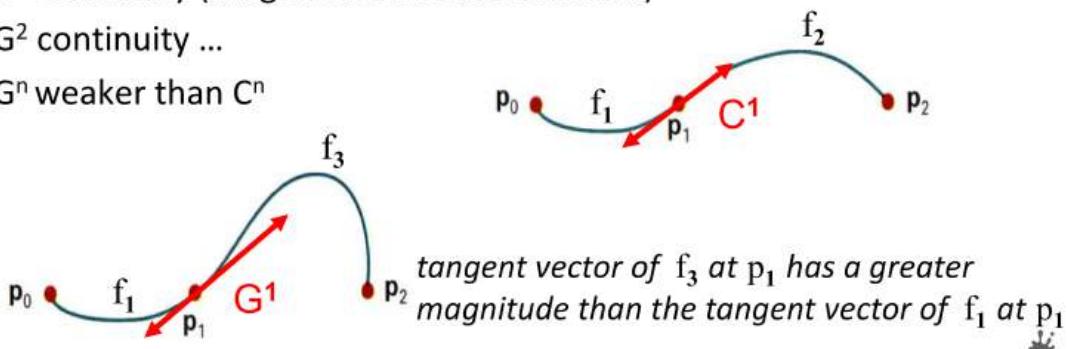
parametric continuity conditions (C^n)

$n = \text{number of derivations at section joints that are equal}$

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

 C^0 continuity **C^1 continuity** **C^2 continuity****geometric continuity conditions (G^n)**

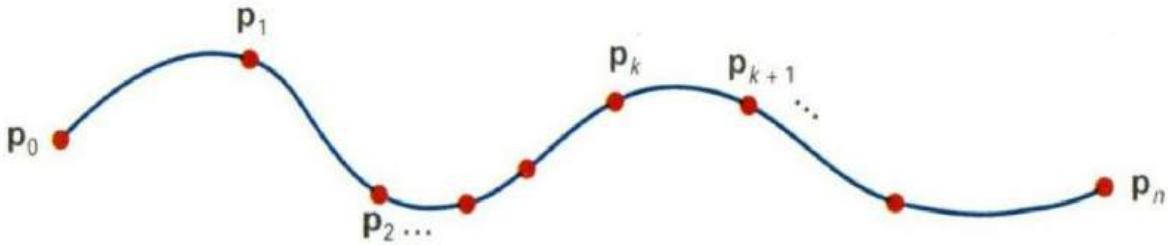
- derivations at joints have different magnitudes
- $G^0 (=C^0)$ continuity
- G^1 continuity (tangent vectors are collinear)
- G^2 continuity ...
- G^n weaker than C^n



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.

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

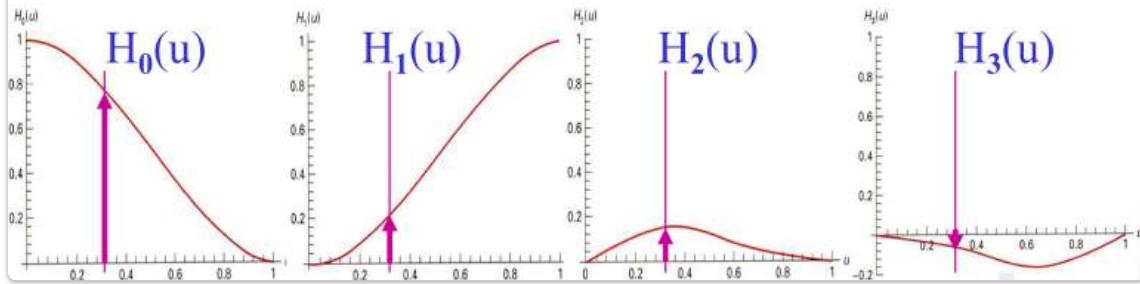
$$\begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{a}_k \\ \mathbf{b}_k \\ \mathbf{c}_k \\ \mathbf{d}_k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

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:

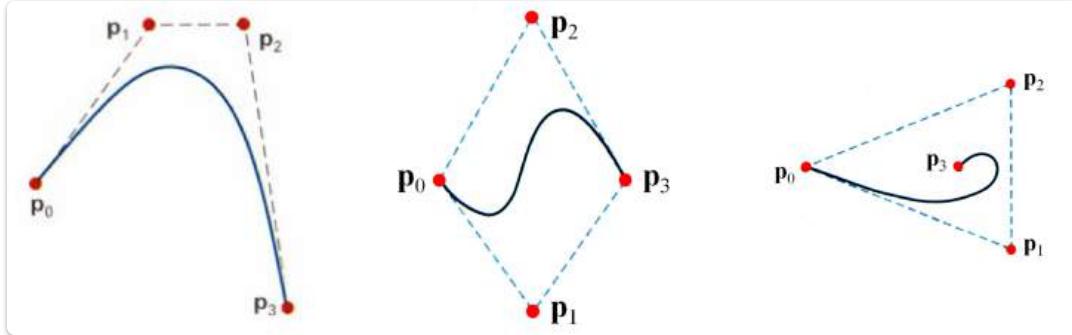


Bézier-Kurven

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

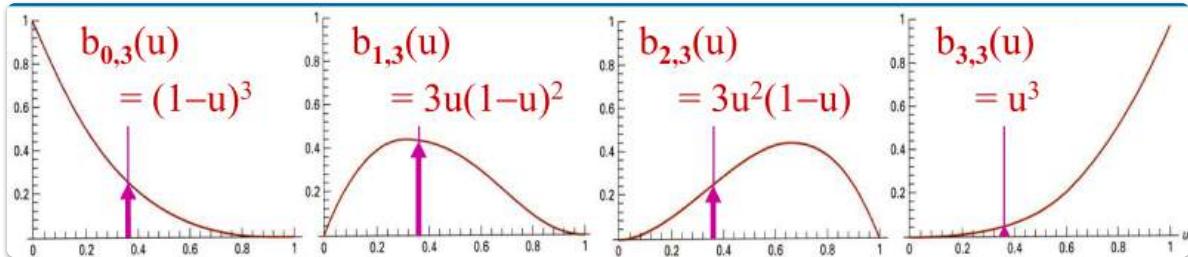
- Von Pierre Bézier um 1960 für die Beschreibung von Autokarosserien bei Renault beschrieben.
- Eine *approximierende* Kurve, die sogenannte *Bernstein-Polynome* $b_{k,n}$ als Gewichtsfunktionen für die Kontrollpunkte verwendet.
- Jeder Kurvenpunkt ist das gewichtete Mittel aller Kontrollpunkte:
 - $p(u) = \sum_{k=0}^n p_k b_{k,n}(u)$, 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).



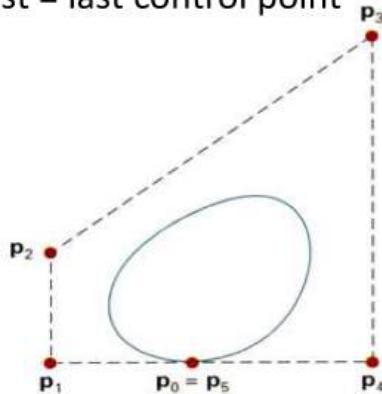
- Einige dieser Eigenschaften erkennt man aus der Form der Bernstein-Polynome $b_{k,n}$.

Weiteres aus den Slides Slides:

Bézier Curves Design Techniques (1)

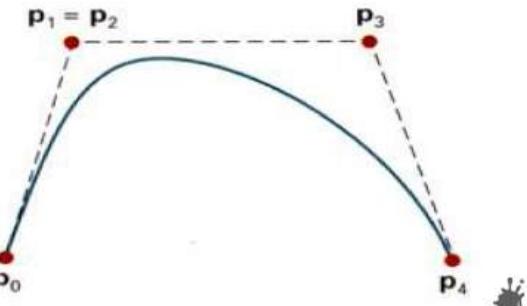
a ***closed Bézier curve***

generated by setting:
first = last control point



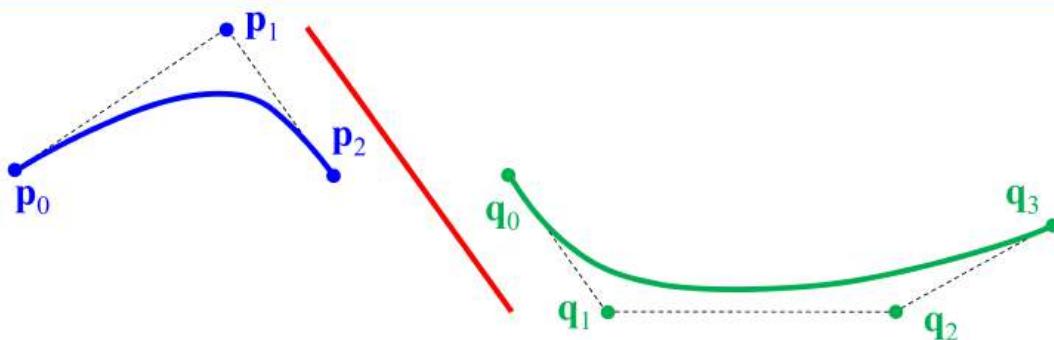
Werner Purgathofer

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.

