DirectX® 9 Shading

Jason L. Mitchell

3D Application Research Group Lead

`JasonM@ati.com`

ATI Mojo Day

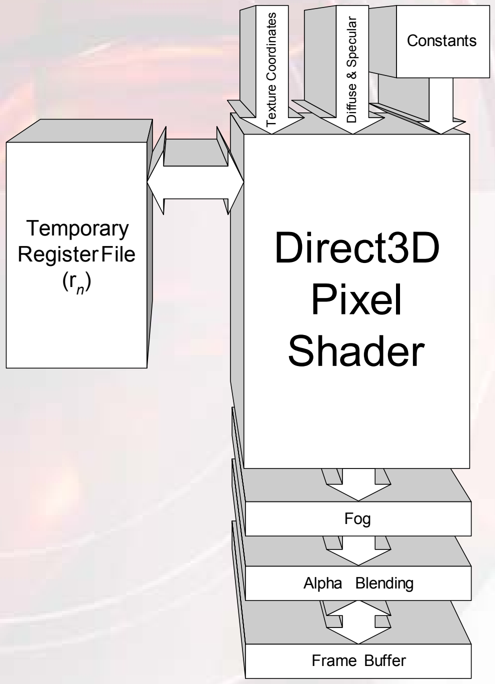
Shading Outline

- DirectX® 8.1 1.4 Pixel Shader Review
  - Depth sprites
- DirectX® 9 2.0 Pixel Shaders
  - Procedural wood in assembly
  - High Level Shading Language
    - Review wood shaders using HLSL
    - Analyze resulting assembly code
  - Image-Space Operations
    - Edge detection for cartoon outlining
    - HDR
  - sRGB and Gamma

ps.1.4 Review

- Introduced in RADEON™ 8500 last year
- Supported by all ps.2.0 parts and beyond
- Available in RADEON™ 9000 and other RADEON™ 8500 derivatives
- Only pixel shader model available in mobile graphics chips from any vendor: RADEON™ Mobility 9000

ps.1.4 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 (ps.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**

ps.1.4 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 α) instruction may only reference up to two constants
- Example constant definition syntax:

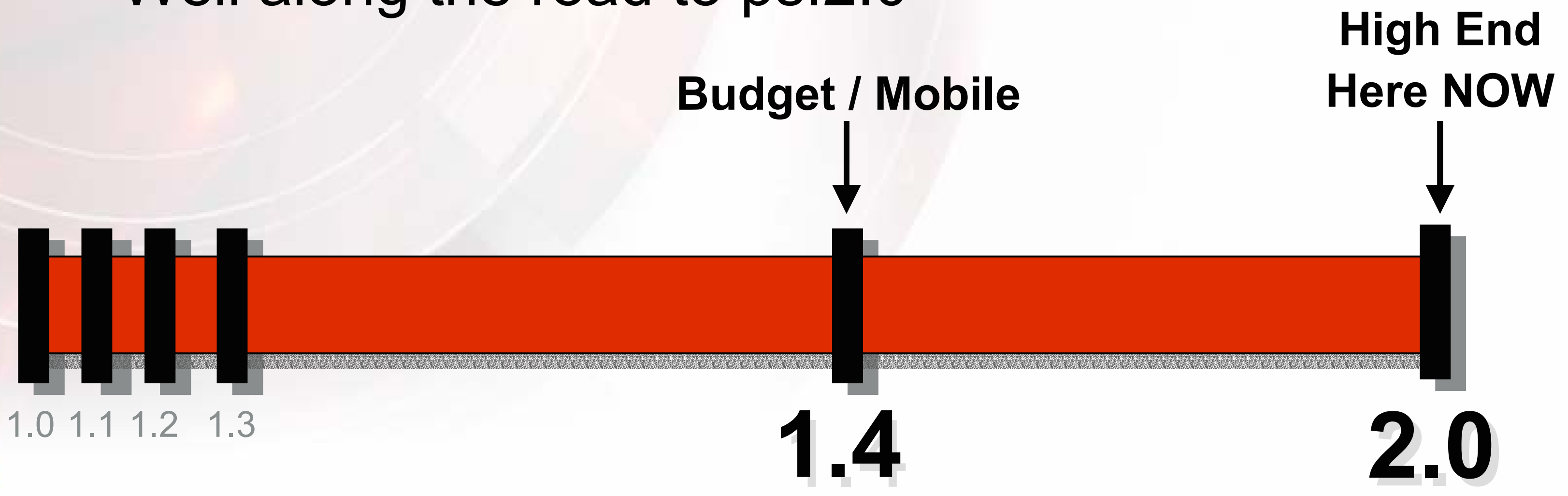
```
def c0, 1.0f, 0.5f, -0.3f, 1.0f
```

ps.1.4 Interpolated Quantities

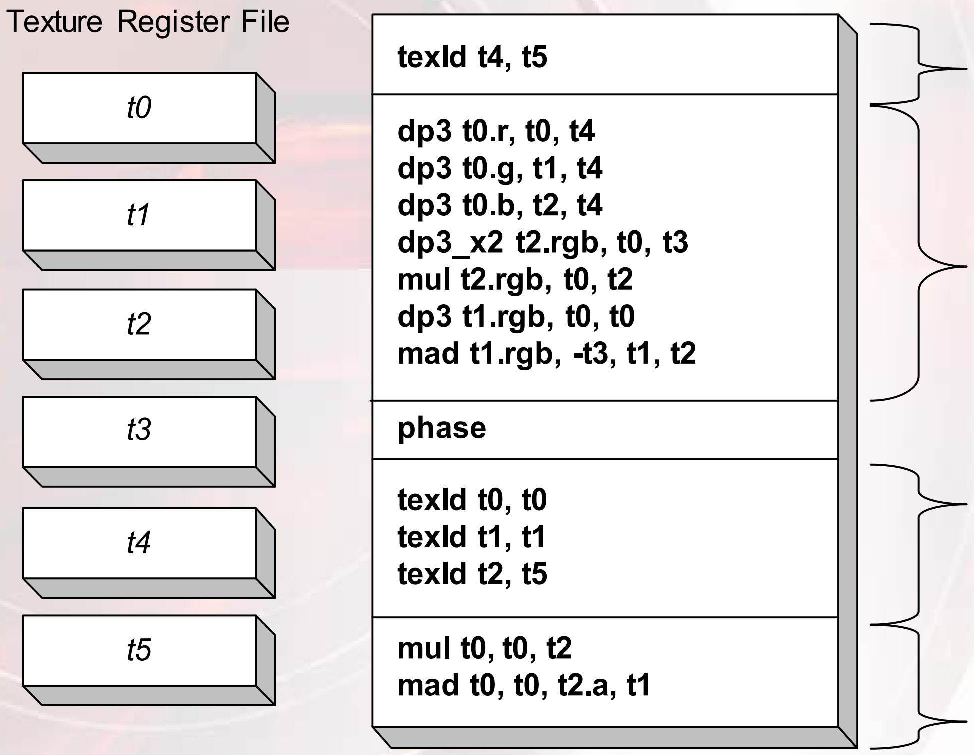
- Diffuse and Specular (v0 and v1)
  - Low precision and unsigned
  - In ps.1.1 through ps.1.3, available only in “color shader”
  - Not available before ps.1.4 *phase* marker
- Texture coordinates
  - High precision signed interpolators
  - Can be used as extra colors, signed vectors, matrix rows etc

ps.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 and instructions than legacy models
- High precision and range of at least -8 to +8
- Well along the road to ps.2.0



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
  - Up to 6 textures
- Address Shader
  - Up to 8 instructions
- Optional Sampling
  - Up to 6 textures
  - 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 register from a texture
- `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 other vendors' chips that don't have clip planes
- `texdepth`
  - Substitute value for this pixel's z!

