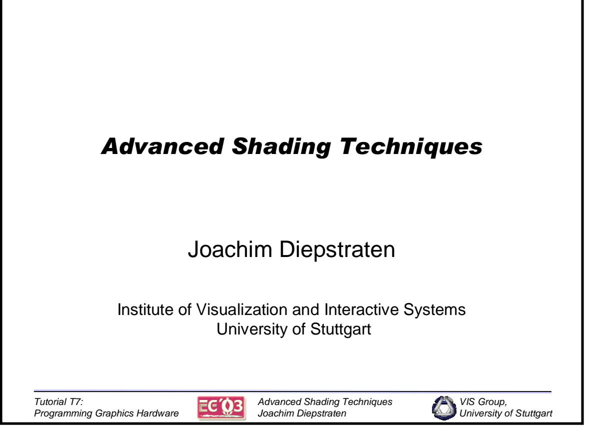
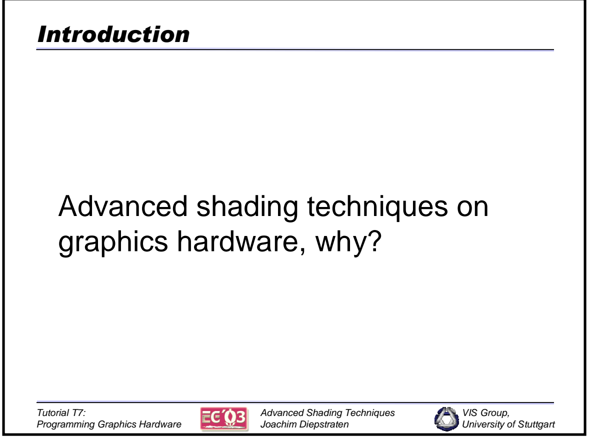
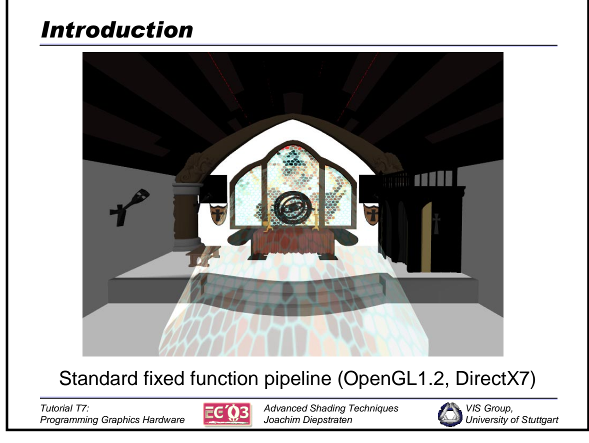
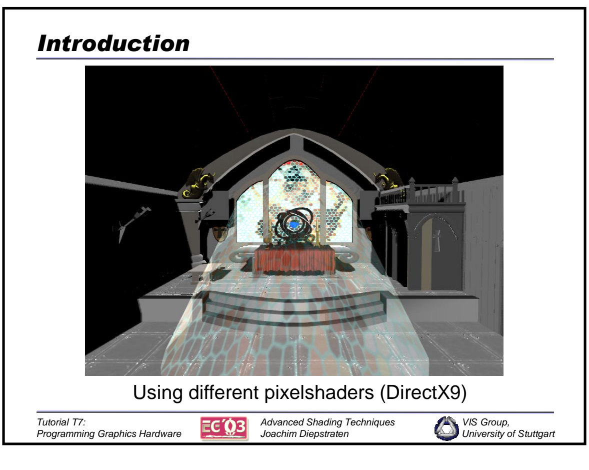
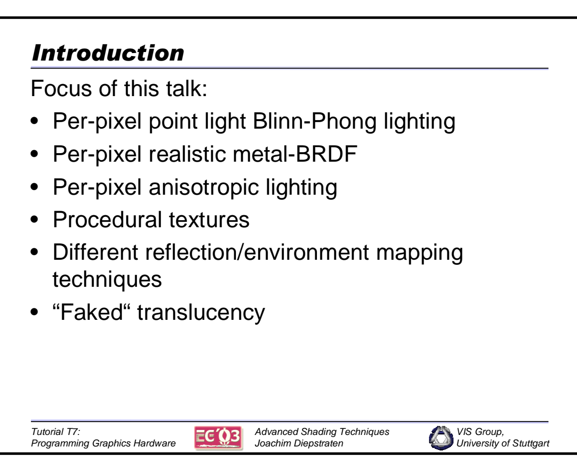
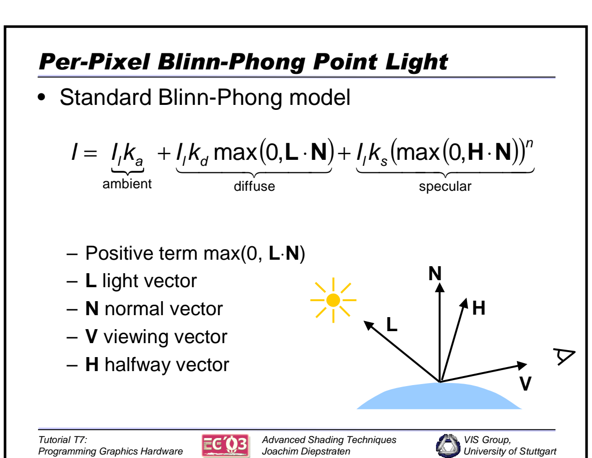

Advanced Shading Techniques
Joachim Diepstraten
Institute of Visualization and Interactive Systems
University of Stuttgart
Tutorial T7: Programming Graphics Hardware
Advanced Shading Techniques Joachim Diepstraten
VIS Group, University of Stuttgart


Introduction
Advanced shading techniques on graphics hardware, why?
Tutorial T7: Programming Graphics Hardware
Advanced Shading Techniques Joachim Diepstraten
VIS Group, University of Stuttgart


