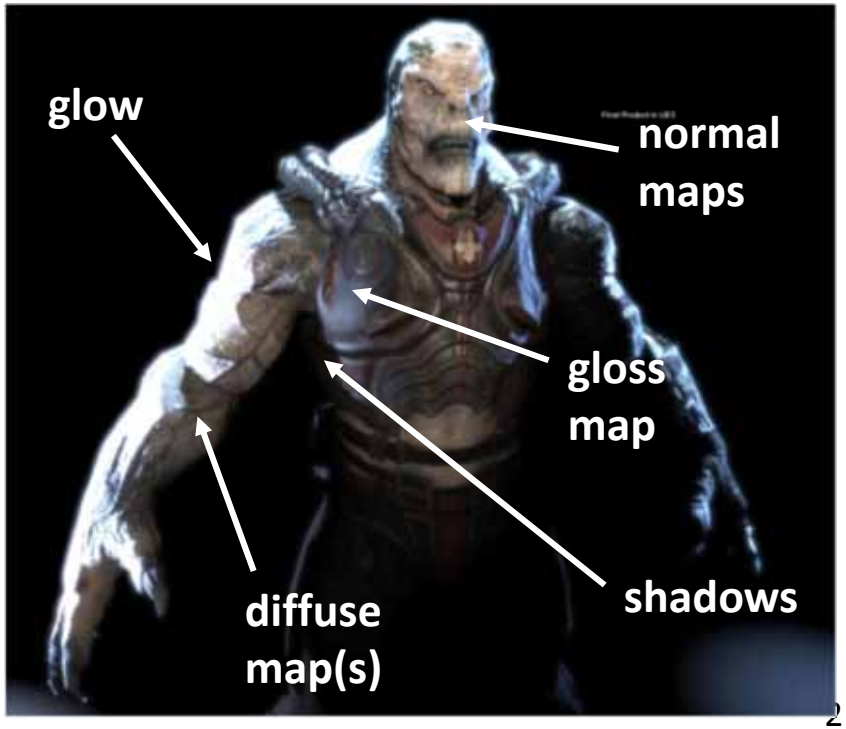
Interaktive Computergrafik

Vorlesung im Sommersemester 2014
Kapitel 1: Einführung, Normal und Displacement-Mapping (Teil 2)

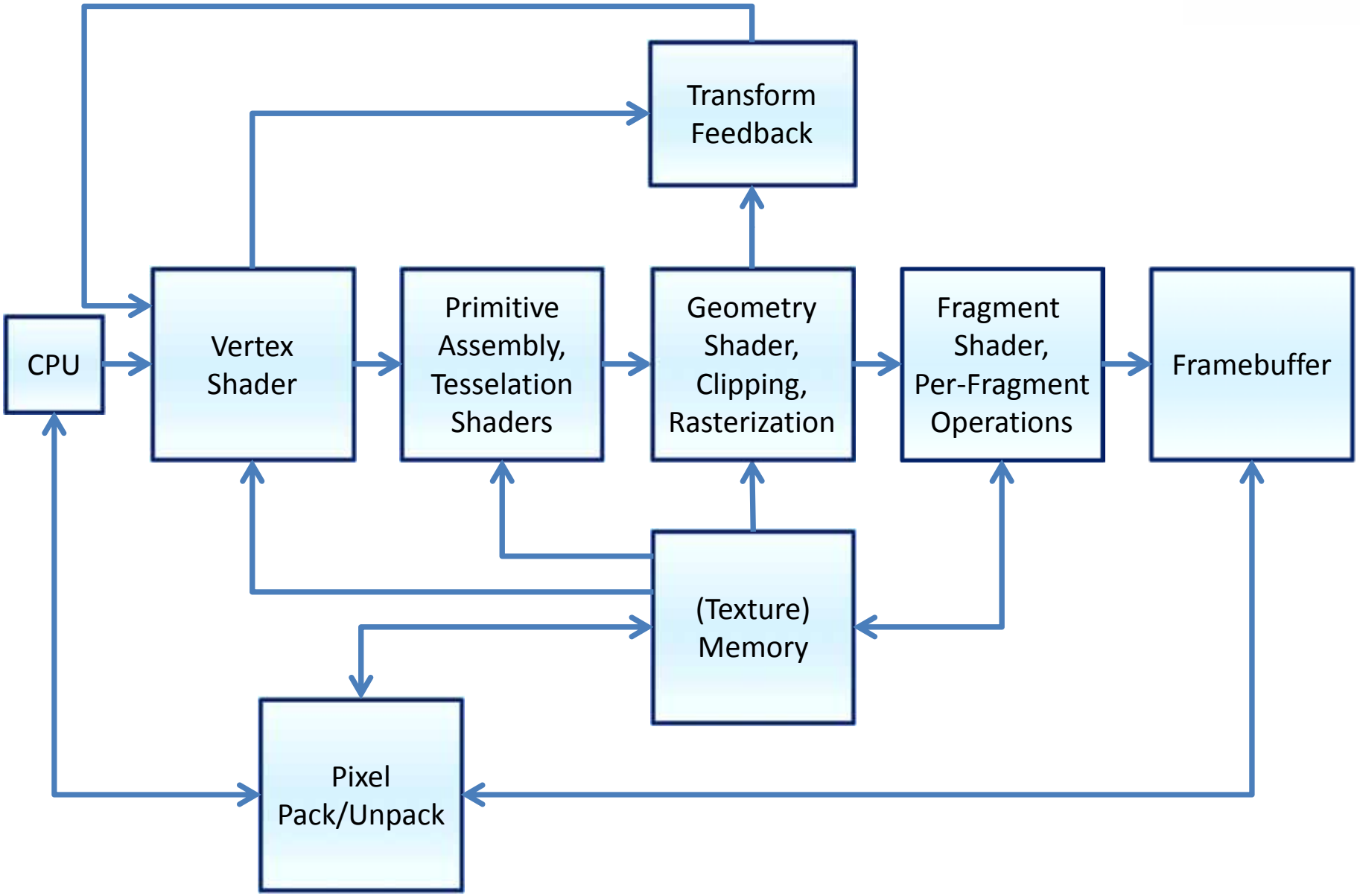
Prof. Dr.-Ing. Carsten Dachsbacher
Lehrstuhl für Computergrafik
Karlsruher Institut für Technologie

Die nächsten Schritte

- InCG verwendet fast immer Grafik-Hardware und meist Rasterisierung
  - wir orientieren uns an modernem OpenGL
  - die Techniken sind aber API-invariant, auch oft hybrid CPU-GPU, …
- Überblick über dieses Kapitels
  - eine kurze Wiederholung zu OpenGL und Shader
  - Rendering-Techniken für detaillierte Oberflächen: Normal und Displacement Mapping



OpenGL(4.x)-Pipeline



Geometrieverarbeitung: Vertex Shader

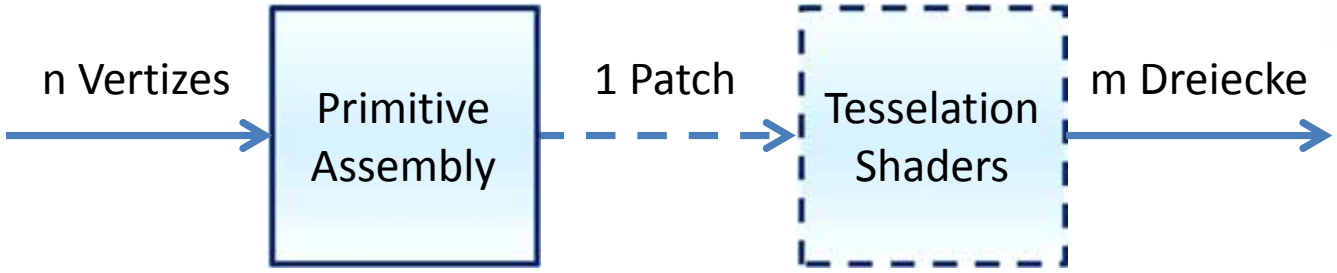
- Programmierung mit der C-ähnlichen OpenGL Shading Language (GLSL)
  - im Kompatibilitätsprofil, d.h. wenn man OpenGL-Altlasten mitschleppt: Zugriff auf OpenGL Zustände (Matrizen, Lichtquellen, …)
- Transformation **einzelner** Vertices und Verarbeitung deren Attribute
- Beispiel eines Vertex Shader / Vertex Program im Kompatibilitätsprofil:

```
void main() {
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Geometrieverarbeitung: Vertex Shader

- Transformation **einzelner** Vertices und Verarbeitung deren Attribute
  - keine Vertex-Erzeugung
  - keine Vertex-Löschung
  - keine Informationen über andere Vertizes
- Berechnung von Attributen, die für Fragmente interpoliert werden sollen
  - z.B. Beleuchtung per Vertex (Gouraud Shading, altmodisch) oder
  - Normalen für Phong Shading

Geometrieverarbeitung: Assembly + Tesselation

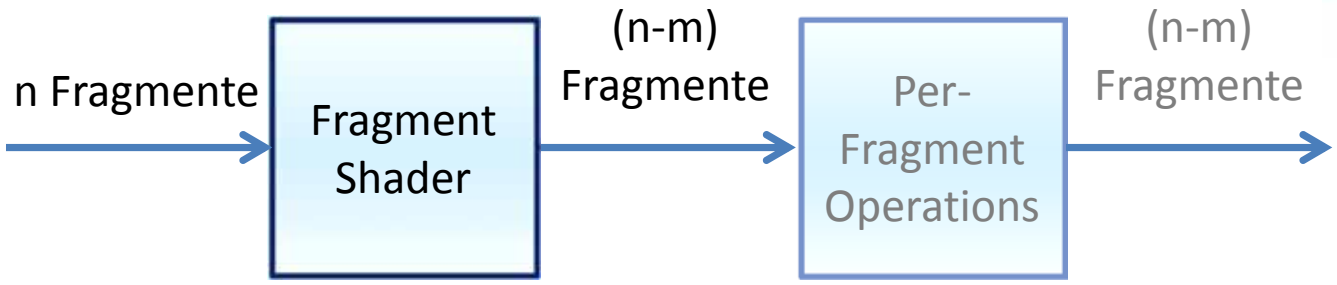


- **Primitive Assembly** setzt aus den Vertices die angeforderten Primitive zusammen und ist **nicht programmierbar**:
  - `GL_POINT, GL_LINE[*], GL_TRIANGLE[*]` in ihren unterschiedlichen Modi `*_FAN, *_LOOP, *_STRIP`
- **Tesselation Shaders**
  - Tesselation Control Shader bestimmt die Unterteilung
  - Tesselator führt sie durch (nicht programmierbar)
  - Tesselation Evaluation Shader arbeitet auf der resultierenden Geometrie
  - spezieller Primitivtyp `GL_PATCH`

Geometrieverarbeitung: Geometry Shader etc.

- Geometry Shader (programmierbar, aber optional)
  - bearbeitet einzelne Primitive (Punkt, Linie, Dreieck)
  - kann Primitive vervielfachen, entfernen oder beliebig umwandeln (Punkte zu Dreiecke etc.)
- Clipping (am Sichtvolumen) ist nicht programmierbar
  - verwirft unsichtbare Punktprimitive
  - verändert Dreiecke durch Schnitt mit Sichtvolumen
- anschließend: perspektivische Division (nicht programmierbar)
- Rasterisierung generiert Fragmente für den Ausgabepuffer…

Fragmentverarbeitung

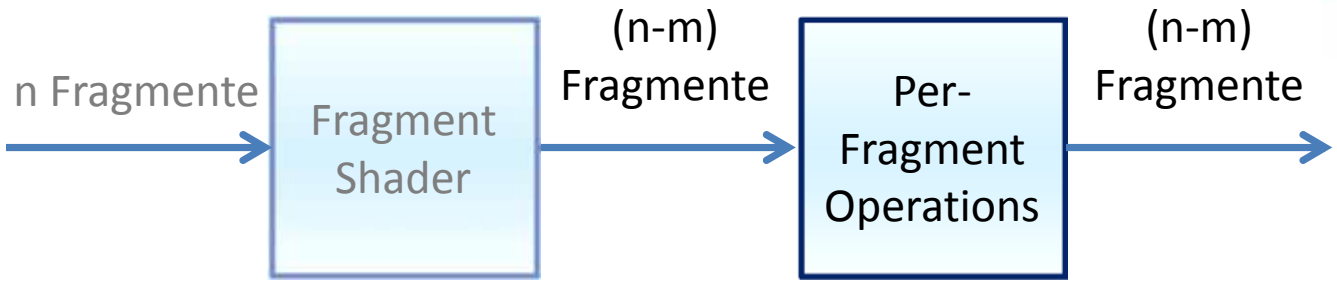


- …für jedes dieser Fragmente wird ein **Fragment Shader/Program** ausgeführt
  - Berechnung von **Farbe**, **Transparenz**, und optional **Tiefe** pro Fragment
    - z.B. Phong Shading, also Beleuchtungsberechnung pro Pixel
  - Eingabe-Attribute werden innerhalb des Primitivs interpoliert
    - z.B. pro-Eckpunkt Normalen, die im Vertex Shader weitergegeben wurden

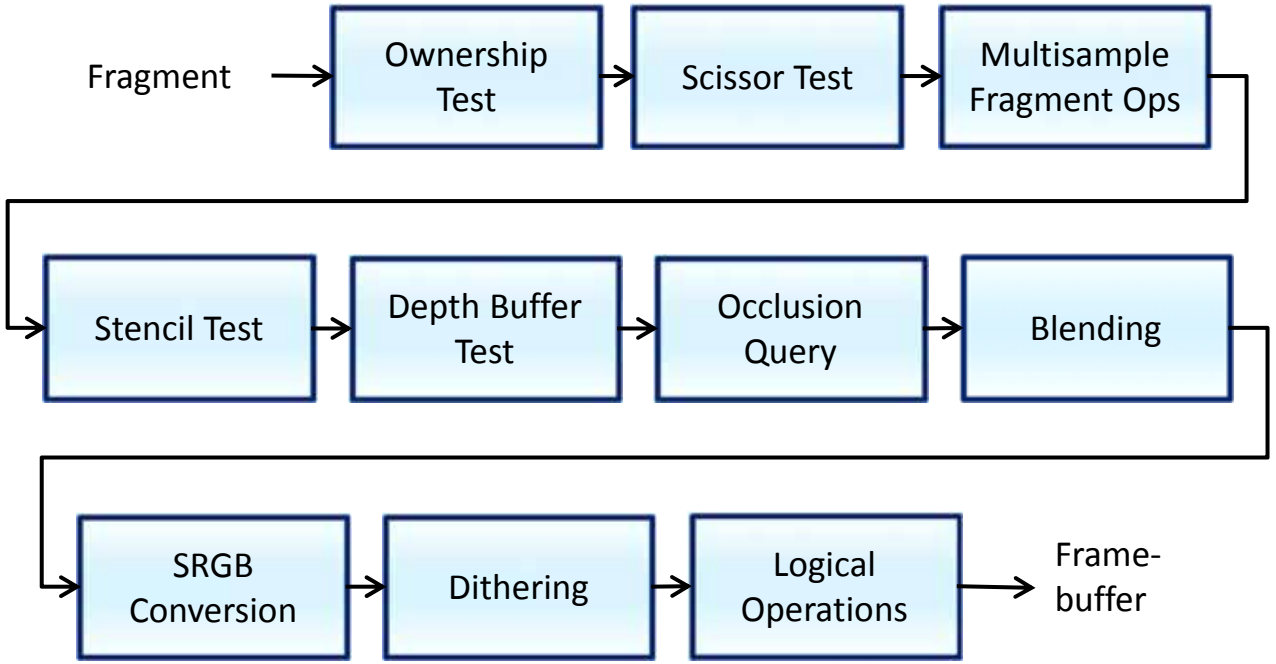
- Beispiel eines Fragment Program im OpenGL 2.0 oder Kompatibilitätsprofil:

```
void main() {
    gl_FragColor = vec4( 1.0, 1.0, 1.0, 0.0 );
}
```

Fragmentverarbeitung

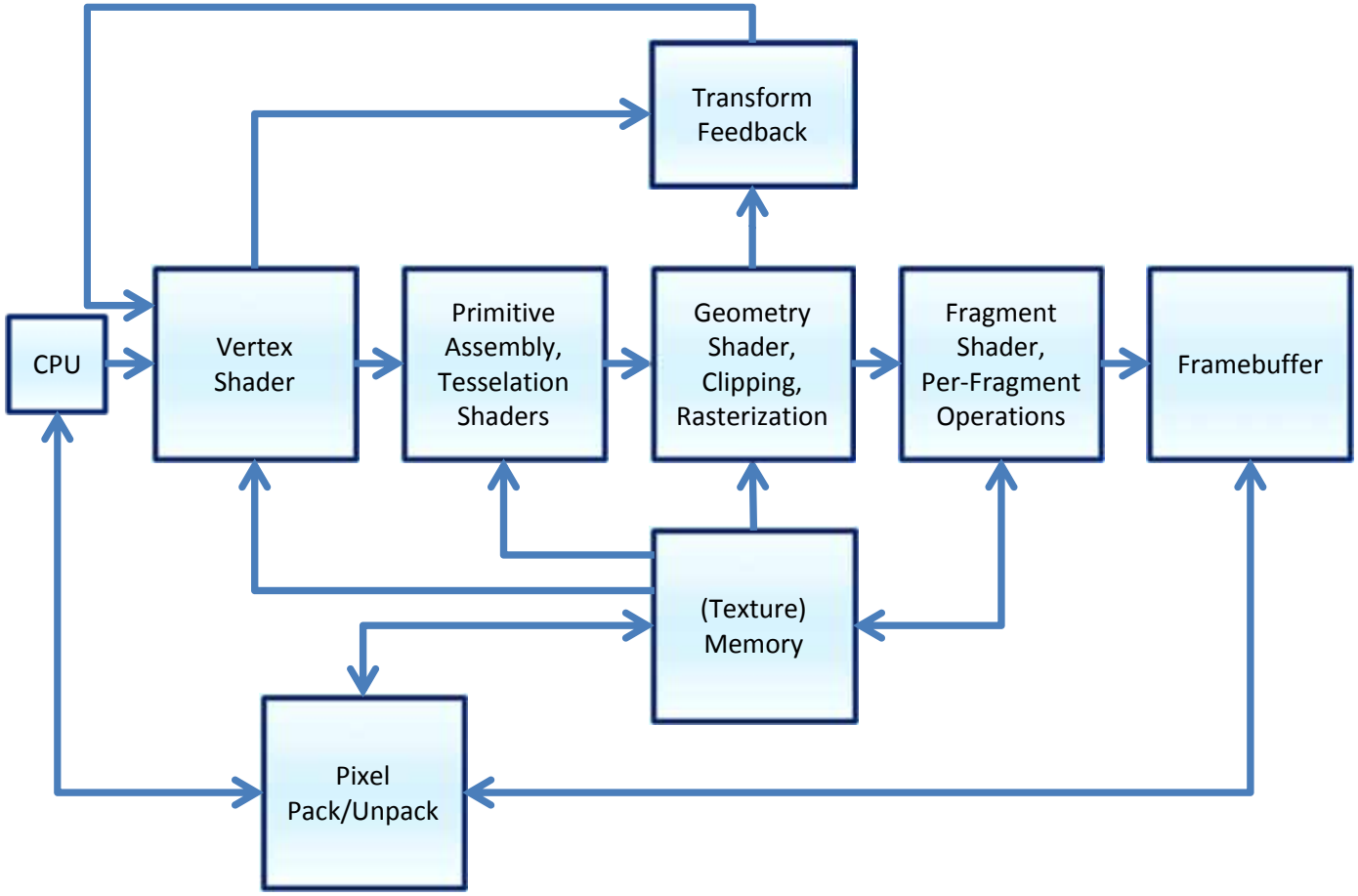


- Fragmentoperationen sind sehr wichtig für Rasterisierungsverfahren
  - Tiefentest, Blending, …



Programmierbare Pipeline allgemein

- jede programmierbare Stufe kann wahlfrei lesend auf Puffer zugreifen
- Ausnahme: der Framebuffer kann nur geschrieben werden
- Transform Feedback schreibt Ergebnisse der Geometrieverarbeitung in einen Puffer



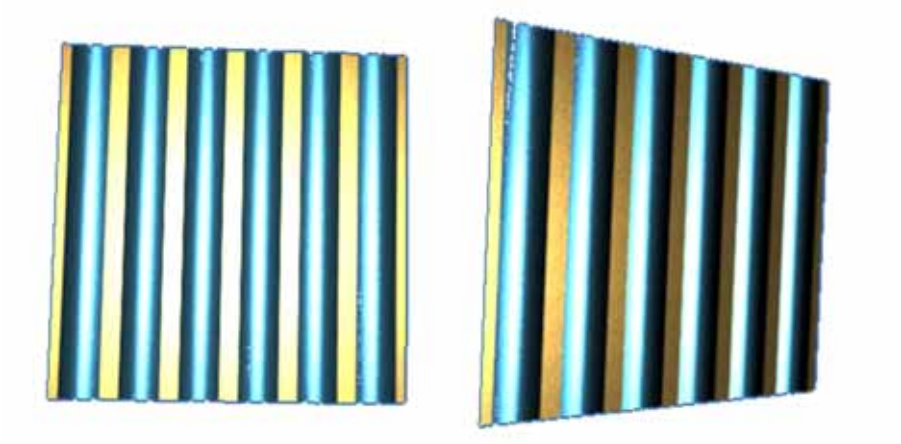
Fahrplan

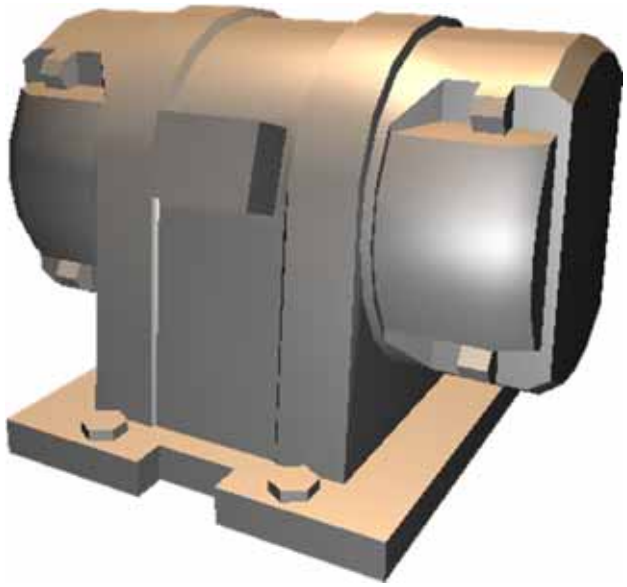
Unsere nächsten Schritte…

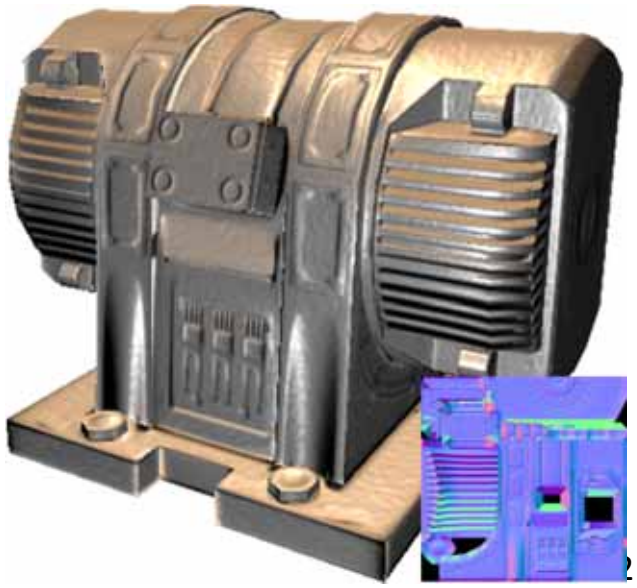
- mehr Oberflächendetail → realistische Beleuchtung
- Schattenverfahren und Voxelisierung → realistische Beleuchtung
- effizientes Shading in komplexen Szenen → Effizienz
- vorberechneter Lichttransport → Effizienz & Realismus
- Culling und Level of Detail → Effizienz
- …

Per-Pixel vs Bump Mapping

- Per-Vertex Lighting (Gouraud Shading)
- Per-Pixel Lighting (Phong Shading)
  - glatte Oberflächen
  - keine Machschen Band-Artefakte
- Bump/Normal-Mapping
  - raues Oberflächendetail nur in der Beleuchtung
  - Normalenvariation aus einer Textur
  - keine Änderung der Geometrie, deshalb mögliche Artefakte an den Silhouetten

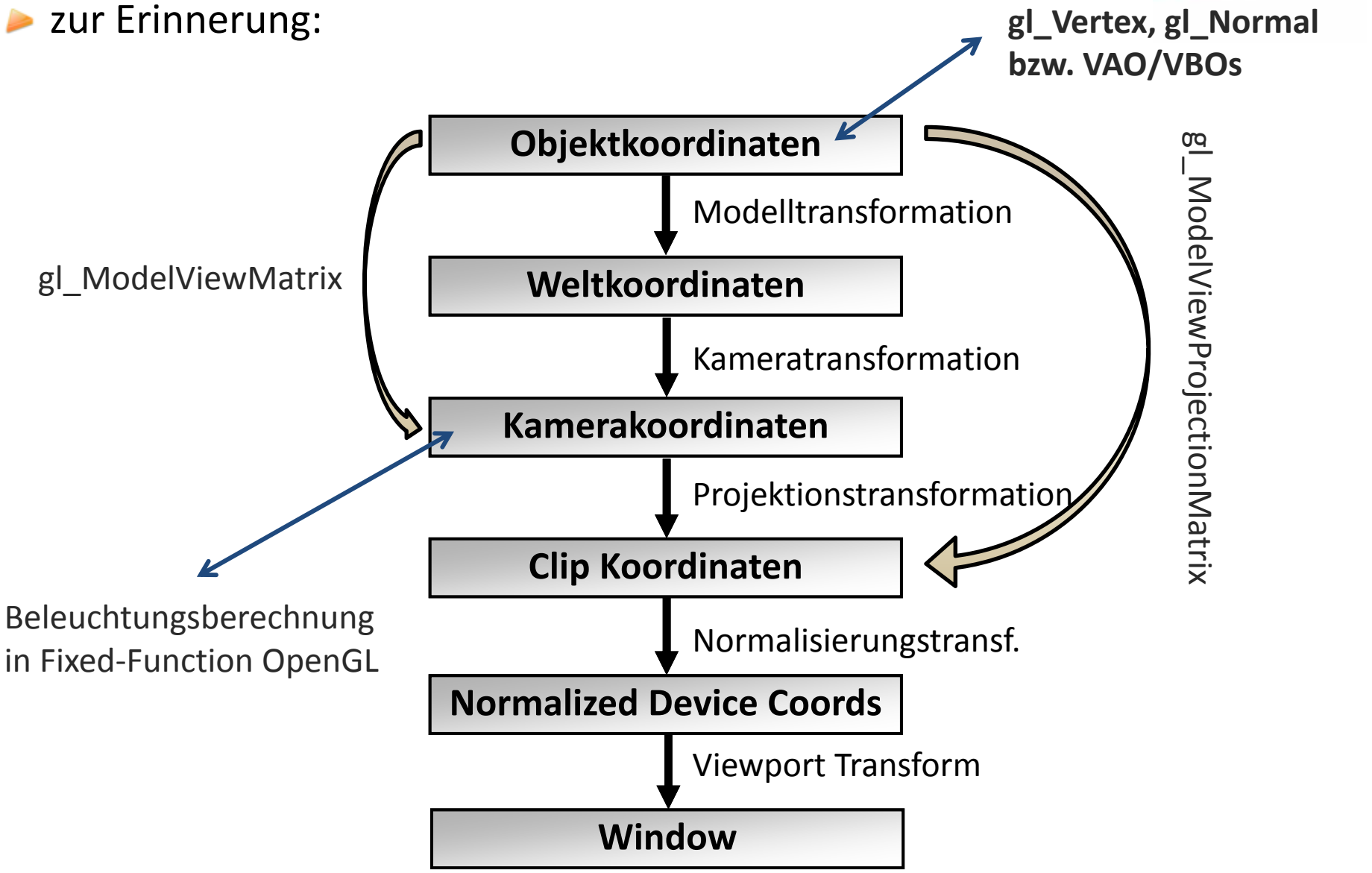






Transformationspipeline

- zur Erinnerung:

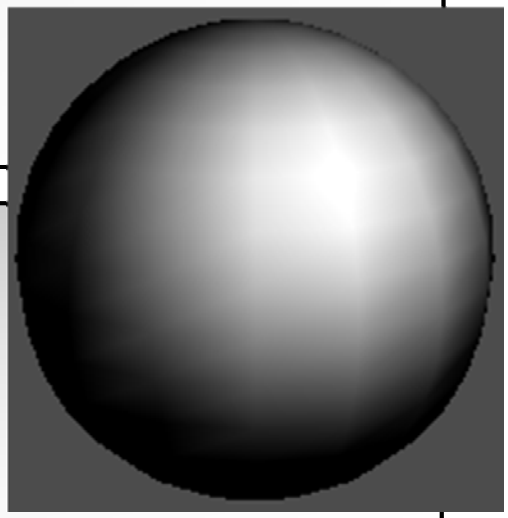


Mehr Infos:
http://www.opengl.org/resources/faq/technical/viewing.htm

GLSL 3.x/4.x Diffuse Beleuchtung

```
uniform mat4 matrixMVP, matrixMV, matrixNrml;
uniform vec3 lightSourcePos;      // Lichtquellenposition Kamerakoord.
in  vec4 in_position;             // Eingabe: Vertexposition
in  vec3 in_normal;               // Eingabe: Vertexnormale
out vec4 color;
void main() {
    gl_Position = matrixMVP * in_position;
    // Beleuchtungsberechnung in Kamerakoordinaten
    vec3 P = vec3( matrixMV * in_position );
    vec3 N = normalize( vec3( matrixNrml * vec4( in_normal, 0.0 ) ) );
    vec3 L = normalize( lightSourcePos – P );
    color = vec4( max( 0.0, dot( L, N ) ) );
}
```

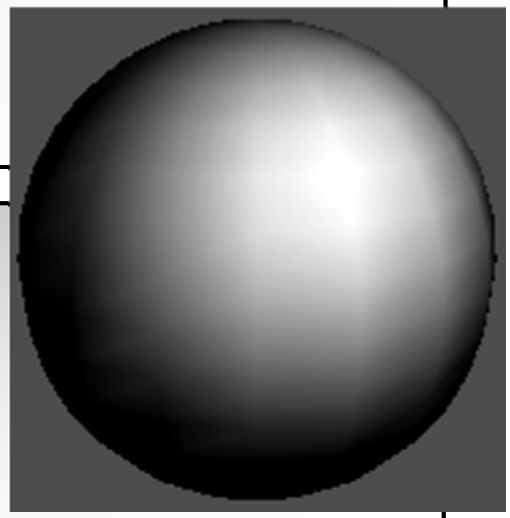
```
in vec4 color;
out vec4 out_color;
void main() {
     out_color = color;
}
```



GLSL 3.x/4.x Diffuse Beleuchtung

```
uniform mat4 matrixMVP, matrixMV, matrixNrml;
uniform vec3 lightSourcePos;        // Lichtquellenposition Kamerakoord.
in  vec4 in_position;               // Eingabe: Vertexposition
in  vec3 in_normal;                 // Eingabe: Vertexnormale
out vec4 color;
void main() {
    gl_Position = matrixMVP * in_position;
    // Beleuchtungsberechnung in Kamerakoordinaten
    vec3 P = vec3( matrixMV * in_position );
    vec3 N = normalize( vec3( matrixNrml * vec4( in_normal, 0.0 ) ) );
    vec3 L = normalize( lightSourcePos – P );
    color = vec4( max( 0.0, dot( L, N ) ) );
}
```

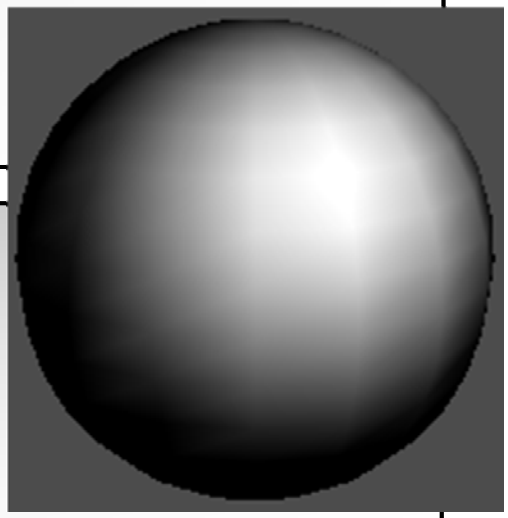
```
in vec4 color;
out vec4 out_color;
void main() {
    out_color = color;
}
```



GLSL 3.x/4.x Diffuse Beleuchtung

```
uniform mat4 matrixMVP, matrixMV, matrixNrml;
uniform vec3 lightSourcePos;        // Lichtquellenposition Kamerakoord.
in  vec4 in_position;               // Eingabe: Vertexposition
in  vec3 in_normal;                 // Eingabe: Vertexnormale
out vec4 color;
void main() {
    gl_Position = matrixMVP * in_position;
    // Beleuchtungsberechnung in Kamerakoordinaten
    vec3 P = vec3( matrixMV * in_position );
    vec3 N = normalize( vec3( matrixNrml * vec4( in_normal, 0.0 ) ) );
    vec3 L = normalize( lightSourcePos – P );
    color = vec4( max( 0.0, dot( L, N ) ) );
}
```

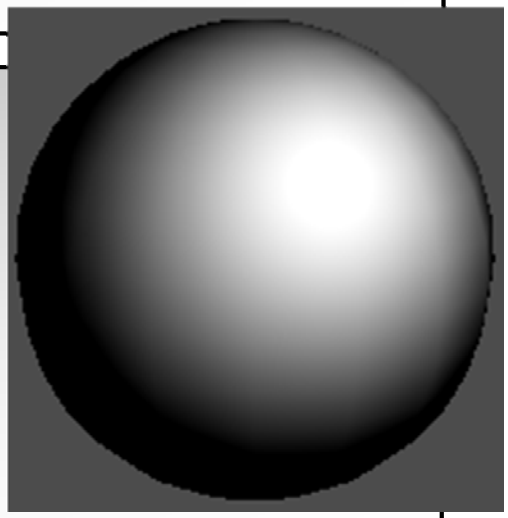
```
in vec4 color;
out vec4 out_color;
void main() {
    out_color = color;
}
```



GLSL 3.x/4.x „Per-Pixel“ Beleuchtung

```
uniform mat4 matrixMVP, matrixMV, matrixNrml;
uniform vec3 lightSourcePos;        // Lichtquellenposition Kamerakoord.
in  vec4 in_position;               // Eingabe: Vertexposition
in  vec3 in_normal;                 // Eingabe: Vertexnormale
out vec3 L, N;
void main() {
    gl_Position = matrixMVP * in_position;
    vec3 P = vec3( matrixMV * in_position );
    N = normalize( vec3( matrixNrml * vec4( in_normal, 0.0 ) ) );
    L = normalize( lightSourcePos – P );
}
```

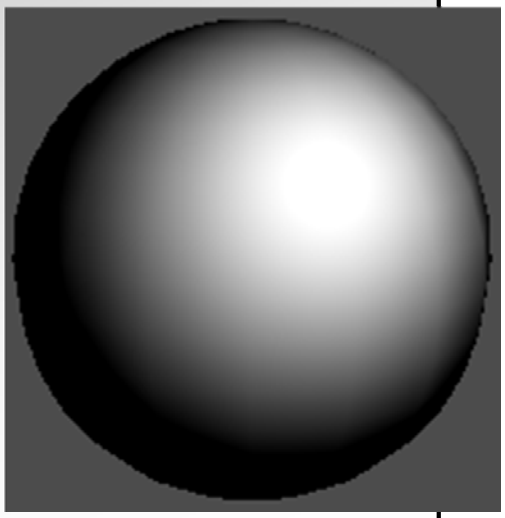
```
in vec3 L, N;
out vec4 out_color;
void main() {
    float kd = max( 0.0, dot( normalize(L),
                              normalize(N) ) );
    out_color = vec4( kd );
}
```



GLSL 3.x/4.x „Per-Pixel“ Beleuchtung (ineffizient)

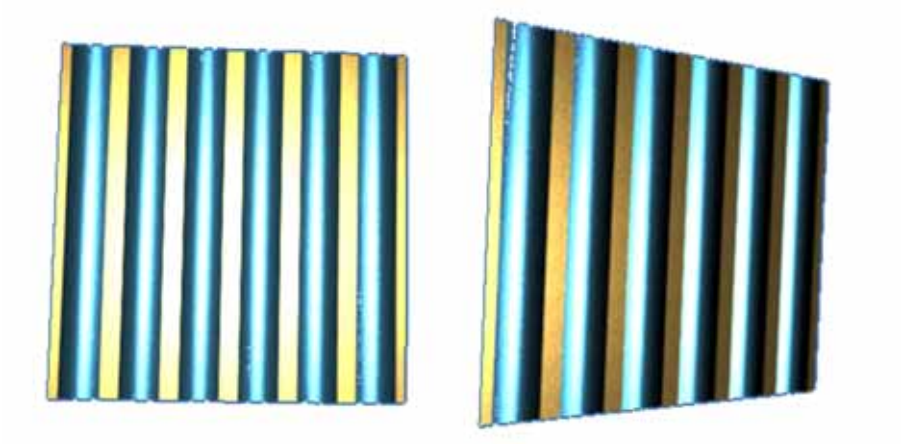
```
uniform mat4 matrixMVP;
in  vec4 in_position;                  // Eingabe: Vertexposition
in  vec3 in_normal;                    // Eingabe: Vertexnormale
out vec4 pos, normal;
void main() {
    gl_Position = matrixMVP * in_position;
    pos = in_position; normal = vec4( in_normal, 0.0 );
}
```

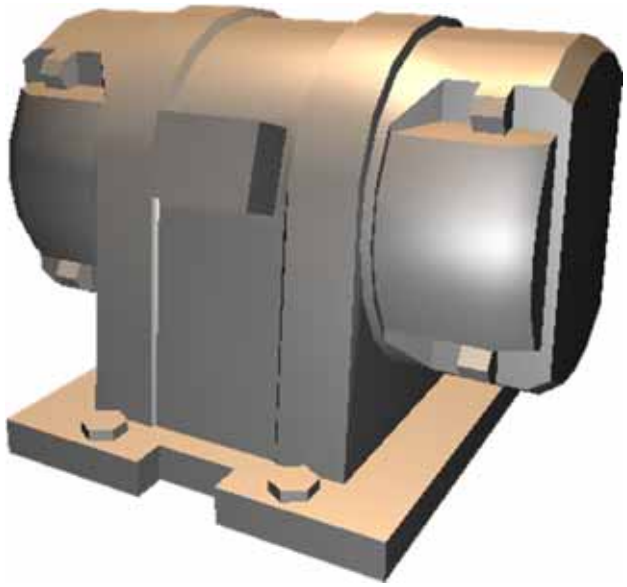
```
uniform mat4 matrixMV, matrixNrml;
uniform vec3 lightSourcePos;           // Lichtquellenposition Kamerakoord.
in vec4 pos, normal;
out vec4 out_color;
void main() {
    vec3 P = vec3( matrixMV * pos );
    vec3 N = normalize( vec3( matrixNrml * normal ) );
    vec3 L = normalize( lightSourcePos – P );
    float kd = …
}
```

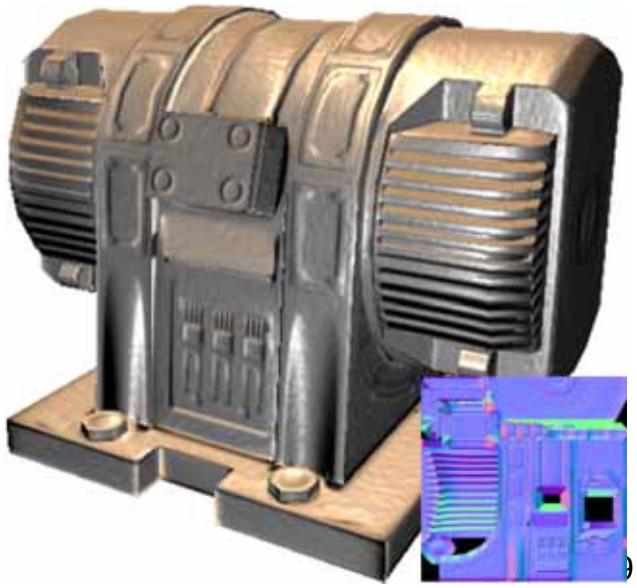


Per-Pixel vs Bump Mapping

- Per-Vertex Lighting (Gouraud Shading)
- Per-Pixel Lighting (Phong Shading)
  - glatte Oberflächen
  - keine Machschen Band-Artefakte
- Bump/Normal-Mapping
  - raues Oberflächendetail nur in der Beleuchtung
  - Normalenvariation aus einer Textur
  - keine Änderung der Geometrie, deshalb mögliche Artefakte an den Silhouetten



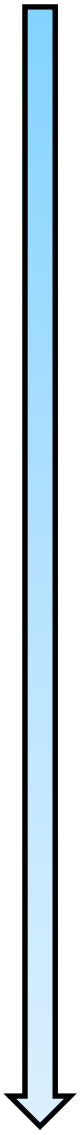




Bump/Normal Mapping

Per-Pixel vs Bump Mapping

- Per-Vertex Lighting (Gouraud Shading)
- Per-Pixel Lighting (Phong Shading)
  - glatte Oberflächen
  - keine Machschen Band-Artefakte
- Bump/Normal-Mapping
  - raues Oberflächendetail nur in der Beleuchtung
  - Normalenvariation aus einer Textur
  - keine Änderung der Geometrie, deshalb mögliche Artefakte an den Silhouetten
- Displacement-Mapping
  - geometrisches Oberflächendetail
- Netze, Polygone



mikroskopisches Detail
im Beleuchtungsmodell
(siehe Fotorealistische Bildsyn.)

Mesostrukturen: zu groß für Reflektanzmodelle, zu klein um effizient durch Polygone repräsentiert zu werden

geometrisches Detail

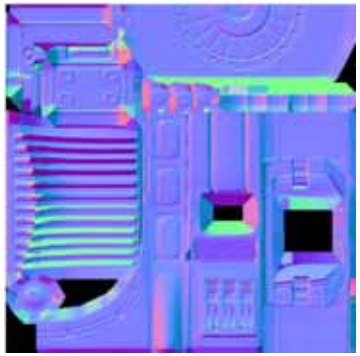
Bump-Mapping: Idee, Problem?

- Grundidee: verwende nicht geometrische Normale, sondern Normalenvektor aus einer Textur für die Beleuchtungsberechnung