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
- Scale and bias (_bx2)
  - Equivalent to _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
```

Projective Textures

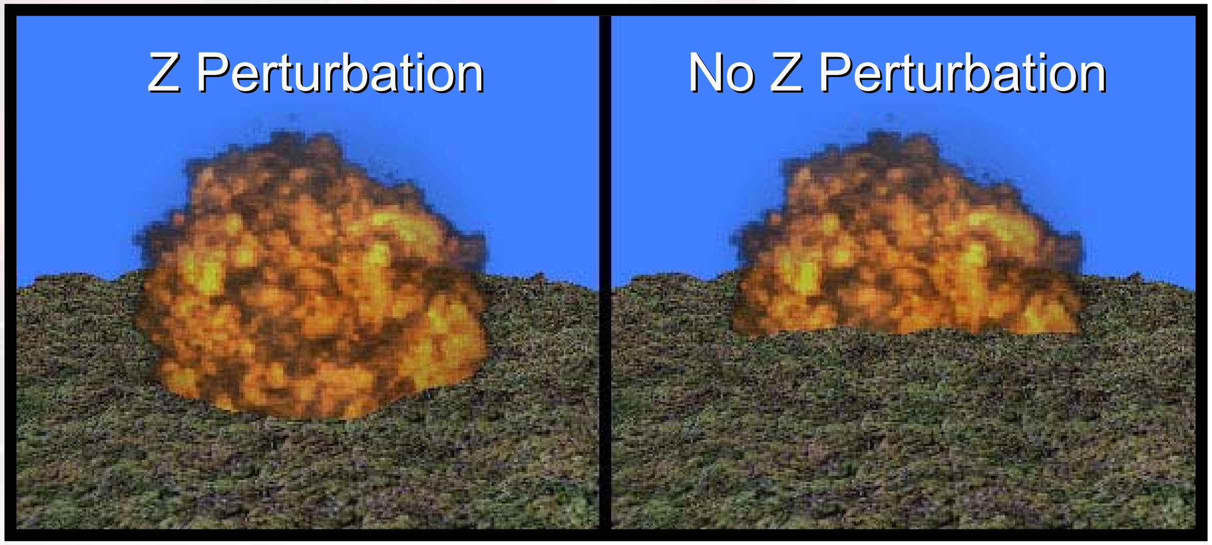
- You can do texture projection on any texld instruction.
- This includes projective *dependent* reads, which are fundamental to doing reflection and refraction mapping of things like water surfaces. This is used in the Nature and Rachel demos.
- Syntax looks like this:

```
texld r3, r3_dz  or
texld r3, r3_dw
```

- Useful for projective textures like the refraction map in the nature demo or just doing a divide.

Depth Sprites

- Encode *delta z* in channel of sprite texture map
- Combine with interpolated of sprite primitive
- Use **`texdepth`** instruction to replace interpolated *z* with perturbed *z*



Vertex Shader for Depth Sprite Sample

```
// c0..c3 - MVP
// c9.x - billboard size
// c10.xyz - billboard position
// c11.xy - bias and scale
// c12.xy - texture scroll
// c12.z - texture scale
vs.1.1

// Compute billboard size and position
mov r0, c9.xxxw
mad r1, v0, r0, c10
// Transform position
m4x4 oPos, r1, c0

// Base texture coordinates
mov oT0, v7
// Compute texture coordinates for procedural smoke
mul oT1, v7, c12.zwww
mad oT2, v7, c12.z, c12.xyww

// Use texture coordinate interpolators to send down
// the Z bias and scale.
mov oT3, c11
```

Depth Sprite ps.1.4 Code

```
 ps.1.4
 def c0, 1, 0, 0, 0

 texld r0, t0                  // Fetch depth information
 texcrd r3.xyz, t3             // Pass through depth scale and bias