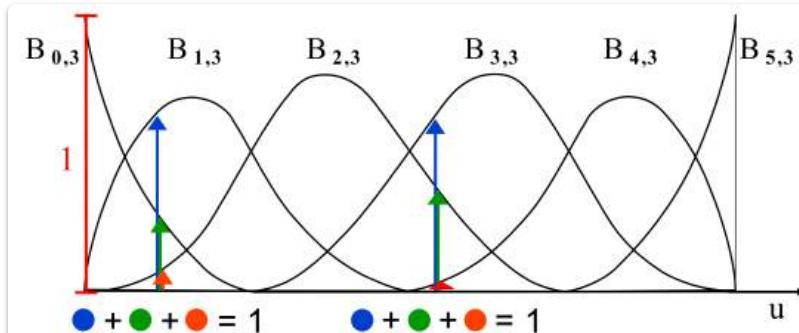
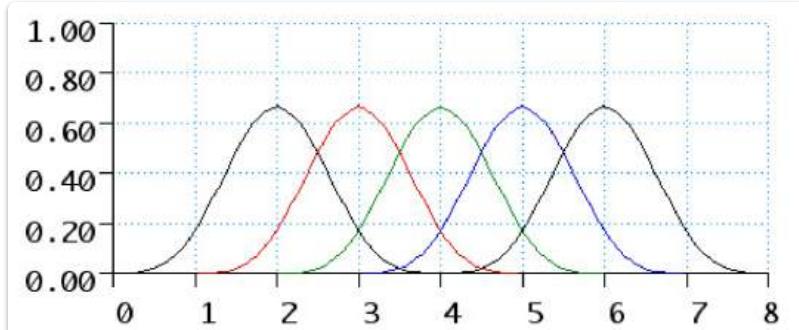
Cubic Bézier Curve Matrix Notation

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

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

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:

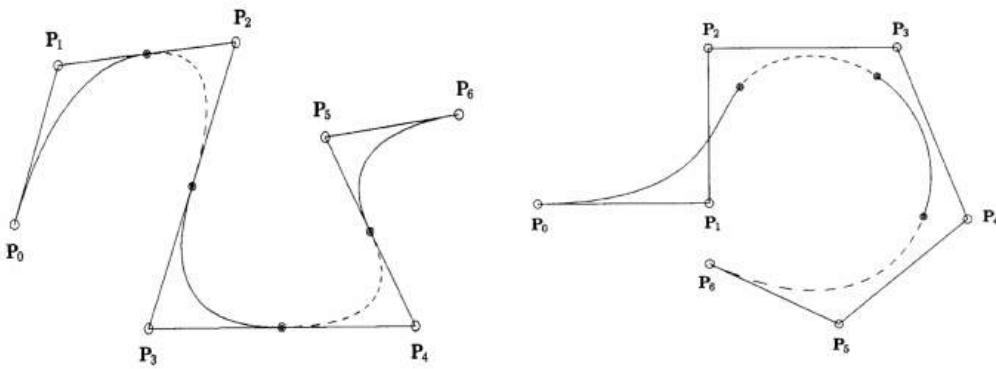
$$\sum_{k=0}^n B_{k,d}(u) = 1$$

- 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



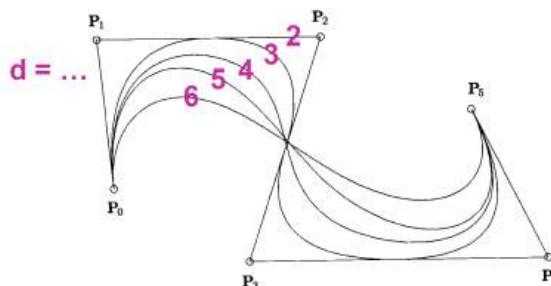
d describes, how many control points influence any point on the curve

$d = 2$ linear

$d = 3$ quadratic

$d = 4$ cubic

...



for $d=n+1$ you get Bézier curves!

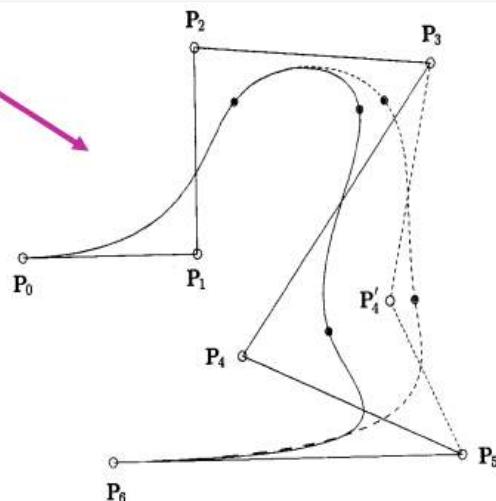
Hauptunterschiede zu Bézierkurven:

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

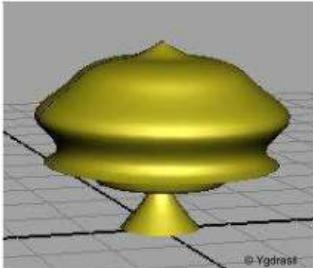


NURBS (Non-Uniform Rational B-Splines)

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



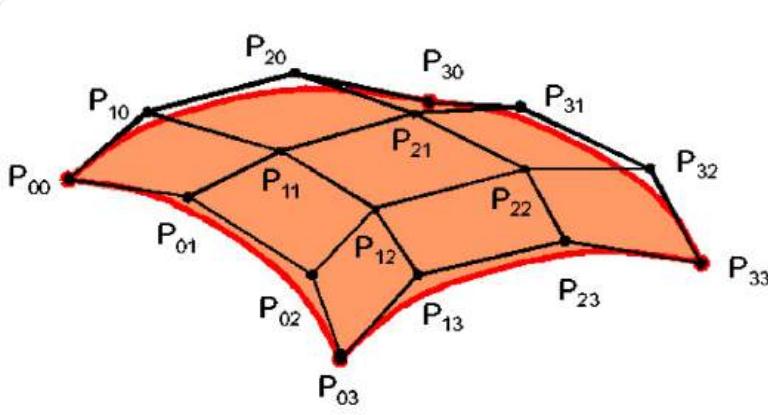
Freiformflächen -- Bézier- und B-Spline-Flächen

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.

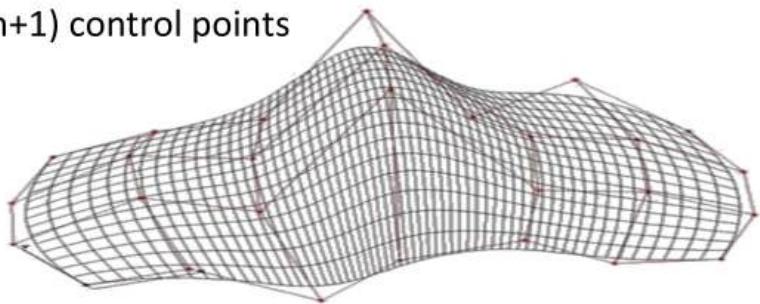


- Jedes Parameterpaar (u, v) führt zu einem Punkt der entstehenden Fläche.

- 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!



