# Interaktive Computergrafik

## Vorlesung im Sommersemester 2017
## Kapitel 3: Deferred Shading und Bildraum-Techniken

Prof. Dr.-Ing. Carsten Dachsbacher
Lehrstuhl für Computergrafik
Karlsruher Institut für Technologie

Karlsruhe Institute of Technology

# Inhalt

- in diesem Kapitel befassen wir uns mit Techniken, die zum Einsatz kommen wenn **Shading teuer** ist, z.B. aufgrund von
  - Beleuchtungsberechnung durch viele Lichtquellen
    (wir ignorieren zunächst das Problem der entsprechend teuren Schattenberechnung)
  - prozeduraler Texturierung
  - teuren Datenstrukturen (3D-Textur/Irradiance Volumes mit Octrees)
  - wenn wir Ambient Occlusion, Tiefenunschärfe, indirekte Beleuchtung usw. (approximativ im Bildraum) berechnen möchten

- wir beschäftigen uns in diesem Kapitel *noch nicht* mit
  - Entfernen von Geometrie außerhalb des View Frustum oder Anpassung der Tessellierung
  - beides ist natürlich ebenfalls wichtig für effizientes Rendering

# Viele Lichtquellen?


Need For Speed: The Run, 2600 LQ

Starcraft 2 [Blizzard 2010], 25 LQ

# Teures Shading?


Bilder: RTT DeltaGen
http://www.rtt.ag/en/offering/software/author/rtt-deltagen

# Motivation

## Single-Pass Lighting (für mehrere Lichtquellen)

- einfachste Möglichkeit: ein Vertex- und Fragment Programm für alle Aufgaben

```
für jedes Objekt
    zeichne Objekt und behandle alle LQ in einem Shader
```

- Problem: auch verdeckte Flächen werden mit einem (teuren) Fragment-Programm rasterisiert

- Handhabung verschiedener Materialien und Lichtquellen schwierig
  - Verzweigungen für Material-Lichtquellen-Kombinationen
  - begrenzte Anzahl von Interpolatoren (GLSL in/out), d.h. nicht beliebig viele Werte pro Vertex deren Resultat weitergegeben werden kann

# Motivation

## Multi-Pass Lighting

- Rendering mit Trennung nach Lichtquellen

```
für jede Lichtquelle
  für jedes Objekt (optional: im Einflussbereich der LQ)
    framebuffer += Beleuchtung( Objekt, Licht )
```

- Vorteil: einfachere Shader
- aber viele Rendering-Aufrufe: Anzahl Objekte × Anzahl Lichtquellen
- Hauptproblem: viele Aufgaben fallen mehrfach an
  - Geometrieverarbeitung
  - Rasterisierung
  - Texturfilterung
  - …

# Motivation

## Depth-Only Pass

- GPUs unterstützen einen speziellen „Depth-Only“-Render-Modus
  - fülle in einem ersten Render-Durchgang nur den Tiefenpuffer: Rendering schneller, wenn Schreiben in den Frame-Buffer deaktiviert
  - zeichne dann das eigentliche Bild, wobei die GPU dafür sorgt, das für verdeckte Flächen (fast) kein Fragment-Shader angestoßen wird

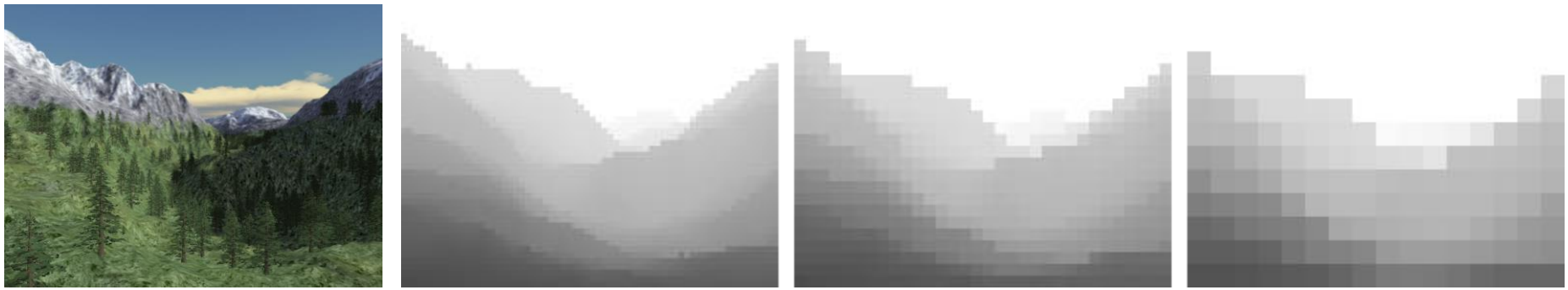
```
glColorMask( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE );
Depth-Only Pass
glColorMask( GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE );
glDepthMask( GL_FALSE );

für jede Lichtquelle
  für jedes Objekt (optional im Einflussbereich der LQ)
    framebuffer += Beleuchtung( Objekt, Licht )
```

- trotzdem fallen Aufgaben mehrfach an:
  - Geometrieverarbeitung und Rasterisierung
  - Texturfilterung (aber nur für sichtbare Flächen)

# Hierarchischer Z-Buffer

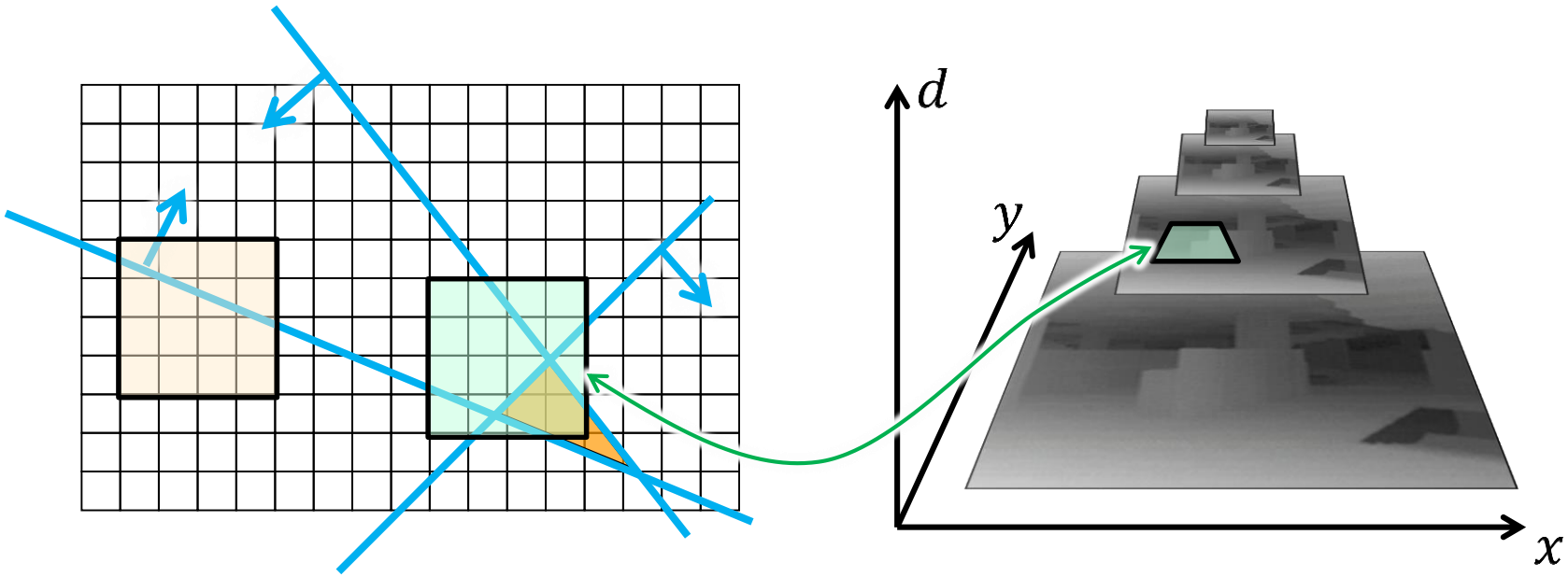
- Ausschließen von verdeckten Flächen für Fragment-Programme findet im Bildraum auf $N \times N$ Pixel-Blöcken (typ. $N = 4$ od. $N = 8$) statt (Funktionalität ist transparent für den Programmierer)
- Idee: Hardware speichert **hierarchische Tiefeninformation**
  Hierarchical Z-Buffer Visibility, Greene et al., SIGGRAPH‘93
  - feinste Stufe = herkömmlicher Tiefenpuffer
  - gröbere Stufen werden ähnlich Mip-Maps erzeugt, speichern aber den **maximalen Tiefenwert** eines $2 \times 2$ Blocks der nächstfeineren Stufe



Bilder: http://rastergrid.com/blog/2010/10/hierarchical-z-map-based-occlusion-culling/

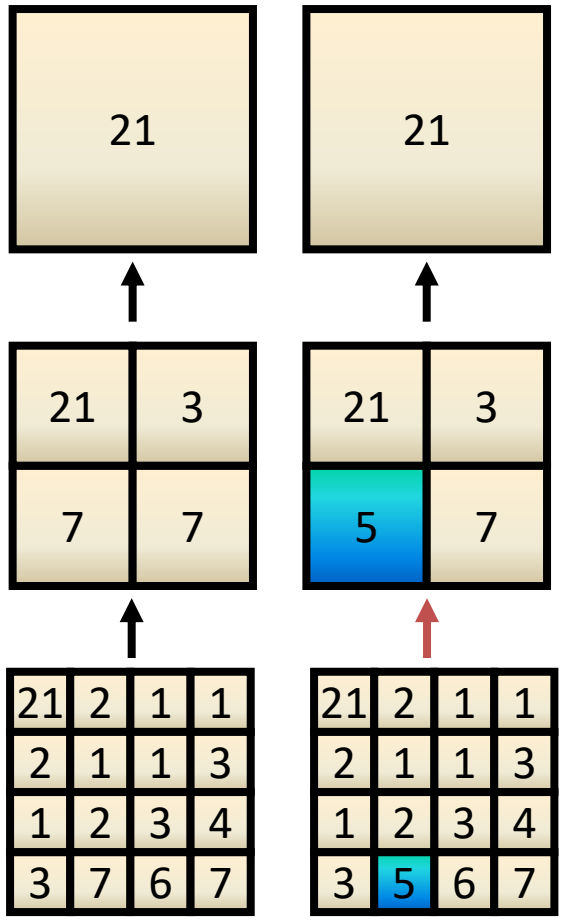
# Hierarchischer Z-Buffer

- GPUs verwenden zur Rasterisierung den Kantentest-Ansatz mit Binning
- zuerst werden größere Pixel-Blöcke getestet, ob sie Teile des Dreiecks enthalten, dann (rekursiv) kleinere
- für jeden Block der kleiner-gleich $N \times N$ Pixel ist wird…
  - … Stufe in der $z$-Pyramide bestimmt, in der 1 Pixel den Block abdeckt
  - der Block kann nicht sichtbar sein, wenn der minimale Tiefenwert des Dreiecks dort größer als die maximale Tiefe aus der $z$-Pyramide ist



# Hierarchischer Z-Buffer

- bei Änderungen des Tiefenpuffers muss die Hierarchie durch Propagieren der geänderten Werte konsistent gehalten werden
- implementiert in moderner Grafikhardware
  - z.B. ATI HyperZ-Technology, NVIDIA Early-Z Rejection
  - funktioniert nur, wenn
    - Fragment Shader keine Tiefenwerte verändern
    - der Tiefentest innerhalb eines Rendering-Durchgangs nicht geändert wird
- idealerweise zeichnet man die Szene trotzdem von vorne nach hinten: der HZB funktioniert auch innerhalb eines Rendering-Durchgangs



# Deferred Shading

- bei allen bisherigen sog. „Forward Rendering“-Strategien fallen Aufgaben mehrfach an: Geometrieverarbeitung, Rasterisierung, Texturfilterung
  → Deferred Shading adressiert diese Probleme

- Deferred Shading entkoppelt das Rendering in zwei Phasen
  - **Geometriephase**: Sammeln der **Information im Bildraum** über sichtbare Flächen für die Beleuchtungsberechnung
  - **Beleuchtungsphase**: Beleuchtungsberechnung nur für sichtbare Flächen, **unabhängig von der Geometrie der Szene**

- Bildraum-Information kann für eine Reihe weiterer Berechnungen genutzt werden, z.B.
  - morphologisches Anti-Aliasing
  - approximatives Ambient Occlusion, indirekte Beleuchtung
  - Kantenfilter für stilistisches oder nicht-photorealistisches Rendering (NPR), …

# Deferred Shading: Geometriephase

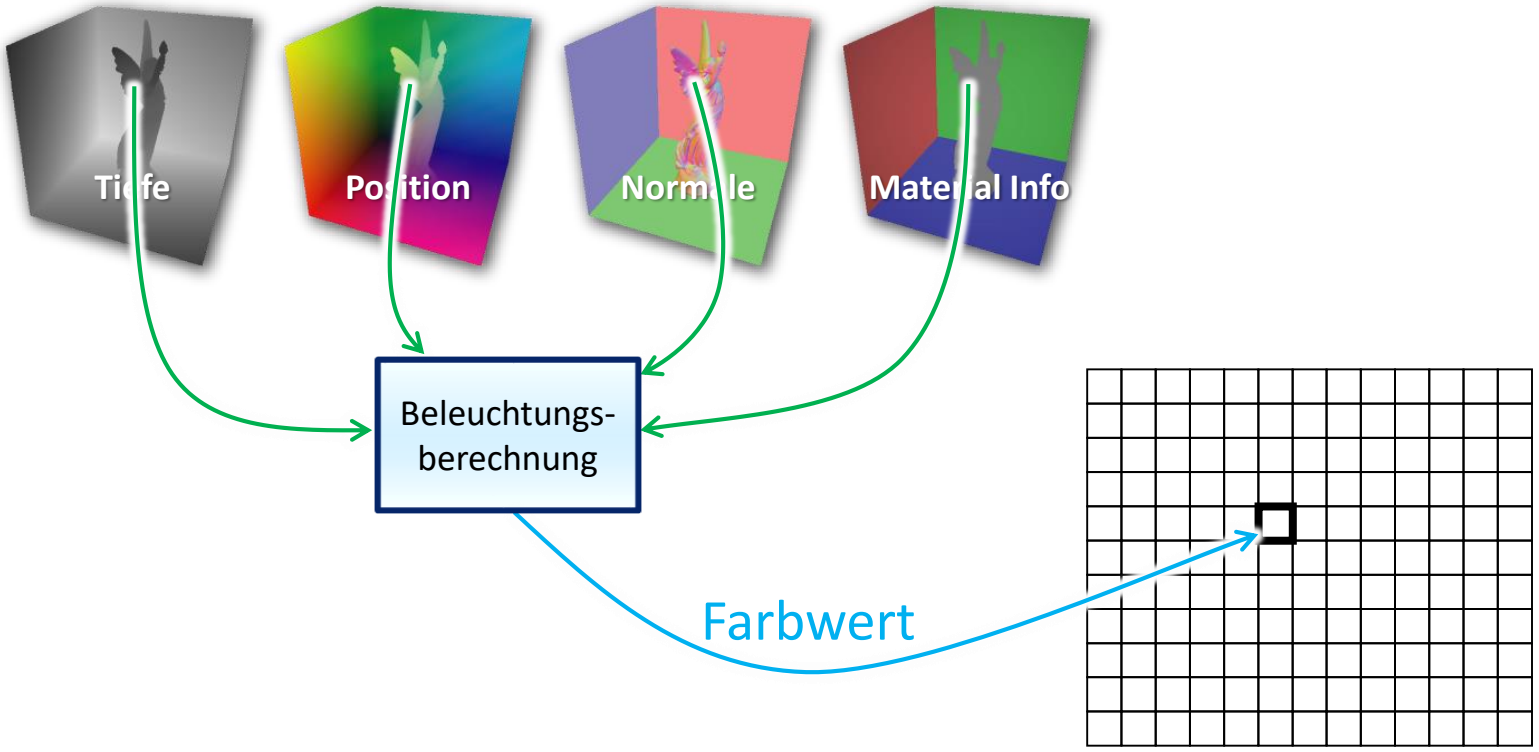
- Füllen des Geometry-Buffer

```
für jedes Objekt:
    schreibe Oberflächeninformationen in „G-Buffer“
```

  - speichere sämtliche Information, die zur Beleuchtungsberechnung an einem Pixel notwendig ist: Position, Normale, BRDF Parameter, … (variiert je nach Implementation)
  - dazu werden mehrere Texturen in Rendering-Auflösung verwendet:

# Deferred Shading: Beleuchtungsphase

- im G-Buffer ist die Information für die Beleuchtungsberechnung eines Pixels gespeichert → führe ein Fragment-Programm für jeden Pixel aus
- dem Fragment-Programm stehen folgende Daten zur Verfügung
  - die Texturen des G-Buffers
  - „globale Informationen“: Position der Lichtquellen und Kamera, Shadow Maps etc. (als Uniforms, UBOs, SSBOs, Texturen)



# Deferred Shading: Beleuchtungsphase

- im G-Buffer ist die Information für die Beleuchtungsberechnung eines Pixels gespeichert → führe ein Fragment-Programm für jeden Pixel aus
- dem Fragment-Programm stehen folgende Daten zur Verfügung
  - die Texturen des G-Buffers
  - „globale Informationen“: Position der Lichtquellen und Kamera, Shadow Maps etc. (als Uniforms, UBOs, SSBOs, Texturen)

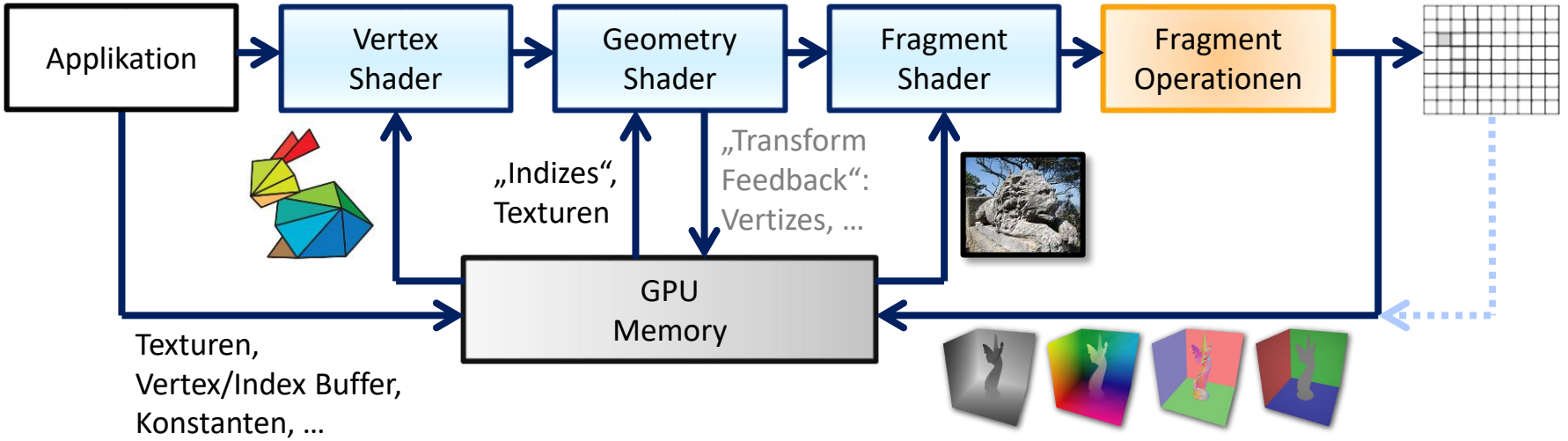
- für jede Lichtquelle wird ein bildfüllendes Rechteck gezeichnet: damit wird für jeden Pixel ein Fragment-Programm ausgeführt

```
für jede Lichtquelle
  für jeden Pixel im Framebuffer (= Zeichne Rechteck)
    Pixel += Beleuchtung( G-Buffer, LQ )
```

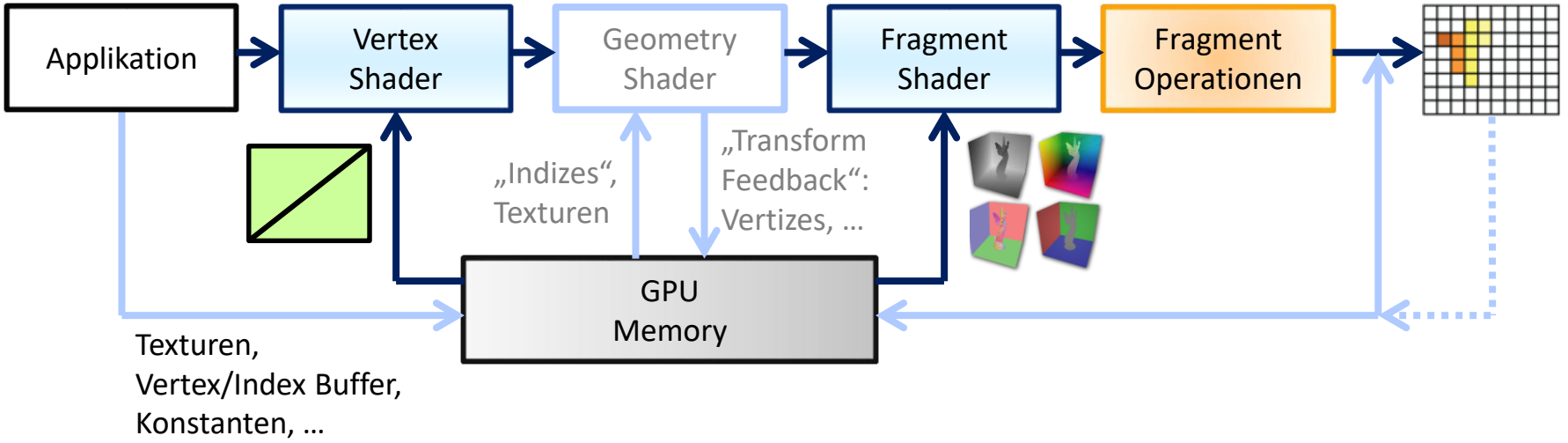
  - Akkumulation der Beiträge der Lichtquellen erfolgt im Framebuffer oder FBOs mittels additivem Blending (**`GL_ONE`**)

# Deferred Shading: Überblick / Datenfluss

- Geometriephase: Rendering der Szenengeometrie



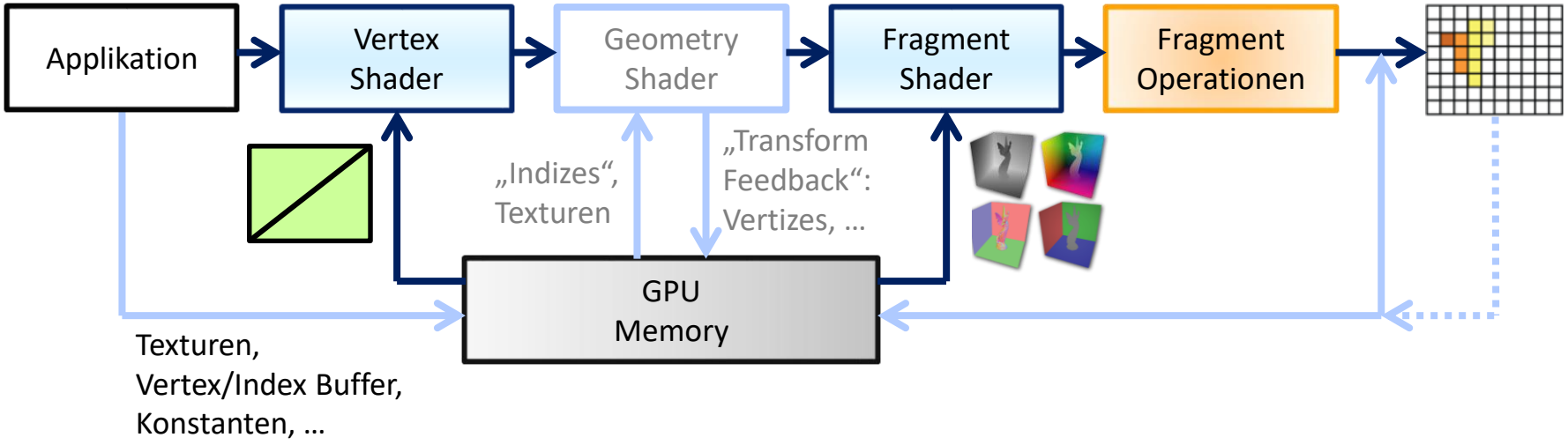

- Beleuchtungsphase: Rendering eines Rechtecks (u.U. mehrfach)



# Deferred Shading

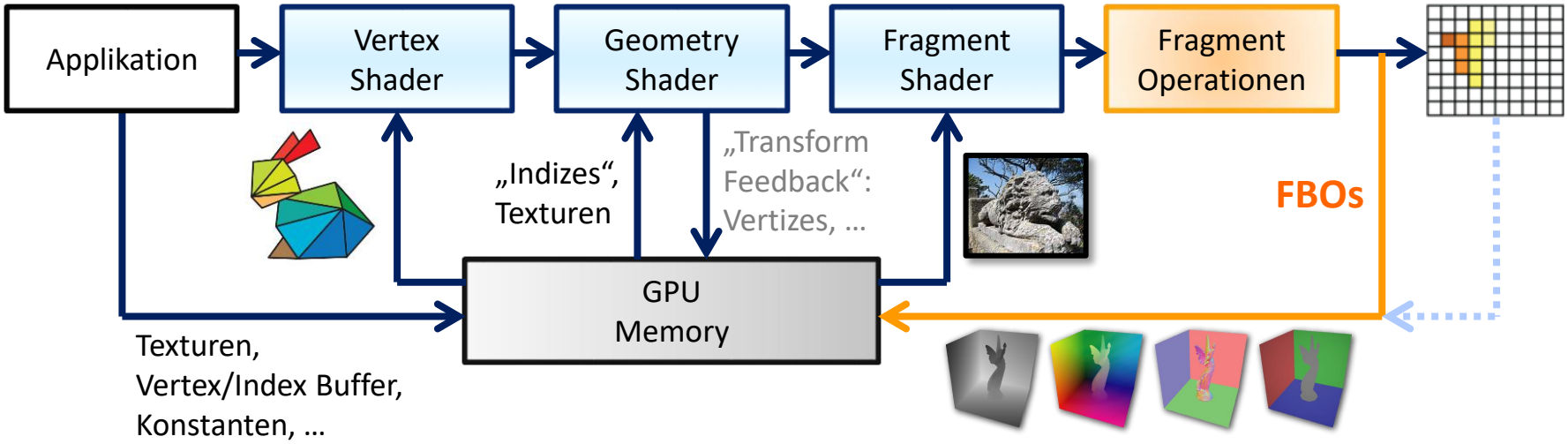
## Vorteile

- keine Beleuchtungsberechnungen für verdeckte Flächen
- vereinfacht komplexere Rendering-Systeme durch Entkopplung von Geometrie und Beleuchtung (Bsp. aufwändige Animation)
- Idee von Deering et al. (1988), praktikabel seit ca. 2002 (GPUs mit Floating Point) – heute in verschiedenen Ausprägungen weit verbreitet
- wir besprechen moderne Varianten und aktuelle Techniken
- gleich: viele kleine Lichtquellen können effizient behandelt werden (und sind evtl. nicht teurer als eine große/helle Lichtquelle)



# Deferred Shading: praktische Aspekte

- betrachten wir ein naheliegendes, naives G-Buffer-Layout
  - pro Pixel: Position (3 Floats), Normale (3 Floats), Material mit diffuser und spekularer Farbe und Phong-Exponent (7 Floats)
  - Speicherbedarf bei FullHD: $1920 \times 1080 \times 13 \times 4$ Byte $\approx 102.8$ MB
  - mind. Bandbreite bei 4K-Auflösung (410 MB G-Buffer):
    $60$ Hz $\times$ $410$ MB $\times$ $2$ (Schreiben + 1-mal Beleuchten) $\geq 48$ GB/s
- zur Erzeugung von G-Buffers verwenden wir **FBOs**
  - leiten die Ausgabe in eigene „Off-Screen“ Buffer
  - Color Buffers: Texturen in die Farbwerte geschrieben werden
  - Renderbuffers: Tiefen- und Stencil-Buffer



# Frame Buffer Objects (FBOs)

## Schritt 1: Anlegen der Texturen für den eigenen Framebuffer

- erzeuge 4 Texturen im Format `GL_RGBA32F`, d.h. mit je 4 Floats pro Texel

```
// Erzeugen von 4 Texturobjekten
Gluint gbuffer[ 4 ];
glGenTextures( 4, gbuffer );

for ( int i = 0; i < 4; i++ ) {
  glBindTexture( GL_TEXTURE_2D, gbuffer[ i ] );

  // Dummy-Aufruf, damit OpenGL weiß welches Texturformat wir
  // anlegen möchten (alternativ z.B. auch GL_RGBA16F)
  glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA32F,
                width, height, 0, GL_RGBA, GL_FLOAT, NULL );
}
```

# Frame Buffer Objects (FBOs)

## Schritt 2: Renderbuffer und FBO anlegen

- als nächstes legen wir die OpenGL Objekte für FBOs und RBOs an

```
// jetzt erzeugen wir die Frame Buffer Objects...
glGenFramebuffers( 1, &frameBufferObject );
glBindFramebuffer( GL_FRAMEBUFFER, frameBufferObject );

// ...und verbinden sie mit den Texturen
for ( int i = 0; i < 4; i++ )
  glFramebufferTexture2D( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0 + i,
                          GL_TEXTURE_2D, gbuffer[ i ], 0 );
```

# Frame Buffer Objects (FBOs)

## Schritt 2: Renderbuffer und FBO anlegen

- als nächstes legen wir die OpenGL Objekte für FBOs und RBOs an

```
// jetzt erzeugen wir die Frame Buffer Objects...
glGenFramebuffers( 1, &frameBufferObject );
glBindFramebuffer( GL_FRAMEBUFFER, frameBufferObject );

// ...und verbinden sie mit den Texturen
for ( int i = 0; i < 4; i++ )
  glFramebufferTexture2D( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0 + i,
                          GL_TEXTURE_2D, gbuffer[ i ], 0 );


// jetzt Erzeugen wir noch das Renderbuffer Objekt
// (letztendlich unseren Tiefenpuffer)
Gluint renderBufferObject;
glGenRenderbuffers( 1, &renderBufferObject );
glBindRenderbuffer( GL_RENDERBUFFER, renderBufferObject );
glRenderbufferStorage( GL_RENDERBUFFER,
                       GL_DEPTH_COMPONENT24, width, height );

// ...und verbinden mit dem aktuell gewählten FBO
glFramebufferRenderbuffer( GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                           GL_RENDERBUFFER, renderBufferObject );
```

# Frame Buffer Objects (FBOs)

## Schritt 3: Rendering und Verwenden der G-Buffers

- …analog zu anderen OpenGL Objekten…

```
glGetIntegerv ( GL_DRAW_BUFFER, &curRenderTarget );

glBindFramebuffer( GL_FRAMEBUFFER, frameBufferObject );

const GLenum buffers[4] = {
     GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1,
     GL_COLOR_ATTACHMENT2, GL_COLOR_ATTACHMENT3 };
glDrawBuffers( 4, buffers );

drawScene(); // hier Zeichnen der Szene...

glDrawBuffers( 1, curRenderTarget );
glBindFramebuffer( GL_FRAMEBUFFER, 0 );

// jetzt können die G-Buffer Texturen verwendet werden, z.B. mit
glBindTexture( GL_TEXTURE_2D, gbuffer[0] ); ...
```

- nicht vergessen: Binding der Fragment-Shader Variablen

```
glBindFragDataLocation( shader, 0, "out_pos" );
glBindFragDataLocation( shader, 1, "out_nrml" );
```

```
out vec4 out_pos, out_nrml;
void main() { ... out_pos = ...; out_nrml = ...; ... }
```

