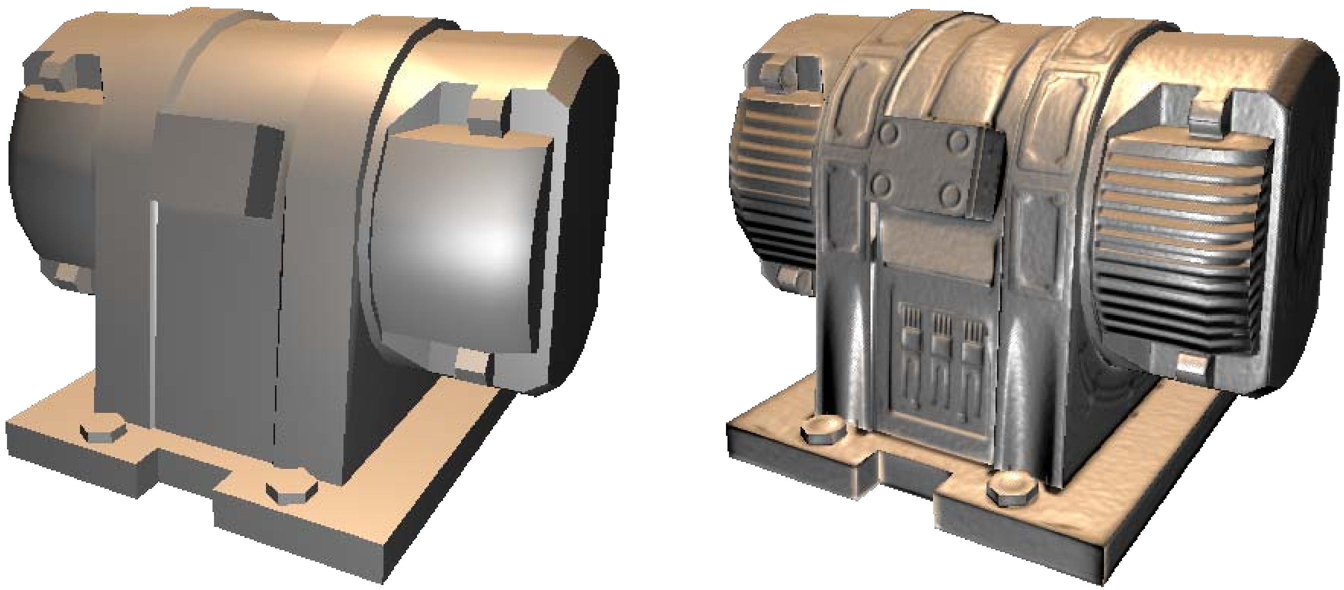
5 Bump Maps, Normal Maps, Displacement Maps

Interactive Computer Graphics

Marc Stamminger

Bump Mapping

- Bump/Normal-Mapping
  - simple geometric model with low detail
  - adapt light normal with more detailled texture
  - surface detail only in lighting
    → silhouettes still from coarse mesh

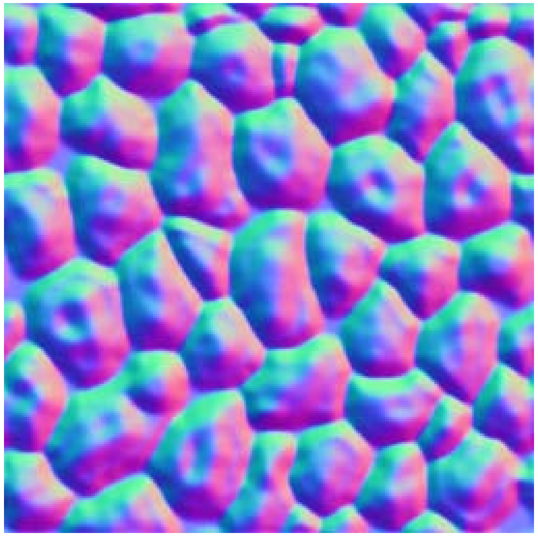


Bump Maps

- Non-Graphics Application: „Solarscreens“

Bump Mapping

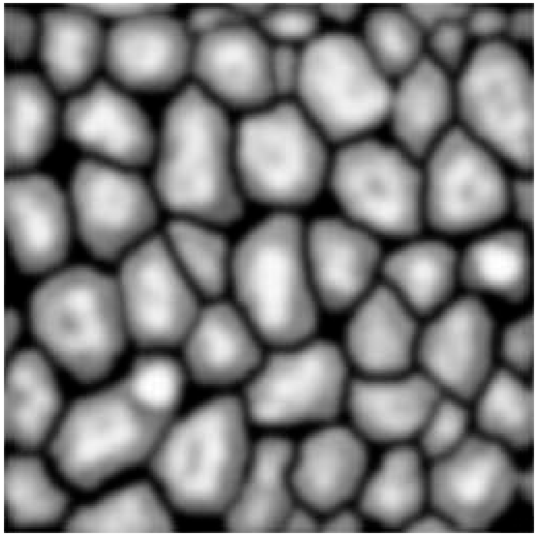
- Normal Map
  - texture with 3D normals encoded in RGB
  - 8 Bit per component sufficient
    - but also 3x10 Bit, 4x16 Bit unsigned, floating point
  - [-1,1] to [0;1]
    - R = x/2 + 0.5, G = y/2 + 0.5, …
    - x = 2R-1, …

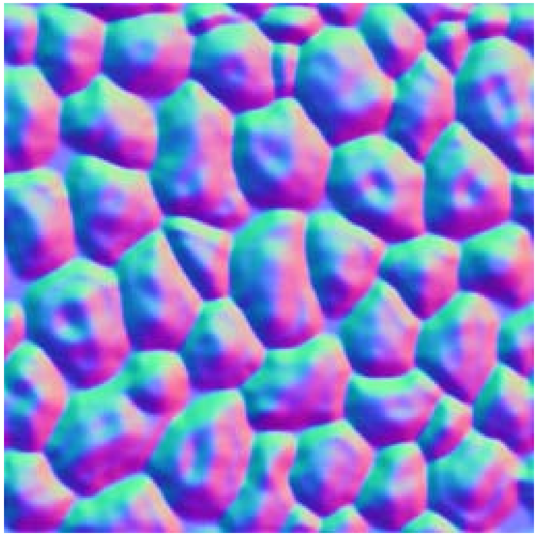


Bump Maps Generation

- from height fields

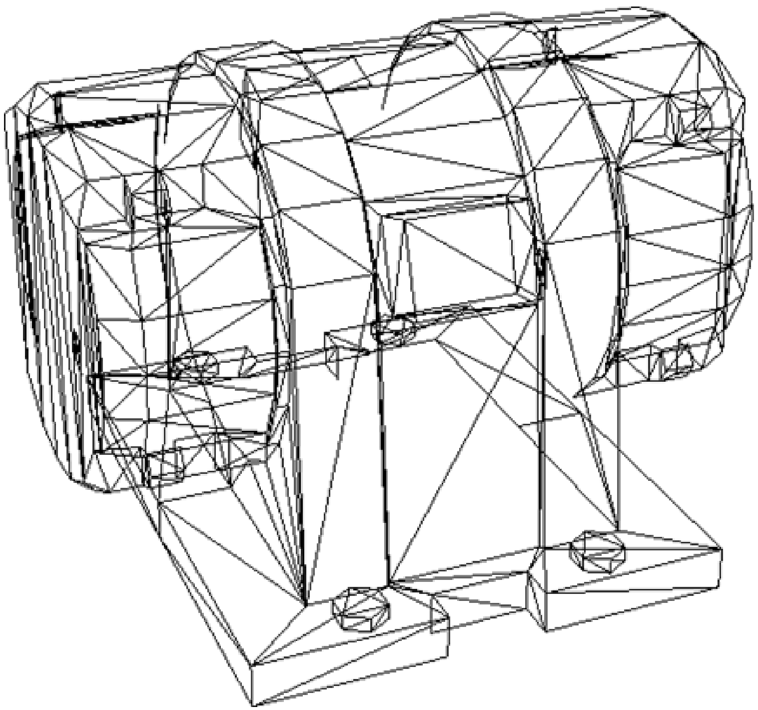
$$N(x,y)=\begin{pmatrix}2\Delta x\\0\\h(x+\Delta x,y)-h(x-\Delta x,y)\end{pmatrix}\times\begin{pmatrix}0\\2\Delta x\\h(x,y+\Delta y)-h(x,y-\Delta y)\end{pmatrix}$$