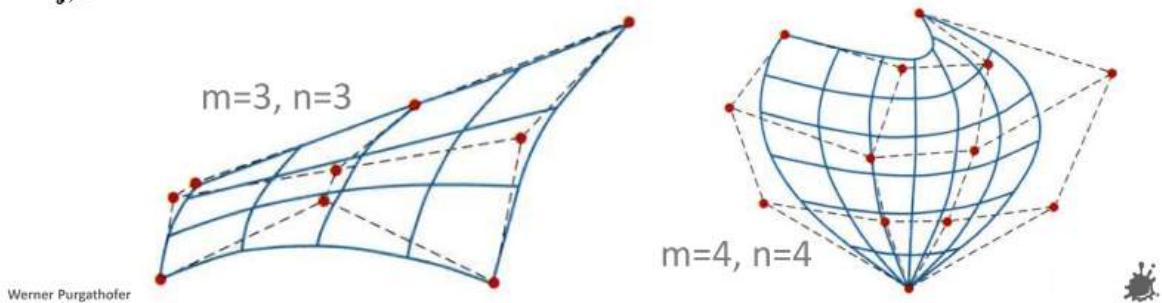
Bézier Surfaces Definition



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

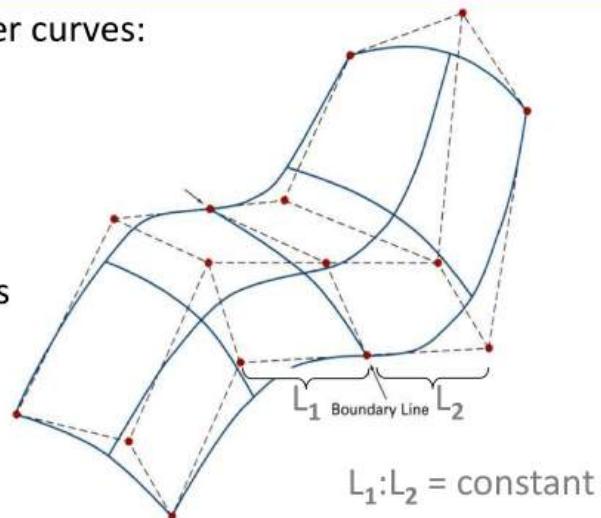


Bézier Surfaces Properties



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:

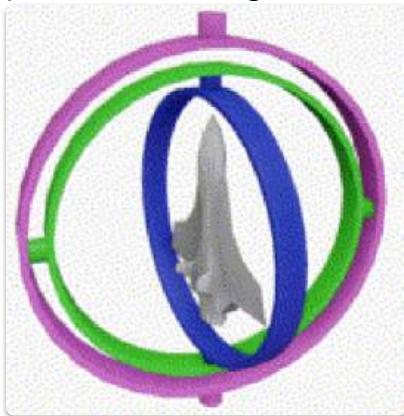
$$\mathbf{p}^{(t_1)} = \begin{bmatrix} 1 & \mathbf{x} \\ 0 & 1 \end{bmatrix} \mathbf{p}^{(t_0)}$$

- 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:

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

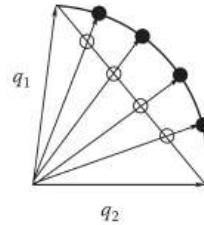


Figure taken from Shirley&Manzner:
Foundations of Computer Graphics, 5th Edition

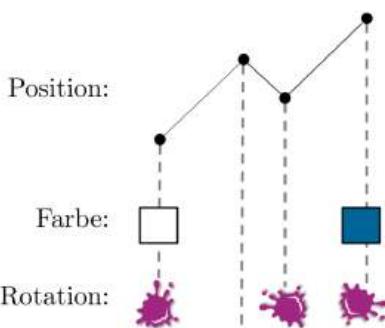
■ Apply a rotation to a point $\mathbf{p}^{(t)}$:

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

Keyframing

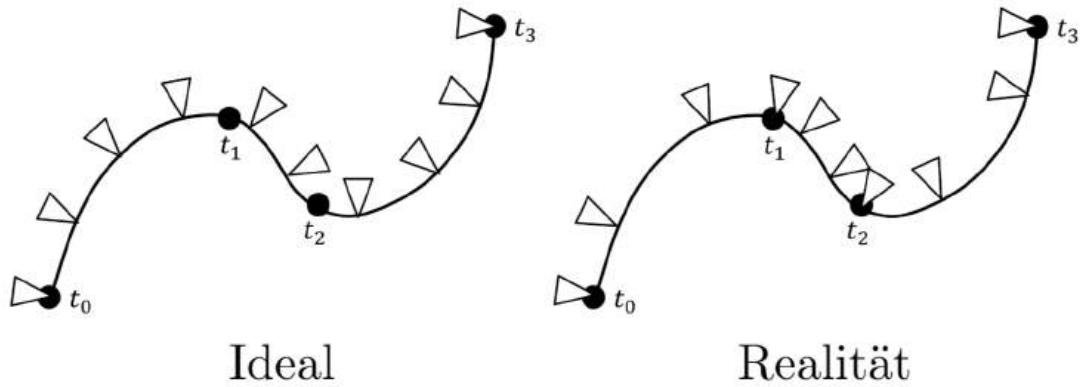
EVC_Skriptum_CG, p.55, p.56

- **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.



Zeit	0	1	2	3	4	5
Position	●			●	●	●
Farbe	●					●
Rotation	●				●	●

- 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.

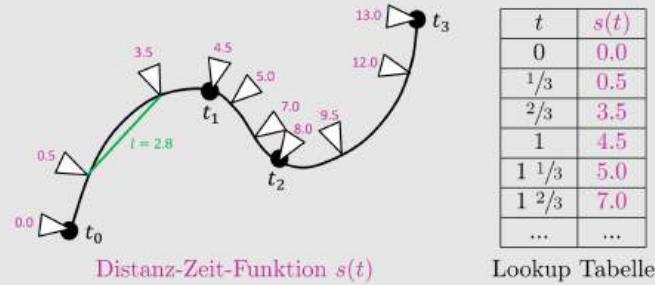


- Um dies zu erreichen, führen wir die **Distanz-Zeit-Funktion** $s(t) : \mathbb{R} \rightarrow \mathbb{R}$ ein.
- Sie gibt im Grunde an, wie weit wir entlang einer Kurve bereits gereist sind, d.h. sie ordnet jedem Zeitpunkt die bis dahin zurückgelegte Entfernung zu.
- Während $s(t)$ ursprünglich kontinuierlich 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:

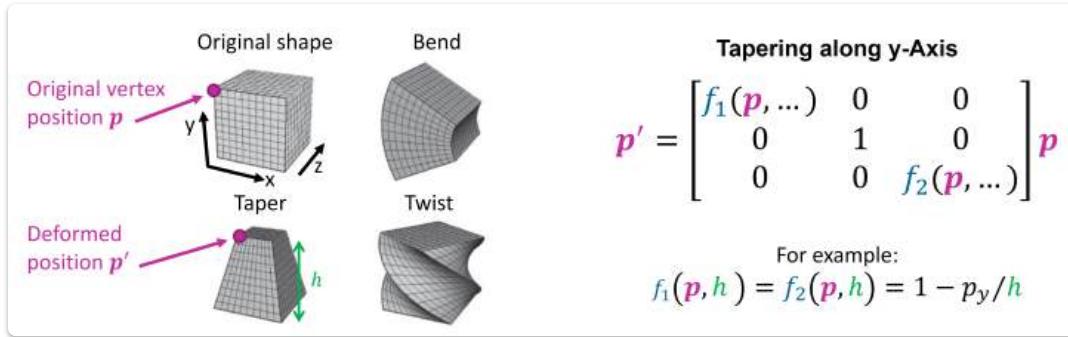
$$t = \frac{1.3 + 1.6}{2} = 2.5, \quad \text{da } 6.0 \text{ in der Mitte von } 5.0 \text{ und } 7.0 \text{ ist (siehe Tabelle).}$$

Deformationen

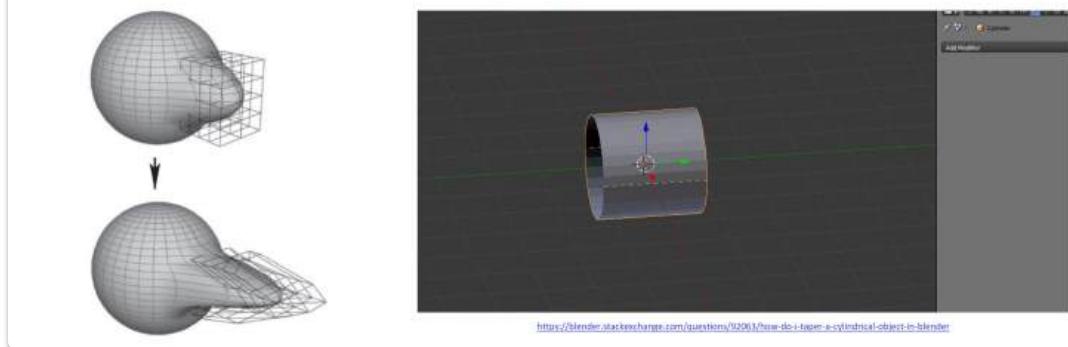
[EVC_Skriptum_CG, p.57](#)

- **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.

