Real-Time Rendering (Echtzeitgraphik)

Dr. Michael Wimmer
wimmer@cg.tuwien.ac.at

Shading and Lighting Effects

Overview

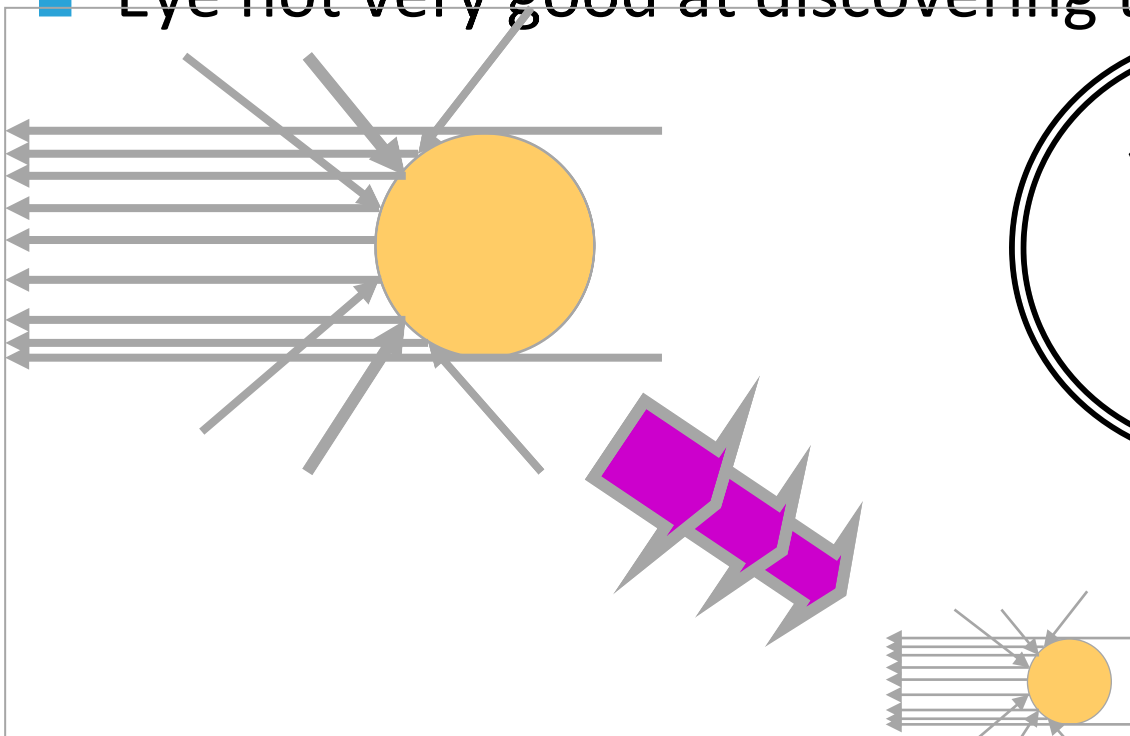
- Environment mapping
  - Cube mapping
  - Sphere mapping
  - Dual-paraboloid mapping
- Reflections, Refractions, Speculars, Diffuse (Irradiance) mapping
- Normal mapping
- Parallax normal mapping
- Advanced Methods

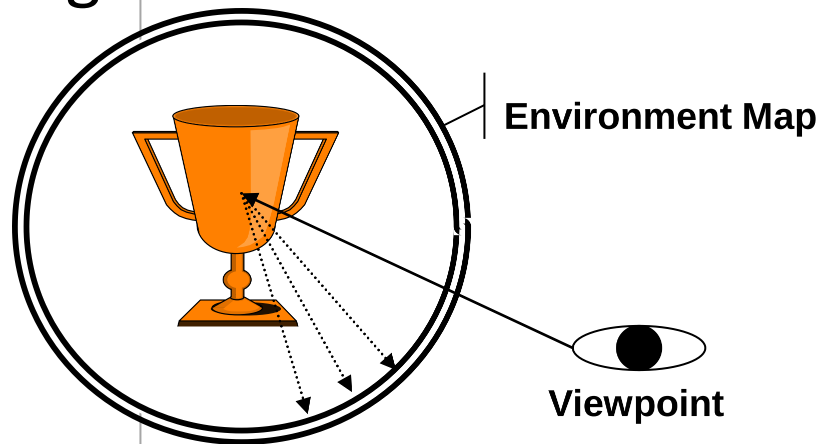
Environment Mapping

- Main idea: fake reflections using simple textures

Environment Mapping

- Assumption: index envmap via orientation
  - Reflection vector or any other similar lookup!
- Ignore (reflection) position! True if:
  - reflecting object shrunk to a single point
  - OR: environment infinitely far away
- Eye not very good at discovering the fake





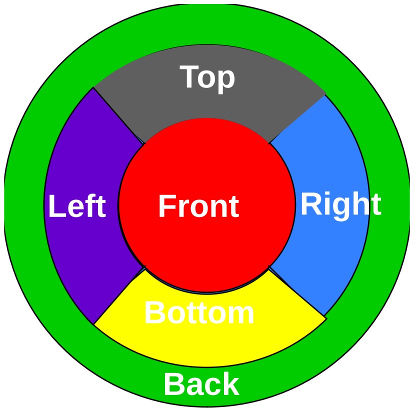
Environment Mapping

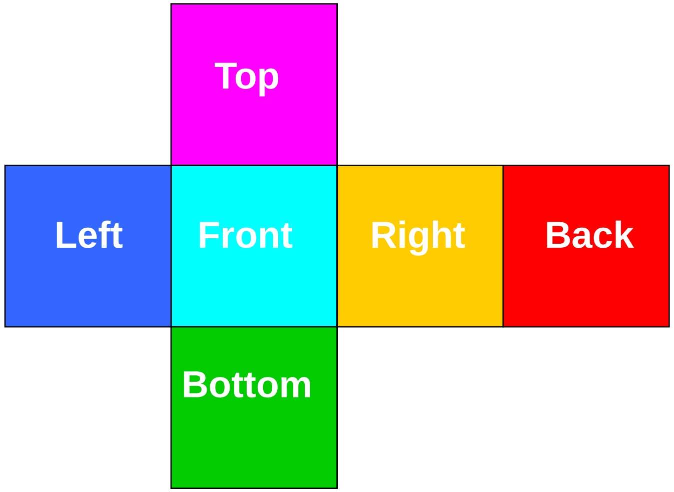
- Can be an “Effect”
  - Usually means: “fake reflection”
- Can be a “Technique” (i.e., GPU feature)
  - Then it means:
    “2D texture indexed by a 3D orientation”
  - Usually the index vector is the reflection vector
  - But can be anything else that’s suitable!

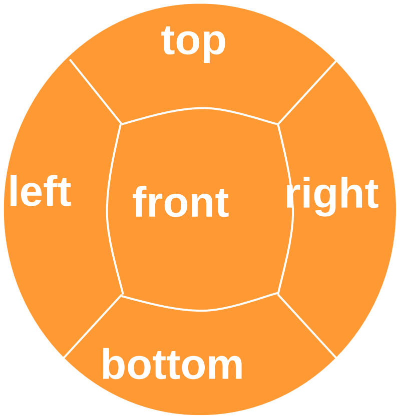
Environment Mapping

- Uses texture coordinate generation, multitexturing, new texture targets…
- Main task:
  Map all 3D orientations to a 2D texture
- Independent of application to reflections

Sphere　　Cube　　Dual paraboloid

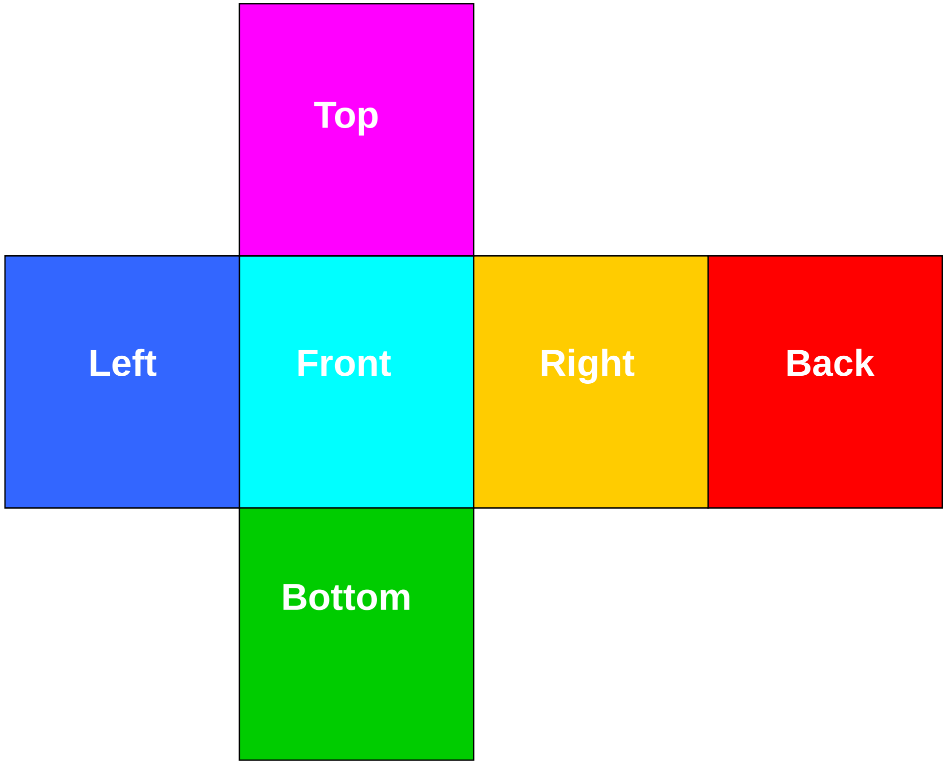


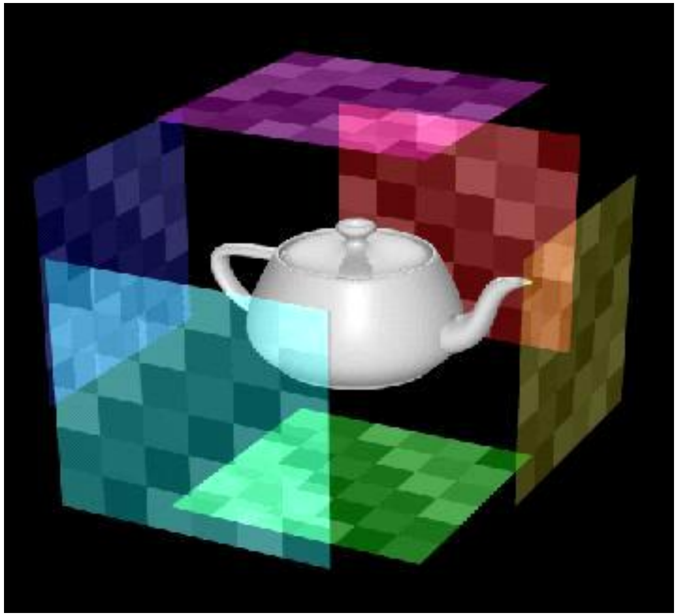




Cube Mapping

- OpenGL texture targets

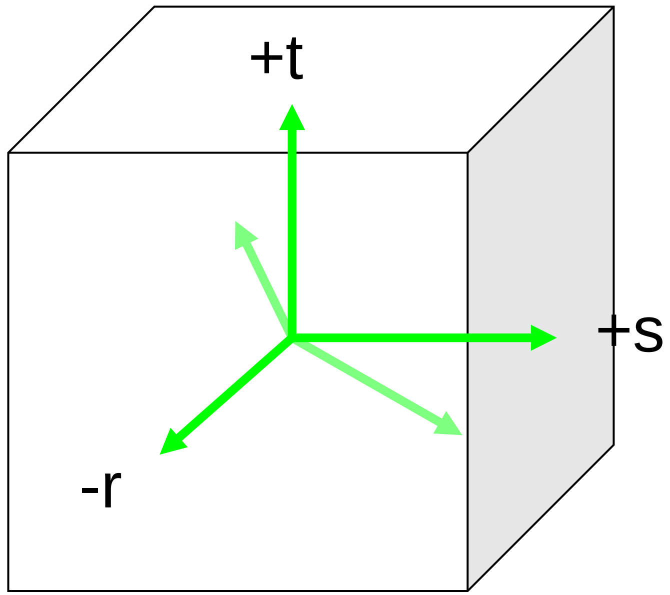




```
glTexImage2D(
GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGB8,
w, h, 0, GL_RGB, GL_UNSIGNED_BYTE, face_px);
```

Cube Mapping

- Cube map accessed via ***vectors*** expressed as 3D texture coordinates (s, t, r)