 // Compute new depth based on depth information stored in the texture
 mad r5.r,-  r0.r, r3.x, r3.y
 mov r5.g, c0.x

 phase

 texld r1, t1                  // Fetch two noise textures
 texld r2, t2
 texld r3, t0                  // Fetch explosion animation
 texdepth r5                   // Output new depth

 mad_d4 r0.rgb, r1, r0.r, r2 // Combine noise textures to create smoke
+mul r0.a, r0.r, r0.r          // Compute attenuation based on depth

 mov_sat r2, c1.r              // Clamp constant value for lrp

 lrp r0, r2, r3, r0            // Interpolate between fireball and smoke

 mul r0.a, r0.a, c1.g          // Fade out the effect
```

Flat Explosion

```
 ps.1.4

 texld r0, t0                  // Fetch depth information
 texld r1, t1                  // Fetch two noise textures
 texld r2, t2
 texld r3, t0                  // Fetch explosion animation

 mad_d4 r0.rgb, r1, r0.r, r2 // Combine noise to make smoke
+mul r0.a, r0.r, r0.r          // Compute attenuation from depth

 mov_sat r2, c1.r              // Clamp constant value for lrp

 lrp r0, r2, r3, r0            // Interpolate between fireball
                               // and smoke

 mul r0.a, r0.a, c1.g          // Fade out the effect
```

RADEON™ 9700 Pixel Shaders

- DirectX® 9 2.0 pixel shaders
- `ATI_fragment_program` in OpenGL
- Floating point pixels
- Longer programs
  - Up to 64 ALU instructions
  - Up to 32 texture instructions
  - Up to 4 levels of dependent read

2.0 Pixel Shader Instruction Set

- ALU Instructions
  - **add**, **mov**, **mul**, **mad**, **dp2add**, **dp3**, **dp4**, **frc**, **rcp**, **rsq**, **exp**, **log** and **cmp**
- ALU Macros