```
in vec3 L;
in vec2 texCoord;
void main() {
    float kd = max( 0.0, dot( normalize(L), texture( sampler, texCoord ) ) );
    … }
```

- Frage: in welchem KoSys soll die Berechnung stattfinden?

- eine Möglichkeit: in Objektkoordinaten
  - nur möglich in Verbindung mit einem Texturatlas (für jeden Punkt auf der Oberfläche eine spezifische Normale)

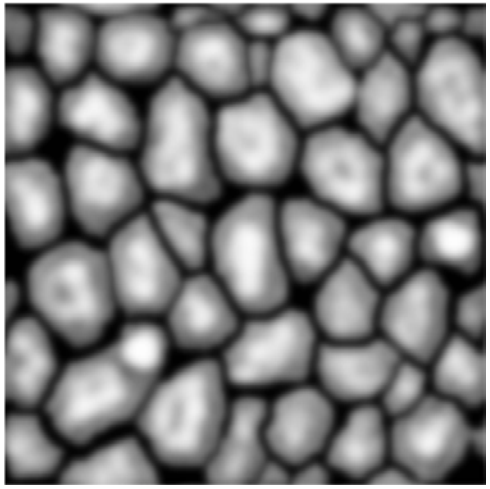


Bump-Mapping: Idee, Problem?

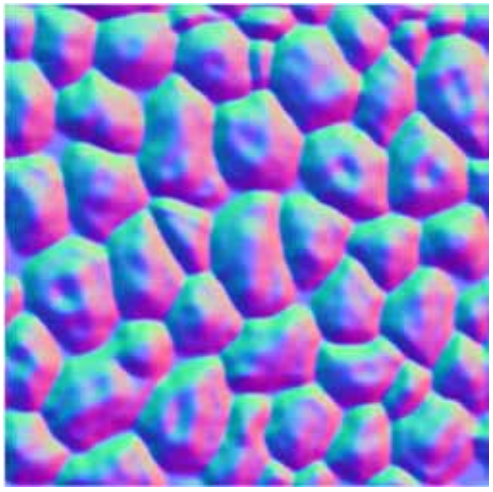
- Grundidee: verwende nicht geometrische Normale, sondern Normalenvektor aus einer Textur für die Beleuchtungsberechnung

```
in vec3 L;
in vec2 texCoord;
void main() {
    float kd = max( 0.0, dot( normalize(L), texture( sampler, texCoord ) ) );
    … }
```

- … funktioniert nicht mit der üblichen Erzeugung von Texturen:



Höhenkarte $B(s,t)$

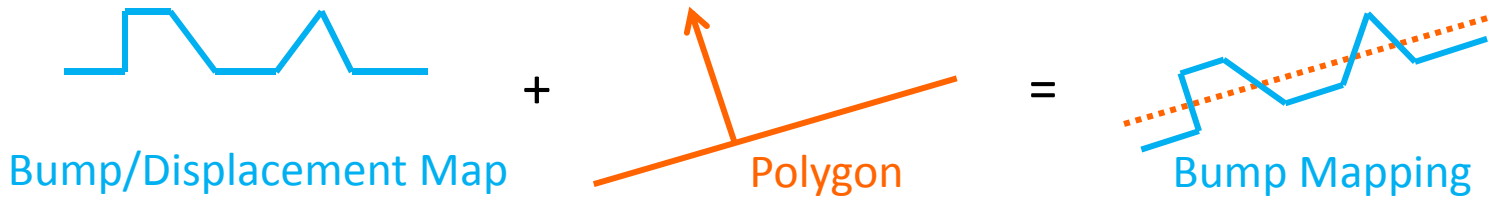


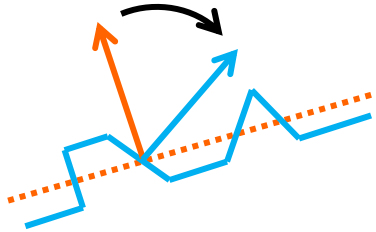
Normal Map

Farbtextur

Bump-Mapping: Prinzip

- Modulation des Normalenvektors berechnet aus der Veränderung einer Basisfläche durch eine Bump Map oder Displacement Map (diese möchte man unabhängig von der Geometrie und Atlas angeben)
- die Fläche bleibt also geometrisch flach und nur die Normalen variieren pro Pixel

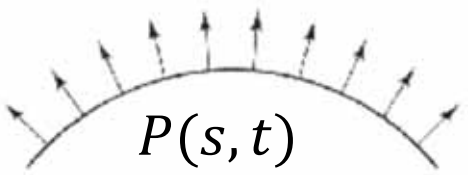




veränderte Normale
für die Beleuchtungsberechnung

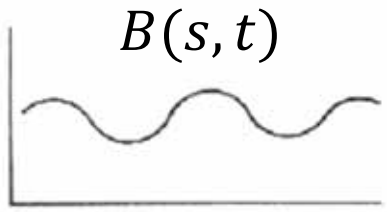
Bump-Mapping: Theorie

- Fläche $P(s,t)$ (z.B. Dreieck oder Bézier-Patch) mit Normale $N(s,t) = \frac{\partial P}{\partial s} \times \frac{\partial P}{\partial t}$



- durch die Bump-Map $B(s,t)$ veränderte Fläche

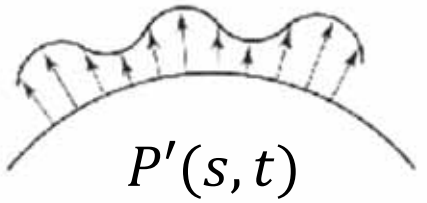
$$P'(s,t) = P(s,t) + B(s,t) \cdot N(s,t)$$

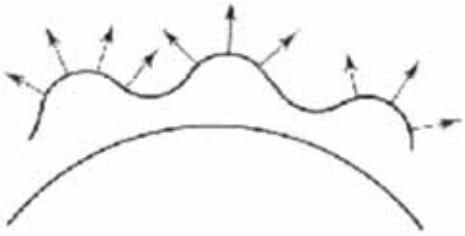


- wie ändert sich die Normale?
  - $N := N(s,t), P := P(s,t), \ldots$

$$N'(s,t) = \frac{\partial P'}{\partial s} \times \frac{\partial P'}{\partial t} = \cdots =$$

$$N + \frac{\partial B}{\partial s}\left(N \times \frac{\partial P}{\partial t}\right) + \frac{\partial B}{\partial t}\left(N \times \frac{\partial P}{\partial s}\right) + \underbrace{\frac{\partial B}{\partial s}\frac{\partial B}{\partial t}(N \times N)}_{=0}$$





- benötigt die Ableitung der Bump-Map und die lokale Tangentialebene der Fläche

Bump-Mapping: Beispiel

- $P(s,t) = s \cdot (1,0,0)^T + t \cdot (0,1,0)^T$

  $N(s,t) = \frac{\partial P}{\partial s} \times \frac{\partial P}{\partial t} = (1,0,0)^T \times (0,1,0)^T = (0,0,1)^T$

- durch die Bump-Map $B(s,t)$ veränderte Fläche

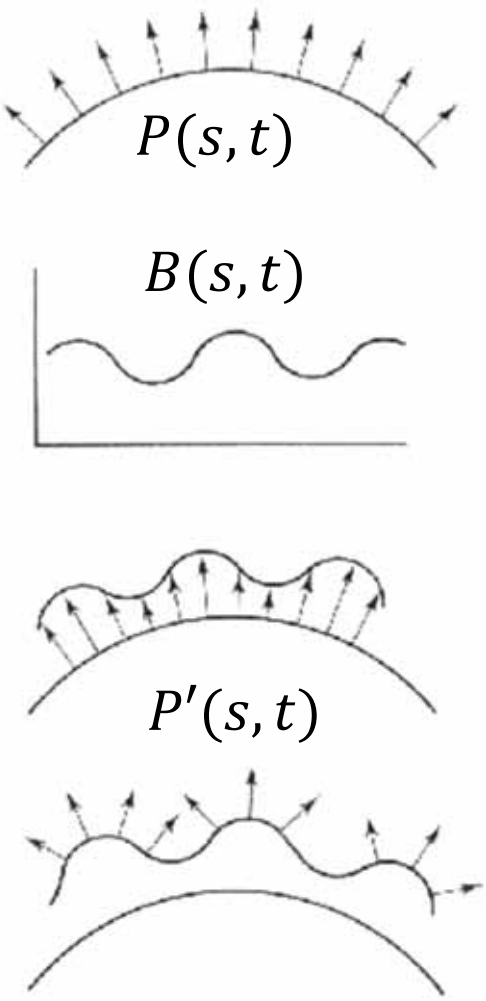
  $P'(s,t) = \big(s,t,B(s,t)\big)$

- wie ändert sich die Normale?
  - $N := N(s,t), P := P(s,t), \ldots$

  $N'(s,t) = \frac{\partial P'}{\partial s} \times \frac{\partial P'}{\partial t} = \cdots =$

  $N + \frac{\partial B}{\partial s}(1,0,0)^T + \frac{\partial B}{\partial t}(0,1,0)^T$

- benötigt die Ableitung der Bump-Map
  und die lokale Tangentialebene der Fläche



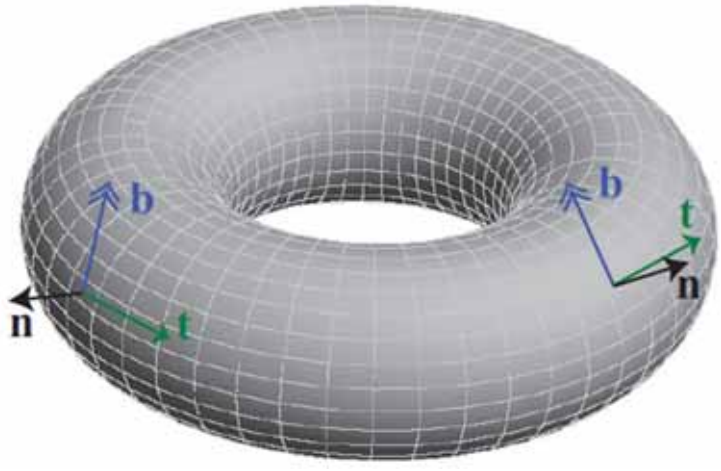
Bump-Mapping und Tangentenraum

- Fläche $P(s,t)$ (z.B. Dreieck oder Bézier-Patch) mit Normale $N(s,t) = \frac{\partial P}{\partial s} \times \frac{\partial P}{\partial t}$ und Bump-Map $B(s,t)$

- Tangentenvektor $T = \left(N \times \frac{\partial P}{\partial t}\right)$ und Bitangente $B = \left(N \times \frac{\partial P}{\partial s}\right)$ liegen tangential zur Oberfläche und spannen zusammen mit der Normale den **Tangentenraum** auf

- nehmen wir zunächst an: $B \perp T$
  - aufgrund der Eigenschaften des Kreuzprodukts ergibt sich dann: $T = \frac{\partial P}{\partial s}$ und $B = \frac{\partial P}{\partial t}$

    (haben wir am Beispiel auf der letzten Folie gesehen)



Bump-Mapping: Theorie

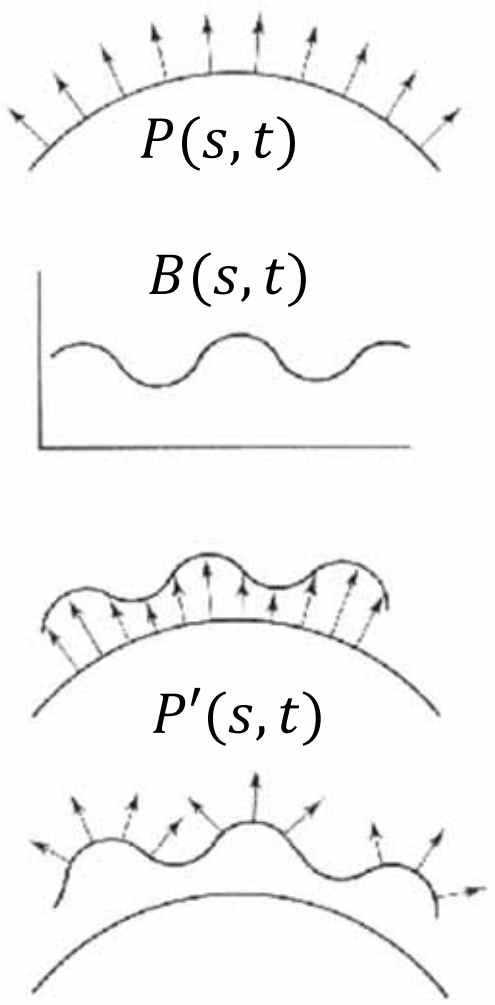
- wie ändert sich die Normale?

$$N'(s,t) = N + \frac{\partial B}{\partial s}\underbrace{\left(N \times \frac{\partial P}{\partial t}\right)}_{= T = \frac{\partial P}{\partial s}} + \frac{\partial B}{\partial t}\underbrace{\left(N \times \frac{\partial P}{\partial s}\right)}_{= B = \frac{\partial P}{\partial t}}$$

  - Ableitung der Bump-Map und lokale Tangentialebene der Fläche

- T ist ausgerichtet entlang des s-Parameters der Fläche *und* der Bump-Map (analog B entlang der t-Koordinate)

- bei Körpern wie Kugel, Torus, … *könnten* Texturkoordinaten und Tangentenraum einfach zusammen bestimmt werden



Bump-Mapping und Tangentenraum

Beobachtungen, übliches Szenario

- bei (Dreiecks-)Netzen mit beliebigen Texturkoordinaten
  - Zugriff auf eine Bump-Map mit Texturkoordinaten s und t
  - Berechnung der modifizierten Normale durch $\frac{\partial B}{\partial s}$ und $\frac{\partial B}{\partial t}$ erfordert einen Tangentenraum der nach s und t ausgerichtet ist!
  - Tangentenraum muss also aus den TexKoord berechnet werden!

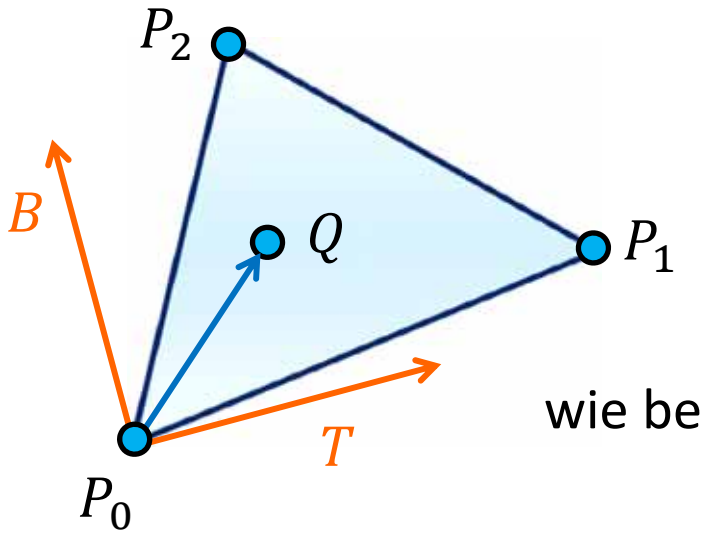
Tangentenraum (Tangent Space)

Tangentenraum für Dreiecksnetze mit Texturkoordinaten

- geg. Dreieck mit Eckpunkten P_i und Texturkoordinaten (s_i, t_i), $i = 0..2$
- wir möchten den Tangentenraum also so ausrichten, dass T (bzw. B) entlang der Ableitung der Texturkoordinate zeigt
- dazu stellen wir einen Punkt Q auf dem Dreieck (bzw. in seiner Ebene) durch die (Textur-)Koordinaten (s, t) dar:

$$Q - P_0 = (s - s_0)T + (t - t_0)B$$

$$P(s,t) = P_0 + (s - s_0)T + (t - t_0)B$$



wie bestimmt man T und B?

Tangentenraum (Tangent Space)

- platzieren wir Q bei P_1 und P_2, dann erhalten wir
  - $Q_1 = P_1 - P_0 = \Delta s_1 T + \Delta t_1 B$ mit $\Delta s_1 = s_1 - s_0,\ \Delta t_1 = t_1 - t_0$
  - $Q_2 = P_2 - P_0 = \Delta s_2 T + \Delta t_2 B$ mit $\Delta s_2 = s_2 - s_0,\ \Delta t_2 = t_2 - t_0$

- daraus erhalten wir ein Gleichungssystem:

$$\begin{bmatrix} Q_{1,x} & Q_{1,y} & Q_{1,z} \\ Q_{2,x} & Q_{2,y} & Q_{2,z} \end{bmatrix} = \begin{bmatrix} \Delta s_1 & \Delta t_1 \\ \Delta s_2 & \Delta t_2 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

  - unbekannt sind T und B, gegeben sind:
    - Vertizes des Dreiecks und somit Q_1, Q_2 und
    - Texturkoordinaten der Vertizes, also Δs_i, Δt_i
  - lösen durch Matrixinversion $\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$

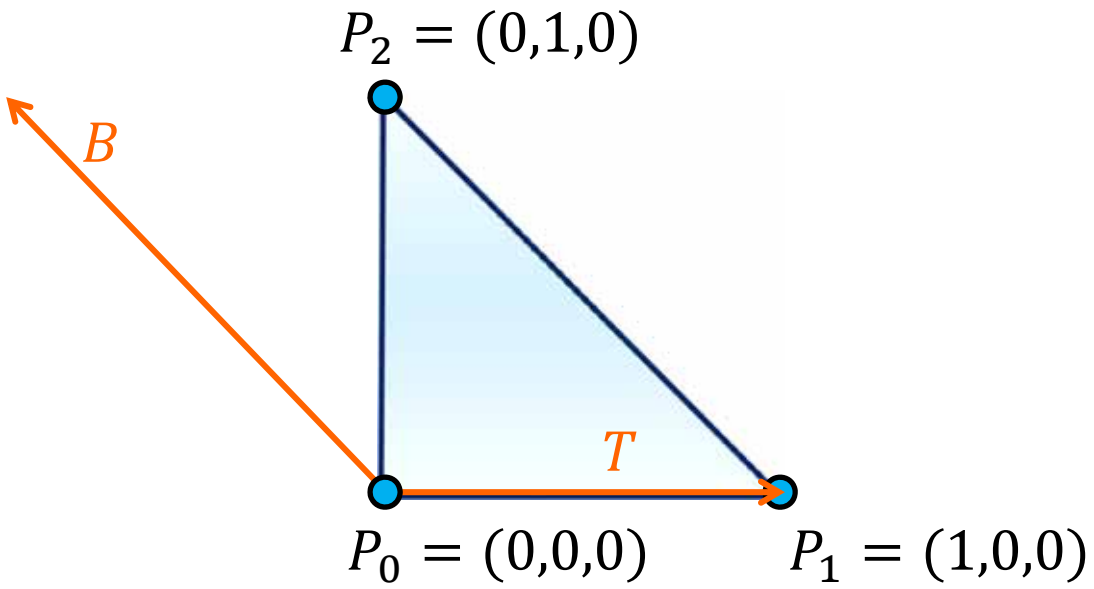
- damit ist die Berechnung von Vektoren T und B – die die Tangentialebene aufspannen – für **ein** Dreieck möglich

- Achtung: diese sind i.A. aber **nicht normalisiert und nicht orthogonal**

Tangentenraum (Tangent Space)

Beispiel

- Texturkoordinaten $(s_0, t_0) = (0,0), (s_1, t_1) = (1,0), (s_2, t_2) = (1,1)$
- $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$ mit $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$
- $\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \end{bmatrix}$



Tangentenraum (Tangent Space)

- die resultierenden Vektoren sind nicht orthonormal (aber $T, B \perp N$)
  - Gram-Schmidtsche Orthonormalisierung: erzeugt zu jedem System lin. unabh. Vektoren ein Orthonormalsys. (hier gezeigt: allg. Vorgehen)
  - $N' = \frac{N}{\|N\|}$
  - $T' = T - (N' \cdot T)N'$ anschließend: normalisiere T'
  - $B' = B - (N' \cdot B)N' - (T' \cdot B)T'$ dann: normalisiere B'
- TBN-Matrix: Transformation von Tangentenraum in Objektraum

$$\mathrm{M} = \begin{bmatrix} T'_x & B'_x & N'_x \\ T'_y & B'_y & N'_y \\ T'_z & B'_z & N'_z \end{bmatrix} \qquad \mathrm{M}^{-1} = \mathrm{M}^{T}$$

(nur nach Orthonormalisierung!)

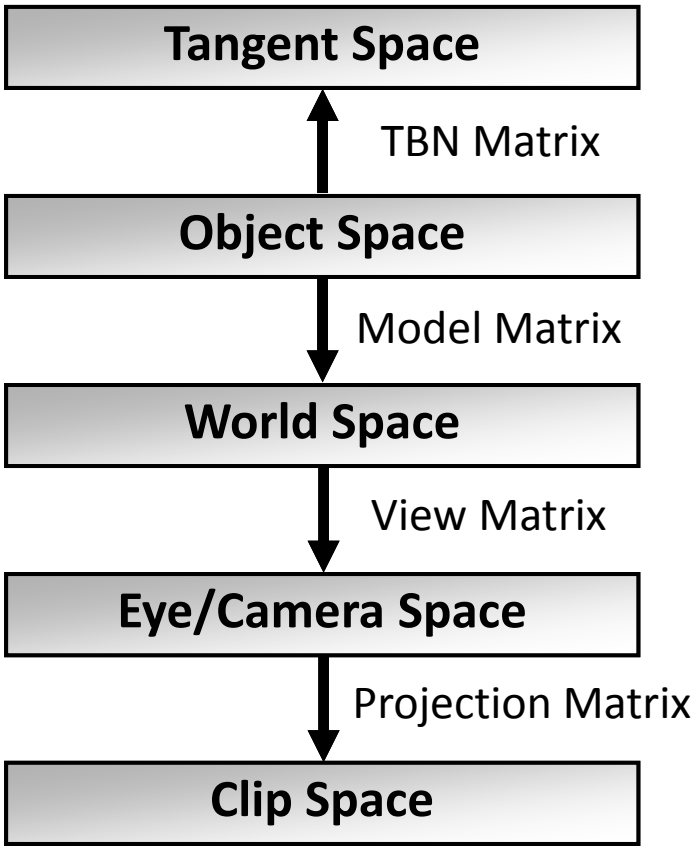
- um einen **Tangentenraum pro Vertex** zu berechnen gehen wir wie bei der Berechnung von Vertex-Normalen vor:
  - bestimme alle Dreiecke, die den Vertex als Eckpunkt haben
  - bilde den Mittelwert aller Dreiecks-Tangentenräume

Beleuchtungsberechnung im Tangentenraum

- veränderte Normale: $N' = N + \frac{\partial B}{\partial s} \cdot T + \frac{\partial B}{\partial t} \cdot B$
- **im Tangentenraum** T, B, N ist die Normale der ursprünglichen Fläche $(0,0,1)^T$, die veränderte Normale entsprechend $N'_{TS} = \left(\frac{\partial B}{\partial s}, \frac{\partial B}{\partial t}, 1\right)^T$
- in der Praxis: zur Beleuchtungsberechnung transformieren wir die Richtung zur Lichtquelle in den Tangentenraum
  - Eingabe:
    - Position der Lichtquelle (in Objekt-/Weltkoordinaten) L_w
    - Position des Oberflächenpunktes in Objekt-/Weltkoordinaten V_w
    - Tangentenraum T, B, N (orthonormalisiert)
  - Transformation:
    - Vektor zur Lichtquelle $L = (L_w - V_w)/|L_w - V_w|$
    - Richtung im Tangentenraum $L_{TS} = (L \cdot T, L \cdot B, L \cdot N)^T$

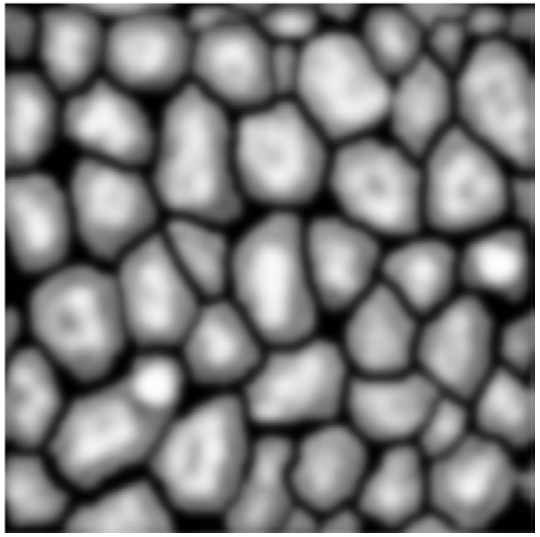
Koordinatensysteme

- die Beleuchtungsberechnung kann in jedem beliebigen Koordinatensystem durchgeführt werden
- unterschiedliche KoSys eignen sich für bestimmte Zwecke besser oder schlechter
  - Objekt-Koordinaten:
    nativer Raum der Vertex-Normalen
  - Weltkoordinaten:
    nativer Raum der Lichtquellen/EnvMaps
  - Aug-/Kamerakoordinaten:
    nativer Raum des View Vectors
  - Tangent Space/Tangenten Raum:
    für Bump/Normal Mapping

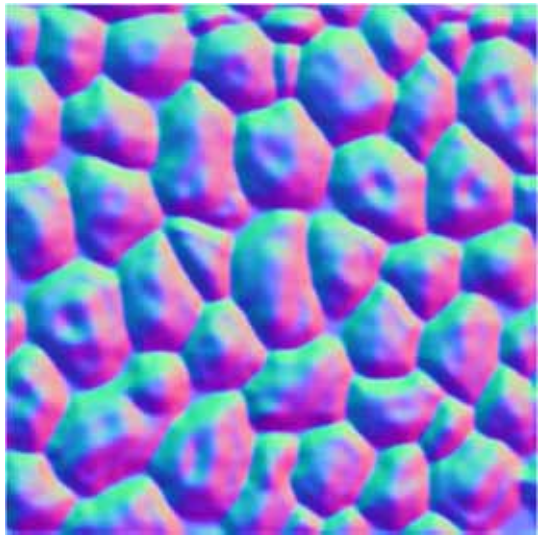


Maps – was?

- Begriffe / begriffliche Überschneidungen
  - **Bump Map**: Normale wird vor der Beleuchtung verändert, dazu wird eine Graustufen-Bitmap bzw. entsprechende Gradienten gespeichert
  - **Normal Map**: Ersetzt die Normale für die Beleuchtungsberechnung, d.h. man speichert $N'_{TS} = \left(\frac{\partial B}{\partial s}, \frac{\partial B}{\partial t}, 1\right)^T$ (normalisiert)
    - im Prinzip kann man auch Normalen in Objekt- oder Weltkoordinaten speichern, verhindert aber Wiederverwendbarkeit
  - **Displacement Mapping**: verändert die Oberfläche auch geometrisch



Höhenkarte $B(s,t)$



Normal Map

Farbtextur

Normal-Maps und Bump-Maps

Farbe

Höhenkarte

Gradient

Normal Map

Normale - x

Normale - y

Normale - z

Normal Maps - Texturformate

- Normal Maps als RGB-Textur gespeichert, indem $N_{TS} = \left(n_x, n_y, n_z\right)^T$ mit $n_x, n_y, n_z \in [-1; 1]$ abgebildet wird auf
  - Farbwerte $R, G, B \in [0; 1]$
    - $R = \frac{n_x}{2} + \frac{1}{2}, G = \frac{n_y}{2} + \frac{1}{2}, B = \cdots$
    - „Auspacken“ vor der Verwendung im Shader mit $n_x = 2R - 1$, …
  - oft genügt eine Genauigkeit von 8 Bit pro Komponente (schwierig v.a. bei glatten glänzenden Oberflächen)
  - spezielle Texturformate die für Normal Maps geeignet sind, z.B.
    - „A2R10G10B10“: 3×10 Bit
    - „R16G16“: n_x, n_y Komponenten mit 16 Bit speichern, $n_z > 0$ wird berechnet mit $n_z = \left(1 - n_x^2 - n_y^2\right)^{1/2}$
- Randnotiz: „On Floating-Point Normal Vectors“, Meyer et al.
  - theoretisch: Diskretisierung in $2^{50.2}$ uniform verteilte Richtungen ebenso „genau“ wie SP-Floating Point Darstellung (Praxis: 52 Bit)

GLSL Normal Mapping

- für das folgende Beispiel brauchen wir einen Tangentenraum
- Hilfsfunktion zur Berechnung eines Tangentenraums
  - das ist ein **Hack**: verwenden Sie ansonsten immer den korrekten Tangentenraum!