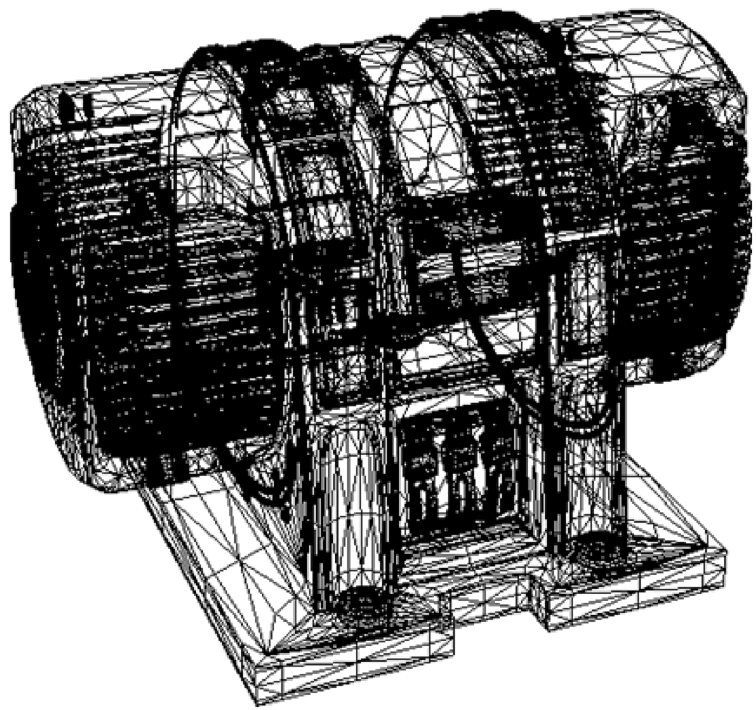


Bump Maps Generation

- Low/High Poly models



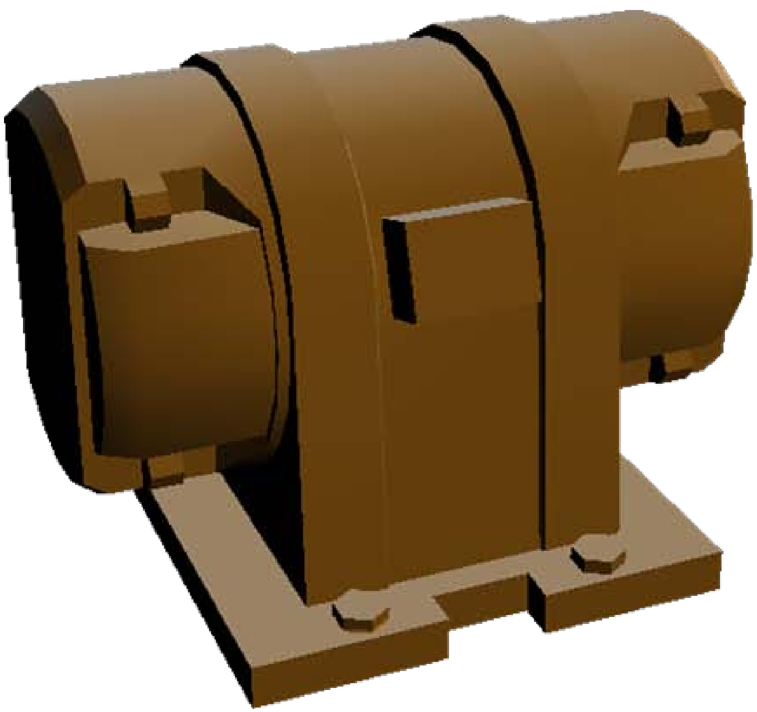
848 tris

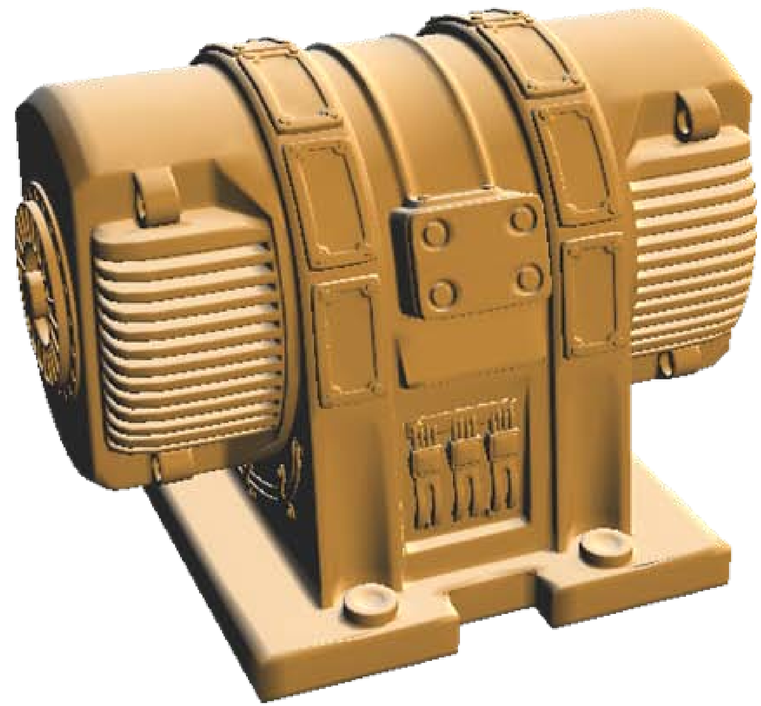


~115.000 tris

Bump Maps Generation

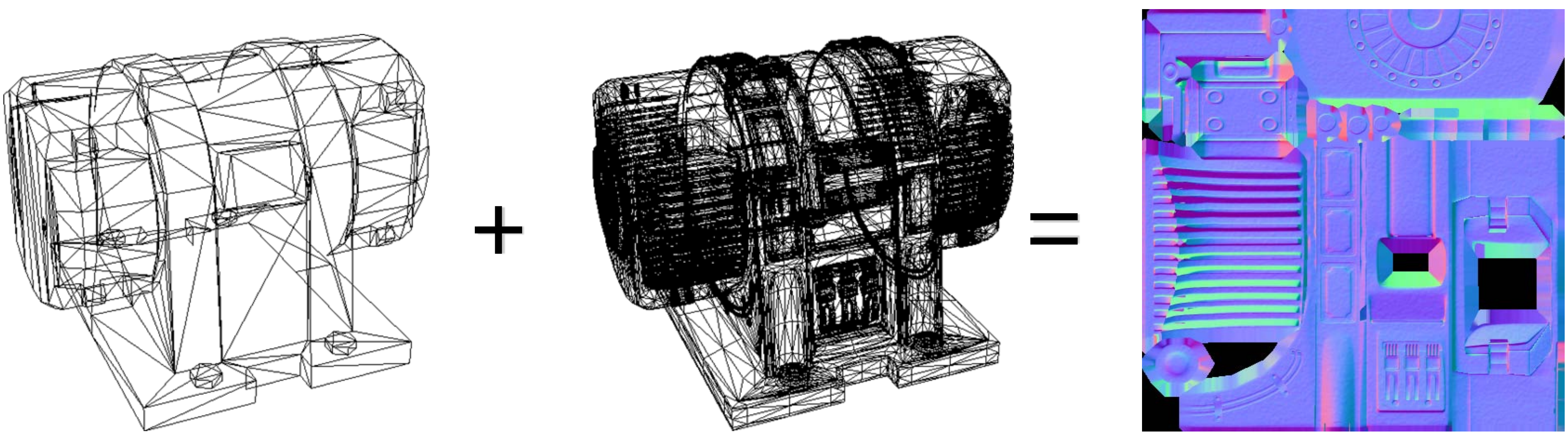
- Low/High Poly models





Bump Maps Generation

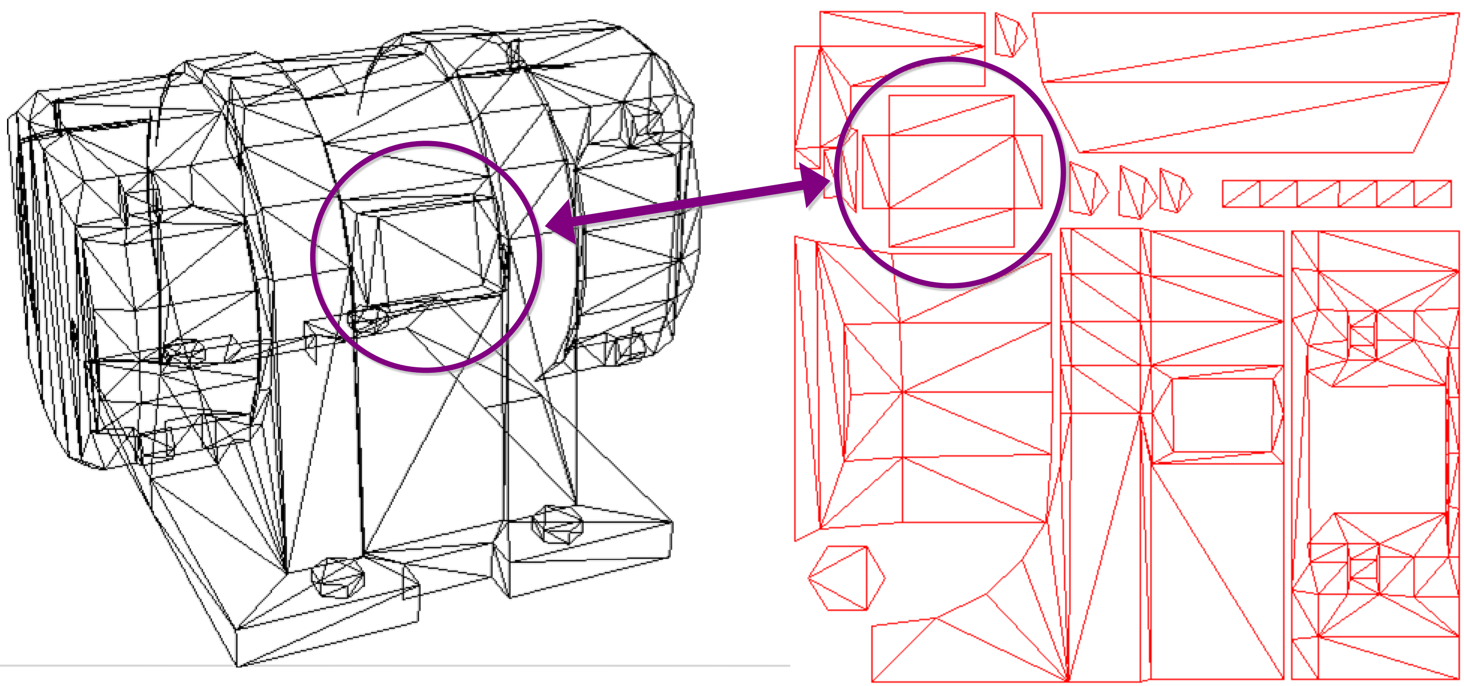
- missing detail in coarse mesh reintroduced by normal map
- generation by different tools
  - ATI Normalmapper, NVIDIA Melody, Crytek Polybump, xNormal (www.xnormal.net)



LGDV
LEHRSTUHL GRAPHISCHE DATENVERARBEITUNG

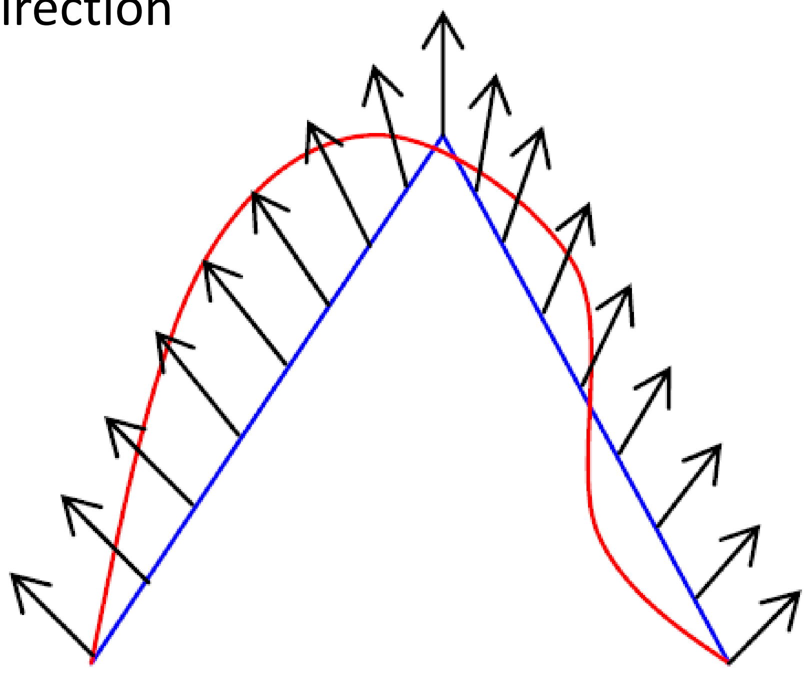
Bump Maps Generation

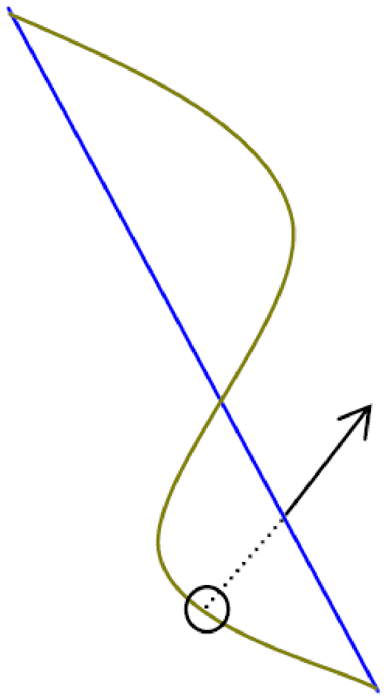
- Input
  - fine mesh: vertices and normals
  - coarse mesh: additional 2D parameterization
    - „texture atlas“



Bump Maps Generation

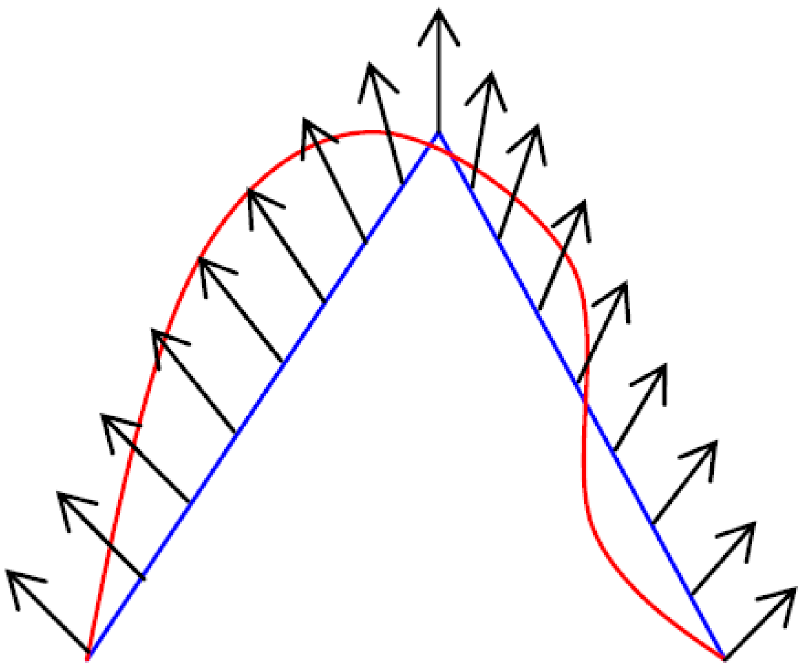
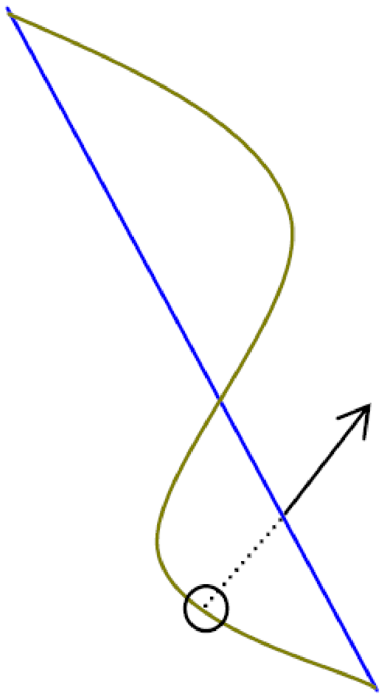
- for each texel of bump map
  - determine coordinate (on low-poly mesh)
  - determine closest intersection with high-poly mesh in normal direction