Introduction
Standard fixed function pipeline (OpenGL1.2, DirectX7)
Tutorial T7: Programming Graphics Hardware
Advanced Shading Techniques Joachim Diepstraten
VIS Group, University of Stuttgart


Introduction
Using different pixelshaders (DirectX9)
Tutorial T7: Programming Graphics Hardware
Advanced Shading Techniques Joachim Diepstraten
VIS Group, University of Stuttgart


Introduction
Focus of this talk:
Per-pixel point light Blinn-Phong lighting
Per-pixel realistic metal-BRDF
Per-pixel anisotropic lighting
Procedural textures
Different reflection/environment mapping techniques
"Faked" translucency
Tutorial T7: Programming Graphics Hardware
Advanced Shading Techniques Joachim Diepstraten
VIS Group, University of Stuttgart


Per-Pixel Blinn-Phong Point Light
Standard Blinn-Phong model
$I = \underbrace{I_l k_a}_{\text{ambient}} + \underbrace{I_l k_d \max(0, \mathbf{L} \cdot \mathbf{N})}_{\text{diffuse}} + \underbrace{I_l k_s (\max(0, \mathbf{H} \cdot \mathbf{N}))^n}_{\text{specular}}$
Positive term max(0, L·N)
L light vector
N normal vector
V viewing vector
H halfway vector
N
H
L
V
Tutorial T7: Programming Graphics Hardware
Advanced Shading Techniques Joachim Diepstraten
VIS Group, University of Stuttgart

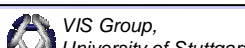
delivered by
EUROGRAPHICS
DIGITAL LIBRARY
www.eg.org
diglib.eg.org

Per-Pixel Blinn-Phong Point Light

- Add attenuation factor for point lights:
- Simple inverse square law

$$I_l = I_{max} 1/|P - L_{pos}|^2$$

- Precompute attenuation factor in vertex shader
- When lazy and enough fragment processing power do it at fragment level

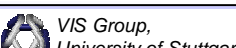


Per-Pixel Blinn-Phong Point Light

- Vertex Shader 1.1 (DirectX8.1) code:

```
vs.1.1
dcl_position v0
dcl_normal v1
dcl_texcoord0 v2
// Transform and output position
m4x4 oPos, v0, c0 //c0 MVP
m4x4 r0, v0, c4   //c4 M
m4x4 r1, v0, c8   //c8 MV
mov oT0, v1
mov oT1, r0
mov oT2, r1
```

- Simply pass data to fragment processor



Per-Pixel Blinn-Phong Point Light

- Pixel Shader 2.0 (DirectX9) code:

```
// Phong per-pixel lighting
// c0-c3 inverse transposed modelview matrix
// c4    light position .w (falloff range)
// c5.x  1/(falloff range) c5.y 0.5 c5.z kd c5.w ks
// c6    eye position
// c7    specular material color
// c8    diffuse material color
ps.2.0
dcl_2d s0 // sampler for exponential lookup
dcl t0 // normal
dcl t1 // point in world space
dcl t2 // point in camera space
def c9, 0.0, 0.0, 0.0, 0.0 // additional constants
```


Per-Pixel Blinn-Phong Point Light

- Pixel Shader 2.0 (DirectX9) code (cont.):

```
// transform normal
m3x3 r0.xyz, t0, c0
// normalize normal vector
nrm r1, r0
// compute light vector direction
sub r2, t1, c4 // P-Lpos
// compute distance between light and object
dp3 r4, r2, r2
rsq r5, r4.x // 1/|P-Lpos|
rcp r4, r5.x // |P-Lpos|
mul r4, r4, r4
// normalize light vector
nrm r3, r2
dp3 r0, r1, r3 // diffuse term N.L
max r0, r0, c9.x // max(0,N.L)
```


Per-Pixel Blinn-Phong Point Light

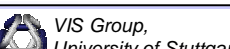
- Pixel Shader 2.0 (DirectX9) code (cont.):

```
// compute attenuation
sub r2, c4.w, r4.x
mul r2, c5.x, r2
abs r8, r2
cmp r2, -r8, c9.x, r2 // check if <0 then 0
// diffuse part
mul r0, r0, r2         // I*N.L
mul r0, r0, c8         // I*ColDiffuse*N.L
mul r0, r0, c5.z       // I*kd*N.L*ColDiffuse
// specular part
sub r4, c6, t2         // compute view vector
nrm r5, r4
```


Per-Pixel Blinn-Phong Point Light

- Pixel Shader 2.0 (DirectX9) code (final part):

```
// compute halfway vector (Blinn)
add r2, r3, r5       // L+V
mul r2, r2, c5.y     // L+V/2 = H
dp3 r2, r2, r1       // H.N
texld r2, r2, s0     // lookup (H.N)^k
mul r2, r2.x, c7     // ColSpec*(H.N)^k
mul r2, r2, c5.w     // I*ColSpec*(H.N)^k
add r0, r0, r2.x     // I*ks*ColSpec*(H.N)^k
add r0, r0, c11      // add ambient term
mov oC0, r0          // set output color
```

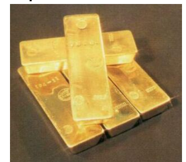


Per-Pixel Blinn-Phong Point Light

- Results (Demo)

Per-Pixel Realistic Metal-BRDF

- Properties of metal surfaces:



  - Diffuse reflectance is usually negligible
  - Color reflected determined by fresnel function
  - Shininess depends on surface roughness
  - At great incident angles peak of reflection lobe greater than angle of incident

Tutorial T7: Programming Graphics Hardware | Advanced Shading Techniques Joachim Diepstraten | VIS Group, University of Stuttgart

