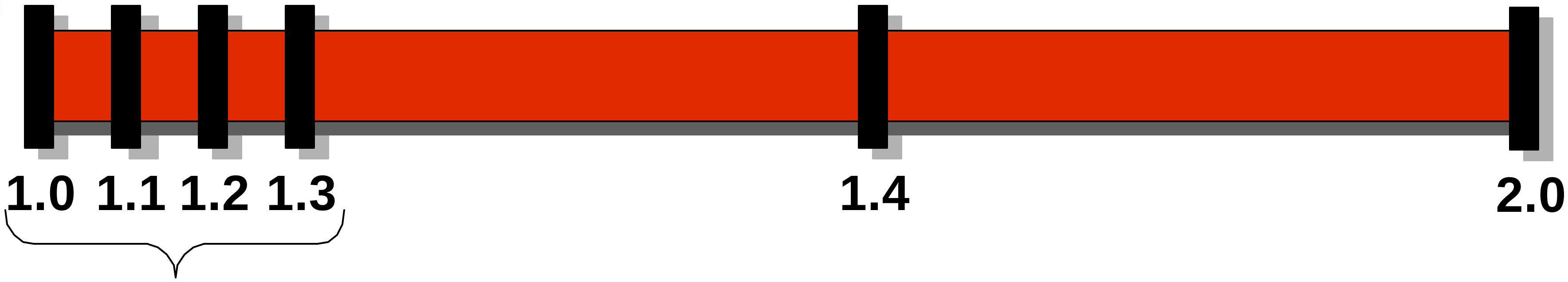
1.4 Pixel Shaders

Jason L. Mitchell
3D Application Research Group Lead
JasonM@ati.com

Outline

- Pixel Shader Overview
- 1.1 Shader Review
- 1.4 Pixel Shaders
  - Unified Instruction set
  - Flexible dependent texture read
- Image Processing
- 3D volume visualizations
  - Dynamic transfer functions
- Effects on 3D Surfaces
  - Per-pixel lighting
    - Diffuse
    - Specular – Dealing with halfangle denormalization
  - Per-pixel Fresnel Term
  - Bumpy Environment mapping
    - Bumped Cube mapping
    - Projected dudv
  - Per-pixel anisotropic lighting
- ShadeLab Tool
  - Good for playing around with shaders in real-time
  - Generates Vertex Shader code on the fly to feed pixel shader

The Road to ps.2.0 in DX9



- 4 Textures
- Fixed Address Ops
- 8 Color Ops

- 6 Textures (12 fetches)
- Unified Instruction Set
- 16 Instructions

- 8 Textures (16 fetches)
- Expanded Unified Instruction Set
- 64 Instructions

What is a Pixel Shader?

- A Pixel Shader is a set of microcode that you download to the GPU. These little programs execute on the GPU and operate on pixels and texels like the legacy multitexture pipeline
- Much more flexible than the legacy multitexture pipeline
- Multitexturing is still available, though not at the same time as Pixel Shading. Set the current pixel shader to 0 to use traditional multitexture

Pixel Shader API

Assemble and create the shader:

```
D3DXAssembleShader( strOpcodes,
lstrlen(strOpcodes), 0, NULL, &m_pD3DXBufShader,
&pBuffer );

m_pd3dDevice->CreatePixelShader(
(DWORD*)m_pD3DXBufShader->GetBufferPointer(),
&m_hPixelShader );
```

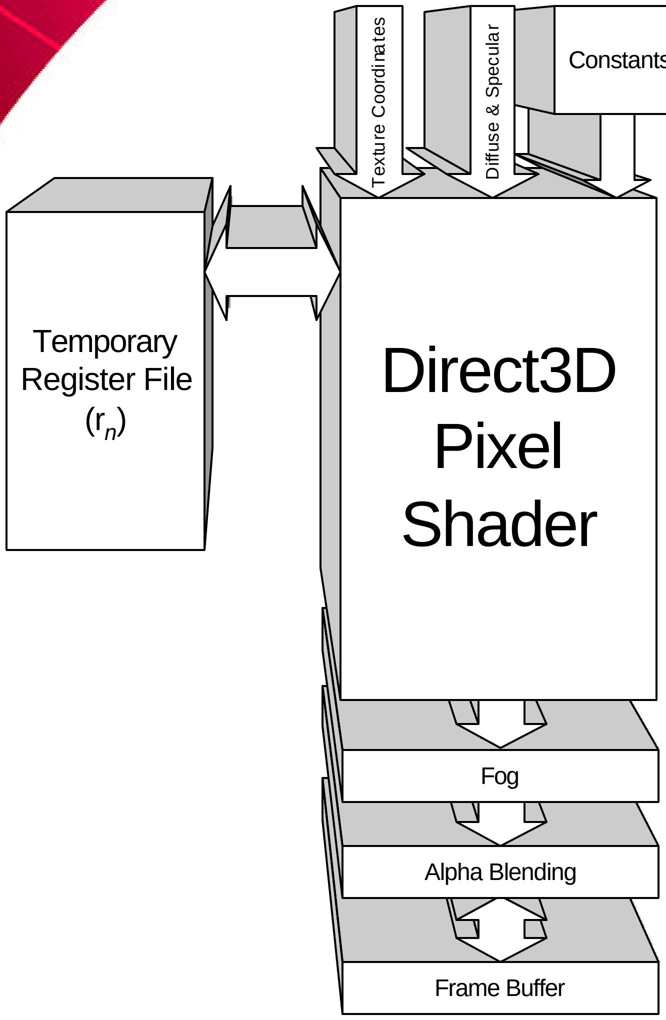
Set the current pixel shader:

```
m_pd3dDevice->SetPixelShader(m_hPixelShader);
```

Clean up on the way out:

```
m_pd3dDevice->SetPixelShader(0);
m_pd3dDevice->DeletePixelShader(m_hPixelShader);
```

Pixel Shader In's and Out's



- Inputs are texture coordinates, constants, diffuse and specular
- Several read-write temps
- Output color and alpha in r0.rgb and r0.a
- Output depth is in r5.r if you use texdepth (v 1.4)
- No separate specular add when using a pixel shader
  - You have to code it up yourself in the shader
- Fixed-function fog is still there
- Followed by alpha blending

Constants

- Eight read-only constants (c0..c7)
- Range -1 to +1
  - If you pass in anything outside of this range, it just gets clamped
- A given co-issue (rgb and a) instruction may only reference up to two constants
- Example constant definition syntax:

```
def c0, 1.0f, 0.5f, -0.3f, 1.0f
```

Interpolated Quantities

- Diffuse and Specular (v0 and v1)
  - Low precision and unsigned
  - In ps.1.1 through ps.1.3, available only "color shader"
  - Not available before ps.1.4 phase marker
- Texture coordinates
  - High precision signed interpolators
  - Can be used as extra colors, signed vectors, matrix rows etc

1.1 Model

- Four Textures
- Color Shader
  - Low Range and precision
  - 8 instructions
- Preceded by Address Shader
  - Fixed set of modes like bumped cubic environment mapping, a pair of dp3s etc
  - In fact, you can write them all down…

The 1.1 Address Shaders

One instruction

```
; One texture
ps.1.1
tex t0
```

```
; Texcoord as color
ps.1.1
texcoord t0
```

```
; Mimic a clip plane
ps.1.1
texkill t0
```

Two instructions

```
; Two textures
ps.1.1
tex t0
tex t1
```

```
; Two texcoords as colors
ps.1.1
texcoord t0
texcoord t1
```

```
; Mimic two clip planes
ps.1.1
texkill t0
texkill t1
```

```
; One texbem
ps.1.1
tex t0
texbem t1, t0
```

```
; One texbeml
ps.1.1
tex t0
texbeml t1, t0
```

```
; Color AR remapping
ps.1.1
tex t0
texreg2ar t1, t0
```

```
; Color GB remapping
ps.1.1
tex t0
texreg2gb t1, t0
```

```
; Sample and texcoord
ps.1.1
tex t0
texcoord t1
```

Three instructions

```
; 3 textures
ps.1.1
tex t0
tex t1
tex t2
```

```
; Mimic 3 clip planes
ps.1.1
texkill t0
texkill t1
texkill t2
```

```
; 2 samples & a texcoord
ps.1.1
tex t0
tex t1
texcoord t2
```

```
; One texbeml and a texcoord
ps.1.1
tex t0
texbeml t1, t0
texcoord t2
```

```
; One texbem and a sample
ps.1.1
tex t0
texbem t1, t0
tex t2
```

```
; One texbem and a texcoord
ps.1.1
tex t0
texbem t1, t0
texcoord t2
```

```
; One texbeml and a sample
ps.1.1
tex t0
texbeml t1, t0
tex t2
```

```
; Sample and 2 texcoords
ps.1.1
tex t0
texcoord t1
texcoord t2
```

```
; Three texcoords as colors
ps.1.1
texcoord t0
texcoord t1
texcoord t2
```

```
; 3x2 matrix multiply
ps.1.1
tex t0
texm3x2pad t1, t0
texm3x2tex t2, t0
```

The 1.1 Address Shaders

Four Instruction Shaders

```
; 4 textures
ps.1.1
tex t0
tex t1
tex t2
tex t3
```

```
; Mimic clip planes
ps.1.1
texkill t0
texkill t1
texkill t2
texkill t3
```

```
; 3 Samples and one texcoord
ps.1.1
tex t0
tex t1
tex t2
texcoord t3
```

```
; 2 Samples and 2 texcoords
ps.1.1
tex t0
tex t1
texcoord t2
texcoord t3
```

```
; 1 texbem and 2 samples
ps.1.1
tex t0
texbem t1, t0
tex t2
tex t3
```

```
; 1 Sample and 3 texcoords
ps.1.1
tex t0
texcoord t1
texcoord t2
texcoord t3
```

```
; Two texbems
ps.1.1
tex t0
texbem t1, t0
tex t2
texbem t3, t2
```

```
; 4 texcoords
ps.1.1
texcoord t0
texcoord t1
texcoord t2
texcoord t3
```

```
; 1 texbem, a sample and a texcoord
ps.1.1
tex t0
texbem t1, t0
tex t2
texcoord t3
```

```
; 1 texbem and 2 texcoords
ps.1.1
tex t0
texbem t1, t0
texcoord t2
texcoord t3
```

```
; Two texbemls
ps.1.1
tex t0
texbeml t1, t0
tex t2
texbeml t3, t2
```

```
; One texbeml and two samples
ps.1.1
tex t0
texbeml t1, t0
tex t2
tex t3
```

```
; 1 texbeml, a sample and a texcoord
ps.1.1
tex t0
texbeml t1, t0
tex t2
texcoord t3
```

```
; 3x2 multiply and texcoord
tex t0
texm3x2pad t1, t0
texm3x2tex t2, t0
texcoord t3
```

```
; 1 texbeml and 2 texcoords
ps.1.1
tex t0
texbeml t1, t0
texcoord t2
texcoord t3
```

```
; 3x2 multiply & sample
ps.1.1
tex t0
texm3x2pad t1, t0
texm3x2tex t2, t0
tex t3
```

```
; 3x3 multiply
ps.1.1
tex t0
texm3x3pad t1, t0
texm3x3pad t2, t0
texm3x3tex t3, t0
```

```
; 3x3 matrix multiply and
; reflect constant vector
tex t0
texm3x3pad t1, t0
texm3x3pad t2, t0
texm3x3spec t3, t0
```

```
; 3x3 matrix multiply and
; reflect interpolated vector
tex t0
texm3x3pad t1, t0
texm3x3pad t2, t0
texm3x3vspec t3, t0
```

1.2 Shaders

- Still CISC like 1.1
- Same instruction count
- 4 new tex instructions
  - texreg2rgb
  - texdp3tex
  - texdp3
  - texm3x3
- 1 new argument modifier
  - .b replicate blue
  - Valid only in an alpha op
- 2 new arithmetic instructions
  - cmp – Conditionally chooses between s1 and s2 based on s0 compared with zero
  - dp4 – 4-element dot product

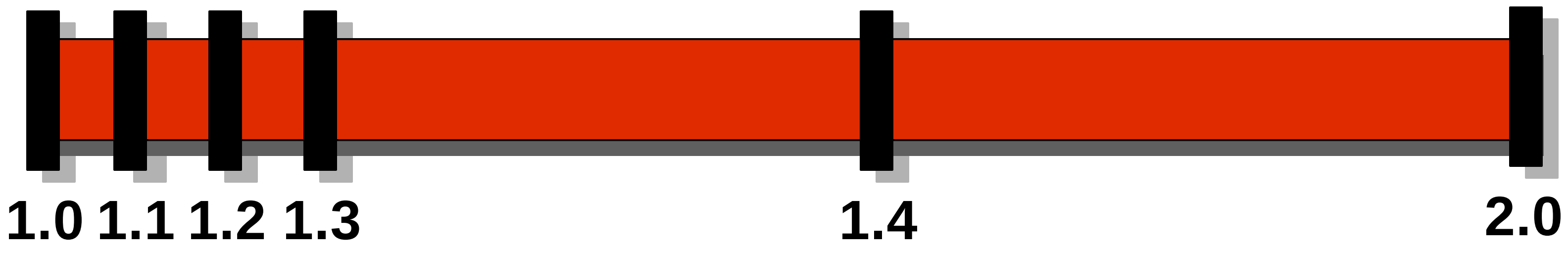
1.3 Shaders

- One additional tex instruction
  - texm3x2depth t2, t0
  - Performs two dot products to obtain z and w. The depth for the current pixel is set to z/w. If w == 0, the result is 1.0. If z > w the result is clamped to 1.0.
  - Example

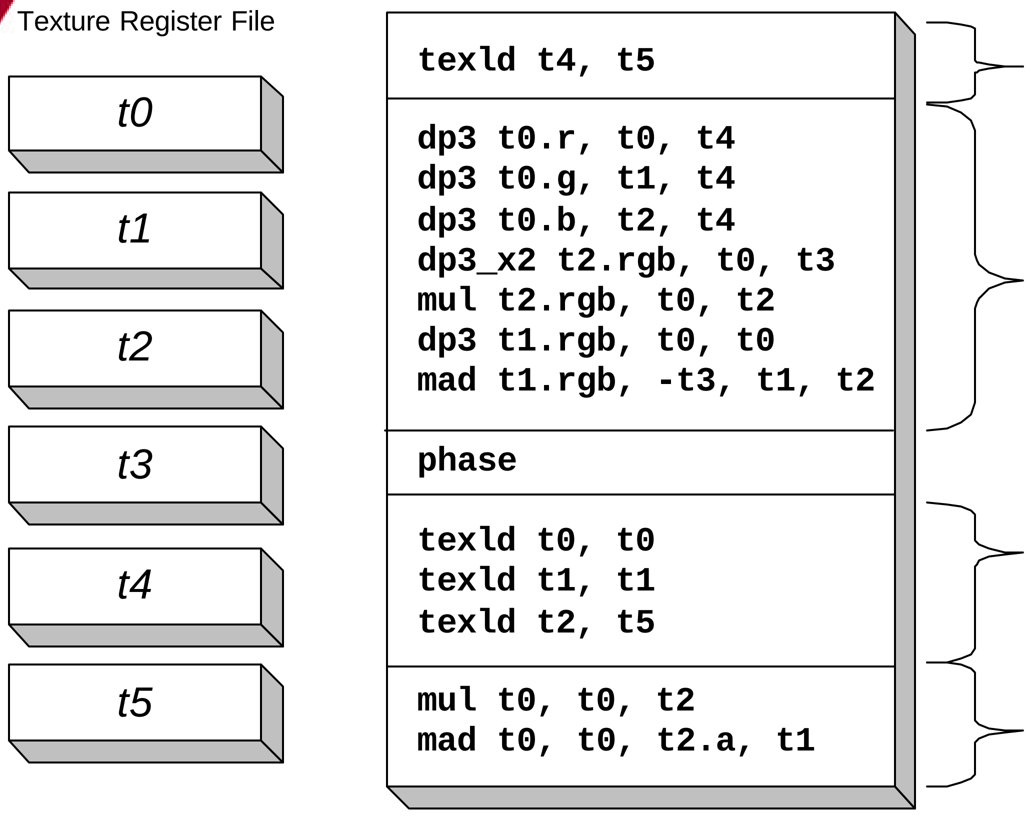
```
tex t0
texm3x2pad t1, t0
texm3x2depth t2, t0
```