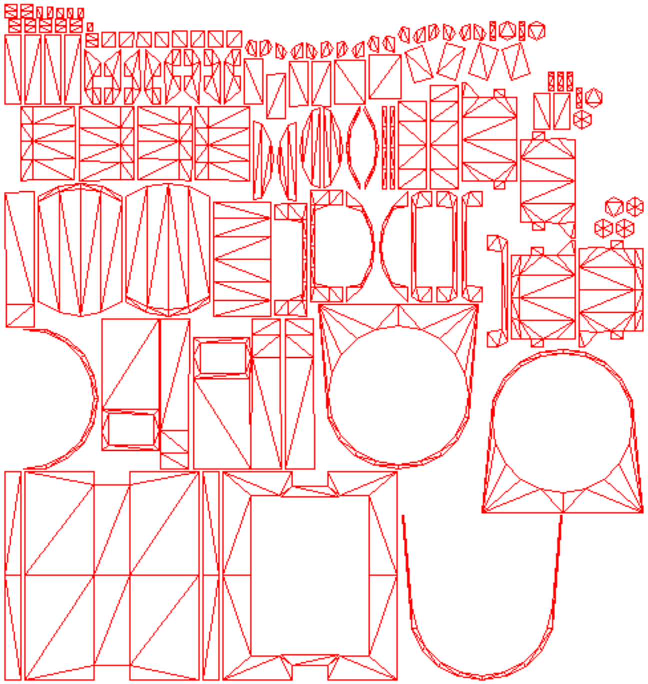
Bump Maps Generation

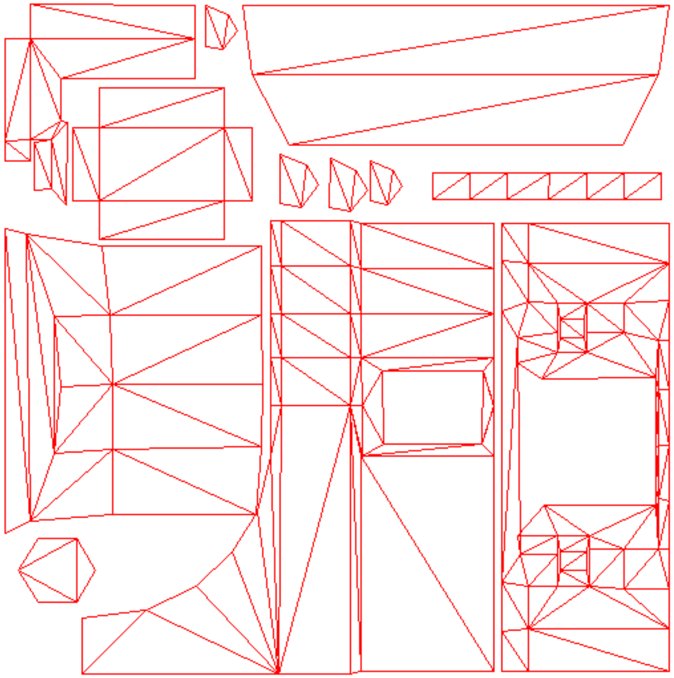
- computation using ray tracing
- intersection can also be in negative direction
- better quality by supersampling

Texture Atlas

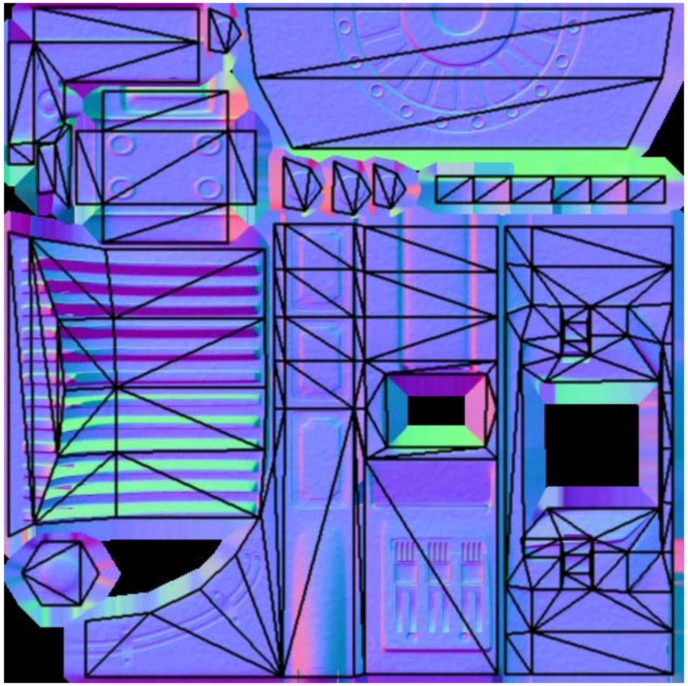
- 2D parameterization
  - automatic vs. manual
  - distortion vs. number of charts

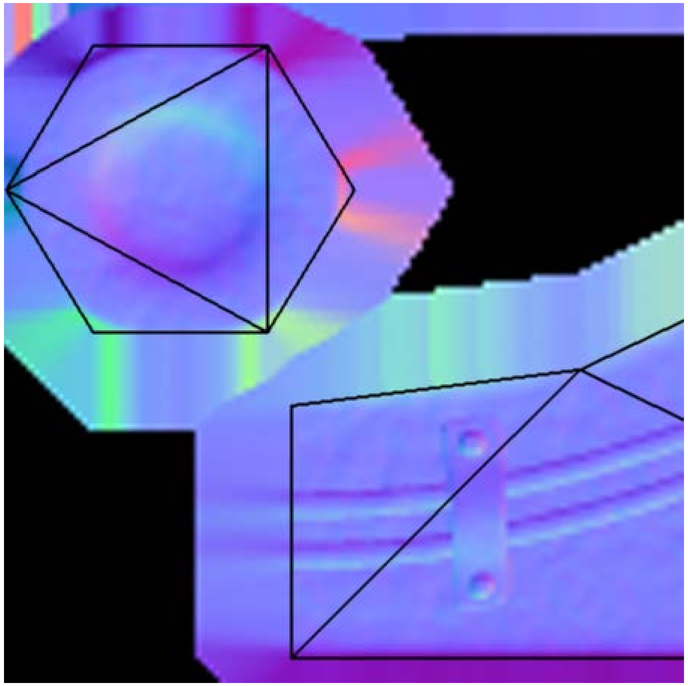




Texture Atlas

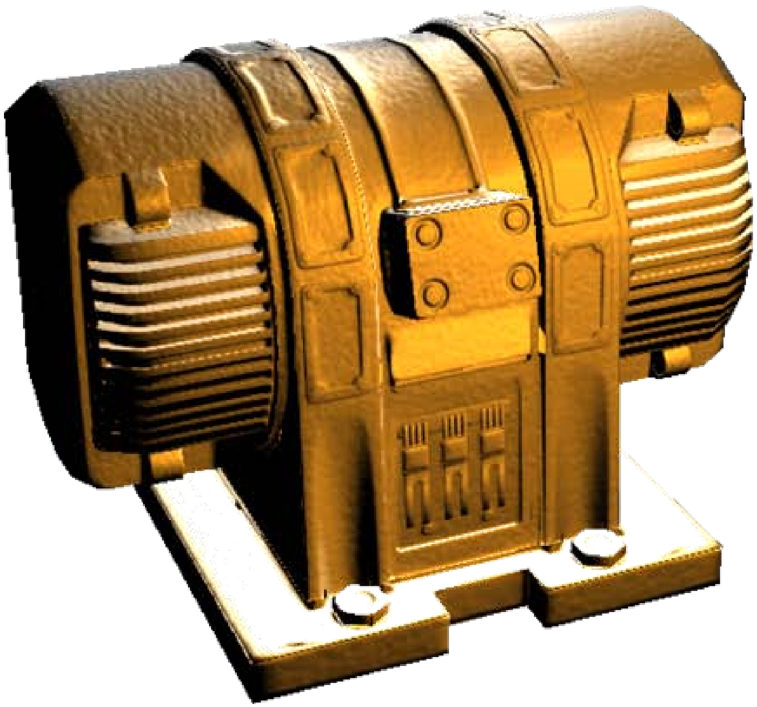
- Texture Lookups
  - bilinear filtering and mip-mapping
    - requires dilatation

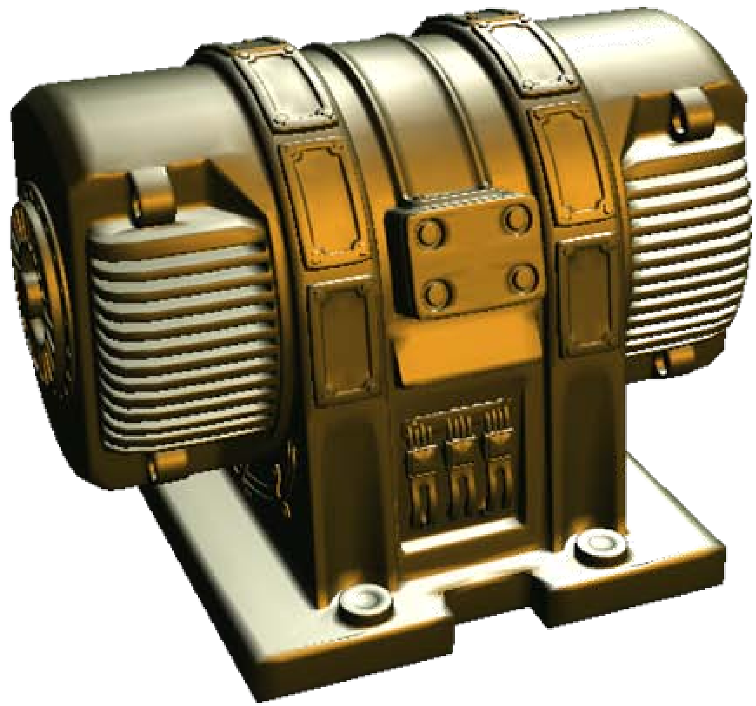




Bump Maps Generation

- result





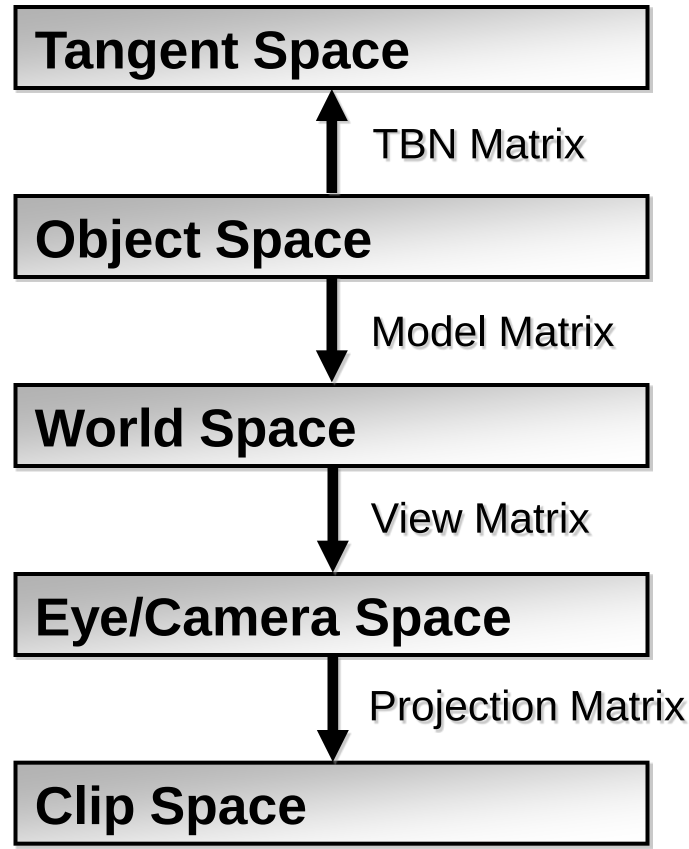
Coordinate Systems

- bump maps: color coded normal
  - but: in which coordinate system?
  - or: why are bump maps blue?