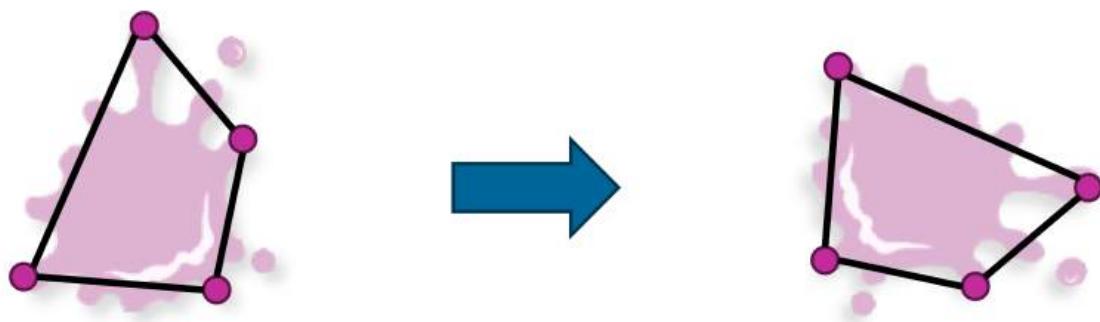
Punkt:

- **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** $m \in \mathbb{R}, [kg]$

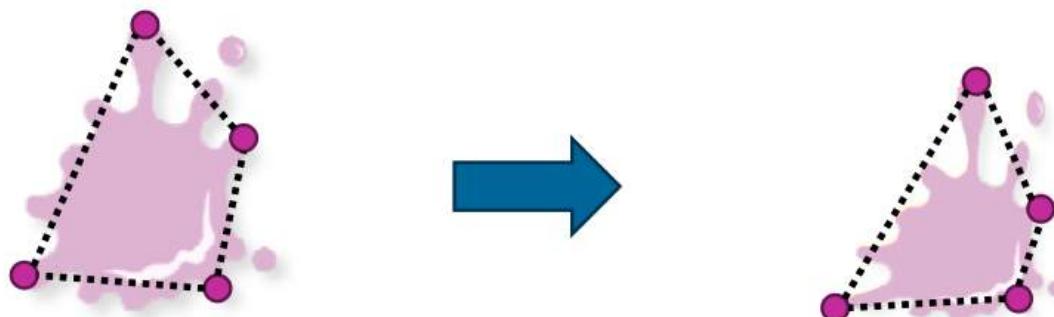
Einschränkung:

- **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



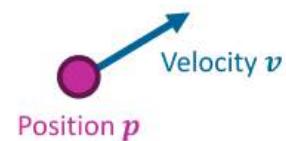
- Distance can change
- Transformations + Deformations

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



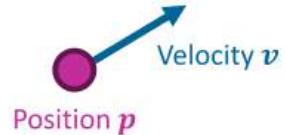
- If there is *constant* velocity, we can thus change the position via:

$$\mathbf{p}^{(t+1)} = \mathbf{p}^{(t)} + \frac{d\mathbf{p}^{(t)}}{dt} = \mathbf{p}^{(t)} + \mathbf{v}$$

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:

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \Delta t \cdot \mathbf{v}^{(t)} + \frac{1}{2}(\Delta t)^2 \mathbf{a}$$

$$\mathbf{v}^{(t+\Delta t)} = \mathbf{v}^{(t)} + \Delta t \mathbf{a}$$

$$\mathbf{a} = \frac{\mathbf{F}}{m}$$

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \int_t^{t+\Delta t} \mathbf{v}^{(t')} dt'$$