# Deferred Shading

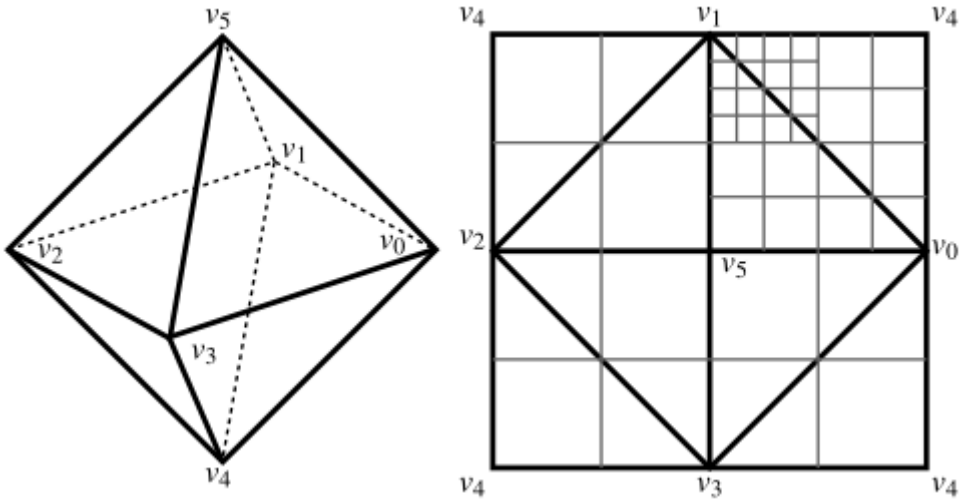
## Performance Aspekte, Optimierungen

- Optimierungspotenzial bei den **G-Buffers**: Reduktion der Daten
  - keine höhere Genauigkeit als nötig, z.B. müssen weder diffuse Farbe noch Normalen in Floating Point Genauigkeit gespeichert werden
  - Eliminieren von redundanten und leicht zu berechnenden Daten (z.B. Normale aus dem Tiefenpuffer? so weit wollen wir nicht gehen…)
  - resultiert in weniger Bandbreite (bei Erzeugung und Lesen der G-Buffer) und evtl. weniger simultanen Render-Targets
    → beides erhöht die Rendering-Performanz

- Optimierung der **Beleuchtungsphase**
  - reduziere Anzahl der Beleuchtungsberechnungen
  - Ausnutzen von Materialeigenschaften

# Optimierungen: Geometriephase (Beispiel 1)

- Speicherung der Normale $\boldsymbol{n} = \left(n_x, n_y, n_z\right)^T$ mit $|\boldsymbol{n}| = 1$
  - Darstellung über sphärische Koordinaten (2 Winkel): Umwandlung erfordert (vergleichsweise teure) trigonometrische Funktionen
  - Darstellung über 2D- Koordinaten einer Oktaeder-Parametrisierung



# Optimierungen: Geometriephase (Beispiel 1)

- Speicherung der Normale $\boldsymbol{n} = (n_x, n_y, n_z)^T$ mit $|\boldsymbol{n}| = 1$
  - Darstellung über sphärische Koordinaten (2 Winkel): Umwandlung erfordert (vergleichsweise teure) trigonometrische Funktionen
  - Darstellung über 2D- Koordinaten einer Oktaeder-Parametrisierung
  - bei der Beleuchtungsberechnung in Kamerakoordinaten gilt:
    - sichtbare Flächen zeigen natürlich zum Betrachter hin
    - die $z$-Komponente der Normale ist also immer positiv

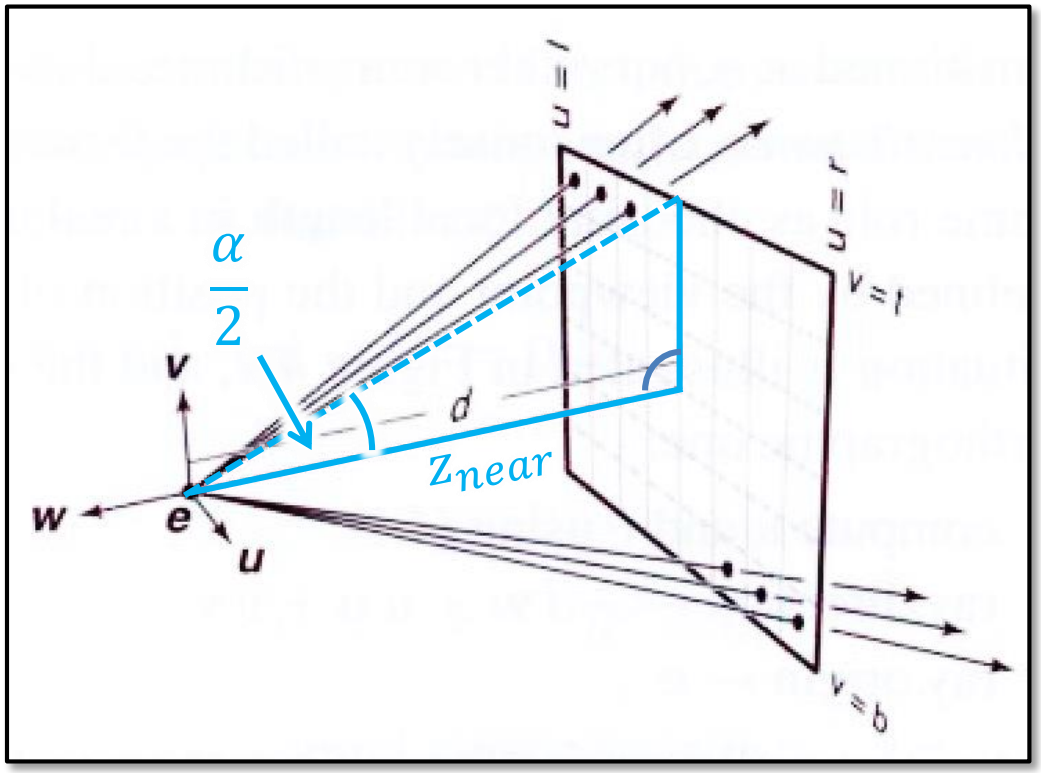
      $\Rightarrow n_z = \left(1 - {n_x}^2 - {n_y}^2\right)^{-1/2}$

- weitere Optimierung: $n_x$ und $n_y$ erfordern keine FP-Genauigkeit
  - es ist möglich Werte zu packen, z.B. 4 Byte-Werte in einen 32 Bit Float zu speichern (oder 1 Float in einem 32-Bit-RGBA-Quadrupel)
  - Berechnung entweder durch Multiplikation und **fract**
    http://smt565.blogspot.com/2011/04/bit-packing-depth-and-normals.html
  - oder mittels GLSL >3.3 Befehlen: **float intBitsToFloat( uint x );**
    siehe: http://www.opengl.org/sdk/docs/manglsl/xhtml/intBitsToFloat.xml

# Optimierungen: Geometriephase (Beispiel 2)

## Berechnung der Position aus Tiefe und Kamera

- es besteht also keine Notwendigkeit die Position zu speichern
- die Position eines Pixels in Weltkoordinaten ist eindeutig bestimmt durch
  - die Kameraabbildung (Öffnungswinkel $\alpha$, Aspect Ratio $^{r}/_{t}$)
  - seinen Tiefenwert (= Entfernung zur Kamera) und
  - die Pixel-Koordinate im G-Buffer



# Optimierungen: Beleuchtungsphase

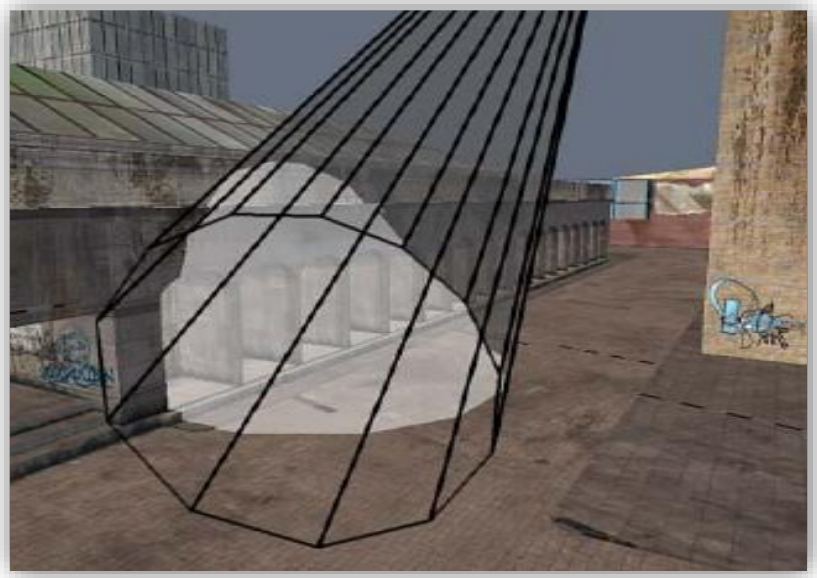
## Performanzaspekte bei der Beleuchtung

- Kosten für additives Blending (Akkumulation der Lichtquellenbeiträge)
  - erfordert Floating Point Framebuffer (weiterer FBO): Beiträge einzelner Lichtquellen können zu klein für 8-Bit-Blending sein
  - erhöhte Speicherbandbreite heute kein allzu großes Problem
  - für physikalisch-basierte Beleuchtungsmodelle und Postprocessing ohnehin unerlässlich
- Kosten der Beleuchtungsberechnung
  - hängen vom Beleuchtungsmodell/BRDF ab (Phong, Cook-Torrance, … )
  - sind **proportional zur Anzahl der beleuchteten Pixel**
- bisher: „globale“ Lichtquellen, die gesamte Szene/Bild beeinflussen durch Zeichnen eines bildschirmfüllenden Rechtecks
  (das Ziel war für jeden Pixel ein Fragment-Programm auszuführen)

# Optimierungen: Beleuchtungsphase

## Idee: Lokale Lichtquellen

- oft haben Lichtquellen nur Einfluss auf einen Teil des Bildes
  - z.B. Einfluss einer Punktlichtquelle im Abstand $r$ ist $\propto 1/r^2$, ab einer bestimmten Entfernung ist der Beitrag vernachlässigbar
- Einflussbereiche lassen sich mit Hüllkörpern beschreiben
  - Punktlichtquelle → Kugel
  - Spot Light → Kegel
  - direktionale Lichtquelle → Quader
- Idee: ein Fragment-Programm soll nur im Einflussbereich ausgeführt werden
  - zeichne dazu die Hüllkörper (mit Kameraabb. und Tiefentest) anstatt eines bildschirmfüllenden Rechtecks

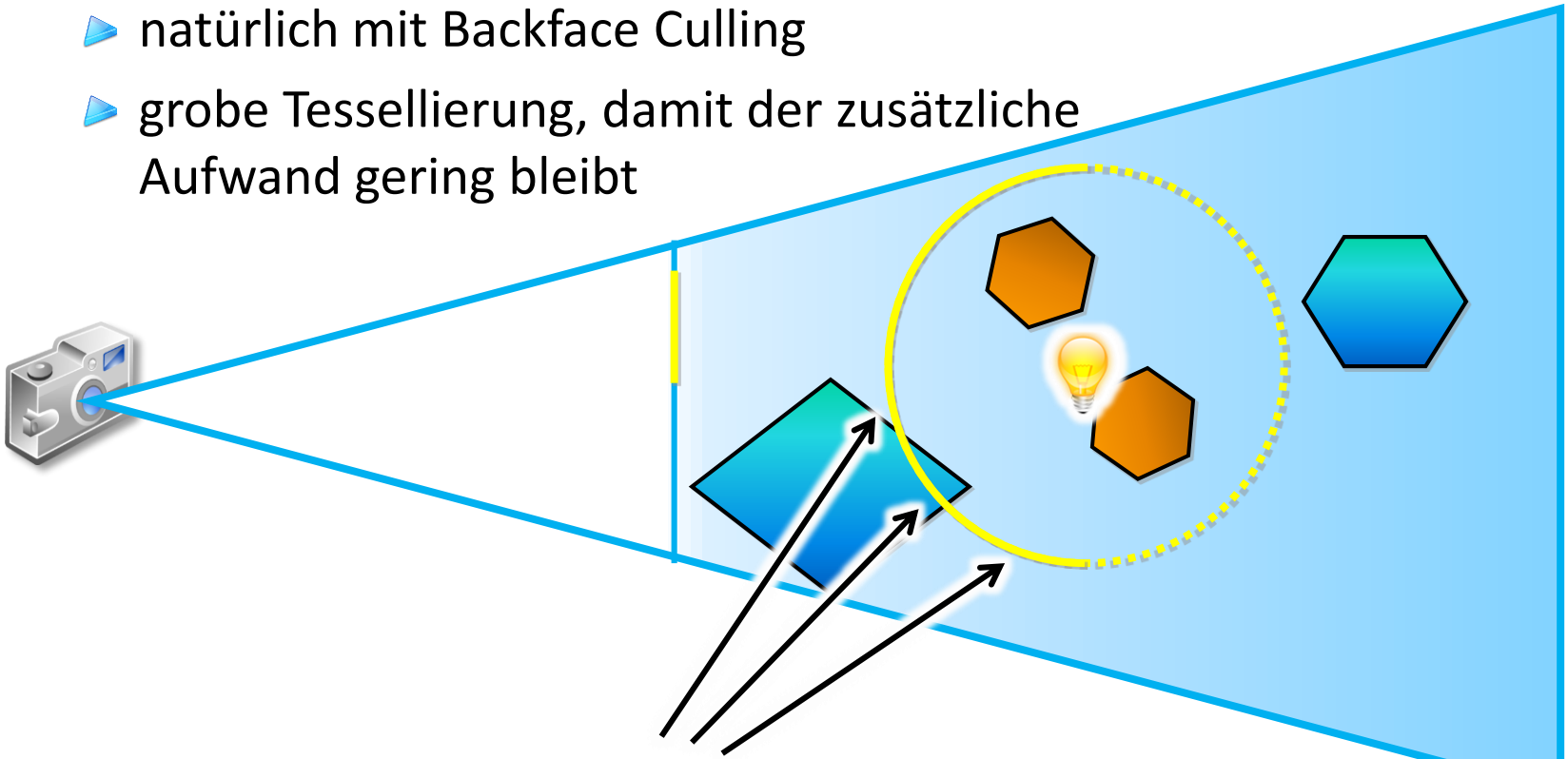


Shawn Hargreaves, GDC2004

# Optimierung durch Hüllkörper

## Beispiel: Punktlichtquelle

- zeichne eine Kugel anstatt eines Rechtecks über das ganze Bild
  - natürlich mit Backface Culling
  - grobe Tessellierung, damit der zusätzliche Aufwand gering bleibt

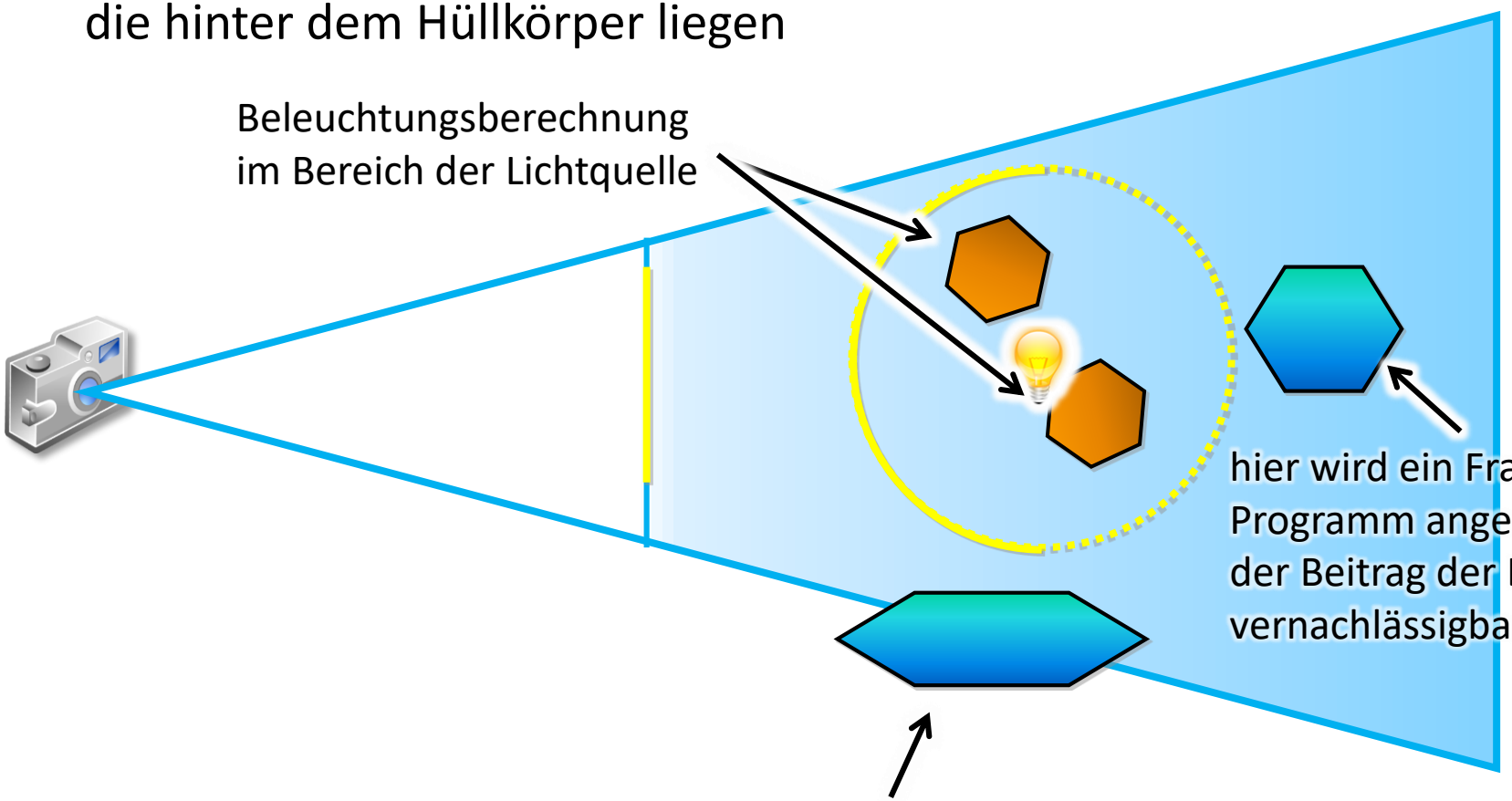


durch den Tiefentest werden für verdeckte Teile der Hüllkörper keine Beleuchtungsberechnungen durchgeführt

# Optimierung durch Hüllkörper

## Beispiel: Punktlichtquelle

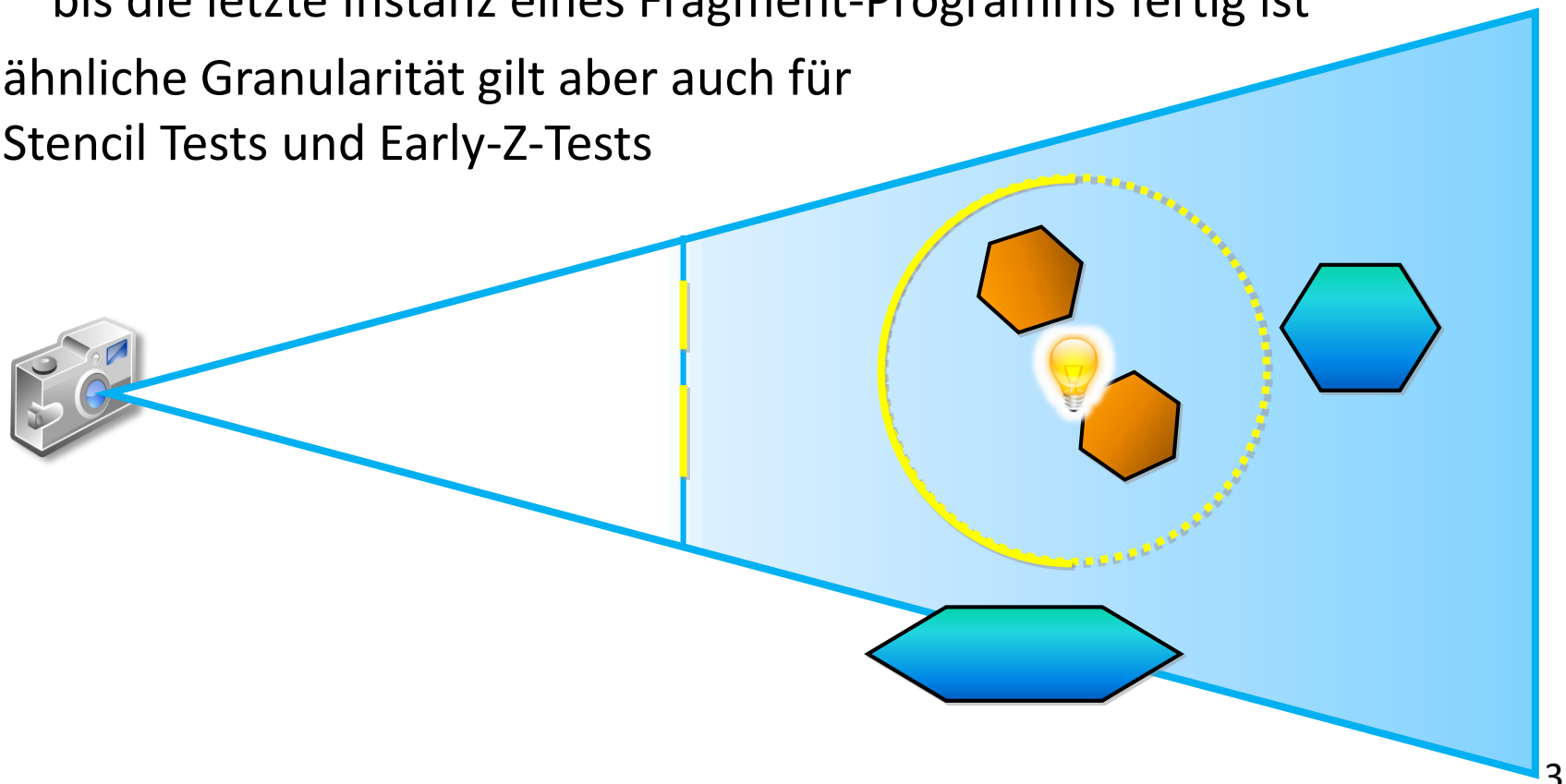
- es gibt noch weiteres Optimierungspotenzial bei Szenenteilen, die hinter dem Hüllkörper liegen



# Optimierung durch Hüllkörper und „Discard“

## Discard in Fragment Programmen

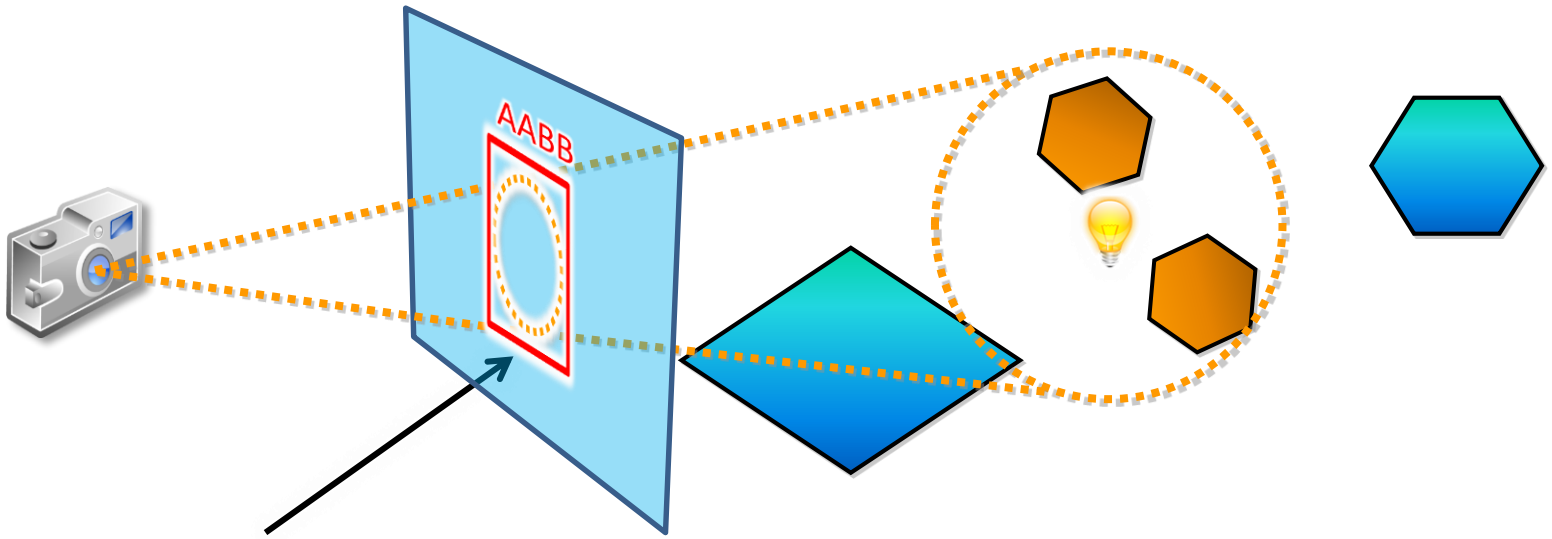
- `discard`-Befehl (nur in Fragment Programmen) beendet die Ausführung: verwerfe Fragmente bei zu großem Tiefenwert
- Achtung: FP werden immer in Blöcken von $4^2$, $8^2$, … Pixel ausgeführt
  - Threads starten zum gleichen Zeitpunkt und sind alle blockiert, bis die letzte Instanz eines Fragment-Programms fertig ist
- ähnliche Granularität gilt aber auch für Stencil Tests und Early-Z-Tests



# Optimierung durch 2D-Hüllkörper

## Beispiel: Punktlichtquelle

- oft werden auch 2D-Hüllkörper **im Bildraum** verwendet, z.B. AABBs
- Vorteil: geringer Aufwand in der Geometrieverarbeitung, einfach zu berechnen und nur wenig unnötig rasterisierte Fläche

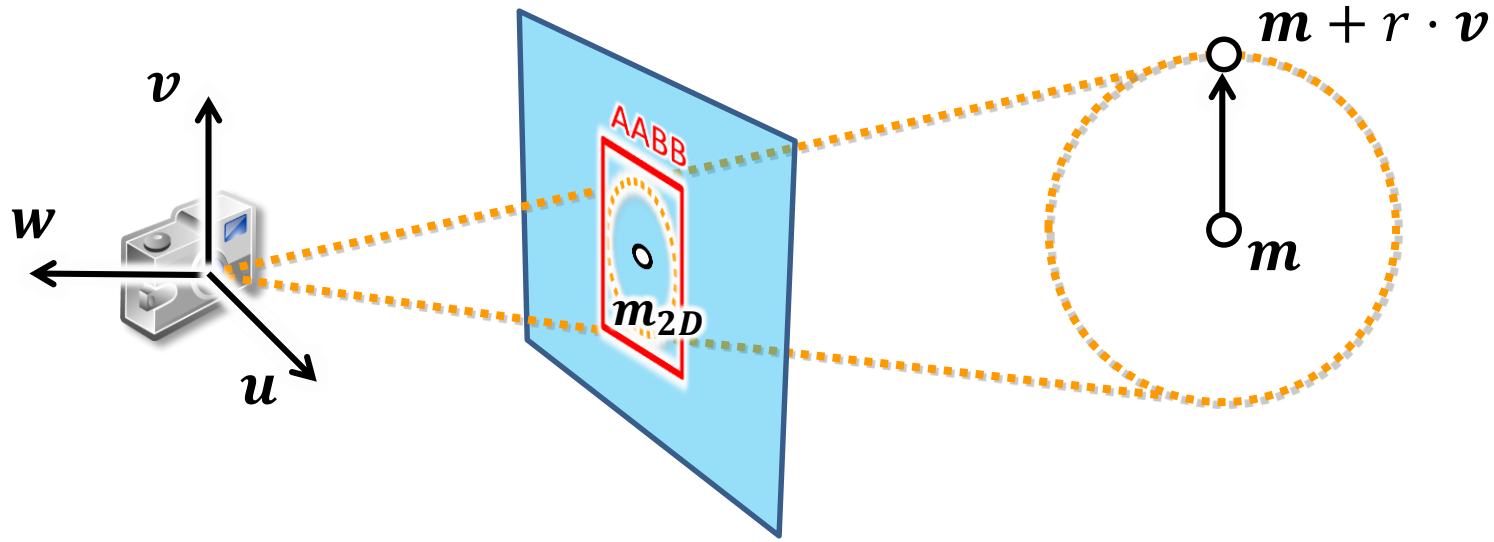



Beleuchtungsberechnung im Einflussbereich der LQ wird durch Zeichnen eines Rechtecks angestoßen (Zeichnen ohne Tiefentest, oder mit konservativ minimalem Tiefenwert)

# Optimierung durch 2D-Hüllkörper

**Beispiel: Berechnung einer AABB einer Kugel (nicht korrekt)**

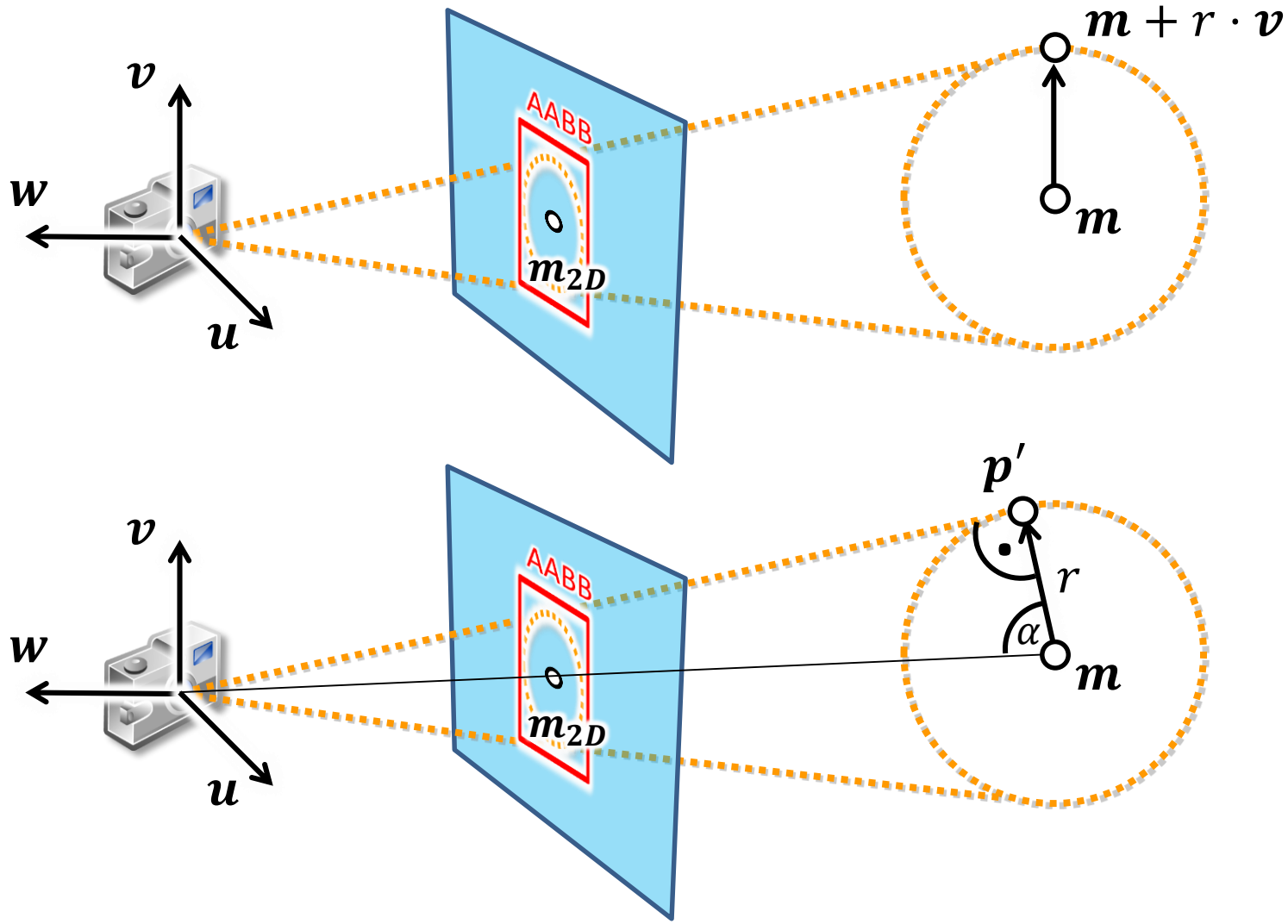
- Idee: projiziere Mittelpunkt und Punkt auf der Silhouette auf Bildebene
- allg. mit dem Kamerakoordinatensystem $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$ kann man Flächen senkrecht/parallel zur Blickrichtung (in Weltkoord.) aufspannen
- geg. Mittelpkt. $\boldsymbol{m}$ in homogenen Koord., View-Projection-Matrix $\mathbf{M_{VP}}$
  - Projektion $\boldsymbol{m_{2D}} = \mathbf{M_{VP}} \cdot \boldsymbol{m}$
  - Punkt auf dem Rand (Radius $r$): $\boldsymbol{p} = \boldsymbol{m} + r \cdot \boldsymbol{v}$ und $\boldsymbol{p_{2D}} = \mathbf{M_{VP}} \cdot \boldsymbol{p}$
  - Kantenlänge der AABB (zentriert um $\boldsymbol{m_{2D}}$): $2|\boldsymbol{p_{2D}} - \boldsymbol{m_{2D}}|$
  - Dehomogenisieren nicht vergessen