- (not so good) solution 1: object coordinates
  - same coordinate system as vertices
  - can be generated by normal mapping tools
  - no reusage of bump maps for different objects!
  - no tiling!

Coordinate Systems

- where do we compute lighting?
  - object space
    - native space for normals
  - world space
    - native space for light sources and environment map
  - eye/camera space
    - nativespace for view vector
  - Tangent Space
    - for bump maps!



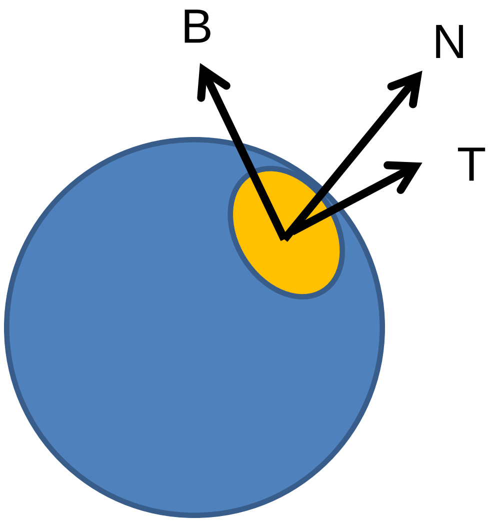
Tangent Space

- better solution: compute light in tangent space
- orthonormal basis in every vertex
  - normal = z-axis (= blue)
  - tangent: gradient of texture coordinate1
  - bitangent: cross product of normal and tangent
  - interpolation of base vectors per pixel

Tangent Space

- surface normal is z-axis of tangent space
- x and y are aligned according to texture coordinate directions

- call vectors T, B, N
- T: tangent vector
- B: bitangent
- N: normal



Tangent Space

- given triangle P_0, P_1, P_2 with texture coordinates (s_i,t_i)
- how to determine tangent space ?

- point Q on triangle with texcoords (s,t):
  $Q = P_0 + (s-s_0)T + (t-t_0)B$

Tangent Space

- $\mathbf{Q_1 = P_1 - P_0}$ $\mathbf{Q_2 = P_2 - P_0}$
- $<\Delta s_1, \Delta t_1> = <s_1 - s_0,\ t_1 - t_0>$ $<\Delta s_2, \Delta t_2> = <s_2 - s_0,\ t_2 - t_0>$

- linear equations
  - $\mathbf{Q_1} = \Delta s_1 T + \Delta t_1 B$
  - $\mathbf{Q_2} = \Delta s_2 T + \Delta t_2 B$

$$\begin{pmatrix} (\mathbf{Q_1})_x & (\mathbf{Q_1})_y & (\mathbf{Q_1})_z \\ (\mathbf{Q_2})_x & (\mathbf{Q_2})_y & (\mathbf{Q_2})_z \end{pmatrix} = \begin{pmatrix} \Delta s_1 & \Delta t_1 \\ \Delta s_2 & \Delta t_2 \end{pmatrix} \begin{pmatrix} \mathbf{T}_x & \mathbf{T}_y & \mathbf{T}_z \\ \mathbf{B}_x & \mathbf{B}_y & \mathbf{B}_z \end{pmatrix}$$

Tangent Space

- invert:

$$\begin{pmatrix} \mathbf{T}_x & \mathbf{T}_y & \mathbf{T}_z \\ \mathbf{B}_x & \mathbf{B}_y & \mathbf{B}_z \end{pmatrix} = \frac{1}{\Delta s_1 \Delta t_2 - \Delta s_2 \Delta t_1} \begin{pmatrix} \Delta t_2 & -\Delta t_1 \\ -\Delta s_2 & \Delta s_1 \end{pmatrix} \begin{pmatrix} (\mathbf{Q_1})_x & (\mathbf{Q_1})_y & (\mathbf{Q_1})_z \\ (\mathbf{Q_2})_x & (\mathbf{Q_2})_y & (\mathbf{Q_2})_z \end{pmatrix}$$

-> delivers unnormalized T and B

Tangent space

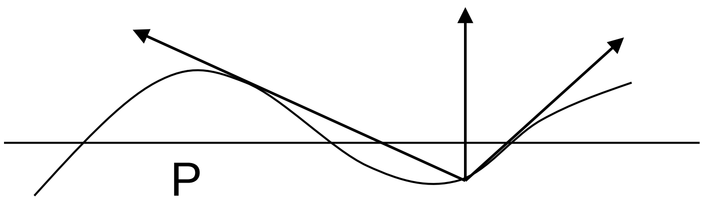
- compute T,B,N for all triangles
- for every vertex: average TBN over surrounding triangles
  → vertex attribute
  → TBN interpolated over triangles
  → per-pixel tangent space

Bump Mapping

- influences lighting computation only
- requires per-pixel lighting (cheap nowadays)
- no shadows due to bumps
  - but: horizon maps, parallax mapping (follow)
- mip-mapping
  - theoretically: supersampling of light results
  - practically: averaged normals
  - unsolved problem, mainly empirical results

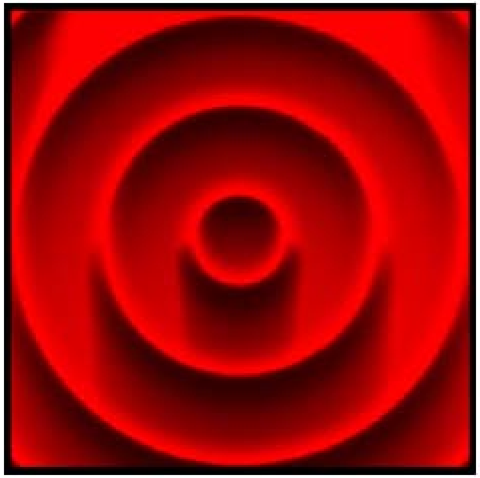
Interactive Horizon Mapping

- Sloan and Cohen 2000
  - bumps should cast shadows
  - compute horizon around a point P
  - horizon height depends on direction
  - shadow if light source below horizon



  - example:

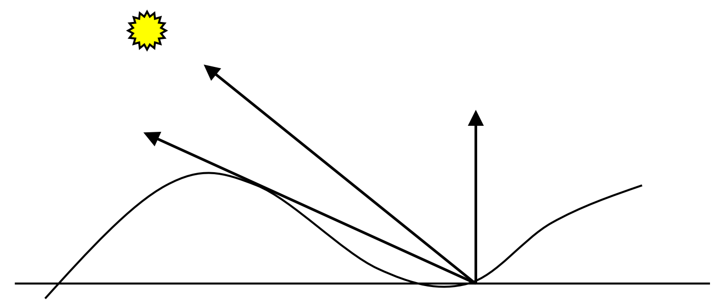
normal map

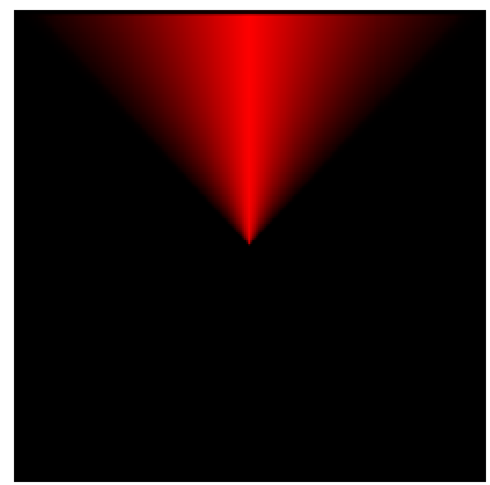


horizon height
for direction NORTH

Interactive Horizon Mapping

- Sloan and Cohen 2000
  - idea: precompute horizon height for eight directions (N,NE,E,…)
  - compute “height angle” of light source
  - for all eight directions:
    - test whether light above horizon -> 0/1 shadow result
    - blend 0/1 results with basis function textures
    - one basis function texture for each direction (one in main direction, blend to zero for other directions)
  - put four textures into one’s RGBA-channels
  - difficult to apply with standard OpenGL, but possible (see paper)

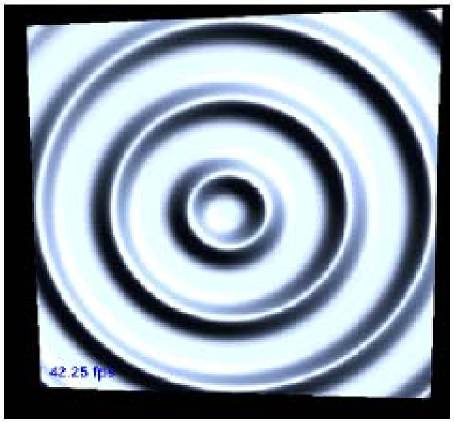




NORTH basis function texture

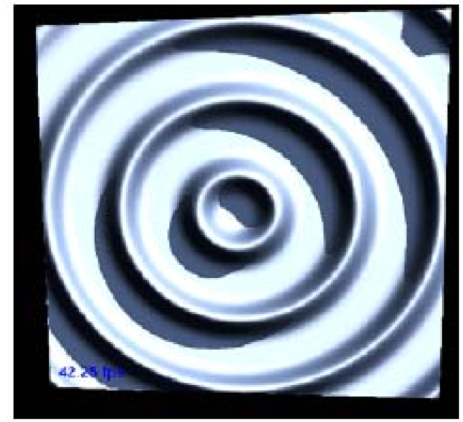
Interactive Horizon Mapping

- examples



without shadows

with shadows



with bright shadows

LGDV
LEHRSTUHL GRAPHISCHE DATENVERARBEITUNG

Parallax Bump Mapping

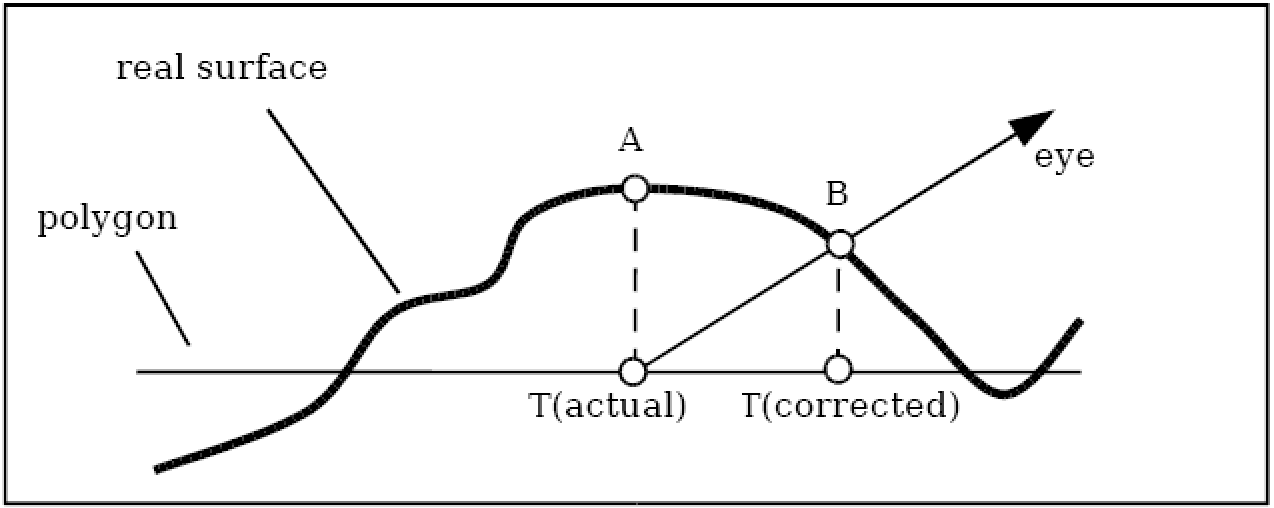
- elegant extension of bump mapping with texture
  - bumps should result in parallax effect on texture
  - estimate parallax due to bump texture
  - apply parallax by texture coordinate offset