Cube Mapping

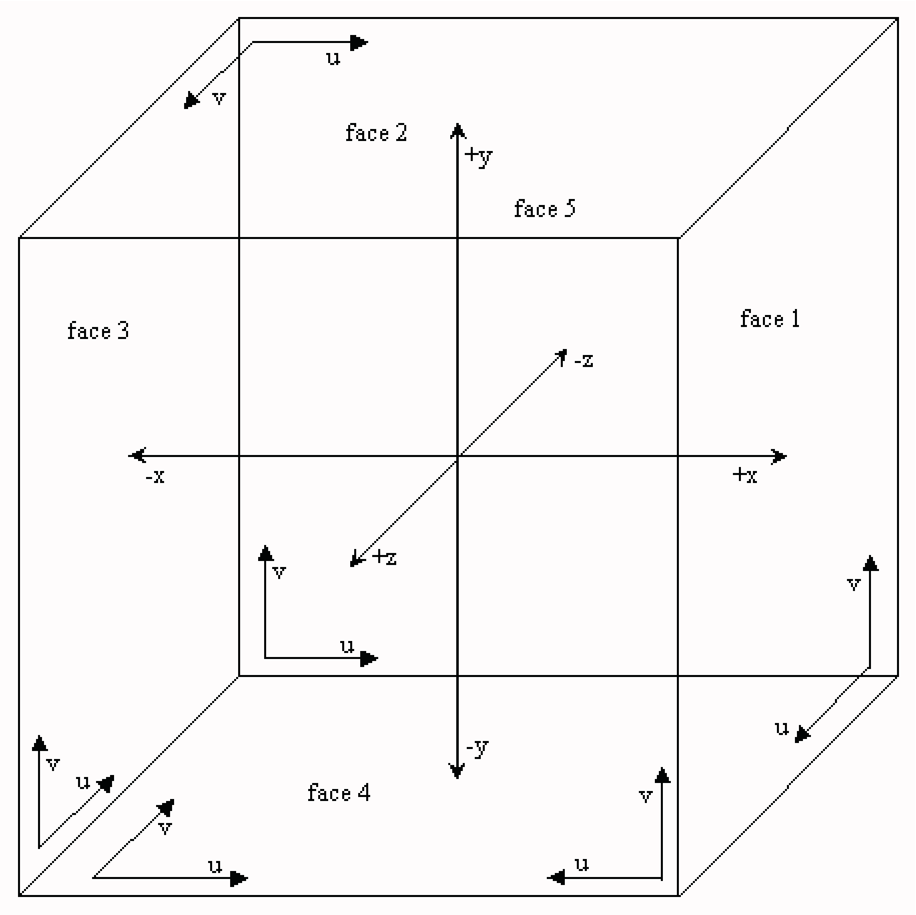
- 3D → 2D projection done by hardware
  - Highest magnitude component selects which cube face to use (e.g., -t)
  - Divide other components by this, e.g.:
    $s' = s / -t$
    $r' = r / -t$
  - (s', r') is in the range $[-1, 1]$
  - remap to $[0,1]$ and select a texel from selected face
- Still need to *generate* useful texture coordinates for reflections

Cube Maps for Env Mapping

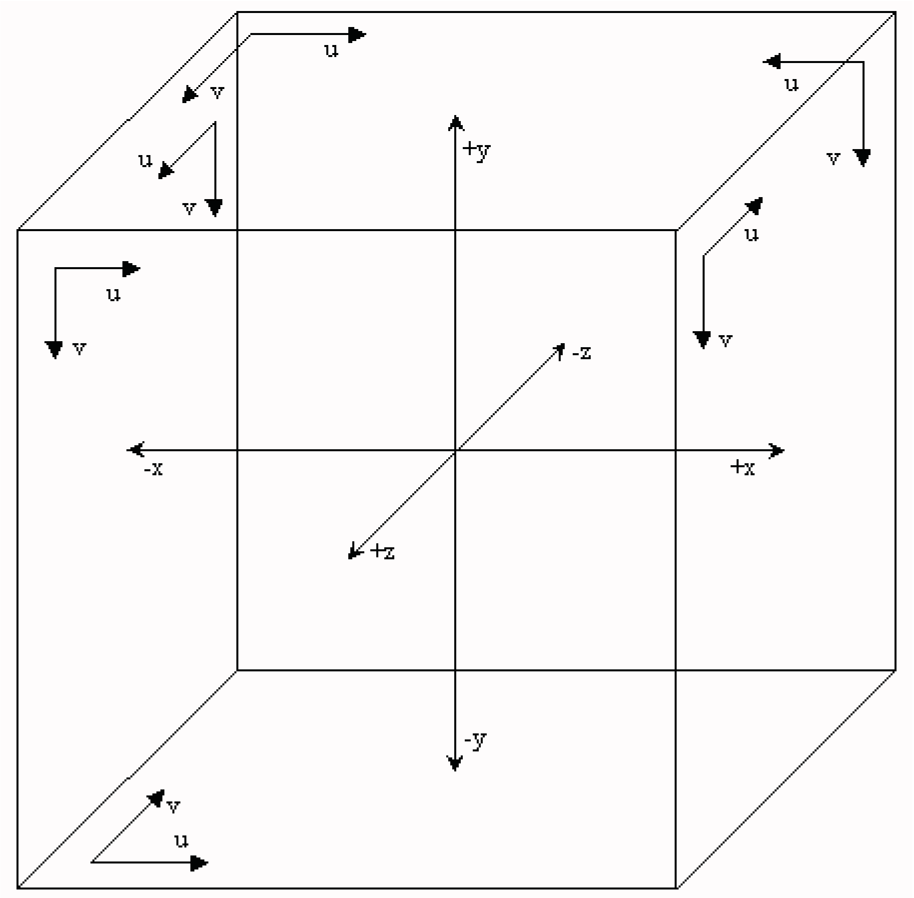
- Generate views of the environment
  - One for each cube face
  - 90° view frustum
  - Use hardware to render directly to a texture
- Use reflection vector to index cube map
  - Generated automatically on hardware:
    ```
    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE,
    GL_REFLECTION_MAP);
    ```

Cube Map Coordinates

- Warning: addressing not intuitive (needs flip)



Watt 3D CG



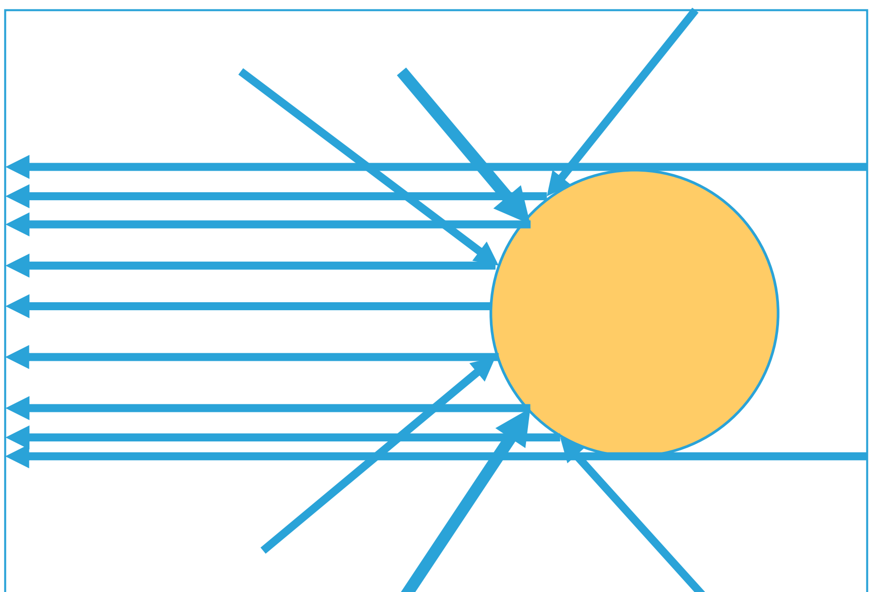
Renderman/OpenGL

Cube Mapping

- Advantages
  - Minimal distortions
  - Creation and map entirely hardware accelerated
  - Can be generated dynamically
- Optimizations for dynamic scenes
  - Need not be updated every frame
  - Low resolution sufficient

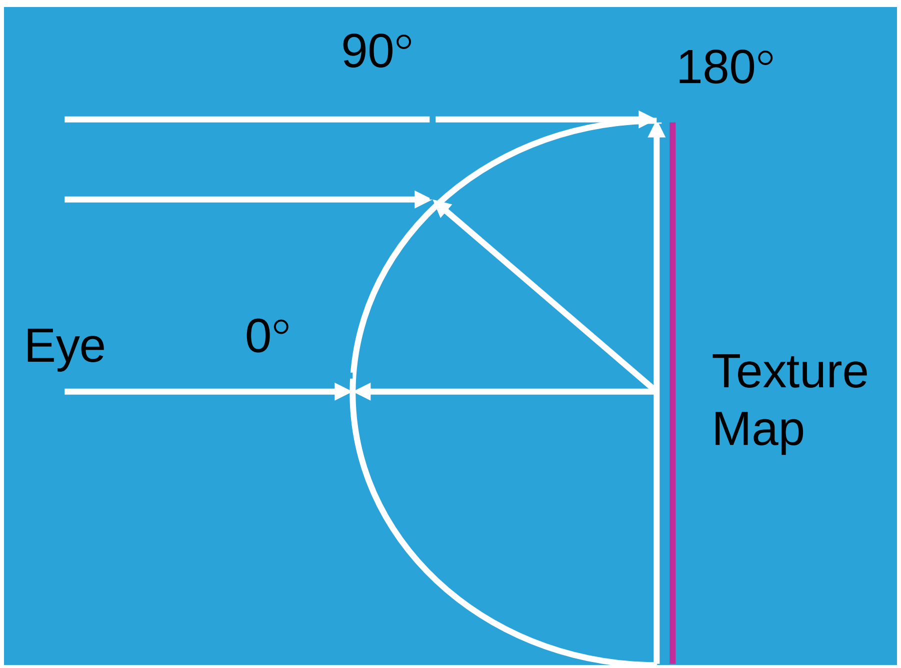
Sphere Mapping

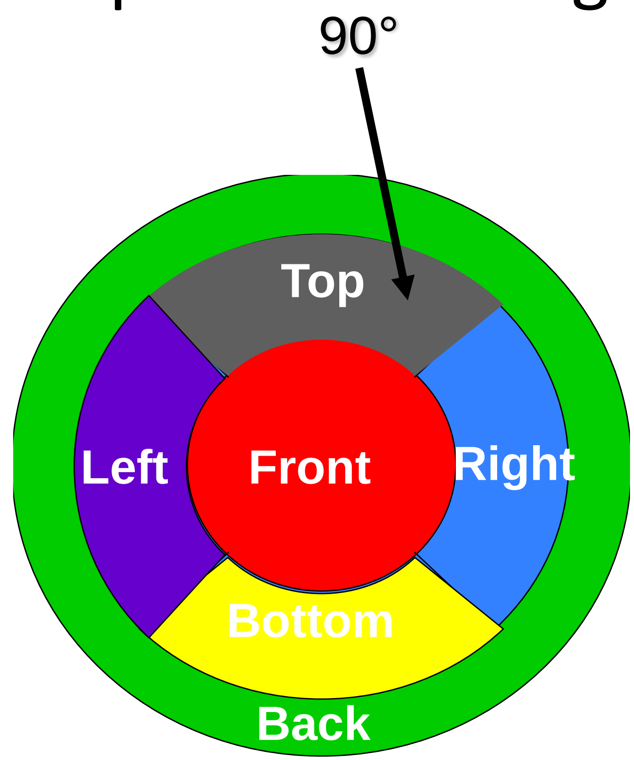
- Earliest available method with OpenGL
  - Only texture mapping required!
- Texture looks like ***orthographic*** reflection from chrome hemisphere
  - Can be photographed like this!



Sphere Mapping

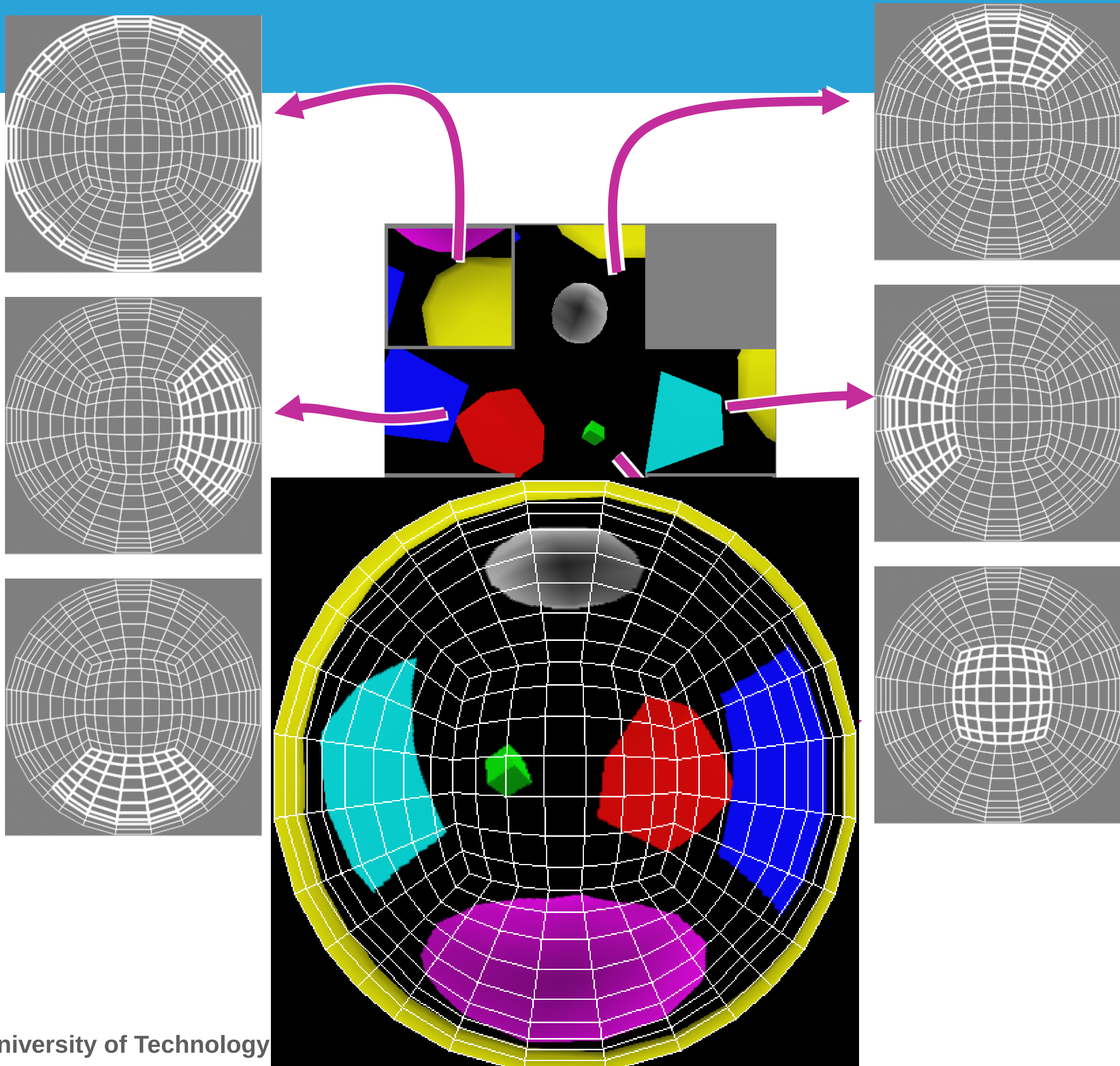
- Maps all reflections to hemisphere
  - Center of map reflects back to eye
  - Singularity: back of sphere maps to outer ring





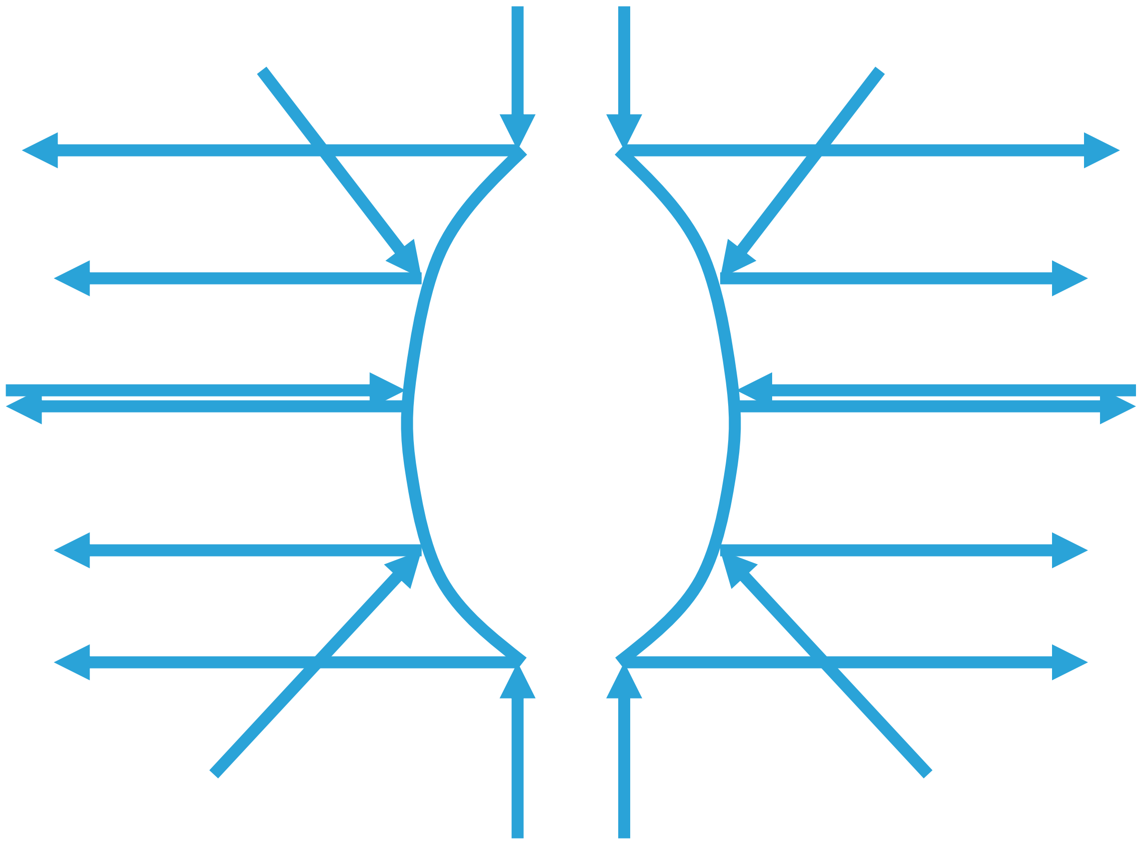
Sphere Mapping

- Texture coordinates generated automatically
  - `glTexGeni(GL_S, GL_TEXTURE_GEN_MODE,`
  - Uses eye-space reflection vector (internally)
- Generation
  - Ray tracing
  - Warping a cube map (possible on the fly)
  - Take a photograph of a metallic sphere!!
- Disadvantages:
  - View dependent → has to be regenerated even for static environments!
  - Distortions



Dual Paraboloid Mapping

- Use orthographic reflection of two parabolic mirrors instead of a sphere



Dual Paraboloid Mapping

- Texture coordinate generation:
  - Generate reflection vector using OpenGL
  - Load texture matrix with $P \cdot M^{-1}$
    - M is inverse view matrix (view independency)
    - P is a projection which accomplishes
      $s = r_x / (1-r_z)$
      $t = r_y / (1-r_z)$
- Texture access across seam:
  - Always apply both maps with multitexture
  - Use alpha to select active map for each pixel

Dual Paraboloid mapping

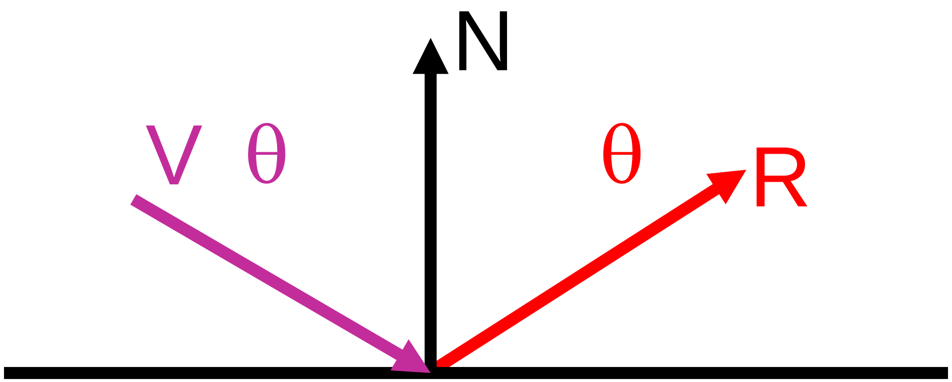
- Advantages
  - View independent
  - Requires only projective texturing
  - Even less distortions than cube mapping
- Disadvantages
  - Can only be generated using ray tracing or warping
    - No direct rendering like cube maps
    - No photographing like sphere maps

Summary Environment Mapping

	Sphere	Cube	Paraboloid
View-	dependent	independent	independent
Generation	warp/ray/ photo	direct rendering/ photo	warp/ray
Hardware required	texture mapping	cube map support	projective texturing, 2 texture units
Distortions	strong	medium	little

Reflective Environment Mapping

- Angle of incidence = angle of reflection



R = V - 2 (N dot V) N

post-modelview view vector

V and N normalized!

- OpenGL uses eye coordinates for R
- Cube map needs reflection vector in world coordinates (where map was created)

→ Load texture matrix with inverse 3x3 view matrix

→ Best done in fragment shader

Example Vertex Program (CG)

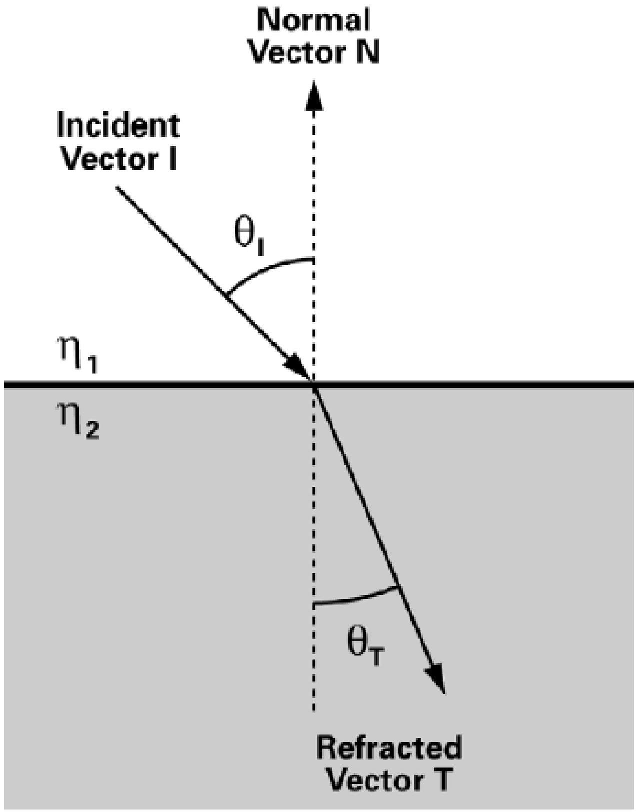
```
void C7E1v_reflection(float4 position   : POSITION,
                      float2 texCoord   : TEXCOORD0,
                      float3 normal     : NORMAL,
                  out float4 oPosition : POSITION,
                  out float2 oTexCoord : TEXCOORD0,
                  out float3 R         : TEXCOORD1,
              uniform float3   eyePositionW,
              uniform float4x4 modelViewProj,
              uniform float4x4 modelToWorld,
             uniform float4x4 modelToWorldInverseTranspose)
{
  oPosition = mul(modelViewProj, position);
  oTexCoord = texCoord;

  // Compute position and normal in world space
  float3 positionW = mul(modelToWorld, position).xyz;
  float3 N = mul((float3x3) modelToWorldInverseTranspose, normal);
  N = normalize(N);

  // Compute the incident and reflected vectors
  float3 I = positionW - eyePositionW;
  R = reflect(I, N);
}
```

Example Fragment Program

```
void C7E2f_reflection(float2 texCoord : TEXCOORD0,
                      float3 R        : TEXCOORD1,

                  out float4 color    : COLOR,

              uniform float reflectivity,
              uniform sampler2D decalMap,
              uniform samplerCUBE environmentMap)
{
  // Fetch reflected environment color
  float4 reflectedColor = texCUBE(environmentMap, R);

  // Fetch the decal base color
  float4 decalColor = tex2D(decalMap, texCoord);

  color = lerp(decalColor, reflectedColor,
   reflectivity);
}
```

Refractive Environment Mapping

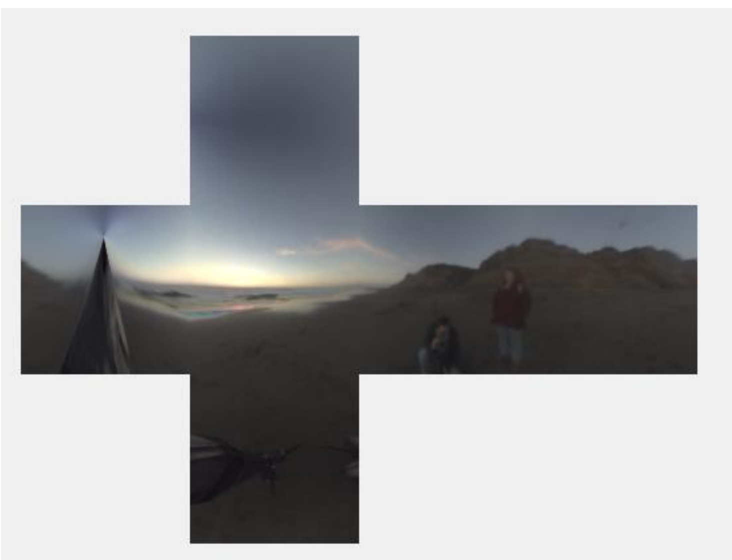
- Use refracted vector for lookup:
  - Snells law: $\eta_1 \sin\theta_I = \eta_2 \sin\theta_T$



Demo

Specular Environment Mapping

- We can prefilter the enviroment map
  - Equals specular integration over the hemisphere
  - Phong lobe (cos^n) as filter kernel
  - R as lookup



Phong filtered

Irradiance Environment Mapping

- Prefilter with cos()
  - Equals diffuse integral over hemisphere
  - N as lookup direction
  - Integration: interpret each pixel of envmap as a light source, sum up!

Diffuse filtered

OGRE Beach Demo

Author: Christian Luksch

http://www.ogre3d.org/wiki/index.php/HDRlib

Environment Mapping Conclusions

- "Cheap" technique
  - Highly effective for static lighting
  - Simple form of image based lighting
    - Expensive operations are replaced by prefiltering
- Advanced variations:
  - Separable BRDFs for complex materials
  - Realtime filtering of environment maps
  - Fresnel term modulations (water, glass)
- Used in virtually every modern computer game

Environment Mapping Toolset

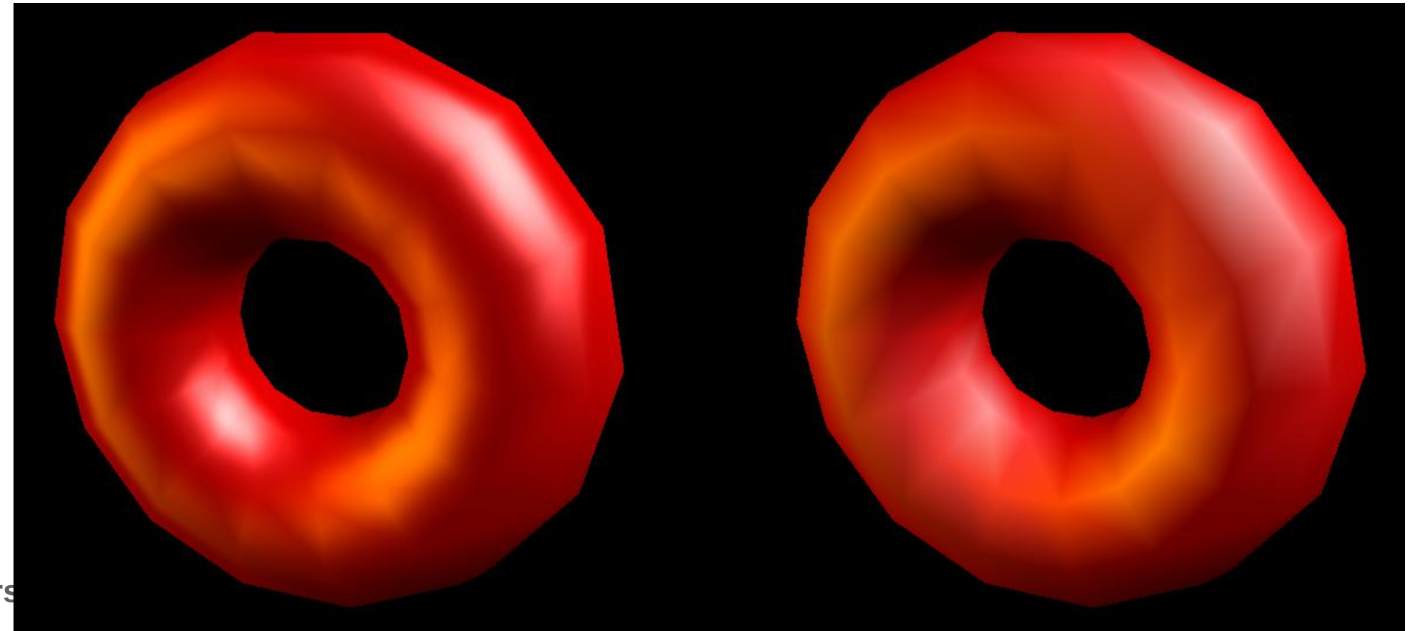
- Environment map creation:
  - AMDs CubeMapGen (free)
    - Assembly
    - Proper filtering
    - Proper MIP map generation
    - Available as library for your engine/dynamic environment maps
  - HDRShop 1.0 (free)
    - Representation conversion
      - Spheremap to Cubemap

Per-Pixel Lighting

- Simulating smooth surfaces by calculating illumination at each pixel
- Example: specular highlights

per-pixel evaluation

linear intensity interpolation



Bump Mapping / Normal Mapping

- Simulating rough surfaces by calculating illumination at each pixel

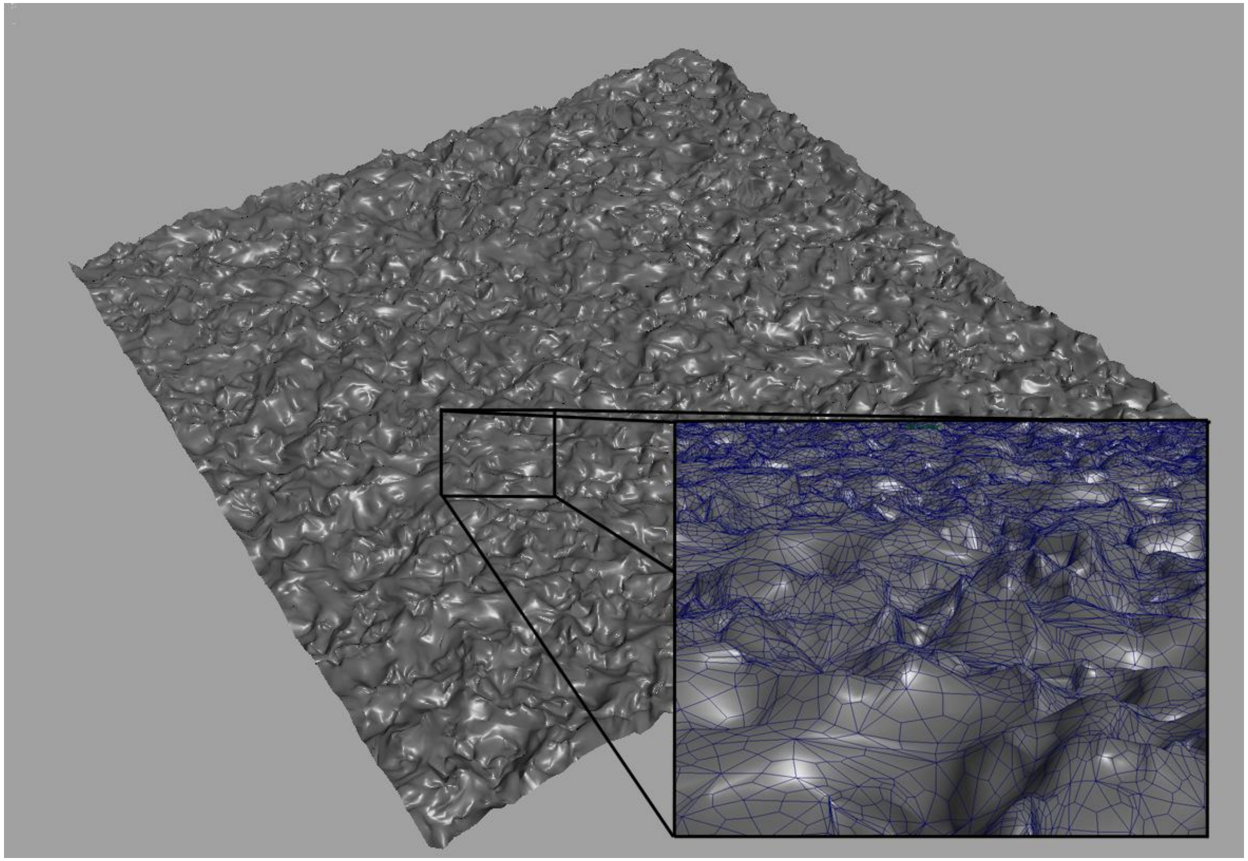
Normal Mapping

- Bump/Normalmapping invented by Blinn 1978.
- Efficient rendering of structured surfaces
- Enormous visual Improvement without additional geometry
- Is a local method (does not know anything about surrounding except lights)

- Heavily used method!
- Realistic AAA games normal map every surface

Normal Mapping

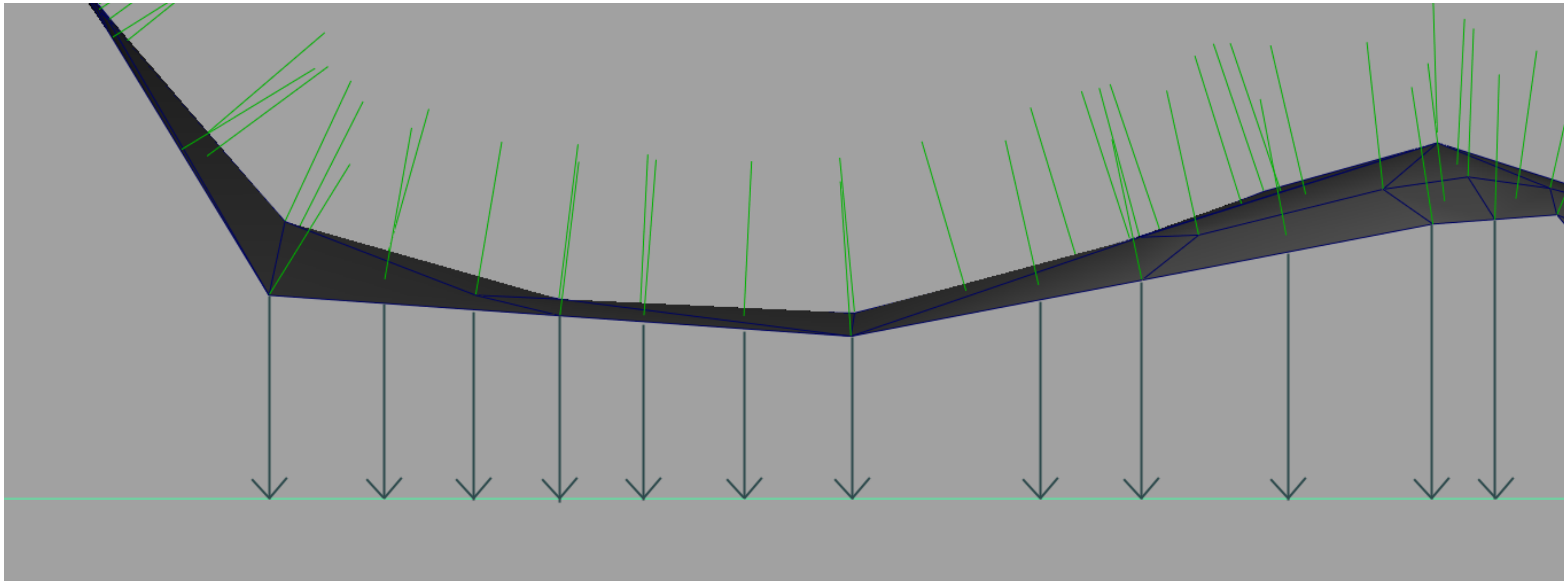
Fine structures require a massive amount of polygons