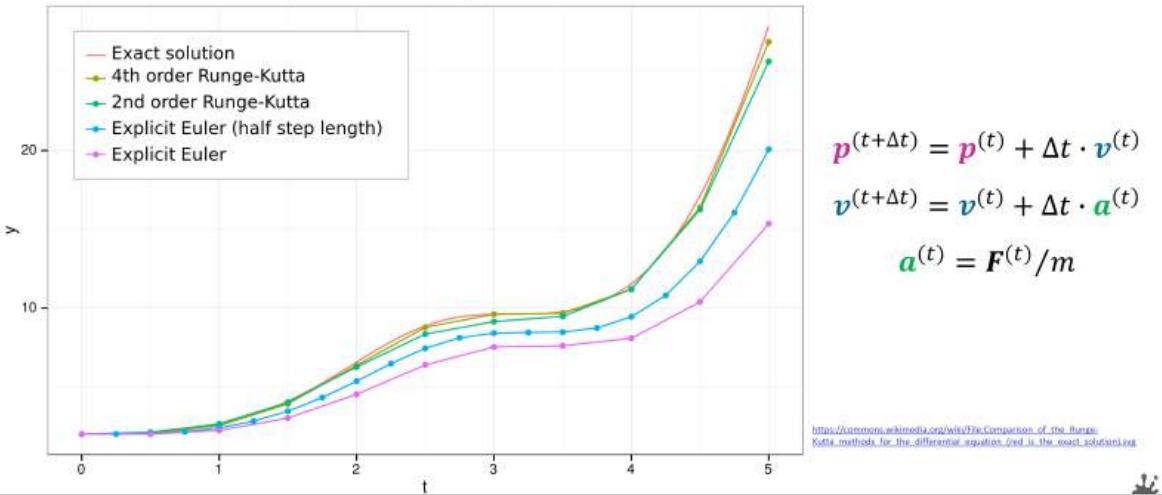
Hard to compute

⇒ 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;

```

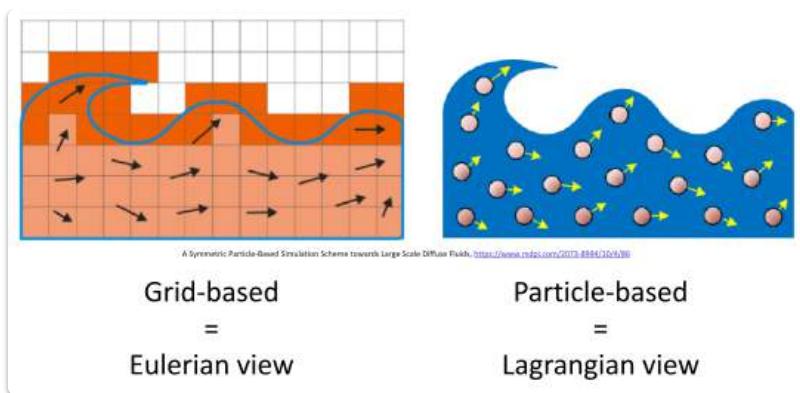
*Simple & more stable
Often used in practice*

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.

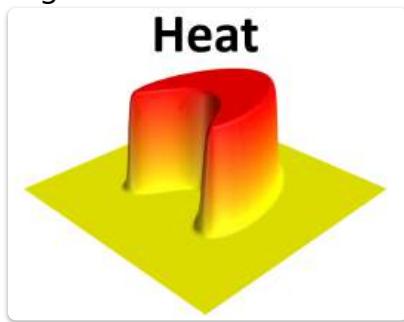
Arten von Differentialgleichungen

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).



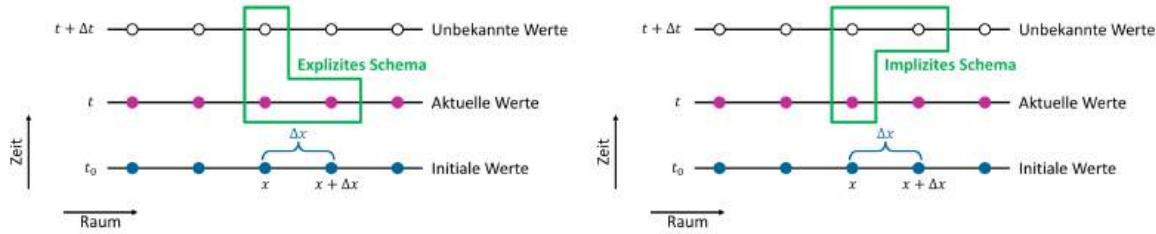
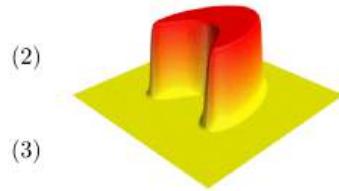
Numerische Berechnung

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

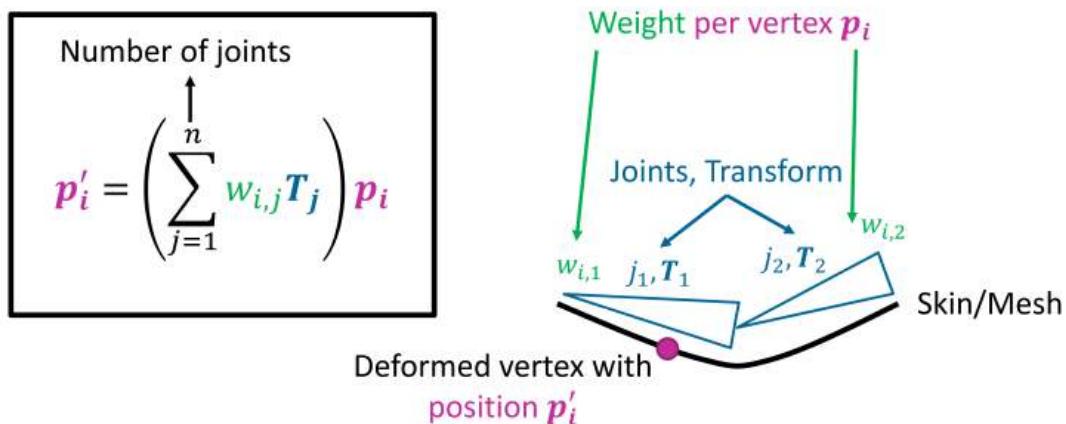
Linear Blend Skinning (LBS) ist eine einfache Methode zur Berechnung aktualisierter Vertex-Positionen basierend auf der Deformation eines zugrundeliegenden Skeletts.

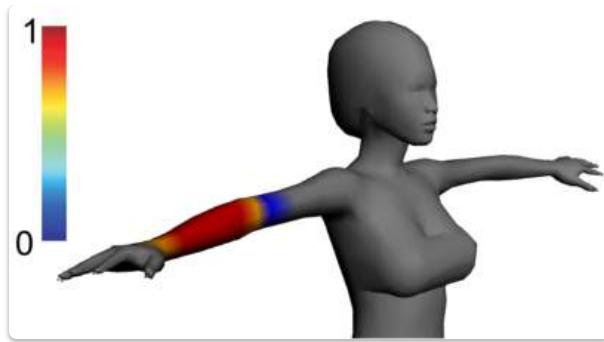
Funktionsweise von LBS

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:

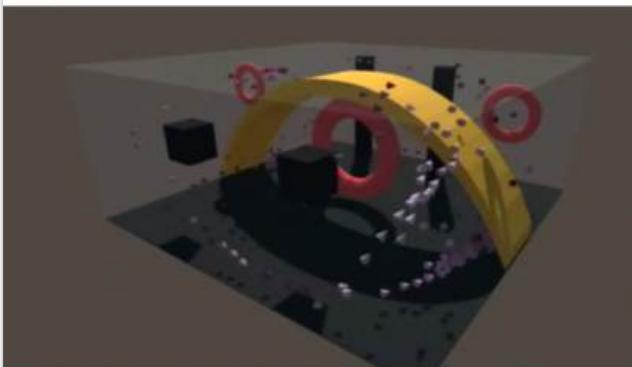
- **Motion-Capture:** Die Bewegungen realer Menschen werden direkt über Kameras und Marker auf den Schauspielern erfasst und dann auf einen virtuellen Charakter übertragen.

Prozedurale Techniken

[EVC_Skriptum_CG](#), p.58

Ähnlich wie physikbasierte Simulationen können **prozedurale Techniken** verwendet werden, um automatisch Animationen zu erstellen.

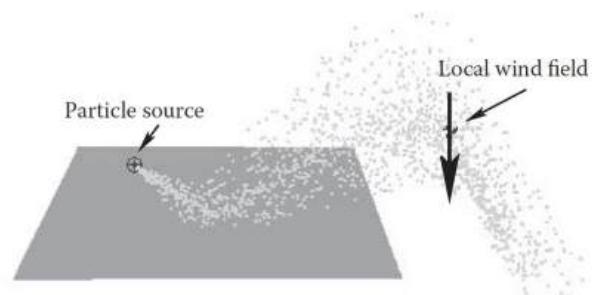
Swarm/Flock/Boids



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).

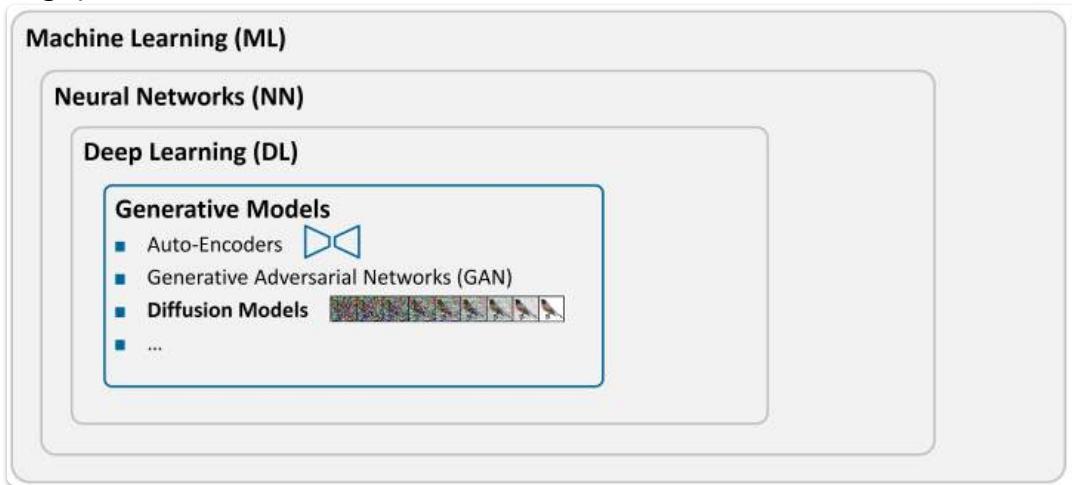
14. Machine Learning für 3D Graphics

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:

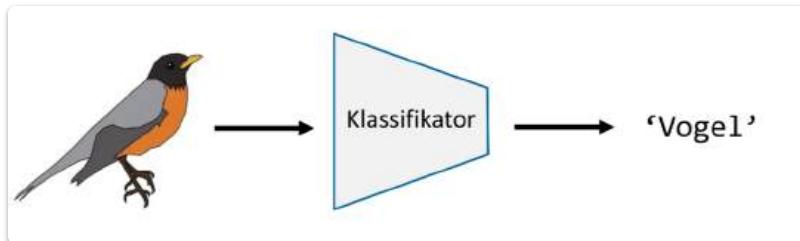
- **Auto-Encoder:**
 - Lernen, Eingabedaten in eine *kleinere Darstellung zu komprimieren*.
 - Rekonstruieren dann die Ausgabe anhand dieser Darstellung (dienen primär der Datenkompression und Merkmalsextraktion, können aber auch generativ eingesetzt werden).
- **Generative Adversarial Networks (GANs):**
 - 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:**

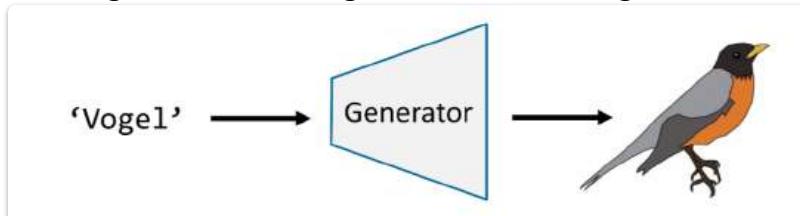
- Nutzen die Idee der *umgekehrten Diffusion* (d.h. Rauschentfernung).
- Erzeugen iterativ eine Ausgabe aus zufälligem Rauschen (beginnen mit Rauschen und entfernen es schrittweise, um ein klares Bild zu erhalten).

Konzeptionelle Vorstellung:

- **Klassifikator (rückwärts):** Ein Klassifikator würde eine hochdimensionale Eingabe (z.B. RGB-Bild) in eine niedrigdimensionale Ausgabe (eine Klassenbezeichnung wie 'Vogel') umwandeln.



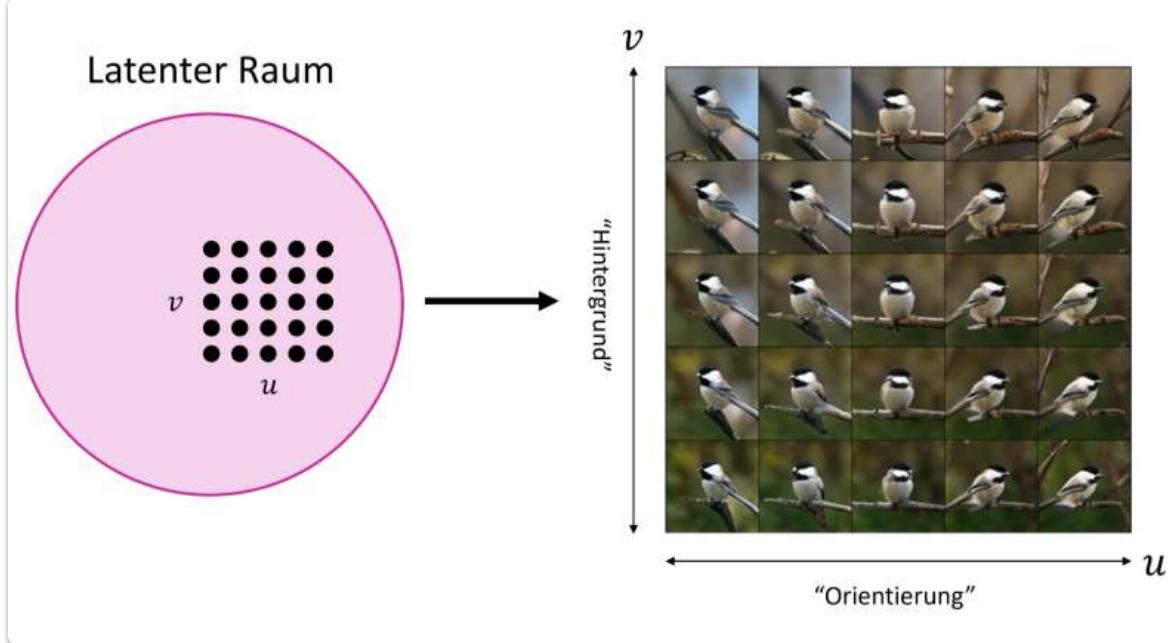
- **Generatives Modell:** Nimmt eine Eingabe mit *geringer Dimensionalität* (z.B. ein Texteingabe oder zufälliges Rauschen) und gibt ein *RGB-Bild* aus.



- 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*.

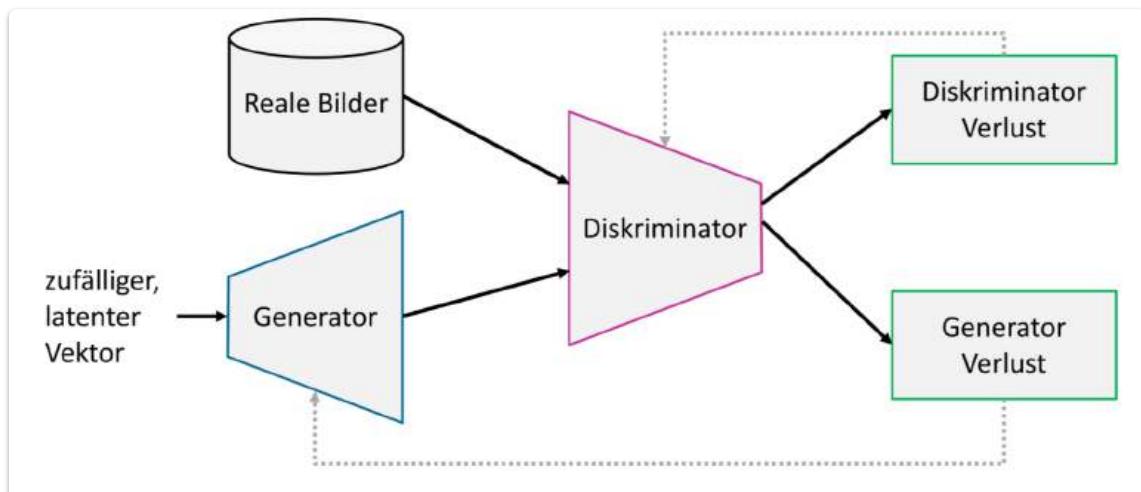
- **Inferenz:**

- 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.



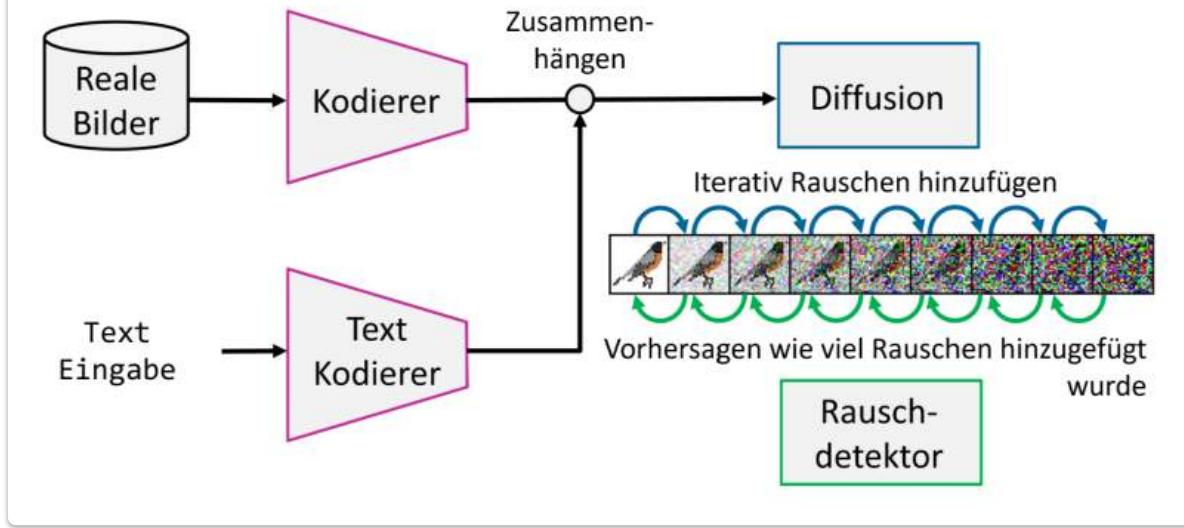
- **Reale Bilder** und **Zufälliger, latenter Vektor** sind die Eingaben.

- 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.



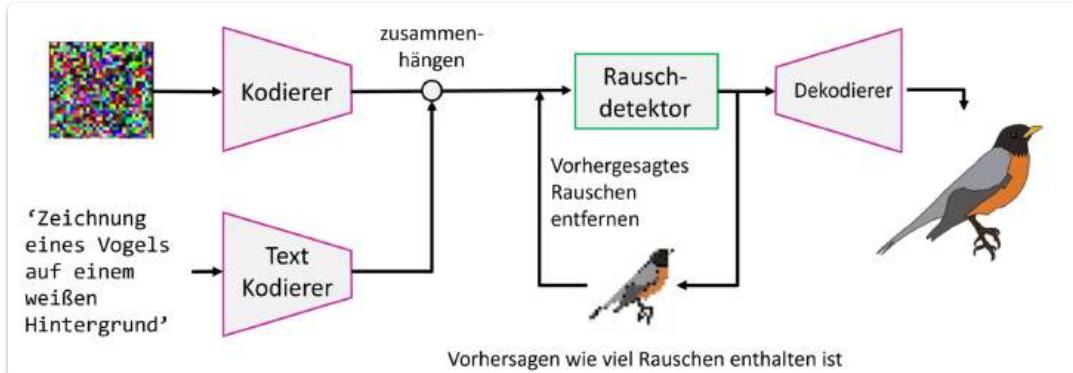