Per-Pixel Realistic Metal-BRDF

- Comparison of well-known existing BRDFs

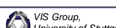
	Recp. Blinn	Cook-Torrance	Ward	He-Torrance
Metallic	No	Yes	Yes	Yes
Physical plausible	Yes	No	No	Yes
Physical based	No	Yes	No	Yes
Off-specular peak	No	Yes	No	Yes

Tutorial T7: Programming Graphics Hardware | Advanced Shading Techniques Joachim Diepstraten | VIS Group, University of Stuttgart

Per-Pixel Realistic Metal-BRDF

- Best choice would be He-Torrance, but
  - Very time-consuming and complex processing
  - Not well suited for hardware shader implementation
  - High number of not-easy to determine parameters
- Second best choice Cook&Torrance,but
  - Still requires many arithmetic operations (arccos, sin, cos)
- Solution: Stretched-Phong model (Neumann et al. Eurographics 1999)

Tutorial T7: Programming Graphics Hardware | Advanced Shading Techniques Joachim Diepstraten | VIS Group, University of Stuttgart



Per-Pixel Realistic Metal-BRDF

- Stretched-Phong-Model properties:
  - Emperical model but physical plausible
  - Satisfies basic properties of metals
  - Replaces $f_{r,Phong}(\vec{L},\vec{V}) = c_n \cdot \cos^n \alpha$ from reciprocal Phong with

$$f_{r,Phong}(\vec{L},\vec{V}) = c_n \cdot \frac{\cos^n \alpha}{\cos \Theta_{\min}}$$

$$c_n = \frac{n+2}{2\pi} \qquad \cos\alpha = (\vec{R}\cdot\vec{V})^+ \qquad \Theta_{\min} = \min(\Theta_{\vec{L}}, \Theta_{\vec{V}})$$

Final equation:

$$f_{r,Phong}(\vec{L},\vec{V}) = c_n \cdot \frac{[(2(\vec{N}\cdot\vec{L})(\vec{N}\cdot\vec{V}) - (\vec{L}\cdot\vec{V}))^+]^n}{\max((\vec{N}\cdot\vec{L}),(\vec{N}\cdot\vec{V}))}$$

Tutorial T7: Programming Graphics Hardware | Advanced Shading Techniques Joachim Diepstraten | VIS Group, University of Stuttgart

Per-Pixel Realistic Metal-BRDF

- Vertex Shader 1.1 (DirectX8.1) code:

```
vs.1.1
dcl_position v0
dcl_normal v1
dcl_texcoord0 v2
// transform and output position
m4x4 r0, v0, c4   //c4 MV
m4x4 oPos, v0, c0 //c0 MVP
mov oT0, v1
mov oT1, r0
```

Tutorial T7: Programming Graphics Hardware | Advanced Shading Techniques Joachim Diepstraten | VIS Group, University of Stuttgart

Per-Pixel Realistic Metal-BRDF

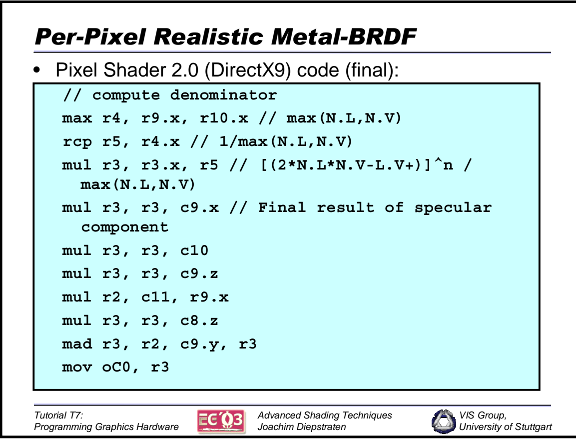
- Pixel Shader 2.0 (DirectX9) code:

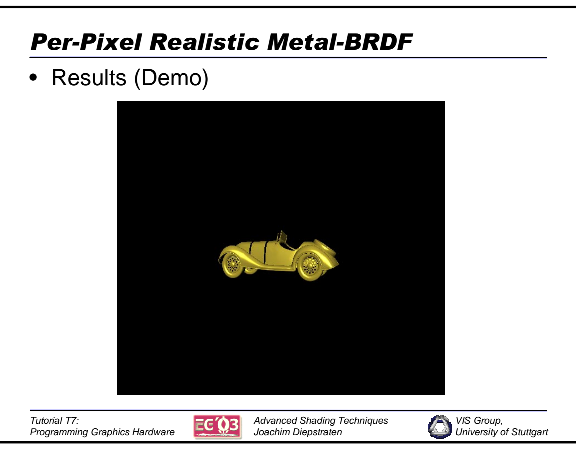
```
// Metal shader / Stretched cosine lobe
ps.2.0
dcl_2d s0 // sampler for power lookup
dcl t0 // normal
dcl t1 // point in camera space
// c5 = View vector
// c6 = Light vector
// c9.x n+2/2*PI .y kd .z ks
// c10 = specular color
// c11 = diffuse color
def c8, 2.0, 1.0, 0.5, 0.0
// transform normal
m3x3 r0.xyz, t0, c0
// normalize normal
nrm r1, r0
```