# Optimierung durch 2D-Hüllkörper

## Beispiel: Berechnung einer AABB einer Kugel

- Punkt $\boldsymbol{p}$ auf dem Rand ist i.A. nicht der Punkt $\boldsymbol{p}'$ auf der Silhouette



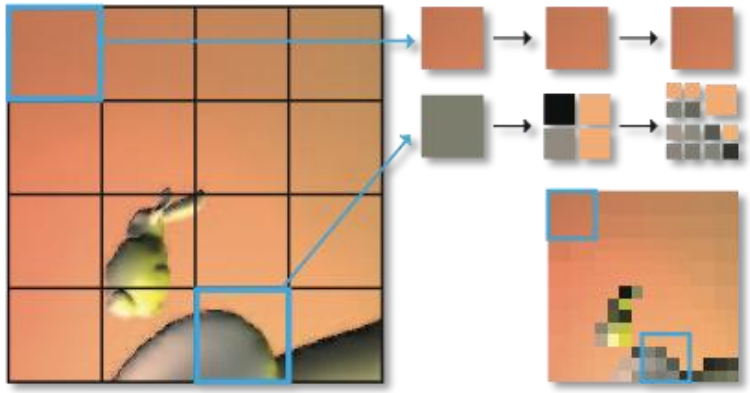
# Schatten

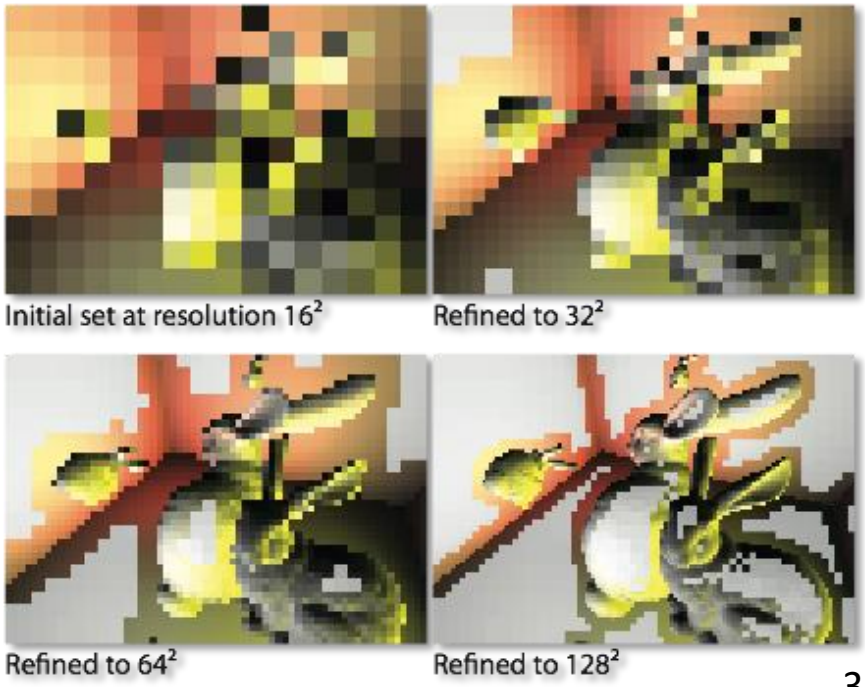
- mit Deferred Shading kann eine große Zahl von Lichtquellen verwendet werden – das Erzeugen der Shadow Maps ist nach wie vor aufwändig
  - Shadow Volumes und werden nicht mit Deferred Shading verwendet: würde prinzipiell funktionieren, aber die Grundidee „Entkoppeln von Geometrie und Beleuchtung“ gilt dann nicht mehr
- oft werfen nur die hellsten/wichtigsten Lichtquellen Schatten
  - dann wird das Resultat des Schattentests u.U. im G-Buffer gespeichert (Beispiel: CryEngine 2)
- Alternative für zu viele Lichtquellen z.B. Voxelisierung und Ray Marching

# Ausblick: Adaptive Verfeinerung (Nichols and Wyman)

**Weniger Shading-Berechnungen durch eine Auflösungshierarchie**

- erzeuge **Min-Max-Mipmap** des Tiefenpuffers zur Detektion von Diskontinuitäten und verwende diese für das Shading:
  berechne Beiträge einer Lichtquelle in grober Auflösung, wenn keine großen Diskontinuitäten in diesem Bereich
- am Ende: Interpolation und Compositing der Bilder
- wie gut kann das funktionieren?



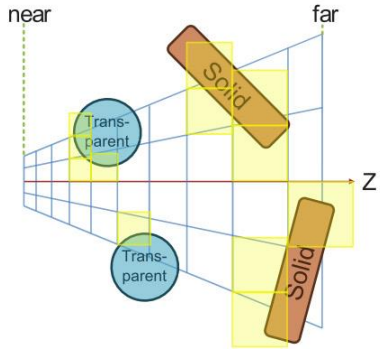


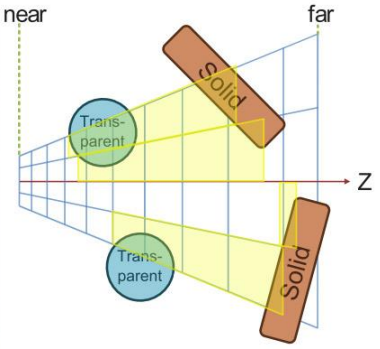
# Ausblick: Tiled and Clustered Shading (+ Varianten)

## Beschleunigung von Forward Shading und Deferred Shading

http://www.cse.chalmers.se/~olaolss/

- erzeuge eine räumliche Hierarchie von Punktlichtquellen
- unterteile Sichtpyramide im Bildraum (2D-Tiles) und in der Tiefe
- bestimme Cluster von Lichtquellen, die für eine Zelle benötigt werden
- Betrachtung von Geometrie-Clustern ermöglicht Backface-Culling von LQ
- darauf aufbauend: Shadow Mapping für viele Lichtquellen



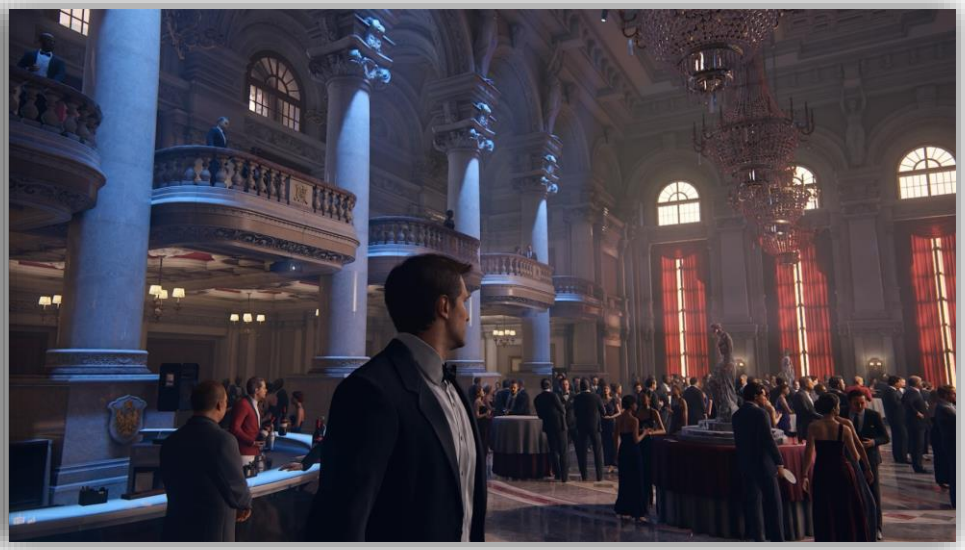



# Materialvielfalt: #Lichtquellen vs. #Materialien

- auch Deferred Shading „leidet“ unter der Vielfalt von Lichtquellen und Materialkombinationen
  - Ausführen eines Fragment-Programms pro Lichtquelle, nicht pro Material
  - d.h. im G-Buffer müssen Materialinformationen gespeichert sein, führt potentiell zu vielen Daten pro Pixel

- mögliche Lösungen
  - Multi-Pass: speichere Materialindex und zeichne jeden Hüllkörper jeder Lichtquelle einmal pro Materialtyp – Daten im G-Buffer werden dann entsprechend interpretiert → zu viele Rendering-Durchgänge
  - besser: speichere den Materialindex (ID) im G-Buffer und lese die Materialparameter aus einer Tabelle aus (mit Verzeigung im Fragment-Programm, oft bezeichnet als Uber-Shader)

# Fallbeispiel: Uncharted 4



# Fallbeispiel: Uncharted 4

## G-Buffer Layout

- 16 Bits (unsigned short) pro Kanal pro Pixel

GBuffer 0

| | | |
|---|---|---|
| R | r | g |
| G | b | spec |
| B | normalx | normaly |
| A | iblUseParent | normalExtra | roughness |

GBuffer 1

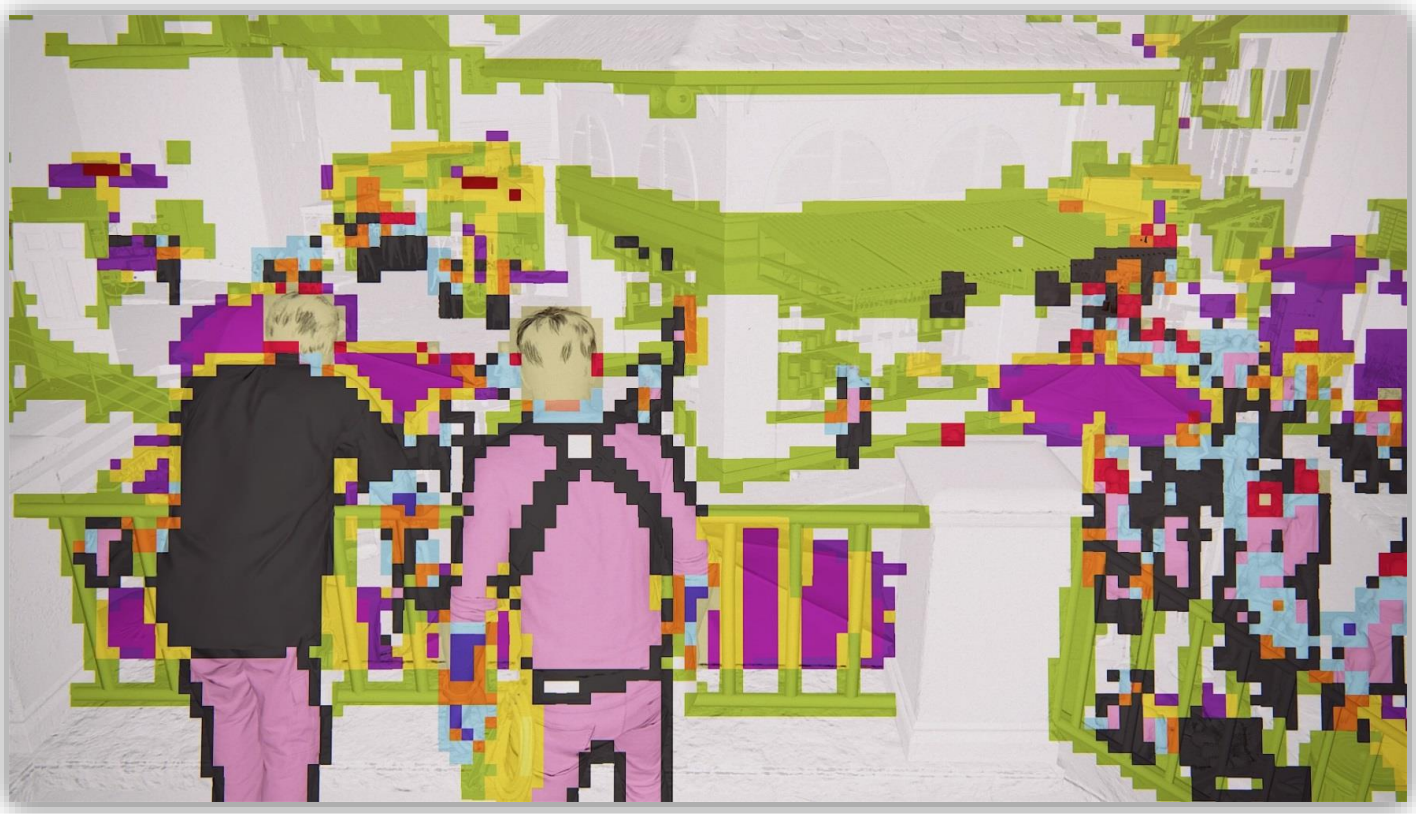
| | | | | |
|---|---|---|---|---|
| R | ambientTranslucency | sunShadowHigh | | specOcclusion |
| G | heightmapShadowing | | sunShadowLow | metallic |
| B | dominantDirectionX | | | dominantDirectionY |
| A | ao | extraMaterialMask | sheen | thinWallTranslucency |

- optional: dritter G-Buffer für Materialien wie Textilien, Haare, Haut, … die sich gegenseitig ausschließen
  - wird nur gelesen/geschrieben, wenn er auch verwendet wird
- keine Material-ID sondern eine Bitmaske: welche Shader-Features werden benötigt?
  - bestimme benötigte Features jeweils für $16 \times 16$ Kacheln
  - über eine Look-Up Tabelle werden Shader (aus einer Bibliothek für alle Kombinationen!) ausgewählt und Shading für die Kacheln berechnet
  - weitere Optimierungen, wenn alle Pixel dasselbe Material haben

# Fallbeispiel: Uncharted 4

**Performanz in einer komplexen Szene**

- 4.0ms ohne Optimierung (Uber Shader)
- 3.4ms (-15%) durch Auswahl des besten Shaders
- 2.7ms (-20% bis -30%) durch Shader ohne Verzweigungen (Optimierung, wenn alle Pixel daselbe Material aufweisen)



# Deferred Shading

**Weitere Aspekte auf dem Weg zu einer „kompletten“ Rendering-Technik**

- Transparenz (immer schwierig bei Rasterisierung)
- Antialiasing
- realistischeres Shading durch Ausnutzen der G-Buffer
  - Ambient Occlusion, indirekte Beleuchtung, Reflexionen, …

# Transparente Flächen

- transparente Flächen können mit Deferred Shading, wie wir es bisher kennengelernt haben, nicht gezeichnet werden:
  wir speichern eben nur die Informationen über die erste sichtbare Fläche

- transparente Objekte werden daher manchmal zuletzt gezeichnet
  - Verwendung von Blending wie bei normalem Forward-Rendering
  - Sortierung von hinten nach vorne
  - „normale“ Beleuchtungsstrategie
  - verwende Tiefenpuffer des G-Buffers

- im Prinzip sind aber „mehrschichtige“ G-Buffer möglich
  - natürlich mit höherem Speicherbedarf
  - Erzeugen durch Depth-Peeling- oder Per-Pixel Linked Lists-Verfahren
  - Depth-Peeling einfacher zu implementieren und u.U. effizienter bei geringer Tiefenkomplexität und großen transparenten Objekten

# Transparente Flächen

## Anwendungen, Ziele

- Transparenz ohne Sortieren der Dreiecke
  (wir erzeugen die Fragmente in tiefensortierter Reihenfolge)
- Transluzenz-Effekte durch Berücksichtigung der Abstände zwischen den Schichtbildern

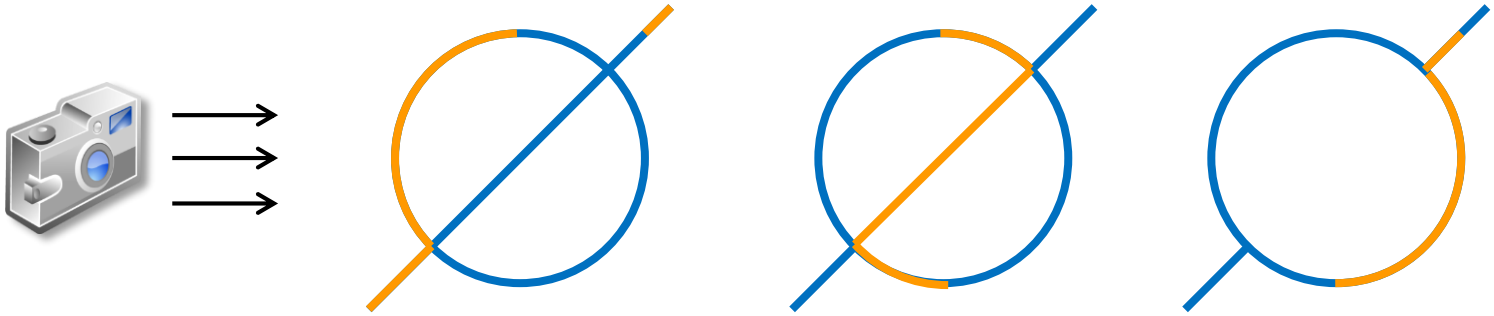
# Depth Peeling

- Idee: extrahiere Schichtbilder von den Oberflächen von der Kamera aus gesehen

erste sichtbare Oberfläche

zweite, dahinter liegende Oberfläche

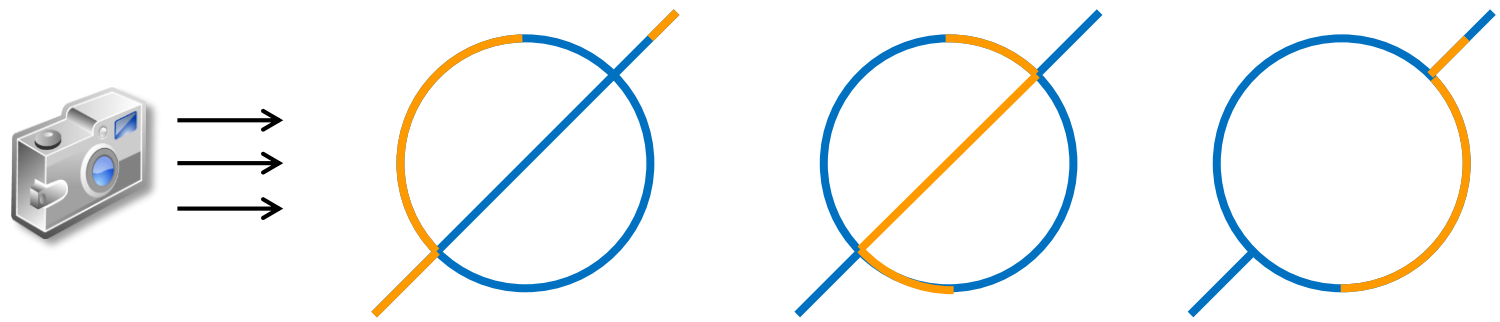
- schematische Darstellung: 3 Tiefenschichten



# Depth Peeling

## Erzeugung/Rendering

- erster Render-Durchgang mit normalem Tiefentest (kleiner-gleich)
  - der Tiefenpuffer wird in einem FBO aufbewahrt
- weitere Render-Durchgänge
  - verwende normalen Tiefentest (kleiner-gleich)
  - verwende **zusätzlichen Tiefentest** (in einem Fragment-Programm), der alle Fragmente mit einer Tiefe verwirft, deren Tiefe nicht größer als die im Tiefenpuffer aus dem vorherigen Durchgang ist



- wie viele Durchgänge? Bis keine Fragmente mehr gezeichnet werden!
  - Test möglich über sog. Occlusion Queries (mehr später)

# Per-Pixel Linked Lists

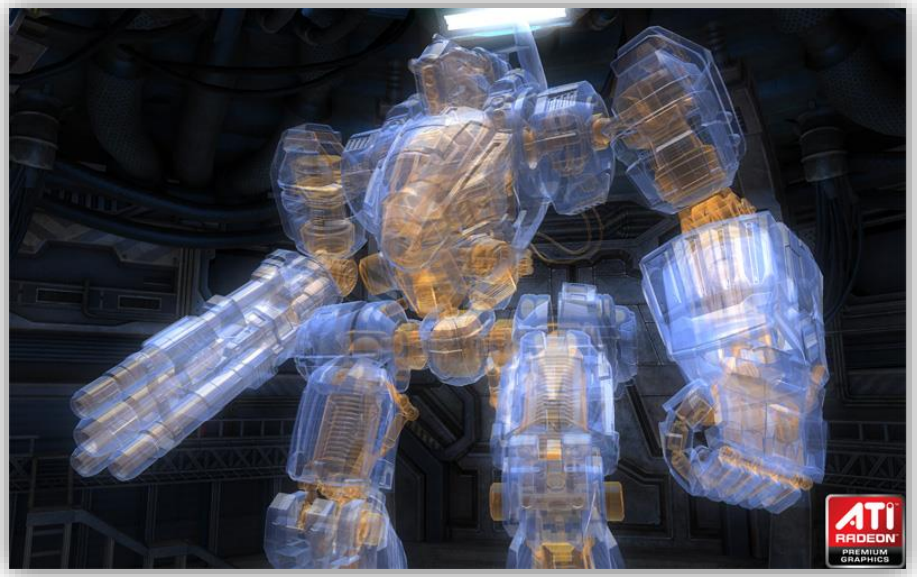
## Transparenz ohne Sortierung der Geometrie

Real-Time Concurrent Linked List Construction on the GPU, Yang et al.

- Ziel: vermeide Sortierung von transparenten Polygonen, reduziere Speicherbedarf und Render-Passes (im Vgl. zu Depth Peeling)

- weitere Anwendungen: Shadow Maps mit semi-transparenten Flächen, Deep Shadow Maps, Voxelisierung, …

- weitere Arbeiten in diesem Kontext: „k-Buffer“-Arbeiten (Bavoil et al.), Hashing auf GPUs, …




https://www.youtube.com/watch?v=j_I1fTG-sSo

Quelle der Folien: Advances in Real-Time Rendering (http://advances.realtimerendering.com/s2010/index.html)

Implementationsdetails: http://blog.icare3d.org/2010/07/opengl-40-abuffer-v20-linked-lists-of.html

# Per-Pixel Linked Lists

**Transparenz ohne Sortierung der Geometrie**

- Schritt 1: zeichne opake Geometrie (wie üblich)
- Schritt 2: Rendering transparenter Objekte
  - speichere Fragmente in einer verketteten Liste für jeden Pixel
  - speichere pro Fragment: Farbe, Alpha, Tiefe
  - hierzu werden ein atomarer Zähler und 2 Puffer benötigt
    - **Node Buffer**: Puffer zum Speichern der Fragmente selbst
    - **Head Pointer Buffer**: ein Integer pro Pixel, der auf das letzte Element der verketteten Liste zeigt, also Index in den Node Buffer
- Schritt 3: Pro-Pixel Compositing

# Per-Pixel Linked Lists

## Erzeugung der verketteten Listen

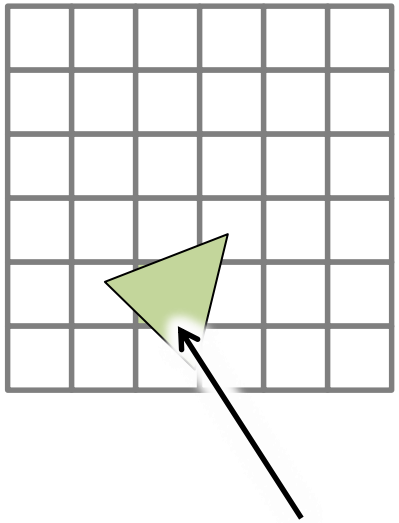
Head Pointer Buffer
(Auflösung wie Render Target)

| | | | | | |
|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |

**globaler, atomarer Zähler**

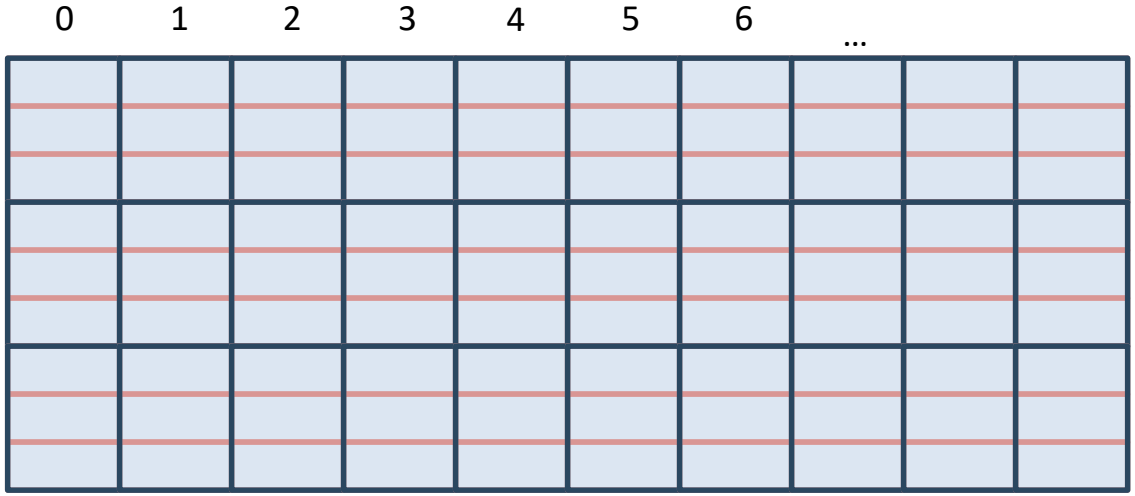
Counter = 0

Render Target



opake Geometrie
aus Schritt 1

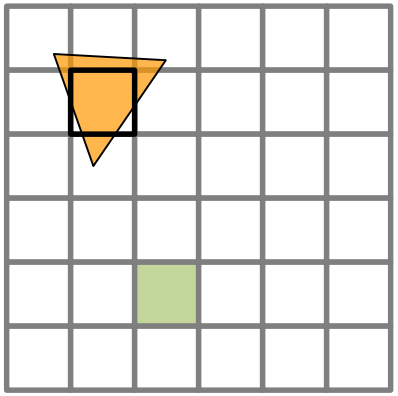
Node Buffer



# Per-Pixel Linked Lists

## Erzeugung der verketteten Listen

Render Target



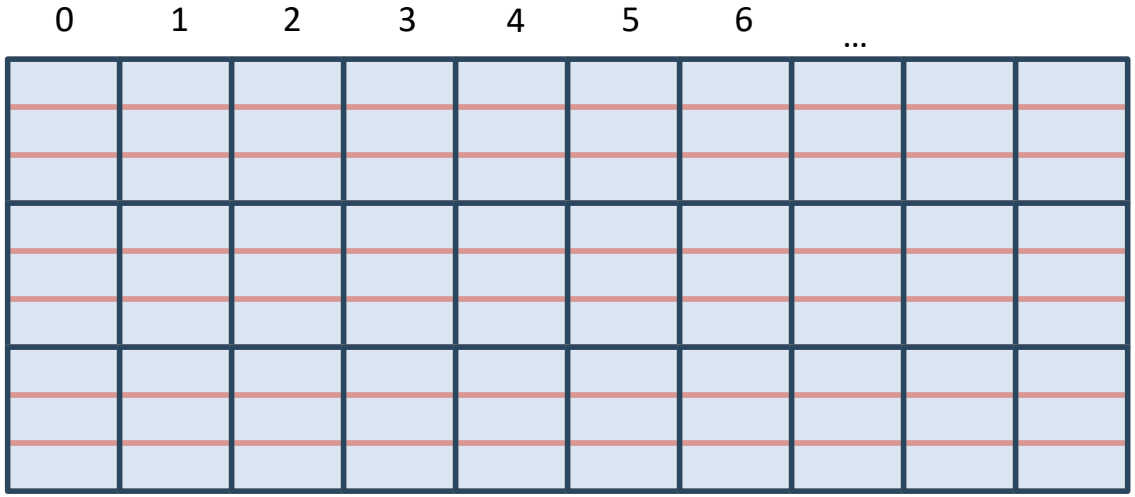
Head Pointer Buffer
(Auflösung wie Render Target)

| | | | | | |
|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |

**globaler, atomarer Zähler**

**Counter = 0**

Node Buffer



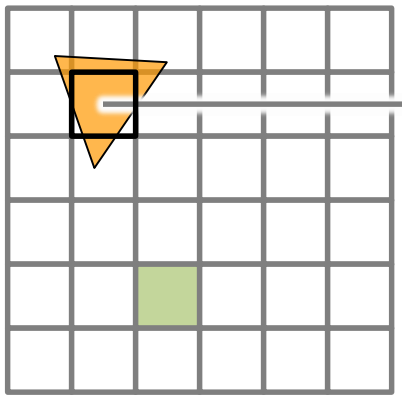
# Per-Pixel Linked Lists

## Erzeugung der verketteten Listen

Head Pointer Buffer
(Auflösung wie Render Target)

| | | | | | |
|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | 0 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |

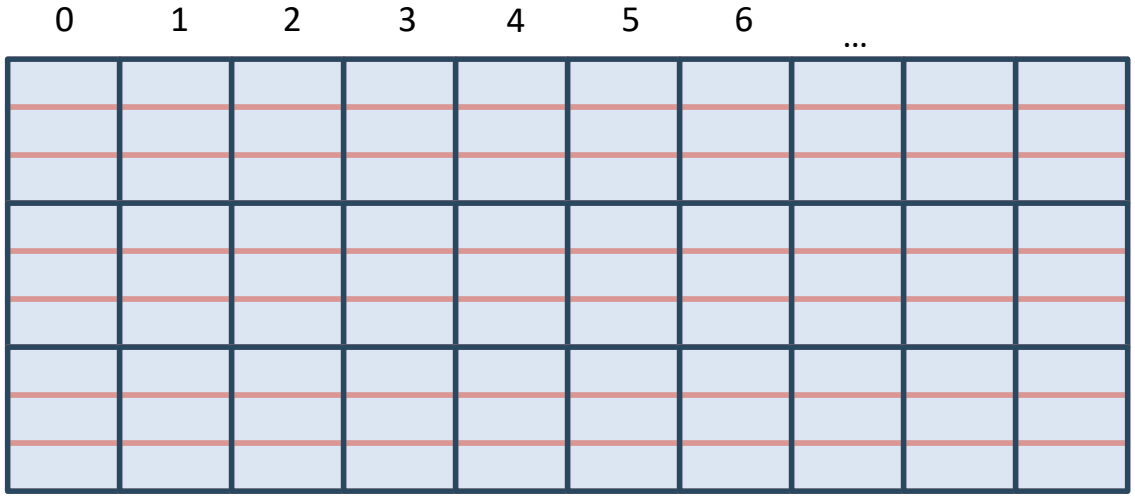
Render Target



**globaler, atomarer Zähler**

**Counter = 1**

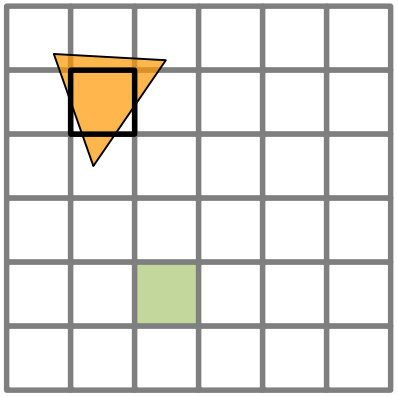
Node Buffer



# Per-Pixel Linked Lists

## Erzeugung der verketteten Listen

Render Target



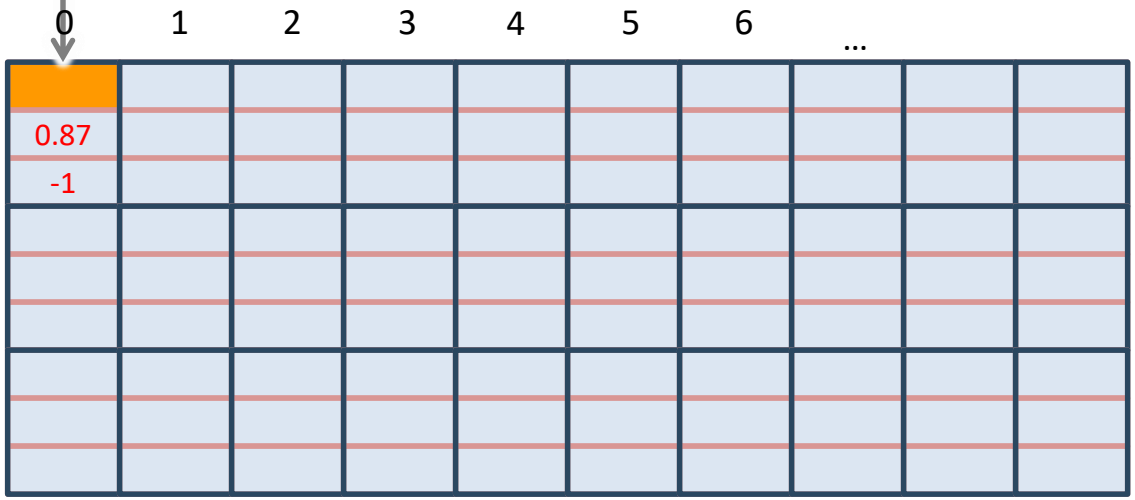
Head Pointer Buffer
(Auflösung wie Render Target)

| | | | | | |
|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | 0 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |

**globaler, atomarer Zähler**

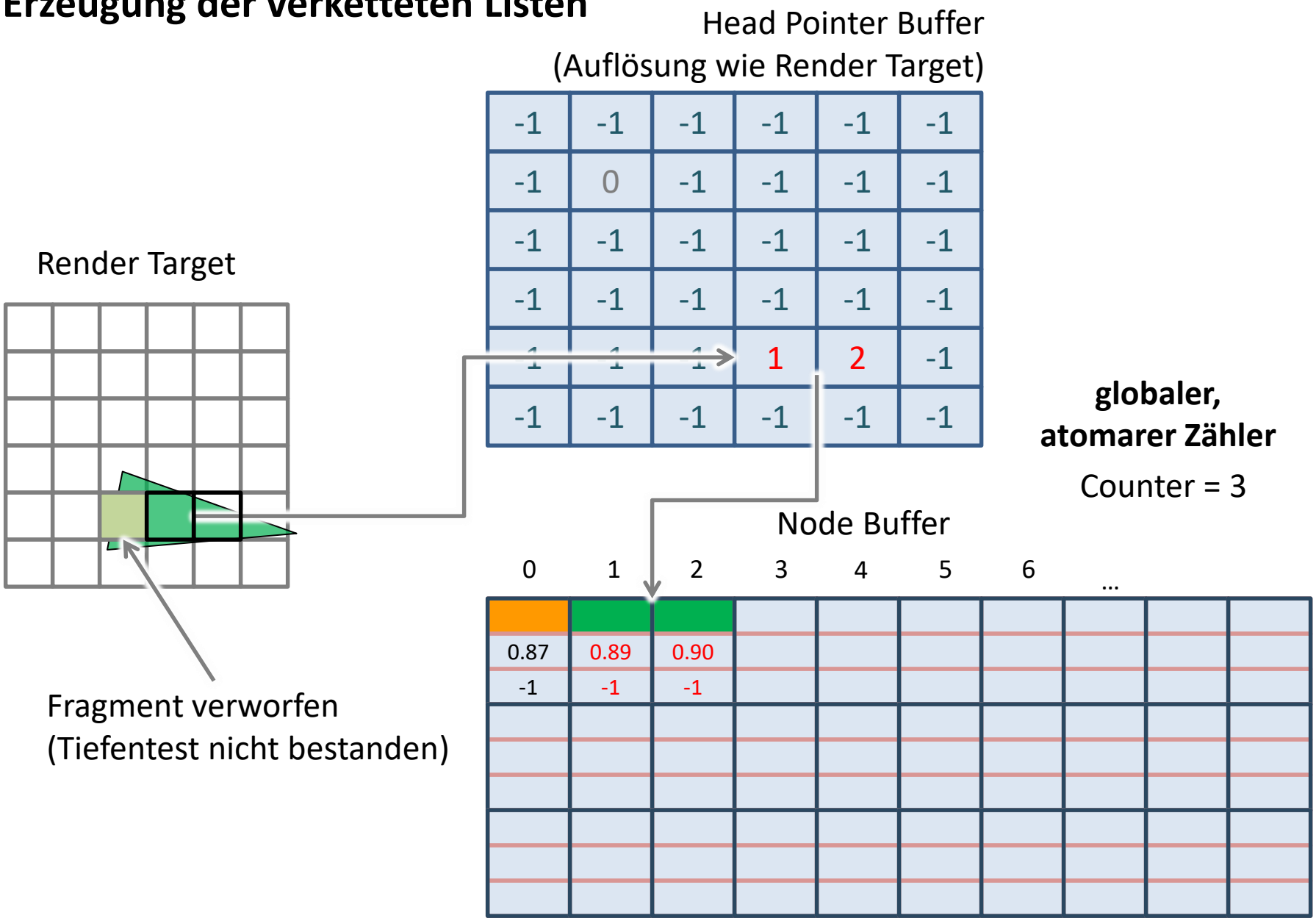
Counter = 1

Node Buffer



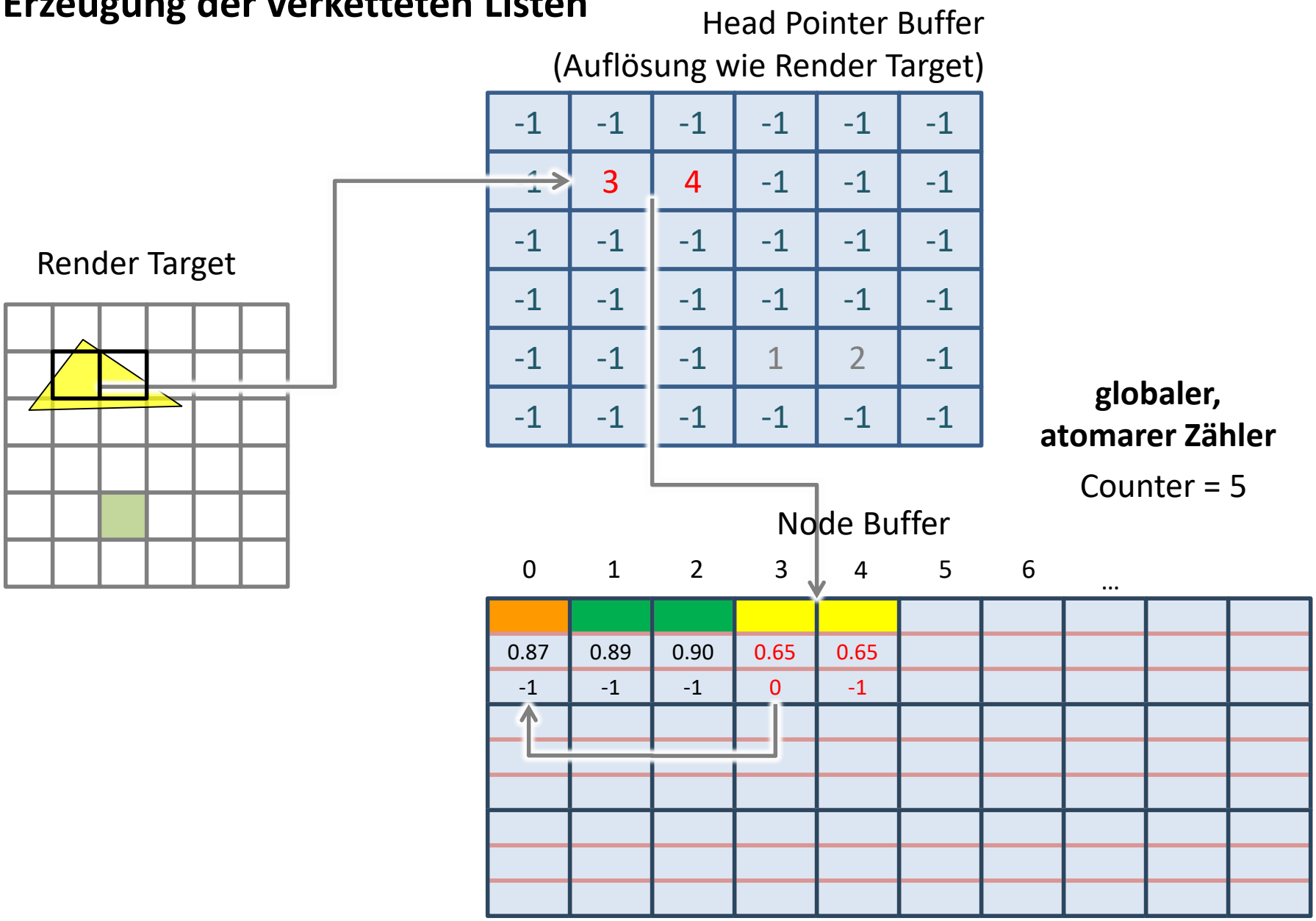
# Per-Pixel Linked Lists

## Erzeugung der verketteten Listen



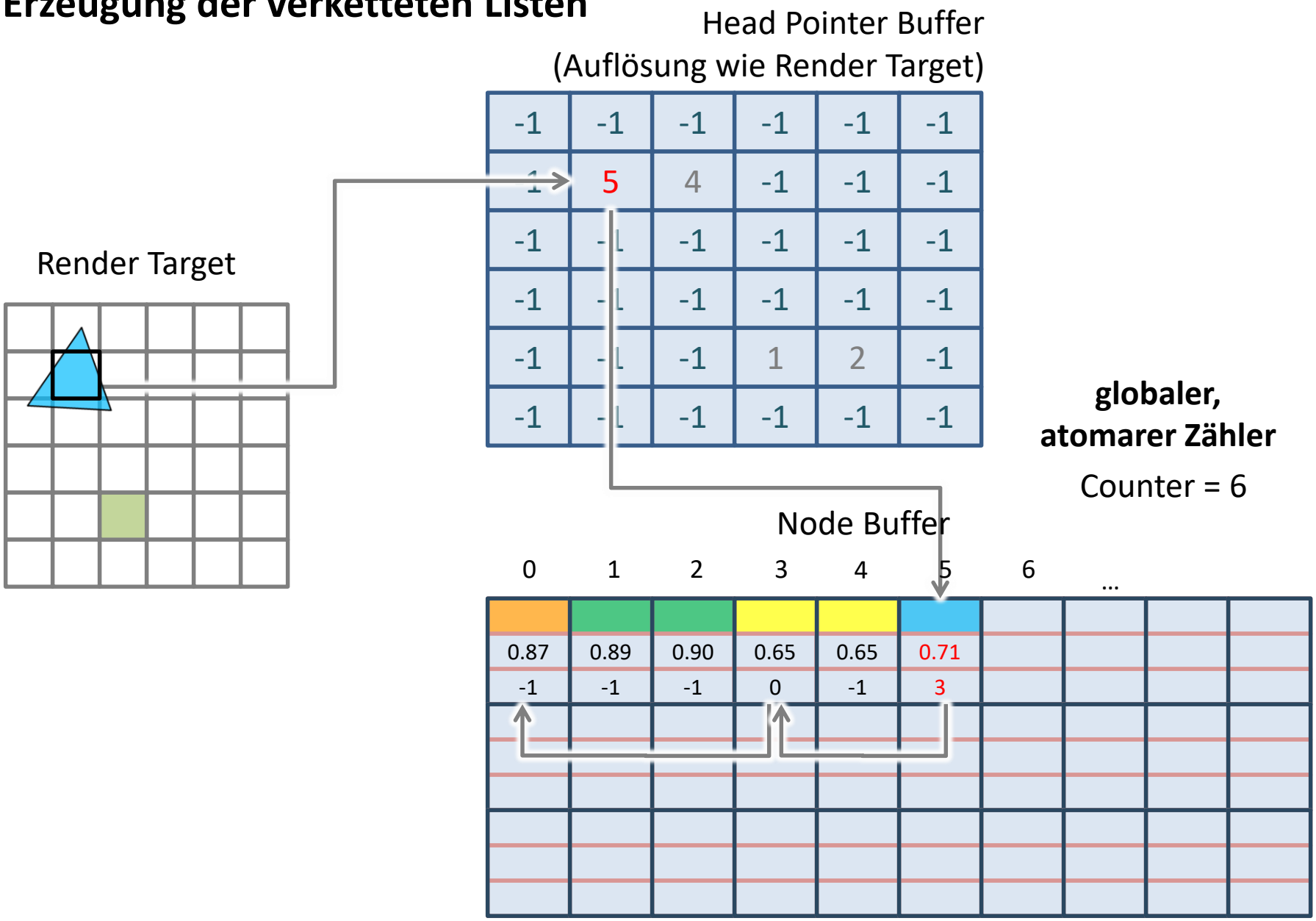
# Per-Pixel Linked Lists

## Erzeugung der verketteten Listen



# Per-Pixel Linked Lists

## Erzeugung der verketteten Listen



# Per-Pixel Linked Lists

## Benötigte OpenGL-Funktionalität

- atomare Zähler (Vorsicht: diese sind je nach GPU langsam)
  - unsigned int, Inkrementieren/Dekrementieren
    **uniform atomic_uint counter; …**
    **atomicCounterIncrement( counter );**
  - keine explizite Synchronisation notwendig
  - http://www.opengl.org/wiki/Atomic_Counter

- Shader Storage Buffer Object (SSBO)
  - (große) Speicherblöcke mit wahlfreien und atomaren Zugriffen
  - z.B. **int atomicExchange( inout int mem, int data );**

- wie wählt man die Größe des Node Buffers?
  - Größe = Anzahl transparenter Fragmente, die den Tiefentest bestehen
  - Bestimmung wieder möglich mit den sog. Occlusion Queries (siehe späteres Kapitel)
  - meist: konservative Abschätzung

# Per-Pixel Linked Lists

**Beispielanwendung: Transparenz ohne Sortierung der Geometrie**

- Schritt 1: zeichne opake Geometrie
- Schritt 2: Rendering transparenter Objekte
- Schritt 3: Pro-Pixel Compositing
  - Zeichnen eines bildschirmfüllenden Rechtecks (oder konservativ großen Hüllkörper) zum Anstoßen eines Fragment Programms
  - Sortierung der verketteten Listen pro Pixel (z.B. Insertion Sort)
  - Compositing der sortierten Fragmente, Blending mit Hintergrundfarbe

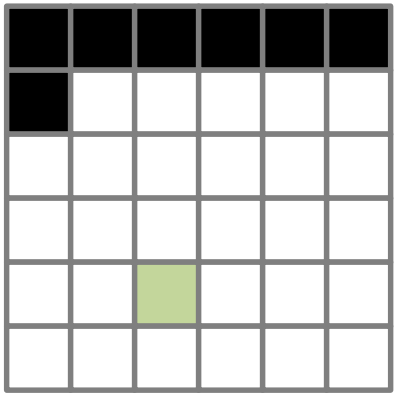
# Per-Pixel Linked Lists

## Compositing, Rendering der Fragmente

Head Pointer Buffer

| | | | | | |
|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | 5 | 4 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | 1 | 2 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |

Render Target



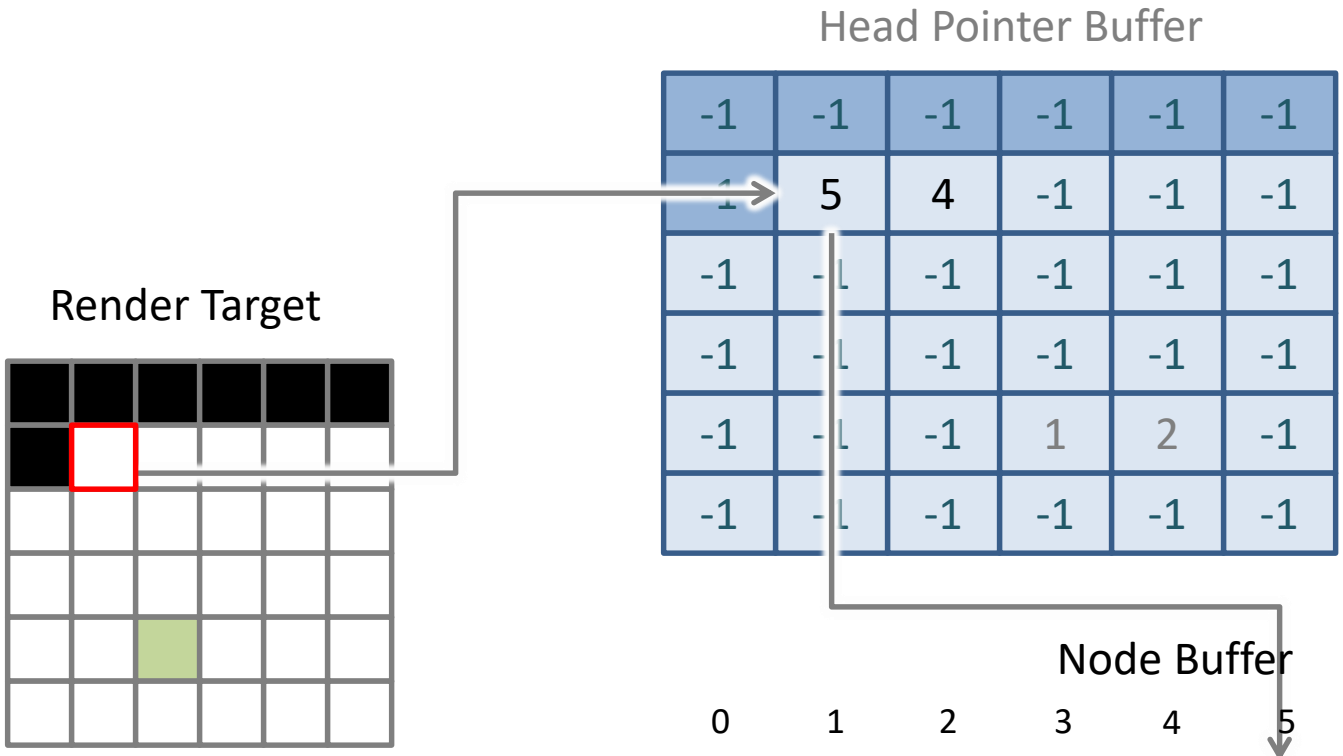
(0,0)->(1,1):
Lese Head Pointer: -1
-1 bedeutet: kein Fragment zu zeichnen

Node Buffer

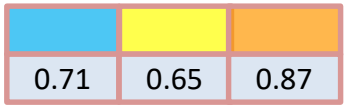
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... |
|---|---|---|---|---|---|---|---|
| 0.87 | 0.89 | 0.90 | 0.65 | 0.65 | 0.71 | | |
| -1 | -1 | -1 | 0 | -1 | 3 | | |

# Per-Pixel Linked Lists

## Compositing, Rendering der Fragmente




(1,1):
Lese Head Pointer: 5
Lese Node Data (5)
Durchlaufe Liste und speichere in temporärem Array

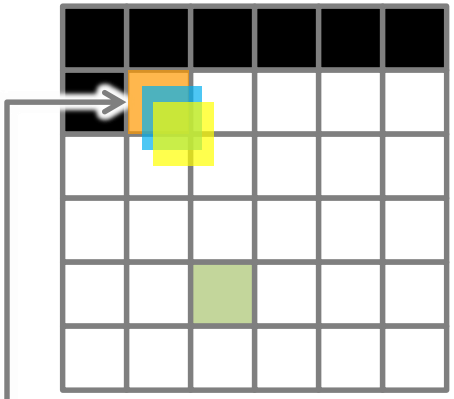


# Per-Pixel Linked Lists

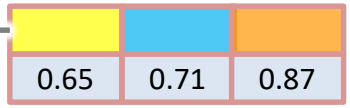
## Compositing, Rendering der Fragmente

| | | | | | |
|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | 5 | 4 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | 1 | 2 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |

Render Target



(1,1):
Sortiere temporäres Array
Blende Farben, Ausgabe



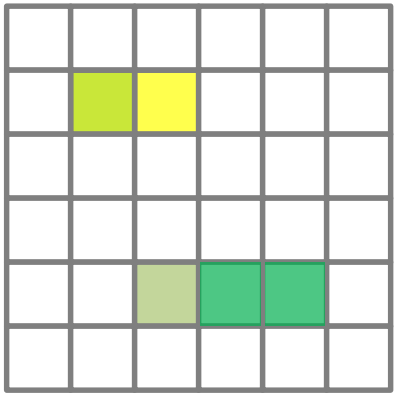

Node Buffer

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... |
|---|---|---|---|---|---|---|---|
| 0.87 | 0.89 | 0.90 | 0.65 | 0.65 | 0.71 | | |
| -1 | -1 | -1 | 0 | -1 | 3 | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

# Per-Pixel Linked Lists

## Compositing, Rendering der Fragmente

Render Target



Head Pointer Buffer

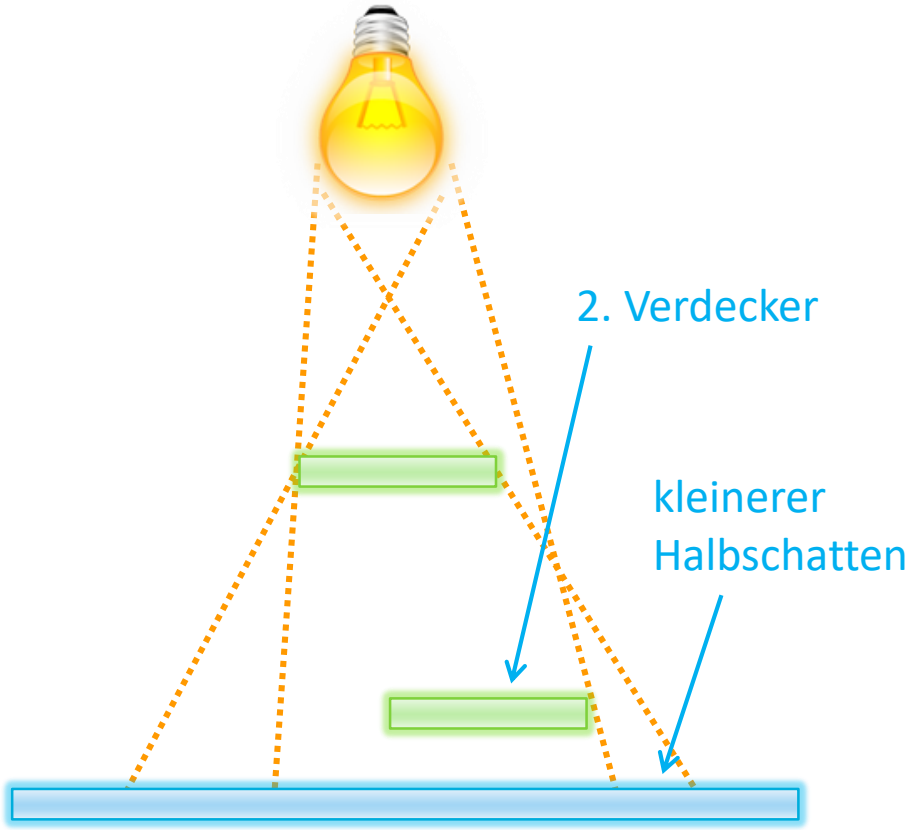
| | | | | | |
|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | 5 | 4 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | 1 | 2 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 |

Node Buffer

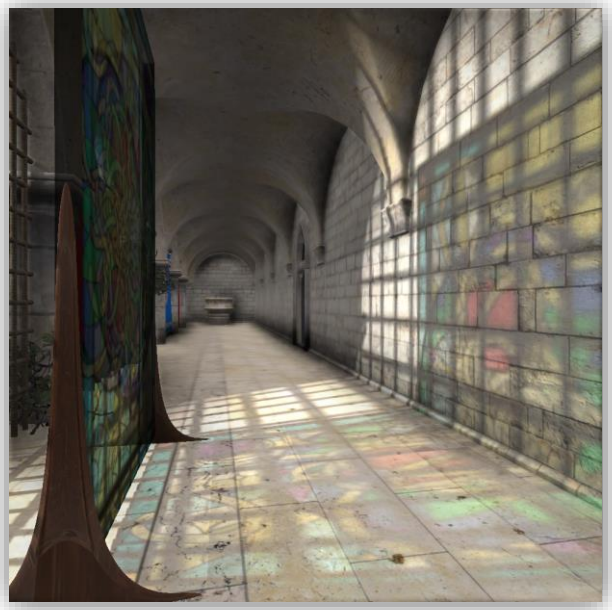
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... |
|---|---|---|---|---|---|---|---|
| 0.87 | 0.89 | 0.90 | 0.65 | 0.65 | 0.71 | | |
| -1 | -1 | -1 | 0 | -1 | 3 | | |

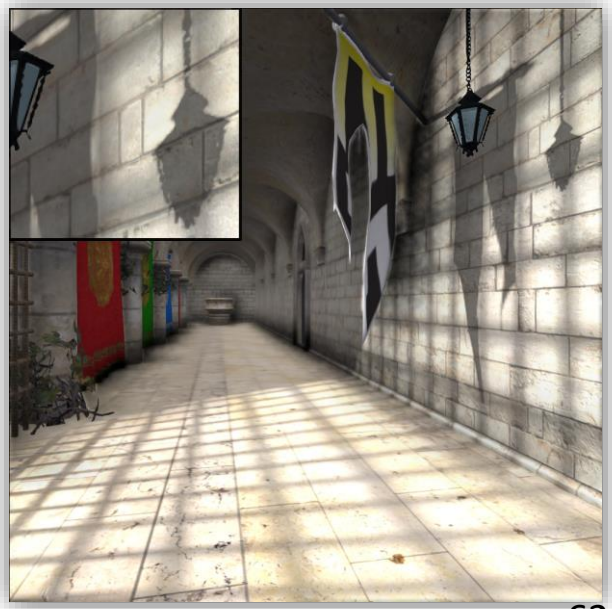
# Filtering Multilayer Shadow Maps for Accurate Soft Shadows

- Anwendung der Pro-Pixel Fragment Lists (oder vergleichbarer Technik) zum Sammeln von Informationen über verdeckte Flächen für Flächenlichtquellen
- → Übung!




http://onlinelibrary.wiley.com/doi/10.1111/cgf.12506/abstract





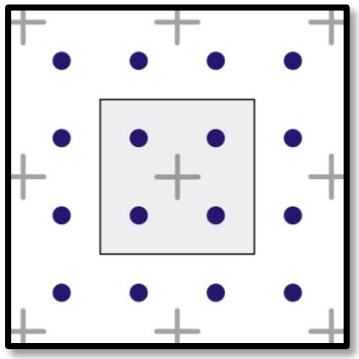
# Deferred Shading

**Weitere Aspekte auf dem Weg zu einer „kompletten“ Rendering-Technik**

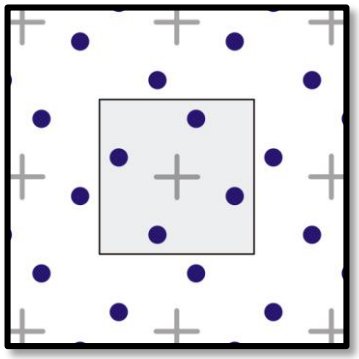
- Transparenz (immer schwierig bei Rasterisierung)
- Antialiasing
- realistischeres Shading durch Ausnutzen der G-Buffer
  - Ambient Occlusion, indirekte Beleuchtung, Reflexionen, …

# Rasterisierung und Antialiasing

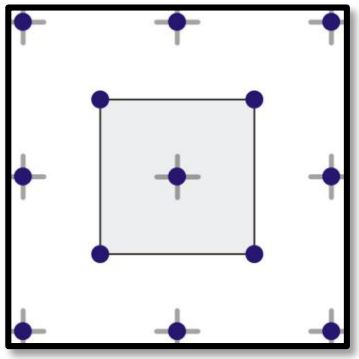
- bei Rasterisierung verwenden GPUs diverse Supersampling-Strategien



einfaches Supersampling



Rotated Grid Supersampling



2x Quincunx (NVIDIA)
im Schnitt:
2 Auswertungen pro Pixel

Bilder: http://de.wikipedia.org/wiki/Antialiasing_(Computergrafik)

- Deferred Shading nicht direkt mit GPU-Antialiasing verwenden: gemittelte Tiefenwerte, Material IDs etc. machen keinen Sinn
- aber: G-Buffer können mit Supersampling erzeugt werden
  - mittels höher aufgelösten FBOs oder mittels Render-Targets mit Hardware-Supersampling (speichern mehrere Samples pro Pixel auf die einzeln zugegriffen werden kann)
  - Antialiasing: Supersampling und Beleuchtungsberechnung auf hoher Auflösung, Mittelung der Farben am Ende

# Morphologisches Antialiasing (MLAA)

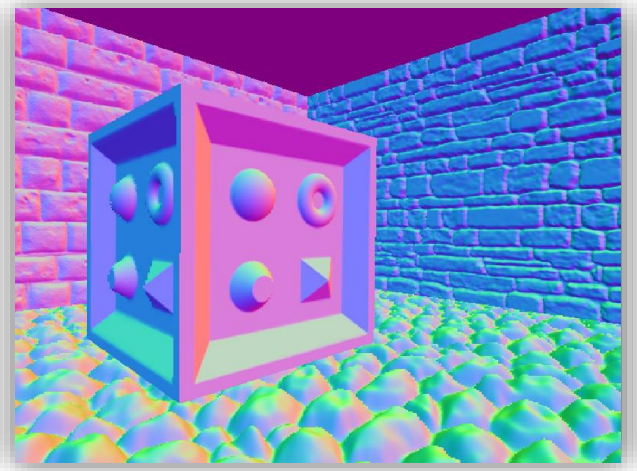
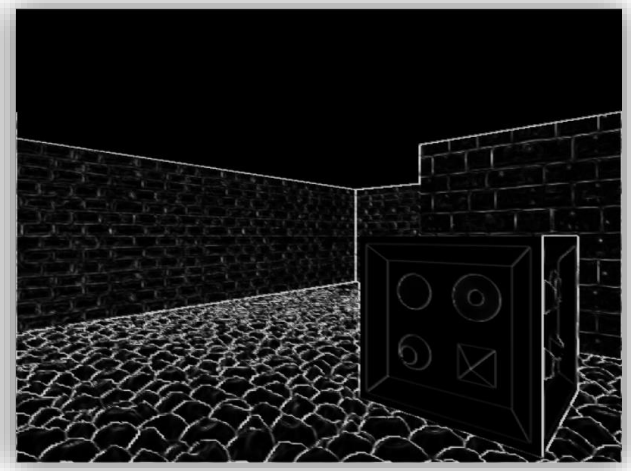
- Annahme: Antialiasing ist vor allem an Objektkanten wichtig und auf den Flächen verhindert Texturfilterung Artefakte
  - weitere Aliasing-Probleme, z.B. Shading mit Normal Maps außen vor

- Grundidee des **morphologischen Antialiasing** (MLAA) ist daher
  - Glättungsoperationen werden an den Kanten eingesetzt
  - Kanten werden **heuristisch** detektiert durch
    - Diskontinuitäten im Tiefenwert, oder
    - unterschiedliche Normale benachbarter Pixel, oder
    - deutliche Farbunterschiede
  - implementiert als ein Fragment Programm, das das ganze Bild **nachträglich bearbeitet**
- Demo: http://divergentcoder.com/rendering/morphological-antialiasing/