- **Reale Bilder** werden von einem **Kodierer** verarbeitet.
- **Text Eingabe** wird von einem **Text Kodierer** verarbeitet.
- Die Ausgaben beider Kodierer werden zusammengeführt ("Zusammenhängen").
- Dies fließt in die **Diffusion**, wo *iterativ Rauschen hinzugefügt* wird.
- Ein Beispielbild zeigt 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

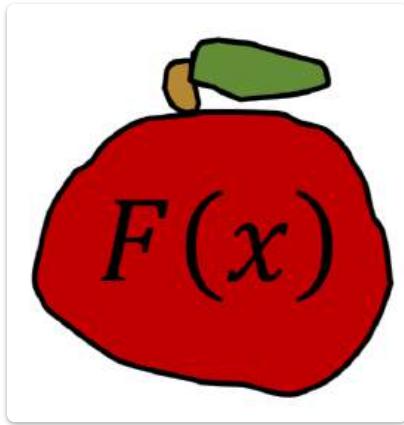
[EVC_Skriptum_CG, p.61](#)

- 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

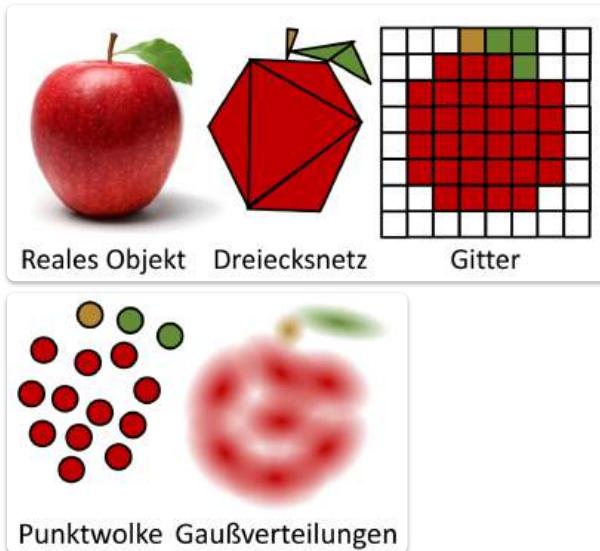
- Sind **Funktionen**, die für jeden Punkt im Raum einen Wert definieren.
- Beispiele:
 - Ob das Objekt vorhanden ist oder nicht.

- 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*.

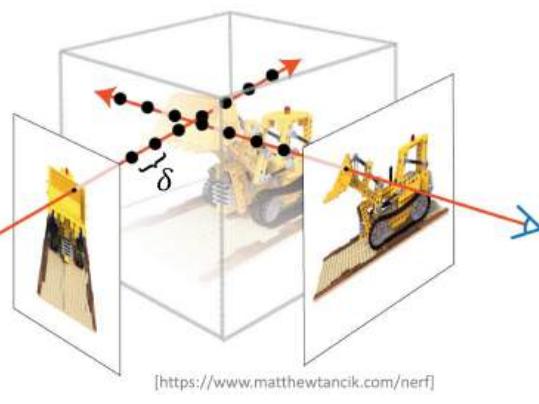
■ Each point in space contains:

- (View dependent) color – c
- Density (transparency) – σ

■ Use *raymarching* to render it

$$C(r) = \sum_{i=1}^N T_i (1 - \exp(-\sigma_i \delta_i)) c_i$$

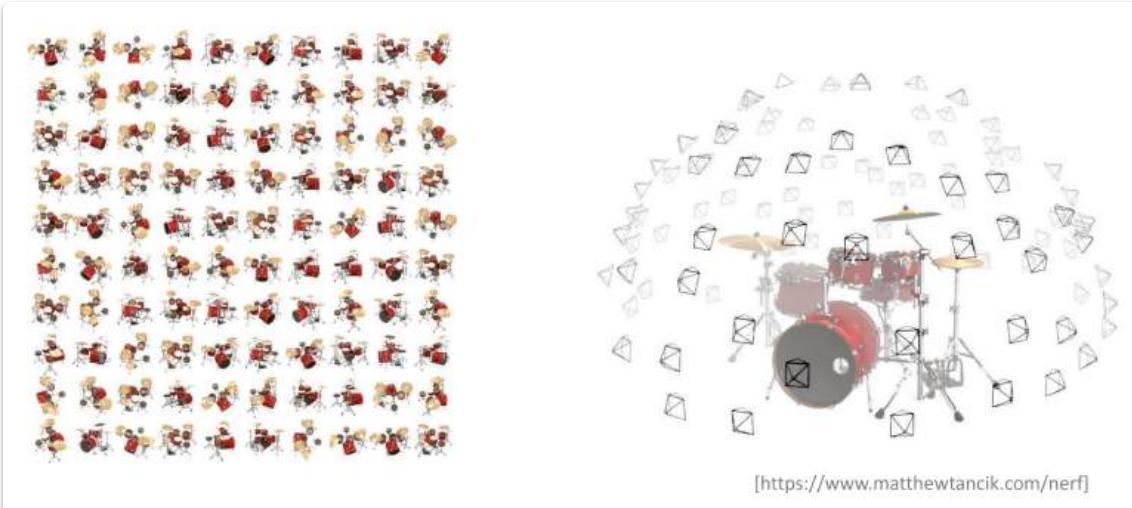
$$T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_j \delta_j\right)$$



Rekonstruktion einer realen Szene mit NeRFs

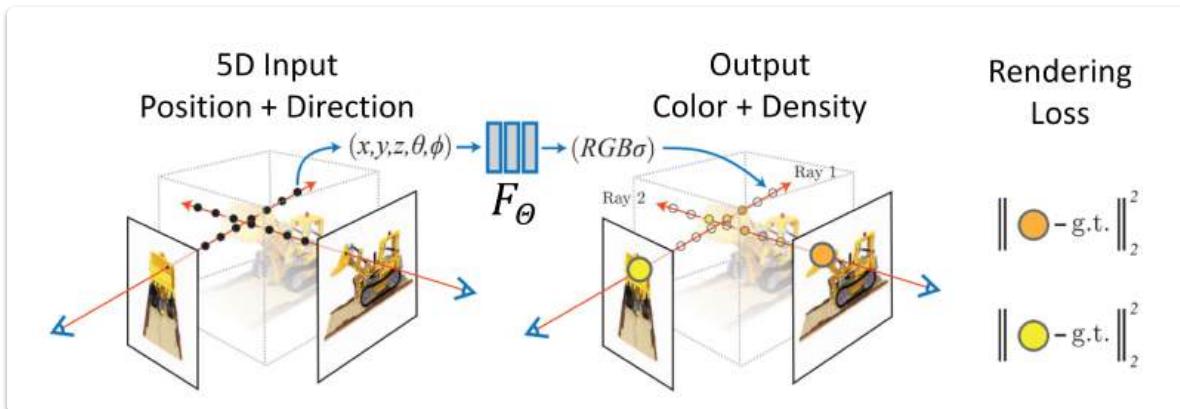
- **Basis:** Es werden *echte Fotografien* als Basis benötigt.
- **Kamerarekonstruktion:** Der *Structure-from-Motion-Algorithmus* (SfM) kann verwendet werden, um die entsprechenden Kamerapositionen zu erhalten, aus denen dann die Szene gerendert wird.

- **Optimierung:** Das gerenderte Bild wird mit dem entsprechenden Basisbild verglichen und das Netzwerk so optimiert, dass es mit diesen Bildern übereinstimmt.