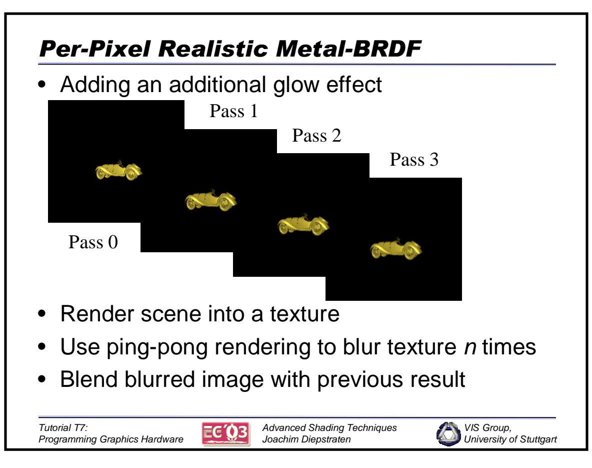

Per-Pixel Realistic Metal-BRDF

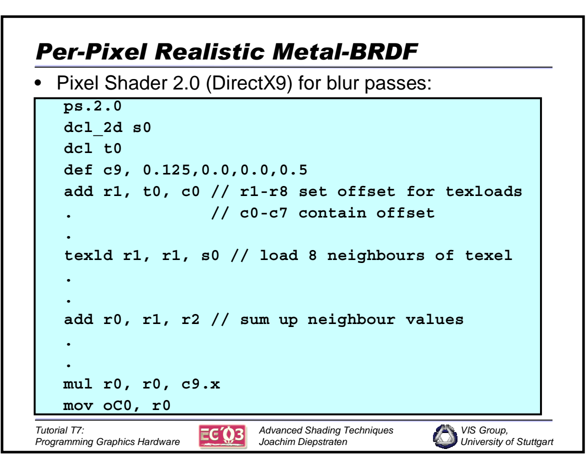
- Pixel Shader 2.0 (DirectX9) code (cont.):

```
dp3 r9, r1, c6 // N.L
// compute view vector
sub r3, c4, t1
// normalize view vector
nrm r2, r3
dp3 r10, r1, r2 // N.V
mul r0, r9.x, r10.x // N.L * N.V
mul r0, r0, c8.x // 2 * N.L * N.V
// compute L.V+
dp3 r5, c6, r2
cmp r5, r5.x, r5, c8.w // check on <0
sub r0, r0, r5 // 2* N.L * N.V - L.V+
texld r3, r0, s0 // lookup specular power
```

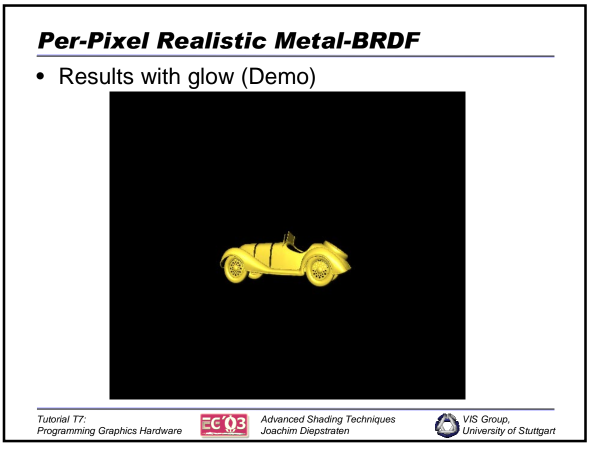






Per-Pixel Realistic Metal-BRDF

- Results with glow (Demo)



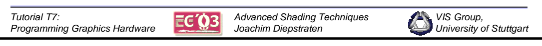
Per-Pixel Anisotropic Lighting

- What do these surfaces have in common?
  - Satin cloth
  - Christmas tree balls (the ornaments covered in threads)
  - Compact Discs
  - Brushed metal
  - Hair
- Answer: they all exhibit anisotropic lighting characteristics

Per-Pixel Anisotropic Lighting

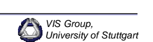
- What creates anisotropy?
- Key to anisotropic surfaces:

  distribution of surface normals along the surface scratches or fibers is different from the distribution across them
- A relatively distant observer sees the lighting result, but not the microstructure



Per-Pixel Anisotropic Lighting

- Lighting a Fiber:
  - Fibers do not have a traditional surface normal!
  - Logically, each point on the thread has an infinite "circle" of normals perpendicular to the tangent at the point.
- Which normal should be used for lighting?
  - Ideally a contribution of each of them
  - Too complex, therefore use most significant one
- Most significant normal?
  - Pick normal co-planar with light vector and view vector
  - This vector will maximize lighting dot product

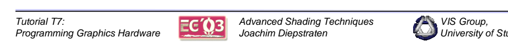


Per-Pixel Anisotropic Lighting

- Instead of determining most significant normal explicit express $\vec{L}\cdot\vec{N}$ and $\vec{V}\cdot\vec{R}$ through $\vec{L}, \vec{V}$, and $\vec{T}$

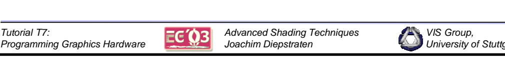
$$\vec{L}\cdot\vec{N} = \sqrt{1-(\vec{L}\cdot\vec{T})^2}$$

$$\vec{R}\cdot\vec{V} = (\vec{L}\cdot\vec{T})(\vec{V}\cdot\vec{T}) - \sqrt{1-(\vec{L}\cdot\vec{T})^2}\times\sqrt{1-(\vec{L}\cdot\vec{T})^2}$$



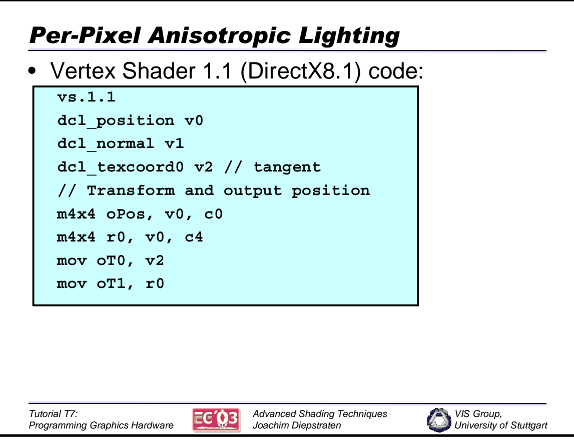
Per-Pixel Anisotropic Lighting

- Determining the Tangent vectors?
  - Can be assigned implicitly for common shapes like disks and spheres, etc.
  - What about arbitrary meshes?
- No explicit parametric basis available
- But most models have textures and texture coordinates
- Determine an *ad hoc* tangent space from the model's mapping between vertices and texture coordinates



Per-Pixel Anisotropic Lighting

- Each vertex consists of:
  - (x, y, z) object-space position
  - corresponding (s, t) on the texture
  - And each triangle in the model has three such vertices
- Per polygon, compute the following plane equations:
  - $A_0\, x + B_0\, s + C_0\, t + D_0 = 0$
  - $A_1\, y + B_1\, s + C_1\, t + D_1 = 0$
  - $A_2\, z + B_2\, s + C_2\, t + D_2 = 0$
- Make a per-polygon tangent:
  - tangent is normalization of $(\partial x/\partial s, \partial y/\partial s, \partial z/\partial s)$
  - or normalization of $(-B_0/A_0, -B_1/A_1, -B_2/A_1)$
- Smooth tangent at each vertex in the same way as normals



Per-Pixel Anisotropic Lighting

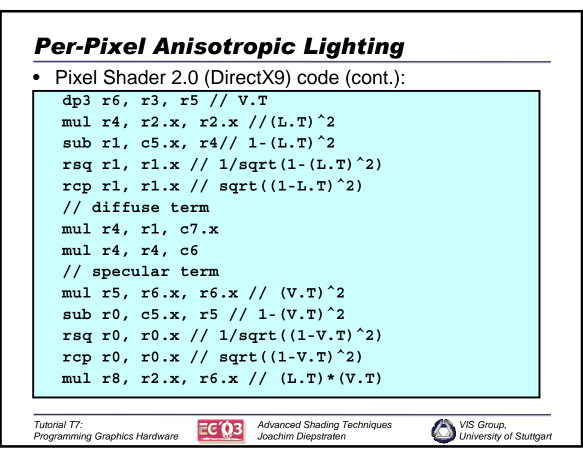
- Vertex Shader 1.1 (DirectX8.1) code:

```
vs.1.1
dcl_position v0
dcl_normal v1
dcl_texcoord0 v2 // tangent
// Transform and output position
m4x4 oPos, v0, c0
m4x4 r0, v0, c4
mov oT0, v2
mov oT1, r0
```


Per-Pixel Anisotropic Lighting

- Pixel Shader 2.0 (DirectX9) code:

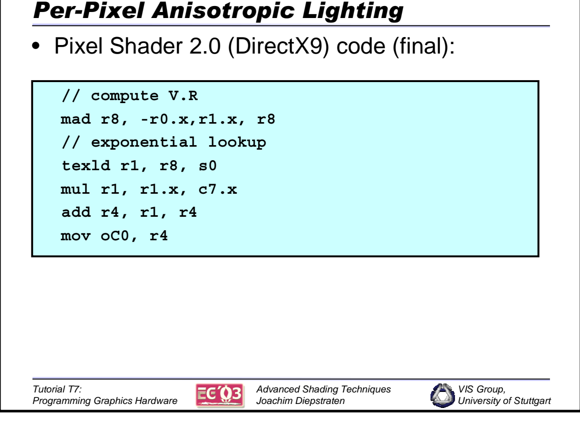
```
// Anisotrop
ps.2.0
dcl t0 // tangent
dcl t1 // point in camera space
dcl_2d s0
def c5, 1.0, 0.0, 0.0,0.0
// compute transformed tanget
m3x3 r0.xyz, t0, c0
nrm r3, r0
// compute view vector
sub r4, c10, t1
nrm r5, r4
dp3 r2, r3, c4 // L.T
```

Per-Pixel Anisotropic Lighting

- Pixel Shader 2.0 (DirectX9) code (cont.):

```
dp3 r6, r3, r5 // V.T
mul r4, r2.x, r2.x //(L.T)^2
sub r1, c5.x, r4// 1-(L.T)^2
rsq r1, r1.x // 1/sqrt(1-(L.T)^2)
rcp r1, r1.x // sqrt((1-L.T)^2)
// diffuse term
mul r4, r1, c7.x
mul r4, r4, c6
// specular term
mul r5, r6.x, r6.x // (V.T)^2
sub r0, c5.x, r5 // 1-(V.T)^2
rsq r0, r0.x // 1/sqrt((1-V.T)^2)
rcp r0, r0.x // sqrt((1-V.T)^2)
mul r8, r2.x, r6.x // (L.T)*(V.T)
```

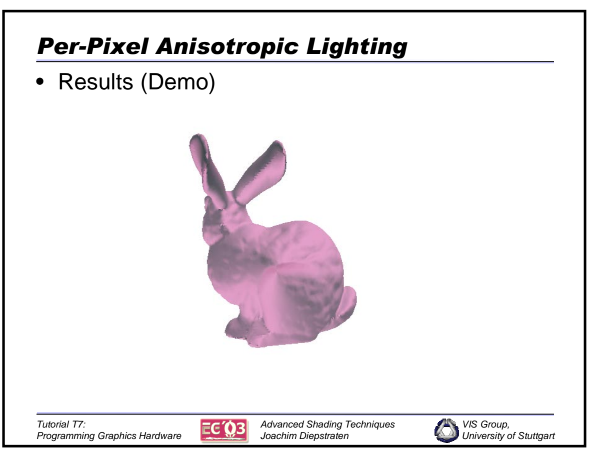
Per-Pixel Anisotropic Lighting

- Pixel Shader 2.0 (DirectX9) code (final):

```
// compute V.R
mad r8, -r0.x,r1.x, r8
// exponential lookup
texld r1, r8, s0
mul r1, r1.x, c7.x
add r4, r1, r4
mov oC0, r4
```


Per-Pixel Anisotropic Lighting

- Results (Demo)



Procedural Textures

- Advantage of procedural textures:
  - Compact (small code compared to textures)
  - No fixed resolution
  - Infinite detail (limited only by precision)
  - Parameterized (easy generation of variations)
  - Solid texturing (avoids 2D mapping problems)
- When to use procedural textures:
  - Animating effects (clouds, fire, water)
  - Where repeating textures would be obvious