```
void  computeTangentSpace( in vec3 N, out mat3 TS ) {
    vec3 NeverCoLinear = vec3( N.y, N.z, -N.x );
    vec3 T = normalize( cross( NeverCoLinear, N ) );
    vec3 B = cross( N, T );
    TS = mat3( T, B, N );
}
```

- … aber gut möglich, dass Sie irgendwann mal irgendein KoSys aufspannen müssen…

GLSL Normal Mapping – Vertex Shader

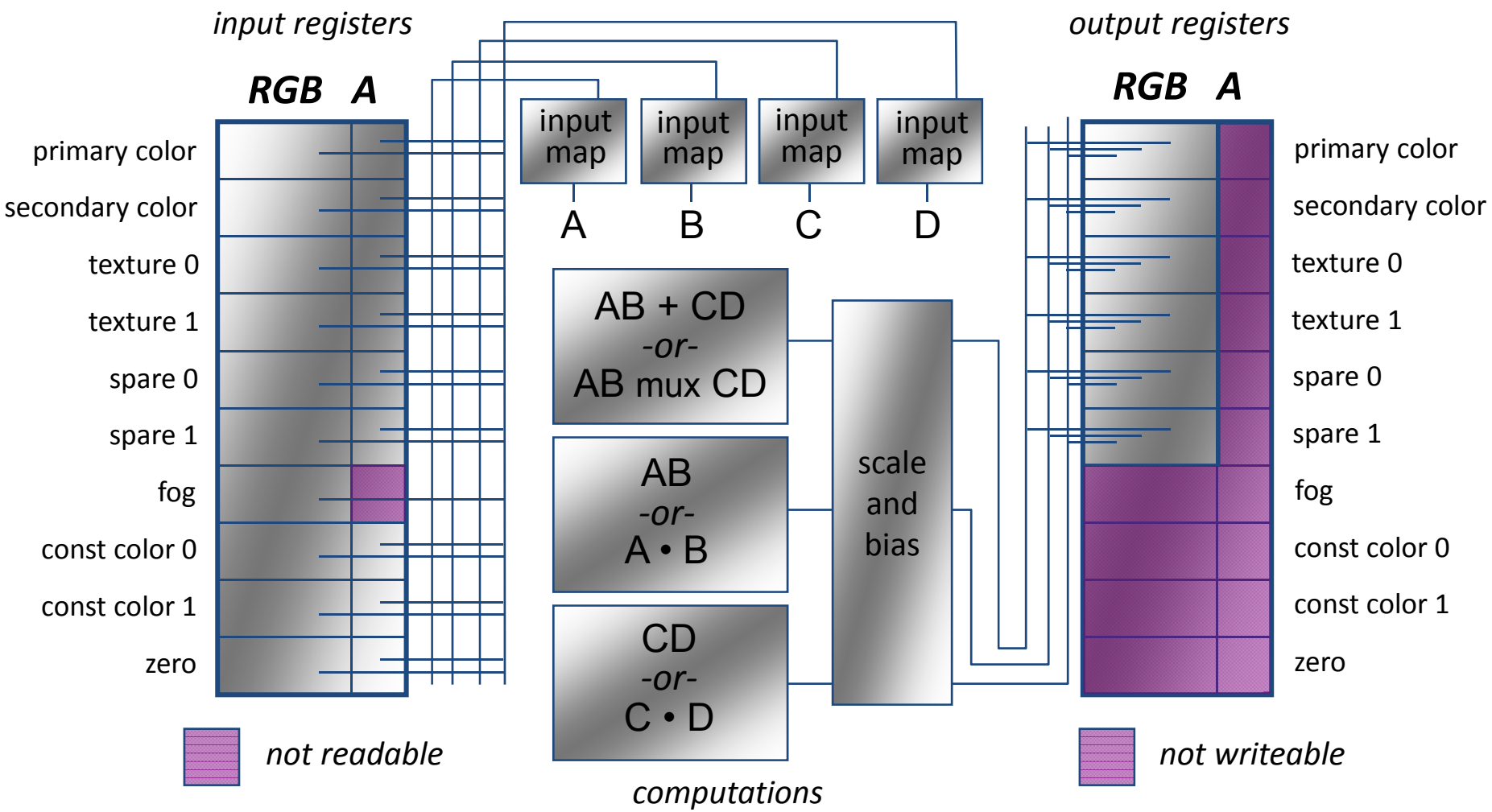
```
uniform mat4     matObject2World;
uniform mat3     matObjectNrml2World;
uniform vec3     vLightPosWS, vCameraPosWS;
in vec4          in_position;
in vec3          in_normal;
out vec3         vLightTS, vViewTS;

void main() {
    …
    vNormal =  ( matObjectNrml2World * vec4( in_normal, 0.0 ) ).xyz;
    vPosition =  ( matObject2World * in_position ).xyz;
    mat3 TBN;
    // Achtung: wir berechnen TBN in Abh. von der Normale in Weltkoord.
    computeTangentSpace( vNormal, TBN ); /* HACK */
    // von Weltkoordinaten in den Tangentenraum:
    vLightTS = normalize( vLightPosWS - vPosition ) * TBN;
    vViewTS = normalize( vCameraPosWS - vPosition ) * TBN;
    …
}
```

GLSL Normal Mapping – Fragment Shader

```
in vec3                 vLightTS, vViewTS;      // siehe letzte Folie
in vec2                 texCoord;               // (nicht gezeigt)
uniform sampler2D       tDiffuse, tBumpmap;     // Sampler
uniform vec4            lightIntensity;         // LQ Attribute
out vec4                out_color;
void main() { …
    texColor = texture2D( tDiffuse, texCoord );
    normalTS = texture2D( tBumpmap, texCoord ) * 2.0 - vec4(1.0);

    // Blinn-Phong Modell
    float diffuse  = max( 0.0, dot( normalTS, vLightTS ) );
    vec3 vHalfwayTS = normalize( vLightTS + vViewTS );
    float specular = pow( max( 0.0, dot( vHalfwayTS, normalTS ) ), 24.0 );

    // Berechnung/Multitexturing nach belieben
    vec4 color = vec4( diffuse  ) * texColor;
    color += vec4( specular ) * glossValue;
    out_color = color * lightIntensity;
}
```

NVidia Register Combiner

- **nur zum Spaß!**



Bump Mapping mit Register Combiner

```
// GENERAL Combiner #0, RGB
// Argb = 3x3 matrix column1 = expand(texture0rgb) = N'
glCombinerInputNV(
  GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_A_NV,
  GL_TEXTURE0_ARB, GL_EXPAND_NORMAL_NV, GL_RGB );
// Brgb = expand(texture1rgb) = L (oder H, falls specular)
glCombinerInputNV(
  GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_B_NV,
  GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV, GL_RGB );

// spare0rgb = Argb * Brgb = expand( texture0rgb ) * expand( texture1rgb ) = L * N'
// im specular fall: spare0rgb = H dot N'
glCombinerOutputNV(
  GL_COMBINER0_NV, GL_RGB,
  GL_SPARE0_NV, GL_DISCARD_NV, GL_DISCARD_NV,
  GL_NONE, GL_NONE, GL_TRUE, GL_FALSE, GL_FALSE );

if ( shadowing == 1 ) {
// GENERAL Combiner #1, RGB
// Argb = 0
glCombinerInputNV(
  GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_A_NV,
  GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB );
// Brgb = 0
glCombinerInputNV(
  GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_B_NV,
  GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB );

if ( specular )
  // Crgb = spare0rgb = H * N'
  glCombinerInputNV(
    GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_C_NV,
    GL_SPARE0_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB ); else
  // Crgb = 1
  glCombinerInputNV(
    GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_C_NV,
    GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_RGB );

// Drgb = spare0rgb = L * N' (oder H * N', falls specular)
glCombinerInputNV(
  GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_D_NV,
  GL_SPARE0_NV, GL_SIGNED_IDENTITY_NV, GL_RGB );

// spare0rgb = ((spare0a >= 0.5) ? spare0rgb^2 : 0) = ((L * N > 0) ? (L * N')^2 : 0)
glCombinerOutputNV(
  GL_COMBINER1_NV, GL_RGB,
  GL_DISCARD_NV, GL_DISCARD_NV, GL_SPARE0_NV,
  GL_NONE, GL_NONE, GL_FALSE, GL_FALSE, GL_TRUE ); }
```

```
// FINAL Combiner
// A = EF
glFinalCombinerInputNV(
  GL_VARIABLE_A_NV,
  GL_E_TIMES_F_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB );
// B = EF
glFinalCombinerInputNV(
  GL_VARIABLE_B_NV,
  GL_E_TIMES_F_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB );
// C = 0
glFinalCombinerInputNV(
  GL_VARIABLE_C_NV,
  GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB );

if ( specular )
  // D = zero = keine extra specular beleuchtung
  glFinalCombinerInputNV(GL_VARIABLE_D_NV,
    GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB ); else
  // D = C0 = ambiente beleuchtung
  glFinalCombinerInputNV(
    GL_VARIABLE_D_NV,
    GL_CONSTANT_COLOR0_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB );

if ( specular )
  // E = spare0rgb = diffuse beleuchtung = H * N'
  glFinalCombinerInputNV(
    GL_VARIABLE_E_NV,
    GL_SPARE0_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB ); else
  // E = 1
  glFinalCombinerInputNV(
    GL_VARIABLE_E_NV,
    GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_RGB );

// F = spare0rgb = diffuse beleuchtung = L * N' (oder H * N' bei specular)
glFinalCombinerInputNV(
  GL_VARIABLE_F_NV,
  GL_SPARE0_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB );

// diffuse RGB color  = A * E * F + D = diffuse modulated mit self-shadowing term +
ambient
// specular RGB color = A * E * F     = specular modulated mit self-shadowing term

// G = spare0a = beitrag der diffusen beleuchtung = L * N'
glFinalCombinerInputNV(
  GL_VARIABLE_G_NV,
  GL_SPARE0_NV, GL_UNSIGNED_IDENTITY_NV, GL_ALPHA );

glEnable( GL_REGISTER_COMBINERS_NV );
```

Bump Mapping mit ARB Vertex/Fragment Program

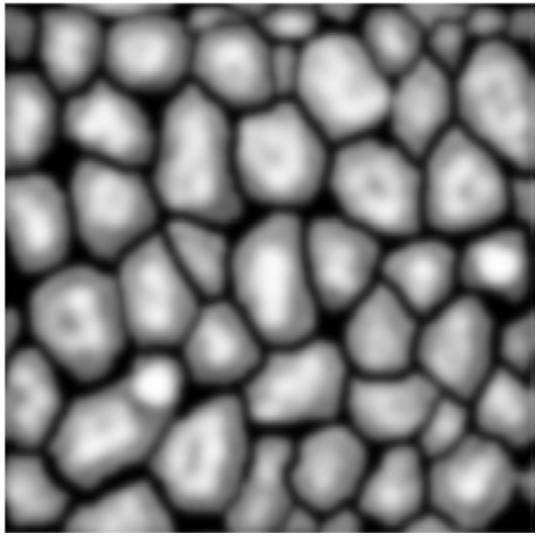
```
!!ARBvp1.0

OPTION ARB_position_invariant;

# light position in object space
PARAM lightPosition    = program.env[1];
# camera position in object space
PARAM cameraPosition   = program.env[2];

# INPUTS
# TEX0           Texture Coordinates
# TEX1           Binormale
# TEX2           Tangente
# NRML           Normale
# POS            Position

ATTRIB           binormal = vertex.texcoord[ 1 ];
ATTRIB           tangent  = vertex.texcoord[ 2 ];
ATTRIB           normal   = vertex.normal;

# OUTPUTS
# TEX0           texture coordinate
# TEX1           light position in tangent space
# TEX2           camera position in tangent space
# TEX3           world space coordinates

OUTPUT tangentLight    = result.texcoord[1];
OUTPUT tangentView     = result.texcoord[2];

TEMP   toLight, toViewer;

MOV result.texcoord[0], vertex.texcoord[0];

MOV result.texcoord[3], vertex.position;

ADD toLight,  lightPosition,  -vertex.position;
ADD toViewer, cameraPosition, -vertex.position;

DP3 tangentLight.x, binormal, toLight;
DP3 tangentLight.y, tangent,  toLight;
DP3 tangentLight.z, normal,   toLight;

DP3 tangentView.x, binormal, toViewer;
DP3 tangentView.y, tangent,  toViewer;
DP3 tangentView.z, normal,   toViewer;

END
```

```
!!ARBfp1.0

OUTPUT color = result.color;

# CONSTANTS
TEMP surfaceColor, bumpNormal, glossFactor, ...;

# CONSTANTS
PARAM constantOne              = {  1.0,  1.0,  1.0,  1.0 };
PARAM constantDouble           = {  2.0,  2.0,  2.0,  2.0 };
PARAM constantAttenuation      = {  1.0,  0.0,  0.1,  0.0 };

# INPUTS
PARAM lightColorAmbient  = program.env[ 0 ]; ...

ATTRIB lightVectorIn   = fragment.texcoord[ 1 ];
ATTRIB viewVectorIn    = fragment.texcoord[ 2 ];
ATTRIB worldSpaceCoord = fragment.texcoord[ 3 ];

# get surface color, normal and gloss factor
TEX surfaceColor, fragment.texcoord[0], texture[0], 2D;
TEX bumpNormal,   fragment.texcoord[0], texture[1], 2D;
TEX glossFactor,  fragment.texcoord[0], texture[2], 2D;

# scale & bias bumpnormal: [0,1] -> [-1,1]
MAD bumpNormal, bumpNormal, constantDouble, -constantOne;

# normalization of the bumped normal
DP3 temp,   bumpNormal, bumpNormal;
RSQ invLen, temp.x;
MUL bumpNormal,   bumpNormal, invLen;

# normalize view vector
DP3 temp,   viewVectorIn, viewVectorIn;
RSQ invLen, temp.x;
MUL viewVector,  viewVectorIn, invLen;

# normalize light vector
DP3 distanceToLight,   lightVectorIn, lightVectorIn;
RSQ invLen, distanceToLight.x;
MUL lightVector,  lightVectorIn, invLen;

# calculate the reflection vector
DP3 temp,  bumpNormal, lightVector;
MUL temp,  temp,  bumpNormal;
MAD reflectionVector, temp, constantDouble, -lightVector;
```

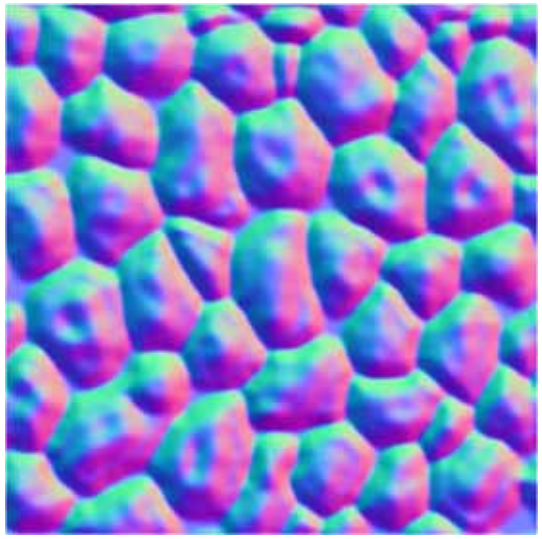
```
# attenuation
MAD attenuation, distanceToLight.z,
    constantAttenuation.z, constantAttenuation.x;
RCP attenuation, attenuation.x;

# N dot L, R dot V, (R dot V)^n, glossFactor*...
DP3_SAT diffuseLight,  bumpNormal, lightVector;
DP3_SAT specularLight, reflectionVector, viewVector;
POW specularLight, specularLight.x, lightSpecExponent.x;
MUL specularLight, glossFactor, specularLight;