- original idea:
  - Tomomichi Kaneko, et al. “Detailed Shape Representation with Parallax Mapping”, ICAT, 2001.
- extended:
  - Walsh, “Parallax Mapping with Offset Limiting”, Infiniscape Tech Report, 2004

Parallax Bump Mapping

Parallax Bump Mapping

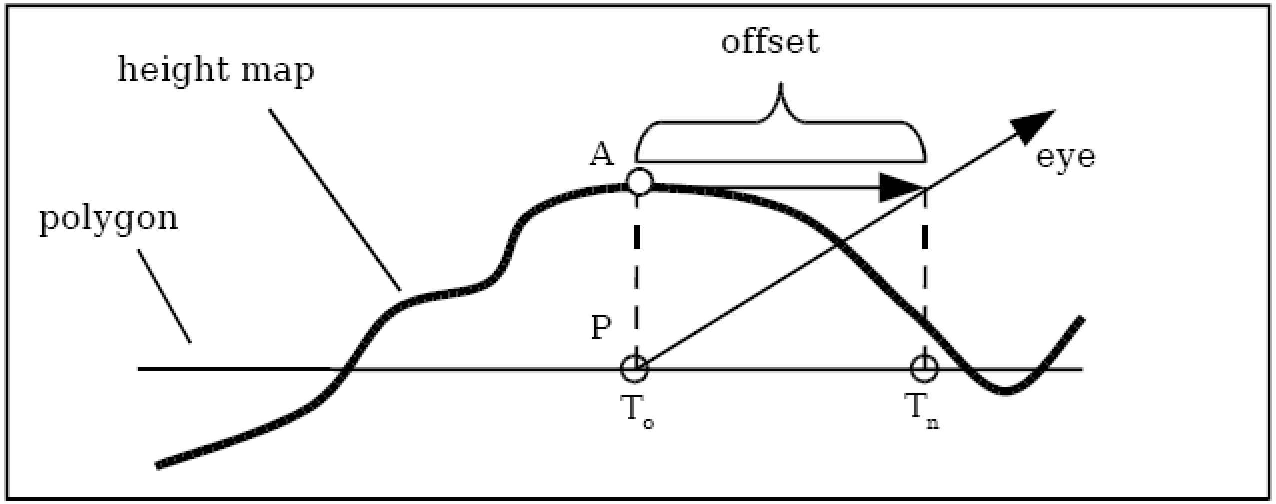
- eye sees polygon point **T(actual)**
- but **T(actual)** is offset to **A** by bump map
- what eye sees is **B**
- and **B** is textured according to **T(corrected)**



- how can we compute **T(corrected)** ?

Parallax Bump Mapping

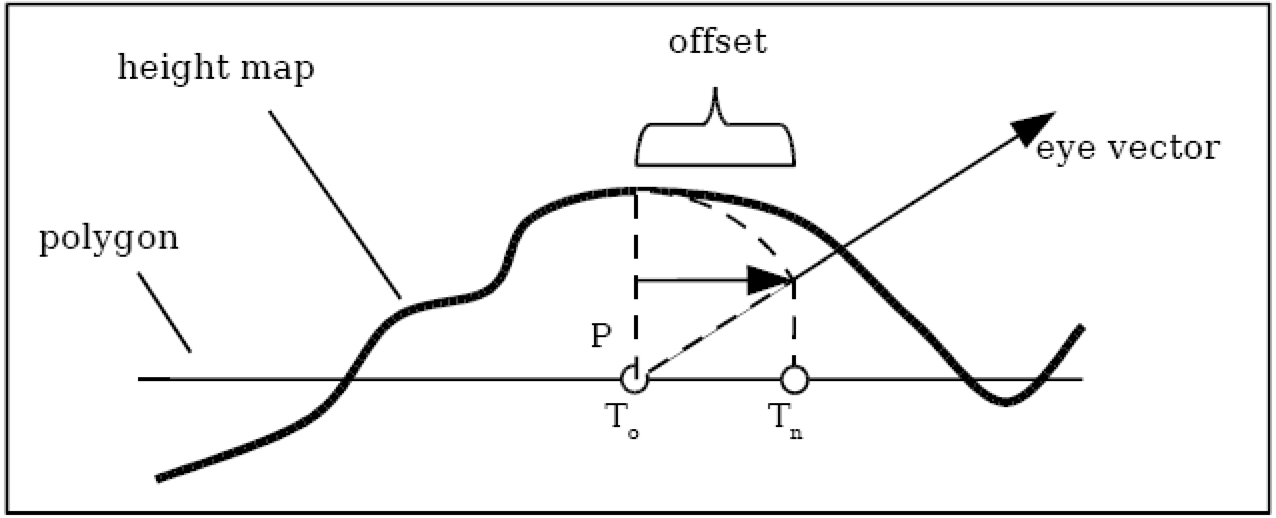
- Kaneko:
  - eye sees polygon point **P** with texture coordinate **T0**
  - **P** is moved to **A** by bump map
  - compute parallax offset by assuming that bump map is constant around **P**
  - new texture coordinate **Tn = T0 + F*eye.xy/eye.z**
    (**eye** in texture coordinate space, **F** factor due to bump height)
  - approximation, in particular for grazing angles (eye.z small)



LEHRSTUHL GRAPHISCHE DATENVERARBEITUNG

Parallax Bump Mapping

- Welsh: parallax mapping with offset limiting
  - Kanekos approximation particularily wrong for grazing angles
  - limit offset for grazing angles:
    Tn = T0 + F*eye.xy

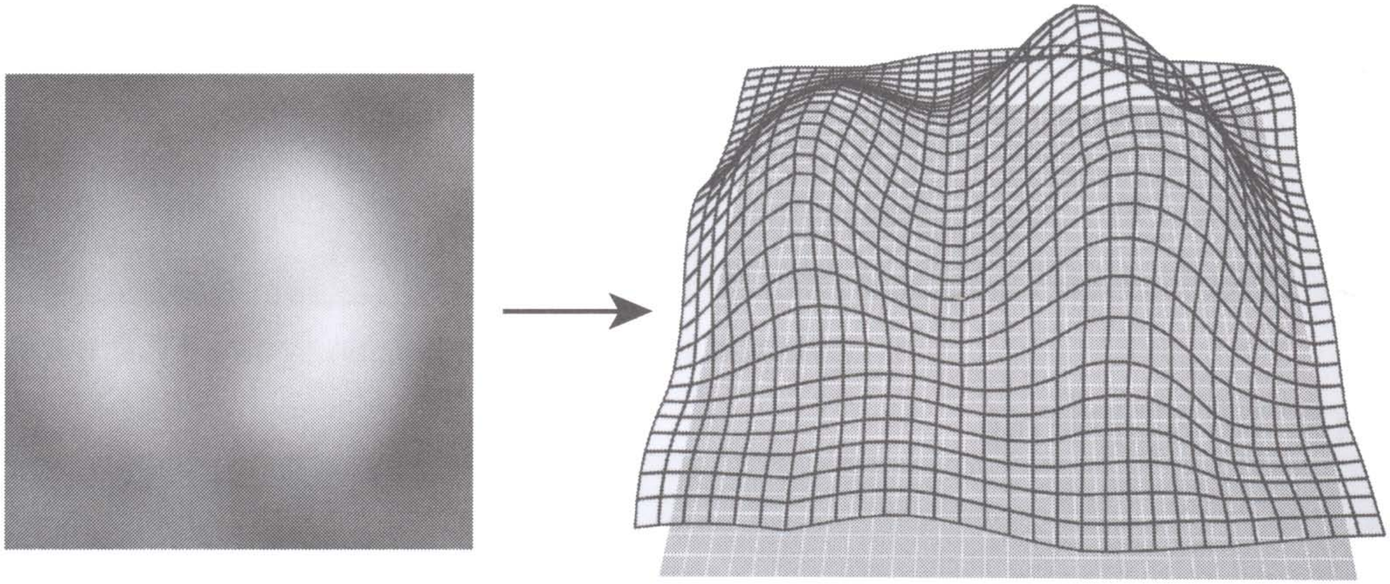


Parallax Bump Mapping

- demo

Displacement Mapping

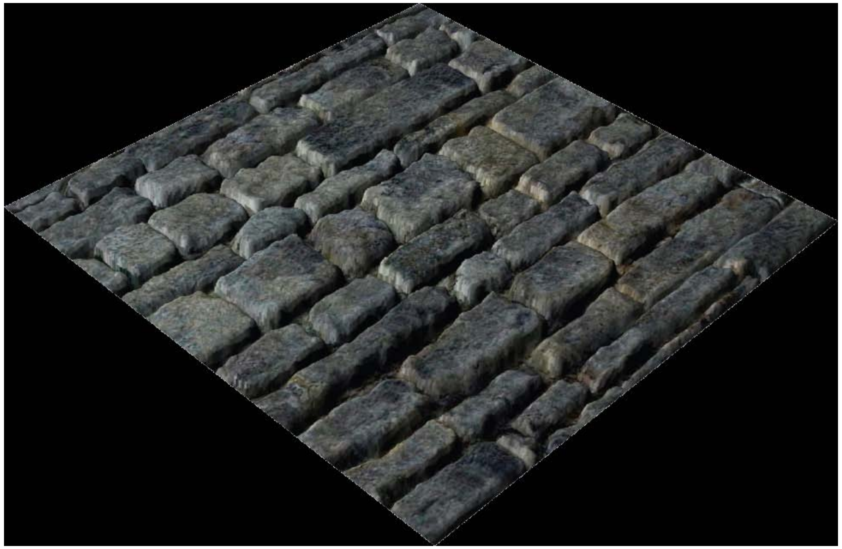
- data like for bump maps
- but really generate displaced geometry