Procedural Textures

- Procedural noise:
  - Noise is an important part of many procedural textures
  - Provides a controlled method of adding randomness to:
    - Color,texture
    - Bump map
    - Animation
    - Terrains, etc
  - Random yet smooth

Procedural Textures

- How to do procedural noise with shaders:
  - Pre-compute 3D Texture containing random values (64x64x64 should be fine)
  - Pre-filtering with tri-cubic filter helps to avoid interpolation artifacts
  - 3 lookups into this 3D texture produces reasonable procedural noise
  - Compute texture coordinates with vertex shader by using „texture matrices“ (basically scaled model transform matrices)

Procedural Textures

- Vertex Shader 1.1 (DirectX8.1) code:

```
vs.1.1
dcl_position v0
dcl_normal   v1
m4x4 oPos, v0, c0
// Transformed Pshade (using texture matrix 0)
m4x4 oT0, v0, c4
// Transformed Pshade (using texture matrix 1)
m4x4 oT1, v0, c8
// Transformed Pshade (using texture matrix 2)
m4x4 oT2, v0, c12
```

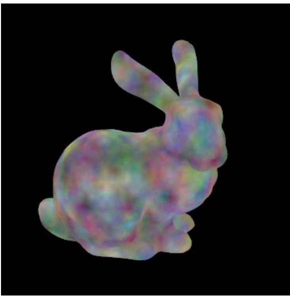

Procedural Textures

- Pixel Shader 2.0 (DirectX9) code:

```
ps.2.0
dcl t0  // noise lookup 1
dcl t1  // noise lookup 2
dcl t2  // noise lookup 3
// Luminance-only Volume noise
dcl_volume s0
// Sample dX from scalar volume noise
texld r3,  t0, s0
// Sample dY from scalar volume noise
texld r4,  t1, s0
// Sample dZ from scalar volume noise
texld r5,  t2, s0
mov r3.y, r4.x  // Put dY in y
mov r3.z, r5.x  // Put dZ in z
mov oC0, r3
```


Procedural Textures

- Results (Demo)



Environment Mapping Techniques

- Shiny/chrome and mirror-like objects are fancy
- Curve-reflections are hard to emulate
- Are “faked” with environment maps
- Many types of environment maps exist (Cubic,spherical, dual-paraboloid)
- Most often used one today: Cubic

Environment Mapping Techniques

- Cubic environment mapping:
  - Fixed function since DirectX7 and OpenGL1.3
  - Automatic texture generation
  - Cubic maps can be used also for other things
- When using shaders:
  - Automatic texture generation disabled!
  - Other additional fancy things are now possible

Environment Mapping Techniques

- Pixel Shader 2.0 (DirectX9) code for normal cube environment-mapping:

```
// c0--c3 Inverse Transpose ModelView Matrix
// c6 eyepos
ps.2.0
dcl_cube s0
dcl t0 // normal in object space
dcl t1 // point in camera space
def c11, 2.0, 0.0, 0.0, 0.0
// normal
m3x3 r0.xyz, t0, c0
// normalize normal
nrm r1, r0
// compute view vector
sub r2, c6, t1
```


Environment Mapping Techniques

- Pixel Shader 2.0 (DirectX9) code for normal cube environment-mapping (cont.):

```
// normalize view vector
nrm r3, r2
// calculate reflection vector
dp3 r4, r1, r3 // N.V
rcp r2, r4.x // 1/N.V
mul r4, r3, r2.x // V * 1/N.V
mad r6, c11.x, r1, -r4 // 2 * N - 1 / N.V * V
texld r0, r6, s0 //lookup in cube map
mov oC0, r0
```

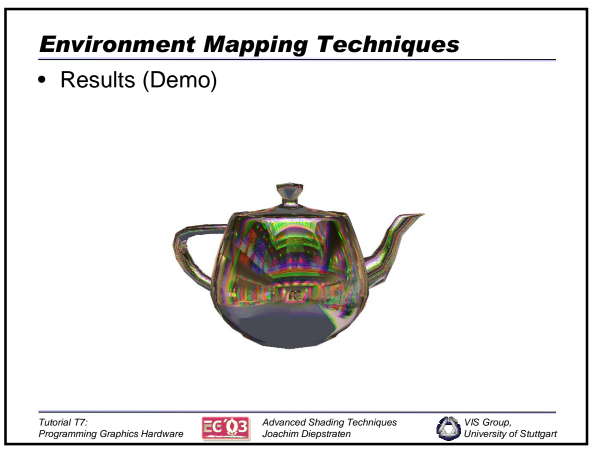

Environment Mapping Techniques

- Results (Demo)

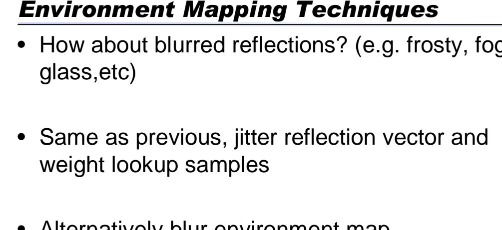
Environment Mapping Techniques

- Nice but lets make it a bit more fancier, by adding a pseudo dispersion effect
- Fake dispersion by using different reflection vectors for each color channel