  - **MIN**, **MAX**, **LRP**, **POW**, **CRS**, **NRM**, **ABS**, **SINCOS**, **M4X4**, **M4X3**, **M3X3** and **M3X2**
- Texture Instructions
  - **texld**, **texldp**, **texldb** and **texkill**

2.0 Pixel Shader Resources

- 12 Temp registers
- 8 4D texture coordinate iterators
- 2 color iterators
- 32 4D constants
- 16 Samplers
- Explicit output registers
  - `oC0`, `oC1`, `oC2`, `oC3` and `oDepth`
  - Must be written with a `mov`
- Some things removed
  - No source modifiers except **negate**
  - No instruction modifiers except **saturate**
  - No Co-issue

Argument Swizzles

- `.r`, `.rrrr`, `.xxxx` or `.x`
- `.g`, `.gggg`, `.yyyy` or `.y`
- `.b`, `.bbbb`, `.zzzz` or `.z`
- `.a`, `.aaaa`, `.wwww` or `.w`
- `.xyzw` or `.rgba` (No swizzle) or nothing
- `.yzxw` or `.gbra` (can be used to perform a cross product operation in 2 clocks)
- `.zxyw` or `.brga` (can be used to perform a cross product operation in 2 clocks)
- `.wzyx` or `.abgr` (can be used to reverse the order of any number of components)

Samplers

- **Separated from texture stages**
- **Sampler State**
  - U, V and W address modes
  - Minification, Magnification and Mip filter modes
  - LOD Bias
  - Max MIP level
  - Max Anisotropy
  - Border color
  - sRGB conversion (new in DirectX® 9)
- **Texture Stage State**
  - Color, Alpha and result Arguments for legacy multitexture
  - Color and Alpha Operations for legacy multitexture
  - EMBM matrix, scale and offset
  - Texture coordinate index for legacy multitexture
  - Texture Transform Flags
  - Per-stage constant (new in DirectX® 9)

ps.2.0 Review – Comparison with ps.1.4

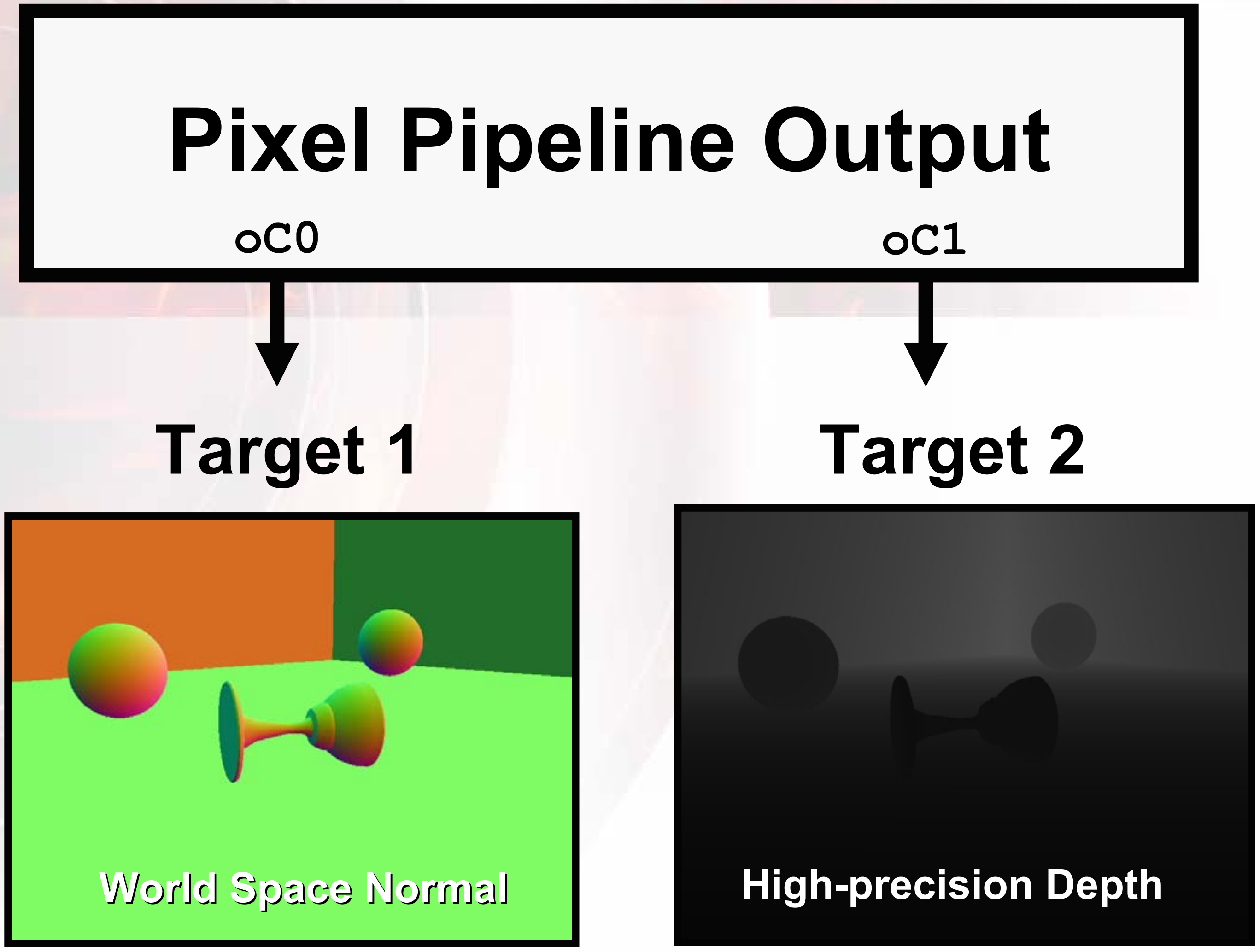
```
ps.1.4