1.4 Model

- Flexible, unified instruction set
  - Think up your own math and just do it rather than try to wedge your ideas into a fixed set of modes
- Flexible dependent texture fetching
- More textures
- More instructions
- High Precision
- Range of at least -8 to +8
- Well along the road to DX9



1.4 Pixel Shader Structure



```
texld t4, t5

dp3 t0.r, t0, t4
dp3 t0.g, t1, t4
dp3 t0.b, t2, t4
dp3_x2 t2.rgb, t0, t3
mul t2.rgb, t0, t2
dp3 t1.rgb, t0, t0
mad t1.rgb, -t3, t1, t2

phase

texld t0, t0
texld t1, t1
texld t2, t5

mul t0, t0, t2
mad t0, t0, t2.a, t1
```

- Optional Sampling
- Address Shader
  - Up to 8 instructions
- Optional Sampling
  - Can be dependent reads
- Color Shader
  - Up to 8 instructions

1.4 Texture Instructions

Mostly just data routing. Not ALU operations per se

- **texld**
  - **Samples data into a map**
- **texcrd**
  - **Moves high precision signed data into a temp register (r_n)**
  - **Higher precision than v0 or v1**
- **texkill**
  - **Kills pixels based on sign of register components**
  - **Fallback for parts that don’t have clip planes**
- **texdepth**
  - **Substitute value for this pixel’s z!**

texld vs tex

- Both cause a register to be filled with sampled data from a map
- tex
  - Unary op
- texld
  - Binary op
  - Context is associated with destination register
    - That means texture handle, filtering modes etc
  - Explicitly specifies dependent reads at the top of phase 2

texcrd vs texcoord

- texcoord clamps input to 0..1 range
  - Basically behaves like a color
  - You have to scale and bias into 0..1 in your vertex shader
  - Very annoying if you're also using the texm3x2 instructions, as you have already found
- texcrd does not clamp 0..1
  - Takes same input range as texm3x2 type instructions
  - Retains pixel pipeline's native precision which is higher than colors

texkill

- Another way to kill pixels
- If you're just doing a clip plane, use a clip plane
  - As a fallback, use texkill for chips that don't support user clip planes
- Pixels are killed based on the sign of the components of registers

texdepth

- Substitute a register value for z
- Imaged based rendering
- Depth sprites

1.4 Pixel Shader ALU Instructions

```
• add   d, s0, s1          // sum
• sub   d, s0, s1          // difference
• mul   d, s0, s1          // modulate
• mad   d, s0, s1, s2      // s0 * s1 + s2
• lrp   d, s0, s1, s2      // s2 + s0*(s1-s2)
• mov   d, s0              // d = s0
• cnd   d, s0, s1, s2      // d = (s2 > 0.5) ? s0 : s1
• cmp   d, s0, s1, s2      // d = (s2 >= 0) ? s0 : s1
• dp3   d, s0, s1          // s0·s1 replicated to d.rgba
• dp4   d, s0, s1          // s0·s1 replicated to d.rgba
• bem   d, s0, s1, s2      // Macro similar to texbem
```

Argument Modifiers

- Negate -r_n
- Invert 1-r_n
    - Unsigned value in source is required
- Bias (_bias)
    - Shifts value down by ½
- Scale by 2 (_x2)
    - Scales argument by 2
- Signed Scaling (_bx2)
    - _bias followed by _x2
    - Shifts value down and scales data by 2 like the implicit behavior of `D3DTOP_DOTPRODUCT3` in `SetTSS()`
- Channel replication
    - r_n.r, r_n.g, r_n.b or r_n.a
    - Useful for extracting scalars out of registers
    - Not just in alpha instructions like the .b in ps.1.2

Instruction Modifiers

- `_x2` - Multiply result by 2
- `_x4` - Multiply result by 4
- `_x8` - Multiply result by 8
- `_d2` - Divide result by 2
- `_d4` - Divide result by 4
- `_d8` - Divide result by 8
- `_sat` - Saturate result to 0..1

- `_sat` may be used alone or combined with one of the other modifiers. i.e. `mad_d8_sat`

Write Masks

- Any channels of the destination register may be masked during the write of the result
- Useful for computing different components of a texture coordinate for a dependent read
- Example:

```
dp3 r0.r, t0, t4
mov r0.g, t0.a
```

- We'll show more examples of this

Range and Precision

- ps.1.4 range is at least -8 to +8
  - Determine with `MaxPixelShaderValue`
- Pay attention to precision when doing operations that may cause errors to build up
- Conversely, use your range when you need it rather than scale down and lose precision. Filter kernel intermediate results are one case.
- Your texture coordinate interpolators are your high precision data sources. Use them.
- Sampling an 8 bit per channel texture (to normalize a vector, for example) gives you back a low precision result

cnd and cmp

- cnd d, s0, s1, s2;   d = (s0 > 0.5) ? s1 : s2
  - Conditionally chooses between s1 and s2
  - In DirectX 8.1, cnd can now perform a component wise comparison of s0 to 0.5 in order to select s1 or s2's component:
    - For s0 = [0.4, 0.5, 0.51, -5.6], d = [s2.r, s2.g, s1.b, s2.a]

- cmp d, s0, s1, s2;   d = (s0 >= 0) ? s1 : s2
  - Conditionally chooses between s1 and s2
  - For s0 = [-0.4, 0.0, 5.0, -6.3], d = [s2.r, s1.g, s1.b, s2.a]
  - New Instruction in DirectX 8.1
  - Useful for absolute value: cmp d, s0, s0, -s0

Examples: Image Filters

- Use on 2D images in general
- Use as post processing pass over 3D scenes rendered into textures
    - Luminance filter for Black and White effect
    - Edge filters for non-photorealistic rendering
    - Glare filters for soft look (see *Fiat Lux* by Debevec)
    - Opportunity for you to customize your look
- Rendering to textures is fundamental. You need to get over your reluctance to render into textures.
- Becomes especially interesting when we get to high dynamic range

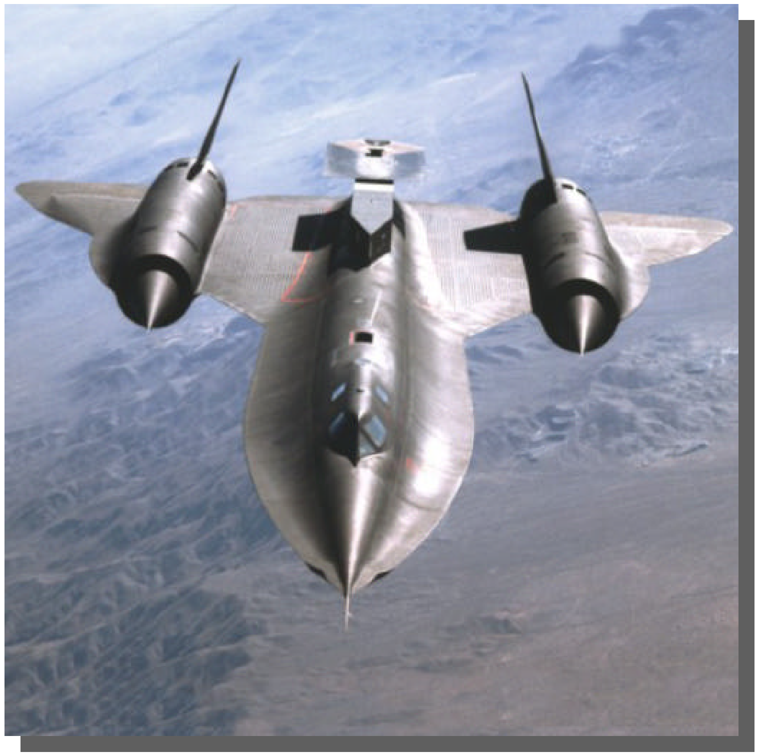
Luminance Filter

- Different RGB recipes give different looks
  - Black and White TV
  - Black and White film
  - Sepia
  - Run through arbitrary transfer function using a dependent read for “heat signature”
- A common recipe is Lum = .3r + .59g + .11b

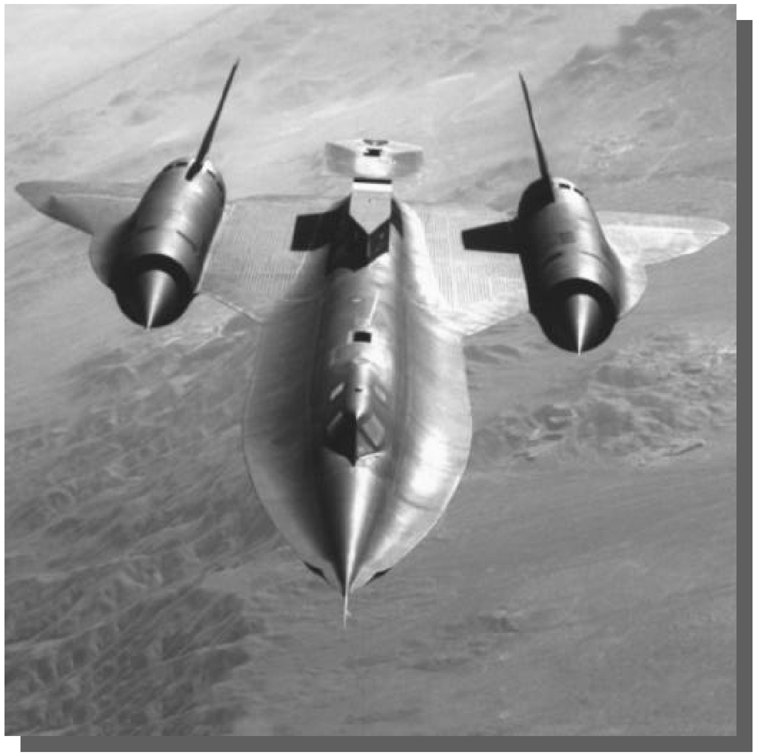
```
ps.1.4
def c0, 0.30f, 0.59f, 0.11f, 1.0f
texld r0, t0
dp3 r0, r0, c0
```

Luminance Filter

Original Image



Luminance Image



Multitap Filters

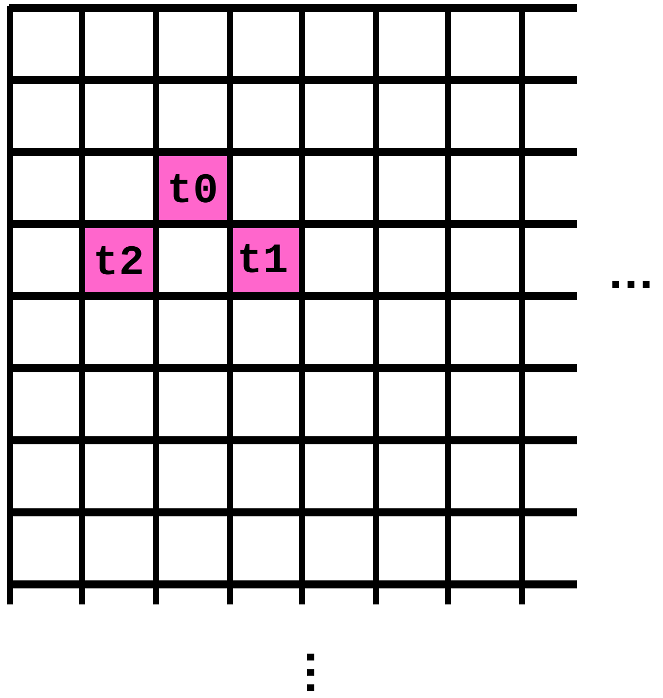
- Effectively code filter kernels right into the pixel shader
- Pre offset taps with texture coordinates
  - For traditional image processing, offsets are a function of image/texture dimensions and point sampling is used
  - Or compose complex filter kernels from multiple bilinear kernels

Edge Detection Filter

- Roberts Cross Gradient Filters

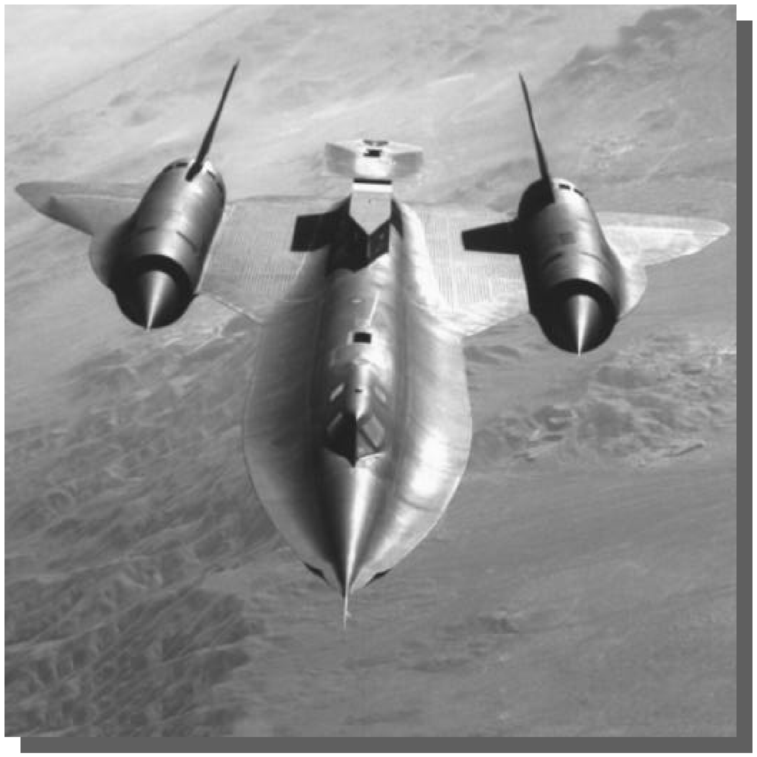
```
ps.1.4
texld r0, t0 // Center Tap
texld r1, t1 // Down & Right
texld r2, t2 // Down & Left
add r1, r0, -r1
add r2, r0, -r2
cmp r1, r1, r1, -r1
cmp r2, r2, r2, -r2
add_x8 r0, r1, r2
```

$$\begin{array}{|c|c|}\hline 1 & 0 \\ \hline 0 & -1 \\ \hline\end{array} \qquad \begin{array}{|c|c|}\hline 0 & 1 \\ \hline -1 & 0 \\ \hline\end{array}$$