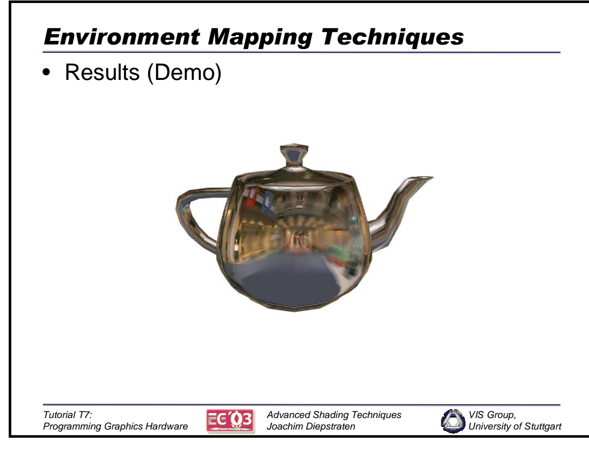
```
...
mad r6, c11.x, r1, -r4 // 2 * N - 1 / N.V * V
texld r0, r6, s0
mov r2.x, r0.x
mad r6, c11.y, r1, -r4 // 1.75 * N - 1 / N.V * V
texld r0, r6, s0
mov r2.y, r0.y
mad r6, c11.z, r1, -r4 // 2.25 * N - 1 / N.V * V
texld r0, r6, s0
...
```

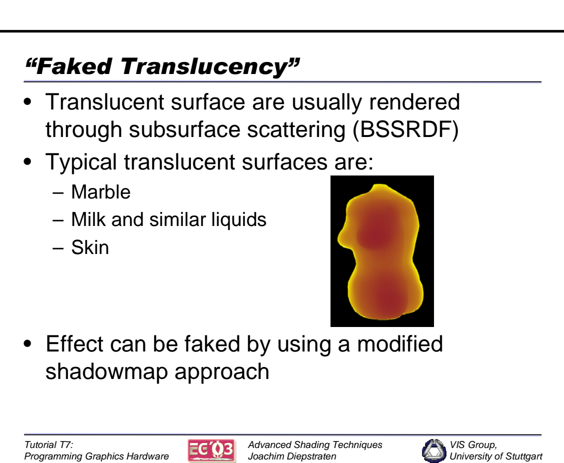


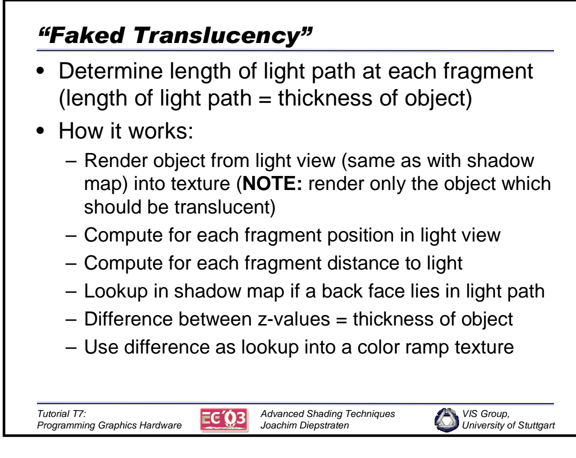
Environment Mapping Techniques

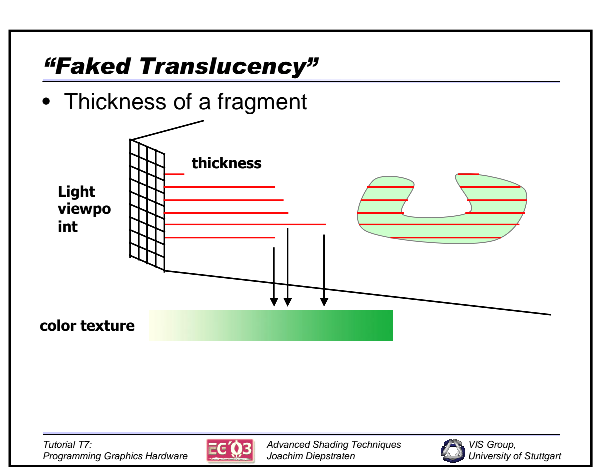


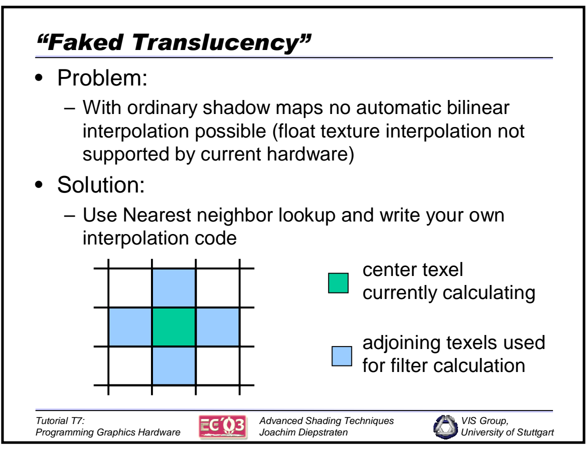
- How about blurred reflections? (e.g. frosty, foggy glass,etc)
- Same as previous, jitter reflection vector and weight lookup samples
- Alternatively blur environment map
  - Unclear how to blur a cube map correctly!
  - Mixing between unblurred, partial blurred and blurred regions on object harder to achieve (would require additional cube maps)

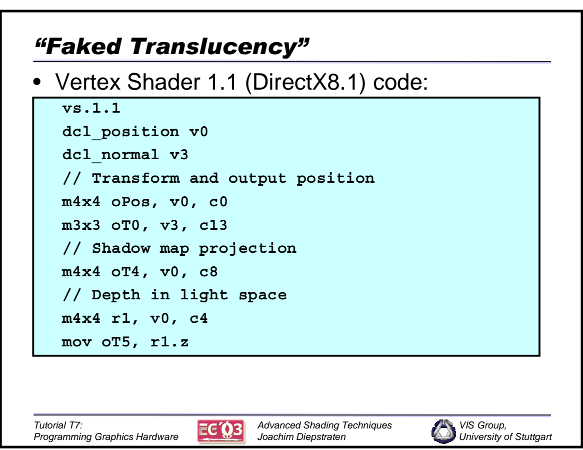










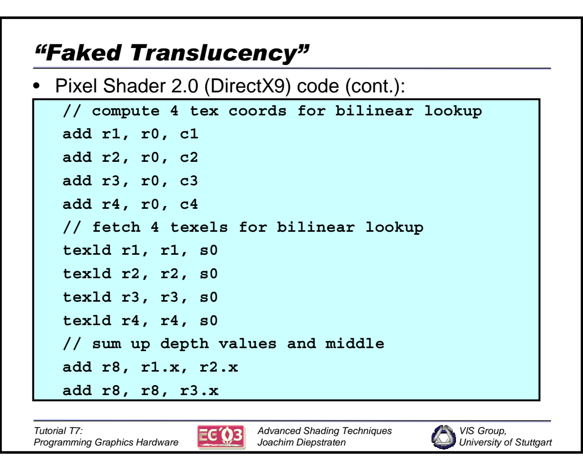


"Faked Translucency"