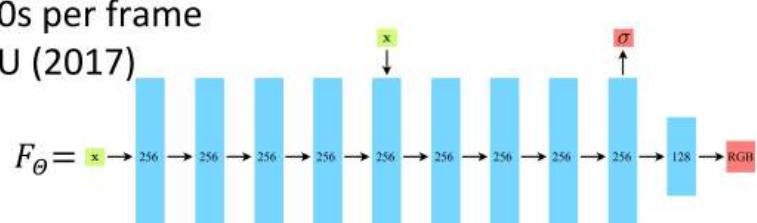


Herausforderungen und Flexibilität

- 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**.

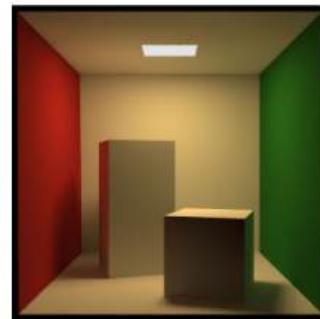


- Approx. 590 000 weights $\approx 3 \text{ MB}$
- 1-2 days to train and 30s per frame on an NVIDIA V100 GPU (2017)



- 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?

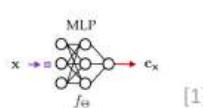


Gaussian Splatting (GS)

[EVC_Skriptum_CG, p.62](#)

Neural Radiance Fields

- Implicit



- Deep Learning



- Ray Marching



Gaussian Splatting

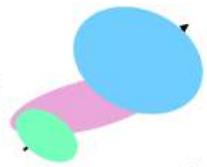
- Explicit



- Simple ML

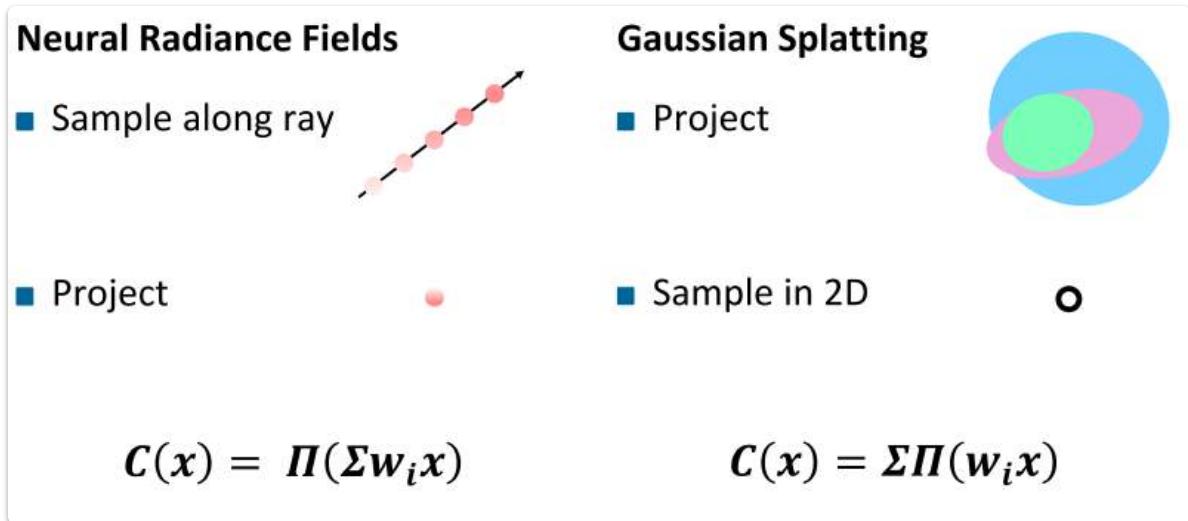


- Blending Gaussians



- 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:

$$\mathbf{X} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma}), \quad \mathbf{X} = X_1, X_2, X_3$$

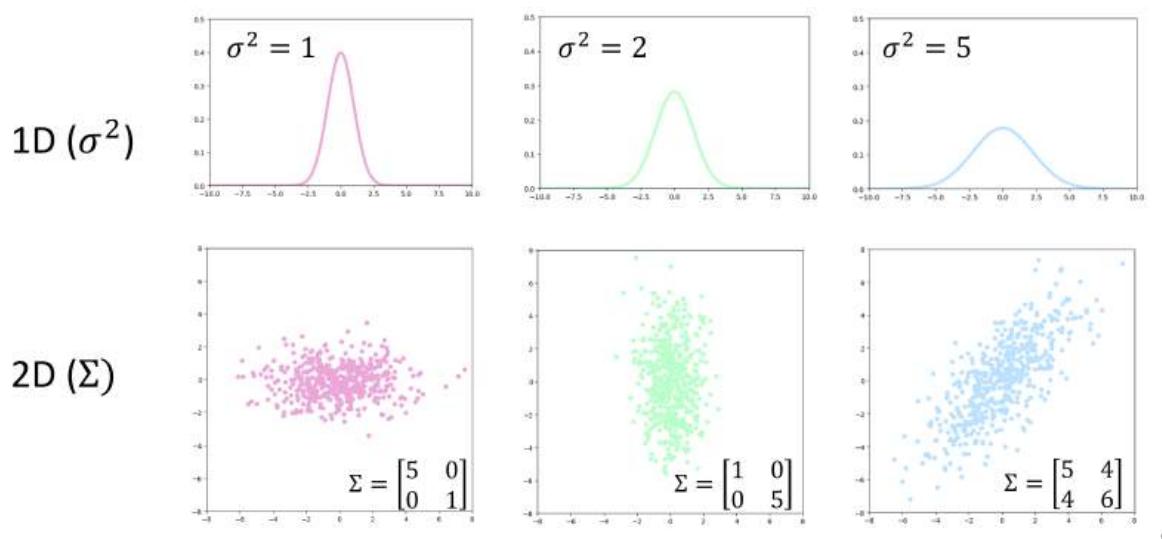
- Position \mathbf{X} , where \mathbf{X} is the vector $\boldsymbol{\mu} - \mathbf{x}$ (\mathbf{x} ... query point)
- Covariance matrix $\boldsymbol{\Sigma}$ (stretching/scaling)

$$G(\mathbf{X}) = e^{-\frac{1}{2}(\mathbf{X})^T \boldsymbol{\Sigma}^{-1} (\mathbf{X})}$$

- 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)

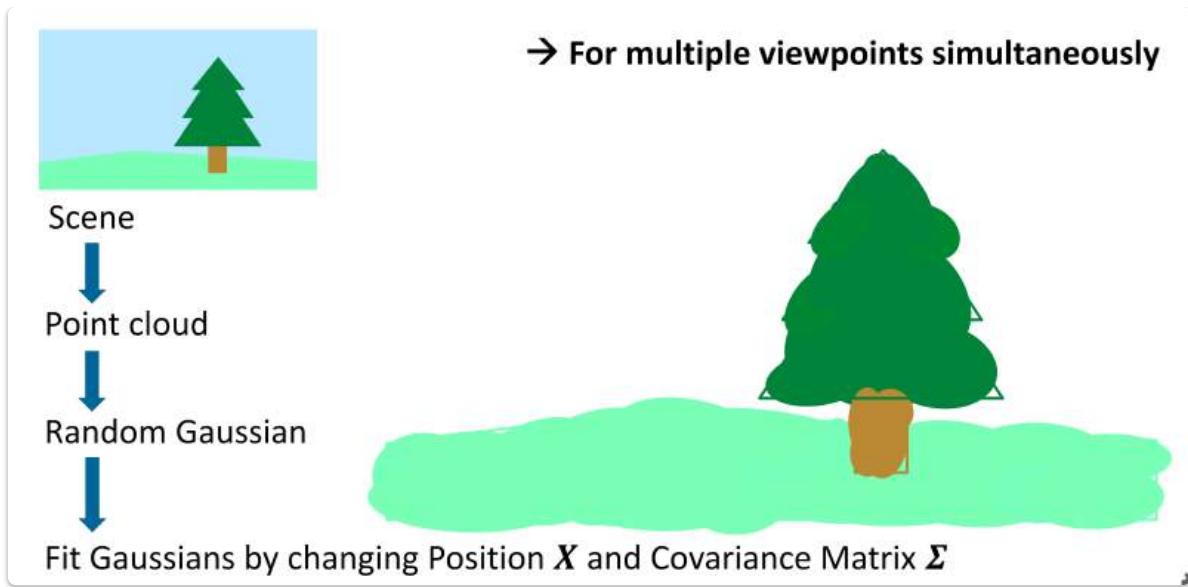


Generative Modelle für 3D Objects

[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.

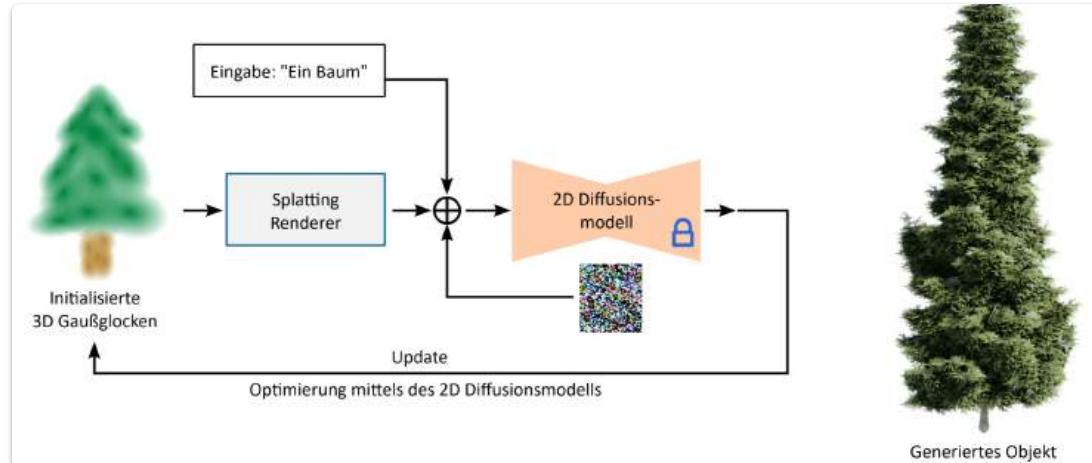
5. Verfeinerung und Detailverbesserung:

- 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:

