

# Einführung in die Spieleprogrammierung

Grafik





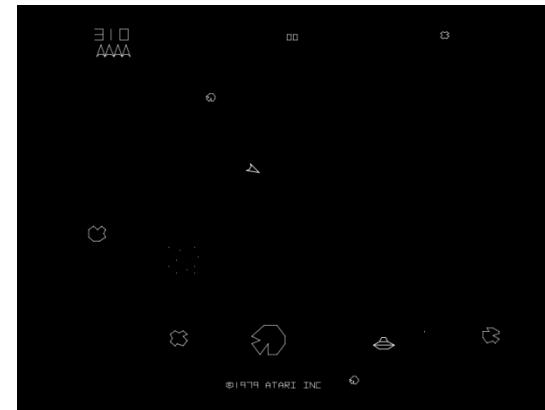
Quelle: Massuch, 2008



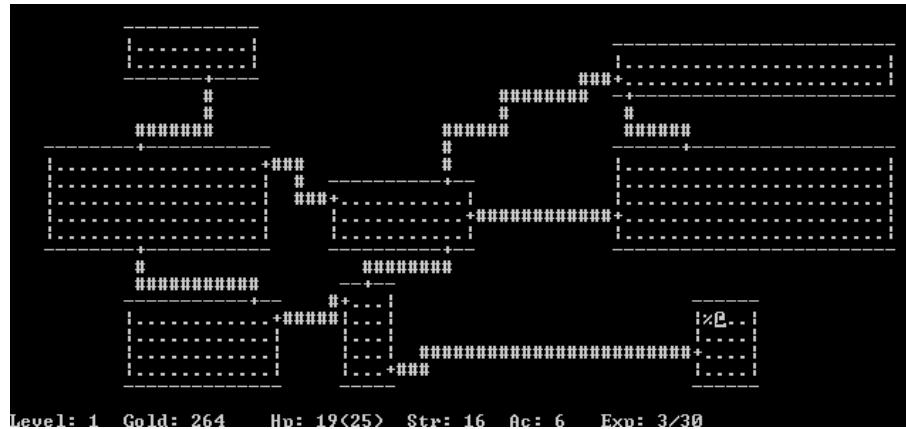
# Evolution der Spielegrafik

- Analoggrafik
- ASCII
- Sprites
- Isometriegrafik
- 2.5D / Pseudo-3D
- 3D
- Stereoskopie
- Shader

- Grafik besteht aus Primitiven wie Punkte, Linien, Kreise, Rechtecke
- Beispiele: Pong, Asteroids, Breakout



- Grafik besteht aus ASCII-Zeichen
- Beispiele: Rogue, Nethack



# Sprites

- Grafik besteht aus (mehreren) extern vorliegenden 2D-Einzelgrafiken
- Beispiele: Pacman, Die Siedler...



# Isometriegrafik

- Im Prinzip ähnlich Sprites, aber perspektivisch verzerrt dargestellt um Dreidimensionalität zu simulieren
- Beispiele: Civilization 2, X-COM, Age of Empires...



# 2.5D / Pseudo-3D

- 3D-Welt, aber Gegner und Objekte als 2D-Sprites („Billboards“)
- Beispiele: Doom, Duke Nukem 3D



- „Echte“ 3D-Umgebung, alle Spielelemente als 3D-Modelle
- Beispiele: Half-Life 2, Oblivion, Crysis 3, ...



- Anfänglich auch 3D als Kaufargument ohne tatsächliche Verbesserung der Grafik, z. B. Simon the Sorcerer 3D oder Prince of Persia 3D



# 2D in 3D

- Komplette 3D-Welt, aber Spieler bewegt sich nur in 2 Dimensionen
- Beispiele: Duke Nukem Manhattan Project, Bionic Commando Rearmed, Trine, Scoundrel



- **Egoperspektive** zeigt Spiel aus der Sicht der Spielfigur, meist nur Hände zu sehen; Beine, Füße, restlicher Körper eher selten
  - Vorteile
    - „Mittendrin-Gefühl“, hoher Grad an Immersion
  - Nachteile
    - Mangelnde Übersicht
    - Aktionen wie Klettern, Hangeln schwerer zu steuern bzw. nachzuvollziehen
    - Nötige Bildschirmanzeigen verhindern evtl. Immersion
- Manche Spiele (z.B. Oblivion) bieten mehrere Perspektiven



# Perspektive in Spielen

- **Drittpersonperspektive** zeigt die Spielfigur selbst in der Welt (und verfolgt sie meist)



- Seitenansicht



- Vogelperspektive



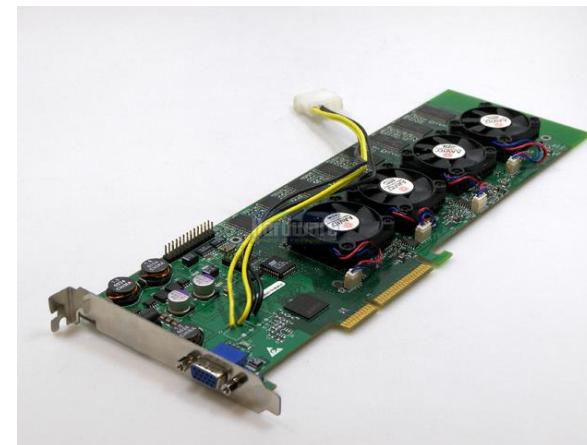
- Verfolgerperspektive



- Spezielle Blickwinkel



- *Früher:* 2D-Grafikkarte, 3D Berechnungen auf CPU
- 1996: Erste brauchbare 3D-Karte (für Spieler) 3dfx Voodoo
- Zunächst nur Addon-Karten zusätzlich zur 2D-Karte
- Im Laufe der Zeit immer weitere Übernahme von Funktionen (z.B. Physikberechnungen) um CPU zu entlasten
- Wichtiges Leistungsmerkmal neben der Speichergröße (→wichtig bei detaillierten Texturen) ist inzwischen die Anzahl der Shadereinheiten



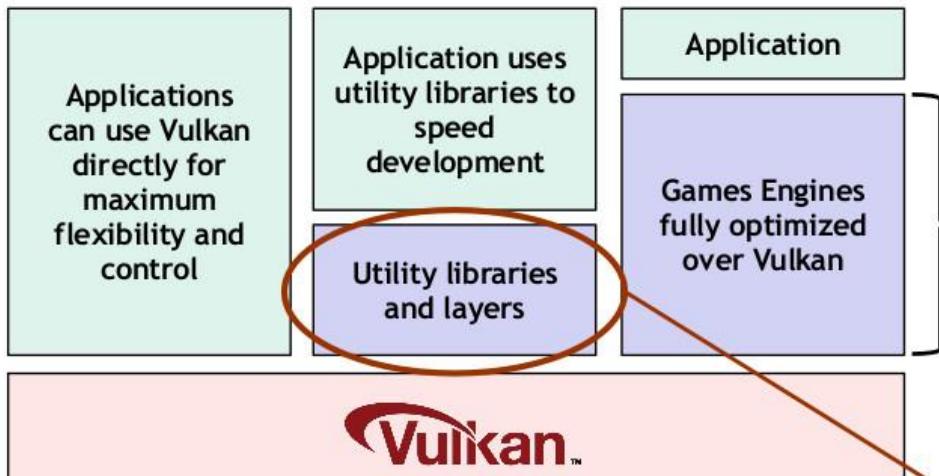
- Aktuelle Grafikkarten unterstützen zwei APIs für in Echtzeit berechnete 3D-Grafik
- Direct3D
  - Von Microsoft 1995 als Teil von DirectX veröffentlicht
  - Nur für Windows (Ausnahme Wine), proprietär
- OpenGL
  - Ursprünglich von Silicon Graphics (SGI) 1992 veröffentlicht
  - Architecture Review Board (ARB) mit verschiedenen Firmen kontrolliert Weiterentwicklung
  - Microsoft war Gründungsmitglied des ARB, zog sich aber 2003 daraus zurück
  - Seit 2006 Übernahme durch Khronos Group (>100 Firmen)
  - Plattformunabhängig, offener Standard

- Neu(2016): Vulkan
  - ähnlich zu Mantel/DirectX12/Metal
  - Standard der Khronos Group
  - Unterstützung durch viele Hersteller Geräte und Softwareschmieden, etwa wie OpenGL
  - Idee:
    - Reduzierter CPU-Overhead entlastet CPU
    - Direkter Zugriff auf Bildpuffer; Nutzung von weiteren Hardware Funktionen



- Neu: Vulkan

## THE POWER OF A THREE LAYER ECOSYSTEM



Developers can choose at which level to use the Vulkan Ecosystem

The same ecosystem dynamic as WebGL  
A widely pervasive, powerful, flexible foundation layer enables diverse middleware tools and libraries

The industry's leading games and engine vendors are participating in the Vulkan working group

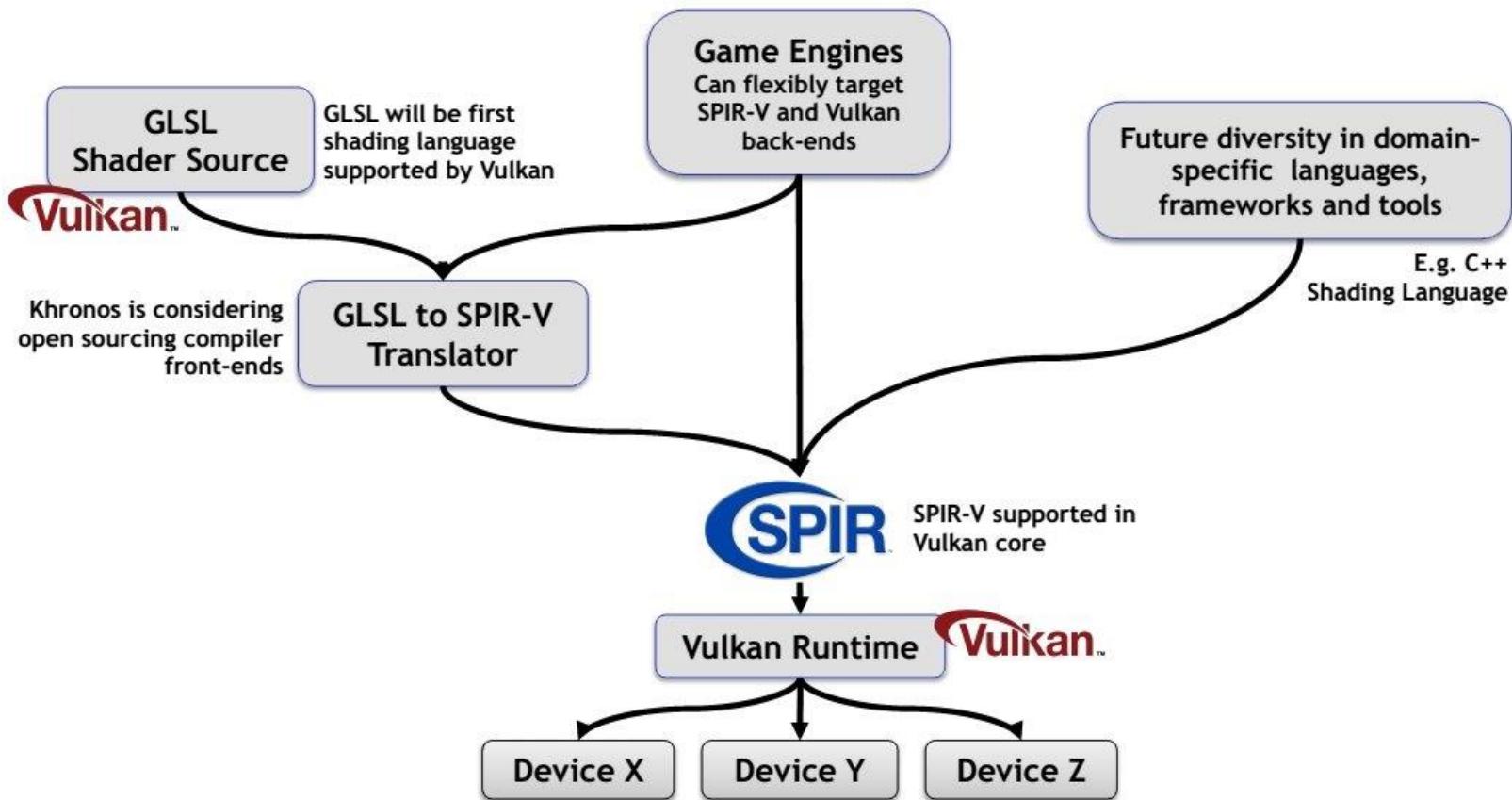


### Rich Area for Innovation

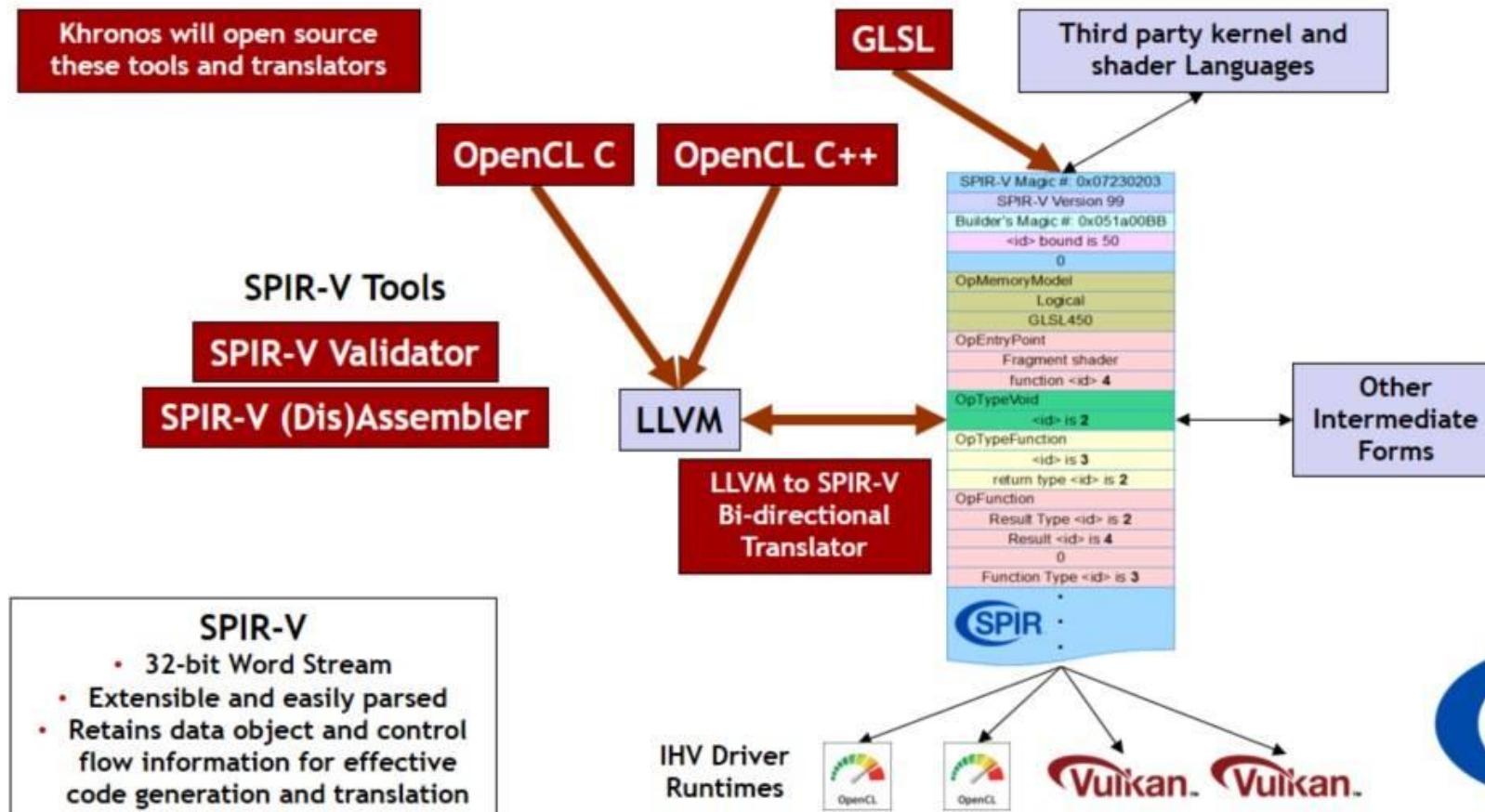
- Many utilities and layers will be in open source
  - Layers to ease transition from OpenGL
    - Domain specific flexibility

- Neu: Vulkan

## Vulkan Language Ecosystem

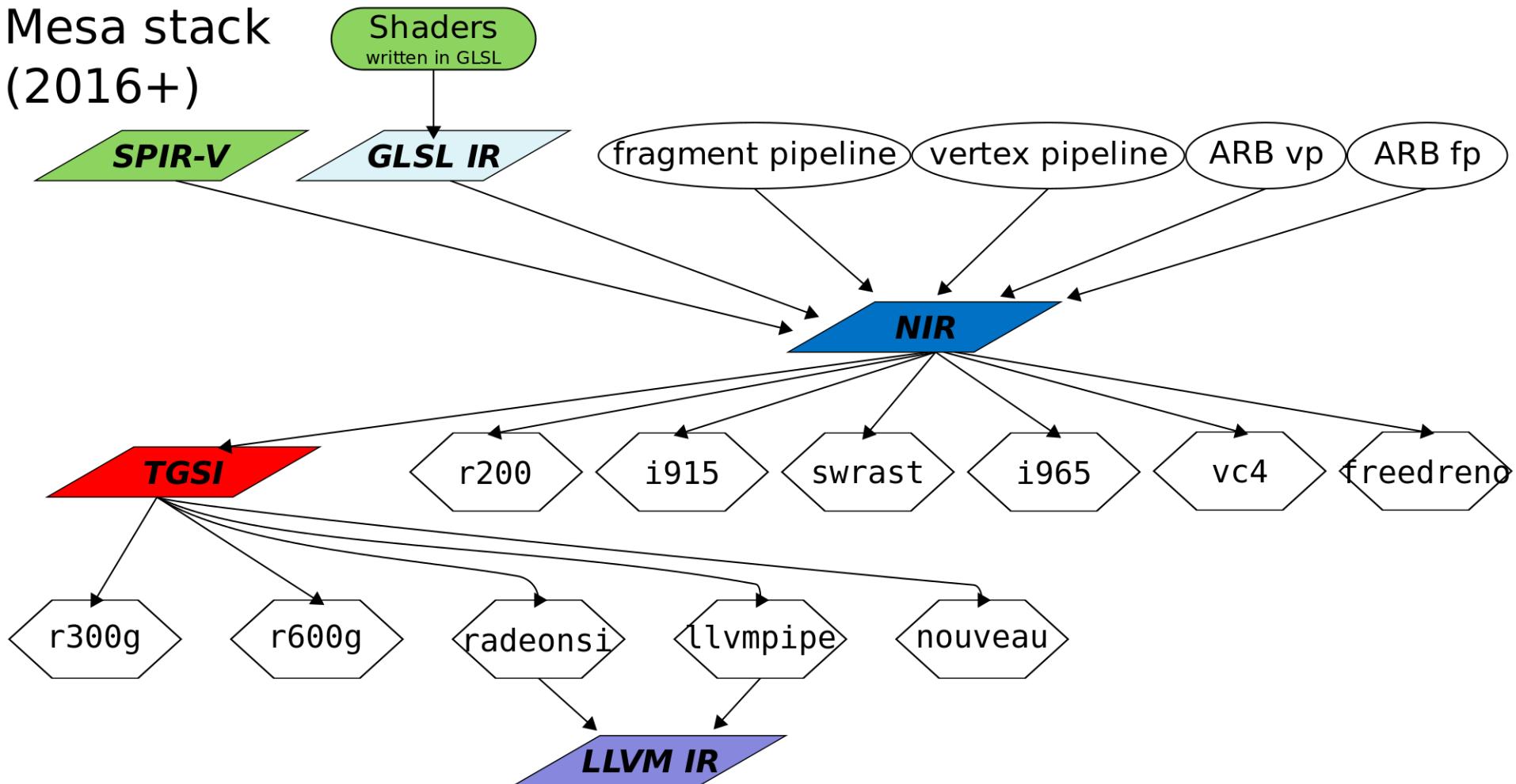


- Neu: Vulkan



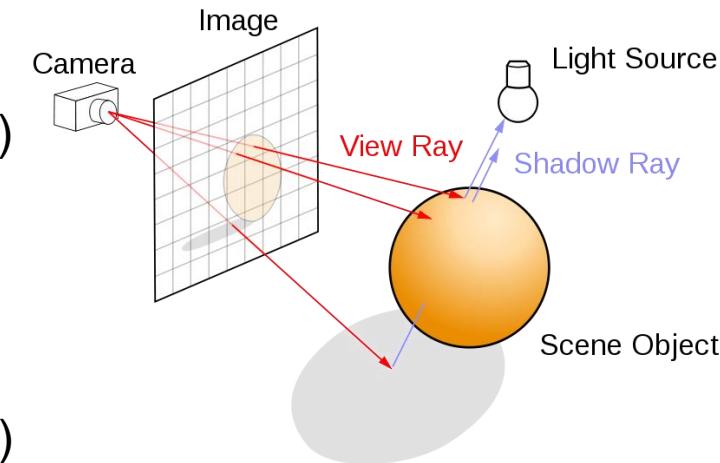
- Driver

Mesa stack  
(2016+)



# Raytracing

- Raytracing liefert fotorealistische Grafik mit exakten Schatten, Reflexionen, Lichtbrechung,  
...
- Möglich durch „Rückverfolgung“ (Tracing) der Lichtstrahlen durch die einzelnen Pixel der Bildschirmausgabe zu den Lichtquellen
- Nicht echtzeitfähig
- OpenGL/Direct3D haben im Gegensatz eine Grafikpipeline bei der Objekt für Objekt gerendert wird (Rasterizer)
- Aber: Verwendung für Zwischensequenzen
- Außerdem: OpenRT unterstützt (eingeschränkt) echtzeitfähiges Raytracing: Quake 3/4/Wars RT, Wolfenstein RT, <http://www.wolfrt.de/>
- Mit Shadern können inzwischen aber oft ähnliche Effekte erzeugt werden
- Manchmal als vereinfachte Version (vgl. SSAO)



- Kombination von zwei leicht versetzten Bildern schafft 3D-Eindruck
- Beispiel: Magic Carpet (1994)
- Auch: „Augsburg City Run“ am Lehrstuhl für Human Centered Multimedia
- Wieder populärer geworden durch neue Kino-Filme (Avatar 2009, ...) und neue (bezahlbare) Hardware (3D-Fernseher, -Bildschirme, -Beamer)
- Außerdem Voraussetzung für VR à la Oculus Rift



- Meistverbreitete Anzeigeverfahren:
  - **(Farb)Anaglyphenverfahren:** Bilder für beide Augen werden übereinander gelegt und aber in Komplementärfarben eingefärbt (z.B. Rot-Cyan). Brille mit Farbfilter „löscht“ das jeweils andere Bild
  - **Shutterbrillen:** Synchron mit der Bildwiederholfrequenz des Bildschirms wird abwechselnd das linke und rechte Auge „zugehalten“
  - **Polarisationsbrillen:** Bildschirm strahlt abwechselnd Bilder für beide Augen aus, diese sind aber unterschiedlich polarisiert (im Kino meist zirkular, 3D Fernseher oft linear). Brille lässt nur richtig polarisiertes Licht durch.

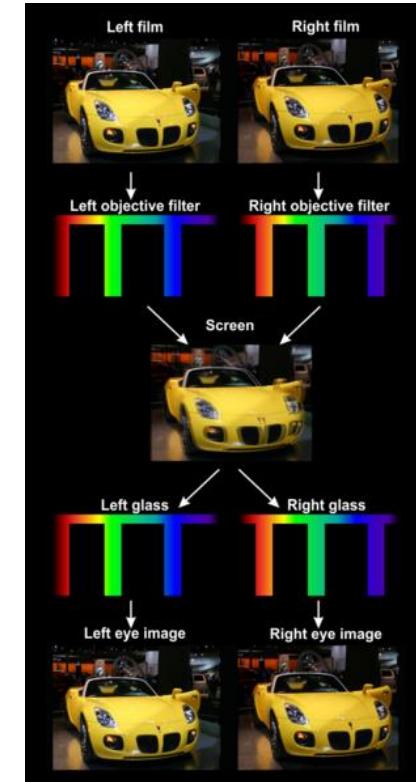


- Meistverbreitete Anzeigeverfahren:
  - **Interferenzbrillen:** Ähnlich zu Polarisationsbrillen, aber das Lichtspektrum der Grundfarben wird mit Interferenzfilter auf beide Augen aufgeteilt. Farbkorrektur gleicht das Abweichende Farbspektrum für links/rechts aus. Einsatz in manchen Kinos (Dolby3D)

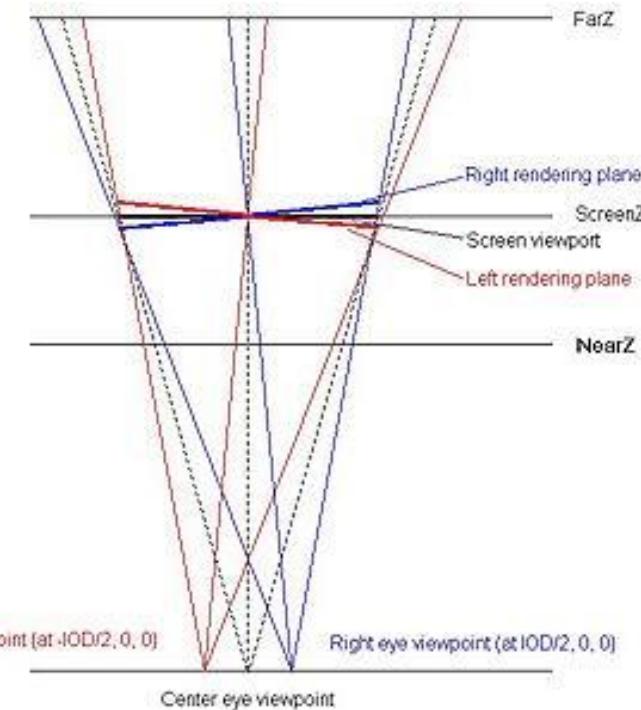


**Head Mounted Displays:** Zwei Bildschirme sind direkt vor den Augen des Benutzers und können die zwei verschiedenen Bilder anzeigen → Oculus Rift

- **Autostereokopie:** Ein Display stellt beide Bilder dar, aber diese sind ähnlich wie bei „Wechselbildern“ gerichtet. In Zukunft soll die Kopfposition (auch von mehreren) Zuschauern verfolgt werden um die Ausrichtung anzupassen.



- In Spielen:
  - Nvidia 3D Vision unterstützt praktische alle DirectX-Spiele. Berechnung erfolgt im Treiber. Nutzt Shutterbrillen mit Synchronisation über eine Infrarot-LED. Spieleprogrammierer hat eigentlich nichts zu tun.
  - **„Toed-in“-Stereo**  
Naiver Ansatz verschiebt und dreht die Kamera für die beiden Augen → Problem: Rendering Plane ist gedreht, wirkt sich hauptsächlich am Bildschirmrand aus



- **Besser: Asymmetrisches Frustum**

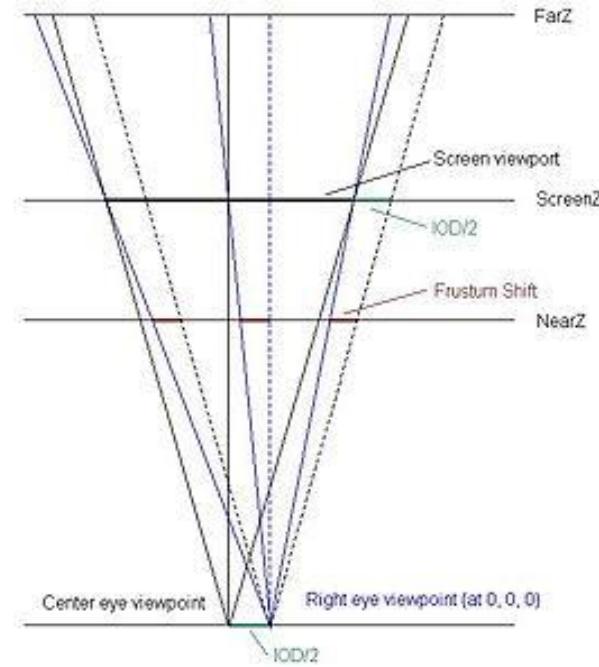
Kamera wird wieder verschoben,  
aber nicht mehr gedreht.

Stattdessen wird das Frustum  
asymmetrisch verschoben

- Übertragung an den Bildschirm:

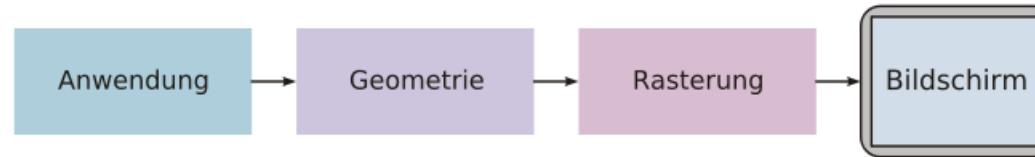
- **Quadbuffering:** linkes  
und rechtes Bild nacheinander,  
separate Bildpuffer

- **Side-by-Side** bzw. Over-and-Under:  
Bilder nebeneinander

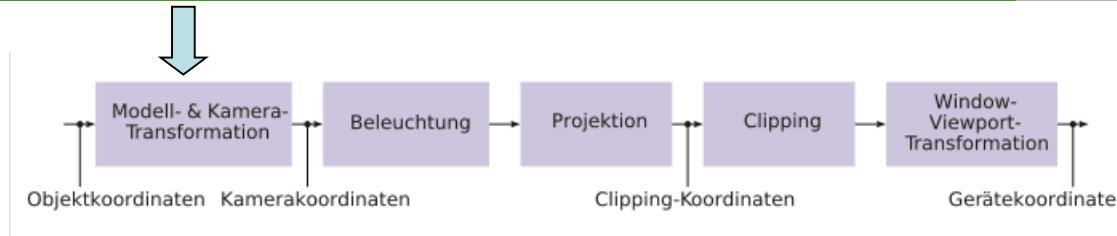


- Geometrie besteht aus Punkten (*Vertices*) im 3D-Raum

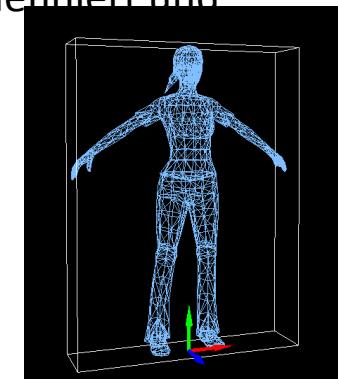
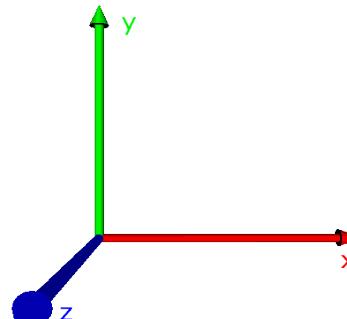


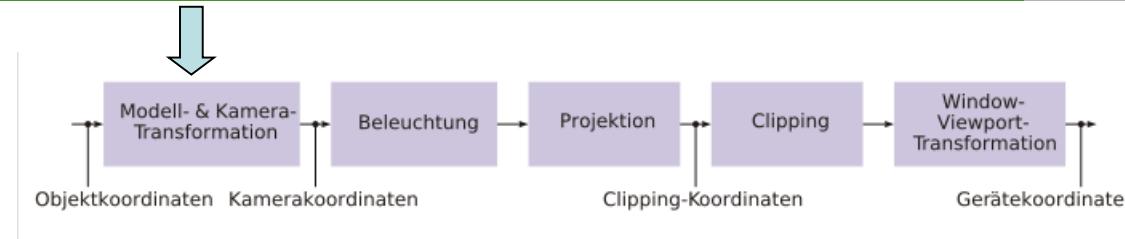


- Anwendung berechnet aktuellen Zustand der Szene basierend auf:
  - Spielereingaben
  - KI „Eingaben“ bzw. Skripte
  - Animationen
  - Physik und Kollisionserkennung
  - ...
- Geometrie berechnet 3D-Szene, projiziert diese in 2D
- Rasterung zeichnet das 2D-Bild in bestimmter Auflösung auf den Bildschirm



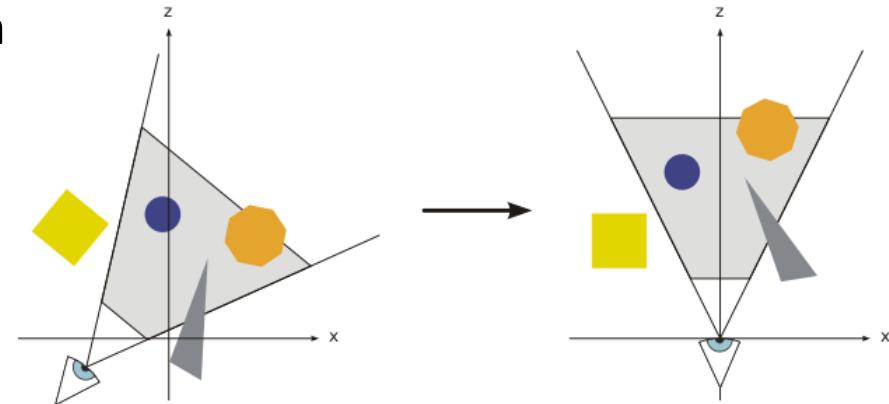
- 3D-Szene definiert ein **globales Koordinatensystem** = Weltkoordinatensystem
- Die Punkte (=Vertices) eines Modells bestimmen dessen Form und sind im **lokalen Koordinatensystem** des Modells (=**Objektkoordinatensystem**)
- Modelle sind im Weltkoordinatensystem **transformiert** um 3D-Szene zusammenzustellen
- Modeltransformation wird auf die Punkte des Modells angewandt (= **model transform**) → Punkte „sind“ nun im globalen Koordinatensystem
- Dann folgt Kameratransformation (= **view transform**) → **Kamerakoordinaten**
- Transformationen sind üblicherweise als 4x4 Matrizen definiert und enthalten: **Skalierung** → **Rotation** → **Translation**

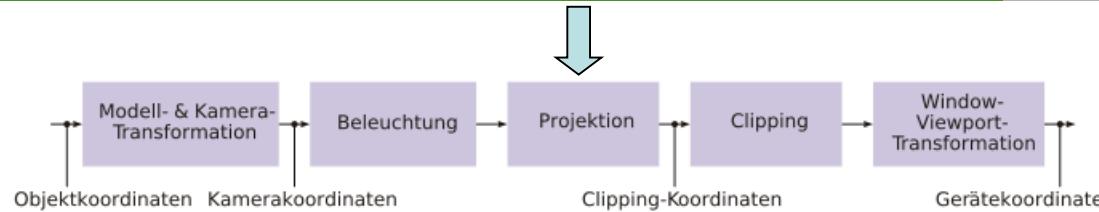




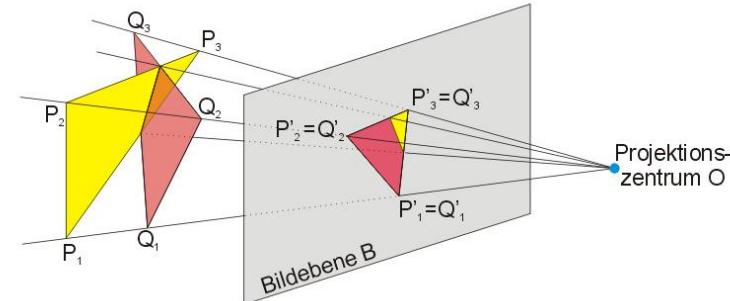
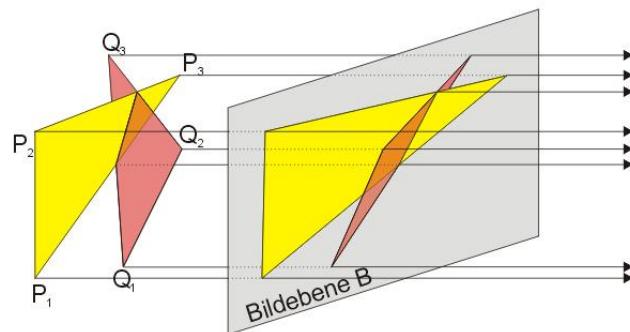
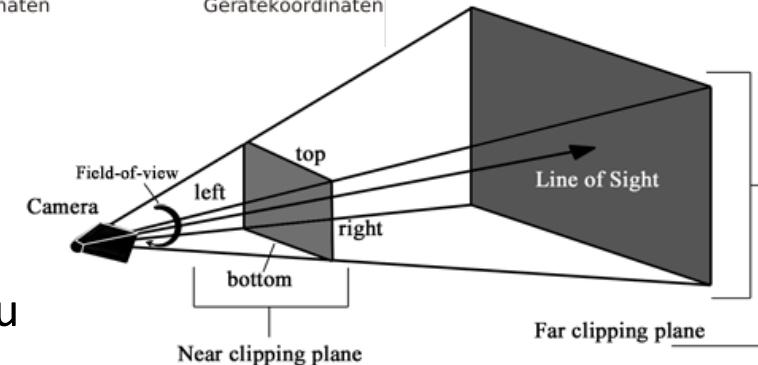
Nach der Modelltransformation folgt die **Kameratransformation**:

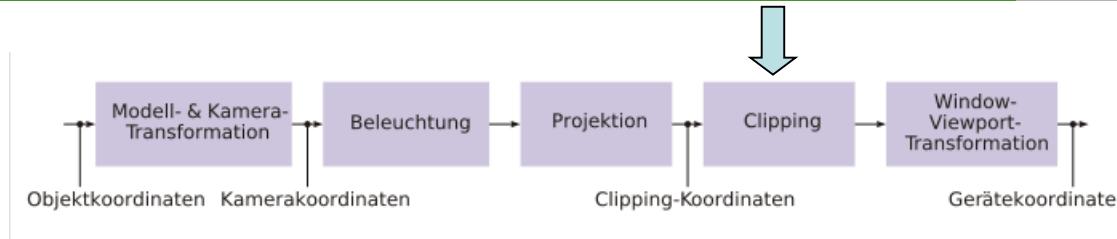
- Szene enthält neben Modellen auch mindestens eine Kamera
  - „Aktive“ Kamera wird bestimmt
  - Szene und Kamera werden so transformiert, dass Kamera sich im Ursprung befindet und in Richtung (negative) Z-Achse blickt  
= Anwendung der inversen Kameratransformation
- Objekte in **Kamerakoordinaten**



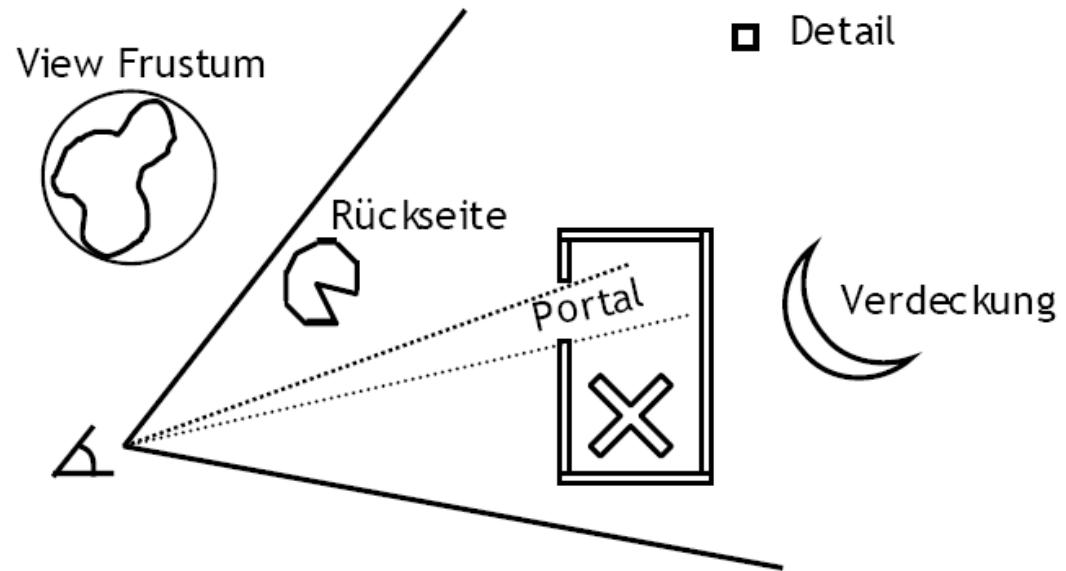


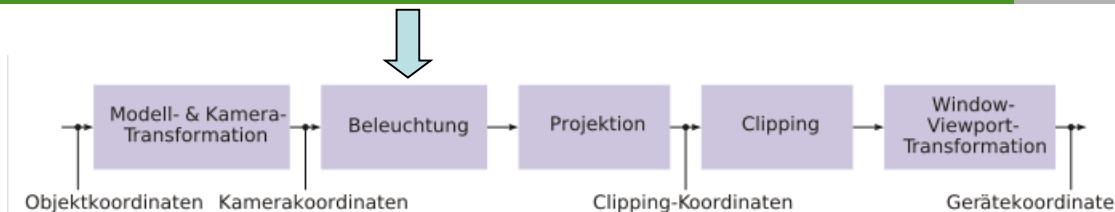
- Viewing Volume oder auch Viewing Frustum (Kegelstumpf) wird bestimmt
- Viewing Volume enthält alle Objekte, die möglicherweise (Clipping) dargestellt werden
- Projektion transformiert das Viewing Volume zu einem Einheitswürfel → Clipping-Koordinaten
  - I.d.R. Zentralprojektion
  - Spezialfall: Parallel- bzw. Orthogonalprojektion



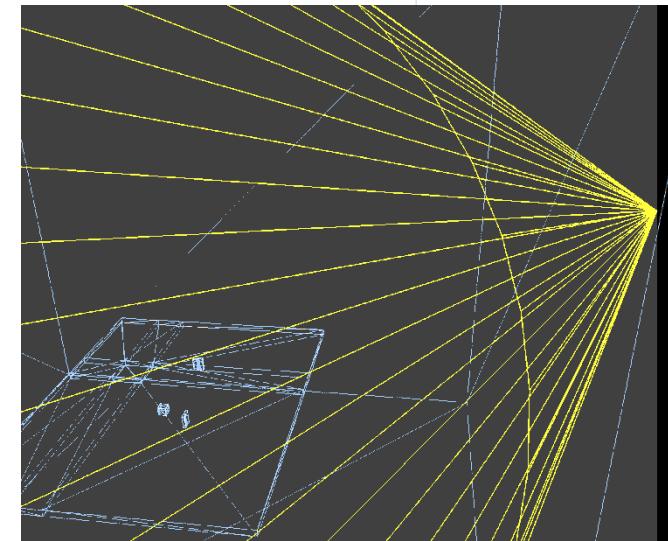
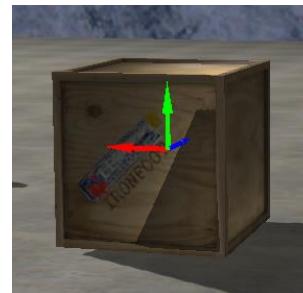
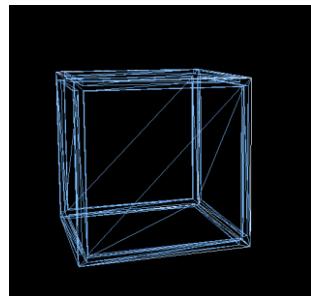


- Clipping bzw. Culling: Bestimmung nicht-sichtbarer Dreiecke der Szene  
Diese werden bei weiteren Berechnungen übersprungen, um Ressourcen zu sparen
- Vorsicht bei Transparenz!
- (View) Frustum Culling
- Backface Culling
- Occlusion Culling
- Detail Culling
- Portal Culling



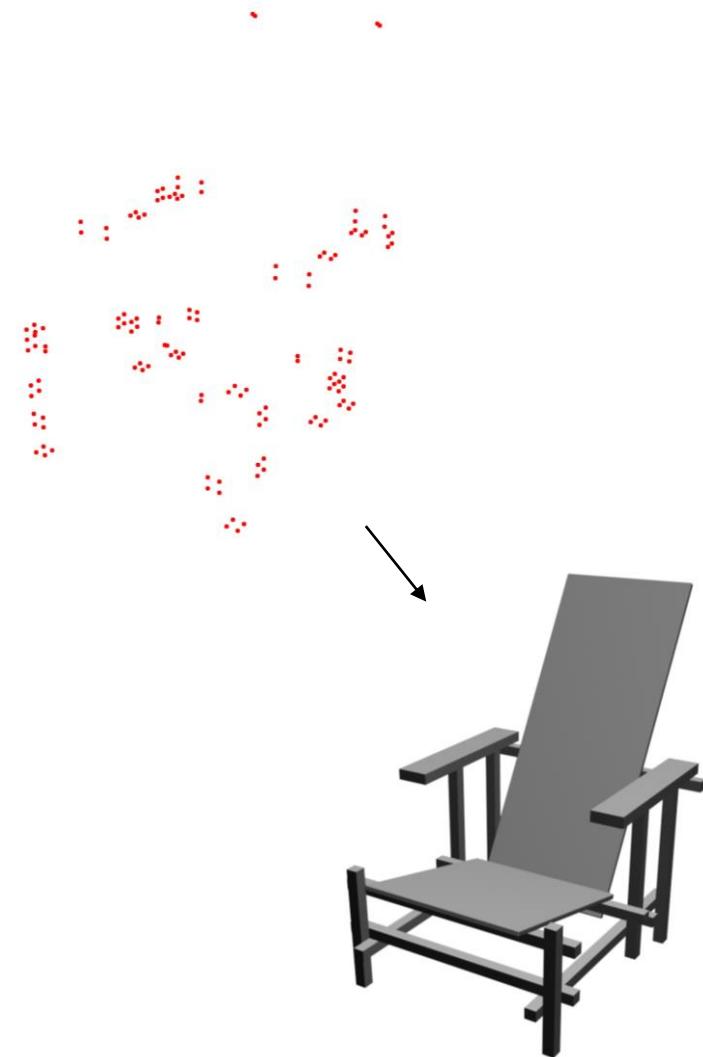
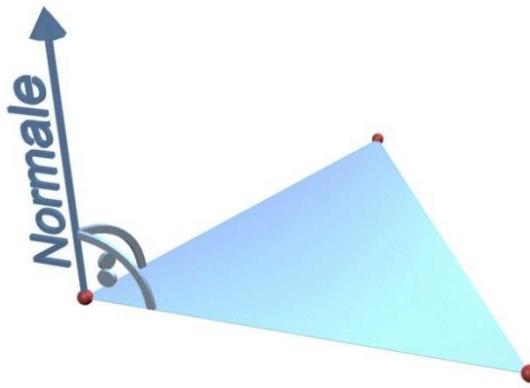


- Szene enthält Punktlichtquellen (Position, Farbe/Textur, Radius, Öffnungswinkel)
- Modelle sind auf Dreiecksbasis mit Texturen überzogen



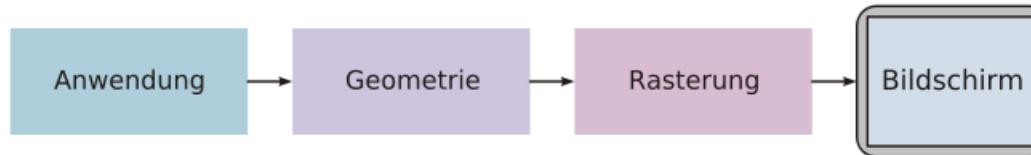
- Anhand der Textur und dem Einfluss der einzelnen Lichtquellen wird für jeden Pixel seine Farbe bestimmt = Shading
- Die Beleuchtung geschieht meist in mehreren Stufen unter Zuhilfenahme verschiedener Puffer, z. B. zur Schattenberechnung

- Vertices bilden Dreiecke
- Normalen bestimmen die Beleuchtung

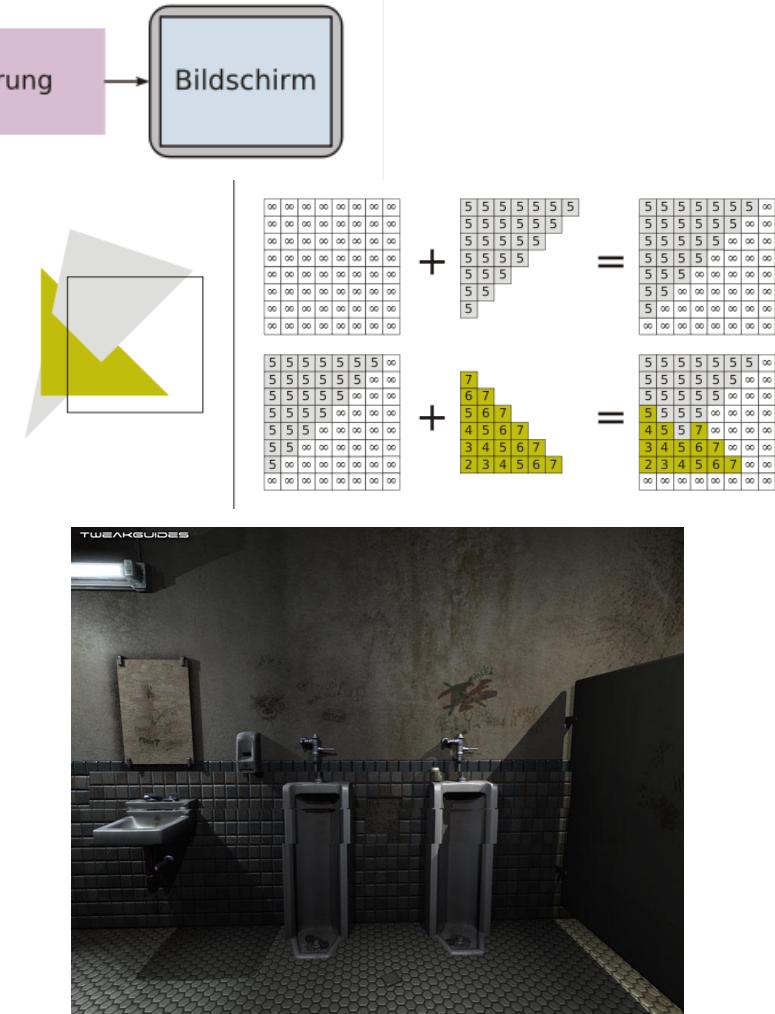


- Dreiecke werden mit Texturen überzogen
- Mapping von Texturkoordinaten auf die Dreiecksflächen

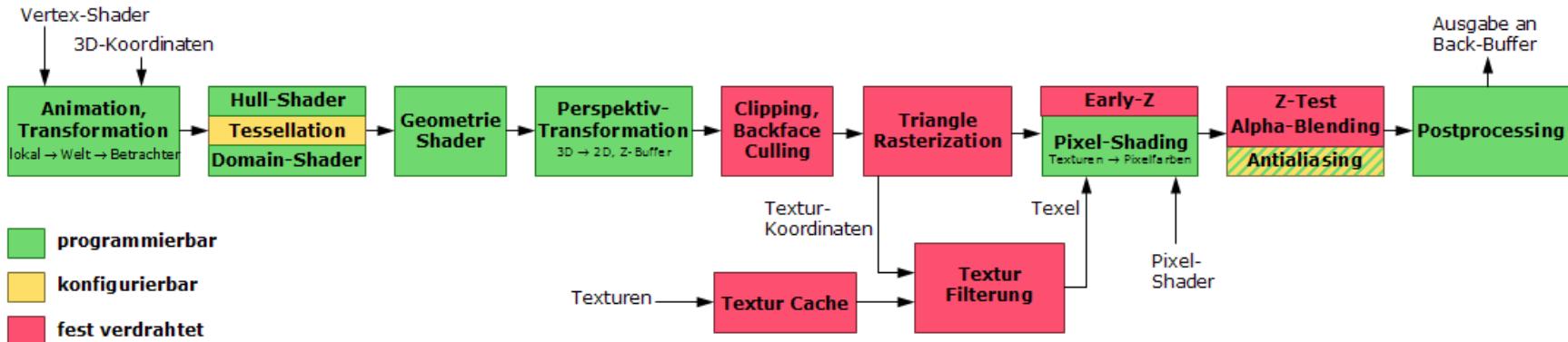


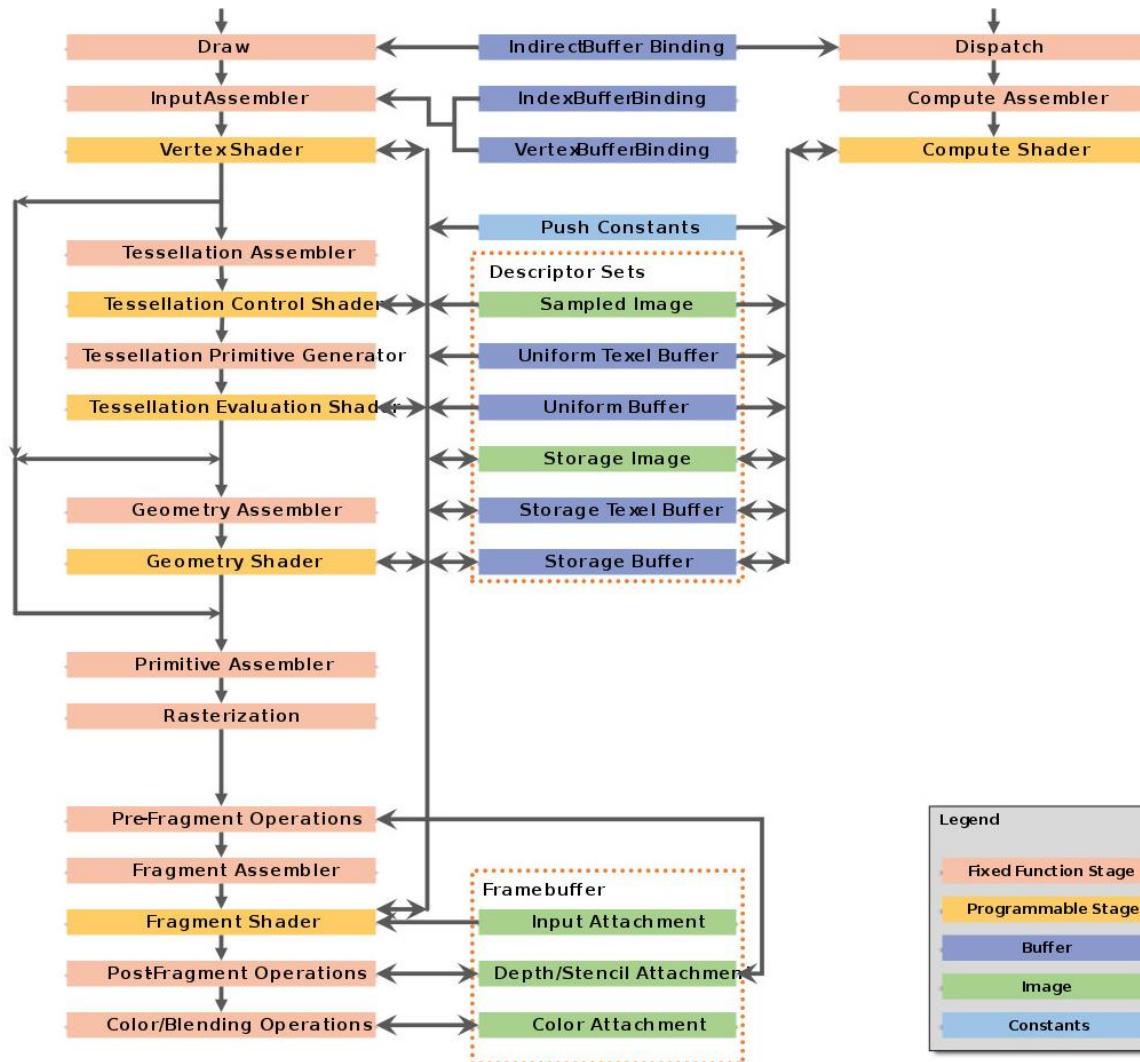


- Zeichnet das in Gerätekordinaten vorliegende Bild auf den Bildschirm, indem alle Primitiven pixelweise gerastert werden (**Pixel Shading**)
- Überlappen oder schneiden sich mehrere Primitive, so muss bestimmt werden, welches der Kamera näher ist und somit gezeichnet werden soll → **Z-Buffering**
- Monitor kann unterschiedliche Bildwiederholfrequenz haben → **Vertical Synchronisation (VSync)** um Tearing zu vermeiden → **Triple/Multiple Buffering** um Anwendung nicht auszubremsen
- Je nach Grobheit des Pixelrasters können sog. Treppeneffekte auftreten → **Anti-Aliasing (Kantenglättung)**



- *Hinweis:* Die vorgestellten Schritte einer Grafikpipeline waren ursprünglich fest verdrahtet, inzwischen sind sie aber deutlich flexibler
- Aktuelles Engines nutzen meist **deferred lighting**, hierbei wird die eigentliche Beleuchtung soweit wie möglich nach hinten verzögert; Zwischenschritte werden in separate Puffer gespeichert, die dann für alle Lichtquellen verwendet werden können, anstatt sie pro Lichtquelle neu zu berechnen → Viel mehr Lichtquellen möglich
- Außerdem sind Pipelines inzwischen **shader-gesteuert**
- Allgemeineres Schema für eine aktuelle Pipeline (DirectX 11-Terminologie):

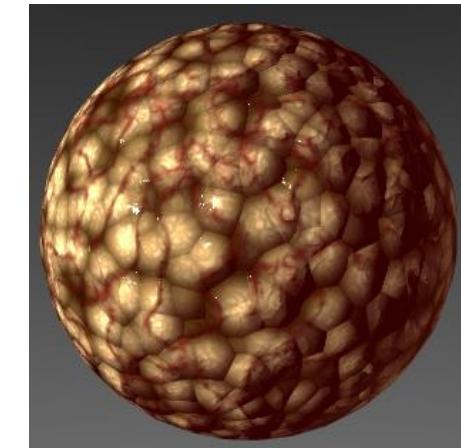
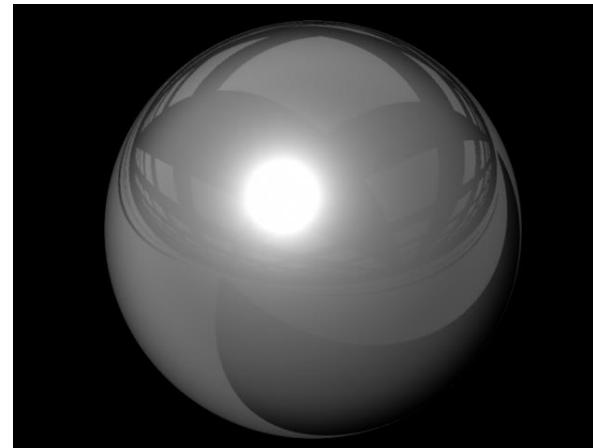
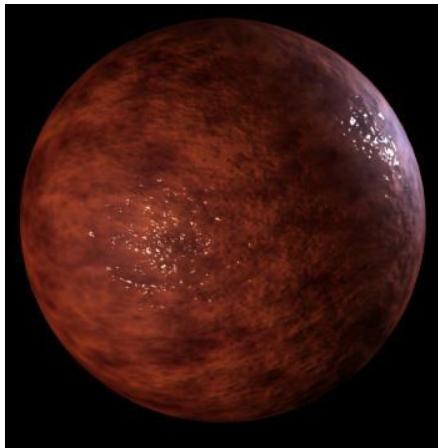




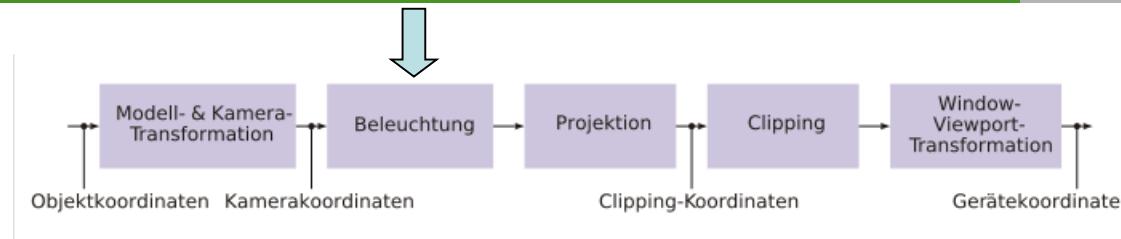
Legend

- Fixed Function Stage
- Programmable Stage
- Buffer
- Image
- Constants

- Berechnung der Wirkung von Licht und Schatten auf verschiedene Materialien, aber auch Postprocessing-Effekte
- Stark parallelisierte Berechnungen direkt auf der GPU, Entlastung der CPU
- Eigene Shader-Sprachen, z. B. HLSL (DirectX), GLSL (OpenGL)
- Beispiele besonderer Shader-Effekte: Wet Shader, Chrome Shader, Organic Shader

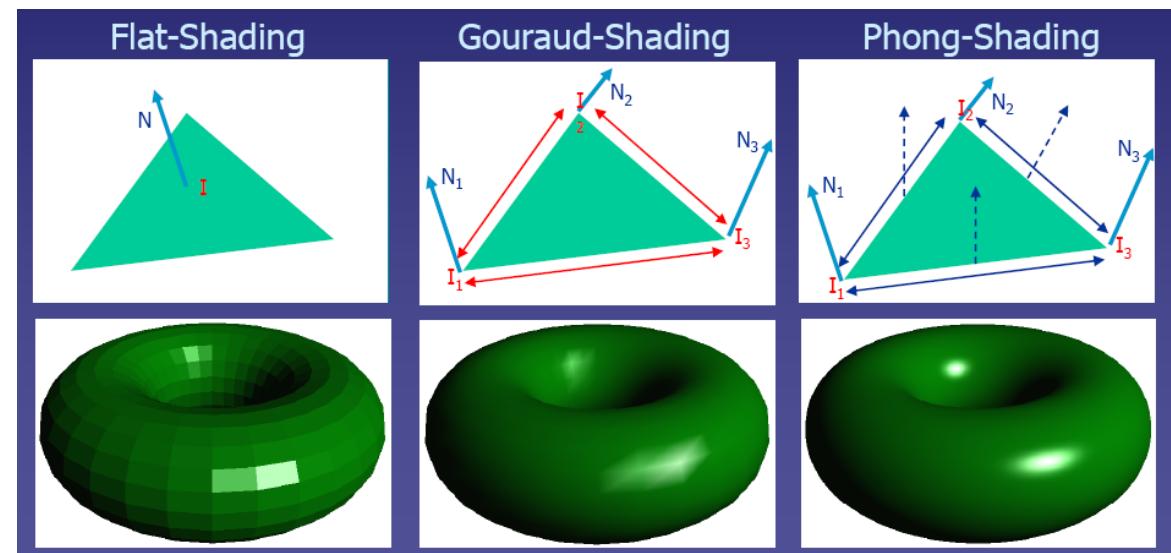


- **Vertex-Shader** transformiert Objektgeometrie (die „Vertices“), z. B. für Wasserwellen
- **Pixel- bzw. Fragment-Shader** für Änderungen je Pixel im Gesamtbild, z. B. für Spiegelungen oder Bloom-Effekte
- Seit DirectX 10 bzw. OpenGL 3.2: **Geometry-Shader** zur Erzeugung von neuer Geometrie, z. B. für Haare (vgl. Partikeleffekte)
- DirectX 11 bzw. OpenGL 4.0 definiert zusätzlich **Tesselation-Shader** zur weiteren Unterteilung von Geometrie
- *Früher:* unterschiedliche Hardware-Einheiten für Pixel- und Vertex-Shader
- *Heute:* **Unified Shader** für alle Shadertypen, aber auch für beliebige andere Berechnungen (*GPGPU, CUDA, OpenCL*)

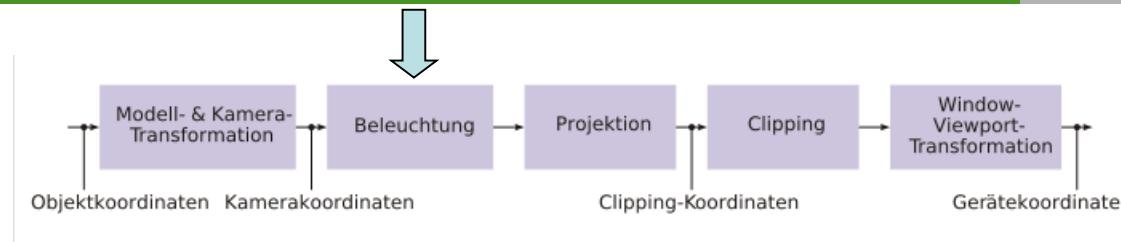


## Shading

- Flat-Shading: Jeder Pixel erhält anhand des Normalenvektors den gleichen Lichtanteil
- Gouraud-Shading:  
Jeder Vertex (Eckpunkt)  
erhält eigenen  
Lichtanteil, restliche Pixel  
werden interpoliert
- Phong-Shading:  
Aus den Normalen der  
Vertices wird für jeden Pixel  
ein eigener Normalenvektor  
interpoliert, jeder Pixel  
erhält dann eigene Berechnung für den Lichtanteil

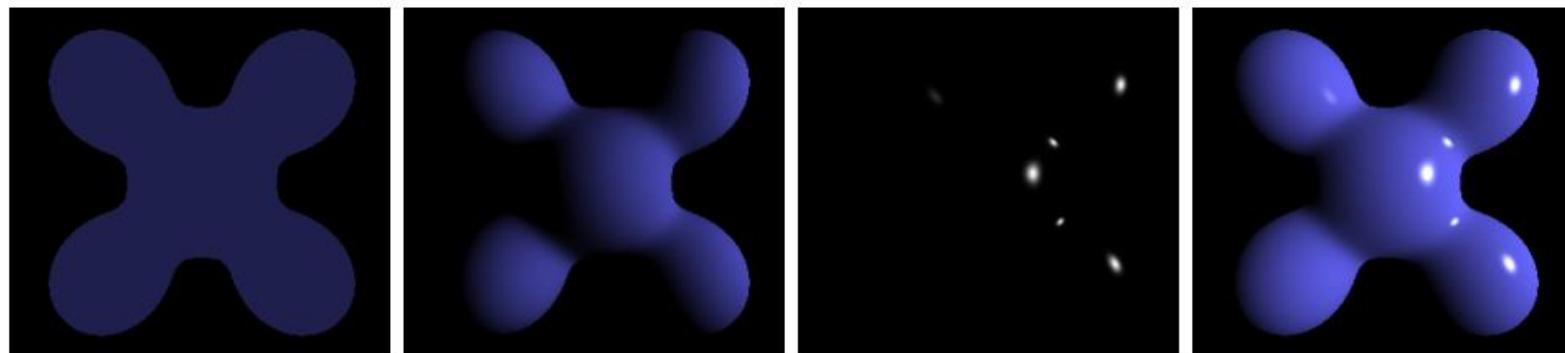


Quelle: Massuch, 2008



## Beleuchtungsmodell

- Raycasting hat normaler Weise ein globales Beleuchtungsmodell → neben direktem Licht wird auch Licht durch Streuung/Reflexion berücksichtigt
- Für Echtzeit-Grafik wird i. d. R. ein vereinfachtes (lokales) Modell genommen, z. B. das Phong-Beleuchtungsmodell:

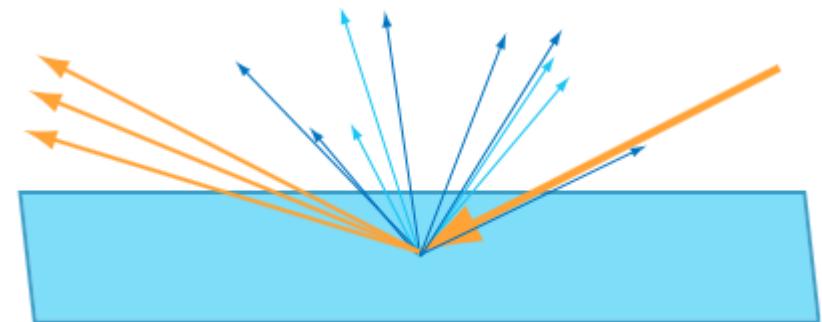
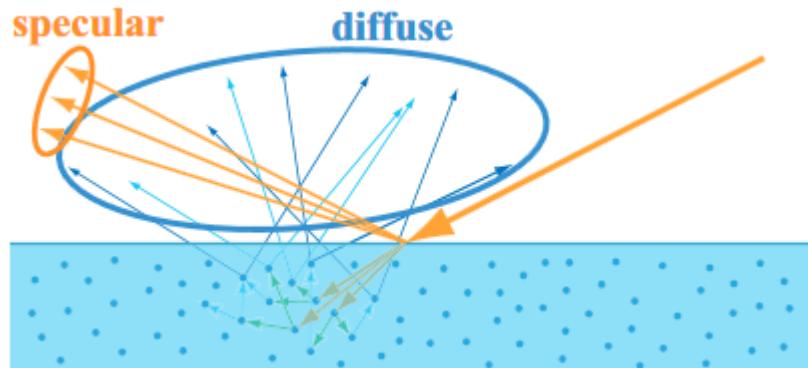


**Ambient** + **Diffuse** + **Specular** = **Phong Reflection**

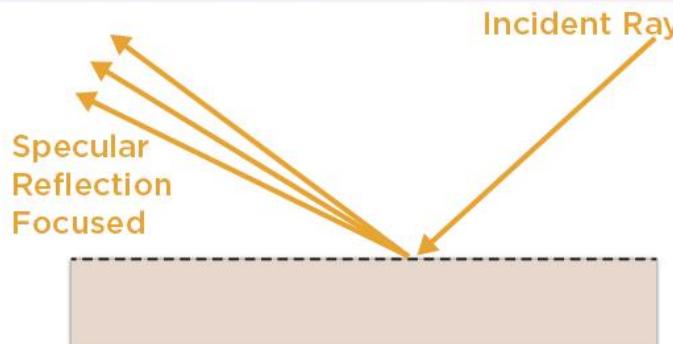
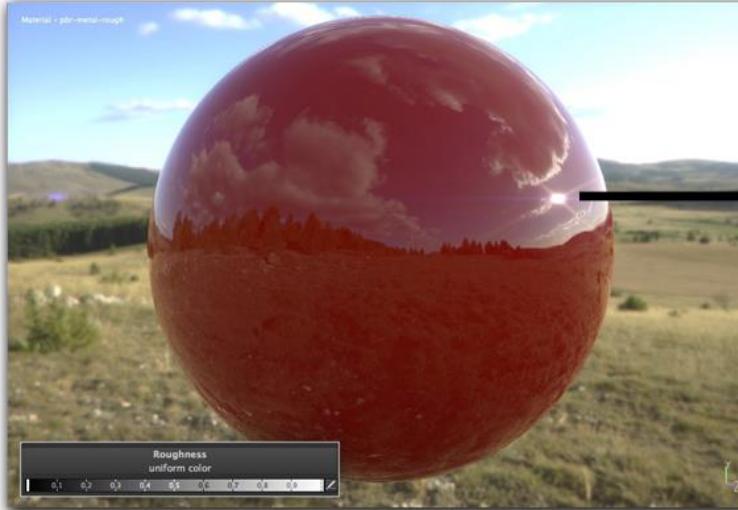
- OpenGL benutzt standardmäßig das Blinn-Beleuchtungsmodell = Phong-Modell mit Optimierungen

Diffuse = Refracturing bzw. Streuung

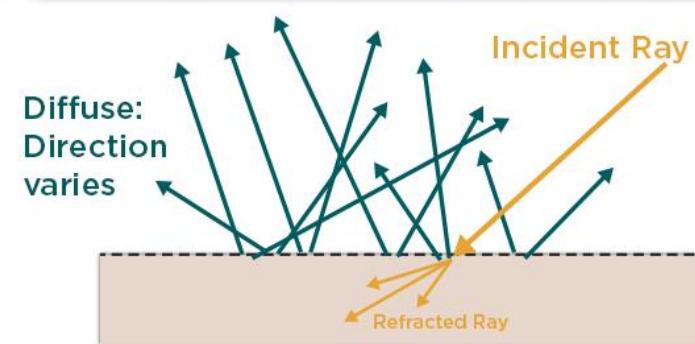
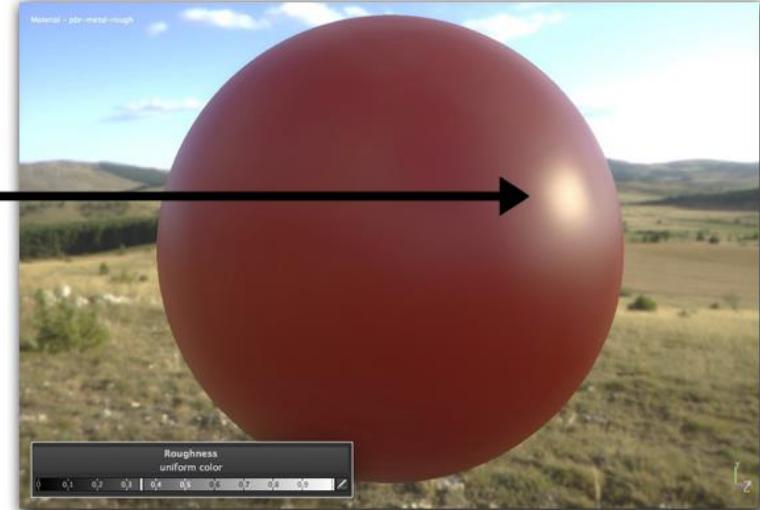
Specular = (Oberflächen) Reflektion



Smoother surface focus specular reflection

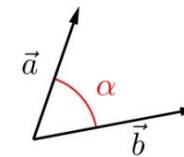
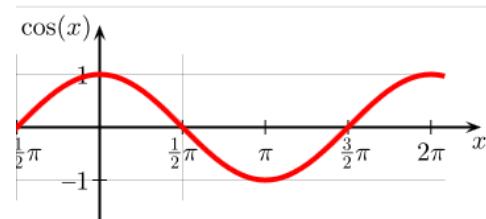


Rougher surface larger dimmer highlight



Skalarprodukt

$$\vec{a} \cdot \vec{b} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$



Mit Hilfe des Skalarprodukts kann man Winkel zwischen 2 Vektoren berechnen.

$$\cos \alpha = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| \cdot |\vec{b}|}$$

Wichtig:  $\vec{a} \perp \vec{b} \Rightarrow \vec{a} \cdot \vec{b} = 0$

$$L_o(\mathbf{v}) = \pi f(\mathbf{l}_c, \mathbf{v}) \otimes \mathbf{c}_{\text{light}} \underline{(\mathbf{n} \cdot \mathbf{l}_c)}.$$

$$L_o(\mathbf{v}) = \left( \mathbf{c}_{\text{diff}} \underline{(\mathbf{n} \cdot \mathbf{l}_c)} + \begin{cases} \mathbf{c}_{\text{spec}} \frac{(\mathbf{r}_v \cdot \mathbf{l}_c)^{\alpha_p}}{0}, & \text{if } (\mathbf{n} \cdot \mathbf{l}_c) > 0 \\ 0, & \text{otherwise} \end{cases} \right) \otimes \mathbf{c}_{\text{light}}.$$

$$L_o(\mathbf{v}) = \left( \mathbf{c}_{\text{diff}} + \mathbf{c}_{\text{spec}} \underline{(\mathbf{r}_v \cdot \mathbf{l}_c)^{\alpha_p}} \right) \otimes \mathbf{c}_{\text{light}} \underline{(\mathbf{n} \cdot \mathbf{l}_c)}.$$

C\_diff: diffuse color

C\_spec: specular color

C\_light: light color

Alpha\_p: Stärke des Glanzlichts

L\_o: „outgoing radiance“

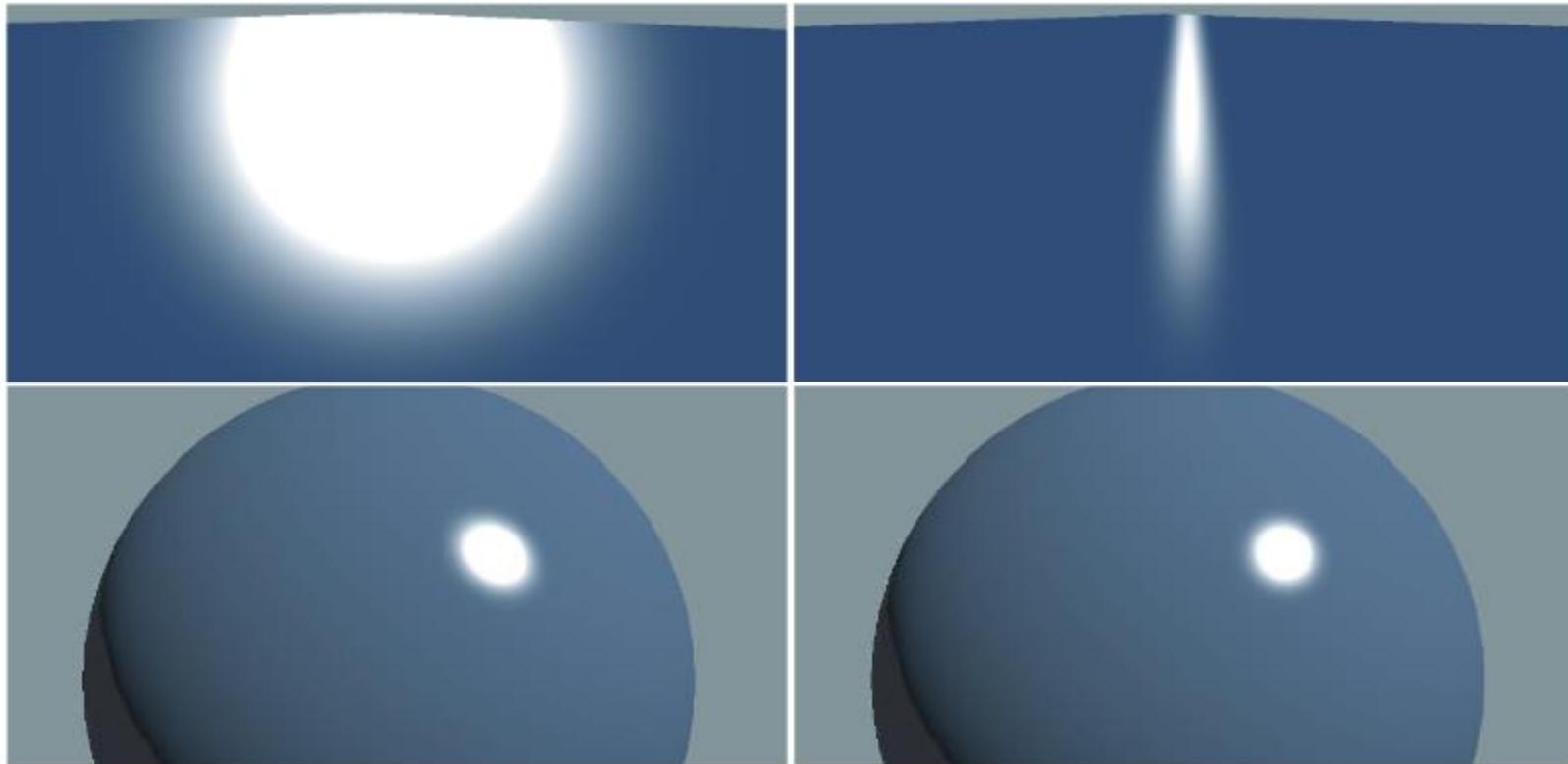
f(l,v): Reaktion der Oberfläche auf Licht

n: normal vector

r\_v: reflection vector

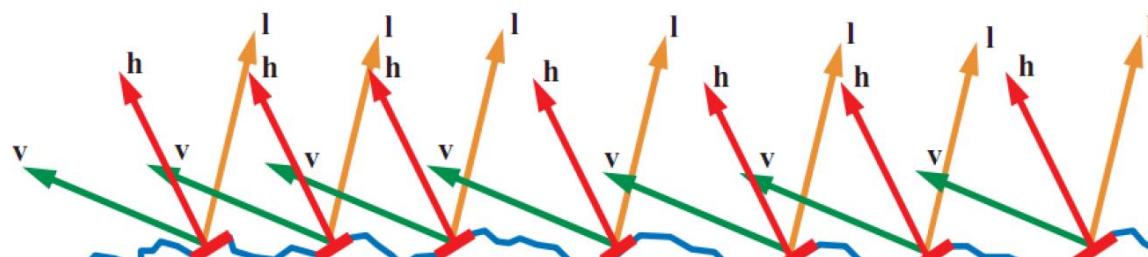
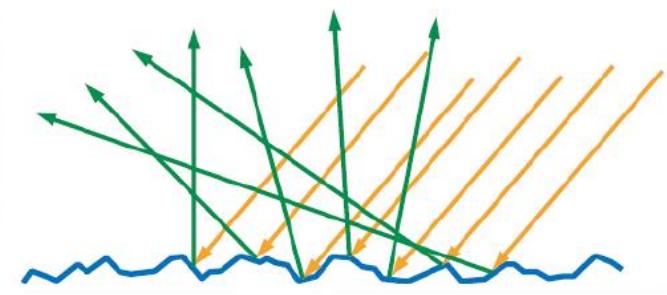
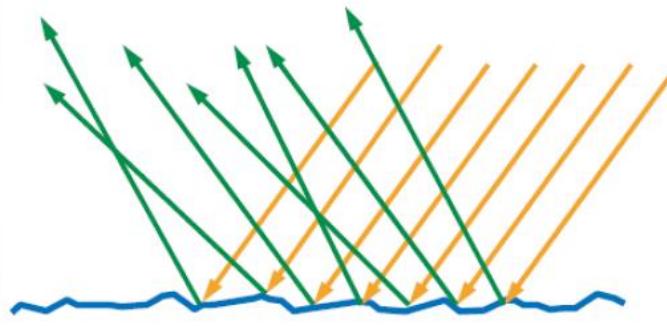
l\_c: point light location vector

## Phong vs Blinn-Phong



Realität ähnelt Blinn-Phong auf Flächen





$$L_o(\mathbf{v}) = \left( c_{\text{diff}} + c_{\text{spec}} (\underline{\mathbf{r}_v \cdot \mathbf{l}_c})^{\alpha_p} \right) \otimes c_{\text{light}} (\underline{\mathbf{n} \cdot \mathbf{l}_c}).$$

Blinn-Phong:

$$L_o(\mathbf{v}) = (\underline{\mathbf{n} \cdot \mathbf{h}})^{\alpha_p} c_{\text{spec}} \otimes c_{\text{light}} (\underline{\mathbf{n} \cdot \mathbf{l}_c}).$$

C\_diff: diffuse color

C\_spec: specular color

C\_light: light color

Alpha\_p: Stärke des Glanzlichts

L\_o: „outgoing radiance“

f(l,v): Reaktion der Oberfläche auf Licht

n: normal vector

r\_v: reflection vector

l\_c: point light location vector

Angepasst

Energieerhaltung



BRDF (Bidirectional Reflectance Distribution Function)

l: light direction

v: view vector

n: normal

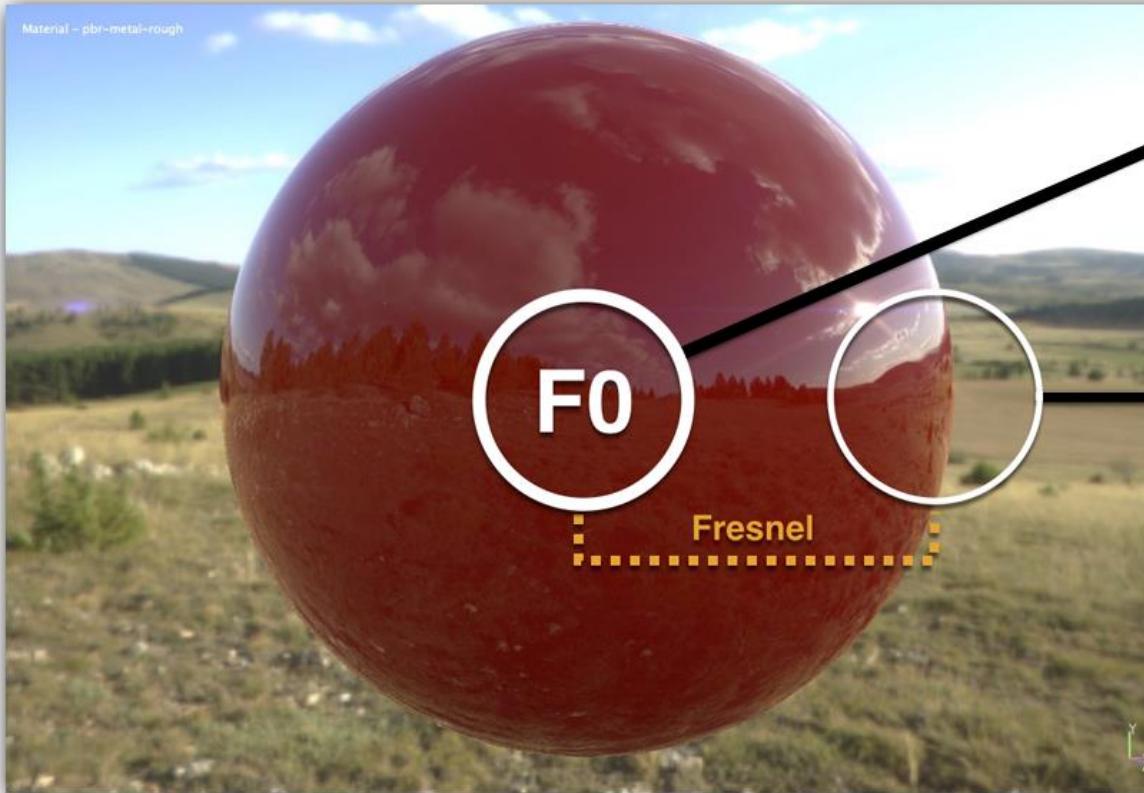
h: half-vector (light, view)

F: Fresnel

G: Geometry

D: Diffuse

$$f_{\mu\text{facet}}(\mathbf{l}, \mathbf{v}) = \frac{F(\mathbf{l}, \mathbf{h})G(\mathbf{l}, \mathbf{v}, \mathbf{h})D(\mathbf{h})}{4(\mathbf{n} \cdot \mathbf{l})(\mathbf{n} \cdot \mathbf{v})}$$



Substance Painter Viewport

$$L_o(\mathbf{v}) = \pi \frac{D(\mathbf{h}) G(\mathbf{l}_c, \mathbf{v}, \mathbf{h})}{4(\mathbf{n} \cdot \mathbf{l}_c)(\mathbf{n} \cdot \mathbf{v})} F(\mathbf{l}_c, \mathbf{h}) \otimes \mathbf{c}_{\text{light}}(\underline{\mathbf{n} \cdot \mathbf{l}_c})$$

$$L_o(\mathbf{v}) = \underline{(\mathbf{n} \cdot \mathbf{h})^{\alpha_p}} \mathbf{c}_{\text{spec}} \otimes \mathbf{c}_{\text{light}}(\underline{\mathbf{n} \cdot \mathbf{l}_c}).$$

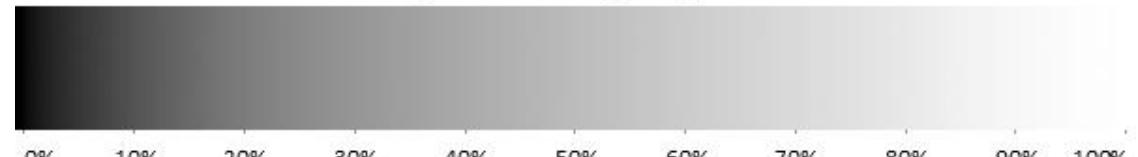
- Fresnel
- Normalisierung

$$\frac{\alpha_p + 2}{8} \underline{(\mathbf{n} \cdot \mathbf{h})^{\alpha_p}} F_{\text{Schlick}}(\mathbf{c}_{\text{spec}}, \mathbf{l}_c, \mathbf{h}) \otimes \mathbf{c}_{\text{light}}(\underline{\mathbf{n} \cdot \mathbf{l}_c}).$$

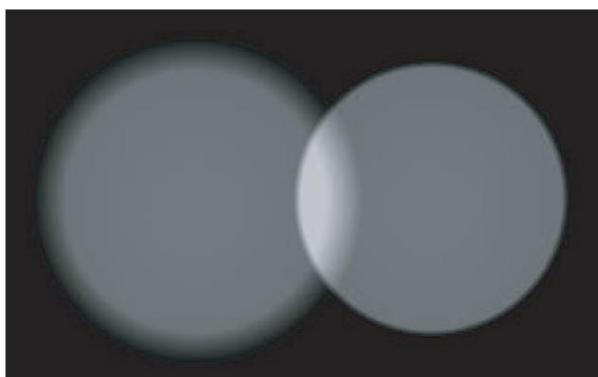
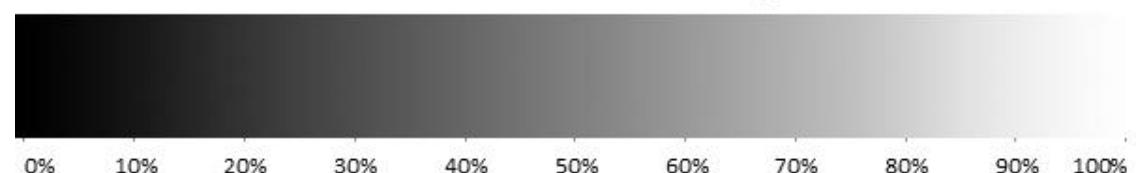
Es fehlen: Transparenz, Subsurface-Scattering...

<http://renderwonk.com/publications/s2010-shading-course/>

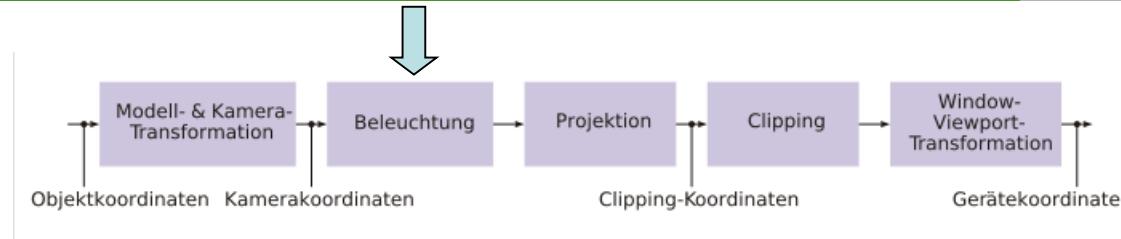
Linearly increasing brightness



sRGB Gamma Corrected Brightness



sRGB (nonlinear, Monitore) vs lineare Kodierung

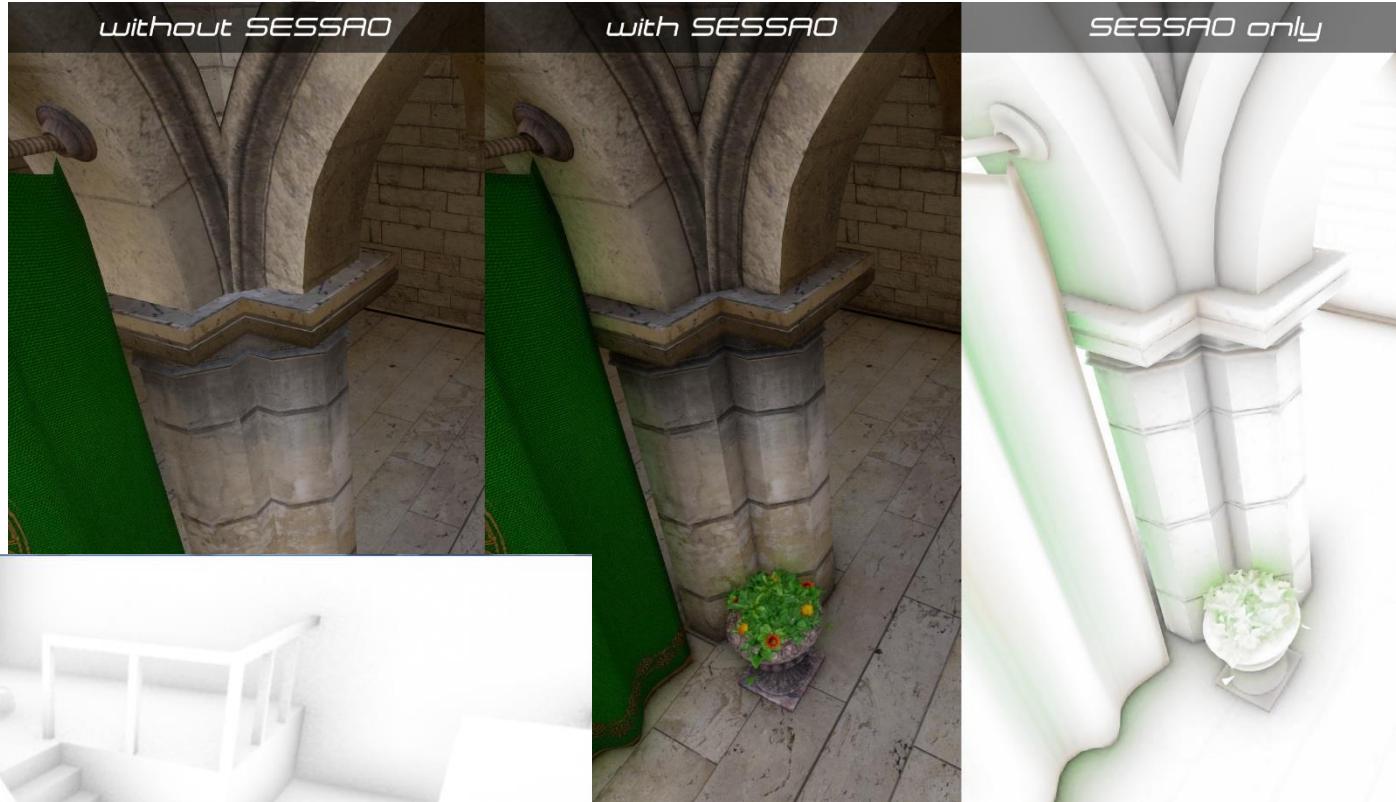


Verbesserung für lokales Beleuchtungsmodell: Screen Space Ambient Occlusion (SSAO)

- Analysiert auf geschickte Weise den Tiefenpuffer (evtl. auch noch Normalen, ...) während dem Rendern um festzustellen, wie viel andere Geometrie in der Nähe eines zu rendernden Pixels ist
- Je mehr Objekte desto weniger ambientes Licht → deutlich realistischer als konstantes ambientes Licht



# Verdeckung und indirekte Beleuchtung





## GDC 2018 Demos

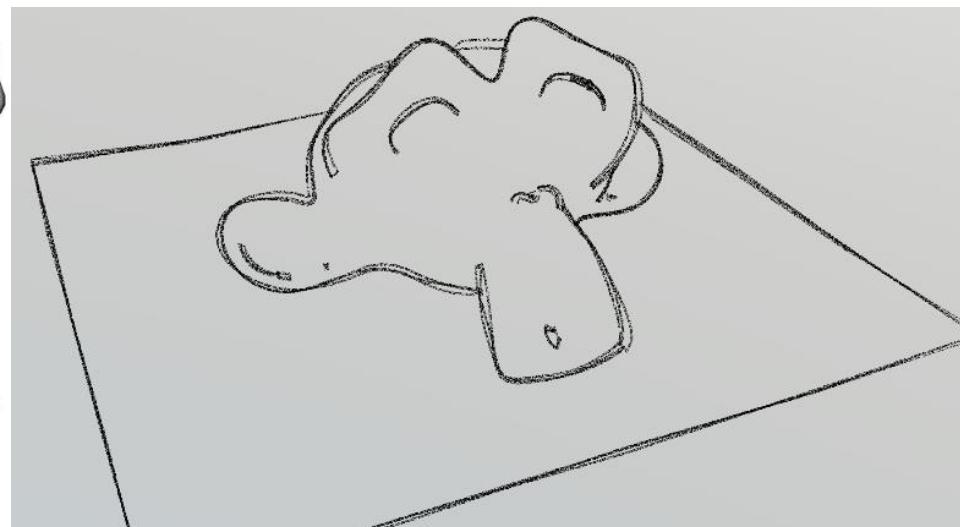
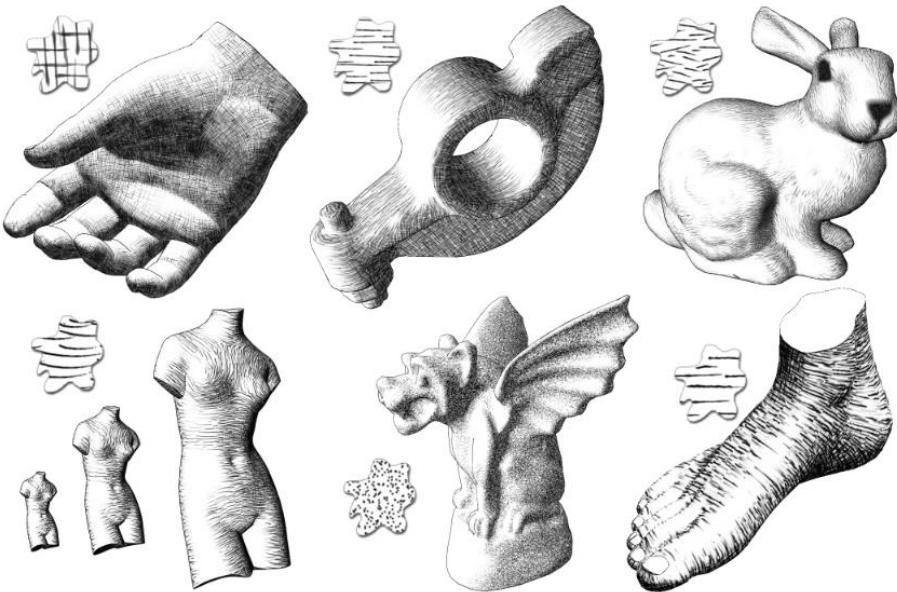
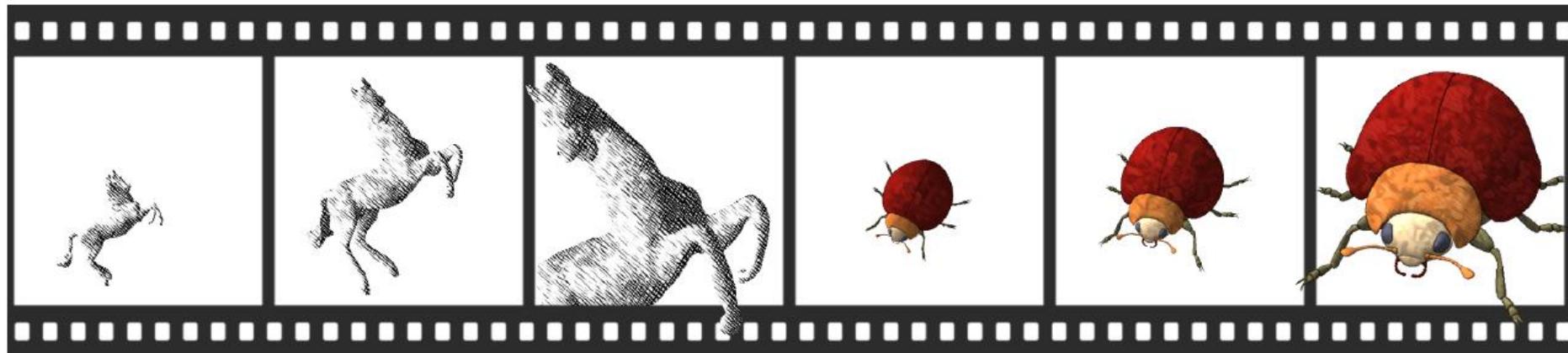
### Raytracing für

- Reflektionen
- Globale Beleuchtung
- Schatten

### Integration in zukünftige APIs

- DirectX RTX
- Vulkan

- Nimmt gerastertes Bild als Basis
- Aufwändiges De-Noising



# Non Photorealistic Rendering



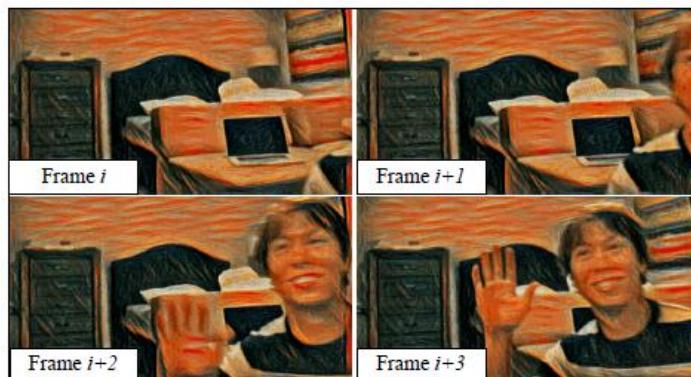
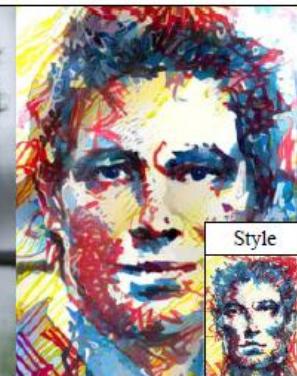
Texture Transfer / Risser et al. (2017)



Color Transfer / Gatys et al. (2016b)



Portrait Stylization / Selim et al. (2016)



Video Stylization / Johnson et al. (2016a; 2016b)

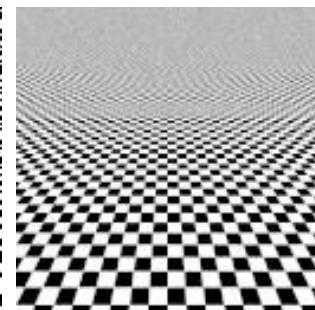
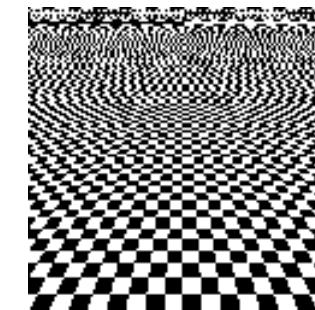
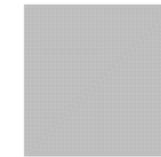
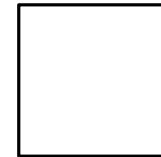


Casual Creativity / Prisma (iOS)

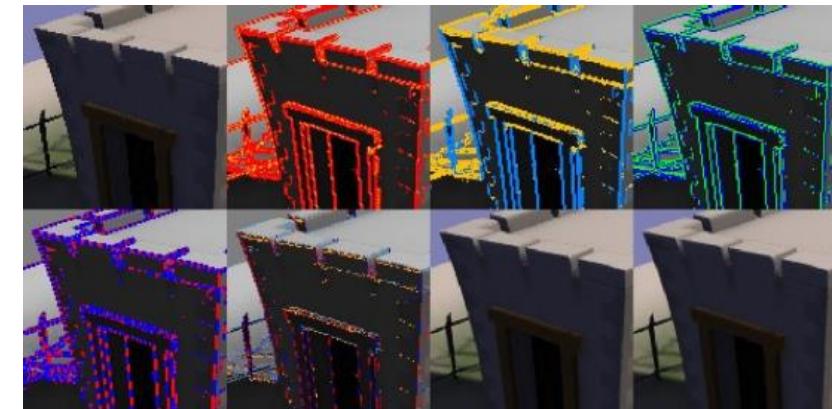


Casual Creativity / likemo.net

- *Beispiel:* Schwarzes Dreieck bedeckt Pixel nur teilweise
- *Ohne Kantenglättung:* Pixelfarbe bestimmt über Pixel-Mittelpunkt
- *Mit Kantenglättung:* Interpolation der Farbe von mehreren Punkten des Pixels
  - z. B. 1x2: 2 Punkte vertikal mittig, Quincunx: 4 Ecken + Mitte, 4x4: 16 verteilte Punkte, ...

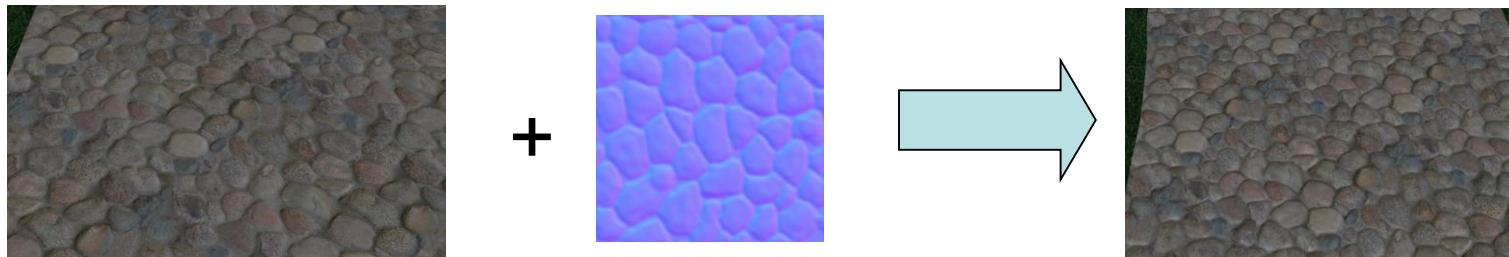


- Rendering eines Bildes mit höherer Auflösung  
→ Downsampling mit Interpolation der Nachbarpixel  
(=**Supersampling AA=SSAA** bzw. **Full-Scene AA =FSAA**)
- Ähnlich: Rendern von mehreren Bildern mit verschobener Kamera in **AccumulationBuffer** (Spezialfall: rotated grid SS **RGSS**)
- *Effizienter. Multi-Sampling (MSAA)*: Nicht alle Berechnungen n mal pro Pixel, sondern ein Teil nur 1 mal pro Pixel
- *Noch schneller. Fast approximate AA (FXAA)*: Filter auf schon gerendertes Bild: 1-3. Horizontale und vertikale Kantendetektion, 4. Abschnitte ähnlicher Luminanz, 5. Berechnung von Sub-Pixel-Verschiebung aus Daten von 1-4, 6. Resampling der Textur anhand der Verschiebungen, 7. Zusätzlicher Weichzeichner auf Kanten
- Ähnlich: **Morphological AA (MLAA)**
- **Temporal AA**: Bewegungen von Objekten werden berücksichtigt
- **DLSS Deep Learning SS**: Neuronale Netze interpolieren



# Bump Mapping

- Erschaffung realistischerer Oberflächen ohne Änderungen der Geometrie
- **Normal Mapping:** Vor der Berechnung der Lichtwerte für einzelne Pixel werden deren Normalenvektoren gemäß einer sog. Normalmap verändert



- **Parallax mapping:** Zusätzliche Verschiebung der Texturkoordinaten abhängig vom Blickwinkel



# Bump Mapping

- Dank späterer Verbesserungen (**Steep Parallax Mapping**) können sich Bumps (Unebenheiten) auch tatsächlich gegenseitig verdecken



Texture Mapped



Normal Mapped



Parallax Mapped



Steep Parallax Mapped

- Displacement mapping:** Es wird doch die Geometrie (Vertices) geändert → Tesselation notwendig



Base Model



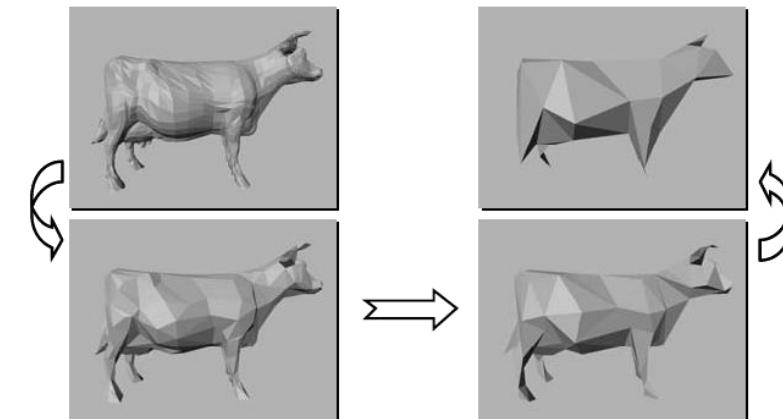
Bump Mapping



Displacement Mapping

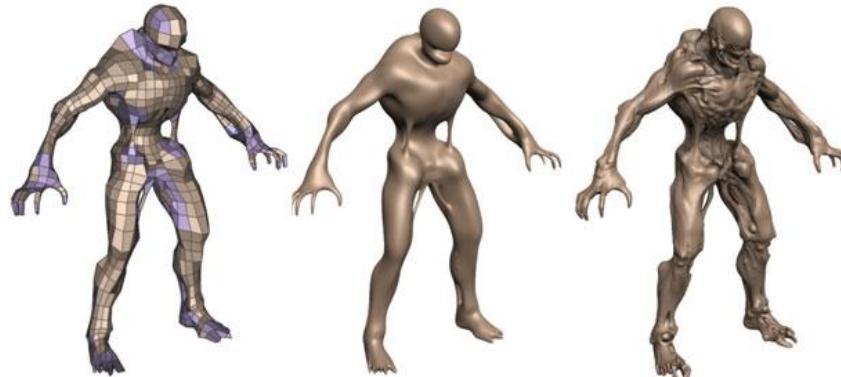
Image courtesy of [www.chromesphere.com](http://www.chromesphere.com)

- Dynamische Anpassung des Detailgrads (Level of Detail, LOD) von Objekten an die Entfernung zum Betrachter um Ressourcen zu sparen:
  - Schnelleres Rendern
  - Reduktion der Komplexität von geometrischen Operationen, z.B. Kollisionserkennung
- Diskretes LOD: Für ein Objekt werden mehrere Modelle mit abnehmendem Detailgrad angefertigt, je nach Distanz werden die Modelle ausgetauscht
- Dynamisches LOD: Je nach Distanz werden Dreiecke dynamisch gesplittet oder vereint
  - *Problem:* „Split“ und „Merge“ sind nicht „sicher“ → Artefakte



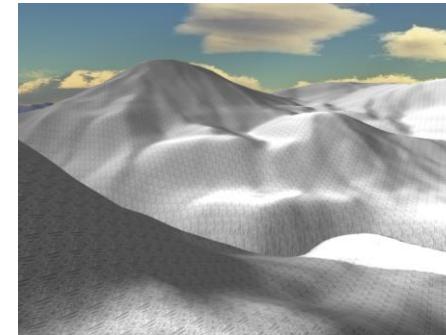
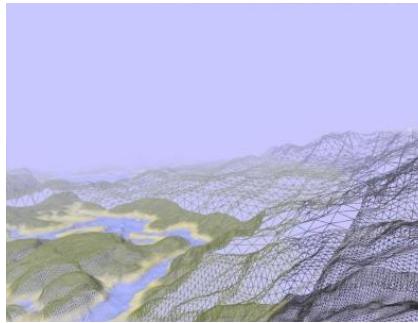
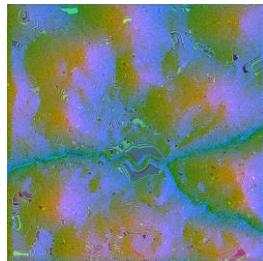
# Tesselation

- ab DirectX 11 bzw. OpenGL 4.0
- Methode für dynamisches Level of Detail
- Erstellung von glatten Objekten durch Aufteilung der Dreiecke des 3D-Modells
- Dem glatten Modell werden ggf. durch Displacement Mapping wieder Details hinzugefügt



# Terrainrendering

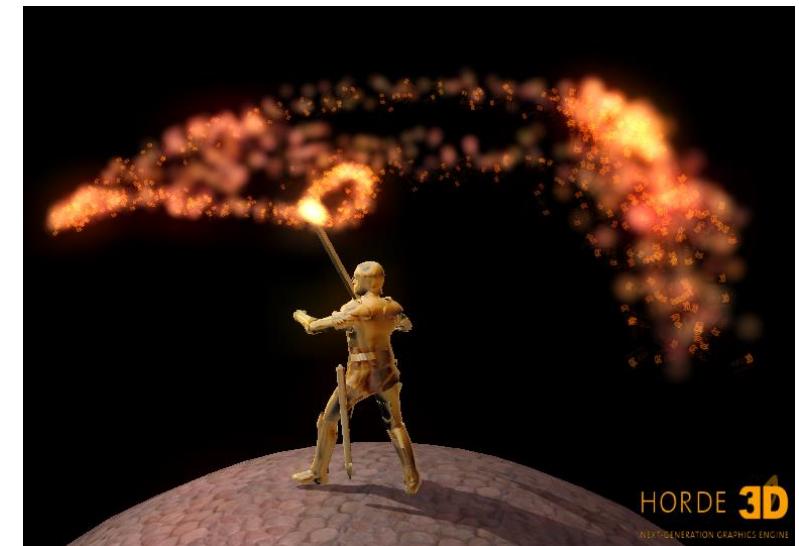
- Automatische Generierung von möglichst realistischen Landschaften
- Höhenwerte in sog. Heightmap gespeichert, Approximation von nicht gespeicherten Punkten



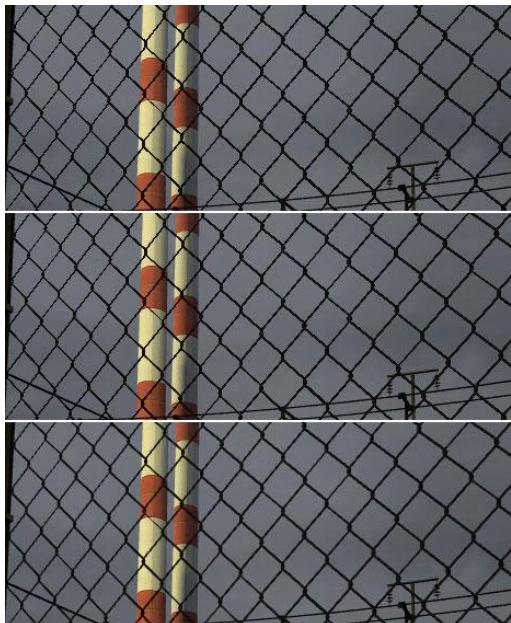
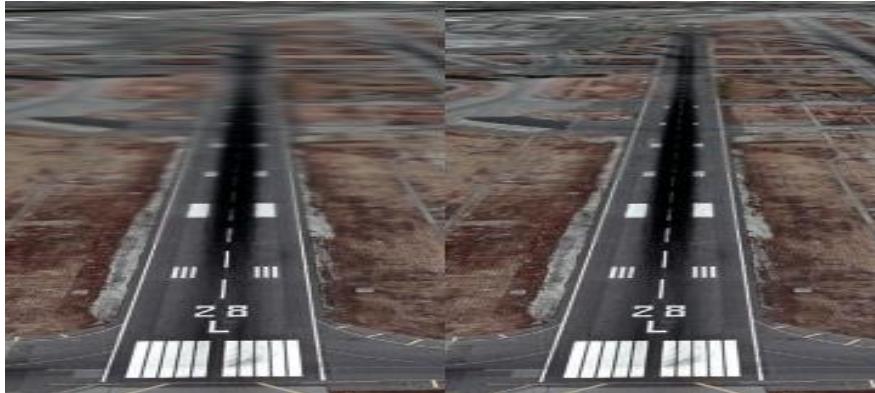
- LOD von Terrains z.B. ROAM (Real-Time Optimally Adapting Meshes) Dynamischer LOD-Algorithmus für Terrains
- Alternative: **Voxel-Grafik** wie in Comanche, Outcast, Minecraft (keine echte Voxel-Grafik!), oder teilweise in Crysis ermöglicht Höhlen und Bögen sowie dynamische Veränderung der Landschaft



- Werden verwendet um bestimmte Effekte zu erzeugen, die sich mit herkömmlichen Render-Techniken nur schwer erzielen lassen:  
Feuer, Rauch, fließendes Wasser, Schnee, Nebel, Fell, Gras, magische Effekte
- Emittor sind 3D-Objekte in der Szene, an denen Partikel entstehen  
Partikel haben bestimmte Eigenschaften, z. B.: Entstehungsrate, Initialvektor, Lebensdauer, Farbe/Textur
- Eigenschaften meist „fuzzy“
- Während ihrer Lebensdauer bewegen sich Partikel gemäß ihres Anfangsvektors, vorgegebenen Pfaden und äußeren Einflüssen (Schwerkraft, Wind)
- Partikel werden entweder als einzelne farbige Pixel oder als Billboards gerendert
- Oft Teil von Physik-Engines, Hardware-Berechnung möglich



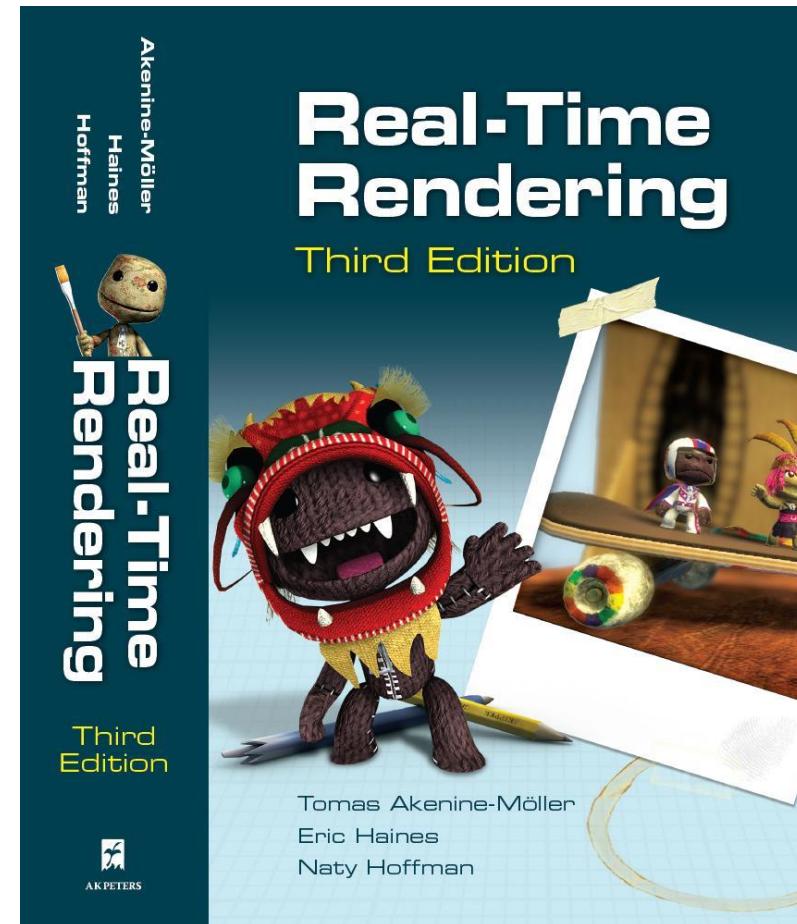
# Verfahren zur Verbesserung der Texturdarstellung



- **Bilineare/Trilineare/Anisotrope Filterung mit Mip Mapping:** „Anti-Aliasing für Texturen“, wirkt Verzerrungen und Detailverlust von Texturen bei hohen Entfernungen und extremen Betrachtungswinkeln entgegen
- **Texture Streaming:** Level of Detail für Texturen, niedrig auflösende Texturen werden bei geringerem Abstand mit hoch auflösenden ersetzt, dynamisches Nachladen (Streaming) von Texturen
- **Textur Kompression** um Speicher zu sparen: am verbreitesten: **S3TC** von S3 → **DXTx/BCx** in DirectX; ursprünglicher Konkurrent: **FXT1** von 3dfx
- **Transparency Anti-Aliasing** zur Kantenglättung in Texturen mit Transparenz

# Buchtipp

Akenine-Möller, Haines, Hoffman: „Real Time Rendering“  
(auch in der Bibliothek!)



SIGGRAPH 2010 Course:  
Physically-Based Shading Models in Film and Game Production

<http://blog.selfshadow.com/publications/s2017-shading-course/>

Real-Time Hatching 2001, Emil Praun et al.

Neural Style Transfer:  
A Paradigm Shift for Image-based Artistic Rendering? 2017, Amir  
Semmo et. al.

NPR Gabor Noise for Coherent Stylization 2010, Pierre Bénard et al.

[http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXA\\_A\\_WhitePaper.pdf](http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXA_A_WhitePaper.pdf)