texld r0, t0 ; base map
texld r1, t1 ; bump map

; light vector from normalizer cube map
texld r2, t2

; half angle vector from normalizer cube map
texld r3, t3

; N.L
dp3_sat r2, r1_bx2, r2_bx2

; N.L * diffuse_light_color
mul r2, r2, c2

; (N.H)
dp3_sat r1, r1_bx2, r3_bx2

; approximate (N.H)^16
; [(N.H)^2 - 0.75] * 4 == (N.H)^16
mad_x4_sat r1, r1, r1, c1

; (N.H)^32
mul_sat r1, r1, r1

; (N.H)^32 * specular color
mul_sat r1, r1, c3

; [(N.L) * base] + (N.H)^32
mad_sat r0, r2, r0, r1
```

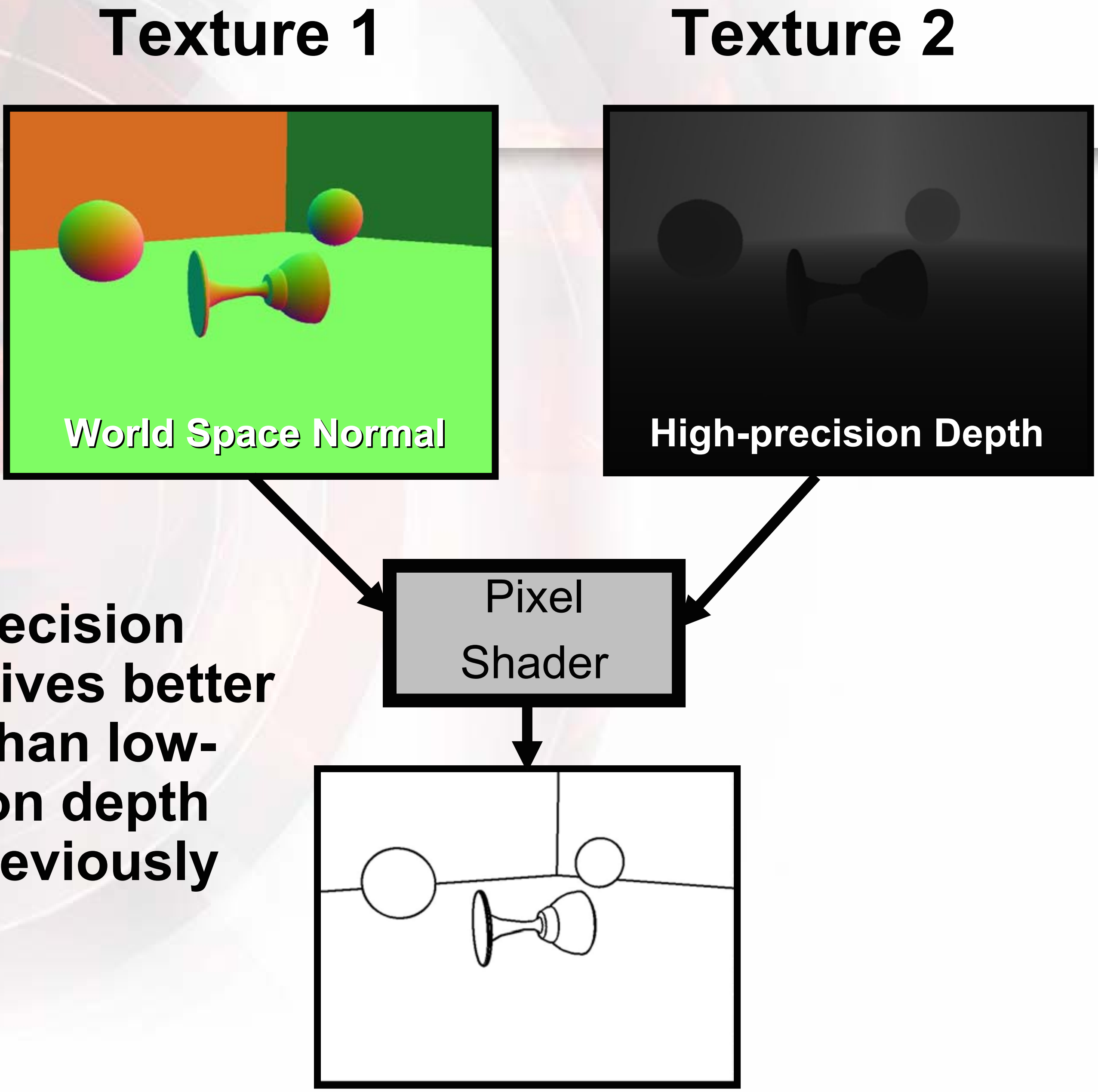
```
ps.2.0