- Pixel Shader 2.0 (DirectX9) code:

```
ps.2.0
dcl t0
dcl t1
dcl_2d s0
dcl_2d s1
def c6, 0.0, 1.0, 1.0, 0.0 // alpha
def c7, 1.0, 0.0, 0.0, 1.0 // 1-alpha
def c8, 0.0, 0.0, 1.0, 1.0 // beta
def c9, 1.0, 1.0, 0.0, 0.0 // 1-beta
def c10, 1.0, 0.25, 0.0, 0.0
// compute center of 2D texture by x/w, y/w
rcp r1, t0.w
mul r0, t0, r1
```

Tutorial T7: Programming Graphics Hardware | Advanced Shading Techniques Joachim Diepstraten | VIS Group, University of Stuttgart



"Faked Translucency"

- Pixel Shader 2.0 (DirectX9) code (cont.):

```
add r8, r8, r4.x
mul r8, r8, c10.y
mad r1, r0, c5.x, c5.y // integer-oriented coords
frc r2, r1  // fractional part (left)
add r3, -r2, c5.z  // 1-fractional part -> right
mul r4, r2.x, c6  // alpha for bilinear
mul r5, r3.x, c7  // 1-alpha for bilinear
add r6, r4, r5    // complete alpha for bilinear
mul r4, r2.y, c8  // beta for bilinear
mul r5, r3.y, c9  // 1-beta for bilinear
add r4, r4, r5    // complete beta components
mul r6,r4,r6      // alpha*beta complete weights
```

Tutorial T7: Programming Graphics Hardware | Advanced Shading Techniques Joachim Diepstraten | VIS Group, University of Stuttgart

"Faked Translucency"

- Pixel Shader 2.0 (DirectX9) code (cont.):

```
// weight sum for bilinear interpolation
dp4 r8, r8, r6
// compute thickness
sub r2, t1, r8
// 1-thickness
sub r2, c10.x, r2
// add square fall of
mul r2, r2, r2
sub r2, c10.x, r2
// lookup in color ramp
texld r2, r2, s2
// compute dotproduct to check which side
// the surface lies to light source
nrm r1, t0
```

Tutorial T7: Programming Graphics Hardware | Advanced Shading Techniques Joachim Diepstraten | VIS Group, University of Stuttgart

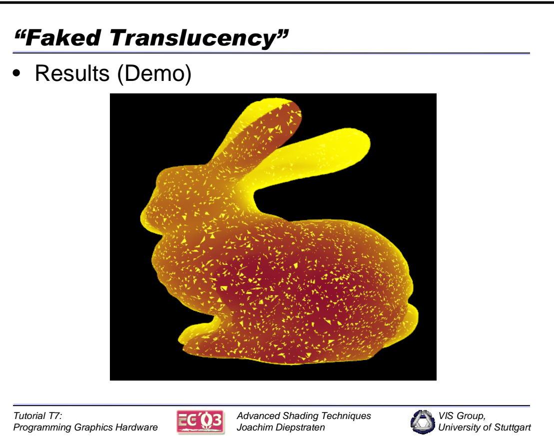
“Faked Translucency”

- Pixel Shader 2.0 (DirectX9) code (final):

```
dp3 r1, r1, c31
sub r5, c10.x, r1
// make a second lookup for light front facing
// surfaces
texld r5, r5, s3
// check if light front facing or not
cmp r1, r1, r5, r2
mov oC0, r1
```

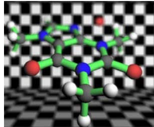

“Faked Translucency”

- Results (Demo)

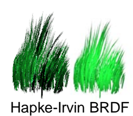
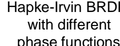


Summary

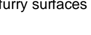
- Many nice shading effects are now possible!
- Other things we could not show:



Depth of field

Hapke-Irvin BRDF with different phase functions



furry surfaces

Dynamic Bump mapping

- Helps to increase realism in interactive 3D content

References

- [Blinn77] J. Blinn. Models of Light Reflection for Computer Synthesized Pictures. Proceedings of the 4th annual conference on Computer graphics and interactive techniques, pages 192-198, 1977
- [Neumann99] L. Neumann, A. Neumann, L. Szimay-Kalos. Compact Metallic Reflectance Models. Computer Graphics Forum Vol.18 no.3, 161-172, 1999
- [James03] Greg James, Direct3D Special Effects. GDC 2003 http://developer.nvidia.com/docs/IO/4449/SUPP/D3DTutorial_EffectsNV.ppt
- [Stalling97] D. Stalling, M. Zöckler, H.-C. Hege. Fast Display of Illuminated Lines. IEEE Transaction on Visualization and Computer Graphics Vol.3 no. 2 ,118-128,1997
- [Blinn76] J. Blinn, M.E. Newell. Texture and Reflection in Computer Generated Images**.** Communications of the ACM Vol. 19, no.10, 542-547, 1976
- [Perlin85] K. Perlin. An Image Synthesizer. Proceedings of the 12th annual conference on Computer graphics and interactive techniques, 287-296, 1985
- [Mitchell02] J.L.Mitchell. Hardware Shading on the ATI Radeon 9700. SIGGRAPH 2002 shading course #17