# Morphologisches Antialiasing (MLAA)

## Kantendetektion

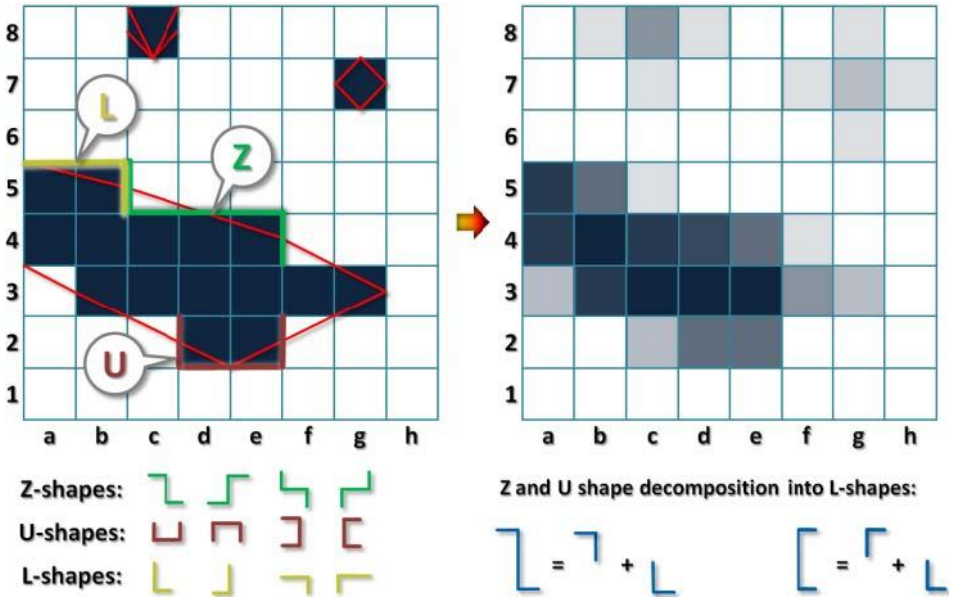
- Beispiel Heuristik: weichen Tiefenwerte oder Normalen der benachbarten Pixel stark ab, so wird ein Pixel als Diskontinuität markiert
- beide Kriterien alleine sind nicht verlässlich
  - bei flachen Blickwinkeln auf eine (glatte) Fläche kann die Tiefe zweier Nachbarpixel stark variieren
  - hintereinander liegende Flächen können dieselbe Orientierung haben
  - beide Kriterien zusammen sind (einigermaßen) verlässlich

# Morphologisches Antialiasing (MLAA)

## Morphologisches Antialiasing nur auf Basis von Farben

- die ursprüngliche Variante, betrachtet sogar nur Farbunterschiede
- drei Schritte ausgehend von einem Farbbild (ohne Supersampling)
  - suche Diskontinuitäten zw. benachbarten Pixeln (Farbdifferenzen)
  - identifiziere Strukturen (Z,U,L-Shapes) in den Diskontinuitäten
  - Überblenden der Farben in der Umgebung dieser Strukturen nach festen Schemata




- Details: Alexander Reshetov, Morphological Antialiasing, http://visual-computing.intel-research.net/publications/papers/2009/mlaa/mlaa.pdf

# Morphologisches Antialiasing (MLAA)

## Beispiel

- ohne MLAA (http://www.eurogamer.net/articles/digitalfoundry-mlaa-360-pc-article)

# Morphologisches Antialiasing (MLAA)

## Beispiel

- mit MLAA (http://www.eurogamer.net/articles/digitalfoundry-mlaa-360-pc-article)

# Morphologisches Antialiasing (MLAA)

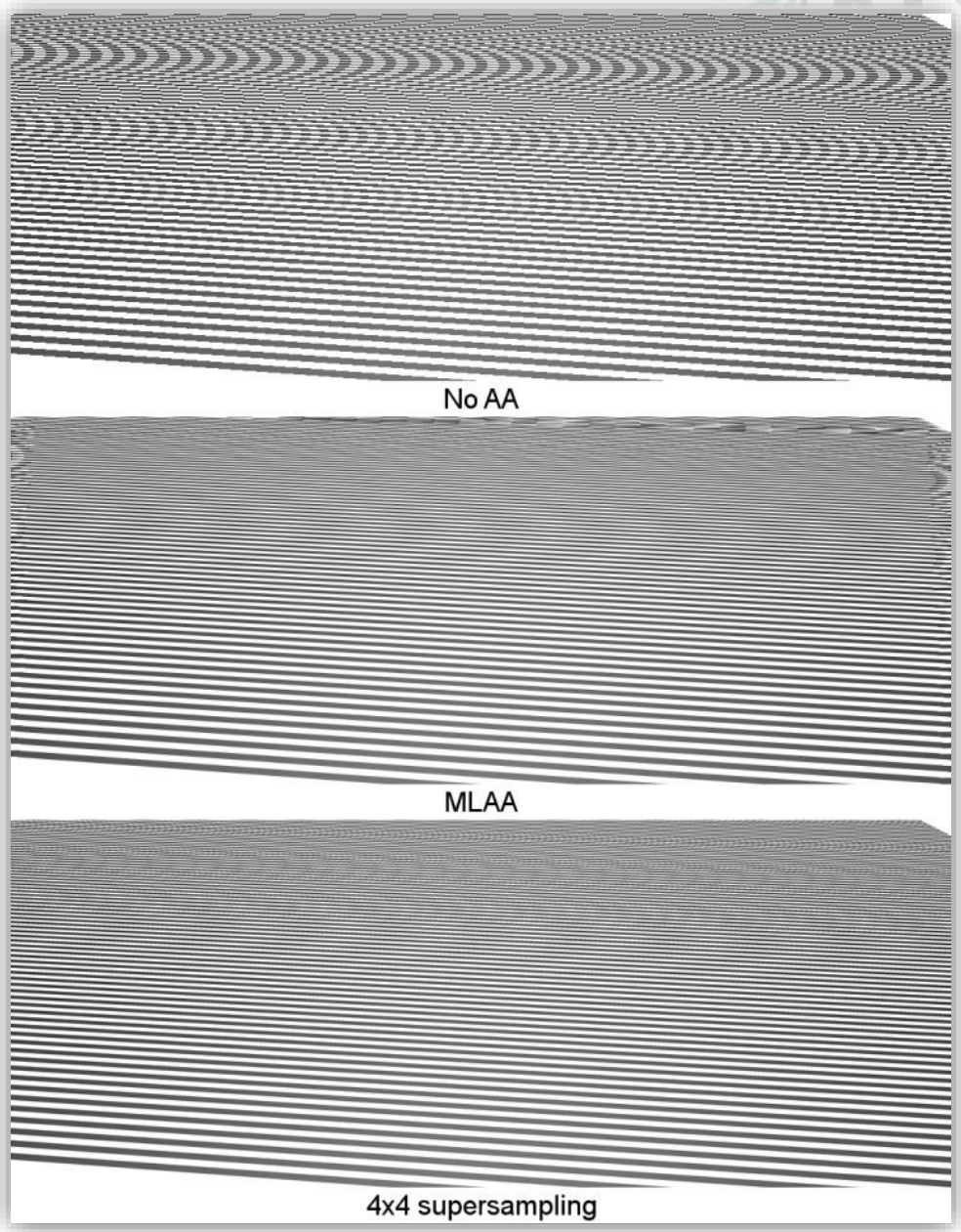
## Anwendungen

- MLAA erfreut(e) sich großer Beliebtheit, z.B. in Lucas Arts Titeln (eine MLAA ähnliche Technik, Bild), God of War III, ATI Grafikkartentreibern, …
- hardwarebedingt eher auf Playstation 3 und PC, aber auch auf XBOX360
- weitere Variante und Demo: http://www.iryoku.com/mlaa/

# Morphologisches Antialiasing (MLAA)

## Offensichtliche Probleme

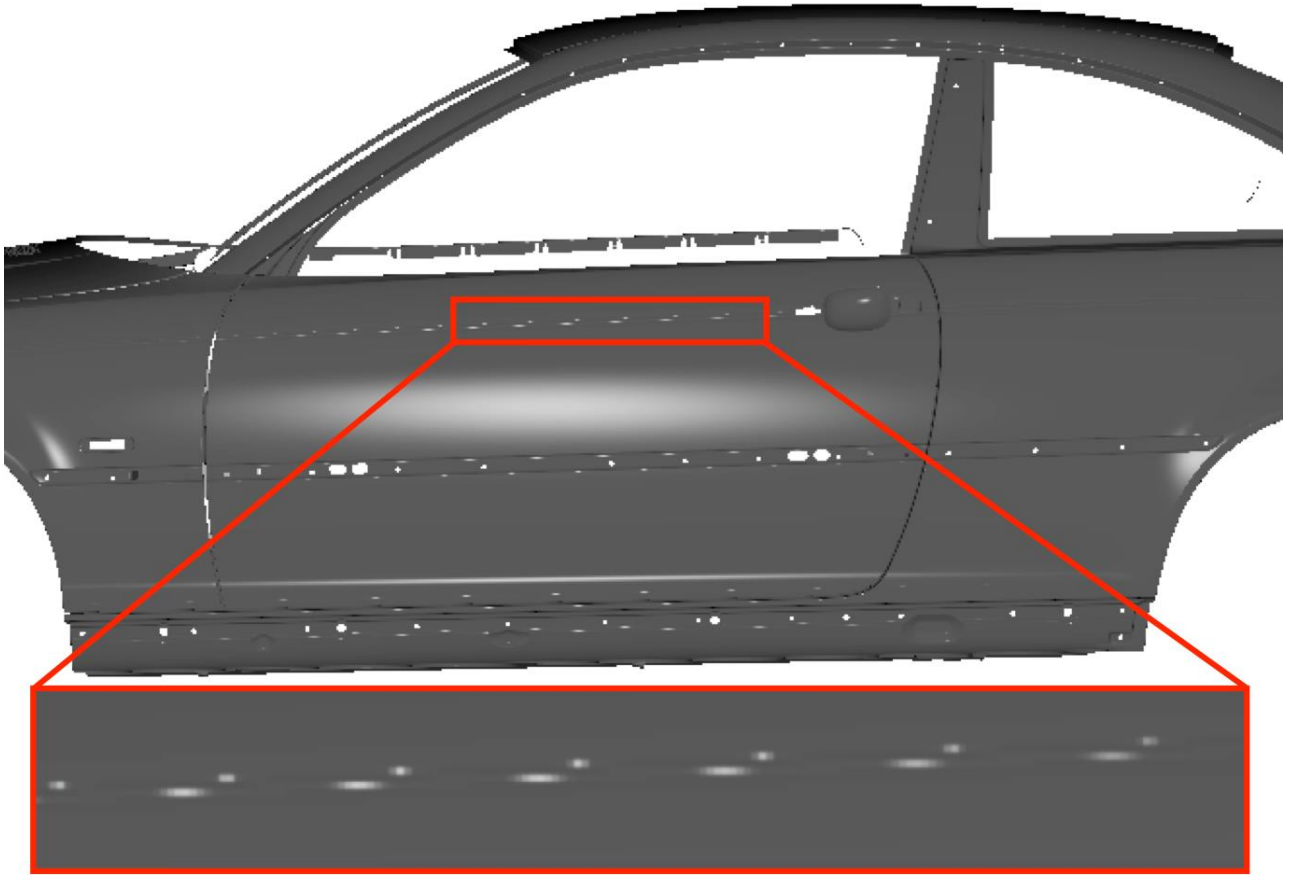
- natürlich kann verlorene Information nicht zurückgewonnen werden
  - es kommt zu Rekonstruktionsfehlern bei zu hohen Frequenzen
  - feine Geometrie kann unterabgetastet werden



# Morphologisches Antialiasing (MLAA)

## Offensichtliche Probleme

- natürlich kann verlorene Information nicht zurückgewonnen werden, z.B. in diesem Beispiel: mangelnde Abtastung eines Glanzlichts



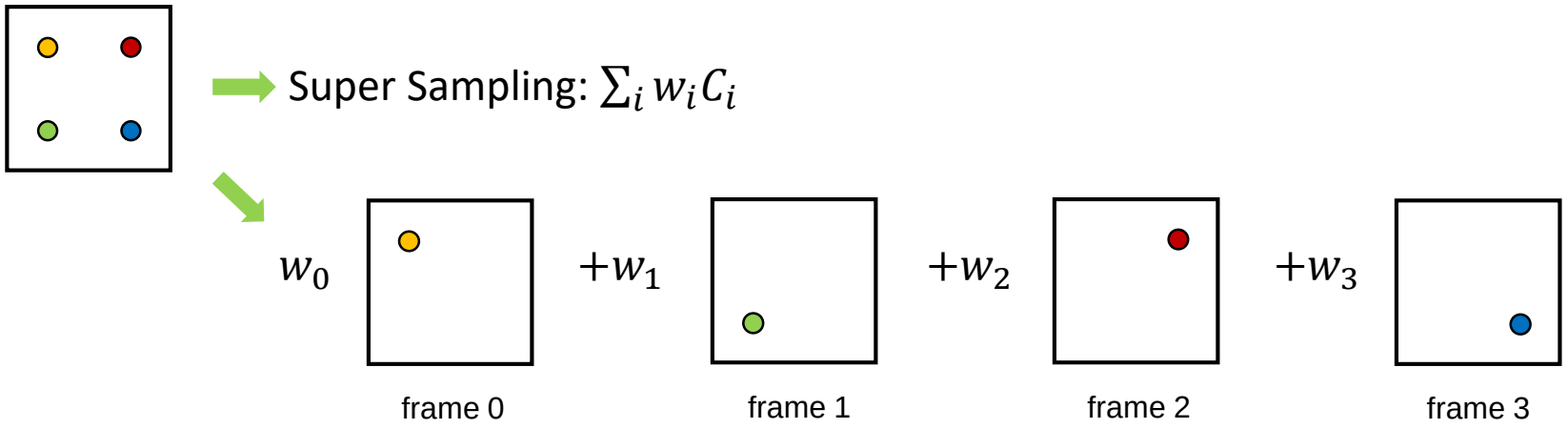
# Temporal Antialiasing

- ideale Lösung für Antialiasing: Vorfilterung
  - nicht immer möglich, aber Fortschritte (siehe andere Kapitel der VL)
- billige Lösung: morphologisches Antialiasing
- brute-force Lösung: mehr Samples
  - klassisches, direktes Super Sampling ist teuer
  - besser sind MSAA, CSAA (insb. für Forward Shading)
- temporales Antialiasing (TAA):
  - mehr Samples, aber nicht alle gleichzeitig

# Temporales Antialiasing

## Amortisation der Samples über mehrere Frames

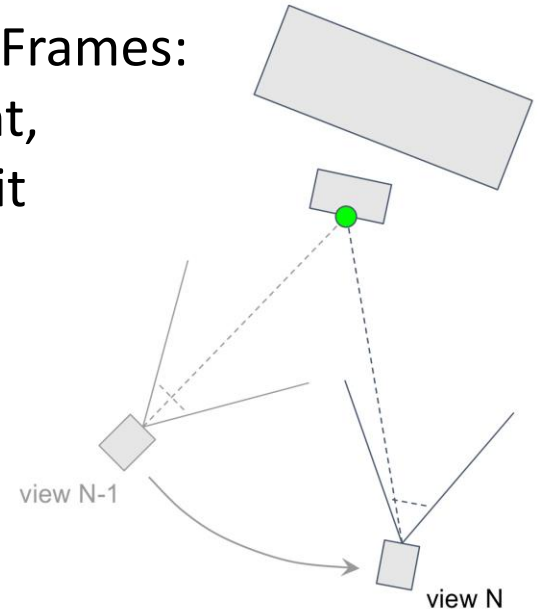
- Grundidee: verwende Samples aus vorhergehenden Frames (dazu leichtes Jittering der Kamera)



# Temporales Antialiasing

## Amortisation der Samples über mehrere Frames

- Grundidee: verwende Samples aus vorhergehenden Frames (dazu leichtes Jittering der Kamera)
- projiziere Fläche in vorherige Frames:
  Position der Fläche ist bekannt,
  berechne Pixelkoordinaten mit
  Objekt, Kamera- und
  Projektionstransformation
  aus dem vorherigen Frame

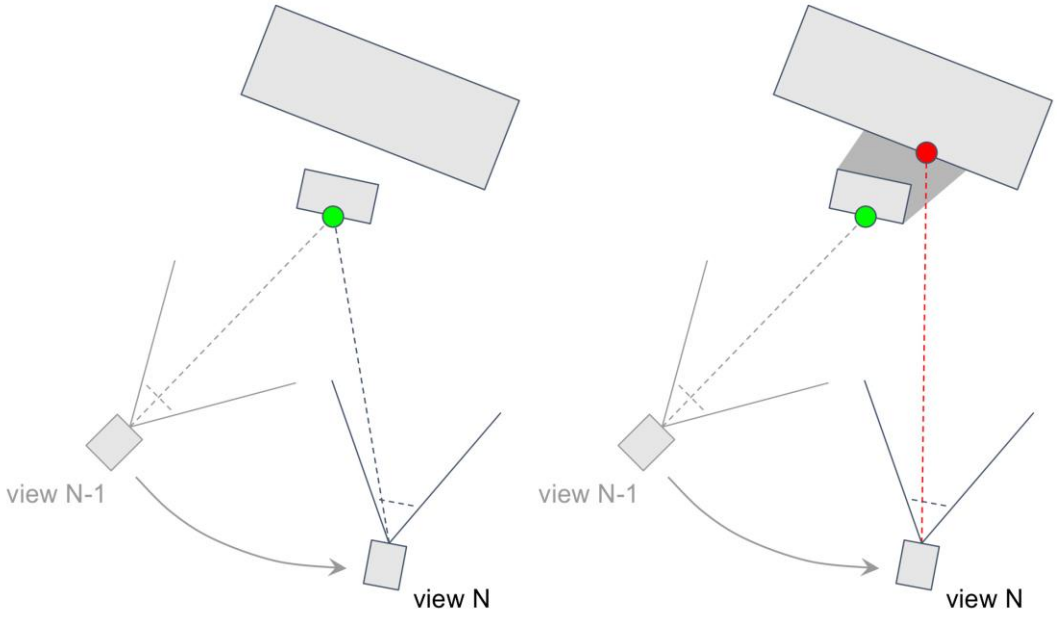



- ursprüngliche TAA-Varianten
  - verwende Samples nur bei
    gleicher Material ID, Tiefe, Normale etc.

# Temporales Antialiasing

## Amortisation der Samples über mehrere Frames

- Grundidee: verwende Samples aus vorhergehenden Frames (dazu leichtes Jittering der Kamera)
- Probleme:
  - Verdeckung (Abb. rechts)
  - veränderte Beleuchtung
  - Bewegung der Objekte
  - keine Glättung der Kanten, nur Antialiasing des Shading (Samples von der „gleichen“ Fläche)




- für einige dieser Probleme kann man sich Lösungen überlegen, allerdings wird das schnell unpraktikabel und anfällig für Artefakte (z.B. Ghosting, Spuren von Glanzlichtern, …)

# Temporales Antialiasing

**Pragmatischer und robuster Ansatz (so durchgeführt für jeden Pixel)**

- projiziere die Fläche (nur) in das vorherige Frame
- Gewichtung der Farbwerte nach exponentiell geglätteten Mittelwert (engl. exponential moving average) mit Parameter $\alpha$ (z.B. $\alpha = 0.1$)
- Pixel-Wert $P_n$ im Frame $n$ aus
  - neu-berechnetem Farbwert $C_n$ und
  - altem Pixel-Wert $P_{n-1}$ mit

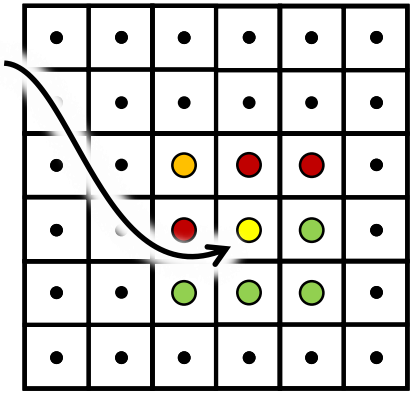
$$P_n = \alpha \cdot C_n + (1 - \alpha) \cdot P_{n-1}$$

# Temporales Anti-Aliasing

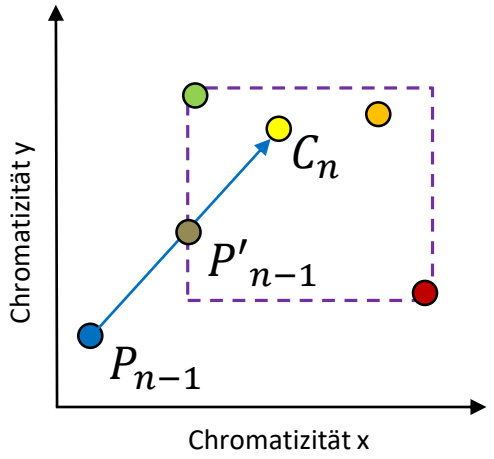
**Pragmatischer und robuster Ansatz (so durchgeführt für jeden Pixel)**

- man möchte nur Farbwerte aus vorherigen Frames verwenden, die konsistent mit $C_n$ sind (Annahme: große Unterschiede ↔ andere Fläche)

- Idee nennt sich Neighborhood Clipping
  - betrachte Nachbarschaft (z.B. $3 \times 3$) des Pixels mit $C_n$
  - bestimme Hüllkörper oder konvexe Hülle der dort gefundenen Chrominanzen
  - wenn $P_{n-1}$ außerhalb, dann berechne Schnitt $\overline{P_{n-1}C_n}$ und verwende diese Farbe

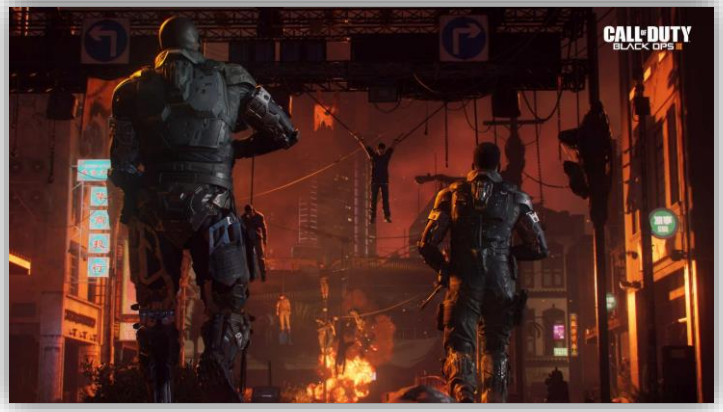


- durch reine Betrachtung von Farbwerten, werden alle Ursachen von Aliasing adressiert, aber
  - u.U. zu starke Glättung
  - Flackern, bei großflächigem Aliasing (z.B. Nachbarschaft komplett verdeckt durch eine rot-grün flackernde Fläche → Clipping liefert immer rot oder grün)



# Deferred Shading – Zwischenfazit

- Probleme sind
  - spezielle und teure Behandlung von Transparenz (immer bei Rasterisierung)
  - Speicherbedarf (Konsolen, Embedded GPUs → Tile-based Renderer)
- Antialiasing: MLAA als „Notlösung“,
  heute: temporale Antialiasing-Verfahren
- wann macht Deferred Shading Sinn?
  - bei einer großen Anzahl von Lichtquellen
  - Entkopplung von Geometrie und Beleuchtung (einfachere Systeme)
  - **wenn die G-Buffer Information noch anderweitig genutzt wird**
- was spricht gegen Deferred Shading?
  heutzutage: wenig!
- es gibt eine Vielzahl DS-Varianten, siehe
  the real-time rendering continuum: a taxonomy
  http://c0de517e.blogspot.de/2016/08/the-real-time-rendering-continuum.html



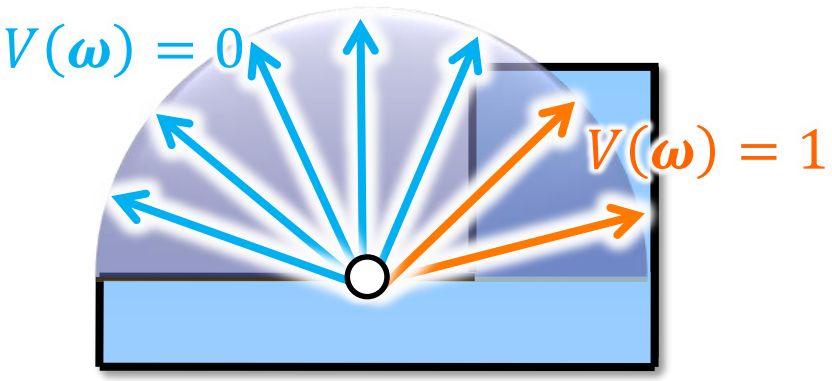
# Deferred Shading

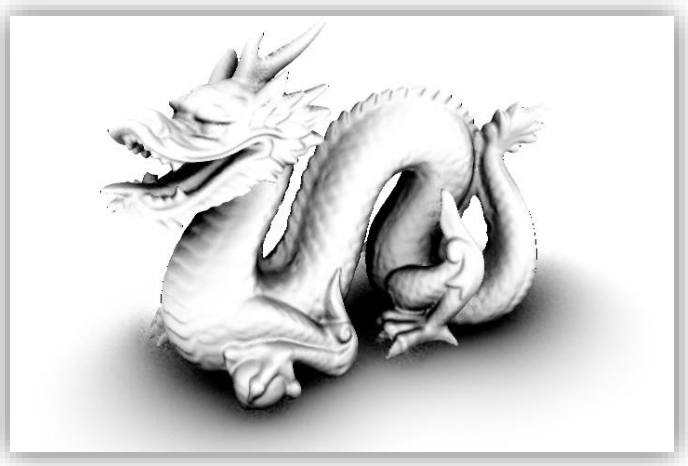
**Weitere Aspekte auf dem Weg zu einer „kompletten“ Rendering-Technik**

- Transparenz (immer schwierig bei Rasterisierung)
- Antialiasing
- realistischeres Shading durch Ausnutzen der G-Buffer
  - Ambient Occlusion, indirekte Beleuchtung, Reflexionen, …

# Ambient Occlusion

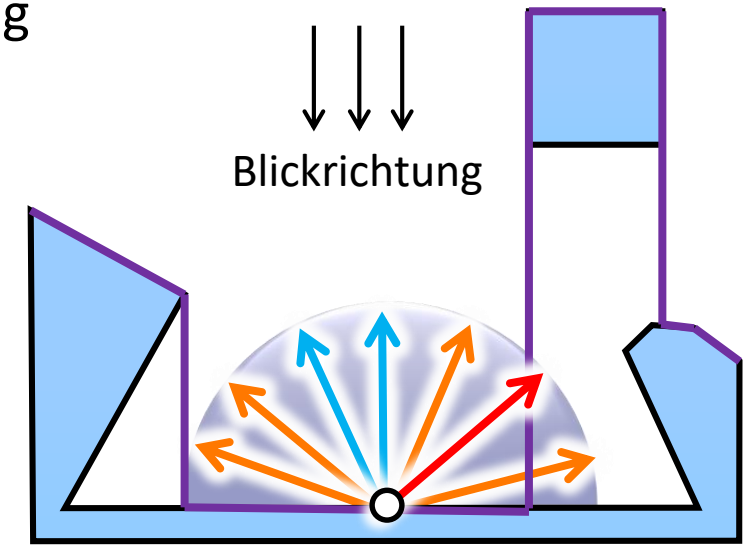
- Ambient Occlusion (oder auch „Accessibility“) beschreibt wie viel Umgebungslicht einen Oberflächenpunkt erreichen kann
  - kann vorberechnet und pro Vertex oder in einem Texturatlas gespeichert werden
  - oft berechnet in der Form $A = 1 - \frac{1}{\pi}\int_{\Omega^+} V(\boldsymbol{\omega})W(\boldsymbol{\omega})(\boldsymbol{\omega}\cdot\boldsymbol{n})d\boldsymbol{\omega}$
    - Sichtbarkeitsfunktion $V(\boldsymbol{\omega})$
    - Gewichtungsfunktion $W(\boldsymbol{\omega})$: weiter entfernte Schnittpunkte in Richtung $\omega$ haben geringeren Einfluss (keine physikalische Entsprechung)



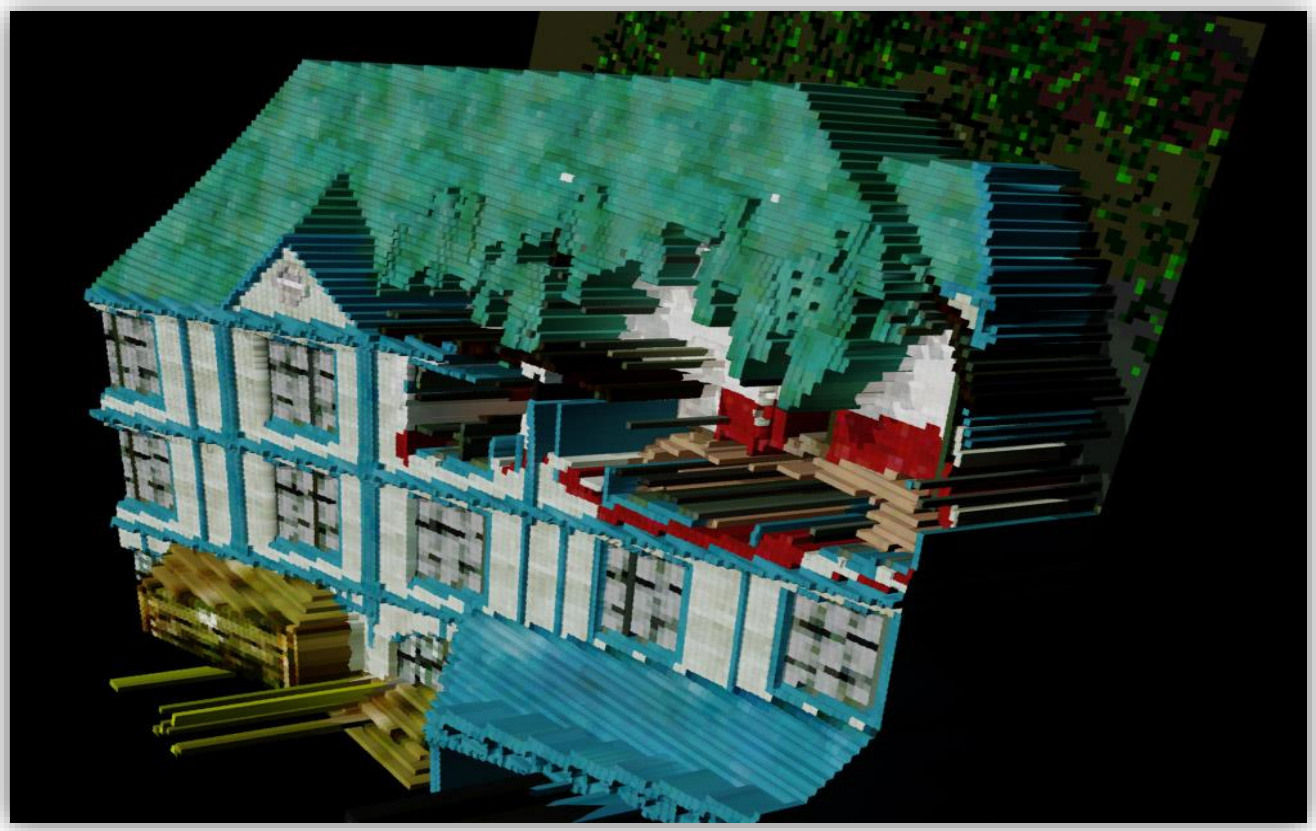
# Ambient Occlusion im Bildraum

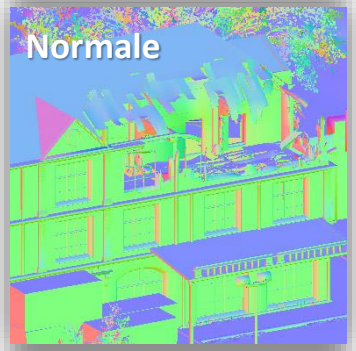
- Approximation von Ambient Occlusion ist auch im Bildraum möglich
  - Inhalt des Tiefenpuffers ist eine Abtastung der sichtbaren Flächen
  - die Information ist natürlich unvollständig, genügt aber (oft) um approximative Beleuchtungseffekte zu berechnen
  - Vorteil: keine Vorberechnung, ein reiner Post-Process
  - Anm. Depth Peeling u.a. liefern auch Informationen über weitere Flächen, deren Berücksichtigung ist aber entsprechend teurer
- **Bsp.**: im Tiefenpuffer abgetasteten Flächen ergeben für viele Strahlen korrekt Schnittpunkte (blau, orange), aber nicht für alle (rot) – allerdings ist $W(\boldsymbol{\omega})$ meist nicht richtig