Displacement Mapping

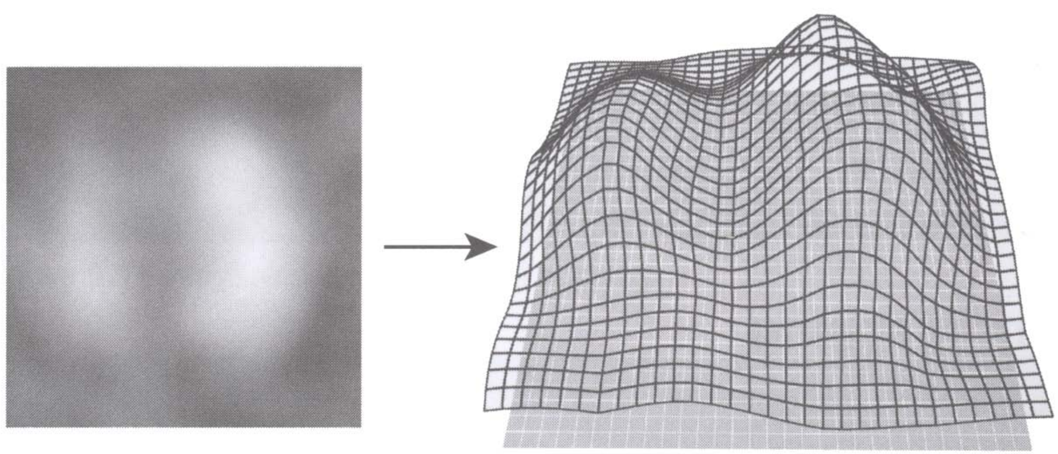
- bump mapping

- displacement mapping



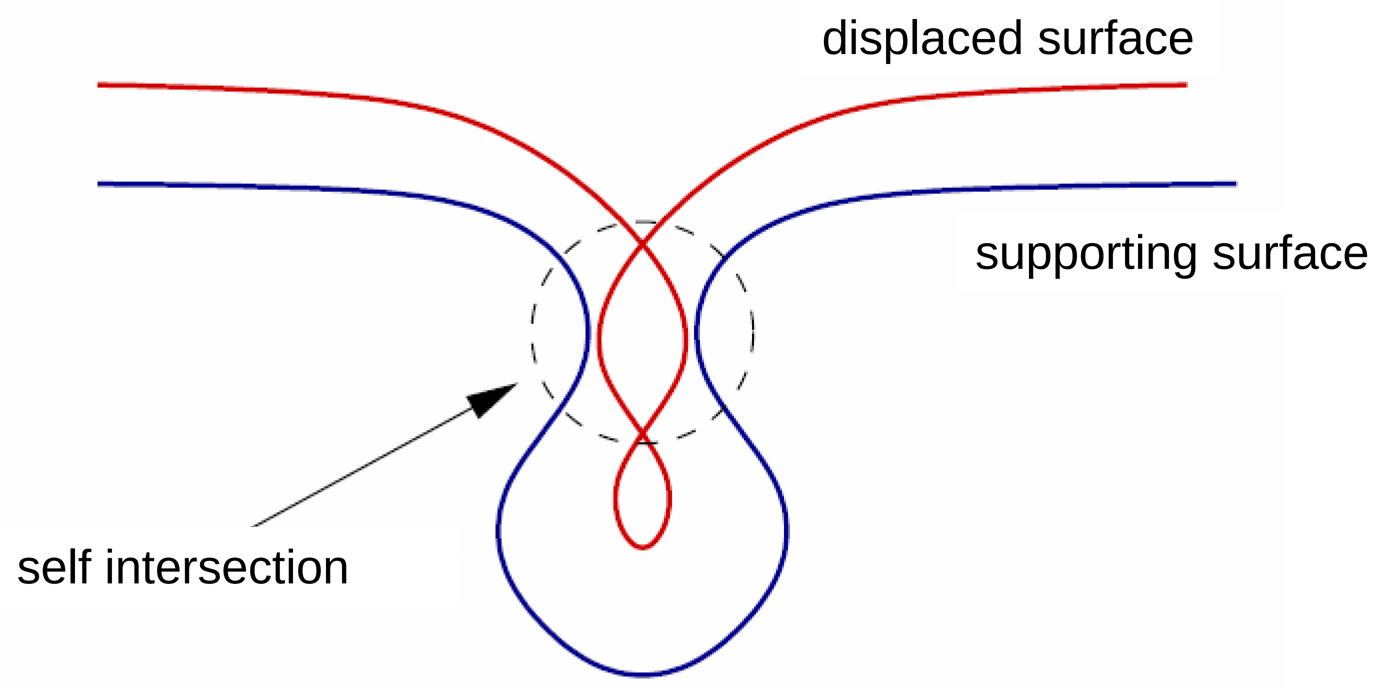
Vertex based displacement maps

- displacement in vertex shader
  - render finely tessellated basis mesh with texture coordinates
  - vertex shader reads texture, displaces vertices
- issues:
  - how to generate tessellated basis mesh ?



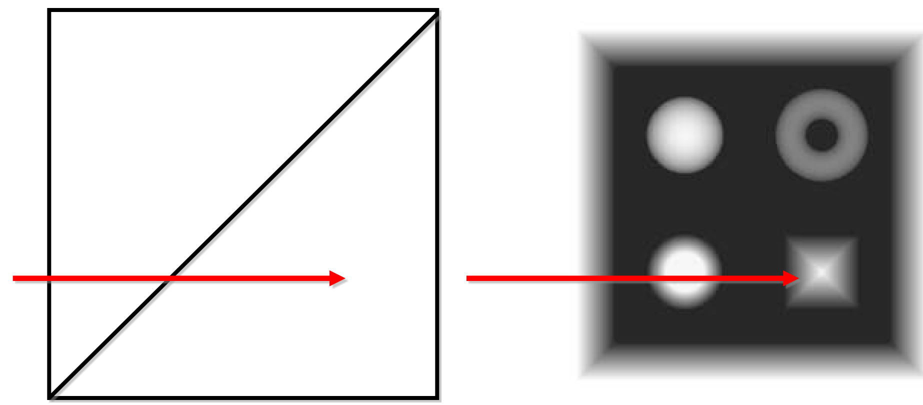
Displacement Mapping

- caveat: self intersections
  - positive displacement in concave regions
  - negative displacement in convex regions



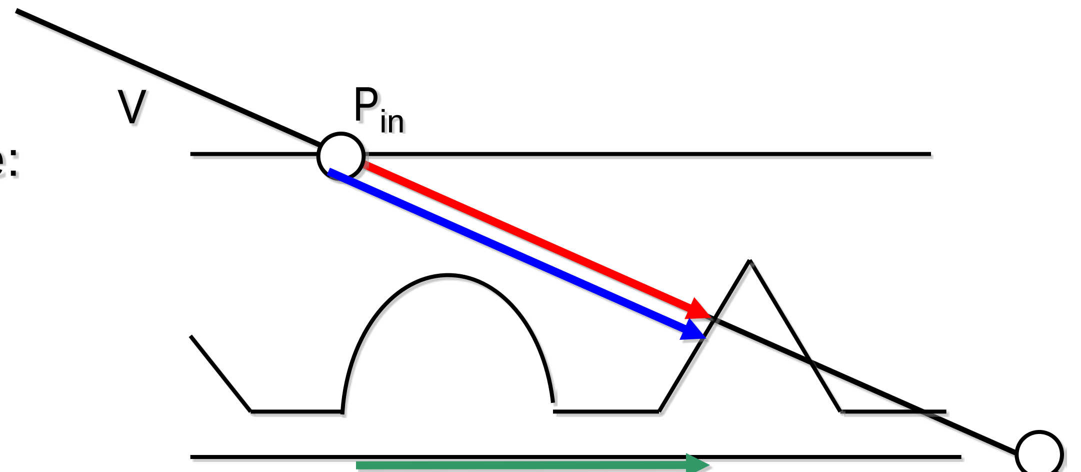
Displacement Mapping by Ray Casting

- coarse mesh + displacement texture
- assume only negative displacement
- → ray casting in
  fragment shader



find intersetion point:

$P=P_{in}+d*V$

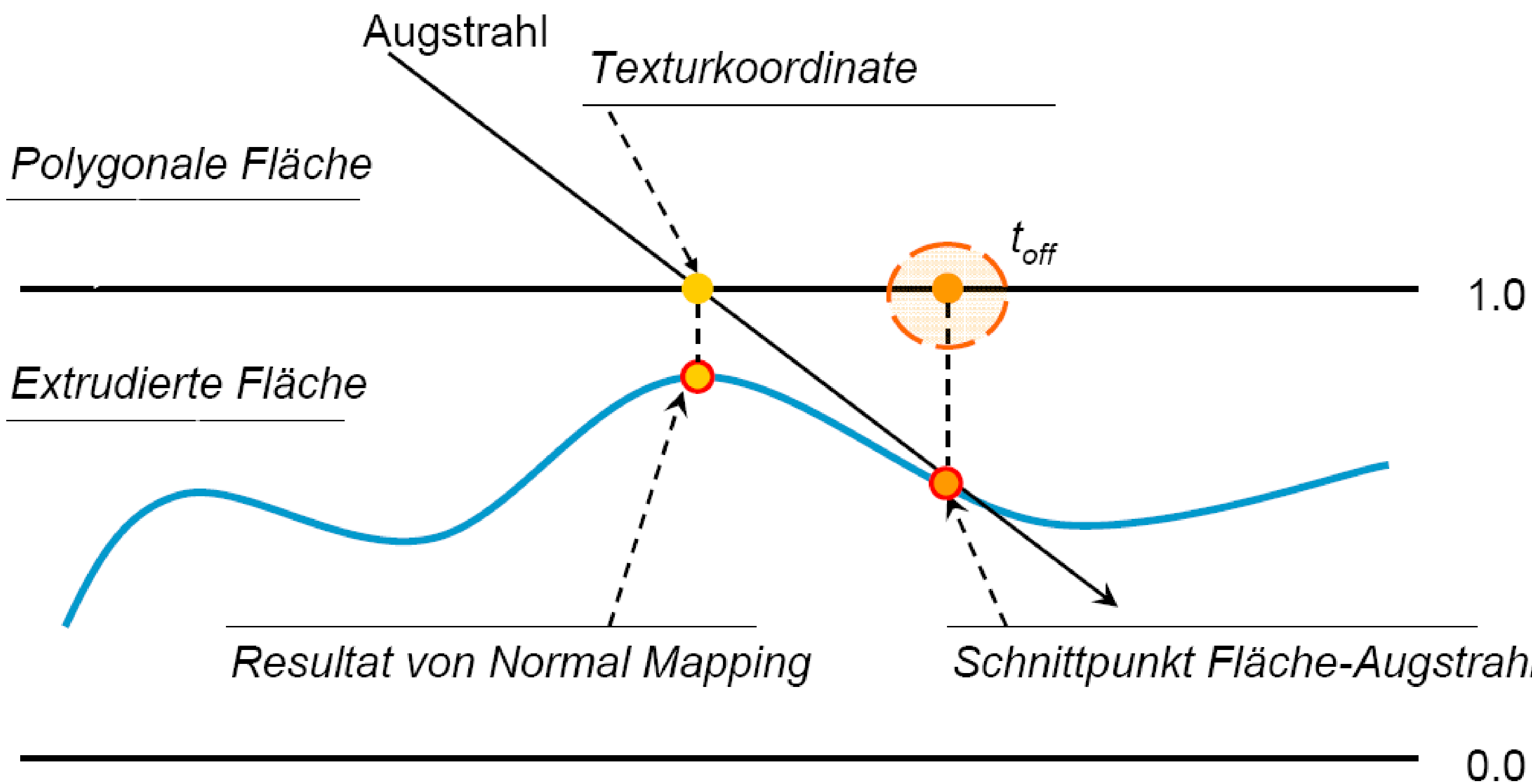


find displaced texture coordinate:

$T=T_{in}+s*V_t$

Displacement Mapping by Ray Casting

- Raycasting in Fragment Shader

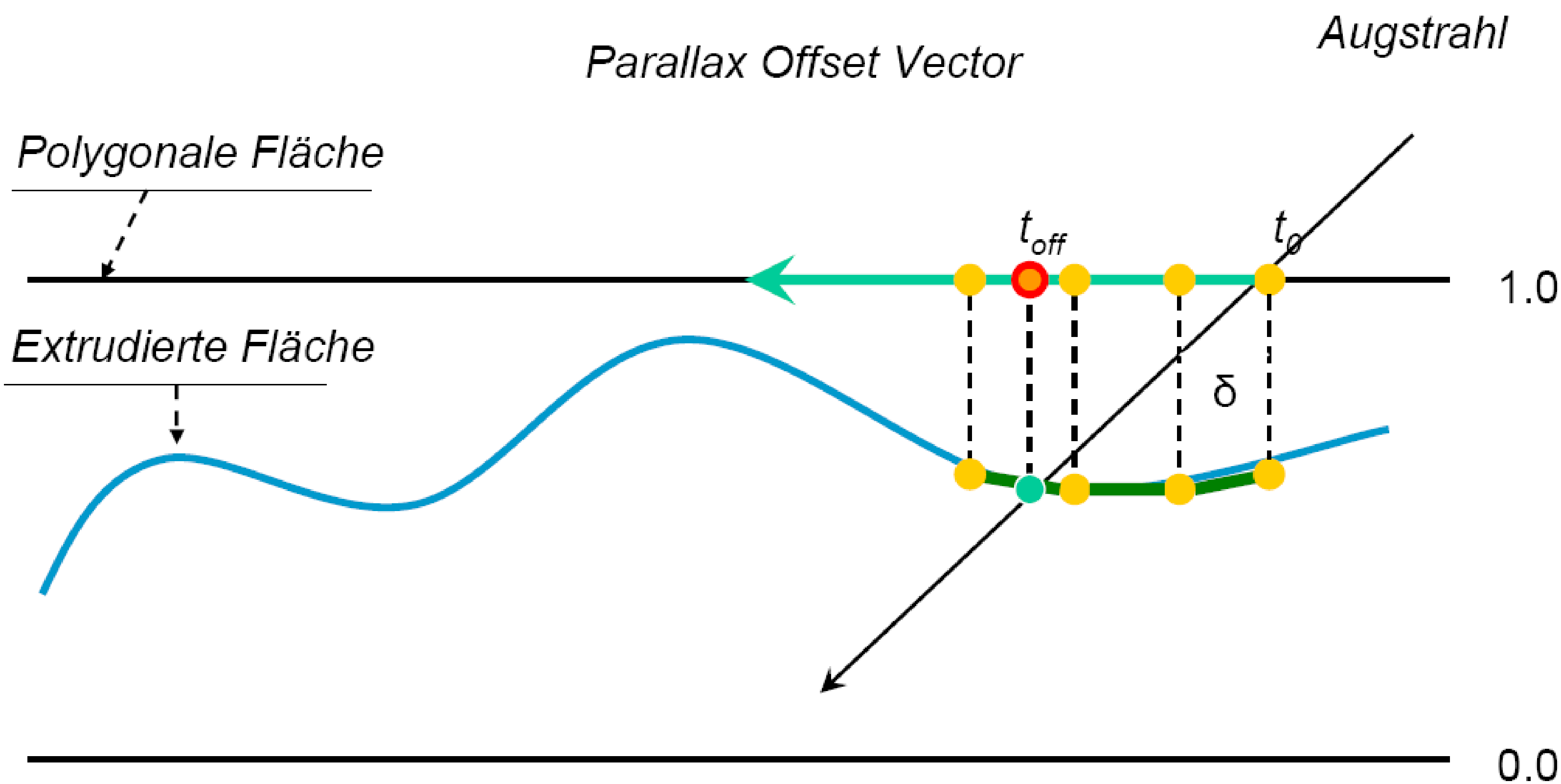


Displacement Mapping by Ray Casting

- per vertex
  - determine eye ray, light ray etc. in tangent space
- per fragment
  - find intersection of eye ray with height field
    → offsetted surface point and texture coordinate
  - + shadow ray to light source

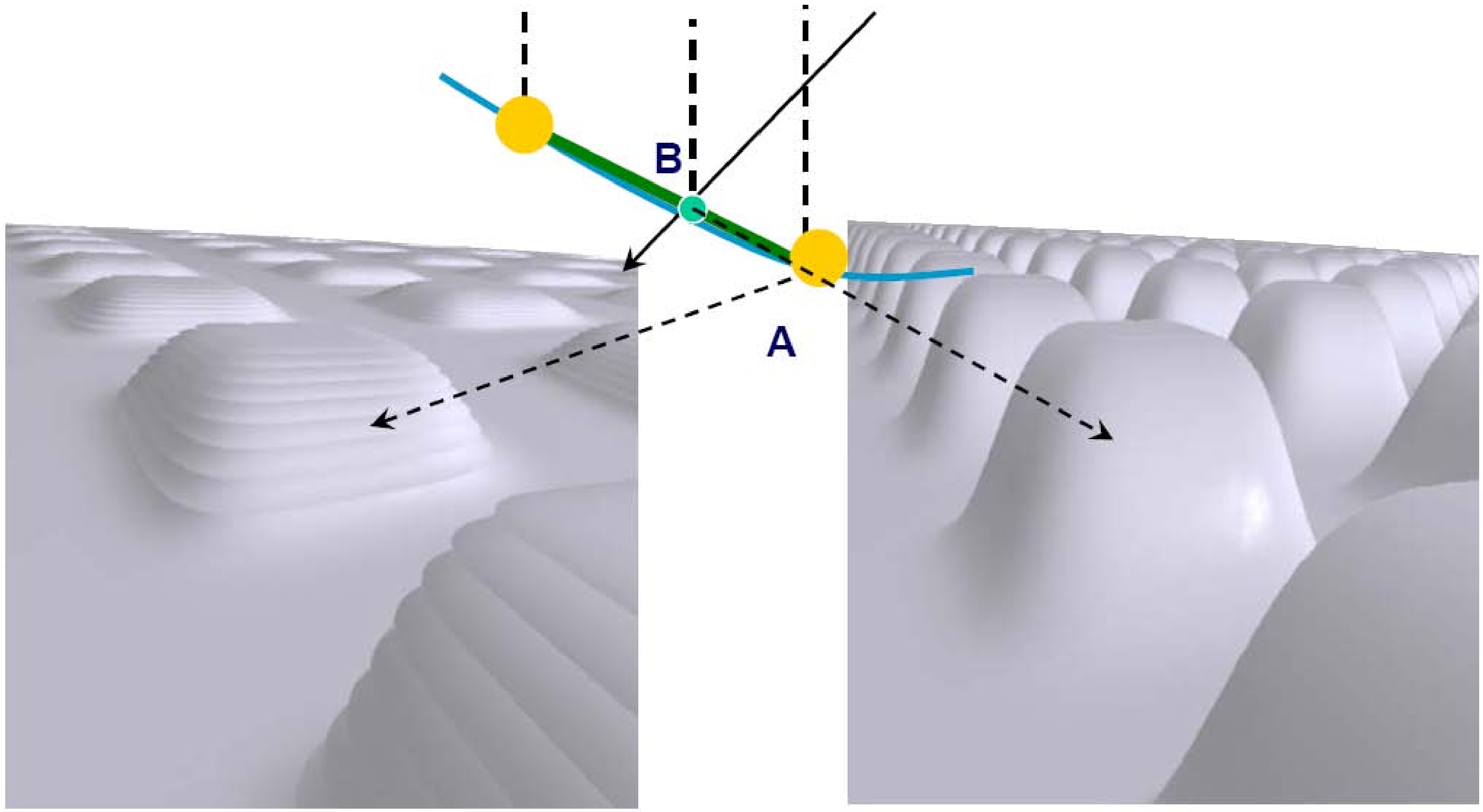
Displacement Mapping by Ray Casting

- walk along ray in small steps, search for intersection
- interpolate height field linearily