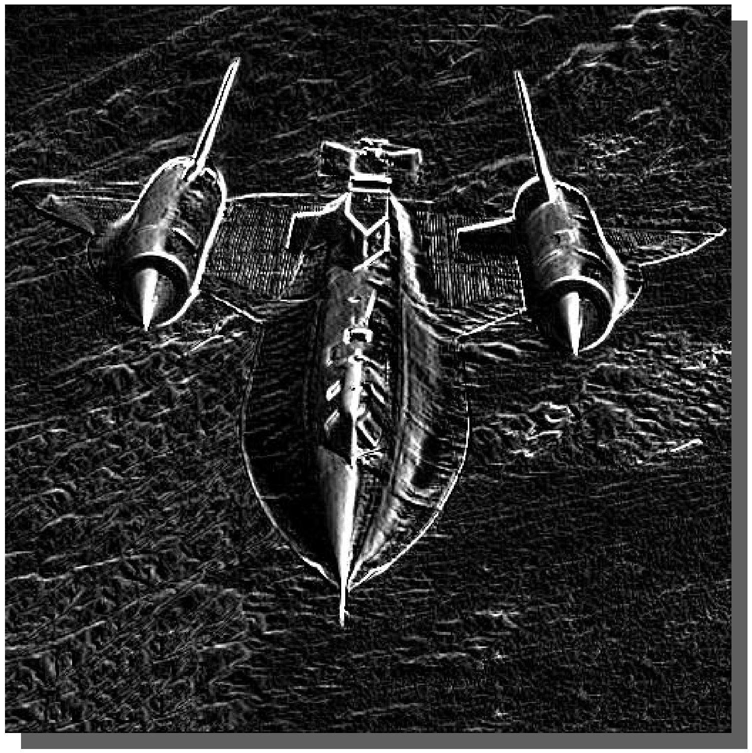


Gradient Filter

Original Image



8 x Gradient Magnitude

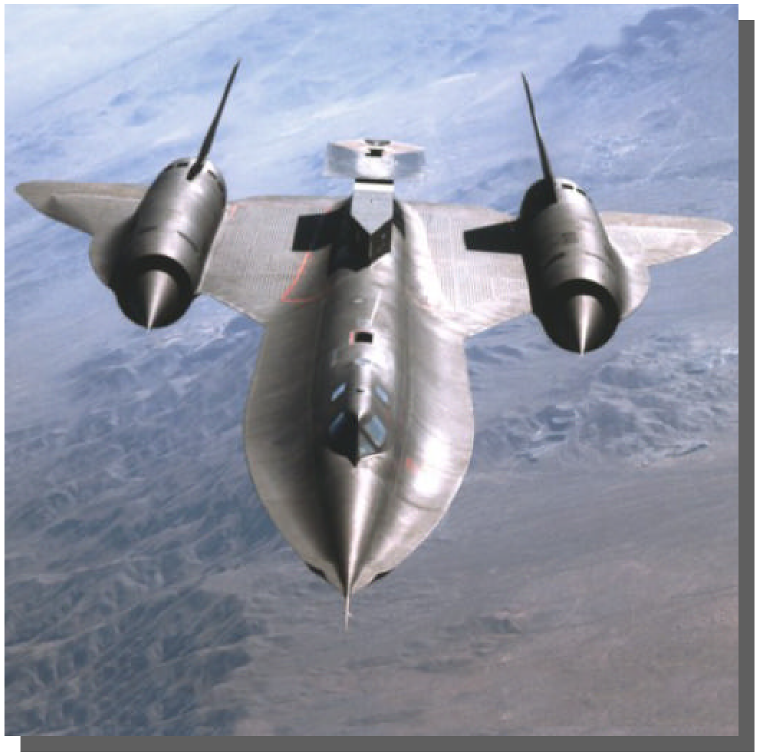


Gradient from Color

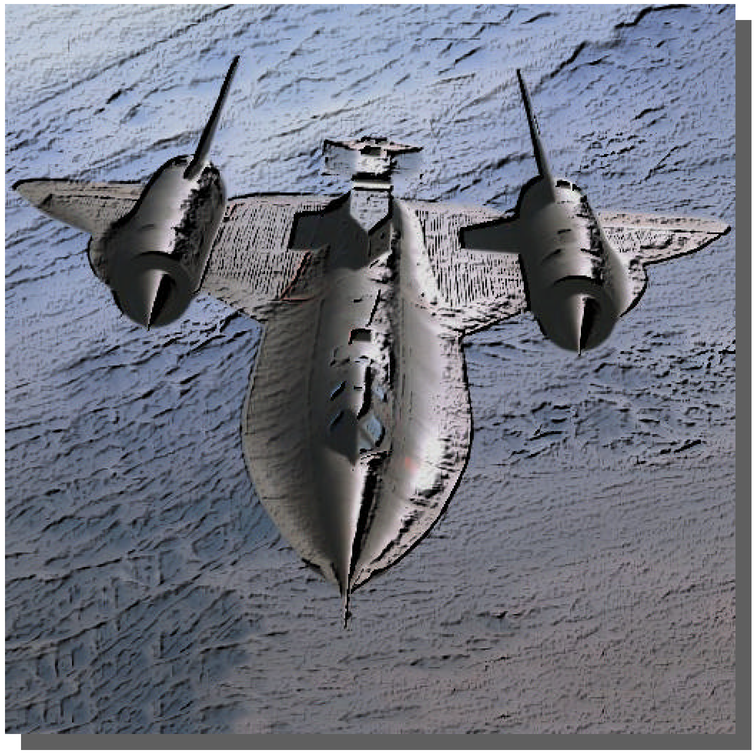
```
ps.1.4
def c0, 0.30f, 0.59f, 0.11f, 1.0f
texld r0, t0        // Center Tap
texld r1, t1        // Down & Right
texld r2, t2        // Down & Left
dp3 r3, r0, c0      // Compute Luminances
dp3 r1, r1, c0
dp3 r2, r1, c0
add r1, r3, -r1     // Compute Differences
add r2, r3, -r2
cmp r1, r1, r1, -r1 // Absolute Values
cmp r2, r2, r2, -r2
add_x8 r1, r1, r2
phase
mul r0.rgb, r0, 1-r1 // Composite with original rgb
+mov r0.a, c0.a
```

Gradient from Color

Original Image

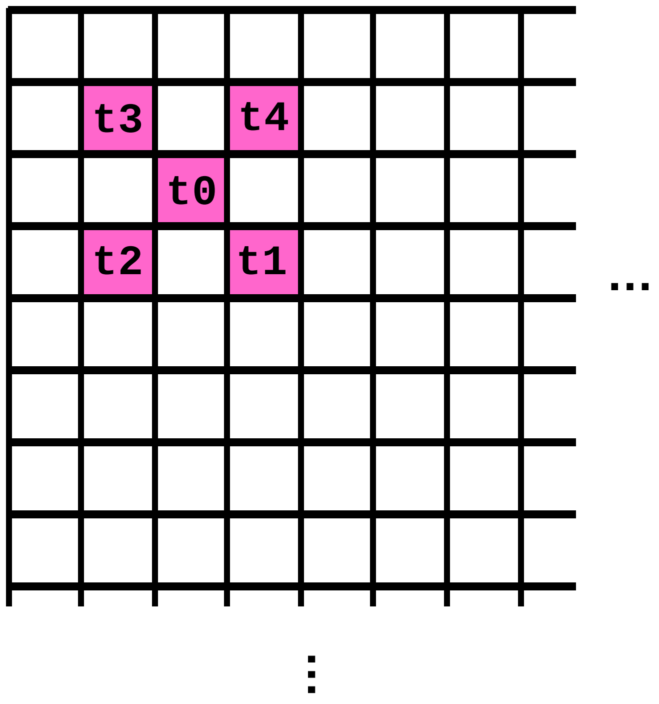


Original x Inverted Gradient Magnitude



Five Tap Blur Filter

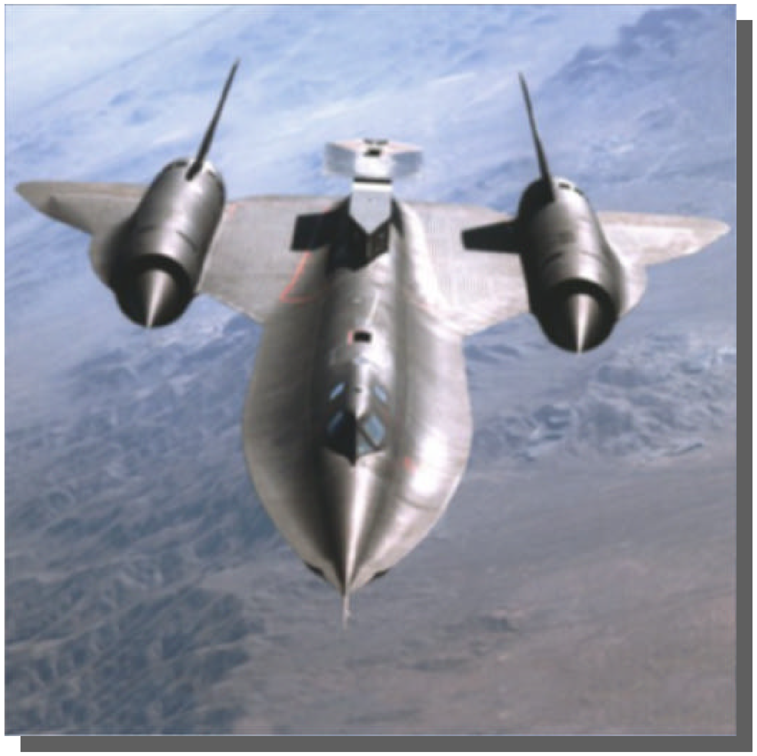
```
ps.1.4
def c0, 0.2f, 0.2f, 0.2f, 1.0f
texld r0, t0 // Center Tap
texld r1, t1 // Down & Right
texld r2, t2 // Down & Left
texld r3, t3 // Up & Left
texld r4, t4 // Up & Right
add r0, r0, r1
add r2, r2, r3
add r0, r0, r2
add r0, r0, r4
mul r0, r0, c0
```