# Ambient Occlusion im Bildraum

- „Rekonstruktion“ der Oberflächen aus einem G-Buffer




# Ambient Occlusion im Bildraum

- sog. „Screen-Space Ambient Occlusion“ (SSAO) Verfahren unterscheiden sich in der Art und Weise, wie die Information im Tiefenpuffer/G-Buffer genutzt wird
- Beispiel: Methode verwendet in Crysis [Mittring07]
  - anstatt Strahlen zu betrachten werden zufällige Punkte in der Umgebung (Kugel) eines Oberflächenpunkts erzeugt
  - $AO \approx \frac{\text{Punkte die Tiefentest bestehen}}{\text{Punkte die Tiefentest nicht bestehen}}$

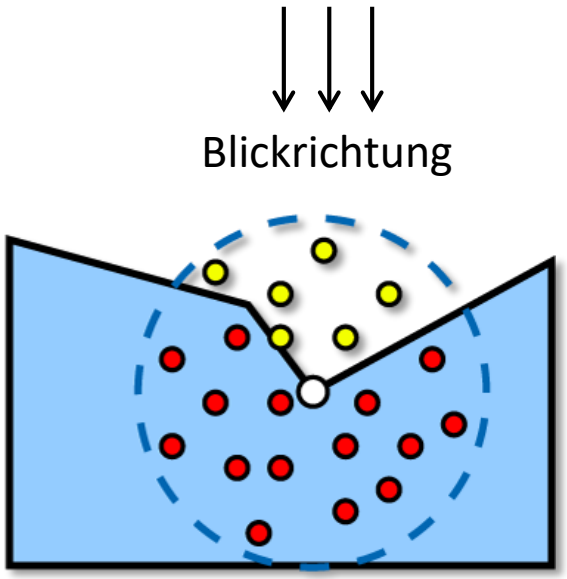
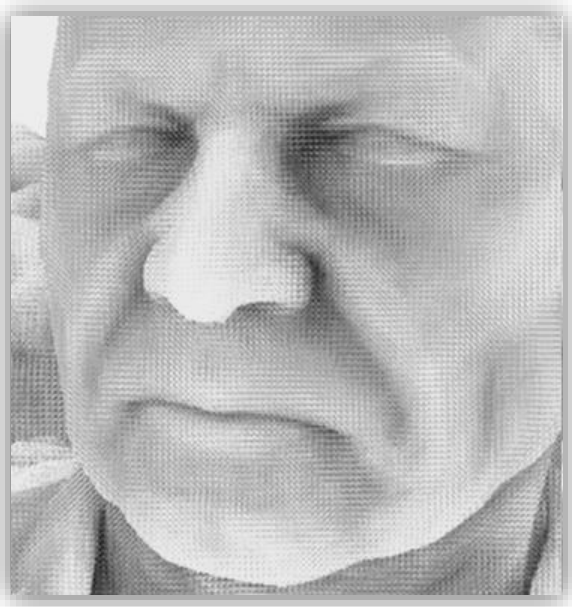



Bild: Crytek

# Ambient Occlusion im Bildraum

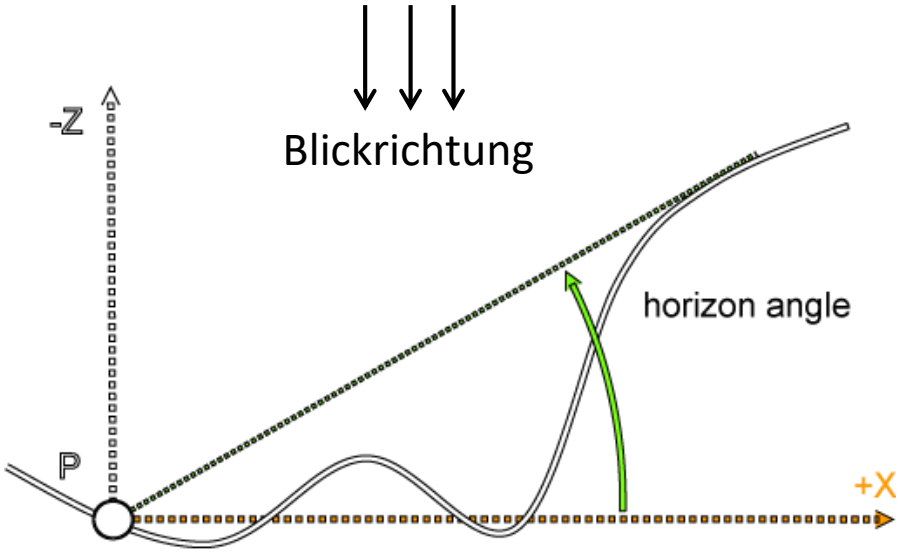
- praktische Aspekte (übliches Vorgehen auch in ähnlichen Fällen):
  - aufgrund der Rechenzeit können nur wenige Punkte pro Pixel (Größenordnung: 8 bis 16) getestet werden → wird immer dasselbe Muster verwendet, treten Artefakte auf (linkes Bild)
  - Unterdrückung durch „zufällige“ Rotation der Testpunkte pro Pixel
  - dadurch entstehendes Rauschen wird abschließend geglättet (temporal, oder mit kantenerhaltenden Filter, siehe späteres Kapitel)

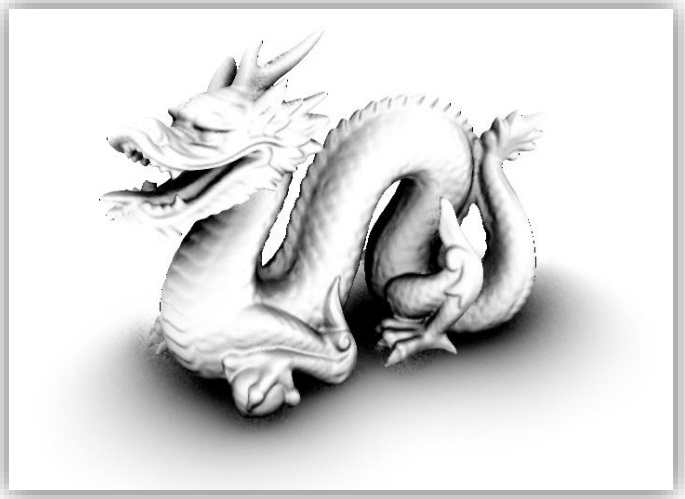




Bilder: Crytek

# Ambient Occlusion im Bildraum

- es gibt dutzende Verfahren und Varianten, die eben vorgestellte Methode ist die einfachste
- aufwändiger und akkurater: Horizon-Based Ambient Occlusion [Bavoil08]
  - interpretiere den Tiefenpuffer als Höhenfeld
  - verfolge Strahlen in der Hemisphäre über einem Oberflächenpunkt durch Ray Marching
  - schätze den Winkel zum „Horizont“ und damit die Verschattung ab
  - teuer, aber sehr gute Resultate



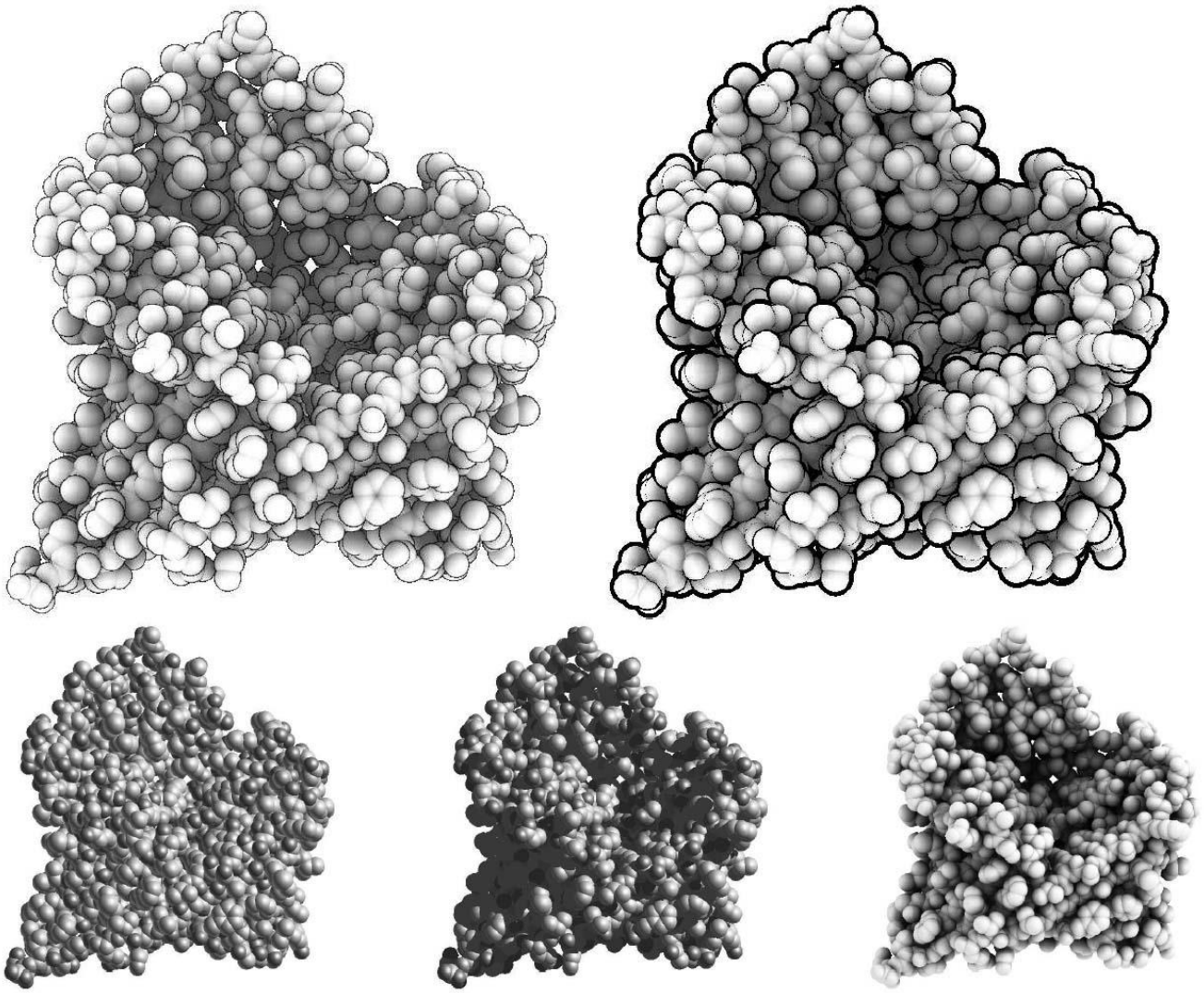
# Ambient Occlusion im Bildraum

- Scalable Ambient Obscurance (Idee: vorgefilterte Tiefenpuffer), HPG 2012, http://graphics.cs.williams.edu/papers/SAOHPG12/
- 3ms für 2560 x 1600 (+ Rand) mit GeForce GTX680

# Ausblicke

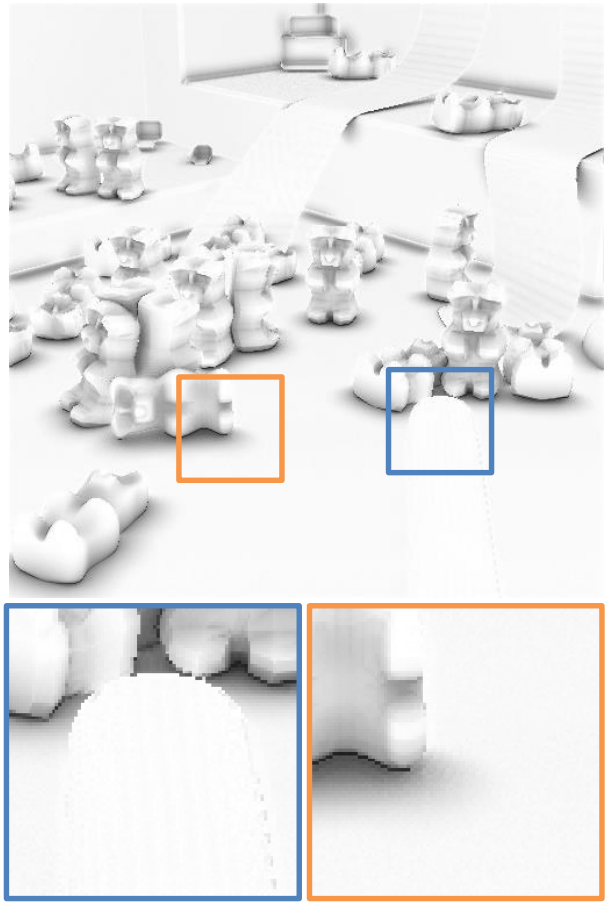
- Anwendung von Ambient Occlusion in der Visualisierung

  M. Tarini, P. Cignoni, and C. Montani. Ambient Occlusion and Edge Cueing for Enhancing Real Time Molecular Visualization. IEEE Trans. Vis. Comp. Graph., 12(5):1237-1244, 2006



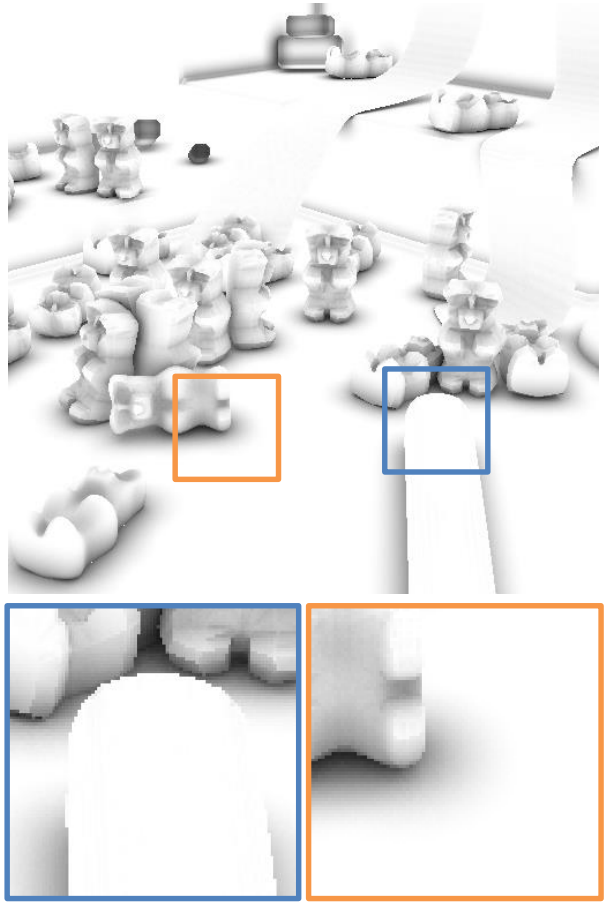
# Ambient Occlusion im Bildraum

## Problem: Fehlende Information (ein Lösungsansatz gleich!)



Horizon Based AO

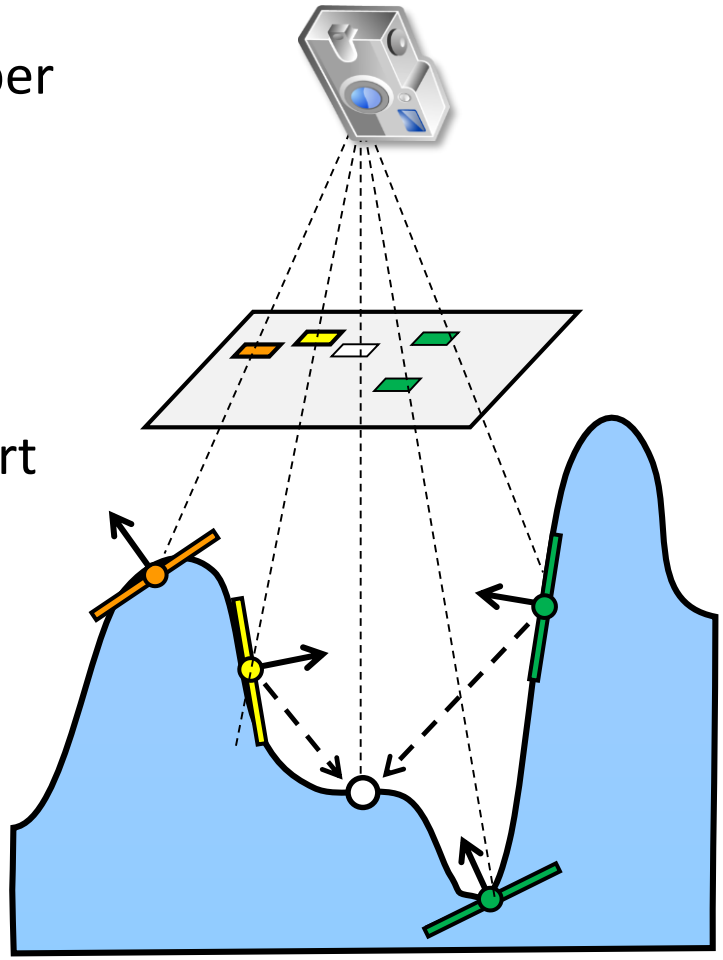
[Bavoil et al. 2008]



Raytracing

# Indirekte Beleuchtung im Bildraum

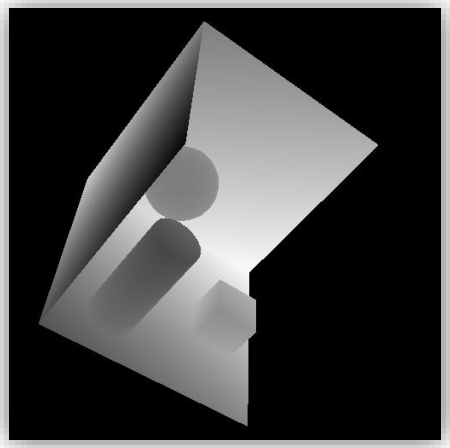
- man kann weitere Informationen aus dem G-Buffer nutzen und approximative indirekte Beleuchtung berechnen
  - jeder Pixel im G-Buffer repräsentiert ein Flächenstück, von dem wir Material, Normale und Position kennen
  - bei der SSAO Berechnung iterieren wir über Pixel in der Nachbarschaft
  - wir können die indirekte (unverschattete) Beleuchtung der entsprechenden Flächenstücke einfach berechnen
  - Prinzip wie Radiosity – Strahlungstransport zwischen zwei Flächen
  - Aufgrund der Rechenzeit: nur Transport zwischen räumlich nahen Flächen



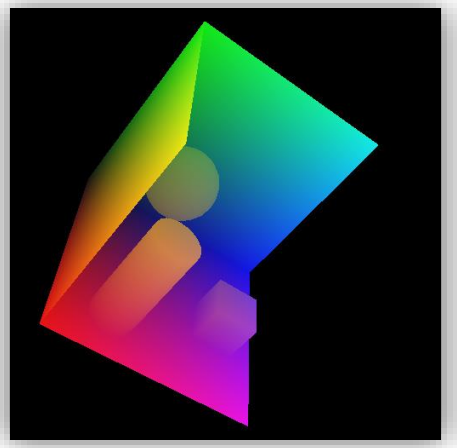
# Reflective Shadow Maps (RSMs)

## RSMs: Ursprung der Screen-Space Beleuchtungsverfahren

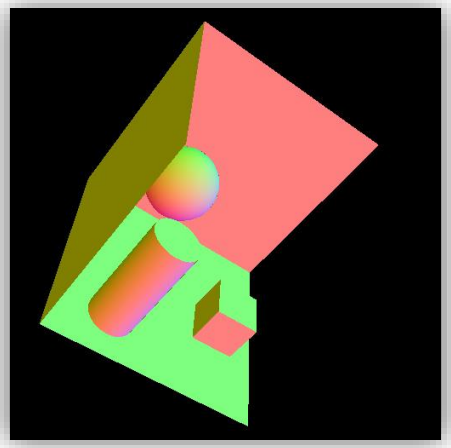
- Beobachtung: Shadow Maps tasten direkt beleuchtete Oberflächen ab
  - speichert man hierzu Position, Normale und das (diffus) reflektierte Licht, so erhält man alle Informationen, die man für einfach indirekte Beleuchtung benötigt → sog. Reflective Shadow Map
  - im Prinzip „viele kleine Lichtquellen“ und eine sog. Many-Lights Methode (siehe FotoBS-Vorlesung)



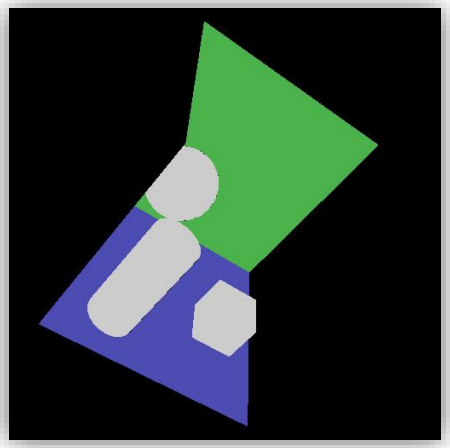
Tiefe



Position



Normale

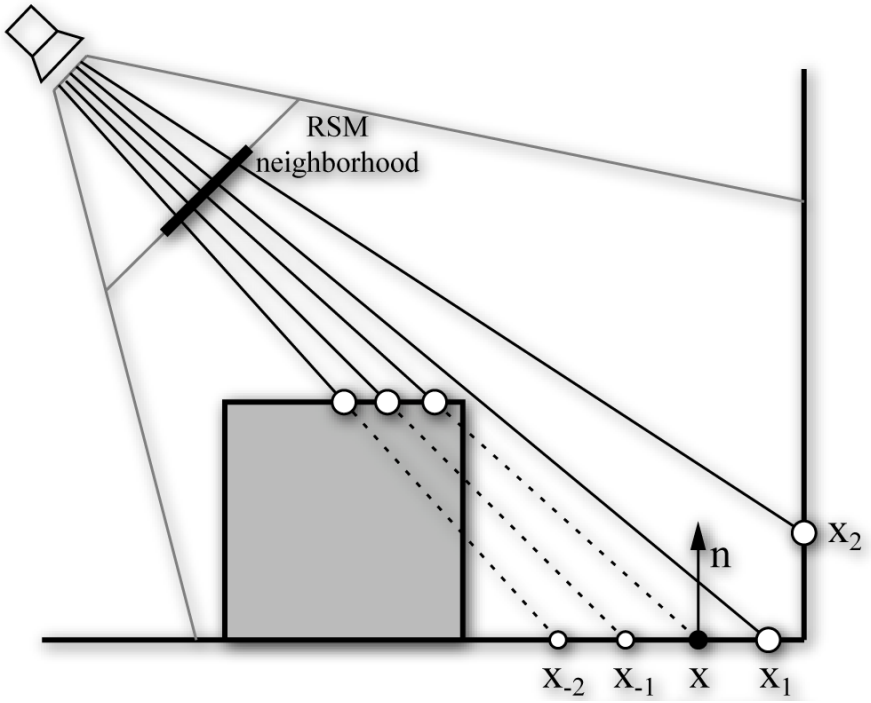


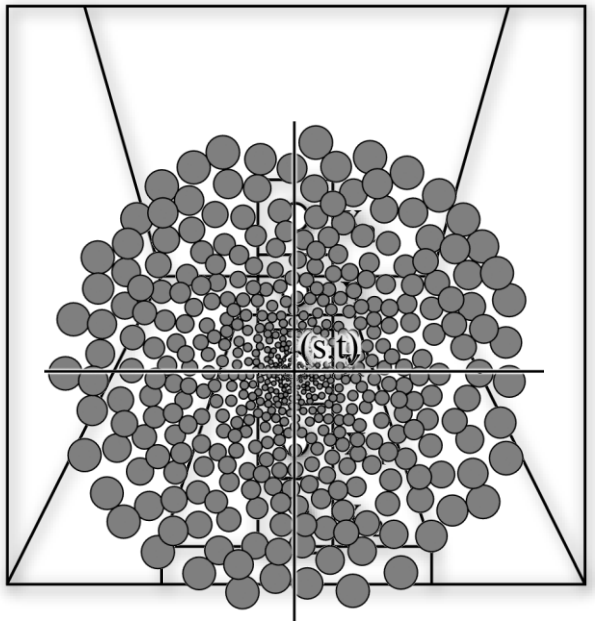
reflektierter Fluss

# Reflective Shadow Maps

## Verwendung von RSMs

- ebenfalls Einsammeln der indirekten Beleuchtung, jetzt aber in der Umgebung des Oberflächenpunktes projiziert in die RSM
- nur einfach-indirekte Beleuchtung (ursprünglich) ohne Verdeckung
  - Verdeckungsberechnung wäre z.B. über Voxelisierung möglich



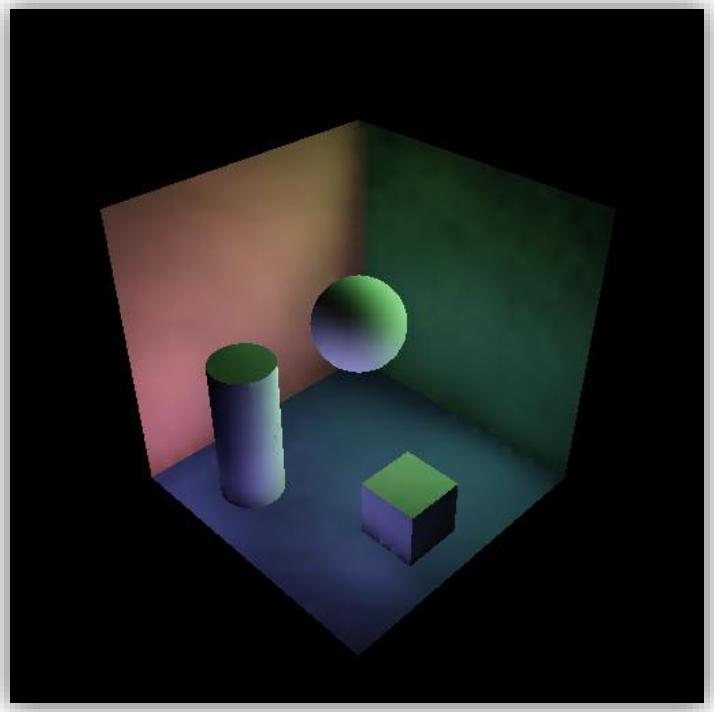



# Reflective Shadow Maps

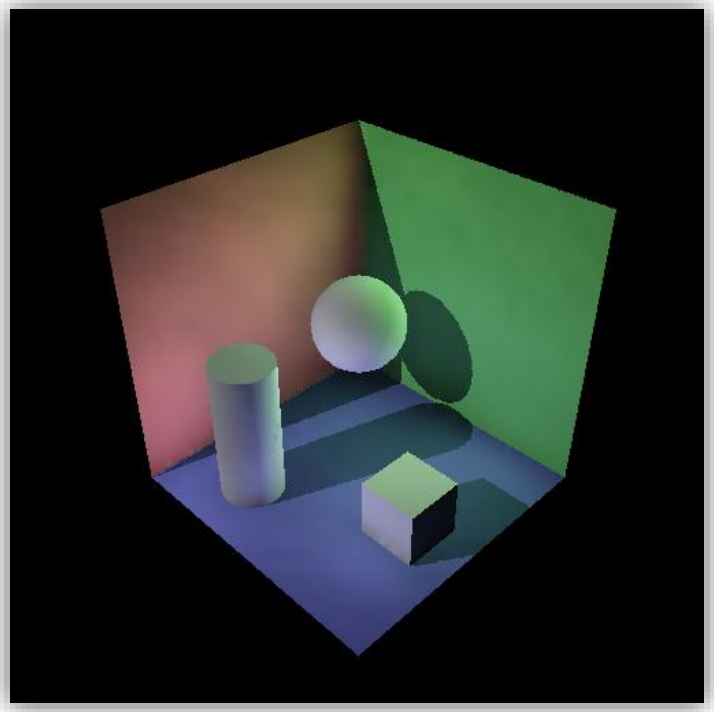
## Einschränkung und Ausblick

- Vorteil: alle direkt beleuchteten Flächen sind in der RSM abgetastet
- allgemeines Konzept, eingesetzt in einigen Verfahren für interaktive globale Beleuchtung (ohne obige Einschränkungen)

- Resultate:



indirekte Beleuchtung



direkte und indirekte Beleuchtung

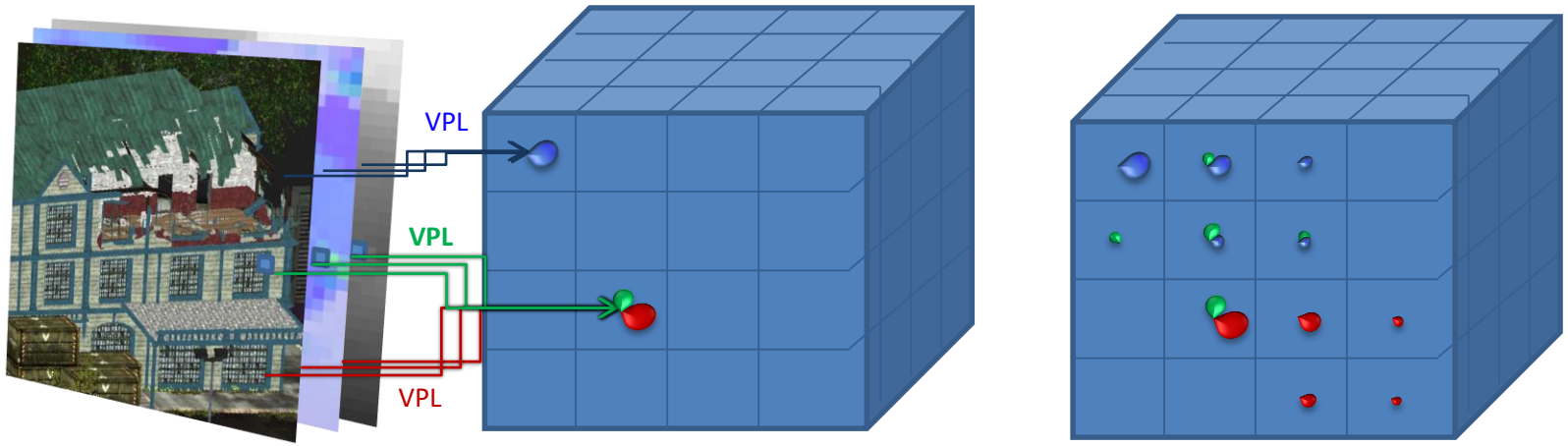
# Reflective Shadow Maps

**DiRT** (http://community.amd.com/community/amd-blogs/amd-gaming/blog/2012/07/03)

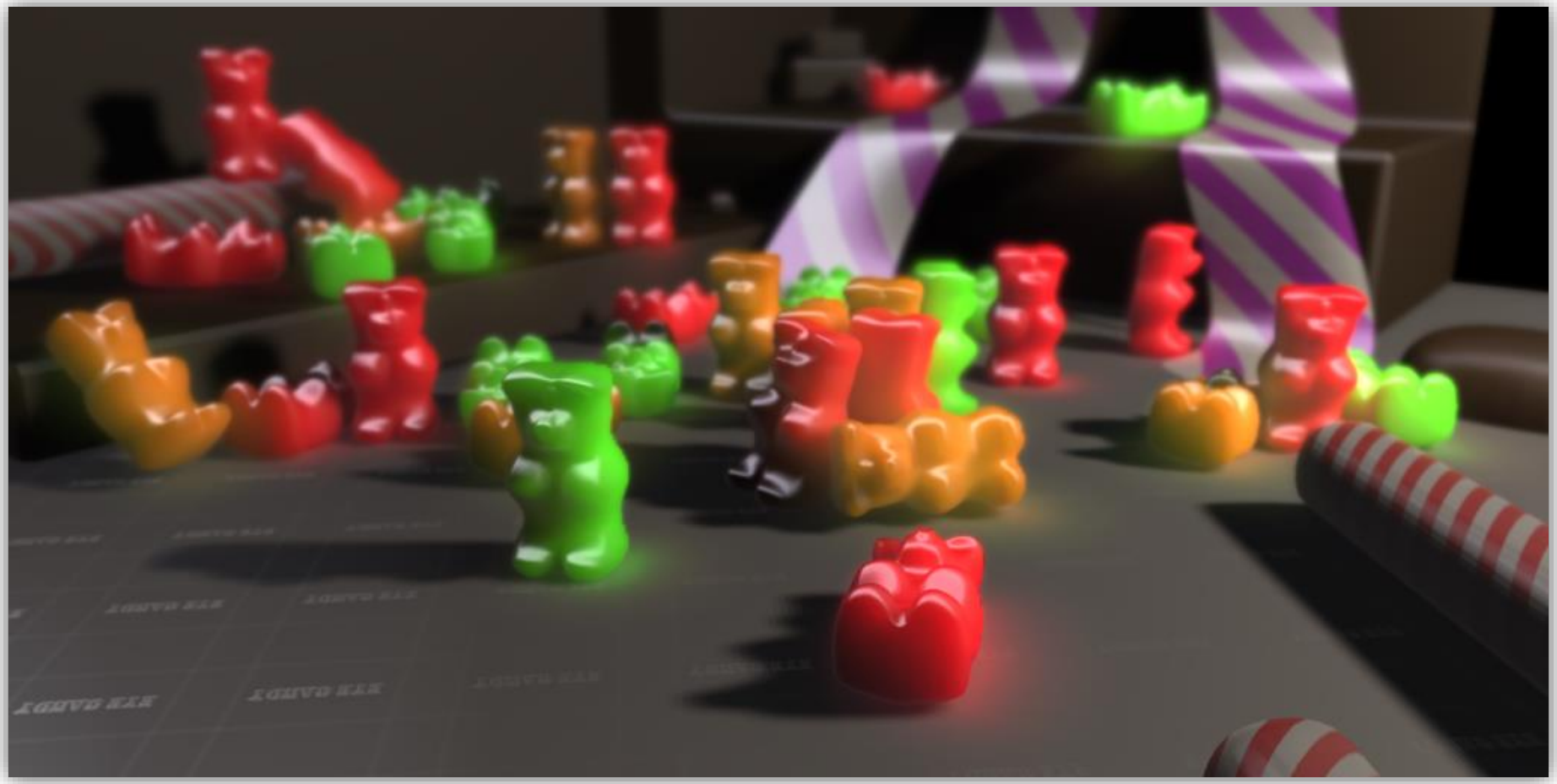

# Cascaded Light Propagation Volumes

- RSMs eingesetzt in der CryEngine 3
  Paper: http://cg.ivd.kit.edu/publikationen.php
- eine Variante der sog. Diskreten Ordinaten Methoden