# light source colors
MUL diffuseLight,  diffuseLight,  lightColorDiffuse;
MUL specularLight, specularLight, lightColorSpecular;

MAD temp,  diffuseLight,  attenuation, lightColorAmbient;
MUL temp2, specularLight, attenuation;
MAD color, surfaceColor, temp, temp2;

END
```

Normal Maps Erzeugen

- aus Farb-/Höhenbildern (manuell, prozedural, …)



Höhenkarte $B(u, v)$



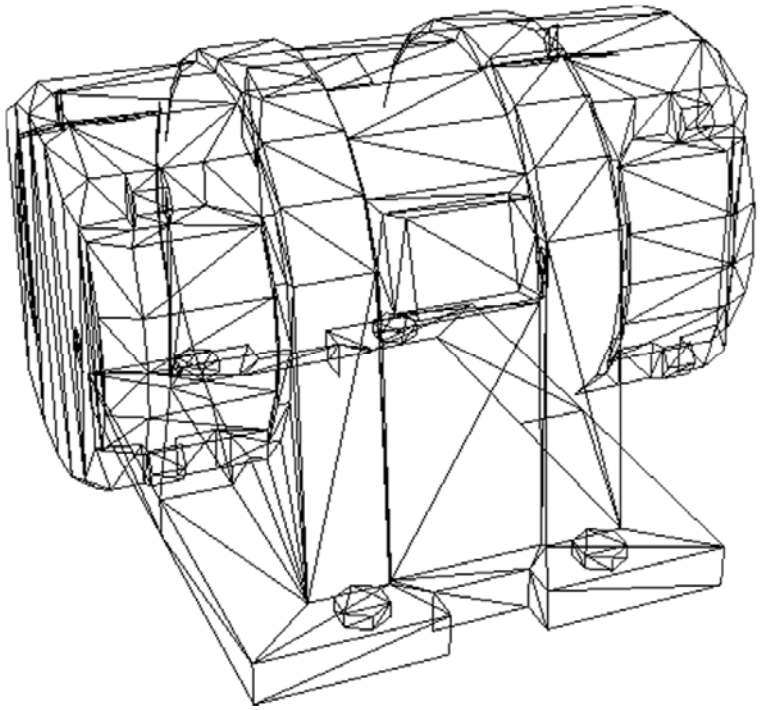
Normal Map

Farbtextur

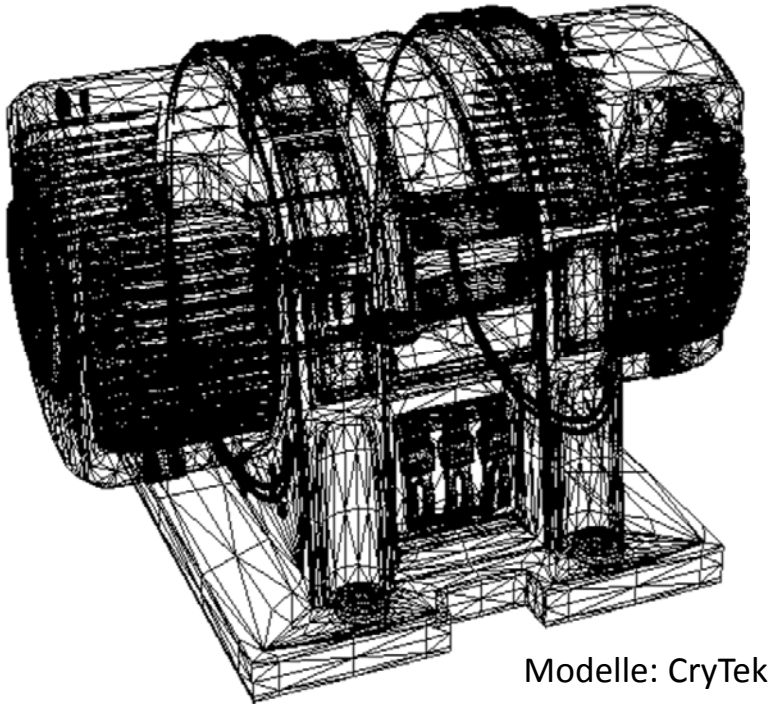
- aus feiner Geometrie…

Normal Maps Erzeugen

- Idee: fehlendes Detail im groben Netz wird in Form von Normalen in einer Textur gespeichert



848 Dreiecke

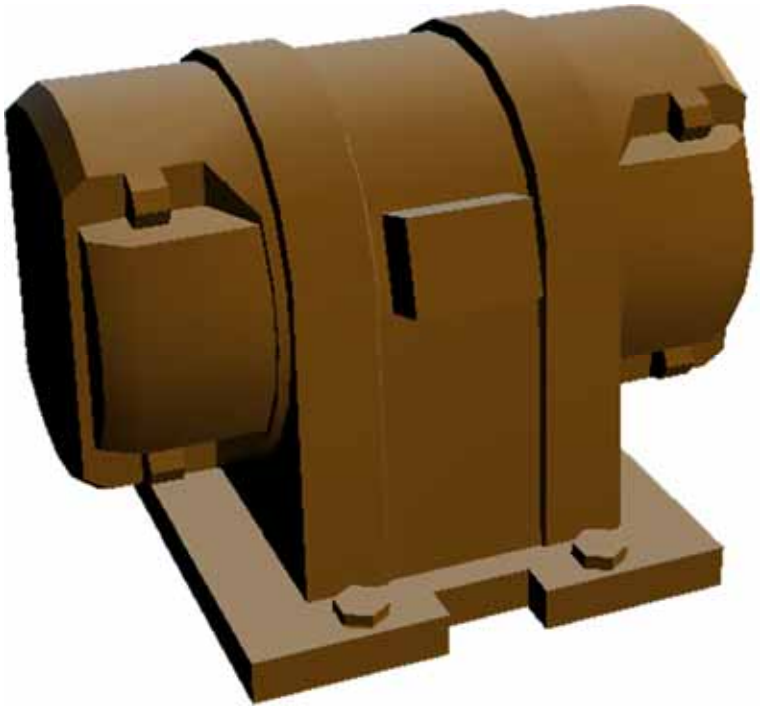


Modelle: CryTek

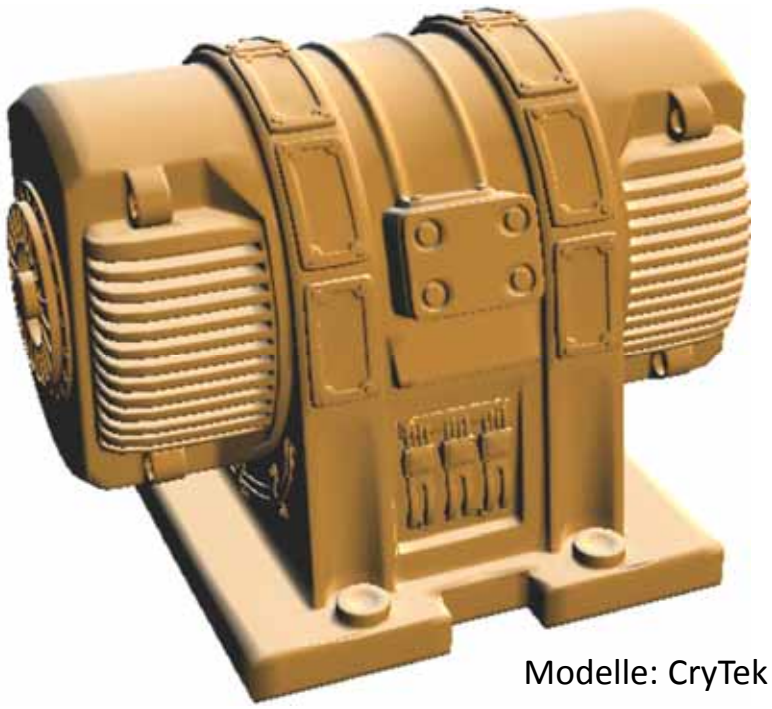
~115.000 Dreiecke

Normal Maps Erzeugen

- Idee: fehlendes Detail im groben Netz wird in Form von Normalen in einer Textur gespeichert



848 Dreiecke

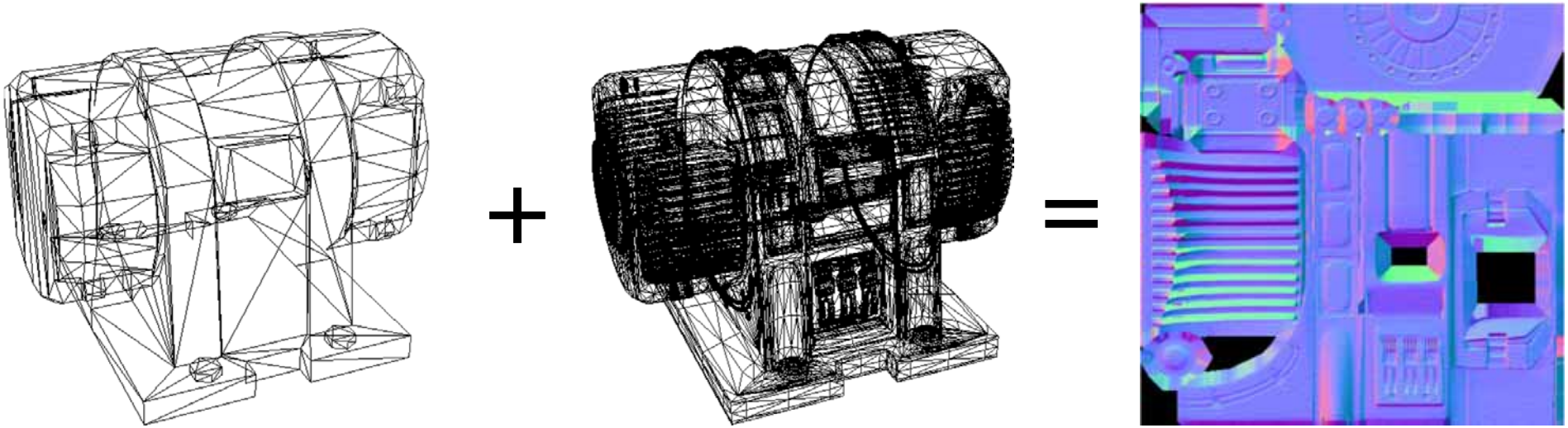


Modelle: CryTek

~115.000 Dreiecke

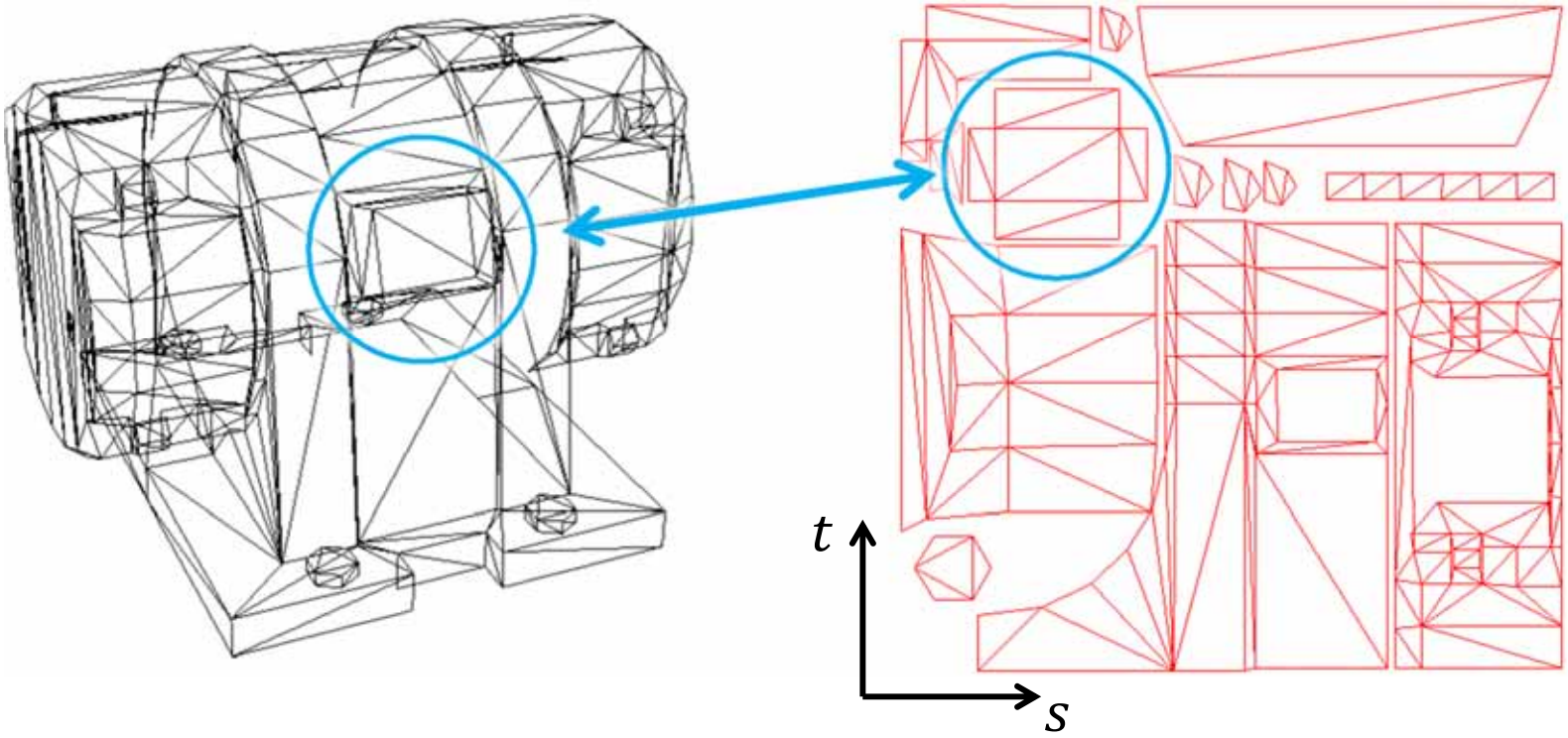
Normal Maps Erzeugen

- Idee: fehlendes Detail im groben Netz wird in Form von Normalen in einer Textur gespeichert
- wir besprechen kurz die Funktionsweise im Prinzip
- es gibt diverse Tools, die diese Berechnung übernehmen
  - xNormal (www.xnormal.net), ATI Normalmapper, NVIDIA Melody, Crytek Polybump, …



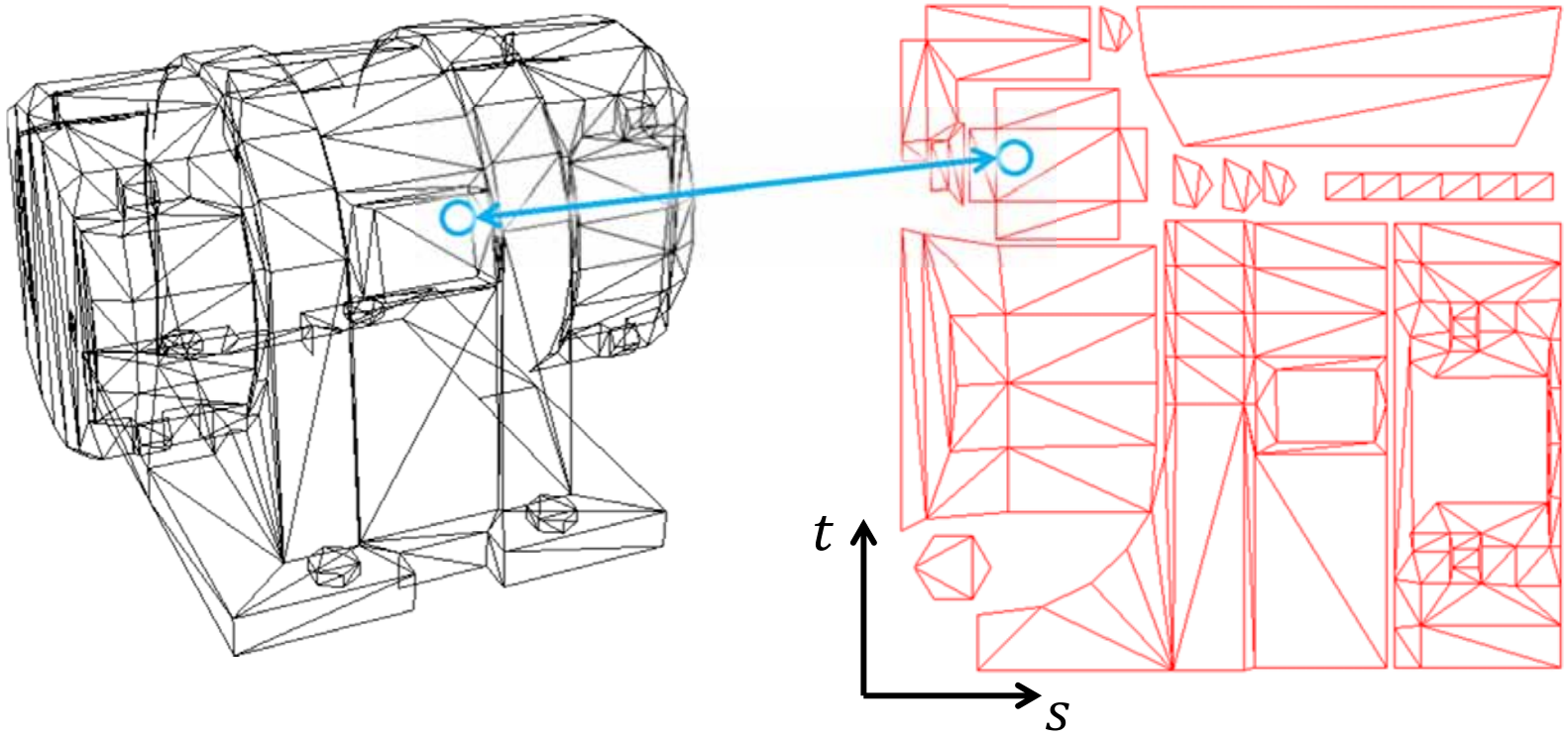
Normal Maps Erzeugen

- Eingabedaten
  - feines Netz: Eckpunkte und Normalen
  - grobes Netz: zusätzlich eine 2D Parametrisierung (Textur-Atlas)



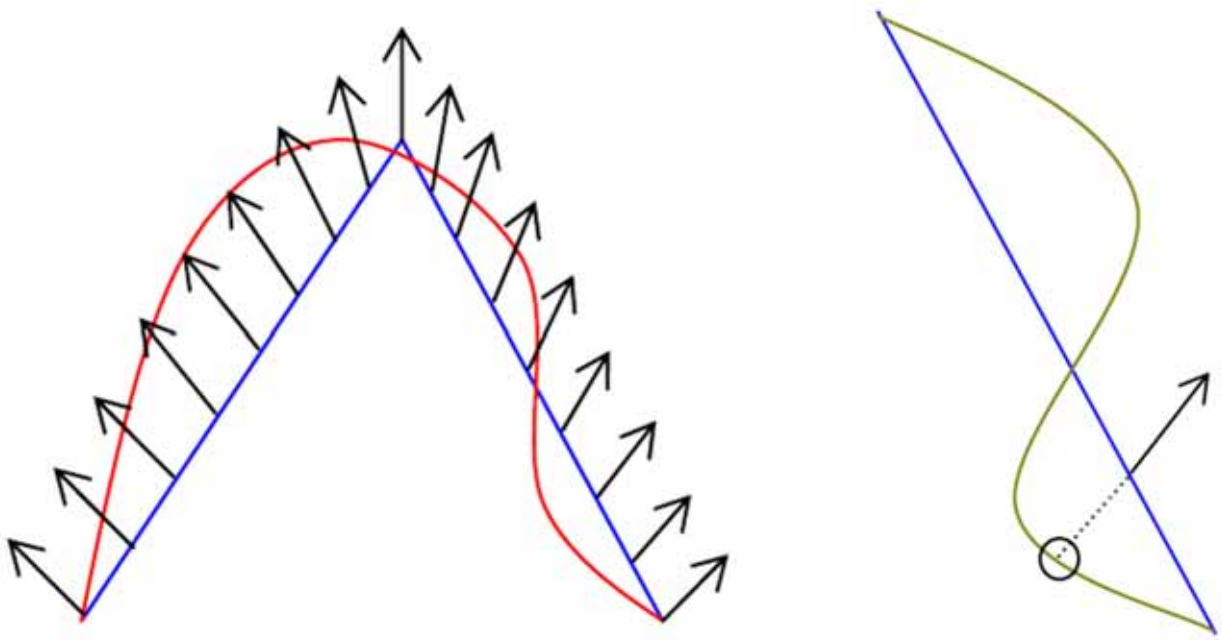
Normal Maps Erzeugen

- für jeden Texel (s, t) in der Bumpmap
  - bestimme Dreieck zu dem er gehört (anhand der Texturkoordinaten)
  - bestimme Koordinate und Normale (auf dem niedrig aufgelösten Dreiecksnetz)



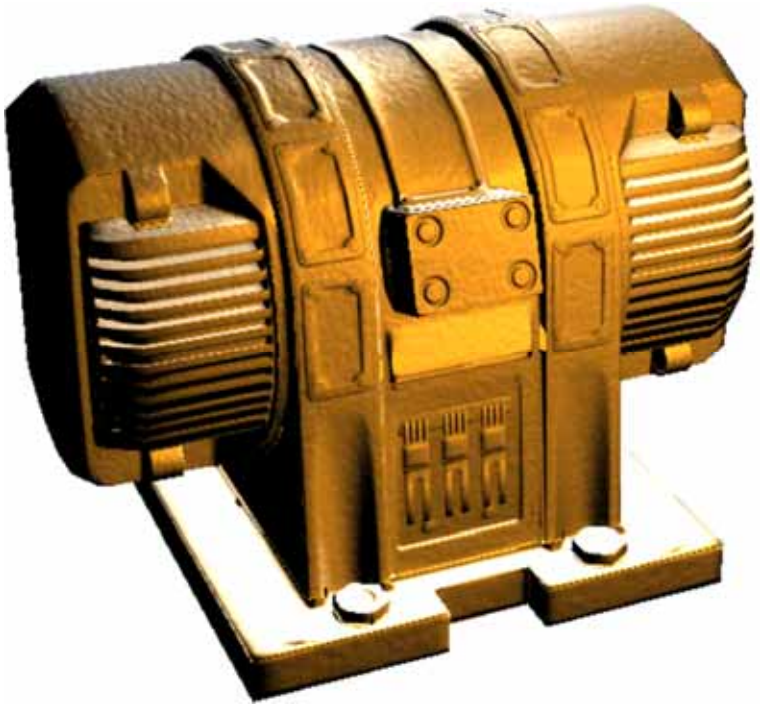
Normal Maps Erzeugen

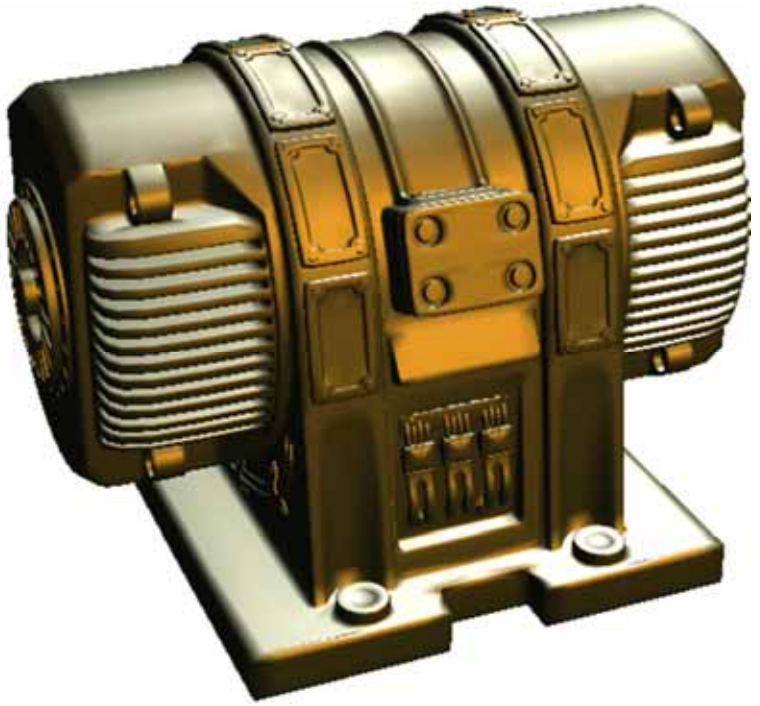
- für jeden Texel (s, t) in der Bumpmap mit Koordinate und Normale (des Punktes auf dem niedrig aufgelösten Dreiecksnetz):
  - bestimme nahsten Schnittpunkt mit dem feinen Netz entlang der Normale (kann auch in negativer Richtung liegen)
  - speichere Normale des feinen Netzes am Schnittpunkt im Textur-Atlas
  - (analog könnte man auch die Entfernung speichern und somit Displacement Maps erzeugen)



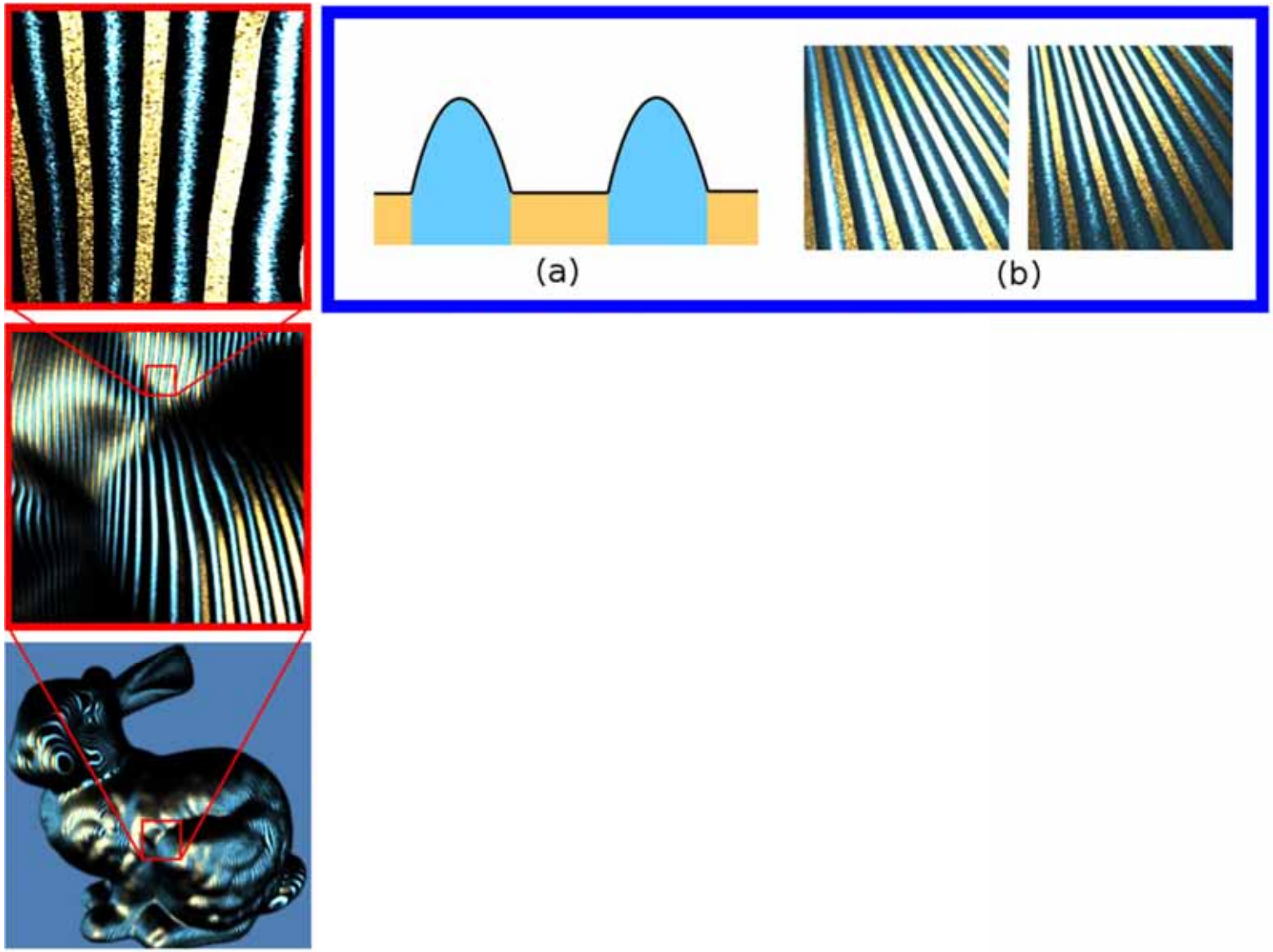
Normal Maps

- Resultat (hier allerdings mit unterschiedlicher Beleuchtung)





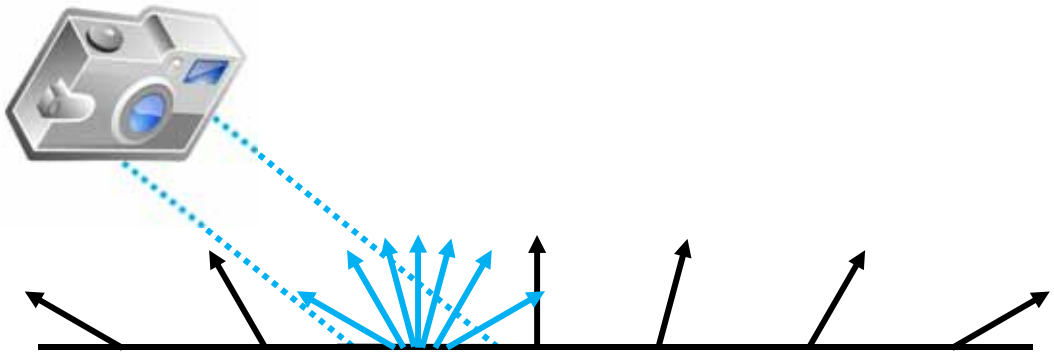
Normal Maps und Texturfilterung

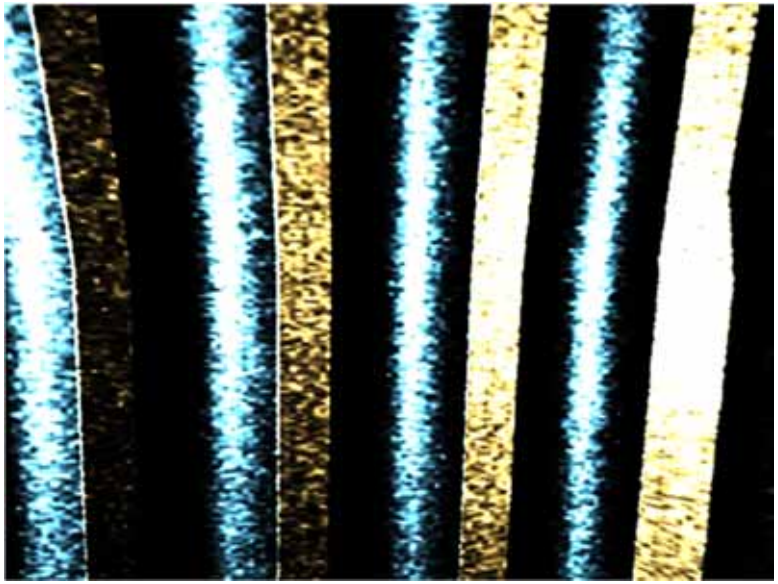


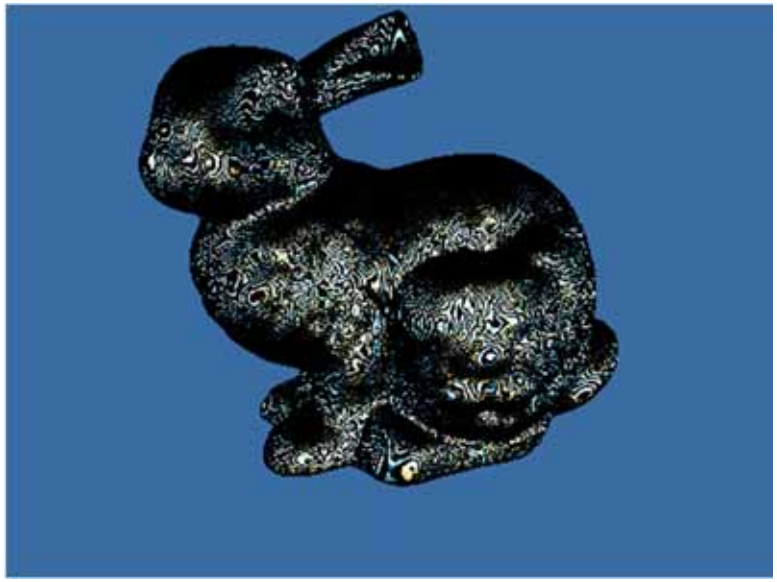
Supersampling
(„ground-truth“)

Normal Maps und Texturefiltering

- Problem bei Verkleinerung (mehrere Texel – also Normalen – pro Pixel)
  - hier: Nearest-Neighbor-Sampling
  - wie berechnet man Mip-Maps von Normal Maps?

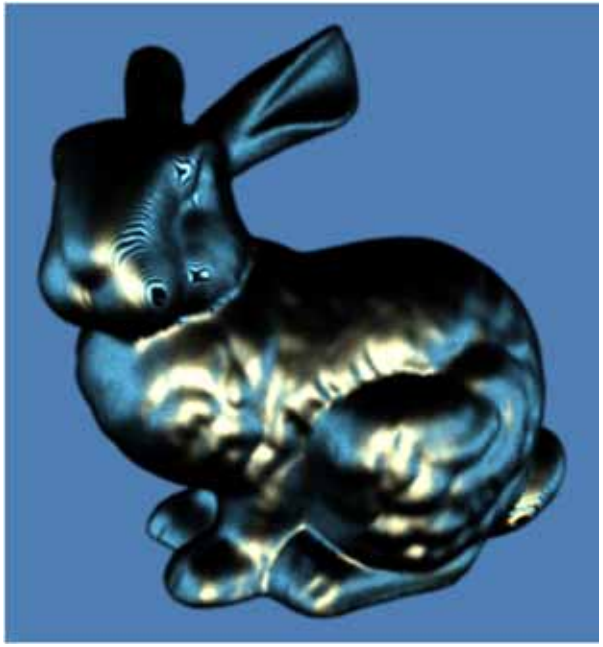






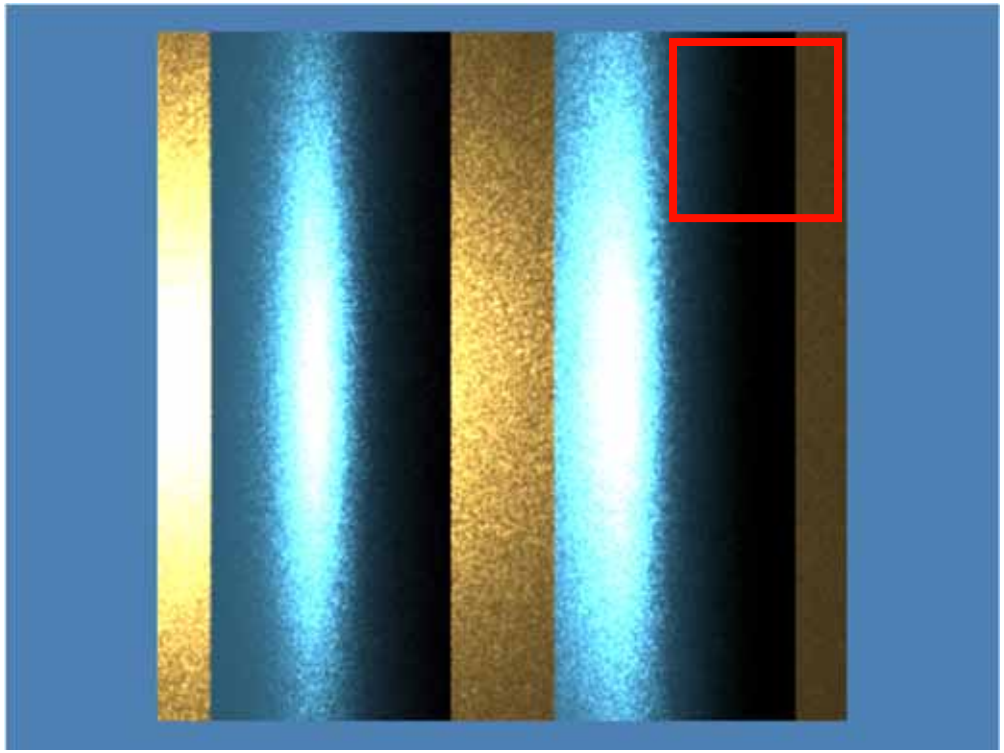
Normal Maps und Texturefiltering

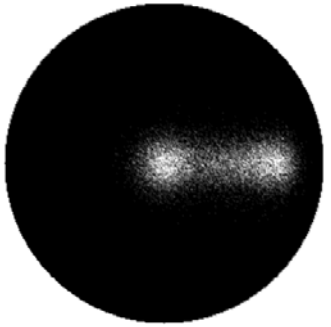
- Supersampling (links): Beleuchtungsberechnung mit vielen Normalen, anschließend Mittelung der Farbe → korrekt!
- Mip-Mapping (rechts): zuerst Mittelung der Normale, anschließend einmalig Beleuchtungsberechnung → Änderung der Materialerscheinung
  - durch Mittelung der Normalen wird die Fläche glatter (Tendenz zur makroskopischen Oberflächennormale)



Normal Distribution Function

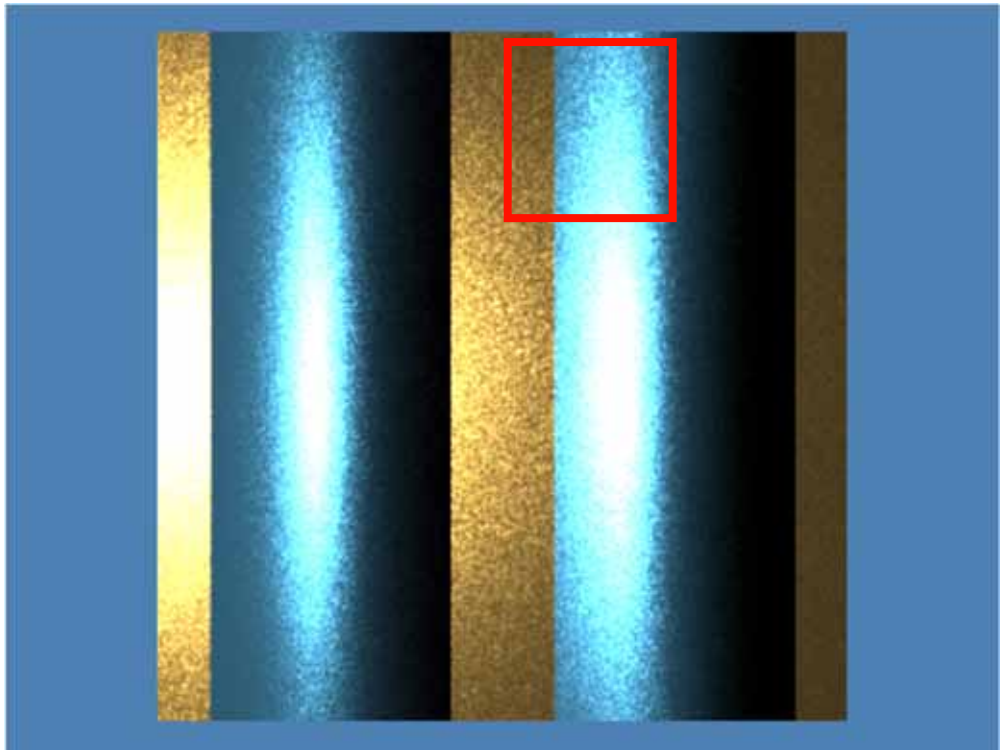
- Idee: speichere eine Verteilung der Normalenrichtung pro Texel
  - benötigt mehr Speicher, Verteilungen können aber „gemittelt“ werden
    → konstanter Speicherbedarf pro Texel

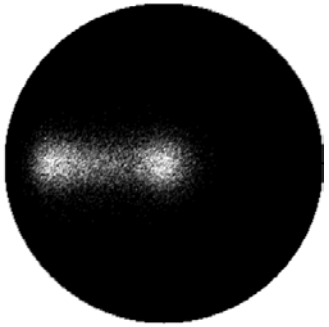




Normal Distribution Function

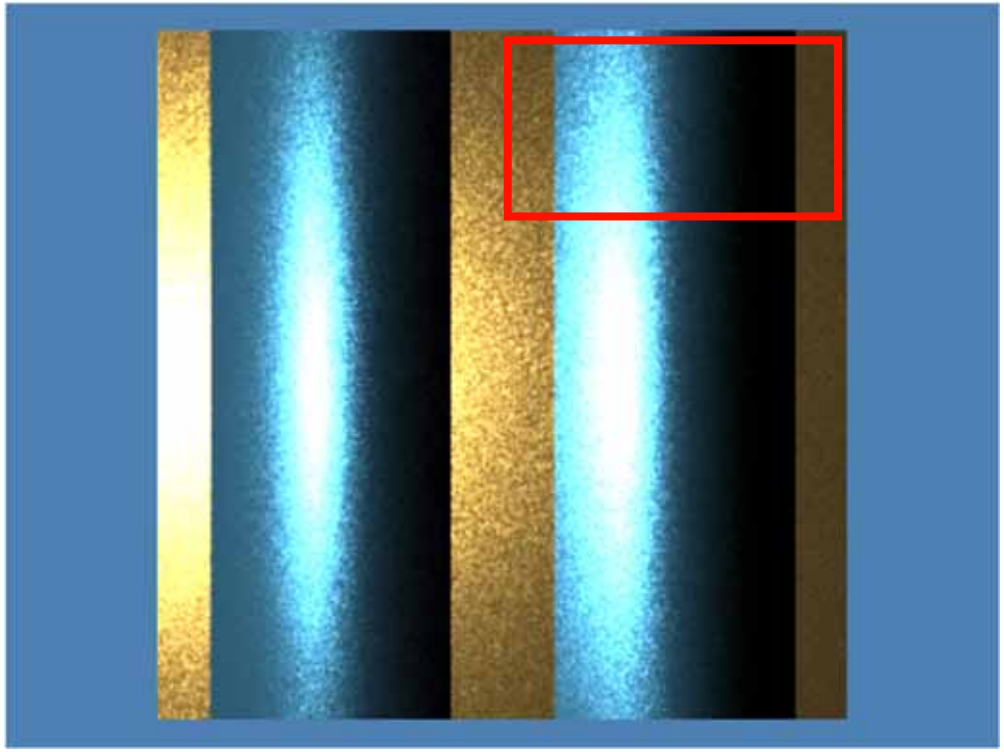
- Idee: speichere eine Verteilung der Normalenrichtung pro Texel
  - benötigt mehr Speicher, Verteilungen können aber „gemittelt“ werden
    → konstanter Speicherbedarf pro Texel

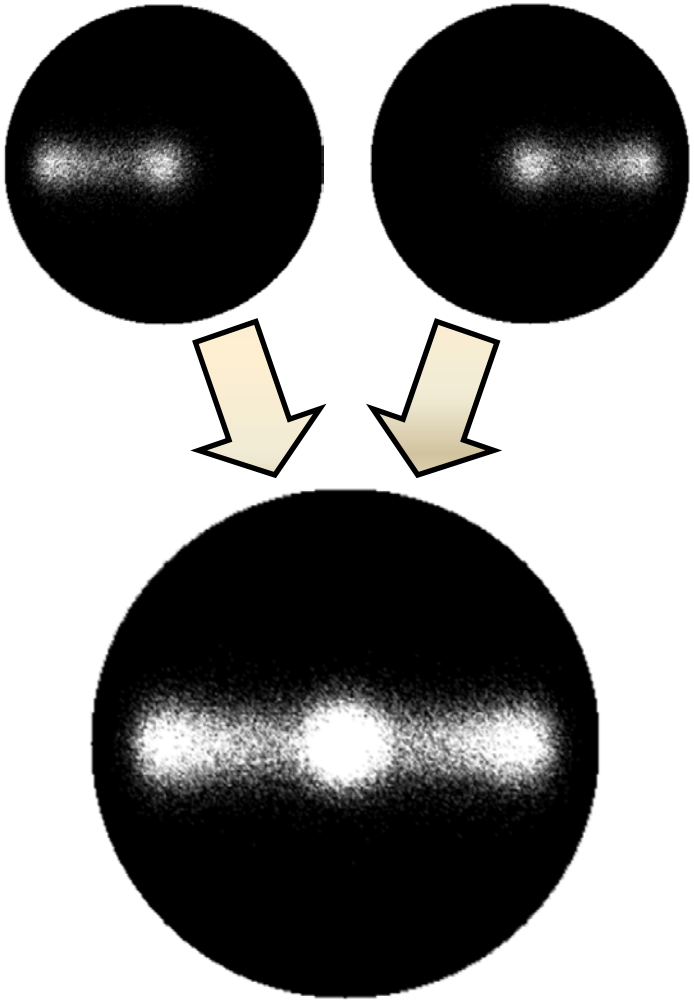




Normal Distribution Function

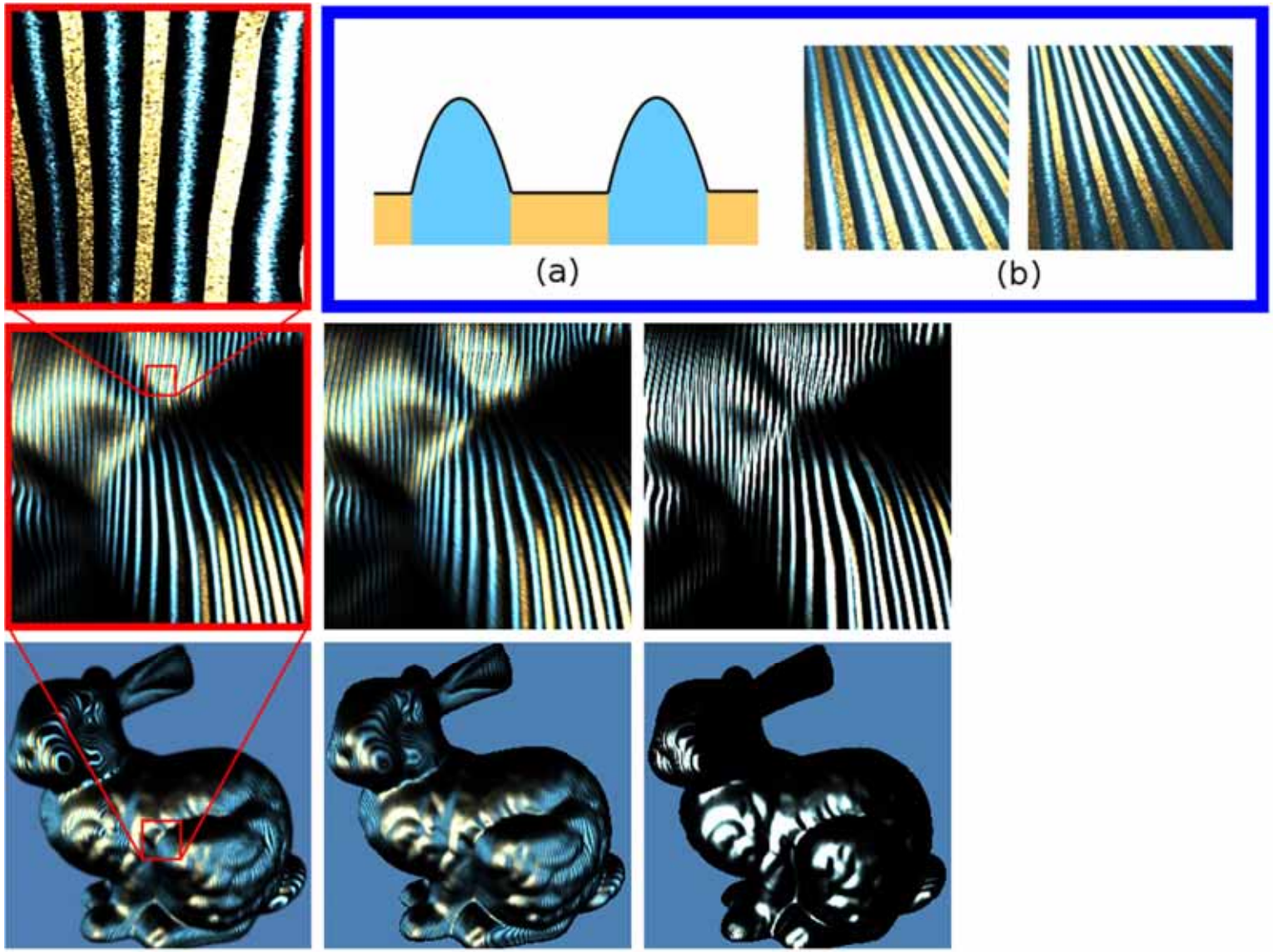
- Idee: speichere eine Verteilung der Normalenrichtung pro Texel
  - benötigt mehr Speicher, Verteilungen können aber „gemittelt“ werden → konstanter Speicherbedarf pro Texel





Frequency Domain Normal Map Filtering
Han et al., SIGGRAPH 2007

Normal Maps und Texturefiltering



Supersampling („ground-truth“)

Han et al. Frequency Domain Normal Map Filtering

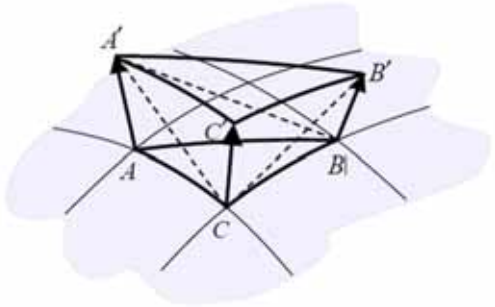
herkömmliches Mip-Mapping

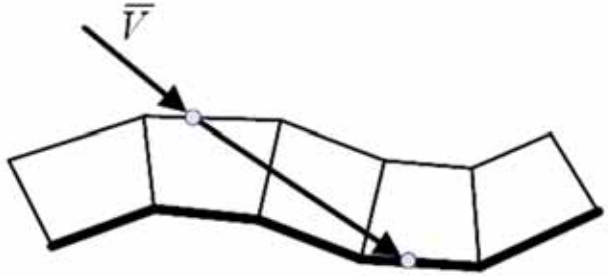
Texture Space Lighting

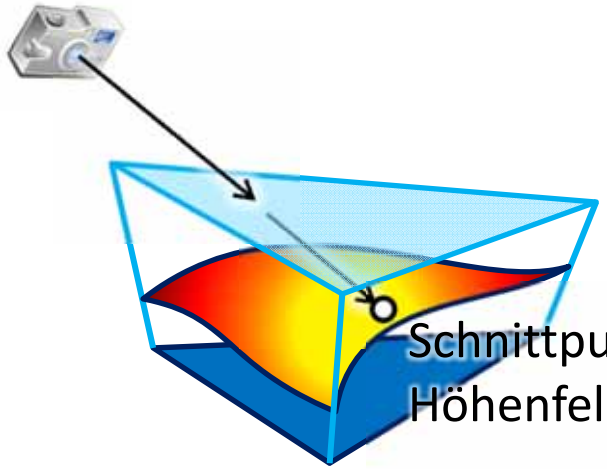
- alternative Lösung: Berechnung der Beleuchtung im Texturraum
  - anschließend Zeichnen des Objekts mit der resultierenden Textur (inkl. der daraus generierten Mip-Maps)
- Nachteil: Beleuchtungsberechnung unabhängig von der Größe und Verdeckung eines Objekts am Bildschirm
- Einsatz oft in Verbindung mit speziellen Beleuchtungstechniken, die Nachbarschaftsinformation benötigen (z.B. Sub-Surface Scattering, rechte Bilder, ATI Demo)

Displacement Mapping