Five Tap Blur Filter

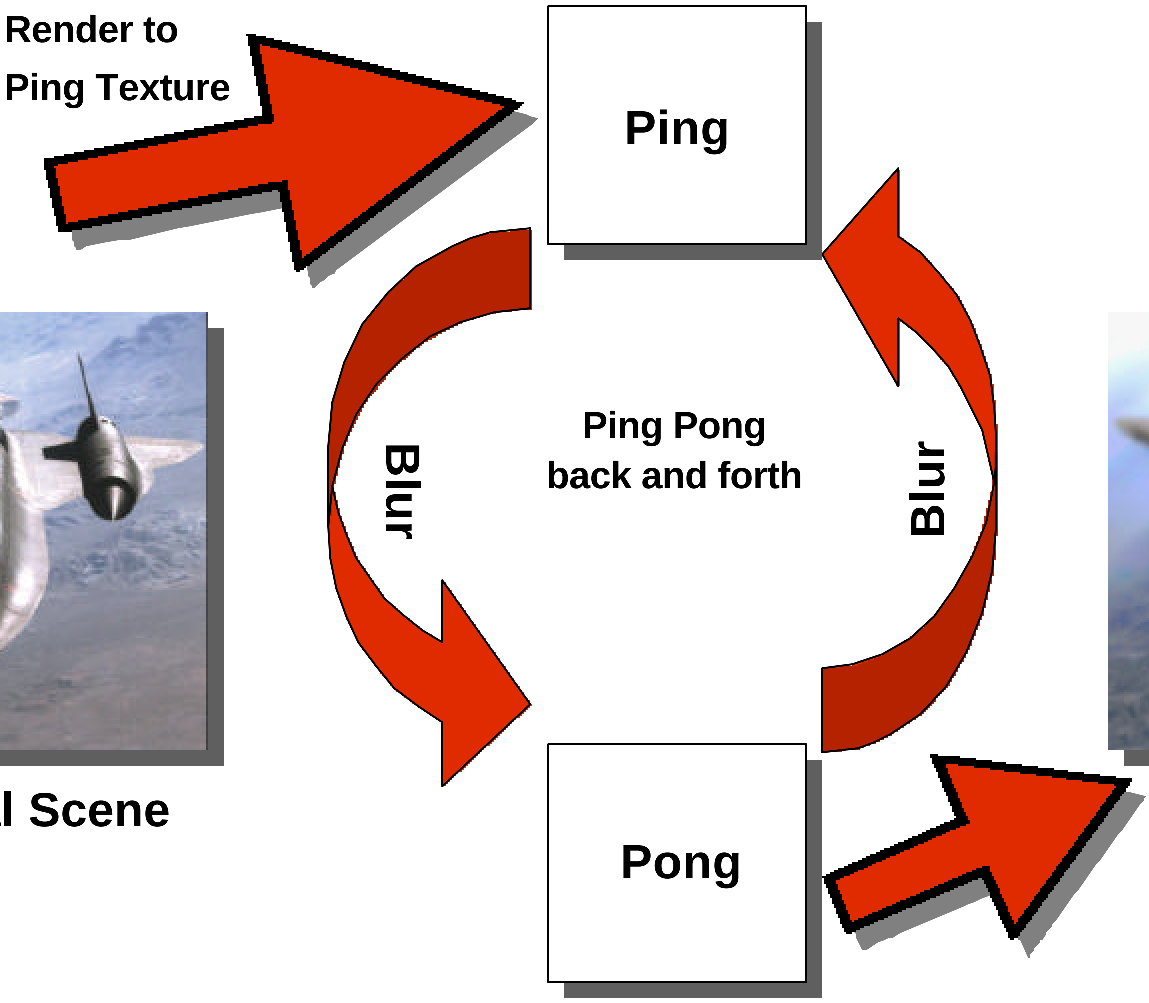
Original Image

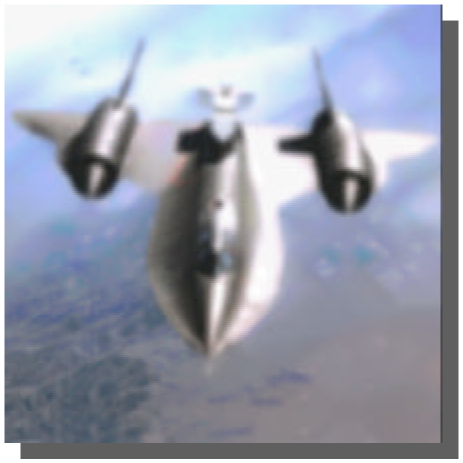
Blurred Image



Ping-Pong Blur Filter

Original Scene





Blurred Image

Glare Filter

- Composite thresholded blur image back on to original scene

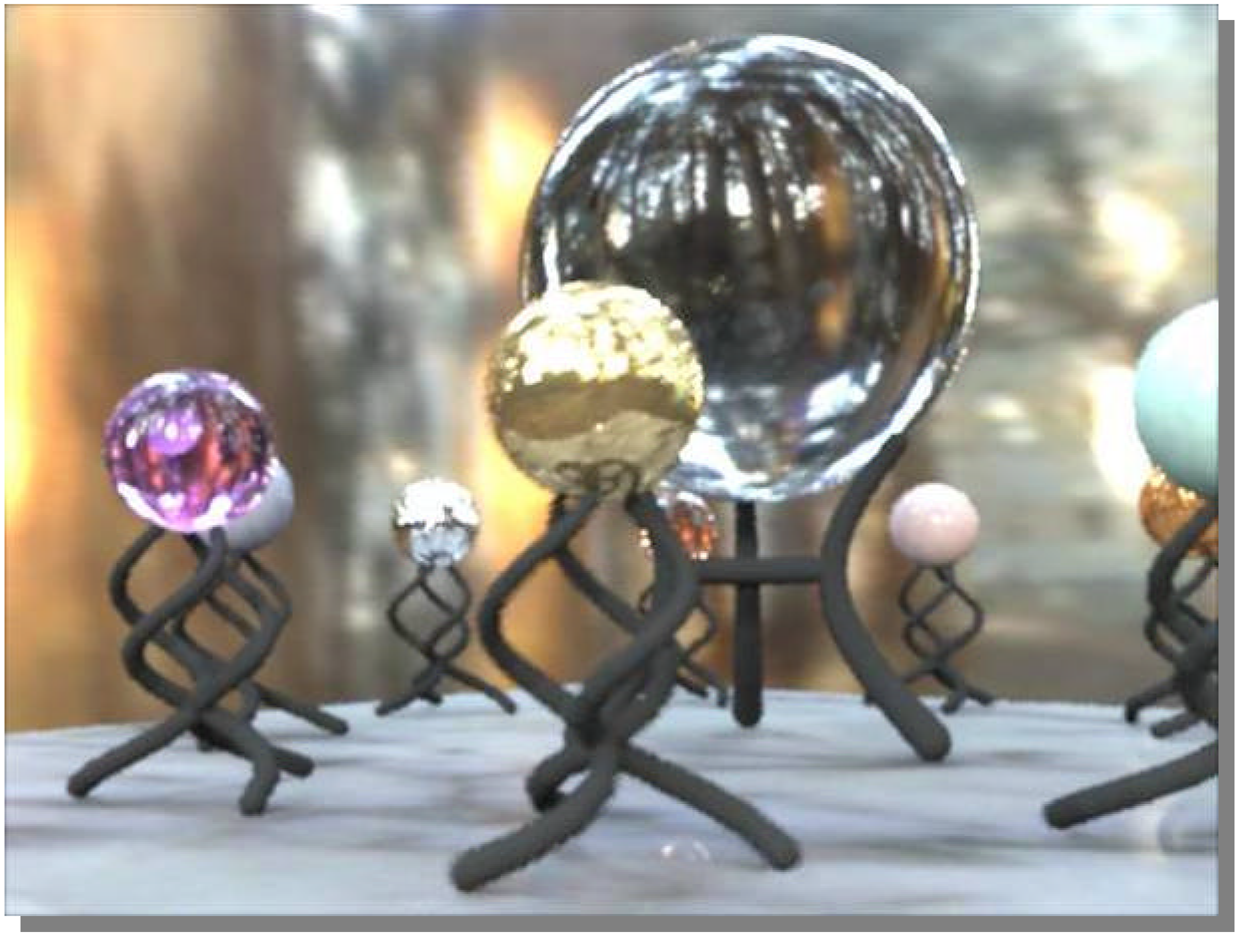


Image from Rendering with Natural Light by Debevec

Glare Filter

- Composite thresholded image back on to original scene

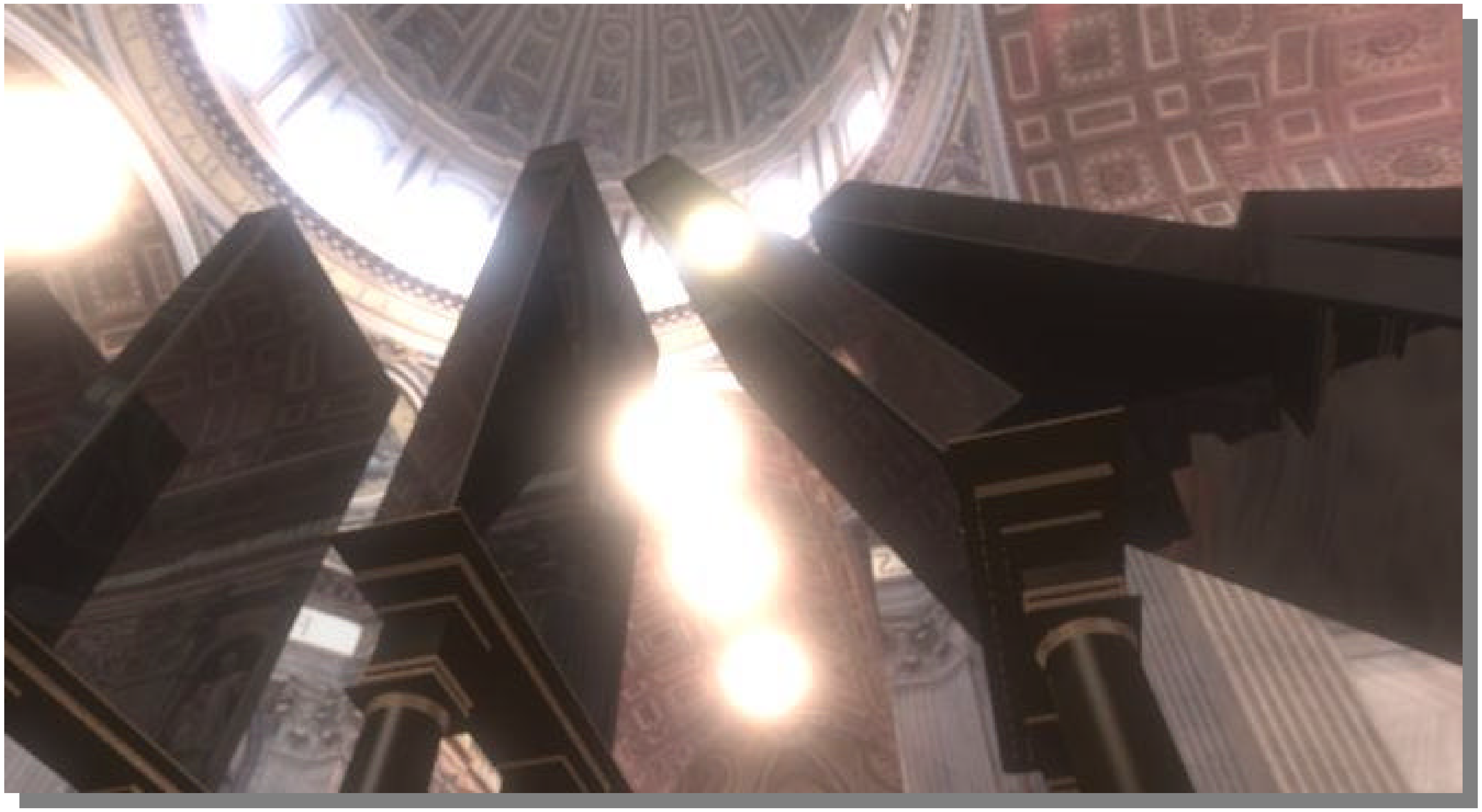
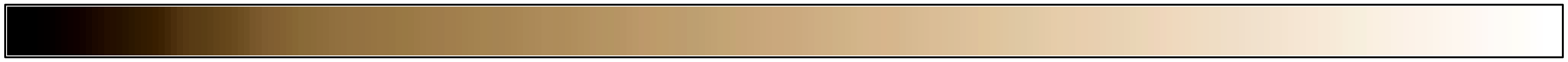


Image from Fiat Lux by Debevec

Sepia Transfer Function

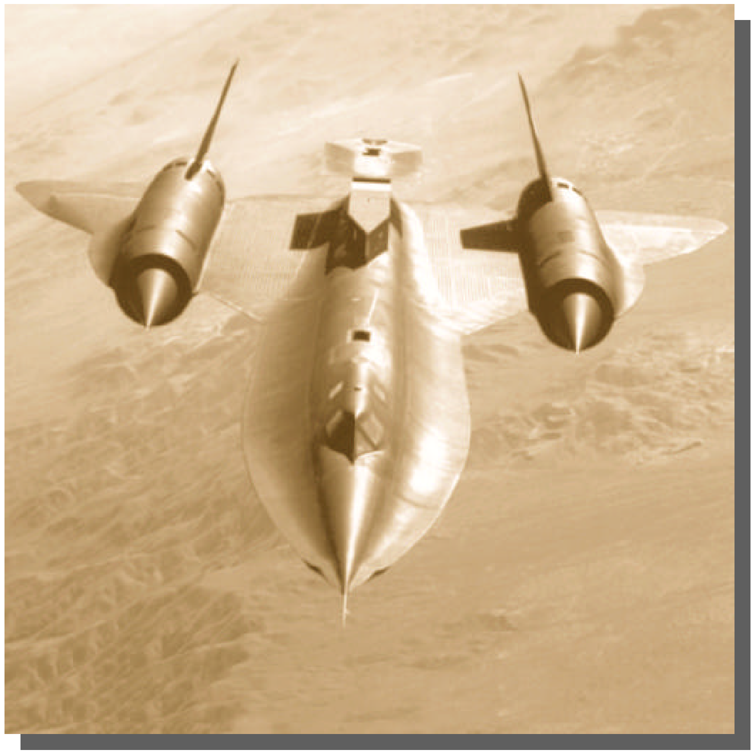
```
ps.1.4
def c0, 0.30f, 0.59f, 0.11f, 1.0f
texld r0, t0
dp3 r0, r0, c0  // Convert to Luminance
phase
texld r5, r0    // Dependent read
mov r0, r5
```


1D Luminance to Sepia map

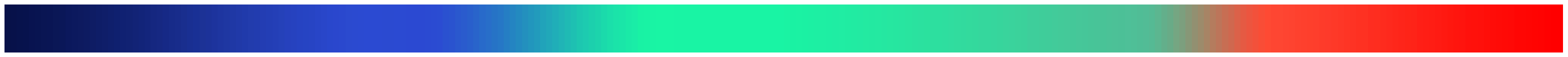
Sepia Transfer Function

Original Image

Sepia Tone Image



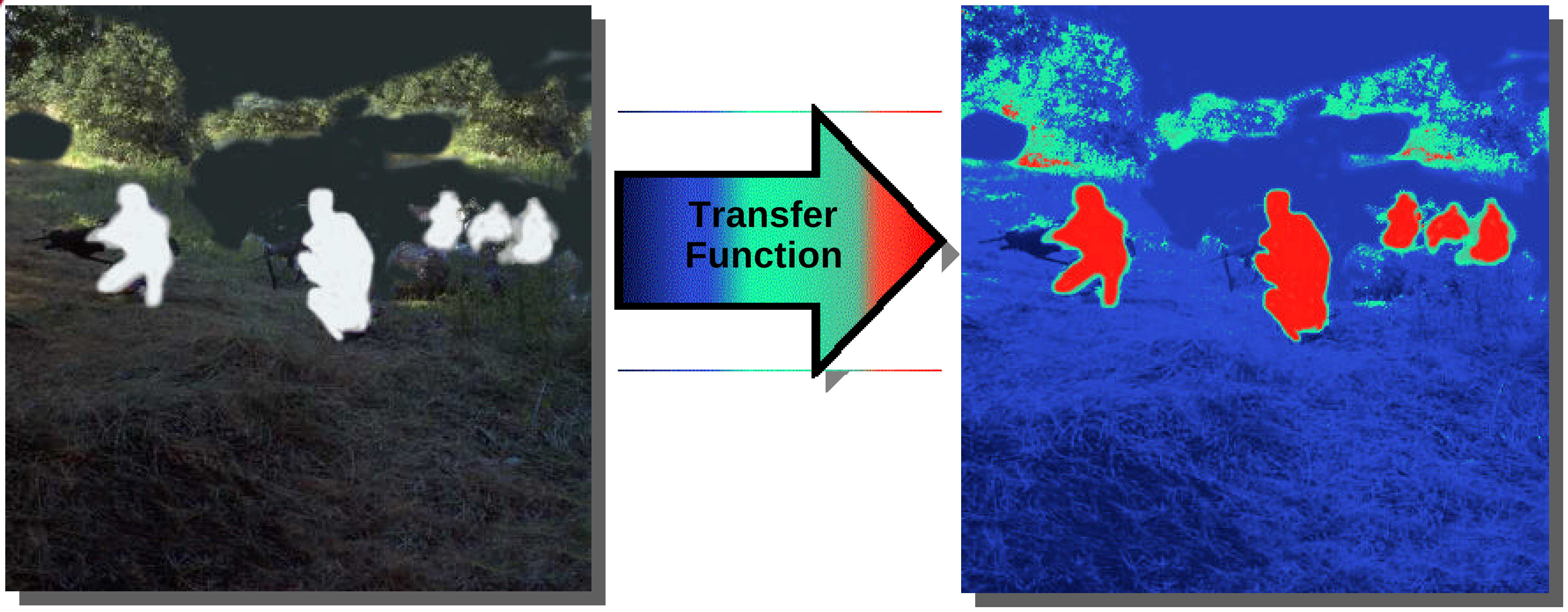
Heat Signature



1D Heat Signature Map

Heat Transfer Function

Heat input image **Heat Signature Image**

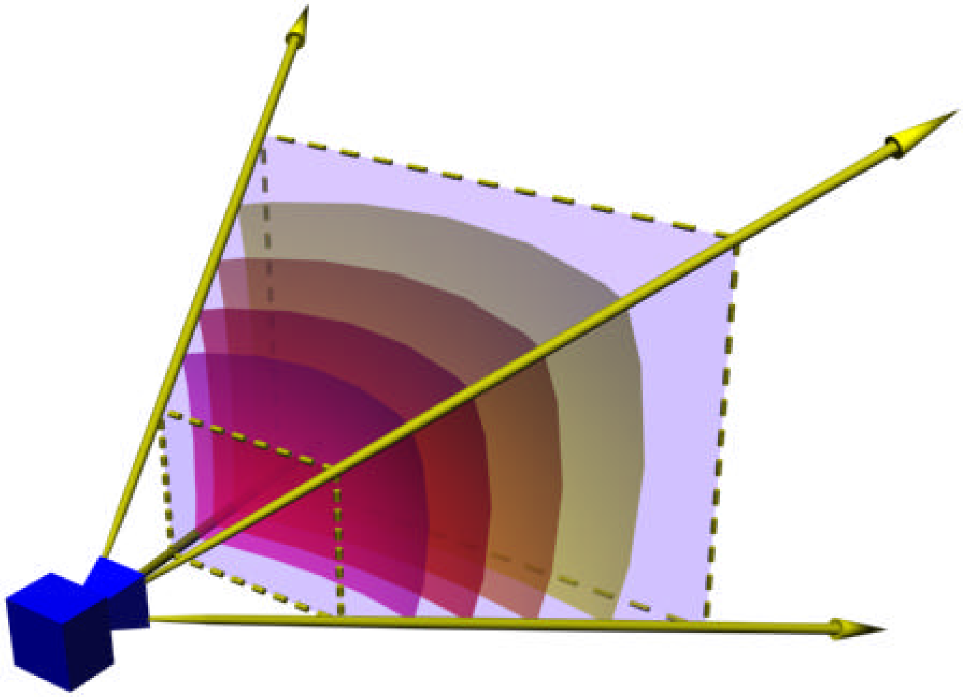


Volume Visualization

- The visualization community starts with data that is inherently volumetric and often scalar, as it is acquired from some 3D medical imaging modality
- As such, no polygonal representation exists and there is a need to “reconstruct” projections of the data through direct volume rendering
- Last year, we demoed this on RADEON™ using volume textures on DirectX 8.0
- One major area of activity in the visualization community is coming up with methods for using ***transfer functions***, ideally dynamic ones, that map the (often scalar) data to some curve through color space
- The 1D sepia and heat signature maps just shown are examples of transfer functions

Volume Visualization

- On consumer cards like RADEON™, volume rendering is done by compositing a series of “shells” in camera space which intersect the volume to be visualized
- A texture matrix is used to texture map the shells as they slice through a volume texture