**Reflective Shadow Map** **Radiance Volume** **Propagierung**



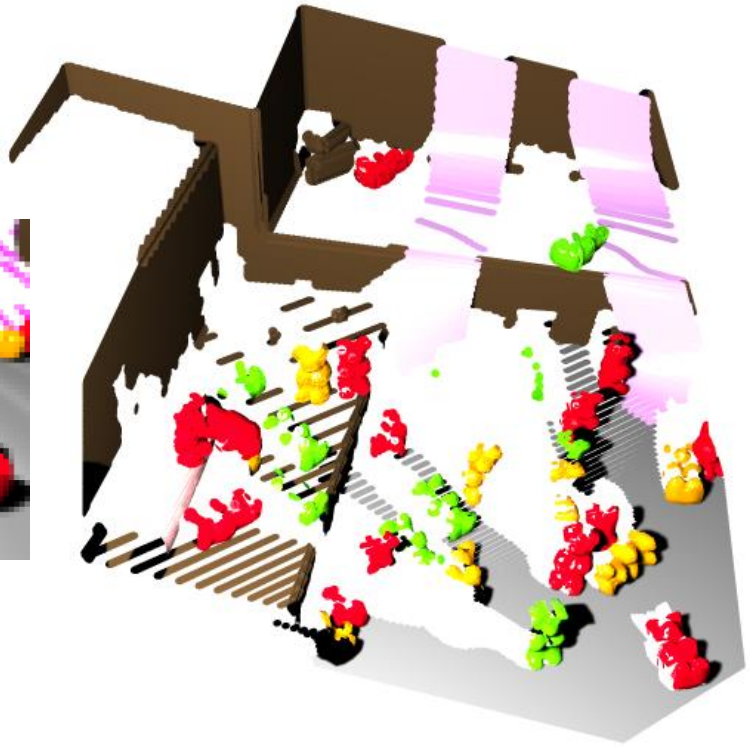
# Deep Screen Space



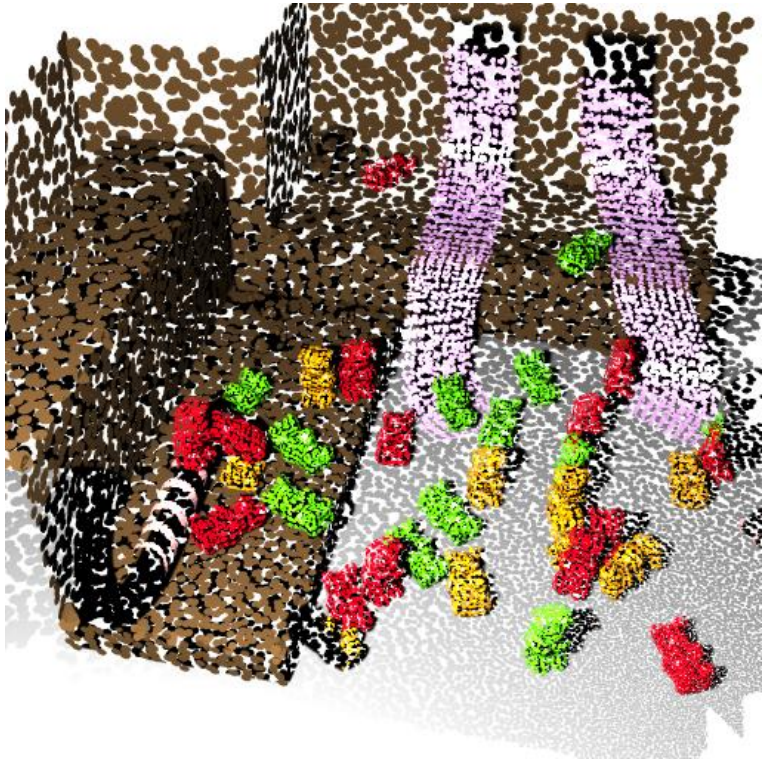
http://people.mpi-inf.mpg.de/~onalbach/DeepScreenSpace/

# Deep Screen Space

Framebuffer



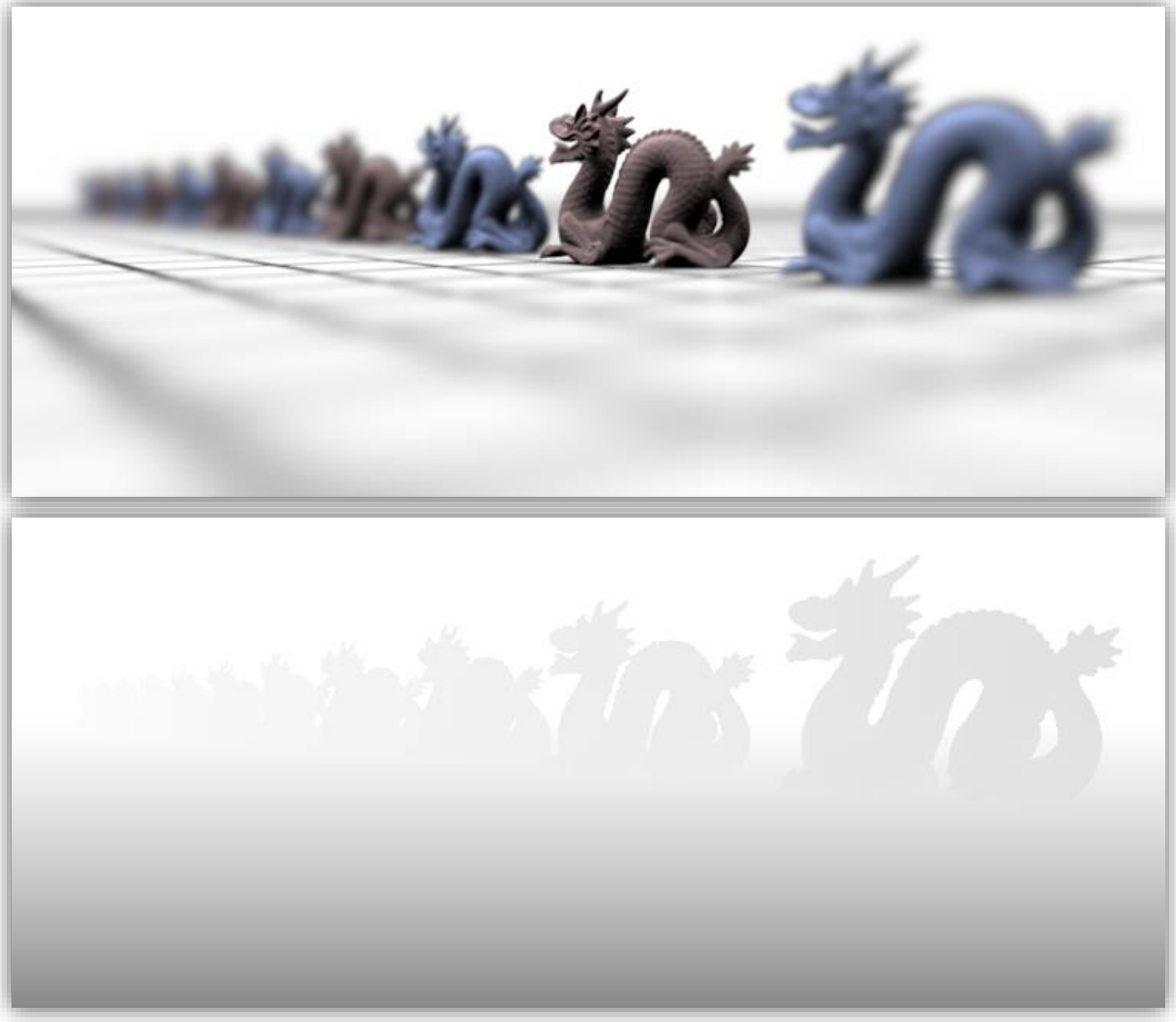
Screen Space Information



Deep Screen Space Information
(berechnet aus Punktwolken)

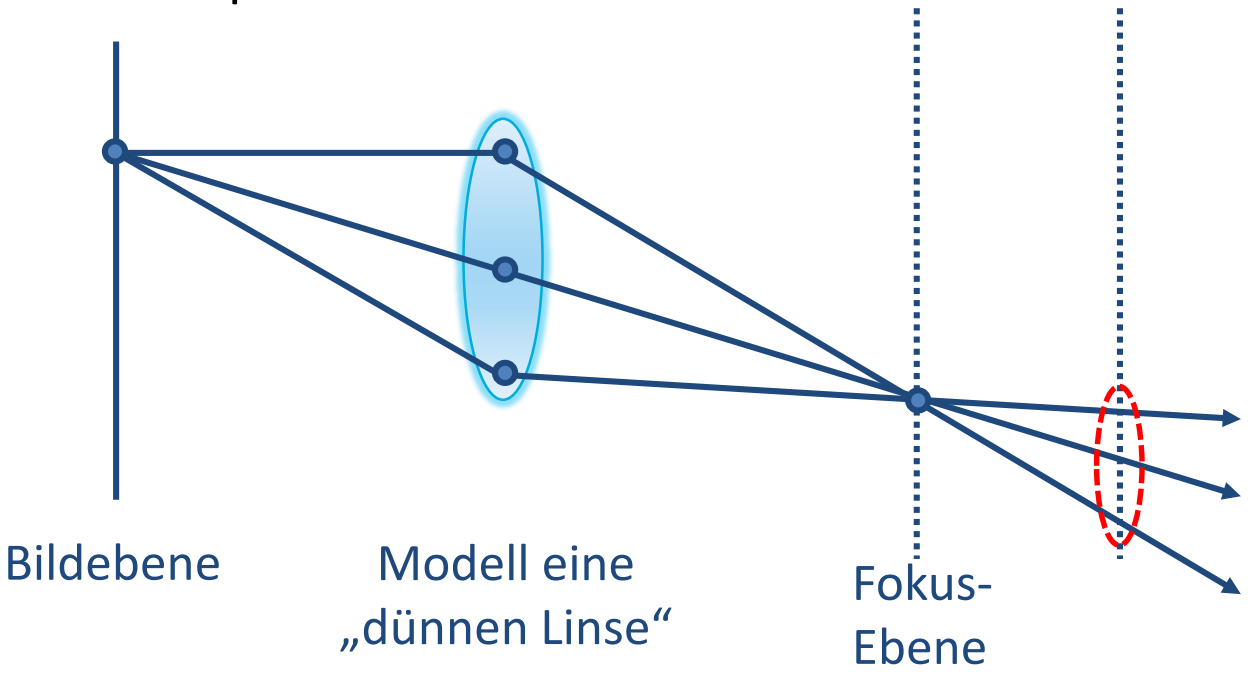
http://people.mpi-inf.mpg.de/~onalbach/DeepScreenSpace/

# Tiefenunschärfe



# Tiefenunschärfe

- plausible Tiefenunschärfe-Effekte lassen sich ebenfalls durch einen „Post-Processing“-Schritt erreichen – ohne das tatsächlich Strahlen verfolgt werden
- Gegenstände in der Fokus-Ebene werden scharf abgebildet – je weiter weg sie davon liegen, desto unschärfer erscheinen sie
  - Grad der Unschärfe durch den „Circle of Confusion“ (CoC) beschrieben
- Idee: verwende einen Unschärfefilter (z.B. Gauß-Filter), dessen Größe sich dem CoC  anpasst

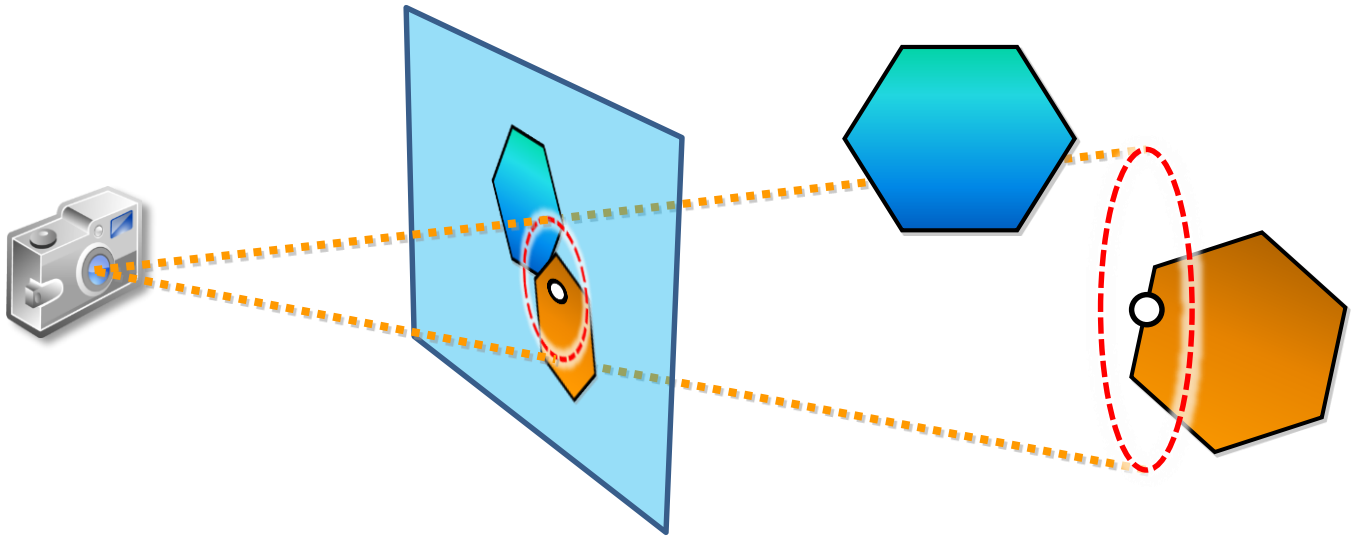


# Tiefenunschärfe

## Tiefenunschärfe-Effekte durch Bildfilter

http://developer.amd.com/wordpress/media/2012/10/Scheuermann_DepthOfField.pdf

- reines Post-Processing, d.h. das fertige Farbbild wird – unter Heranziehen des Tiefenpuffers – gefiltert
- Farbe von Flächen im Fokus, sollen nicht mit der Farbe von Flächen hinter der Fokusebene vermischt werden
- einfacher Trick: Gewichtung der Farbwerte nach Abstand zum gerade betrachteten Punkt (keine Details hier!)

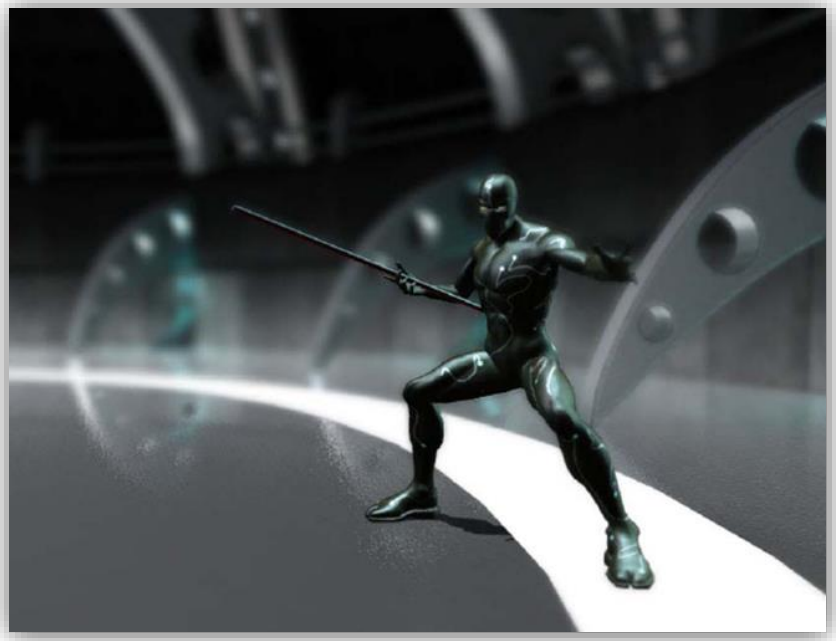


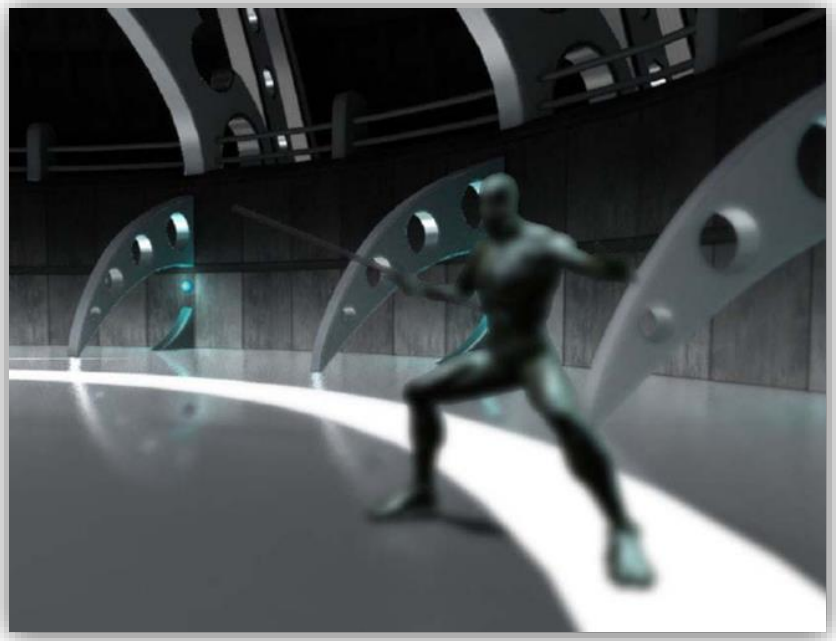
# Tiefenunschärfe

## Tiefenunschärfe-Effekte durch Bildfilter

http://developer.amd.com/wordpress/media/2012/10/Scheuermann_DepthOfField.pdf

- reines Post-Processing, d.h. das fertige Farbbild wird – unter Heranziehen des Tiefenpuffers – gefiltert

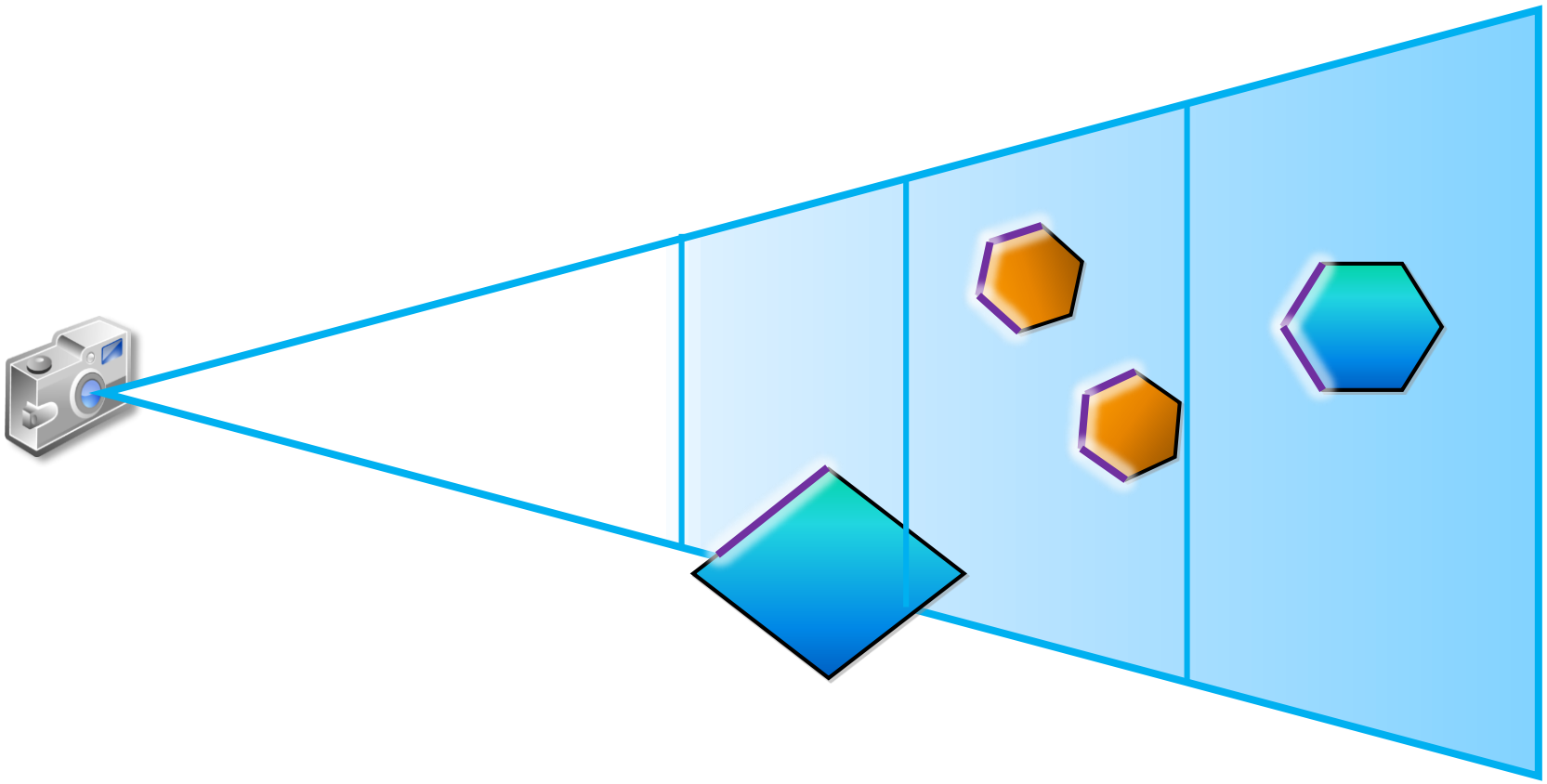




- effizienteres Filterung durch vorgefilterte Texturen (Mipmaps)
- … ein einfaches, günstiges Verfahren mit Grenzen: Information über verdeckte Flächen fehlt natürlich

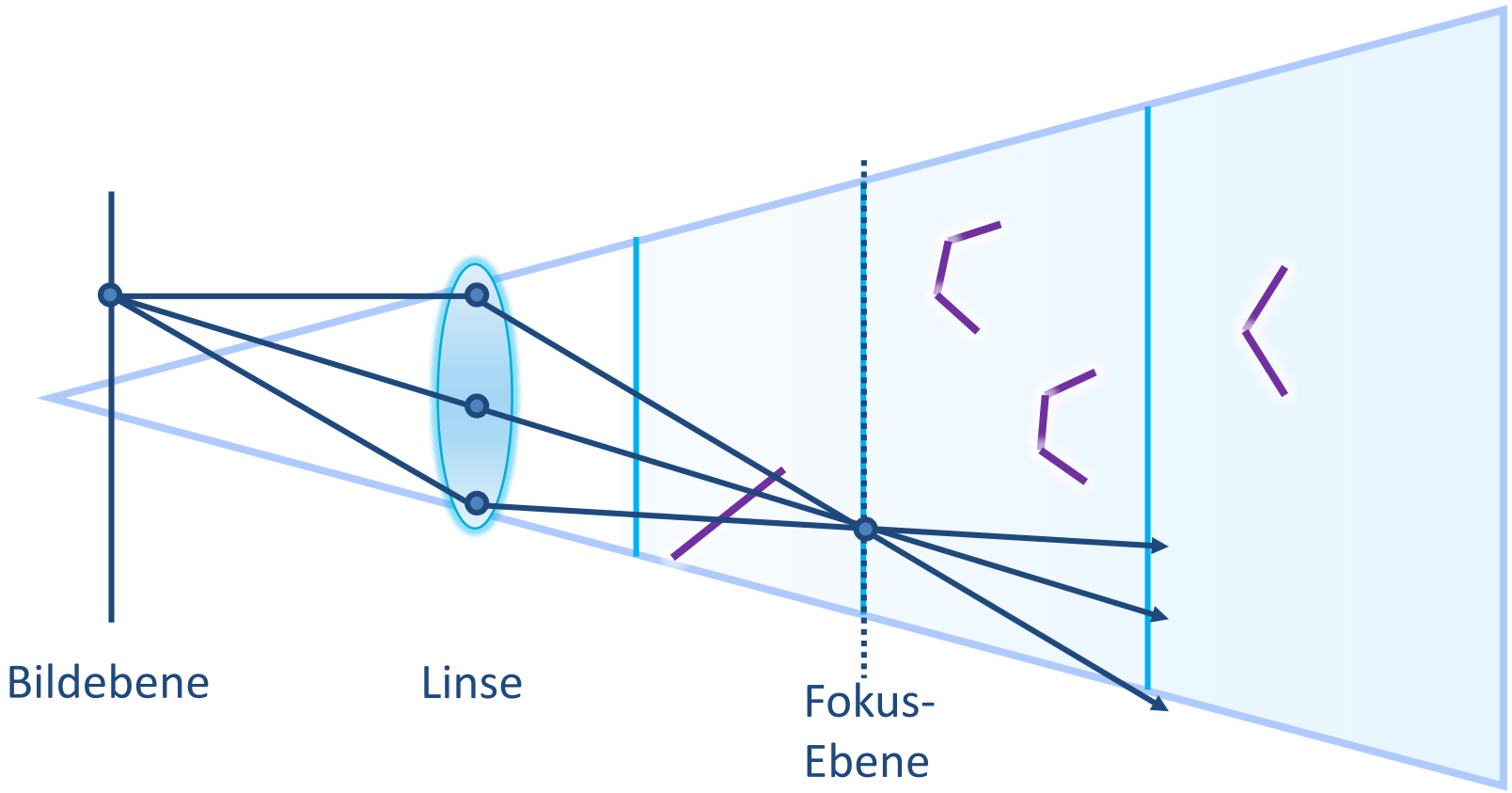
# Tiefenunschärfe

- aktuellere Arbeiten nutzen stochastisches Raytracing in Fragment Programmen
- erster Schritt: erzeuge Tiefenschichten der Sichtpyramide und speichere Farbe und Tiefe



# Tiefenunschärfe

- aktuellere Arbeiten nutzen stochastisches Raytracing in Fragment Programmen
- erster Schritt: erzeuge Tiefenschichten der Sichtpyramide und speichere Farbe und Tiefe
- zweiter Schritt: Raymarching gegen bildbasierte Repräsentation



# Tiefenunschärfe

- Bsp. Real-Time Lens Blur Effects and Focus Control (Lee et al.)