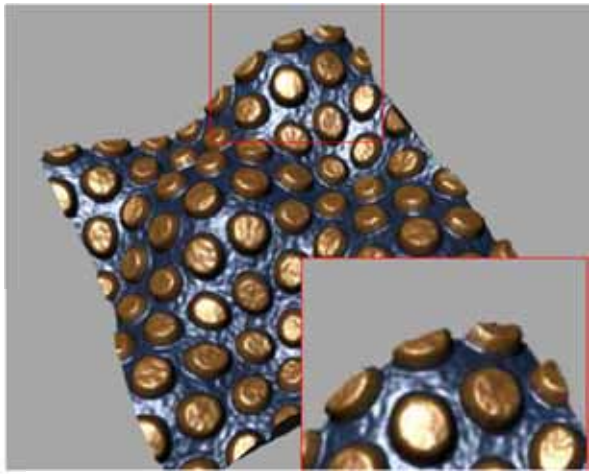
- modifiziere die Oberfläche gemäß einer Höhenkarte auch geometrisch
  - entweder: durch Verschieben von Vertizes
    (hierzu gibt es Tessellation-Shader, siehe späteres Kapitel)
  - oder Aufbringen einer 3D Textur/Höhenfeldes und Strahlverfolgung
    (Inverse Displacement Mapping):







Schnittpunkt mit Höhenfeld



Inverse Displacement Mapping

- Displacement Mapping kann **mit Einschränkungen** durch eine reine Texturierungstechnik erreicht werden
- Idee der einfachen Verfahren: betrachte eine ebene Fläche
  - die Fläche wird „nach Innen“ gezogen
  - der Schnittpunkt des Augstrahls mit der Fläche wird durch **Ray Marching** im Fragment Shader gesucht
  - Achtung – die Annahmen in diesem Verfahren sind:
    1) ebene Fläche und 2) es existiert immer ein Schnittpunkt
    (die beiden Aspekte hängen natürlich zusammen)

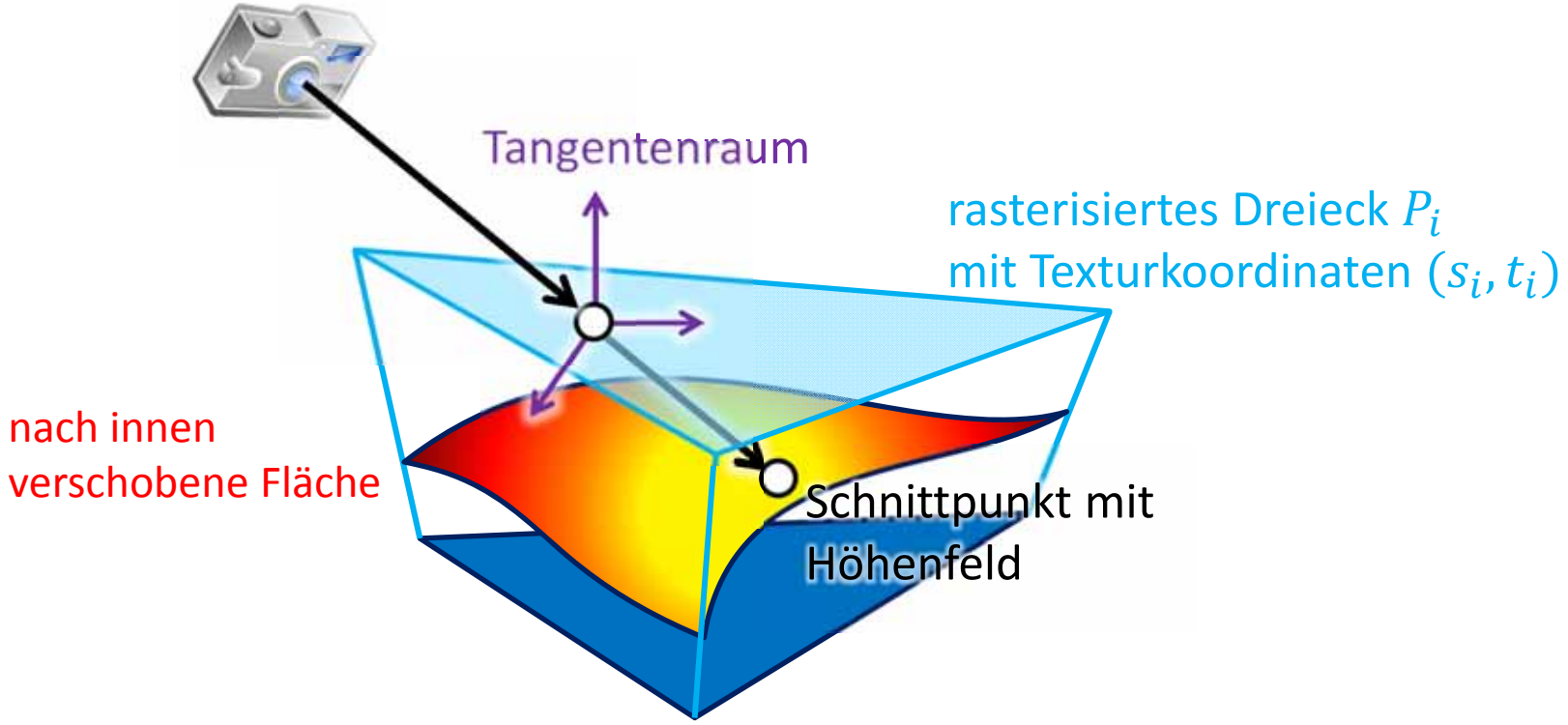
Per-Pixel Lighting

Normal Mapping

Inverse Displacement Mapping

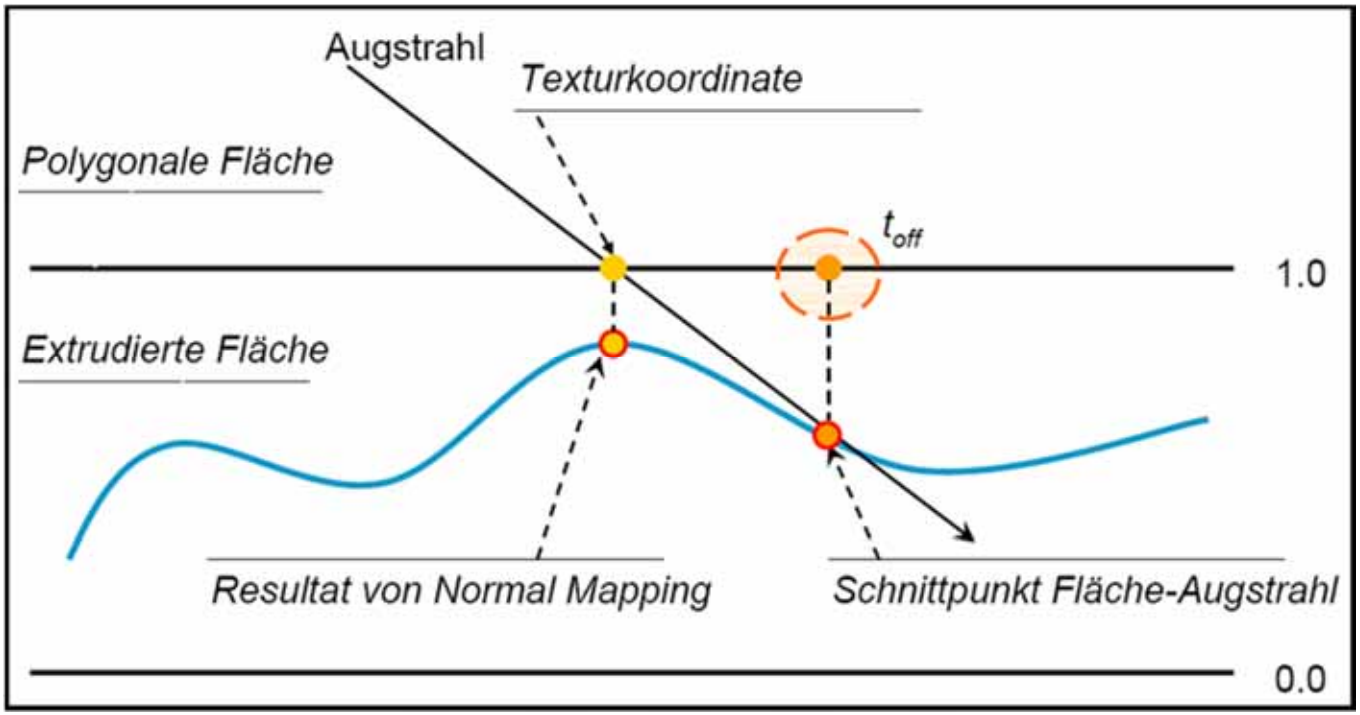
Inverse Displacement Mapping

- transformiere Augstrahl und Lichtrichtung in den Tangentenraum
- finde den Schnittpunkt mit dem Höhenfeld durch Ray Marching
  - Texturkoordinate (für das Höhenfeld) ist am Eintrittspunkt des Strahls bekannt: für diesen Oberflächenpunkt wird der Shader ausgeführt
  - transformierter Augstrahl gibt die Richtung in Texturkoord. vor (hier muss man den Tangentenraum vor der Orthonormalisierung und dementsprechend die Inverse verwenden)



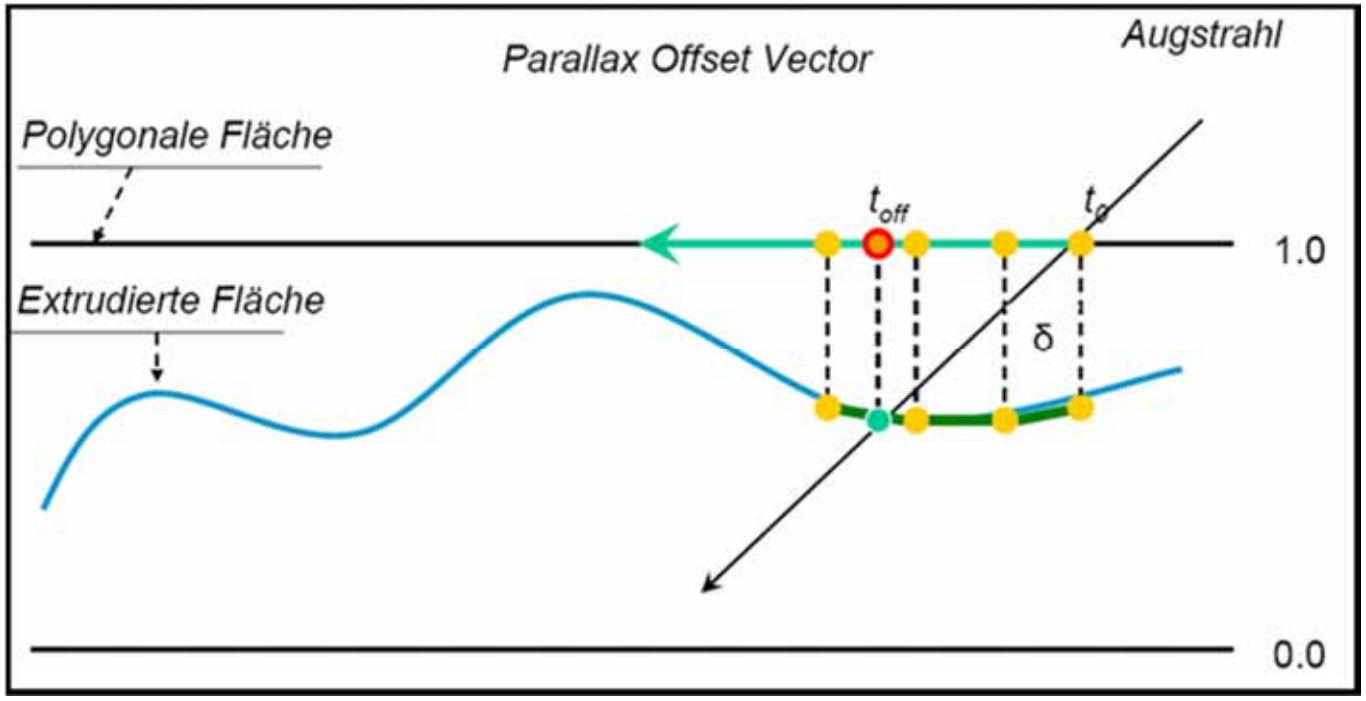
Inverse Displacement Mapping

- transformiere Augstrahl und Lichtrichtung in den Tangentenraum
- finde den Schnittpunkt mit dem Höhenfeld durch Ray Marching
  - Texturkoordinate (für das Höhenfeld) ist am Eintrittspunkt des Strahls bekannt: für diesen Oberflächenpunkt wird der Shader ausgeführt
  - transformierter Augstrahl gibt die Richtung in Texturkoord. vor
- verwende Lichtrichtung, um Beleuchtungs- und Schattenberechnung durchzuführen



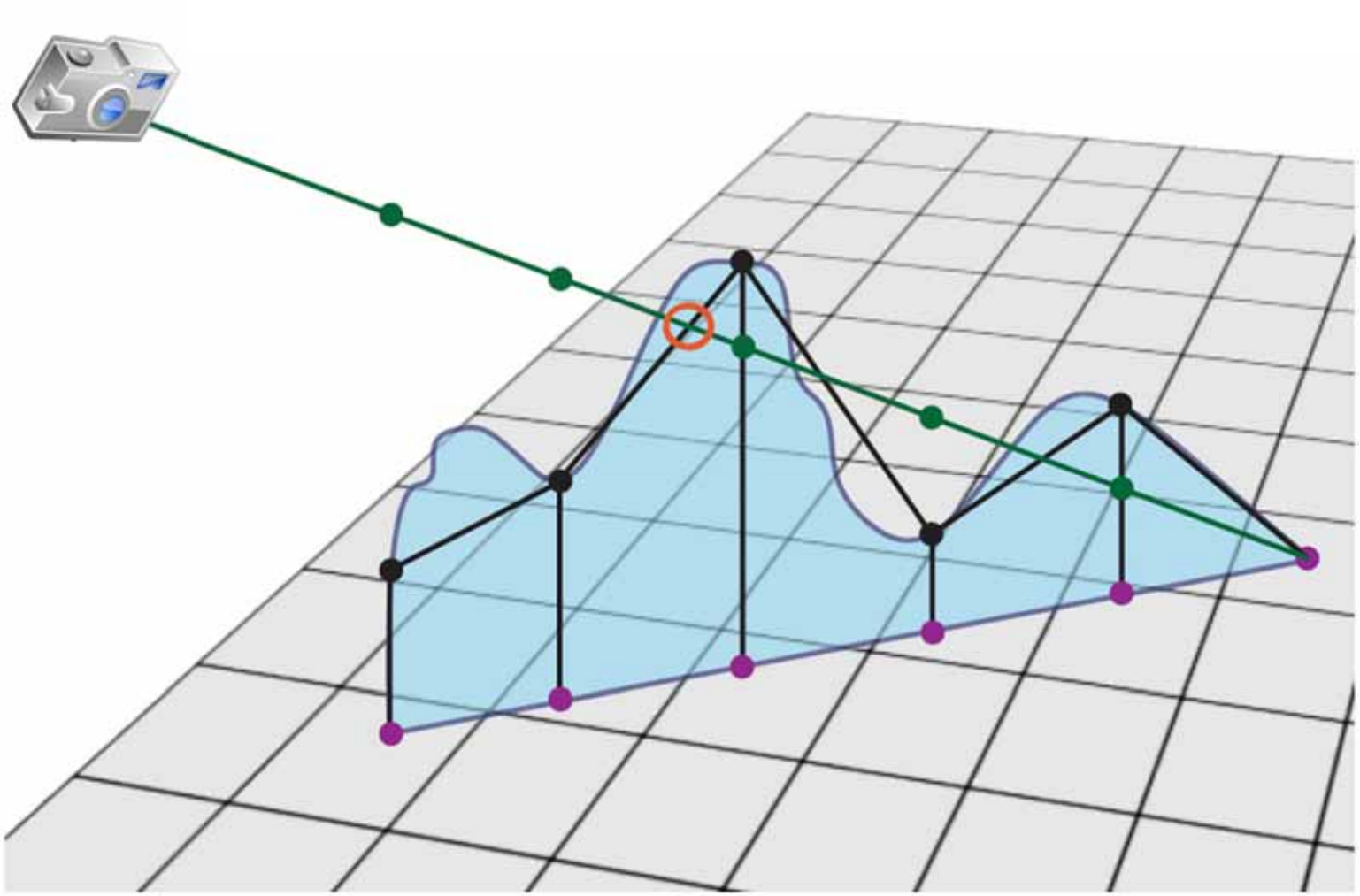
Schnittpunktberechnung

- Ray Marching entspricht einer linearen Suche
  - marschiere in kleinen Schritten am Strahl entlang
  - bis der Strahl unter dem Höhenfeld angelangt ist
- Anm. es gibt verschiedene Strategien den Schnittpunkt zu finden
  - Verfahren ähnlich der Idee von Sphere-Tracing
  - binäre Suche, u.v.m.



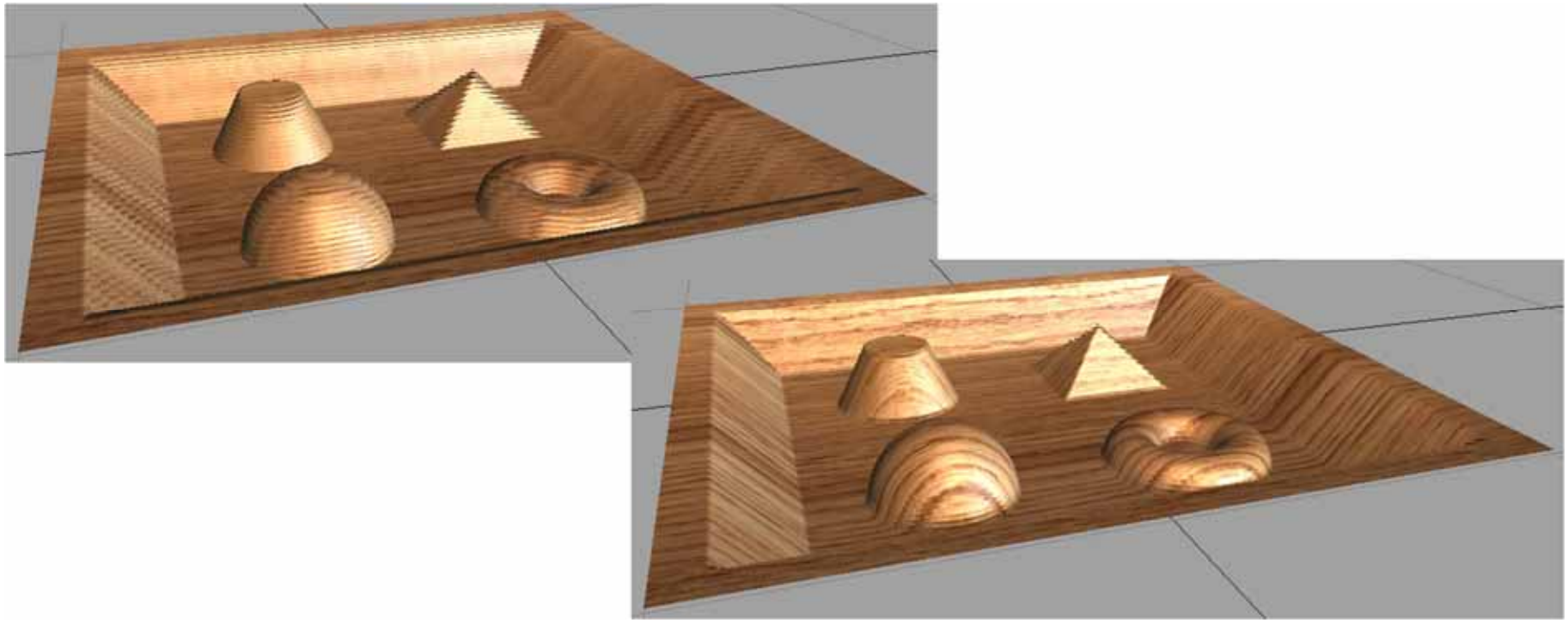
Inverse Displacement Mapping

- eine bessere Näherung des Schnittpunkts erhalten wir im Anschluss unter der Annahme, dass die Fläche stückweise linear ist