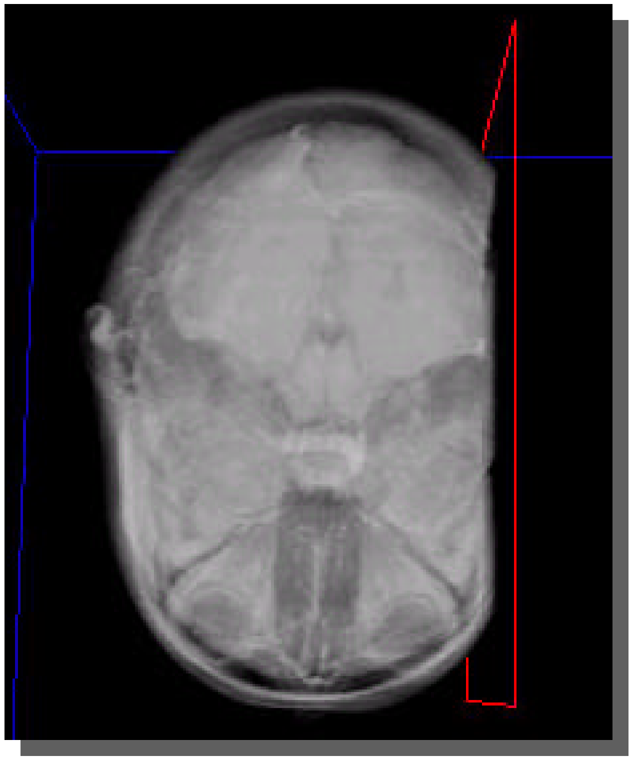

View frustum and VolViz Shells

Dynamic Transfer Functions

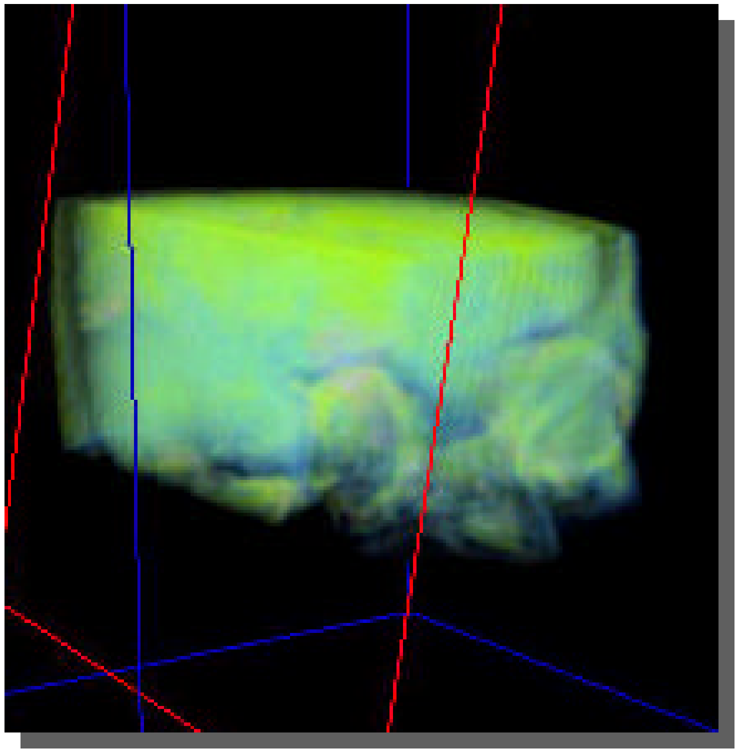
- With 1.4 pixel shaders, it is very natural to sample data from a 3D texture map and apply a transfer function via a dependent read
- Transfer functions are usually 1D and are very cheap to update interactively

Dynamic Transfer Functions

Scalar Data



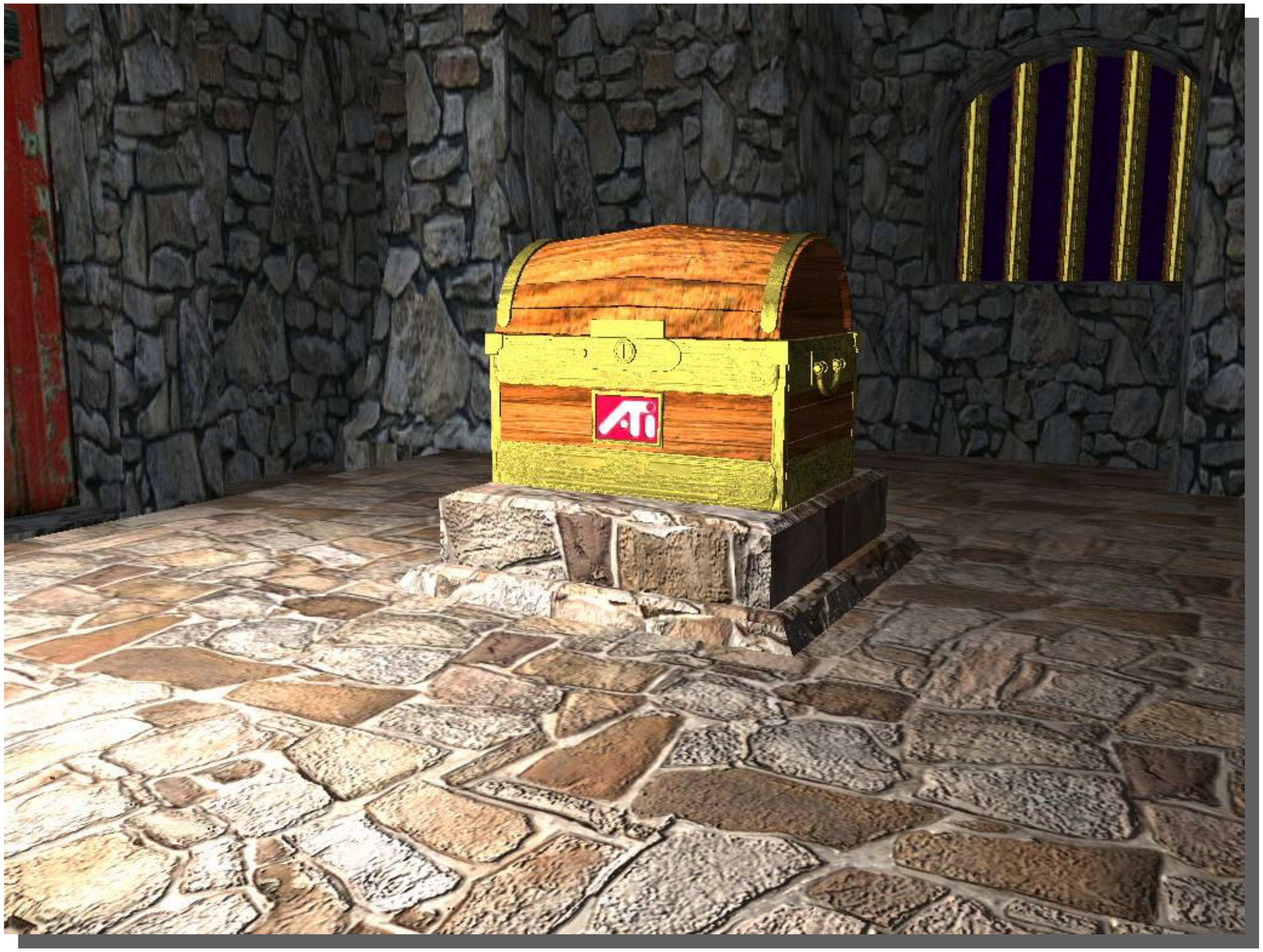
Transfer Function Applied



More Examples: Lighting

Four Diffuse Per-Pixel Lights in one Pass

Per-pixel N·L and Attenuation



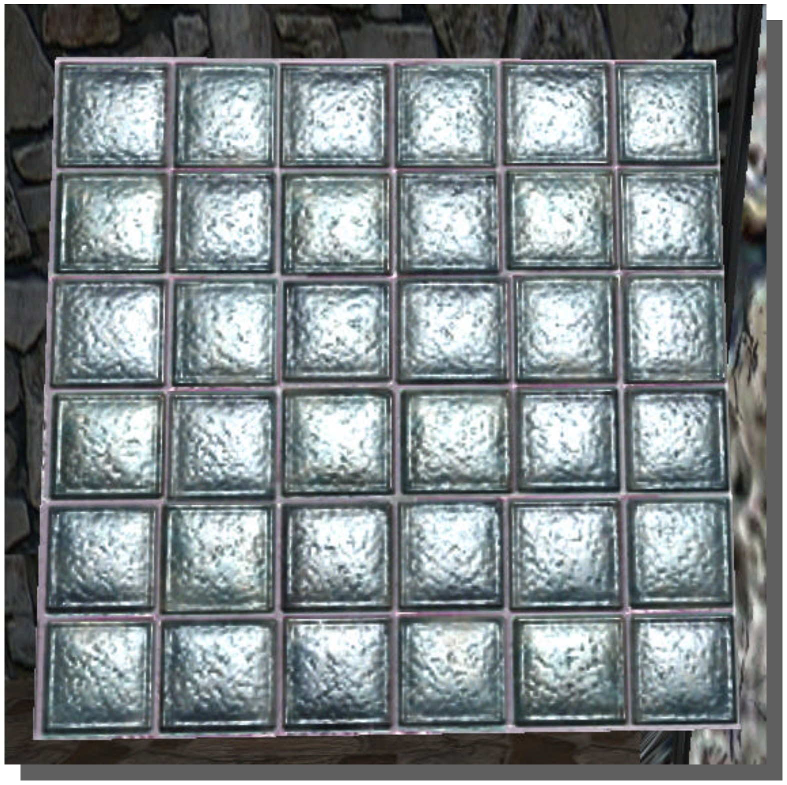
Per-pixel N·L and Attenuation

```
texld  r1, t0      ; Normal
texld  r2, t1      ; Cubic Normalized Tangent Space Light Direction
texcrd r3.rgb, t2  ; World Space Light Direction.
                   ; Unit length is the light's range.

dp3_sat r1.rgb, r1_bx2, r2_bx2  ; N·L
dp3     r3.rgb, r3, r3          ; (World Space Light Distance)^2

phase

texld  r0, t0                   ; Base
texld  r3, r3                   ; Light Falloff Function

mul_x2    r4.rgb, r1, r3        ; falloff * (N·L)
add       r4.rgb, r4, c7        ; += ambient
mul       r0.rgb, r0, r4        ; base * (ambient + (falloff*N·L))
```

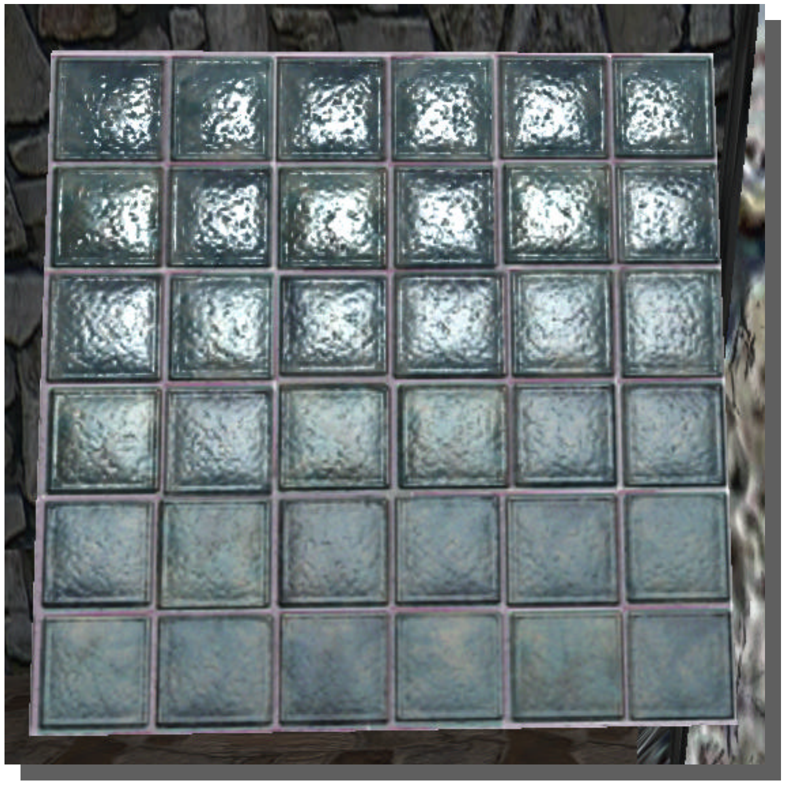
Dependent Read $\longrightarrow$

Variable Specular Power

Constant specular power

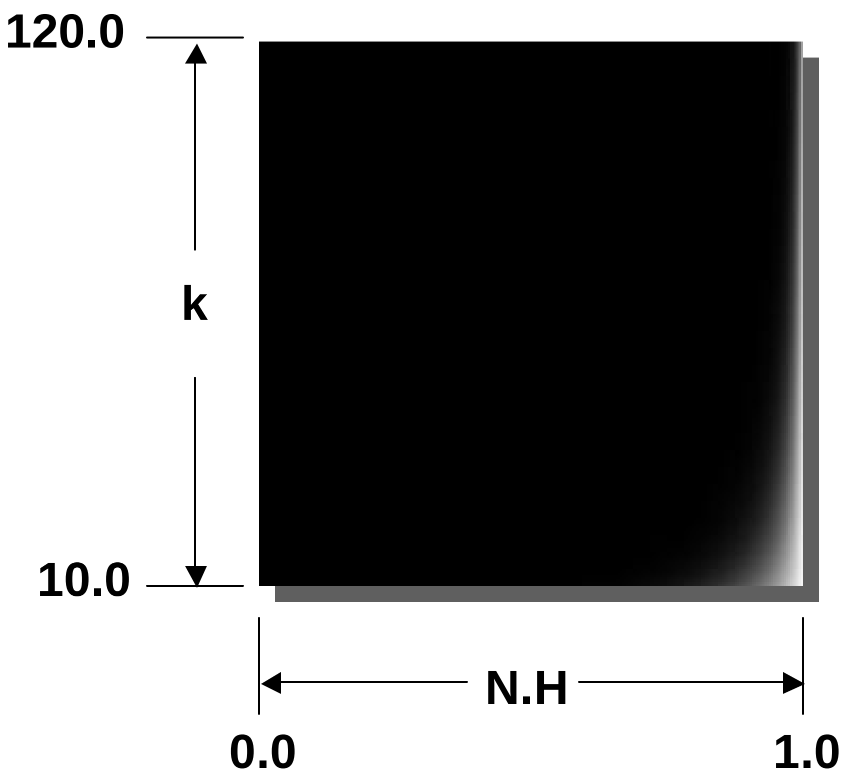


Variable specular power



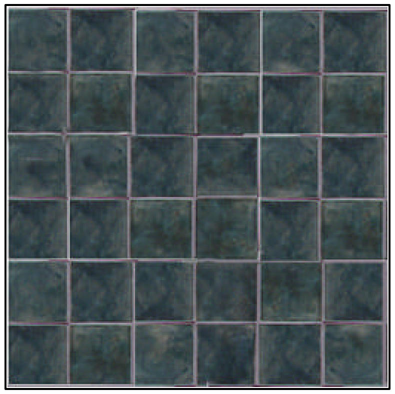
Variable Specular Power

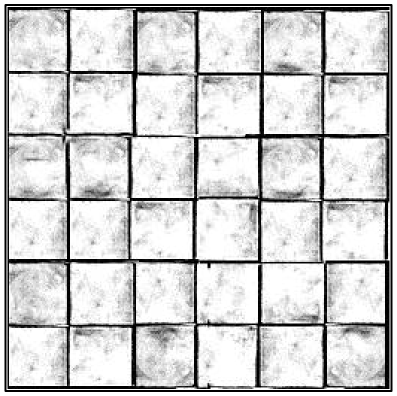
Per-pixel $(N \cdot H)^k$ with per-pixel variation of k