Displacement Mapping by Ray Casting

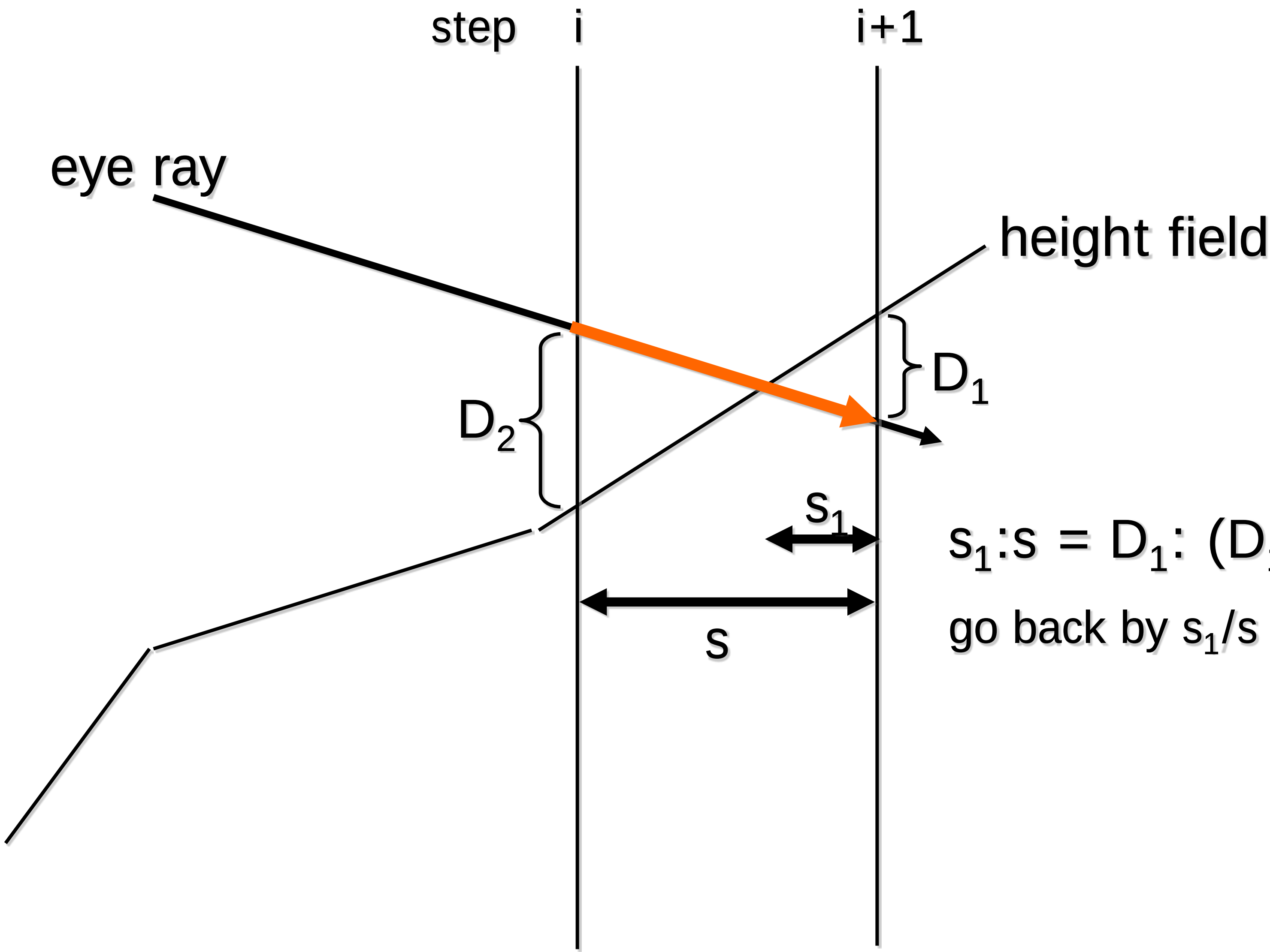
- linear interpolation is important



piecewise constant

piecewise linear

Displacement Mapping by Ray Casting

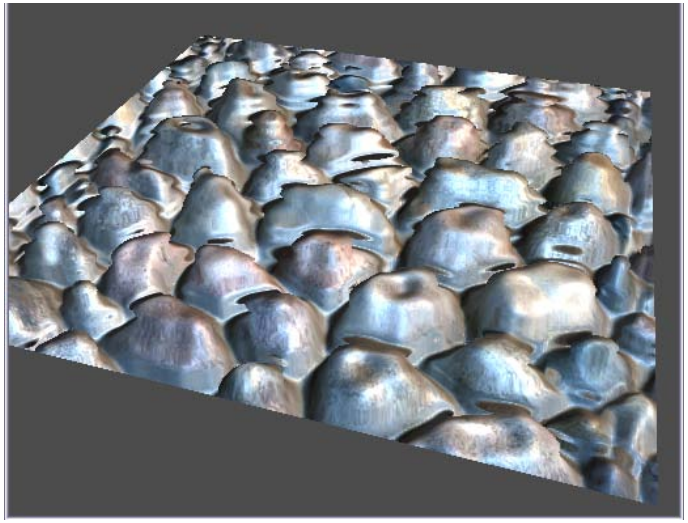


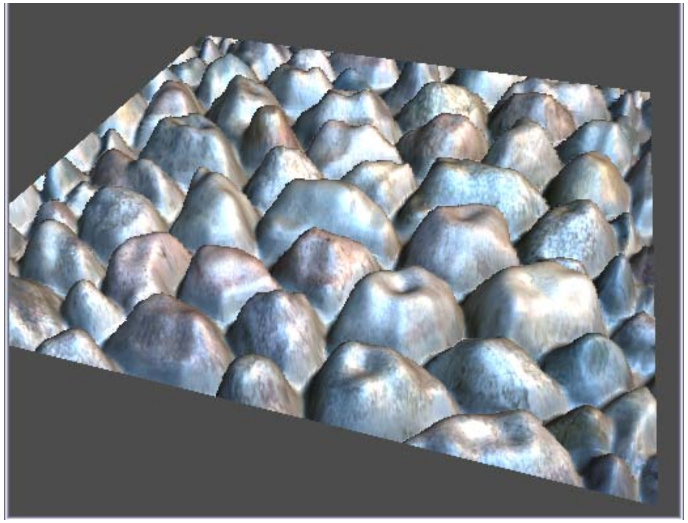
$s_1 : s = D_1 : (D_1 + D_2)$

go back by s_1/s steps

Displacement Mapping by Ray Casting

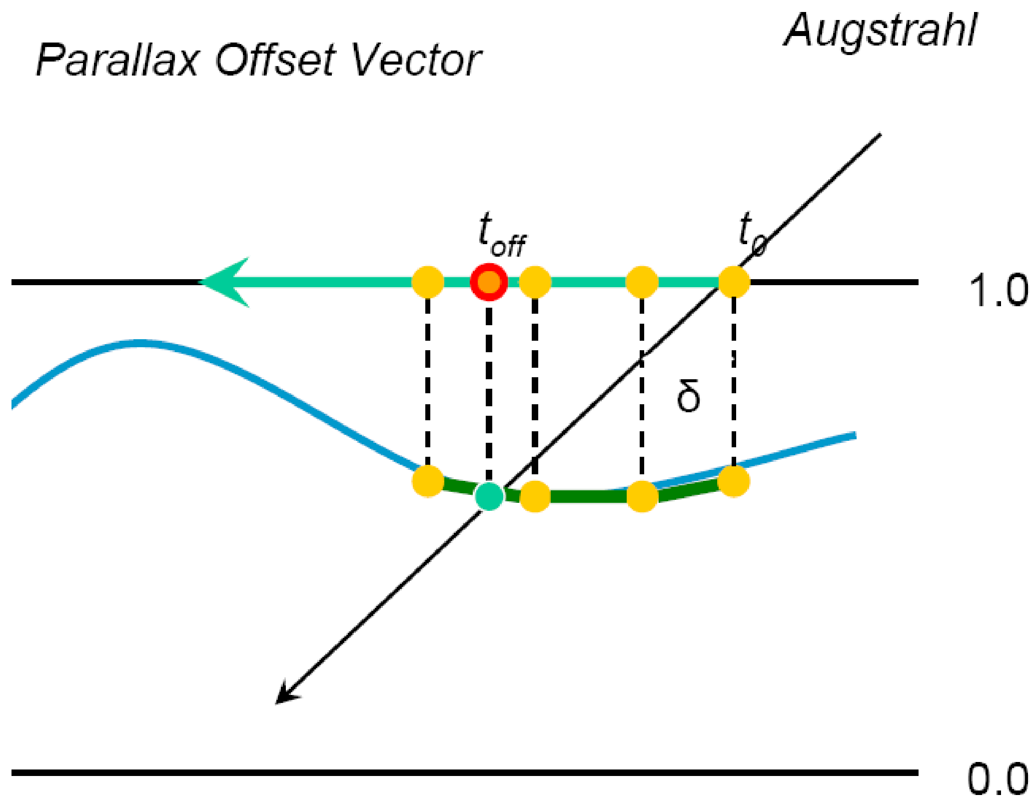
- step size is important





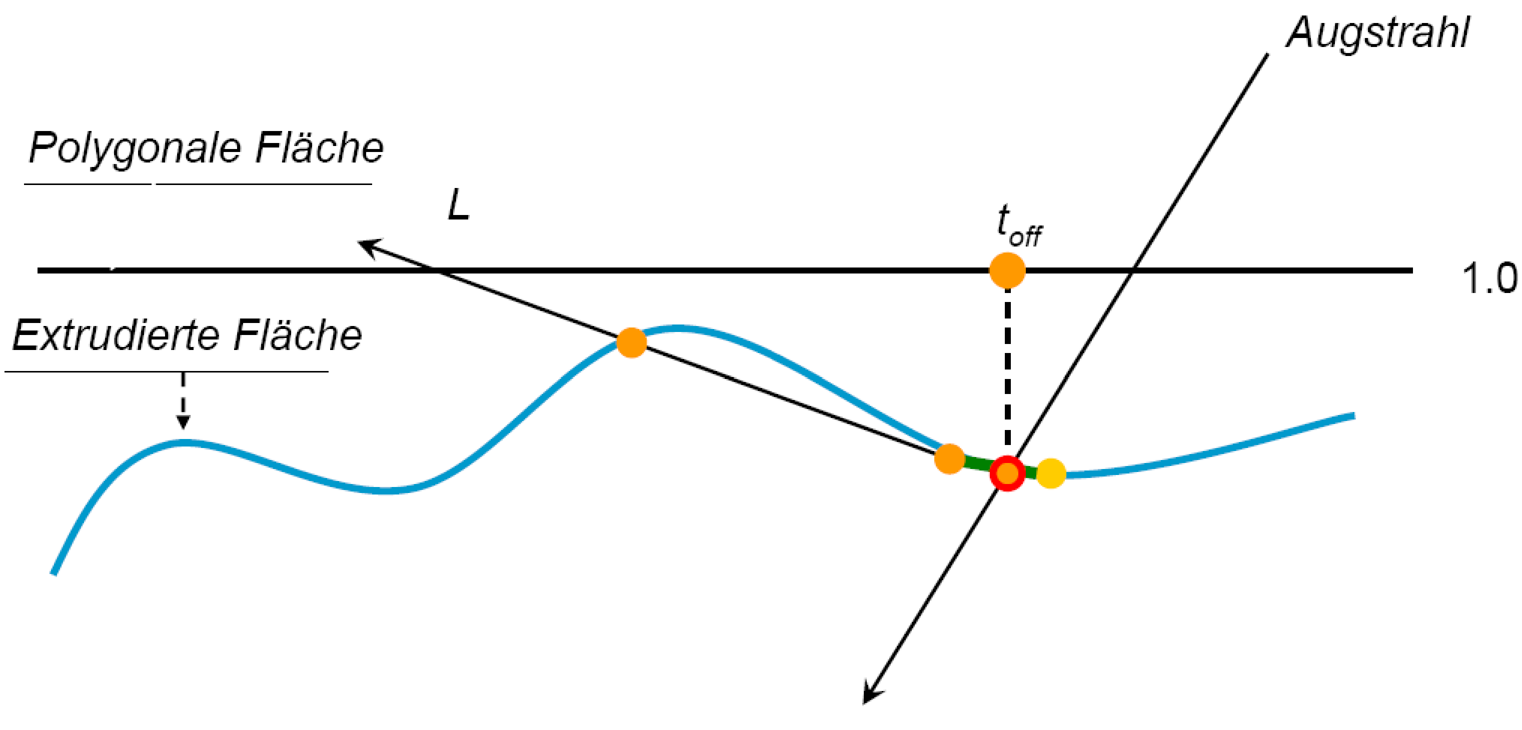
Displacement Mapping by Ray Casting

- adapt step size dynamically
  - select step size such that step size corresponds to pixel size of height map



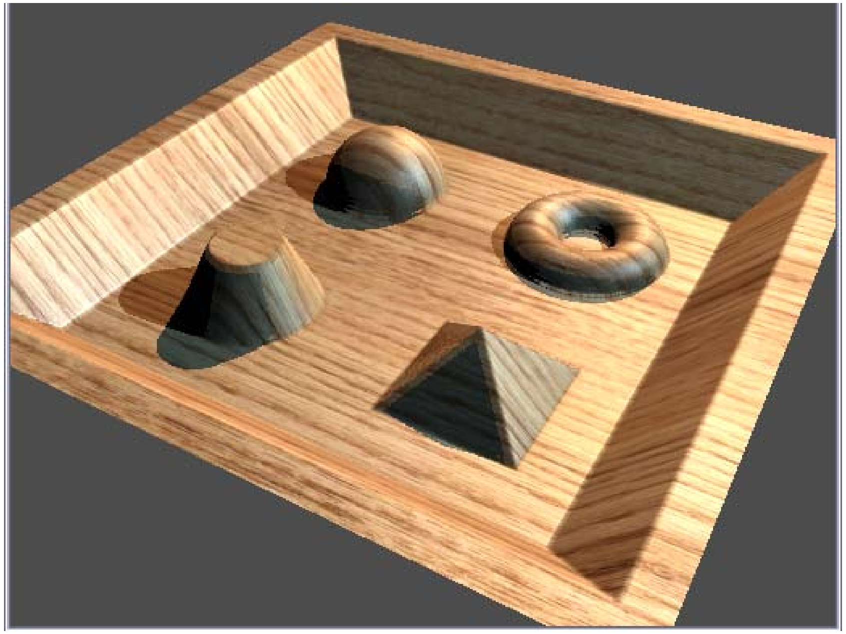
Displacement Mapping by Ray Casting

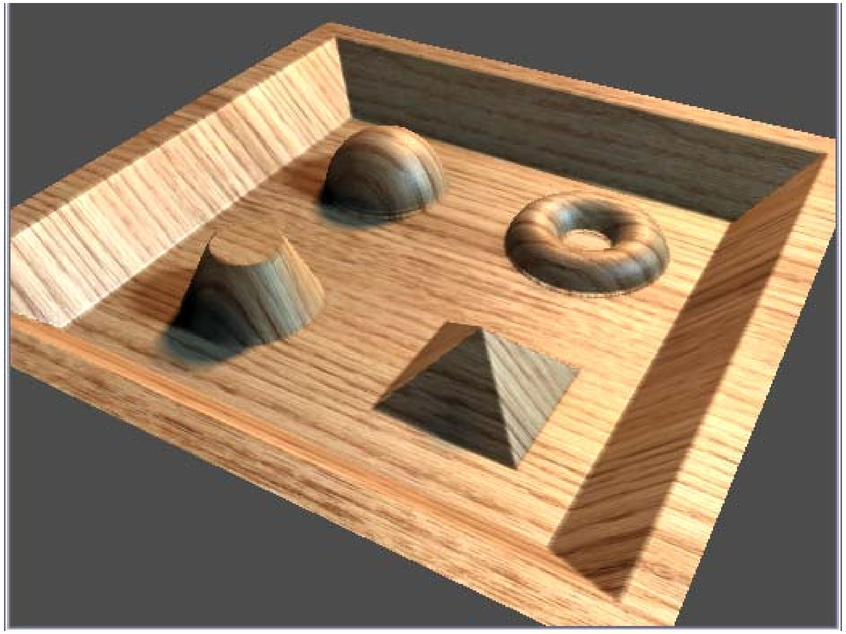
- second ray casting step, now from offsetted hit point back to light



Displacement Mapping by Ray Casting

- trick for soft shadows





Displacement Mapping by Ray Casting

- step along shadow ray towards light source (n steps)
- search largest height difference
  → shadow value
  - max [$(h_i-h_0)/n$*scaling]