Inverse Displacement Mapping

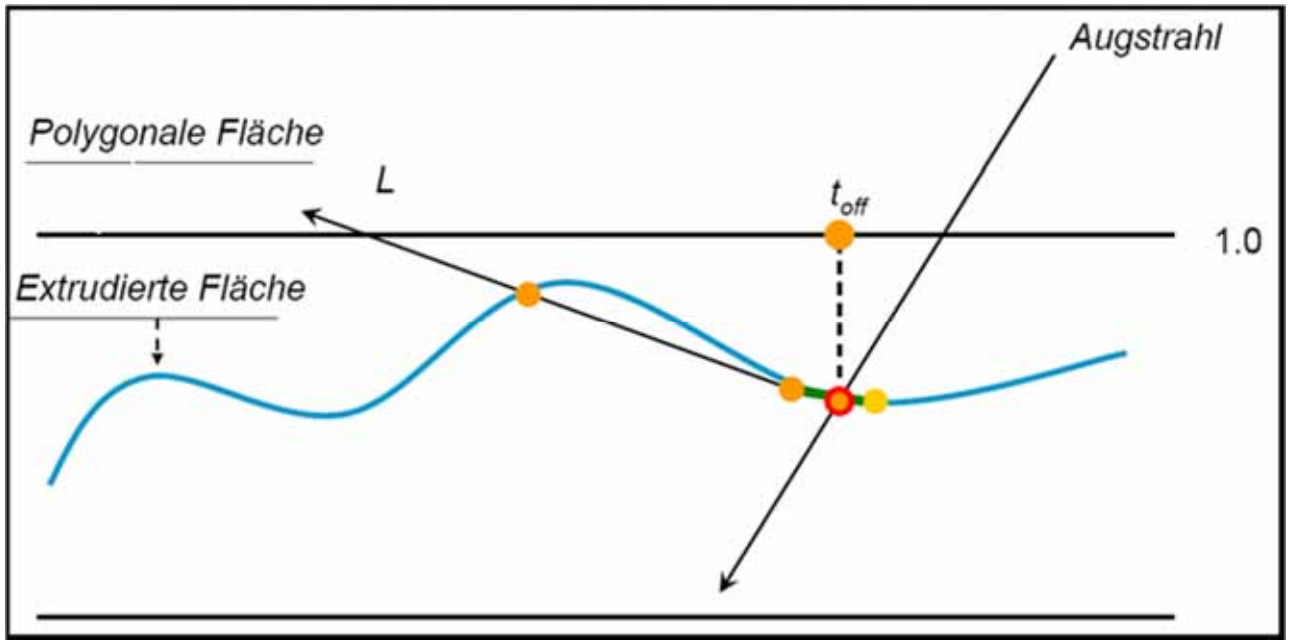
- links: einfache lineare Suche
- rechts: lineare Suche, gefolgt von der Nachbesserung



- Schrittweite / meist genügen 32-128 Schritte für das Ray Marching
  - dynamische Anpassung an die Strahlrichtung: je flacher der Augstrahl, desto weiter ist die Strecke (potentiell), die der Strahl zurücklegt, bevor er das Höhenfeld trifft

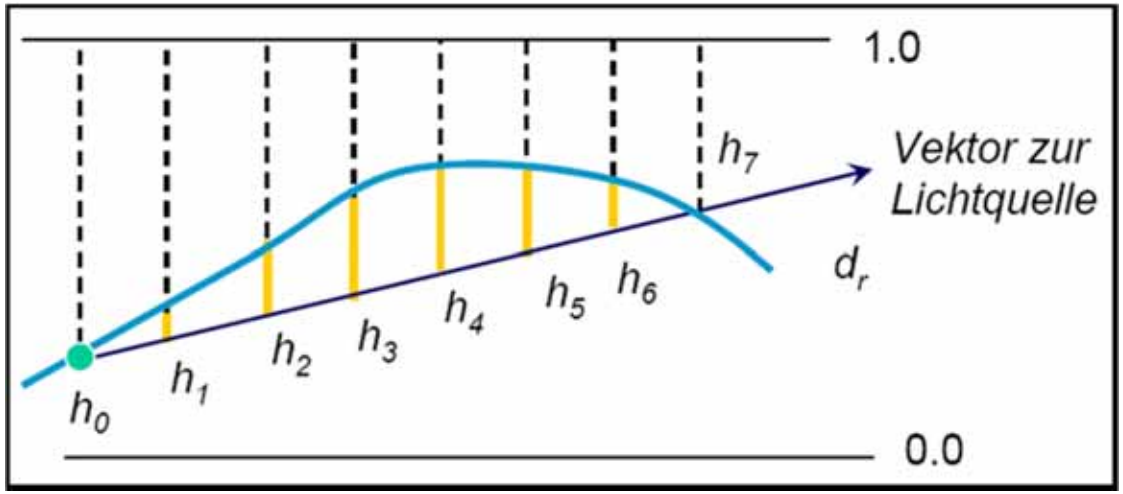
Schatten

- wenn der Schnittpunkt mit der Oberfläche gefunden ist, kann ein zweiter Strahl in Richtung der Lichtquelle verfolgt werden
- wird ein Schnittpunkt mit dem Höhenfeld gefunden, dann verschattet sich die Oberfläche selbst

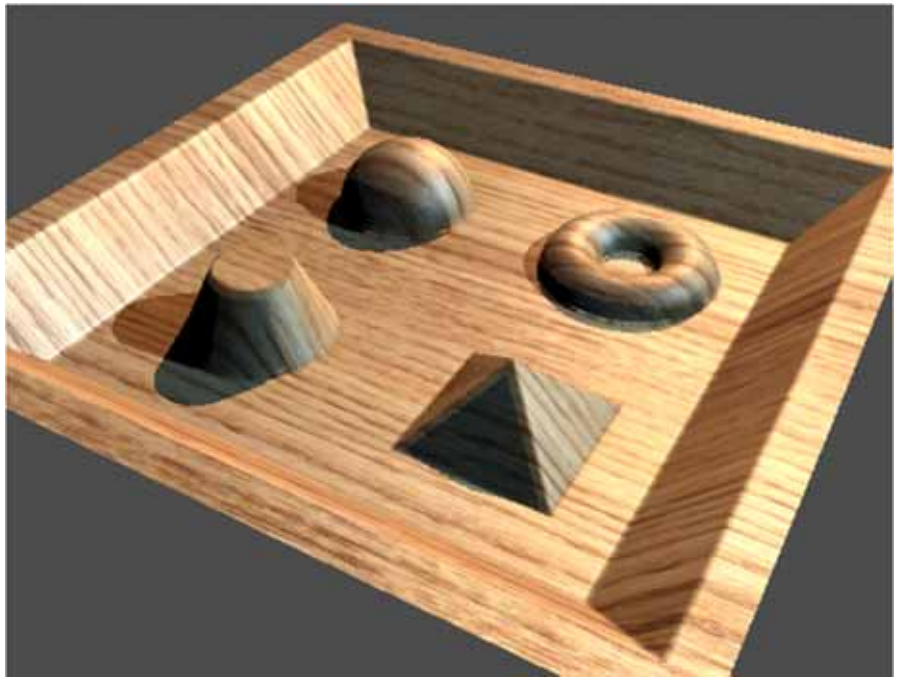


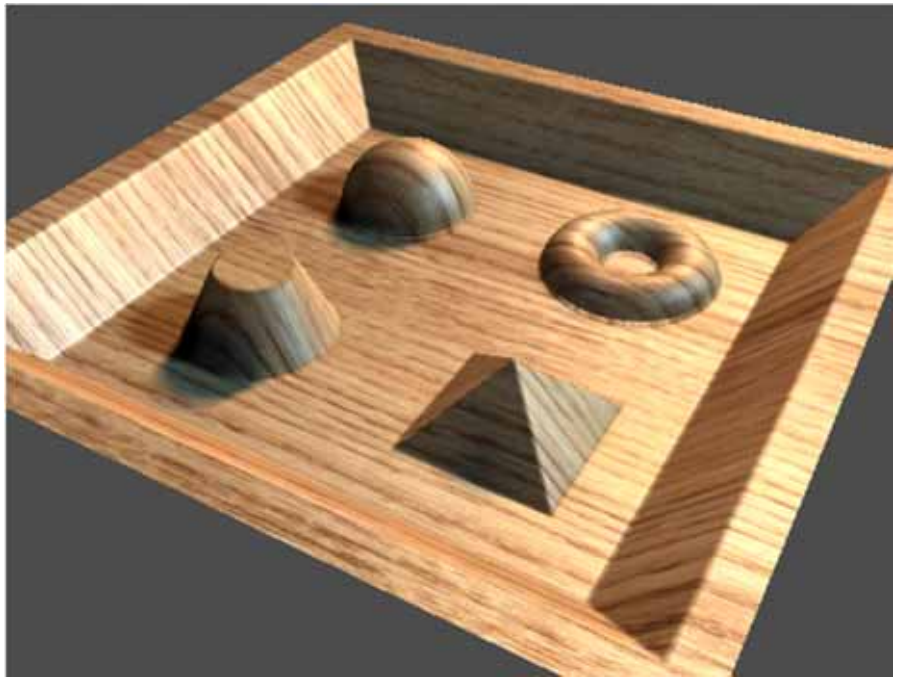
Weiche Schatten

- es können ebenfalls weiche Schatten approximiert werden
- Idee: berechne die „größte Verschattung“ durch das Höhenfeld entlang des Schattenstrahls, z.B. mit
  - Verschattung $\max_i \frac{h_i - h_0}{i+1} \cdot c$ mit
    - h_i ist der Höhenwert des i-ten Schritts entlang des Schattenstrahls
    - Konstante c (z.B. $c = 16$) modelliert Größe der LQ und Extrusion



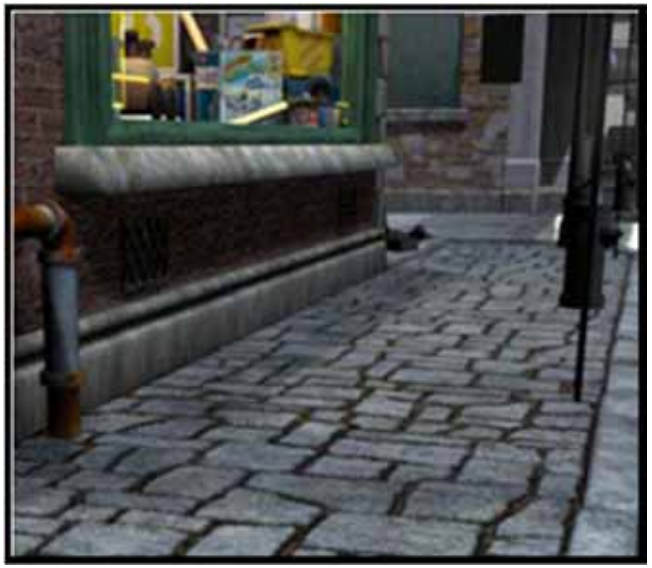
Harte und weiche Schatten





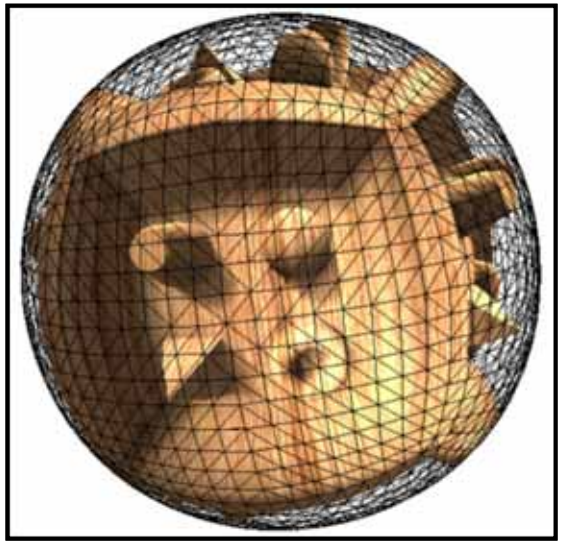
Performance

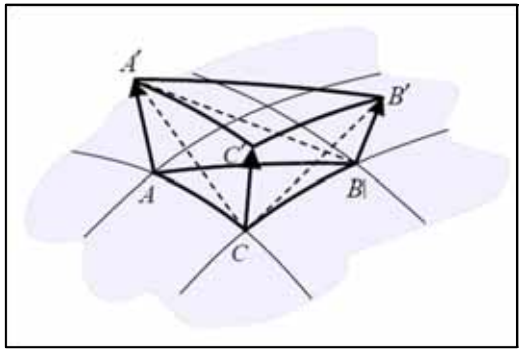
- Kontrolle der benötigten Berechnungszeit durch
  - adaptive Anzahl von Ray Marching Schritten
  - Schattenberechnungen nur für die dem Licht zugewandten Flächen
- Shader Level-of-Detail
  - für entfernte Flächen: berechne nur herkömmliches Bump Mapping
  - Skalierung des Höhenfeldes im Übergangsbereich (Bump Mapping analog zu Inverse Displacement Mapping ohne Verschiebung)

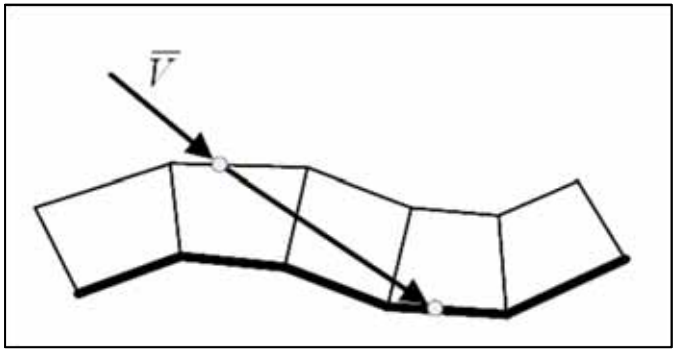


Inverse Displacement Mapping

- einfache Verfahren erzeugen nur Parallaxeffekte und Verdeckung, aber keine Silhouetten
- aufwändigere Ray Marching Verfahren, die Silhouetten korrekt berechnen, sind heute selten…







Inverse Displacement Mapping

- ohne Displacement Mapping

Inverse Displacement Mapping

- mit Displacement Mapping

Inverse Displacement Mapping

- Vertex Shader: Transformation und Extrudieren der Vertizes
- Geometry Shading: erzeuge Prismen und Hüllkörper
- Pixel Shader: Ray Marching
- nicht ganz trivial:
  - Seitenflächen sind bilineare Patches
  - Abbildung Weltkoordinaten ↔ Texturkoordinaten (Jeschke07)
  - viel Arbeit für den Pixel Shader, Flaschenhals ist aber Geometry Shader