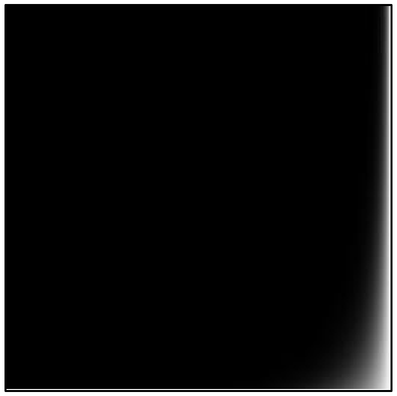


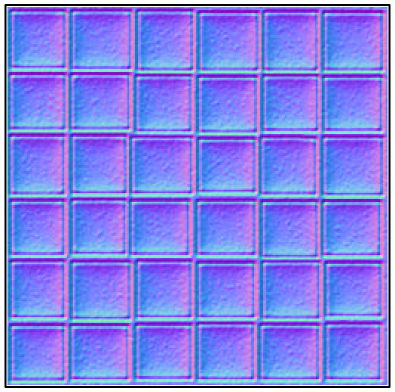
- Base map with albedo in RGB and gloss in alpha
- Normal map with xyz in RGB and k in alpha
- N·H ´ k map
- Should also be able to apply a scale and a bias to the map and in the pixel shader to make better use of the resolution

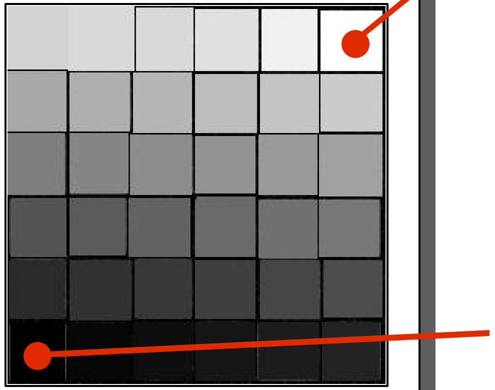
Maps for per-pixel variation of k shader


Albedo in RGB


Gloss in alpha


N·H ´ k map


Normals in RGB



k in alpha

Variable Specular Power

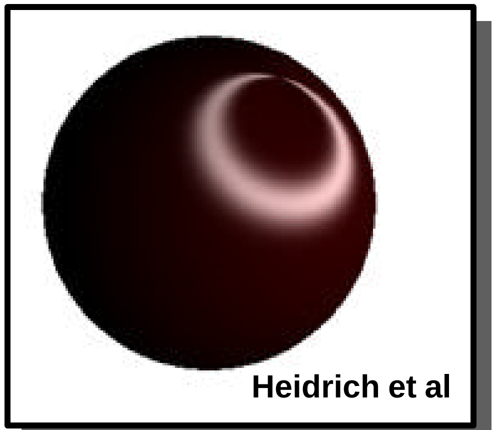
```
ps.1.4
texld  r1, t0       ; Normal
texld  r2, t1       ; Normalized Tangent Space L vector
texcrd r3.rgb, t2   ; Tangent Space Halfangle vector

dp3_sat r5.xyz, r1_bx2, r2_bx2  ; N·L
dp3_sat r2.xyz, r1_bx2, r3      ; N·H
mov     r2.y, r1.a              ; K = Specular Exponent
phase
texld  r0, t0   ; Base
texld  r3, r2   ; Specular NH×K map
add       r4.rgb, r5, c7     ; += ambient
mul       r0.rgb, r0, r4     ; base * (ambient + N·L))
+mul_x2   r0.a, r0.a, r3.a   ; Gloss map * specular
add       r0.rgb, r0, r0.a   ; (base*(ambient + N·L)) +
                             ; (Gloss*Highlight)
```

Dependent Read →

Anisotropic lighting

- We know how to light lines and anisotropic materials by doing two dot products and using the results to look up the non-linear parts in a 2D texture/function (Banks, Zöckler, Heidrich)
- This was done per-vertex using the texture matrix
- With per-pixel dot products and dependent texture reads, we can now do this math per-pixel and specify the direction of anisotropy in a map.

Zöckler et al

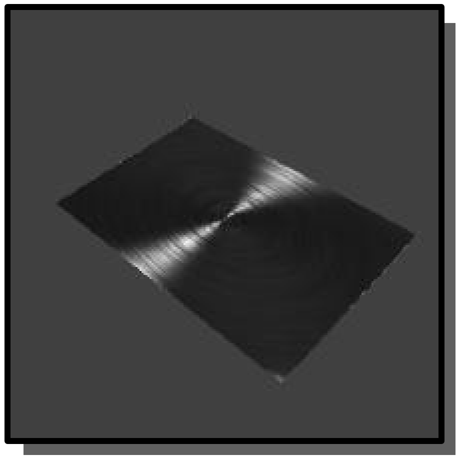


Per-pixel anisotropic lighting

- This technique involves computing the following for diffuse and specular illumination:

Diffuse: $\sqrt{1 - (L \cdot T)^2}$

Specular: $\sqrt{1 - (L \cdot T)^2}\,\sqrt{1 - (V \cdot T)^2} - (L \cdot T)(V \cdot T)$

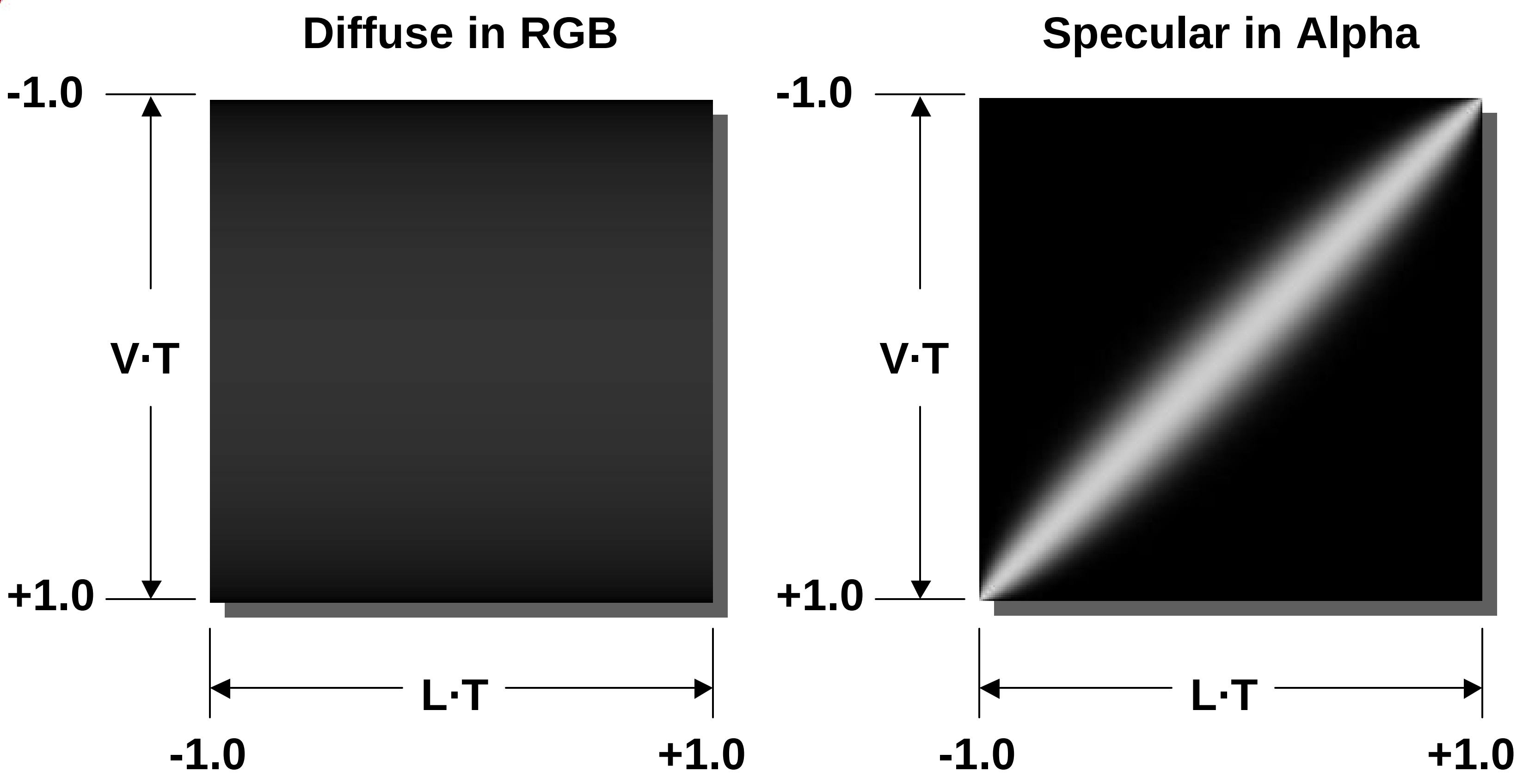


- These two dot products can be computed per-pixel with the `texm3x2*` instructions or just two dp3s in 1.4
- Use this 2D tex coord to index into special map to evaluate above functions
- At GDC 2001, we showed this limited to per-pixel tangents in the plane of the polygon
- Here, we orthogonalize the tangents wrt the per-pixel normal inside the pixel shader

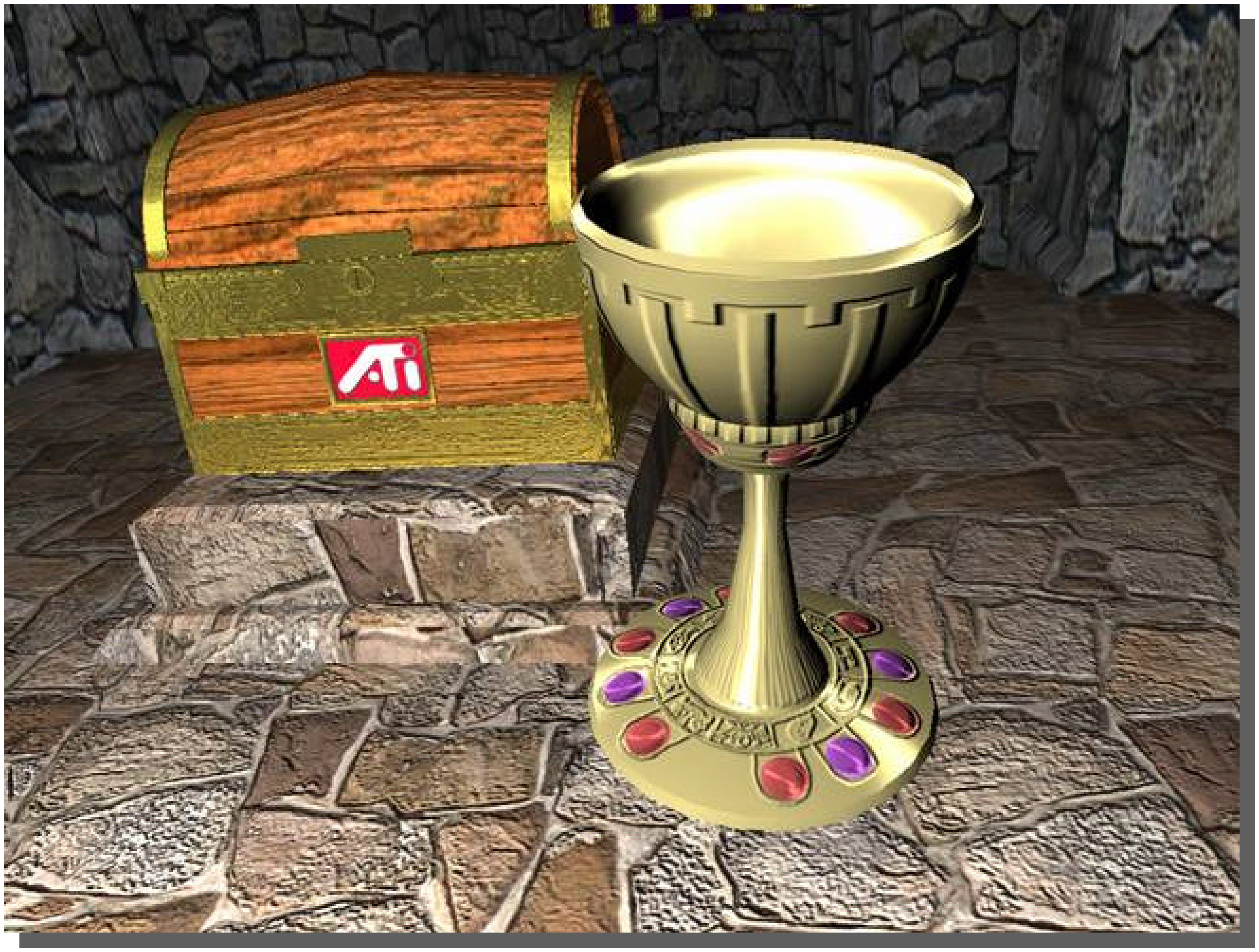
Per-pixel anisotropic lighting

- Use traditional normal map, whose normals are in tangent space
- Use tangent map
- Or use an interpolated tangent and orthogonalize it per-pixel
- Interpolate V and L in tangent space and compute coordinates into function lookup table per pixel.