Too slow for full scene rendering

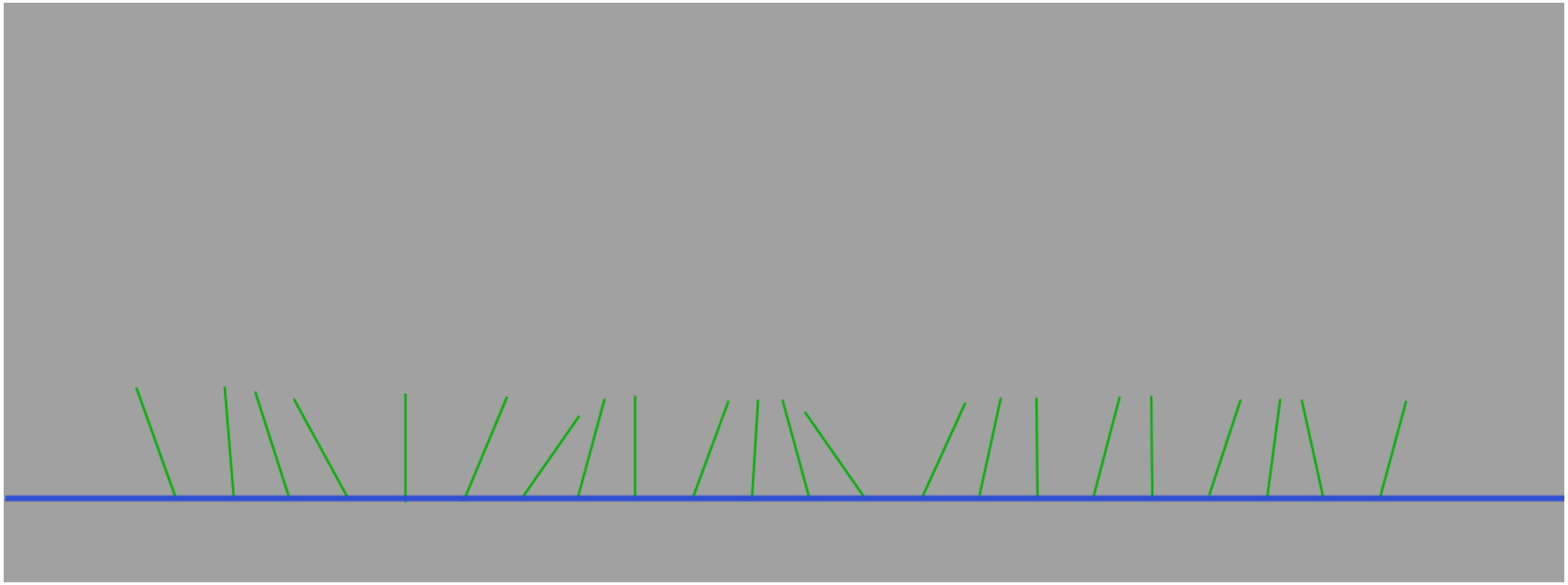
Normal Mapping

- But: perception of illumination is not strongly dependent on position
- Position can be approximated by carrier geometry
  - Idea: transfer normal to carrier geometry



Normal Mapping

- But: perception of illumination is not strongly dependent on position
- Position can be approximated by carrier geometry
  - Idea: transfer normal to carrier geometry



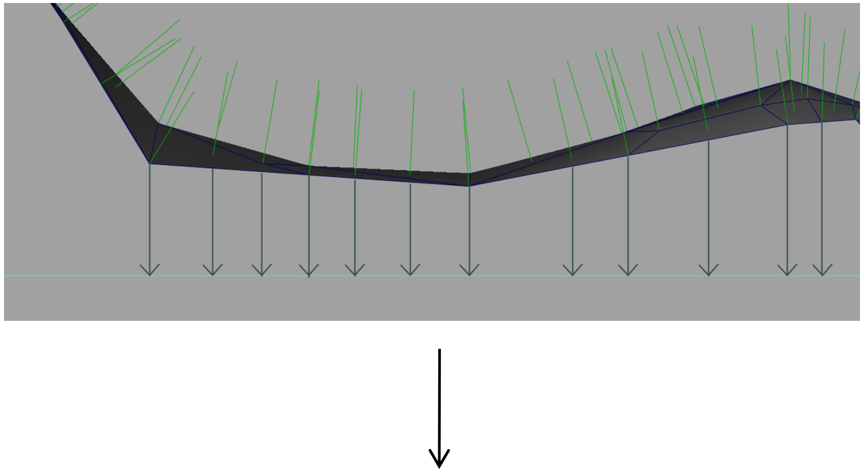
Normal Mapping

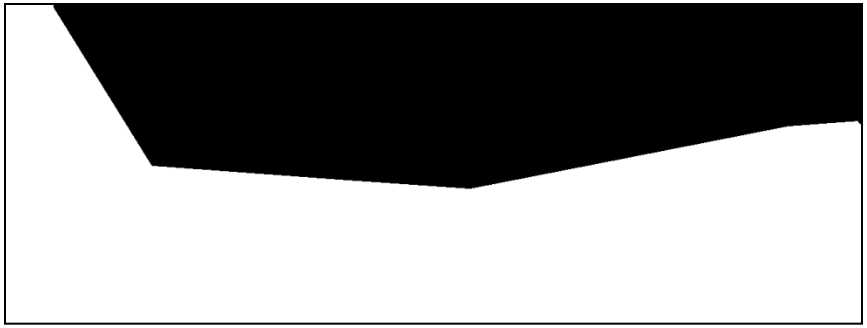
- Result: Texture that contains the normals as vectors

  - Red X
  - Green Y
  - Blue Z
  - Saved as range compressed bitmap ([-1..1] mapped to [0..1])
- Directions instead of polygons!
- Shading evaluations executed with lookup normals instead of interpolated normal

Normal Mapping

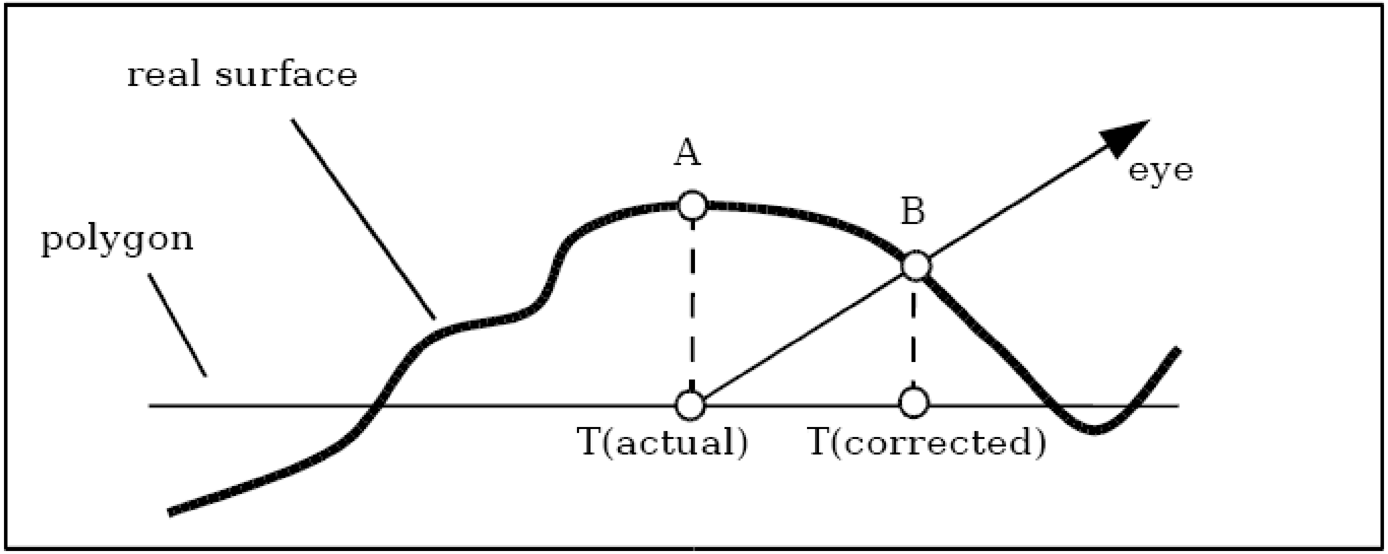
- Additional result is heightfield texture
  - Encodes the distance of original geometry to the carrier geometry





Parallax normal mapping

- Normal mapping does not use the heightfield
  - No parallax effect, surface is still flattened
- Idea: Distort texture lookup according to view vector and heightfield
  - Good approximation of original geometry



Parallax normal mapping

- We want to calculate the offset to lookup color and normals from the corrected position T_n to do shading there

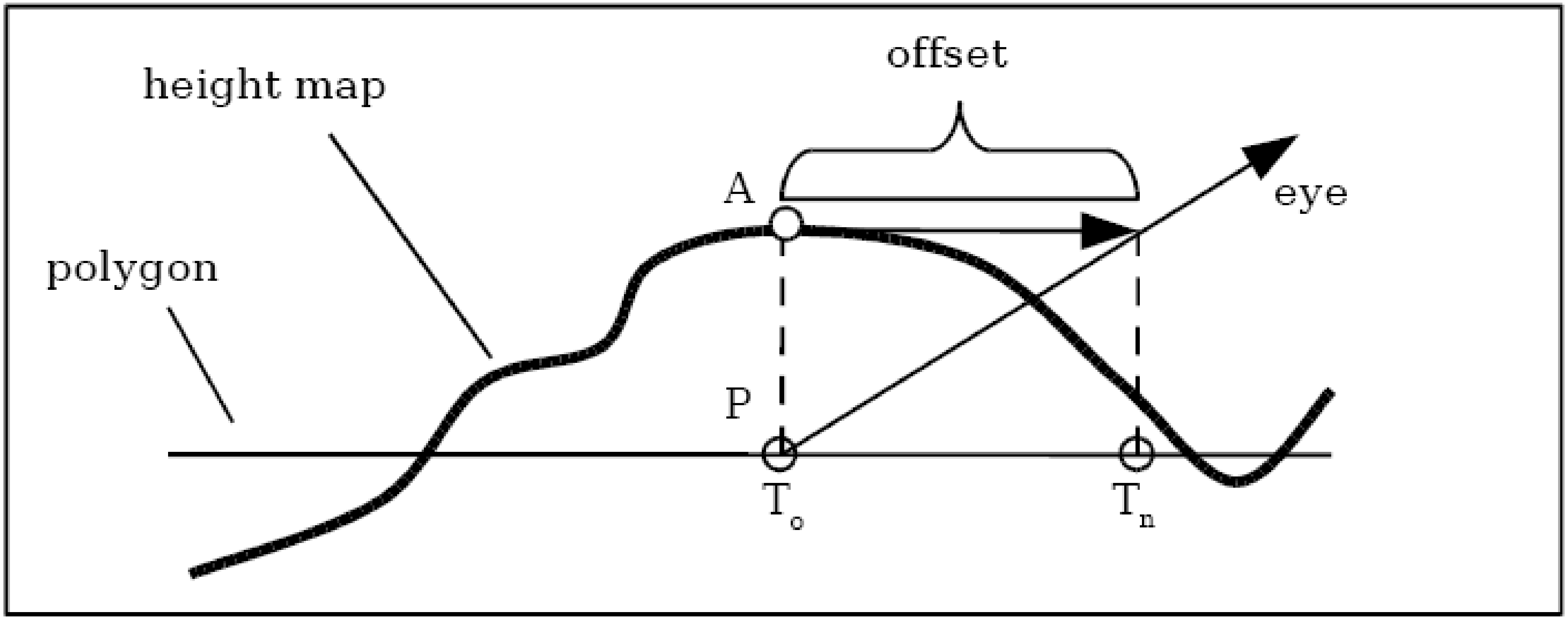
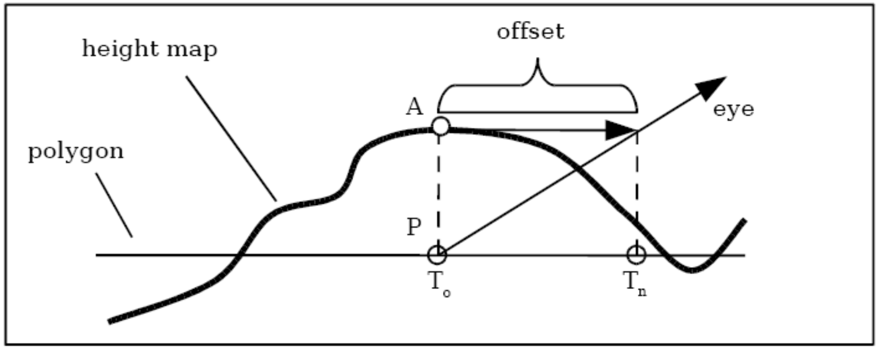


Image by Terry Welsh

Parallax normal mapping

- Rescale heightmap h to appropriate values:
  $h_n = h*s - 0.5s$
  (s = scale = 0.01)



- Assume heightfield is locally constant
  - Lookup heightfield at T_0
- Trace ray from T_0 to eye with eye vector V to height and add offset:
  - $T_n = T_0 + (h_n * V_{x,y}/V_z)$

Offset limited Parallax normal mapping

- Problem: At steep viewing angles, V_z goes to zero
  - Offset values approach infinity
- Solution: we leave out V_z division:

  $T_n = T_0 + (h_n * V_{x,y})$
- Effect: offset is limited

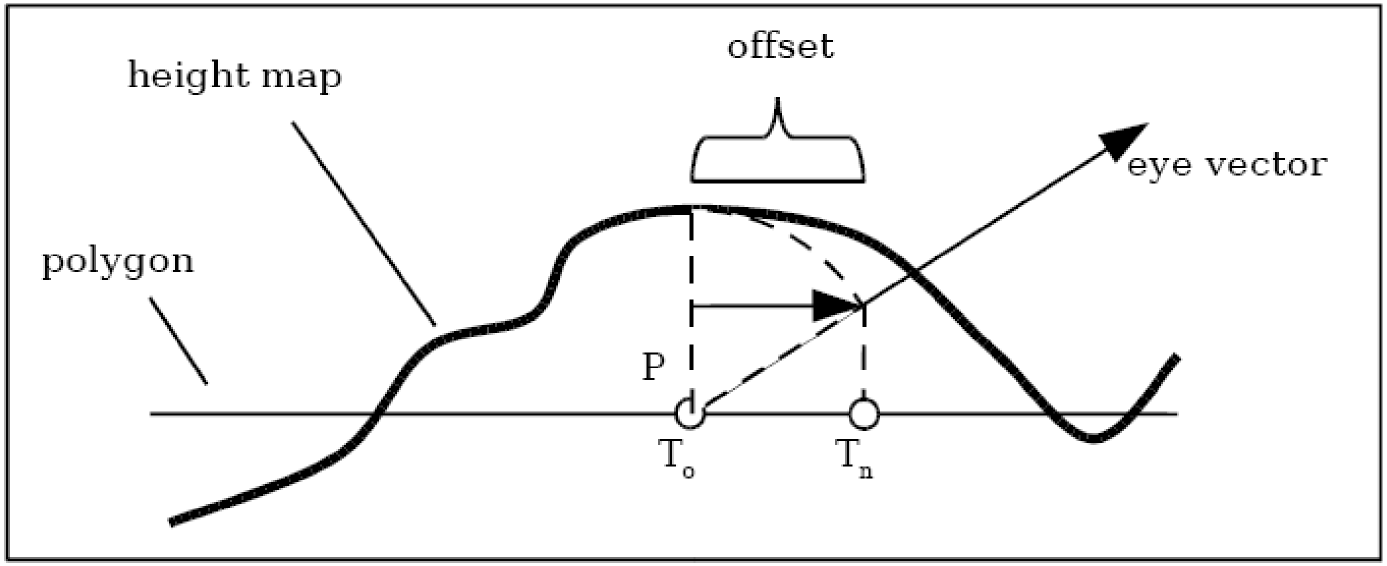
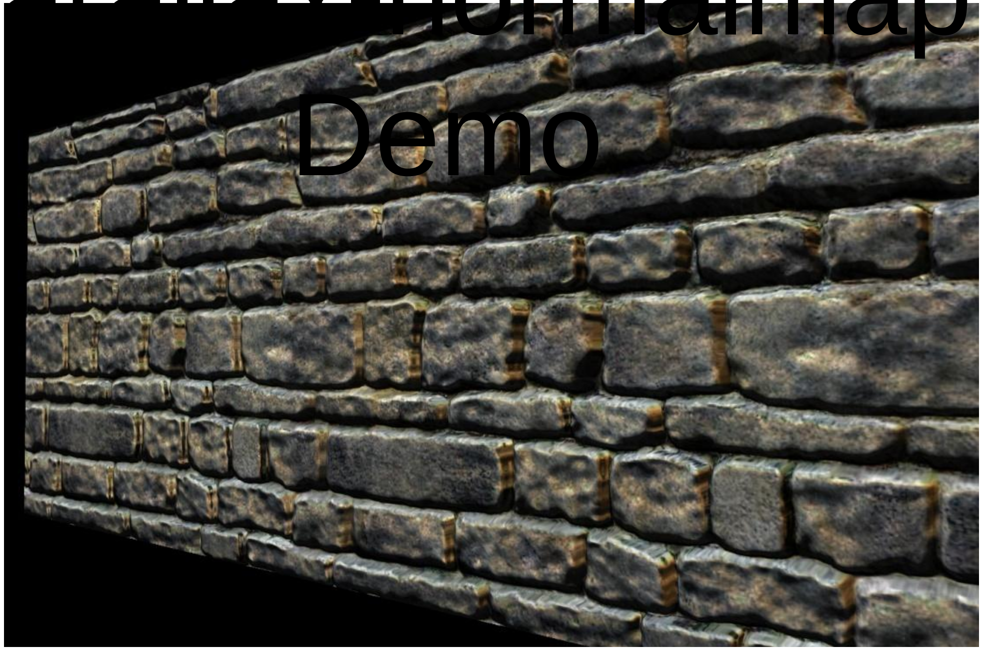


Image by Terry Welsh

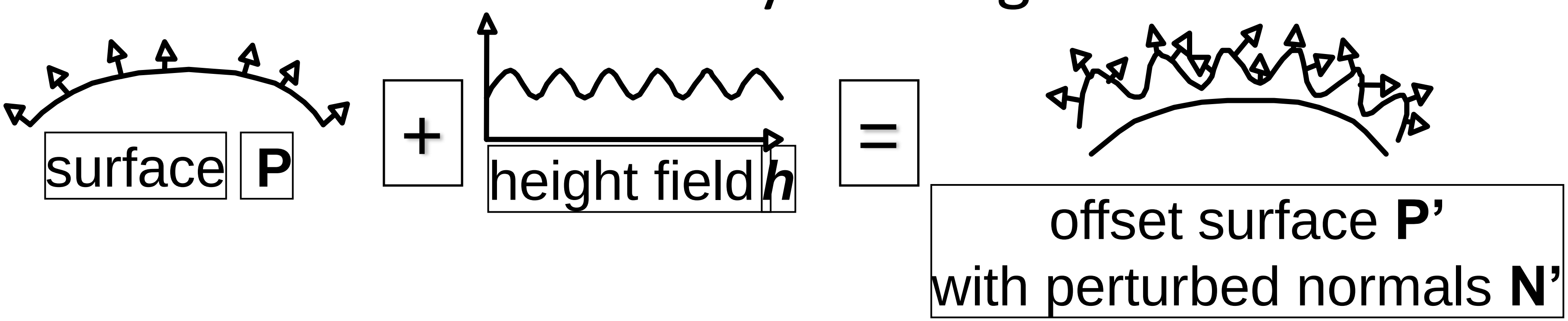
Normalmap
Parallax-normalmap
Demo



Author:Terry Welsh

Bump Map

- Original Bump Mapping idea has theory that is a little more involved!
- Assume a (u, v)-parameterization
  - I.e., points on the surface P = P(u,v)
- Surface P is modified by 2D height field *h*



- P_u, P_v : Partial derivatives:

$$P_u(u,v) = \frac{\partial P}{\partial u}(u,v)$$

  - Easy: differentiate, treat other vars as constant! (or see tangent space)
  - Both derivatives are in tangent plane
- Careful: normal normalization…
  - $N(u,v) = P_u \times P_v$
  - $N_n = N / |N|$

→ Displaced surface:

  - $P'(u,v) = P(u,v) + h(u,v)\, N_n(u,v)$

Mathematics

- Perturbed normal:
  $N'(u,v) \quad = P'_u \times P'_v$
- $P'_u = P_u + h_u N_n + h N_{nu}$
  $\sim P_u + h_u N_n$ (h small)
- $P'_v = P_v + h_v N_n + h N_{nv}$
  $\sim P_v + h_v N_n$

$P' = P + h N_n$

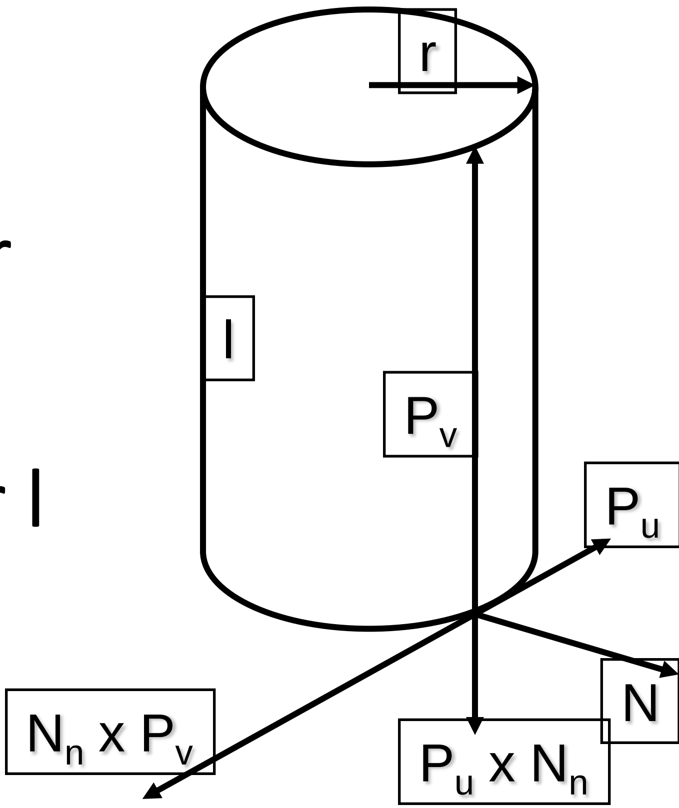
$\rightarrow N' \quad = N + h_u (N_n \times P_v) + h_v (P_u \times N_n)$

$= N + D$ "offset vector"

(D is in tangent plane)

Goal: $N' = N + h_u (N_n \times P_v) + h_v (P_u \times N_n)$

- $P(u,v) = (r \cos u, r \sin u, l\, v)$,
  $u = 0..\ 2\,Pi$, $v = 0..1$
- $P_u = (-\, r \sin u, r \cos u, 0)$, $|P_u| = r$
- $P_v = (0, 0, l)$, $|P_v| = l$
- $N = (r\, l \cos u, r\, l \sin u, 0)$, $|N| = r\, l$
- $N_n = (\cos u, \sin u, 0)$
- $N_n \times P_v = l (\sin u, -\cos u, 0)$
- $P_u \times N_n = (0, 0, -r)$



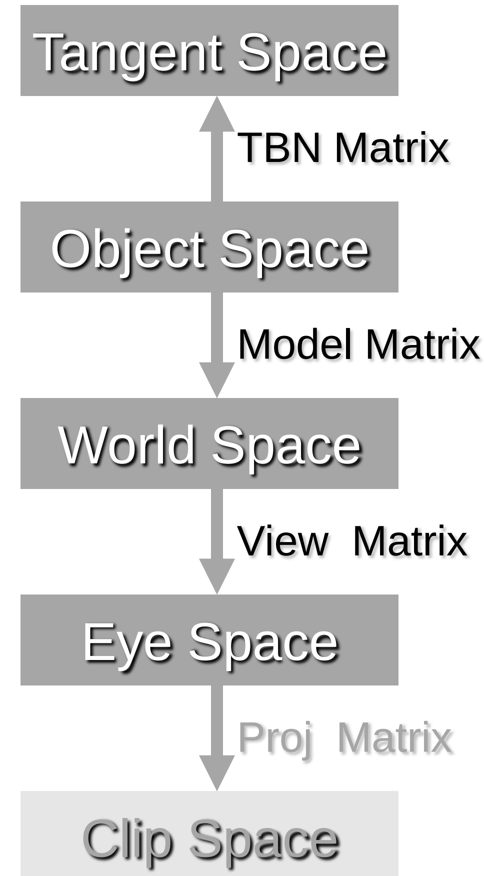
Bump Mapping Issues

- Dependence on surface parameterization
  - $D = f(P_u, P_v)$
  - Map tied to this surface → don't want this!
- What to calculate where?
  - Preproces, per object, per vertex, per fragment
- Which coordinate system to choose?

Coordinate Systems

Problem: where to calculate lighting?

- Object coordinates
  - Native space for normals (N)
- World coordinates
  - Native space for light vector (L), env-maps
  - Not explicit in OpenGL!
- Eye Coordinates
  - Native space for view vector (V)
- Tangent Space
  - Native space for normal maps



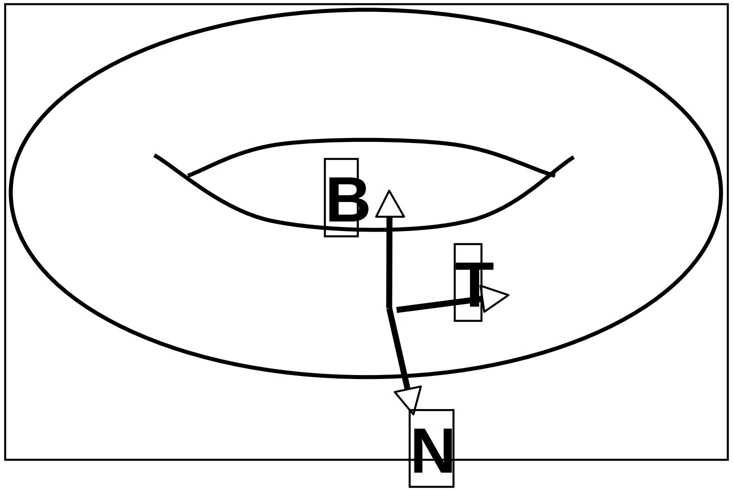
Basic Algorithm (Eye Space)

- For scene (assume infinite L and V)
  - Transform L and V to eye space and normalize
  - Compute normalized H (for specular)
- For each vertex
  - Transform N_n, P_u and P_v to eye space
  - Calculate $B1 = N_n \times P_v$, $B2 = P_u \times N_n$, $N = P_u \times P_v$
- For each fragment
  - Interpolate B1, B2, N
  - Fetch $(h_u, h_v) = \text{texture}(s, t)$
  - Compute $N' = N + h_u B1 + h_v B2$
  - Normalize N'
  - Using N' in standard Phong equation

Tangent Space

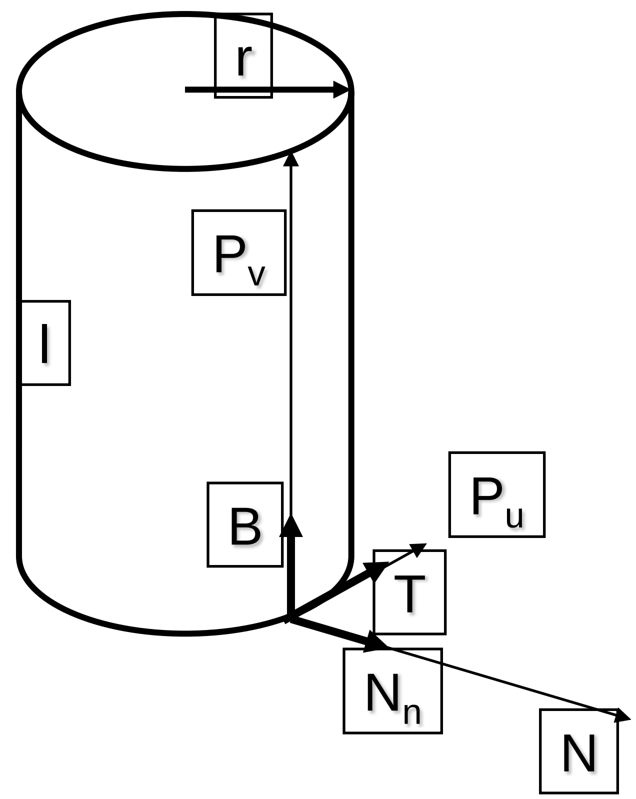
- Concept from differential geometry
- Set of all tangents on a surface
- Orthonormal coordinate system (frame) for each point on the surface:

$$N_n(u,v) = P_u \times P_v \,/\, |P_u \times P_v|$$
$$T = P_u \,/\, |P_u|$$
$$B = N_n \times T$$



- A natural space for normal maps
  - Vertex normal N = (0,0,1) in this space!

- Cylinder Tangent Space:
- $N_n(u,v) = P_u \times P_v \,/\, |P_u \times P_v|$
  $T = P_u \,/\, |P_u|$
  $B = N_n \times T$
- Tangent space matrix:
  TBN column vectors



Fast Algorithm (Tangent Space)

- "Normal Mapping"
- For each vertex
  - Transform light direction L and eye vector V to tangent space and normalize
  - Compute normalized Half vector H
- For each fragment
  - Interpolate L and H
  - Renormalize L and H
  - Fetch N' = texture(s, t) (Normal Map)
  - Use N' in shading

Square Patch Assumption

- $B = P_v / |P_v|$
  - Decouples bump map from surface!
- Recall formula: $N' = N + h_u (N_n \times P_v) + h_v (P_u \times N_n)$
- Convert to tangent space:

$N_n \times P_v = - \mathbf{T} \, |P_v|$

$P_u \times N_n = - \mathbf{B} \, |P_u|$

$|N| = |Pu \times Pv| = |Pu| \; |Pv| \sin \alpha$

$N' = \mathbf{N} - h_u \mathbf{T} \, |P_v| - h_v \mathbf{B} \, |P_u|$ divide by $|Pu| \; |Pv|$

B

T

N

$\rightarrow N' \sim \mathbf{N_n} \sin \alpha - h_u / |P_u| \, \mathbf{T} - h_v / |P_v| \, \mathbf{B}$

Square Patch Assumption

- $N' \sim \mathbf{N_n} \sin\alpha - h_u / |P_u|\, \mathbf{T} - h_v / |P_v|\, \mathbf{B}$
  - Square patch $\rightarrow \sin\alpha = 1$
  - $|P_u|$ and $|P_v|$ assumed constant over patch
- $N' \sim \mathbf{N_n} - (h_u / k)\, \mathbf{T} - (h_v / k)\, \mathbf{B} = \mathbf{N_n} + \mathbf{D}$

Offset Bump Maps

- $N' \sim \mathbf{N_n} - (h_u / k)\,\mathbf{T} - (h_v / k)\,\mathbf{B} = \mathbf{N_n} + \mathbf{D}$
- In tangent space (TBN):
  - $\mathbf{N_n} = (0, 0, 1)$, $\mathbf{D} = (-\,h_u / k, -\,h_v / k, 0)$
- "Scale" of bumps: k
  - Apply map to any surface with same scale
- Alternative: $\mathbf{D} = (-\,h_u, -\,h_v, 0)$
  - Apply k at runtime
- h_u, h_v : calculated by finite differencing from height map

Normal Maps

- Also: normal perturbation maps
- $\mathbf{N'} \sim \mathbf{N_n} - (h_u / k)\,\mathbf{T} - (h_v / k)\,\mathbf{B} = \mathbf{R}\ \mathbf{N_n}$
- R: rotation matrix
- In tangent space (TBN):
  - $\mathbf{N_n} = (0, 0, 1) \rightarrow$ **N'** third row of **R**
  - $N' = \mathrm{Normalize}(-\,h_u / k, -\,h_v / k, 1)$
- "Scale" of bumps: k
- Comparison to offset maps:
  - Need 3 components
  - Better use of precision (normalized vector)

Creating Tangent Space

- Trivial for analytically defined surfaces
  - Calculate P_u, P_v at vertices
- Use ***texture space*** for polygonal meshes
  - Induce from given texture coordinates per triangle
  - $P(u, v) = \boldsymbol{a}\, u + \boldsymbol{b}\, v + \boldsymbol{c} = P_u\, u + P_v\, v + \boldsymbol{c}$!
  - 9 unknowns, 9 equations (x,y,z for each vertex)!
- Transformation from object space to tangent space

$$\begin{bmatrix} L_{tx} & L_{ty} & L_{tz} \end{bmatrix} = \begin{bmatrix} L_{ox} & L_{oy} & L_{oz} \end{bmatrix} \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

Creating Tangent Space - Math

- $P(s, t) = \boldsymbol{a}\, s + \boldsymbol{b}\, t + \boldsymbol{c}$, linear transform!

→ $P_u(s,t) = \boldsymbol{a}$, $P_v(s,t) = \boldsymbol{b}$

- Texture space:
  - $u_1 = (s_1,t_1)-(s_0,t_0)$, $u_2 = (s_2,t_2)-(s_0,t_0)$
- Local space:
  - $v_1 = P_1-P_0$, $v_2 = P_2-P_0$
- $[P_u\ P_v]\ u_1 = v_1$, $[P_u\ P_v]\ u_2 = v_2$
- Matrix notation:
  - $[P_u\ P_v]\ [u_1\ u_2] = [v_1\ v_2]$

Creating Tangent Space - Math

- $[P_u \ P_v] \ [u_1 \ u_2] = [v_1 \ v_2]$

$\rightarrow [P_u \ P_v] = [v_1 \ v_2] \ [u_1 \ u_2]^{-1}$

- $[u_1 \ u_2]^{-1} = 1/| \ u_1 \ u_2 \ | \ \begin{bmatrix} u_{2y} & -u_{2x} \\ -u_{1y} & u_{1x} \end{bmatrix}$

- Result: very simple formula!
- Finally: calculate tangent frame (for triangle):
  - $T = P_u \ / \ |P_u|$
    $B = N_n \times T$

Creating Tangent Space

- Example for key-framed skinned model
  - Note: average tangent space between adjacent triangles (like normal calculation)

bump-skin height field

decal skin (unlit!)

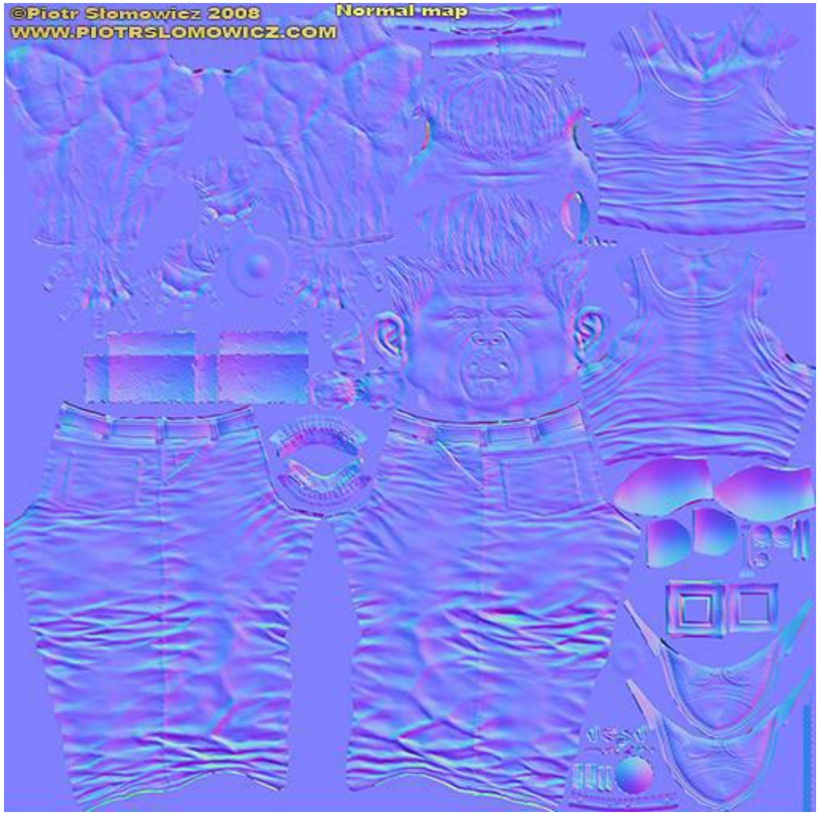
Quake 2 Example

Final result!