# Tiefenunschärfe/Bokeh (Unreal Engine 3)

- „Bokeh“ bezeichnet die Form der Unschärfebereiche
- ausführlichst diskutiert in: http://research.tri-ace.com/s2015.html

# Tiefenunschärfe/Bokeh (Unreal Engine 3)

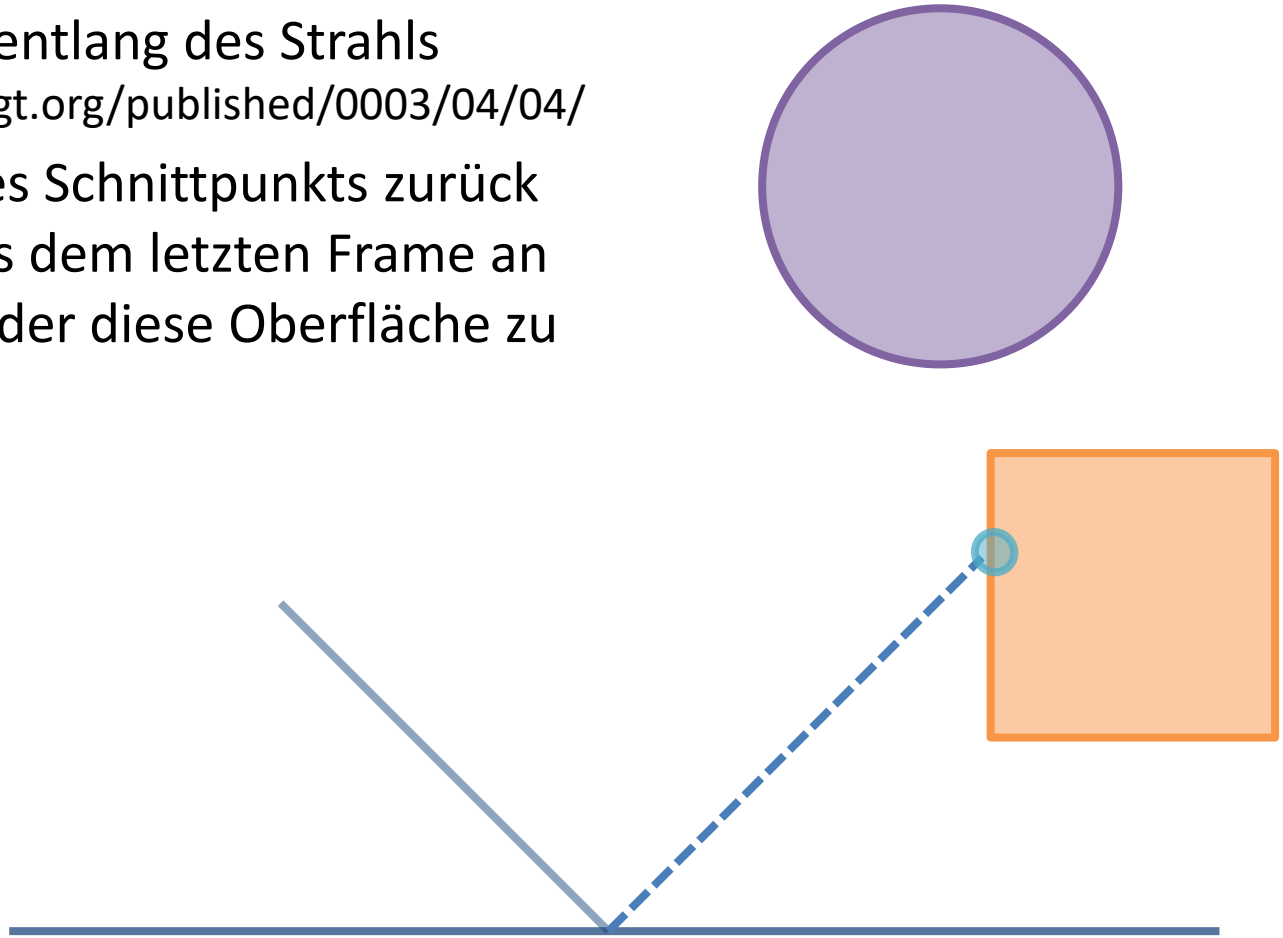
- „Bokeh“ bezeichnet die Form der Unschärfebereiche
- ausführlichst diskutiert in: http://research.tri-ace.com/s2015.html

# Screen Space Reflections (SSR)

## Grundidee für perfekte Spiegelung

- berechne Reflexionsstrahl mit Informationen aus dem G-Buffer
- Raymarching entlang des Strahls
  z.B. mit http://jcgt.org/published/0003/04/04/
- gebe Farbe des Schnittpunkts zurück (z.B. Farbe aus dem letzten Frame an der Stelle, an der diese Oberfläche zu finden war)

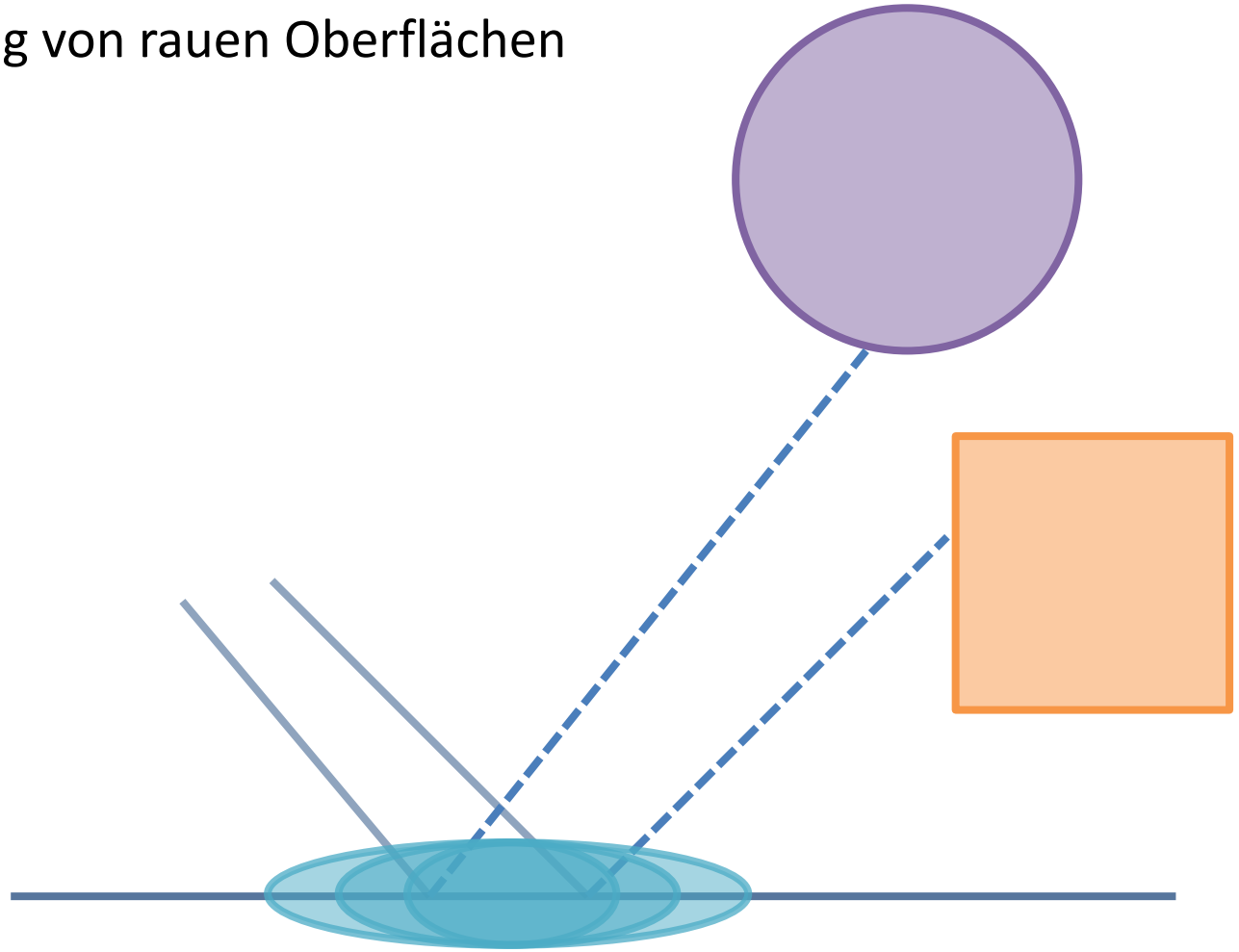
Folien, Abbildungen aus:
Stochastic Screen-Space Reflections,
Stachowiak und Uludag
http://advances.realtimerendering.com/s2015/index.html



# Screen Space Reflections (SSR)

**Einfache Idee für imperfekte Spiegelung (Killzone Shadow Fall)**

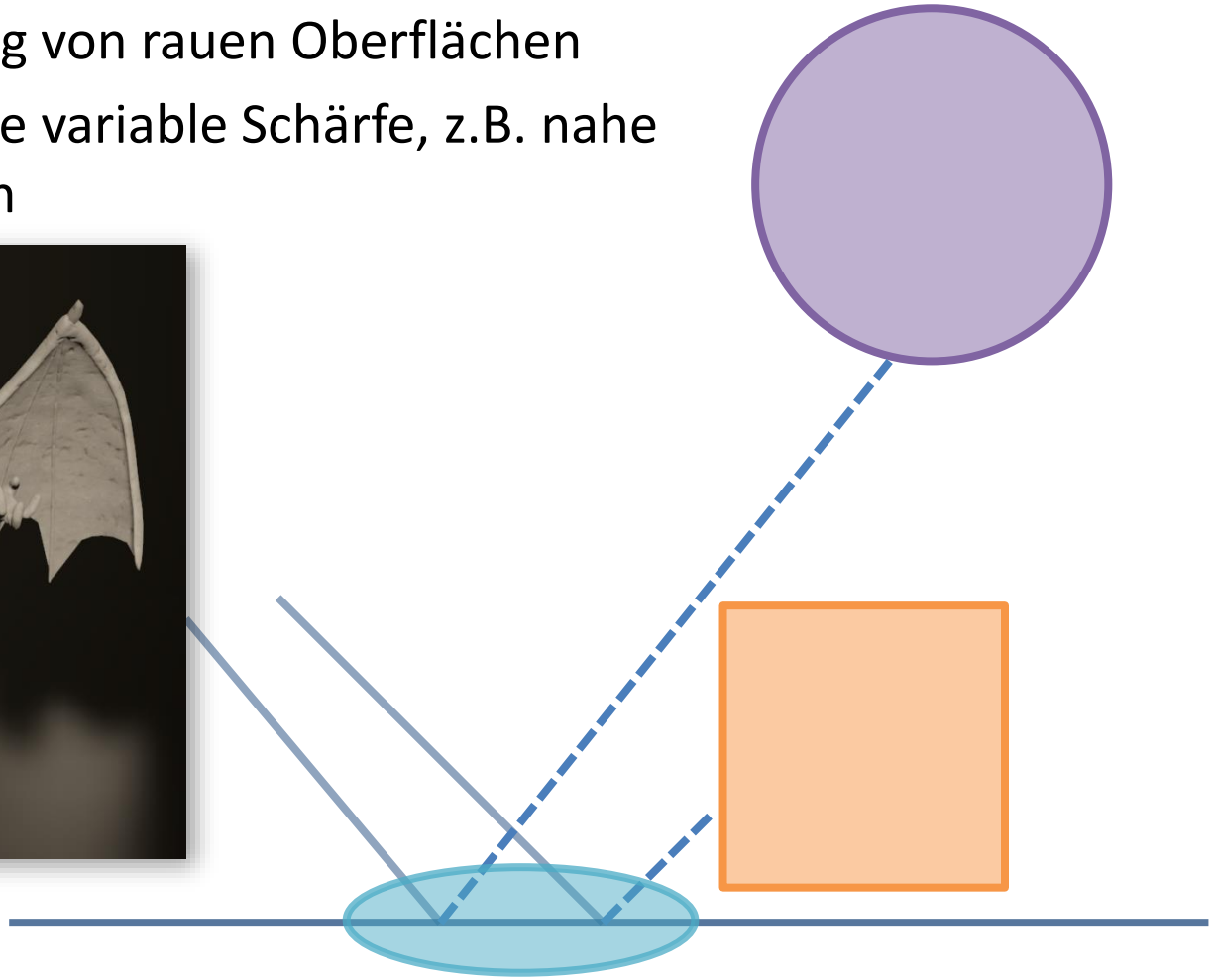
- berechne zunächst perfekte Spiegelung
- verwende vorgefilterte Bilder der Spiegelung zur Darstellung von rauen Oberflächen



# Screen Space Reflections (SSR)

**Einfache Idee für imperfekte Spiegelung (Killzone Shadow Fall)**

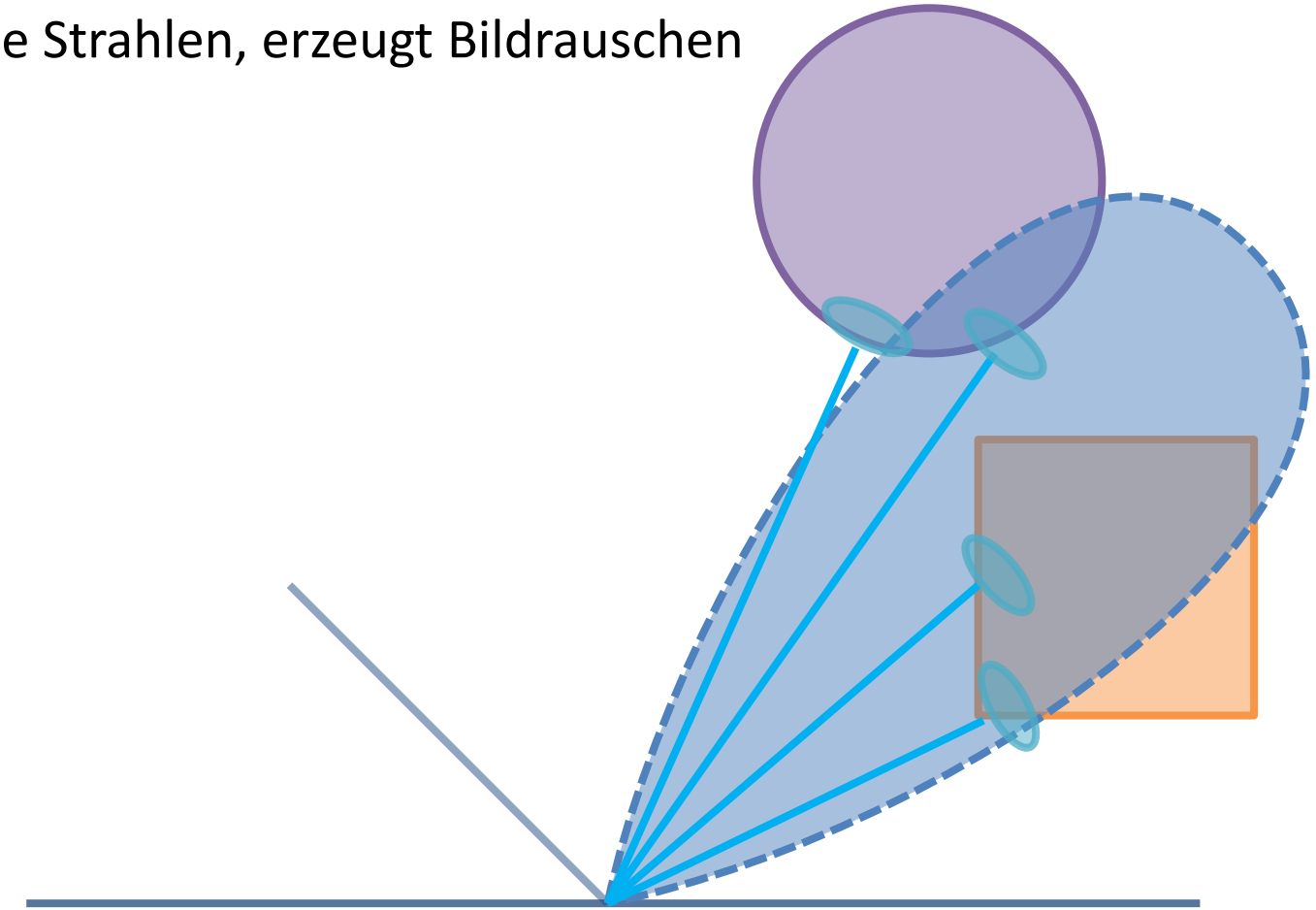
- berechne zunächst perfekte Spiegelung
- verwende vorgefilterte Bilder der Spiegelung zur Darstellung von rauen Oberflächen
- Problem: keine variable Schärfe, z.B. nahe Kontaktkanten



# Screen Space Reflections (SSR)

## Imperfekte Spiegelungen

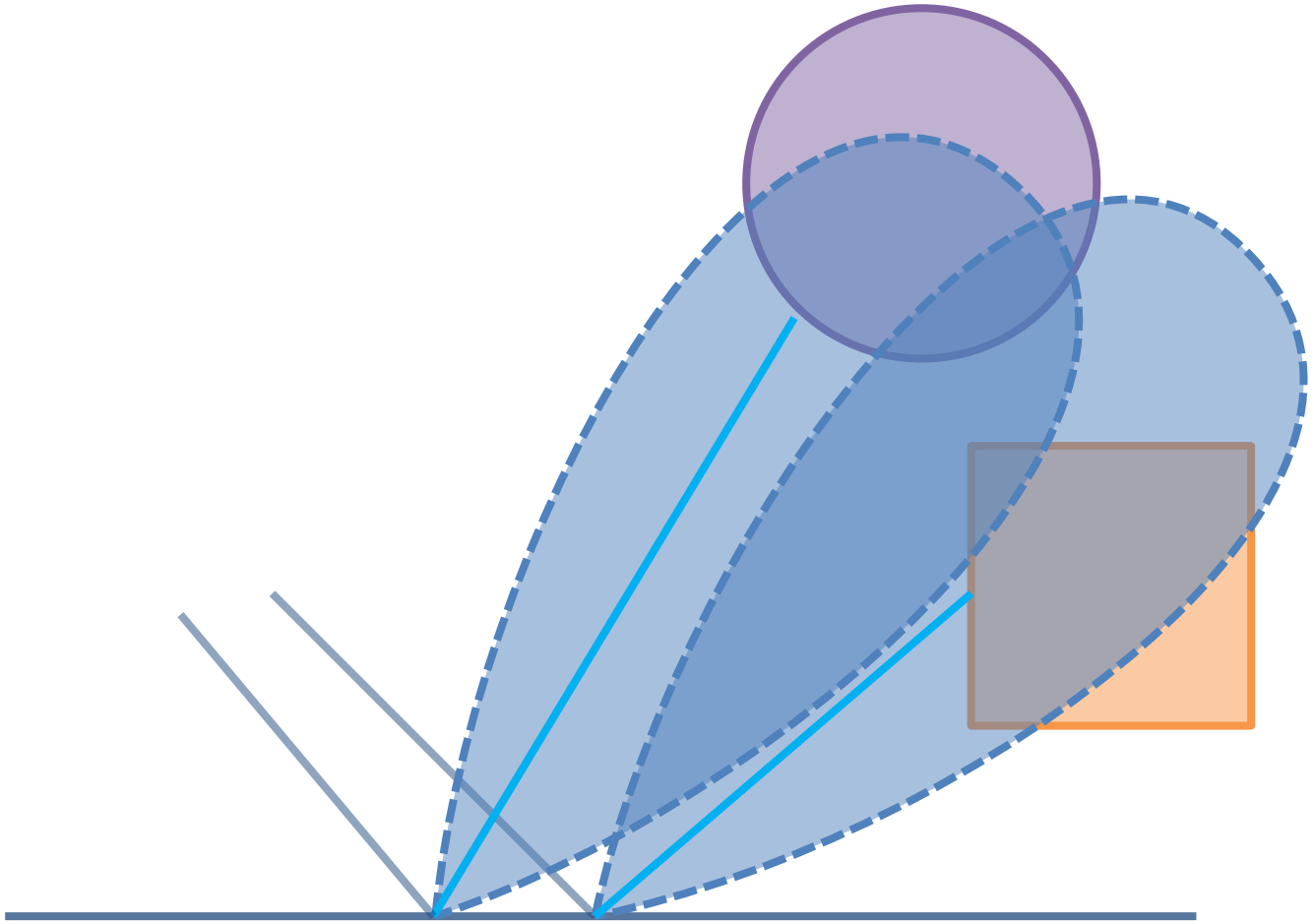
- … erfordern eigentlich ein Abtasten des einfallenden Lichts (vgl. Distributed Raytracing u.a)
- erfordert viele Strahlen, erzeugt Bildrauschen



# Screen Space Reflections (SSR)

## Imperfekte Spiegelungen / Frostbite

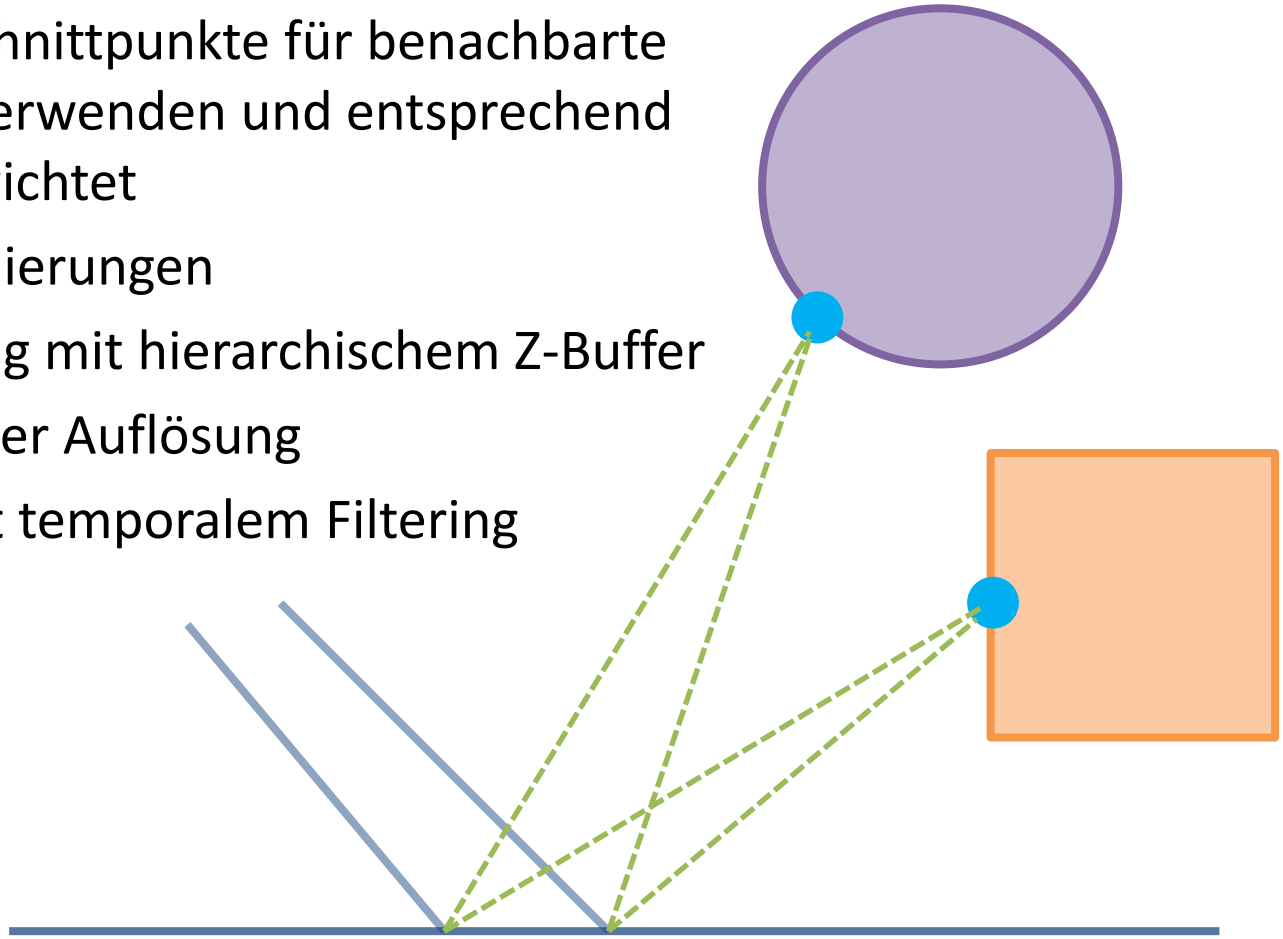
- nur einen oder wenige Strahlen pro Pixel tatsächlich verfolgen



# Screen Space Reflections (SSR)

## Imperfekte Spiegelungen / Frostbite

- nur einen oder wenige Strahlen pro Pixel tatsächlich verfolgen
- gefundene Schnittpunkte für benachbarte Pixel wiederverwenden und entsprechend der BRDF gewichtet
- diverse Optimierungen
  - Raymarching mit hierarchischem Z-Buffer
  - … in niedriger Auflösung
  - … Farbe mit temporalem Filtering



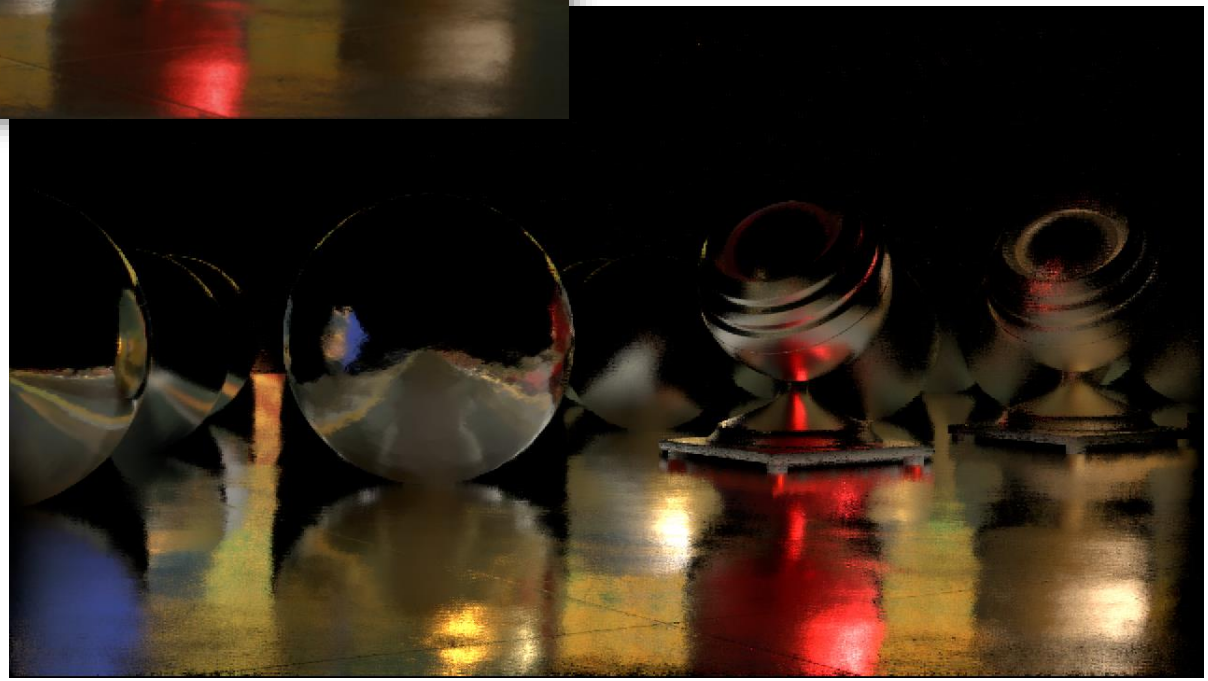

High smoothness


Medium smoothness


Medium smoothness + normals


Variable smoothness

# Screen Space Reflections (SSR)

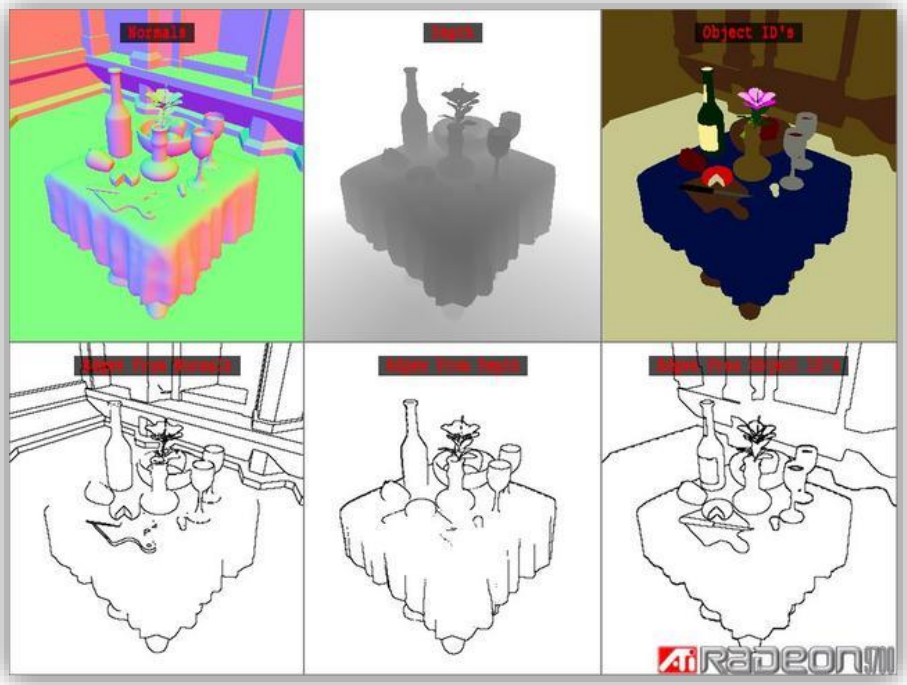


# Blooming

- erzeuge den Effekt von grellem Licht: separiere helle Bildteile (z.B. $x' = \max(0, x - 0.5)$), wende darauf einen Unschärfefilter an ($g * x'$) und addiere das Resultat zum ursprünglichen Bild ($x \mathrel{+}= x'$)
- Details: Lighting and Material of HALO 3, http://www.bungie.net/inside/publications.aspx

# Deferred Shading - Ausblick

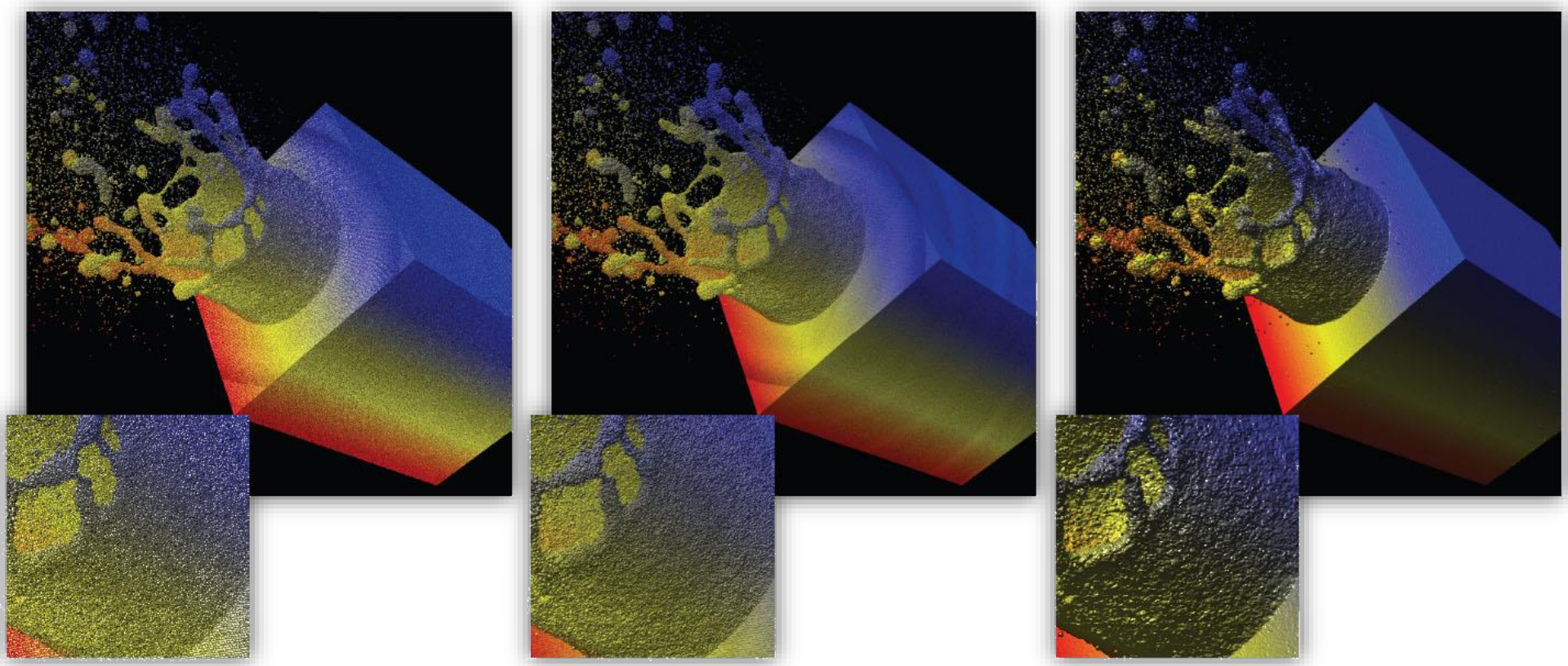
- die Information in den G-Buffers kann vielfältig genutzt werden
  - hier zur Kanten- und Schattenkantendetektion
  - kombiniert mit Hatching für „Non-Photorealistic Rendering“
  - nicht mehr auf den ATI-Webseiten, aber leicht zu finden „ATI Radeon 9700 Hatching“



# Ausblicke

- Anwendung von Deferred Shading in der Visualisierung von Molekulardynamik-Simulationen

  Coherent Culling and Shading for Large Molecular Dynamics Visualization, Grottel et al., Computer Graphics Forum (Proceedings of EuroVis 2010), 2010



# Fazit Bildraumtechniken

- weit verbreitet, meist in Verbindung mit Deferred Shading eingesetzt
- Vorteile von Bildraumtechniken
  - relativ effizient und einfach auf GPUs umzusetzen
  - Ausnutzen der Projektion auf 2D, meist „lokale“ Operationen (z.B. Zugriff auf Texel in der Nachbarschaft)
  - Aufwand ist (nahezu) unabhängig von der Szenenkomplexität, der Geometrierepräsentation und der Beleuchtungsberechnung
- Schwierigkeiten
  - Artefakte durch unvollständige Information, z.B. fehlende Verschattung / fehlendes indirektes Licht
  - → Artefakte, die v.a. bei Animationen störend sein können
  - mehr Informationen können durch Depth Peeling/OIT erhalten werden (ab wann lohnt sich Voxelisierung?)