Per-pixel anisotropic lighting



Anisotropic Lighting Example: Brushed Metal



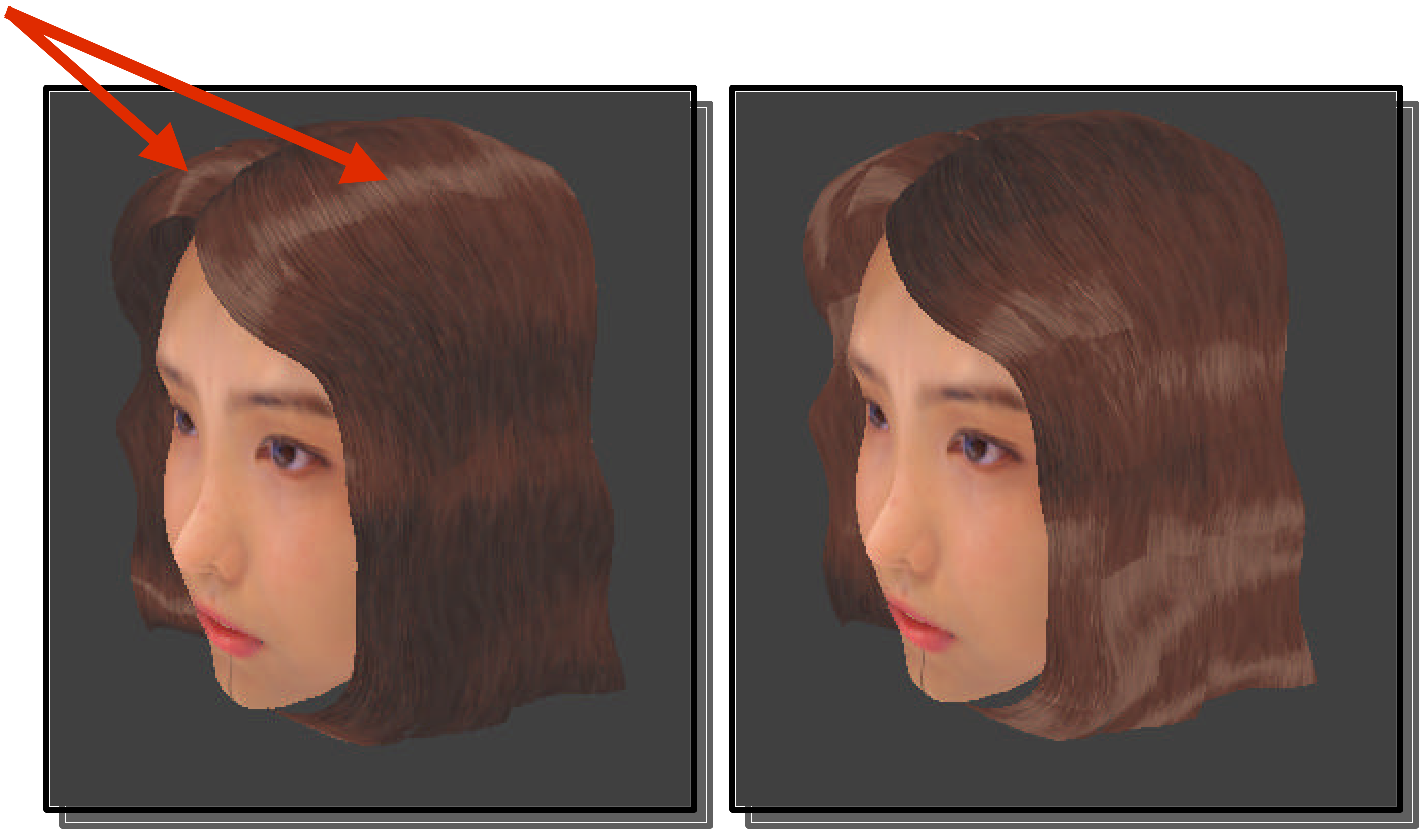
Bumped Anisotropic Lighting

```
ps.1.4
def c0, 0.5f, 0.5f, 0.0f, 1.0f
texld  r0, t0                 ; Contains direction of anisotropy in tangent space
texcrd r2.rgb, t1             ; light vector
texcrd r3.rgb, t2             ; view vector
texld  r4, t0                 ; normal map

; Perturb anisotropy lighting direction by normal
dp3 r1.xyz, r0_bx2, r4_bx2     ; Aniso.Normal
mad r0.xyz, r4_bx2, r1, r0_bx2 ; Aniso - N(Aniso.Normal)

; Calculate A.View and A.Light for looking up into function map
dp3 r5.x, r2, r0               ; Perform second row of matrix multiply
dp3 r5.yz, r3, r0              ; Perform second row of matrix multiply to get a
                               ; 3-vector with which to sample texture 3, which is
                               ; a look-up table for aniso lighting
mad r5.rg, r5, c0, c0          ; Scale and bias for lookup

; Diffuse Light Term
dp3_sat r4.rgb, r4_bx2, r2 ; N.L
phase
texld  r2, r5                  ; Anisotropic lighting function lookup
texld  r3, t0                  ; gloss map
mul r4.rgb, r3, r4.b           ;basemap * N.L
mad r0.rgb, r3, r2.a, r4       ;+= glossmap * specular
mad r0.rgb, r3, c7, r0         ;+= ambient * basemap
```

Anisotropic Lighting Example: Human Hair

- Direction of anisotropy map is used to light the hair

Highlights computed in pixel shader

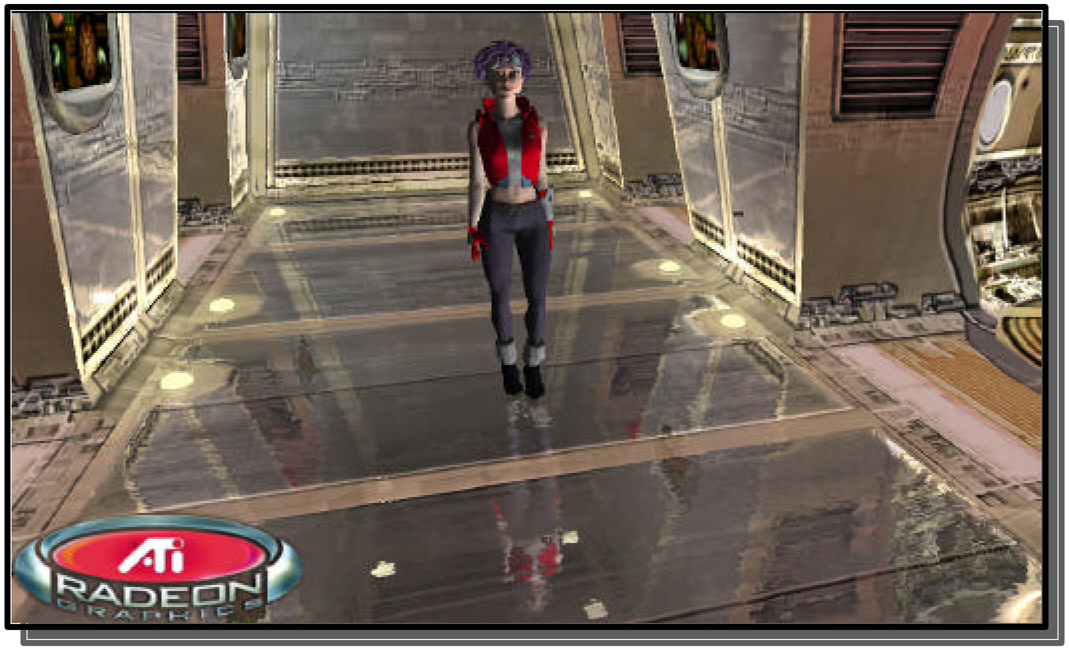


Bumpy Environment Mapping

- Several flavors of this
  - DX6-style EMBM
    - Must work with projective texturing to be useful
  - Could do DX6-style but with interpolated 2x2 matrix
  - But the really cool one is per-pixel doing a 3x3 multiply to transform fetched normal into cube map space
- All still useful and valid in different circumstances.
- Can now do superposition of the perturbation maps for constructive / destructive interference of waveforms
- Really, the distinctions become irrelevant, as this all just degenerates into “dependent texture reads” and the app makes the tradeoffs between what it determines is “correct” for a given effect

Traditional EMBM

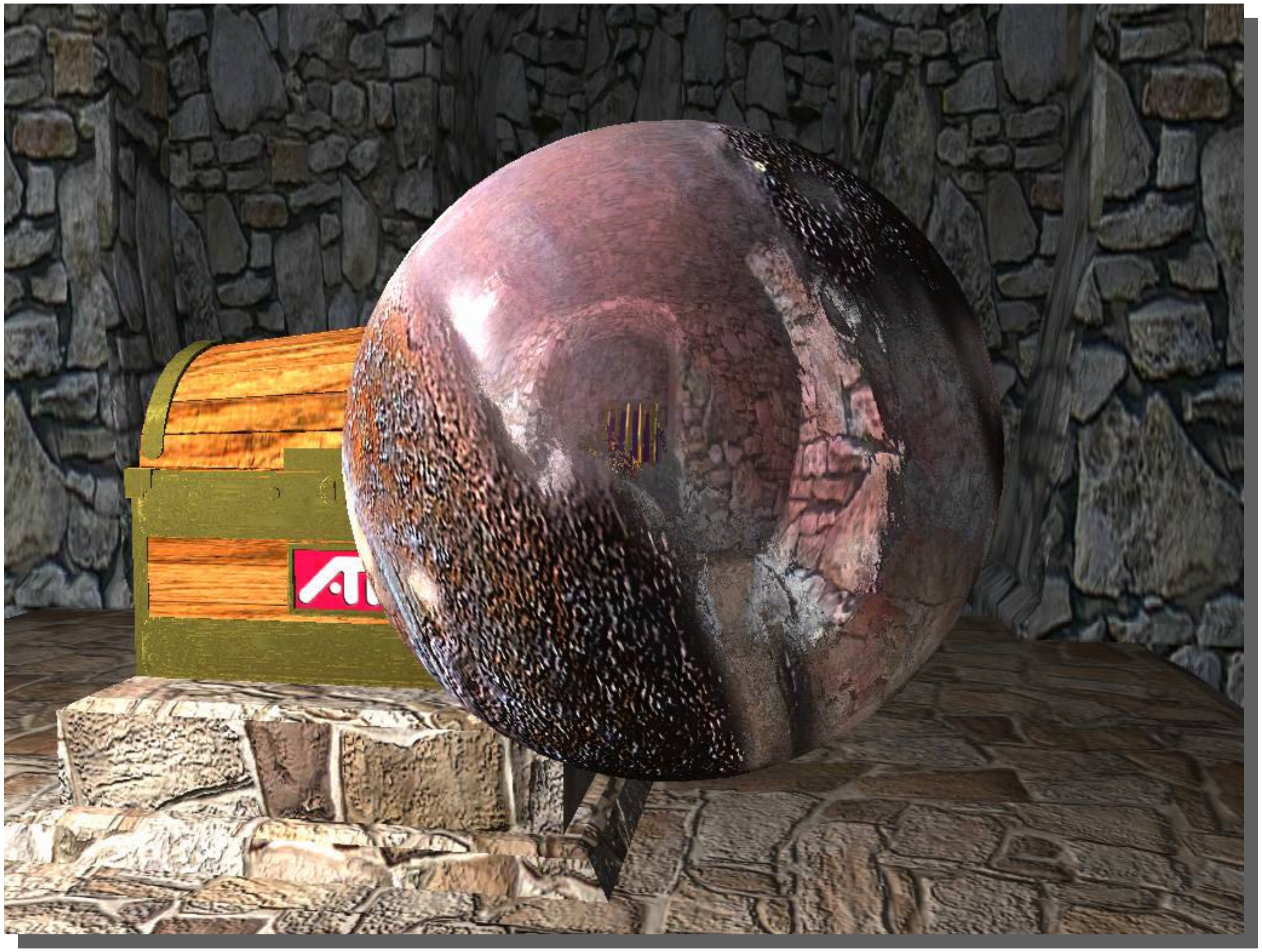
- The 2D case is still valuable and not going away
- The fact that the 2x2 matrix is no longer required to be “state” unlocks this even further.
- Works great with dynamic projective reflection maps for floors, walls, lakes etc
- Good for refraction (heat waves, water effects etc.)



Bumped Cubic Environment Mapping

- Interpolate a 3x3 matrix which represents a transformation from tangent space to cube map space
- Sample normal and transform it by 3x3 matrix
- Sample diffuse map with transformed normal
- Reflect the eye vector through the normal and sample a specular and/or env map
- Do both
- Blend with a per-pixel Fresnel Term!

Bumpy Environment Mapping



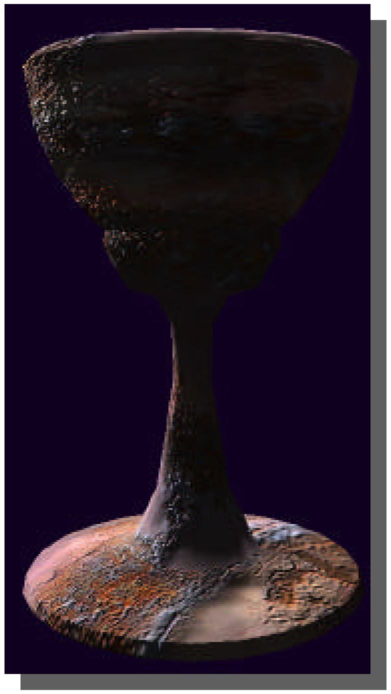
Bumpy Environment Mapping

Dependent Reads

```
texld   r0, t0                 ; Look up normal map
texld   r1, t4                 ; Eye vector through normalizer cube map
texcrd  r4.rgb, t1             ; 1st row of environment matrix
texcrd  r2.rgb, t2             ; 2st row of environment matrix
texcrd  r3.rgb, t3             ; 3rd row of environment matrix
texcrd  r5.rgb, t5             ; World space L (Unit length is light's range)

dp3     r4.r, r4, r0_bx2           ; 1st row of matrix multiply
dp3     r4.g, r2, r0_bx2           ; 2nd row of matrix multiply
dp3     r4.b, r3, r0_bx2           ; 3rd row of matrix multiply
dp3_x2  r3.rgb, r4, r1_bx2         ; 2(N·Eye)
mul     r3.rgb, r4, r3             ; 2N(N·Eye)
dp3     r2.rgb, r4, r4             ; N·N
mad     r2.rgb, -r1_bx2, r2, r3 ; 2N(N·Eye) - Eye(N·N)

phase

texld   r2, r2                     ; Sample cubic reflection map
texld   r3, t0                     ; Sample base map
texld   r4, r4                     ; Sample cubic diffuse map
texld   r5, t0                     ; Sample gloss map

mul     r1.rgb, r5, r2             ; Specular = Gloss * Reflection
mad     r0.rgb, r3, r4_x2, r1      ; Base * Diffuse + Specular
```