Note: Gloss map defines where to apply specular

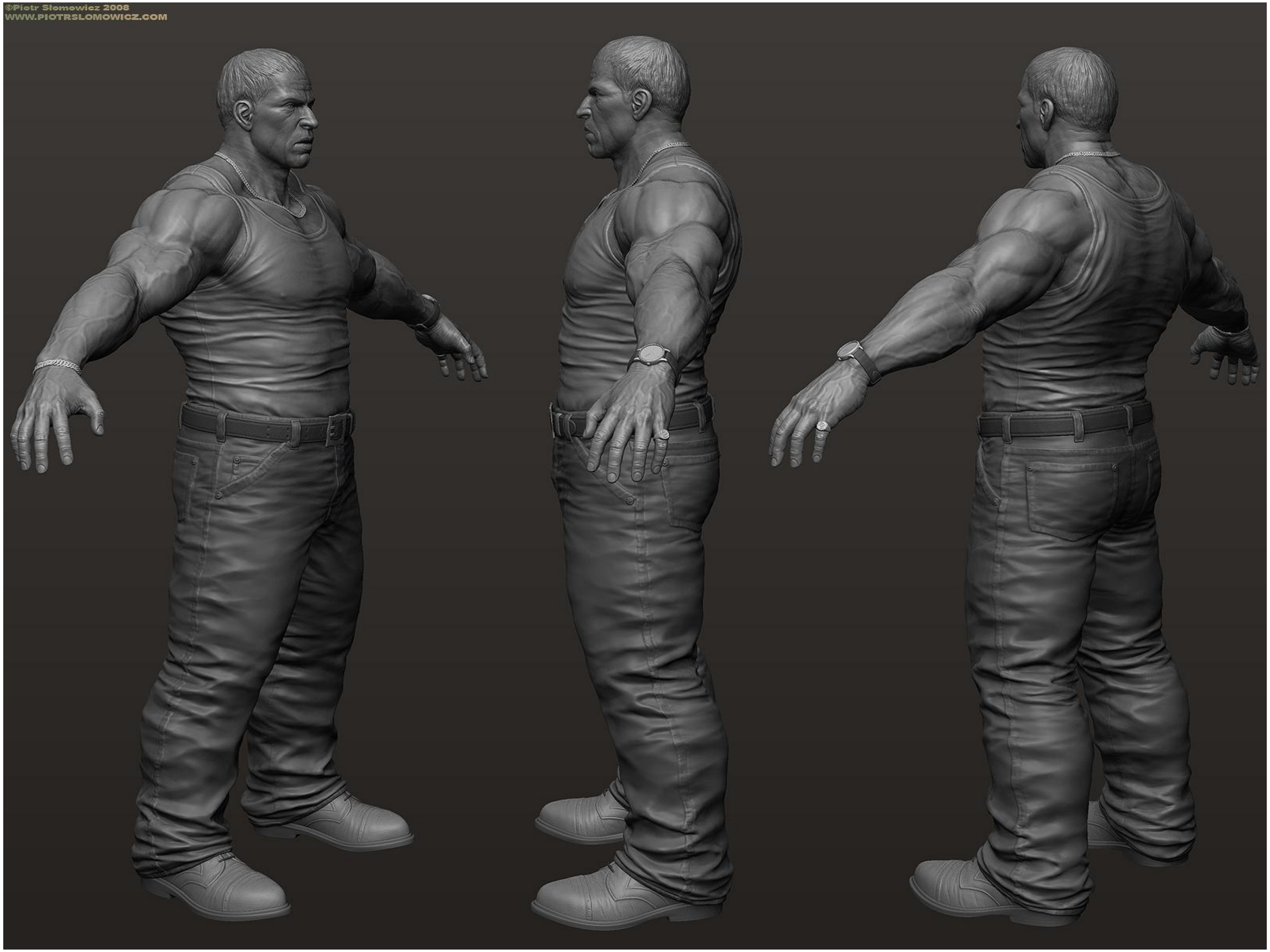
Normal map Example

+



Model by Piotr Slomowicz

Normal map Example



Normal map Example

Normal mapping + Environment mapping

- Normal and Parallax mapping combines beautifully with environment mapping

Demo

EMNM (World Space)

- For each vertex
  - Transform V to world space
  - Compute tangent space to world space transform (T, B, N)
- For each fragment
  - Interpolate and renormalize V
  - Interpolate frame (T, B, N)
  - Lookup N’ = texture(s, t)
  - Transform N’ from tangent space to world space
  - Compute reflection vector R (in world space) using N’
  - Lookup C = cubemap(R)

Normal and Parallax Normal Map Issues

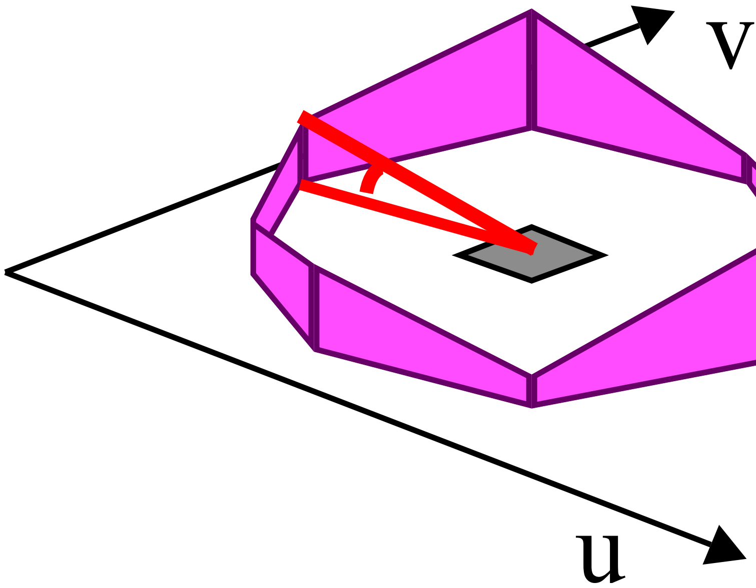
- Artifacts
  - No shadowing
  - Silhouettes still edgy
  - No parallax for Normal mapping
- Parallax Normal Mapping
  - No occlusion, just distortion
  - Not accurate for high frequency height-fields (local constant heightfield assumption does not work)
  - No silhouettes

Normal Mapping Issues

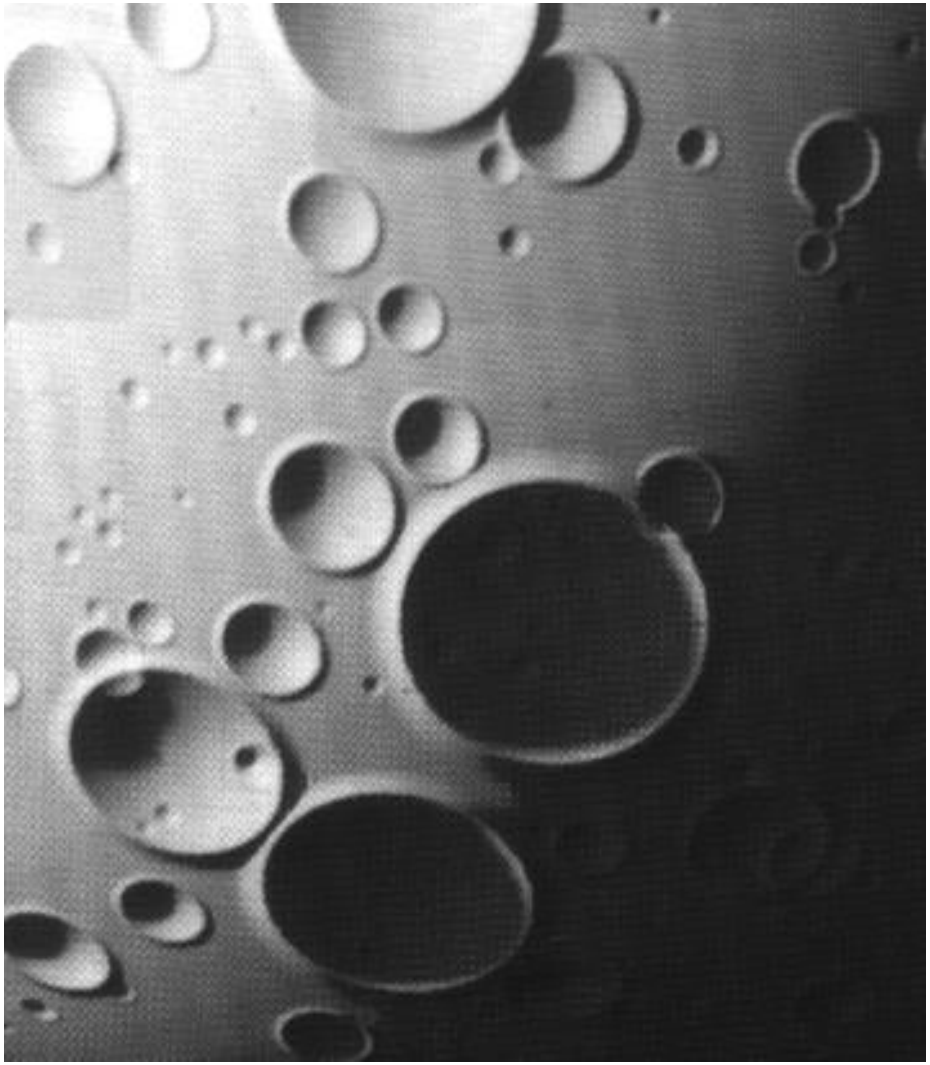
- Normal Mapping Effectiveness
  - No effect if neither light nor object moves!
  - In this case, use light maps
  - Exception: specular highlights

Horizon Mapping

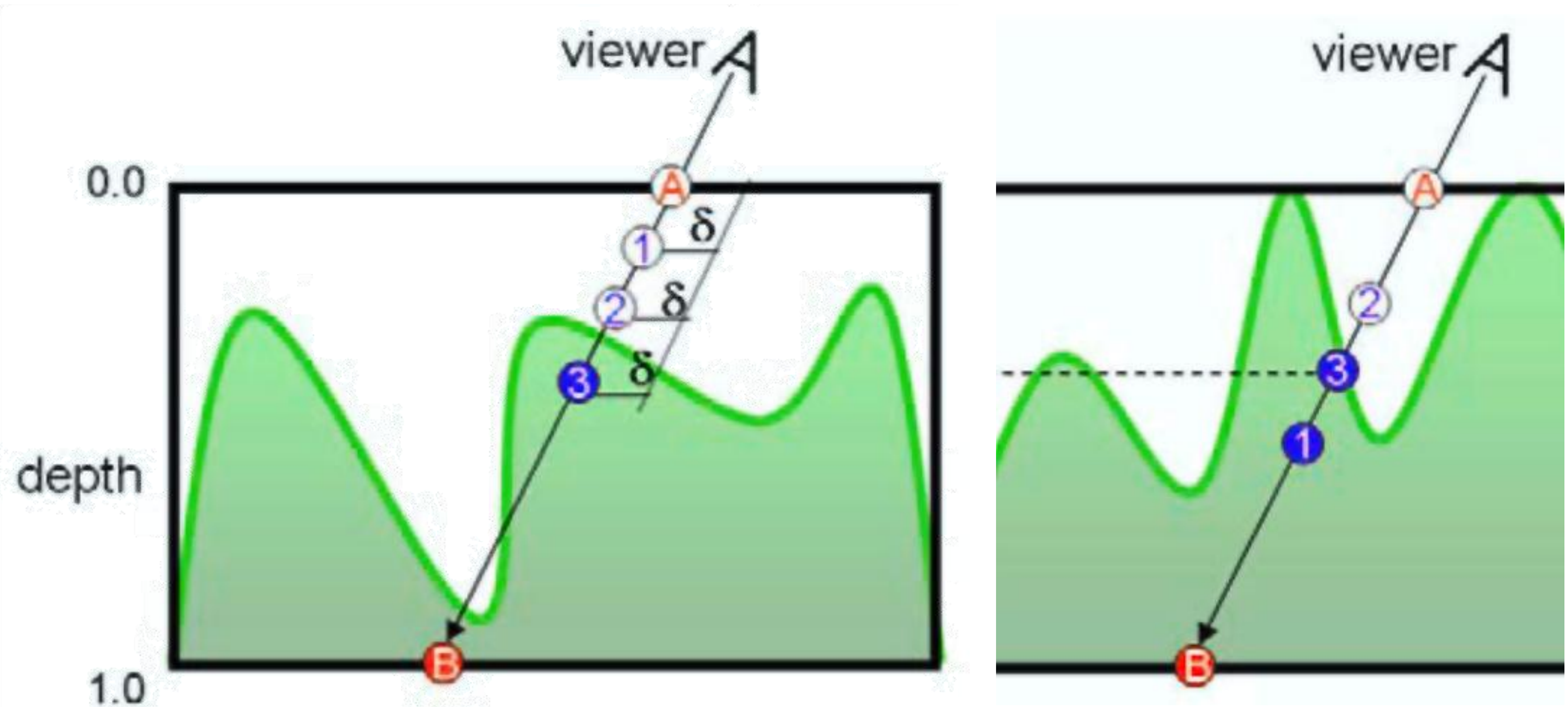
- Improve normal mapping with (local) shadows
- Preprocess: compute *n* horizon values per texel
- Runtime:
  - Interpolate horizon values
  - Shadow accordingly



Horizon Mapping Examples



- At runtime: perform ray casting in the pixel shader
  - Calculate entry (A) and exit point (B)
  - March along ray until intersection with height field is found
  - Binary search to refine the intersection position



Relief Mapping Examples

Texture mapping

Parallax mapping

Relief mapping

Speed considerations

- Parallax-normalmapping
  - ~ 20 ALU instructions
- Relief-mapping
  - Marching and binary search:
  - ~300 ALU instructions
  - + lots of texture lookups

Advanced Methods

- Higher-Order surface approximation relief mapping
  - Surface approximated with polynomes
  - Produces silhouettes
- Prism tracing
  - Produces near-correct silhouette
- Many variations to accelerate tracing
  - Cut down tracing cost
  - Shadows in relief

Normal and Parallax normal map Toolset

- DCC Packages (Blender, Maya, 3DSMax)
- Nvidia Normalmap Filter for Photoshop or Gimp Normalmap filter
  - Create Normalmaps directly from Pictures
    - Not accurate!, but sometimes sufficient
- NVIDIA Melody
- xNormal (free)
- Crazybump (free beta)
  - Much better than PS/Gimp Filters!
- Tangent space can be often created using graphics/game engine

Tipps

- Download FXComposer and Rendermonkey
  - Tons of shader examples
  - Optimized code
  - Good IDE to play around
- Books:
  - GPU Gems Series
  - ShaderX Series
  - Both include sample code!