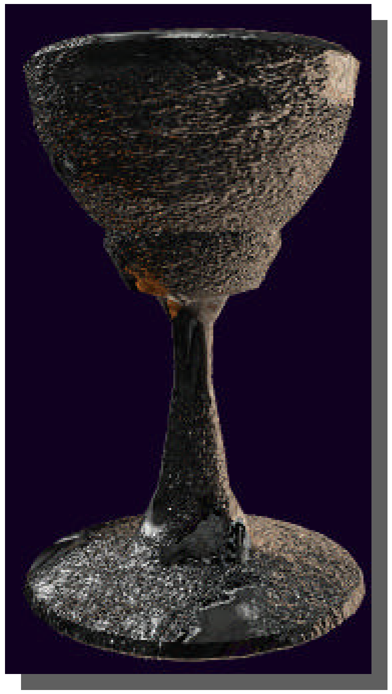
Per-Pixel Fresnel

Per-Pixel Diffuse



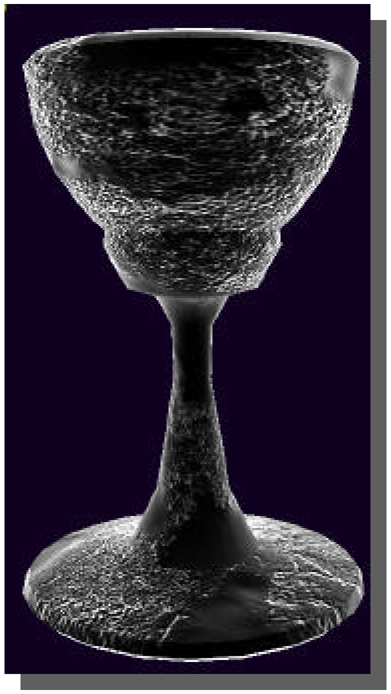
+

Per-Pixel Bumped Environment map



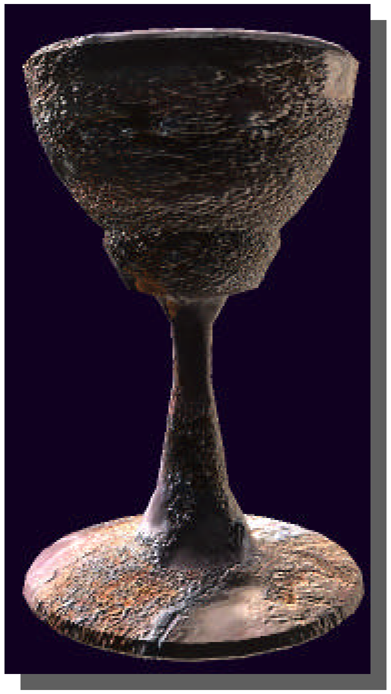
×

Per-Pixel Fresnel



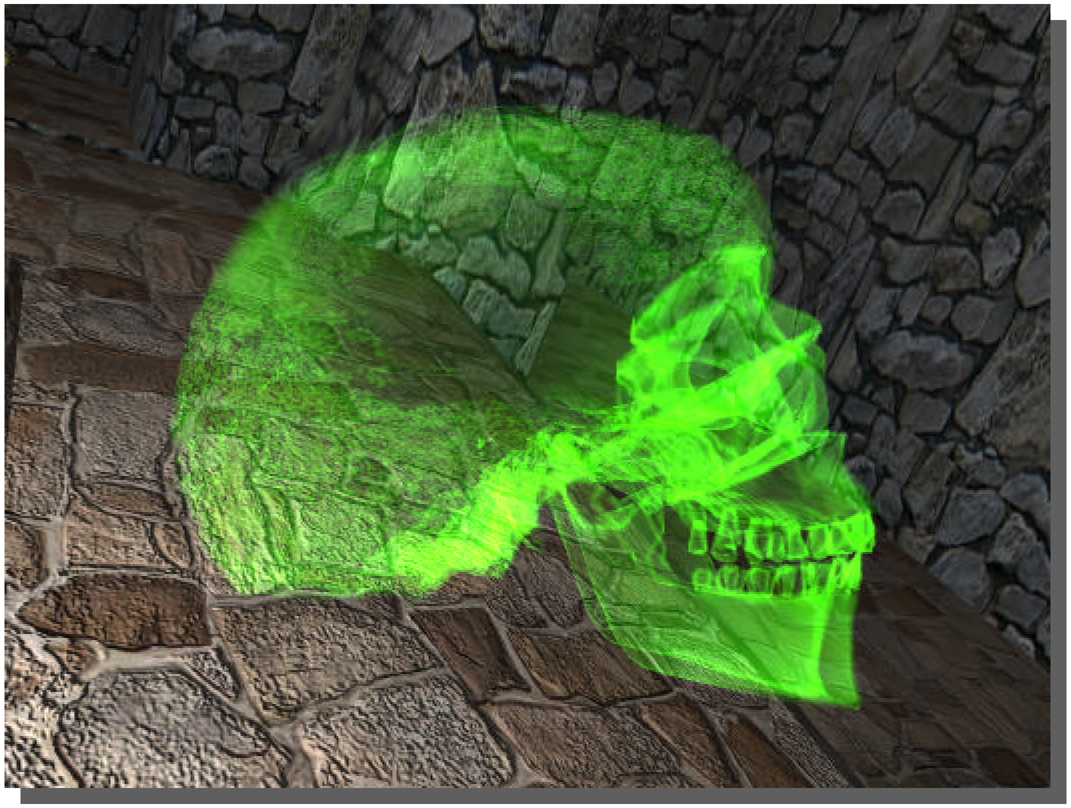
=

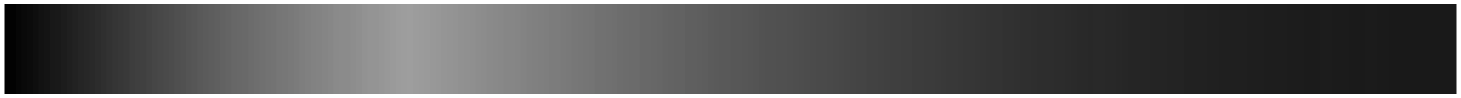
Result



Ghost/Glow Shader

Per-Pixel N·Eye used to Index into Glow Map





Glow Map (which gets multiplied by {0.1, 0.1, 0.5, 0.0, 1.0} to tint it ghostly green

Ghost Shader

```
dp3    r4.r, r4, r0_bx2    ; 1st row of matrix multiply
dp3    r4.g, r2, r0_bx2    ; 2nd row of matrix multiply
dp3    r4.b, r3, r0_bx2    ; 3rd row of matrix multiply
dp3    r5, r4, r1          ; (N·Eye)

phase

texld r2, r5

mov r0.rgb, r2
```

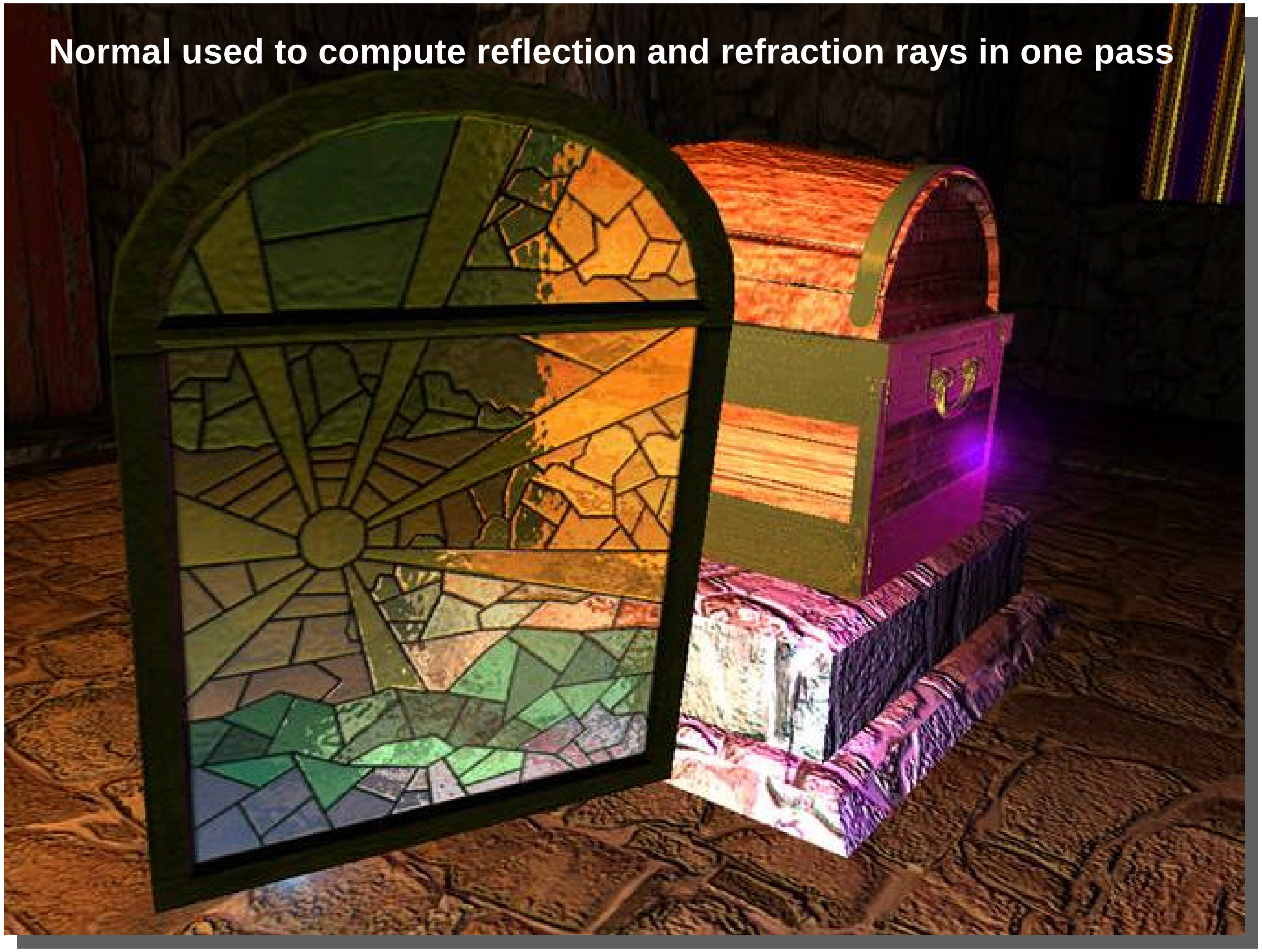
Multi-light Shaders

Four Diffuse Per-Pixel Lights in one Pass

4-light Shader

```
dp3_sat r2.rgb, r1_bx2, r2_bx2 ; *= (N·L1)
mul_x2  r2.rgb, r2, c0         ; *= Light Color
dp3_sat r3.rgb, r1_bx2, r3_bx2 ; Light 2
mul_x2  r3.rgb, r3, c1
dp3_sat r4.rgb, r1_bx2, r4_bx2 ; Light 3
mul_x2  r4.rgb, r4, c2
phase
texld r0, t0
texld r5, t4
dp3_sat r5.rgb, r1_bx2, r5_bx2 ; Light 4
mul_x2  r5.rgb, r5, c3
mul  r1.rgb, r2, v0.x      ; Attenuate light 1
mad  r1.rgb, r3, v0.y, r1 ; Attenuate light 2
mad  r1.rgb, r4, v0.z, r1 ; Attenuate light 3
mad  r1.rgb, r5, v0.w, r1 ; Attenuate light 4
add  r1.rgb, r1, c7        ; += Ambient
mul  r0.rgb, r1, r0        ; Modulate by base map
```

Reflection and Refraction Shader


Normal used to compute reflection and refraction rays in one pass

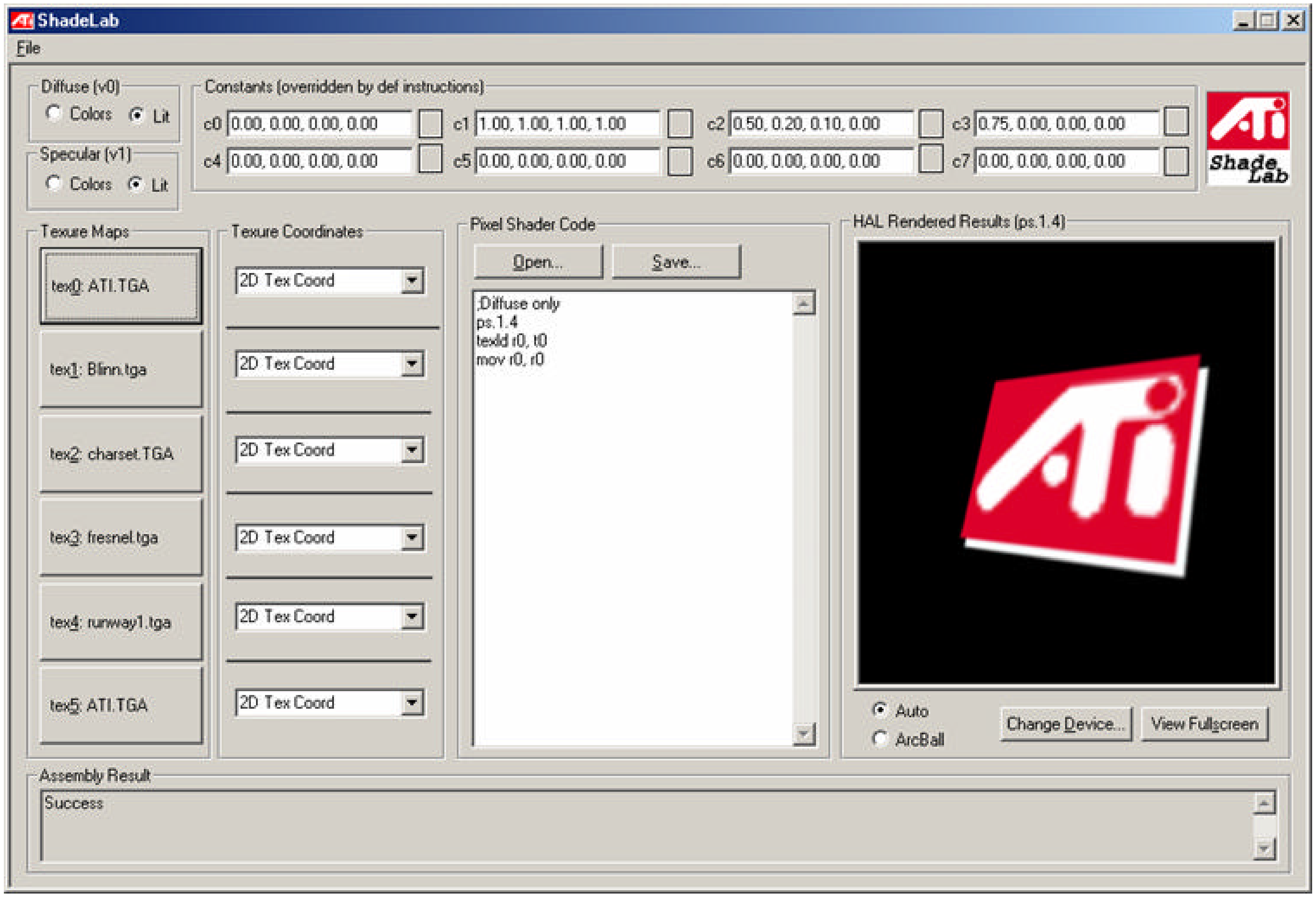
Reflection and Refraction

```
dp3  r4.r, r4, r0_bx2  ; 1st row of matrix multiply
dp3  r4.g, r2, r0_bx2  ; 2nd row of matrix multiply
dp3  r4.b, r3, r0_bx2  ; 3rd row of matrix multiply
mul  r5.rgb, c0.g, -r1_bx2   ; Refract by c0 = index
                             ; of refraction fudge
                             ; factor
mad  r2.rgb, c0.r, -r4, r5   ; Refract by c0 = index
                             ; of refraction fudge
                             ; factor
```

Developing Pixel Shaders

- The good news is that since shaders are a programming language they’re easier to edit and debug on the fly than complex multitexturing setups
- Use tools like *ShadeLab* to play around with them interactively
- As Pixel Shaders grow in complexity comparable to today’s vertex shaders, we’ll need to address the language issues like we are at the vertex shader level today. See Evan Hart’s talk right after lunch for more on this.

ShadeLab



Diffuse (v0): Colors [✓ Lit]

Specular (v1): Colors [✓ Lit]

Constants (overridden by def instructions)

c0	0.00, 0.00, 0.00, 0.00	c1	1.00, 1.00, 1.00, 1.00	c2	0.50, 0.20, 0.10, 0.00	c3	0.75, 0.00, 0.00, 0.00
c4	0.00, 0.00, 0.00, 0.00	c5	0.00, 0.00, 0.00, 0.00	c6	0.00, 0.00, 0.00, 0.00	c7	0.00, 0.00, 0.00, 0.00

Texure Maps	Texure Coordinates
tex0: ATI.TGA	2D Tex Coord
tex1: Blinn.tga	2D Tex Coord
tex2: charset.TGA	2D Tex Coord
tex3: fresnel.tga	2D Tex Coord
tex4: runway1.tga	2D Tex Coord
tex5: ATI.TGA	2D Tex Coord

Pixel Shader Code

```
;Diffuse only
ps.1.4
texld r0, t0
mov r0, r0
```

HAL Rendered Results (ps.1.4)

[✓ Auto] ArcBall

Assembly Result

```
Success
```

The Road to DX9

- 1.4 is a good preparation for how to think about DX9 pixel shaders
- Unified instruction set
- Higher precision
- Vectors, not colors
- Flexible dependent texture reads

Summary

- Pixel Shader Overview
- 1.4 Pixel Shaders
  - Unified Instruction set
  - Flexible dependent texture read
- Image Processing
- 3D volume visualizations
- Effects on 3D Surfaces
  - Per-pixel lighting
  - Per-pixel Fresnel Term
  - Bumpy Environment mapping
  - Per-pixel anisotropic lighting
- ShadeLab Tool
  - Good for playing around with shaders in real-time
  - Generates Vertex Shader code on the fly to feed pixel shader

Call To Action

- Use 1.4 pixel shaders in your games when they are present
- Abstract your shader usage to the point that shaders and shader versions are just part of the dataset
- Start thinking about pixel shaders according to this model…it's where we're going in DX9.

Acknowledgements

- Chris Brennan, Chris Oat, John Isidoro and Dan Baker for insights and demos

Questions