dcl t0
dcl t1
dcl t2
dcl t3

dcl_2d   s0
dcl_2d   s1

texld r0, t0, s0   ; base map
texld r1, t1, s1   ; bump map

dp3 r2.x, t2, t2   ; normalize L
rsq r2.x, r2.x
mul r2, t2, r2.x

dp3 r3.x, t3, t3   ; normalize H
rsq r3.x, r3.x
mul r3, t3, r3.x

mad r1, r1, c4.y, c4.x ; scale and bias N

dp3_sat r2, r1, r2 ; N.L
mul r2, r2, c2     ; N.L * diffuse_light_color
dp3_sat r1, r1, r3 ; (N.H)
pow r1, r1.x, c1.x ; (N.H)^k
mul r1, r1, c3     ; (N.H)^k *specular_light_color

mad_sat r0, r0, r0, r1 ; [(N.L) * base] + (N.H)^k

mov oC0, r0
```

Multiple render targets

- Output up to four colors from pixel shader
  - Write to output registers **`oC0`**, **`oC1`**, **`oC2`**, **`oC3`**
- Useful for intermediate results in multipass algorithms
- Can use as G-buffer [Saito and Takahashi ‘90]
  - Used to optimize NPR outlining example
  - Also improves Depth of Field effect discussed later in this lecture

Multiple render targets





High Precision depth gives better edges than low-precision depth used previously

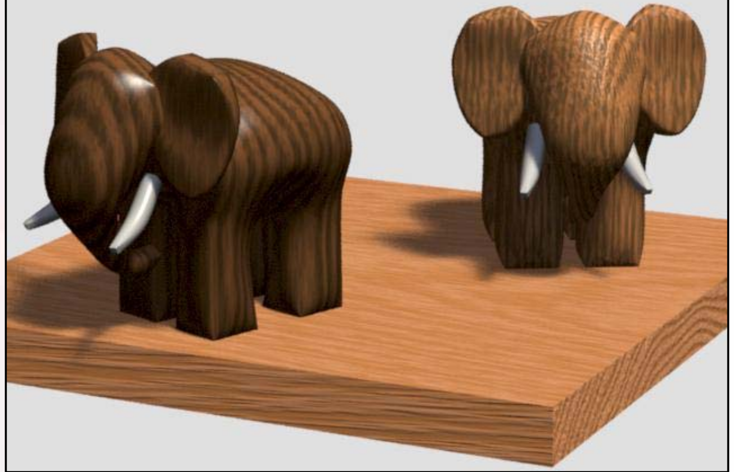
Advanced Surface Types

- IEEE 32-bit surfaces
  - **1-, 2- and 4-channel versions**
- 16-bit float s10e5 surfaces
  - **1-, 2- and 4-channel versions**
- 16-bit fixed point surfaces
  - **1-, 2- and 4-channel versions**
- sRGB
  - **2.2 de-gamma on read**
  - **2.2 gamma on write**

Procedural Wood

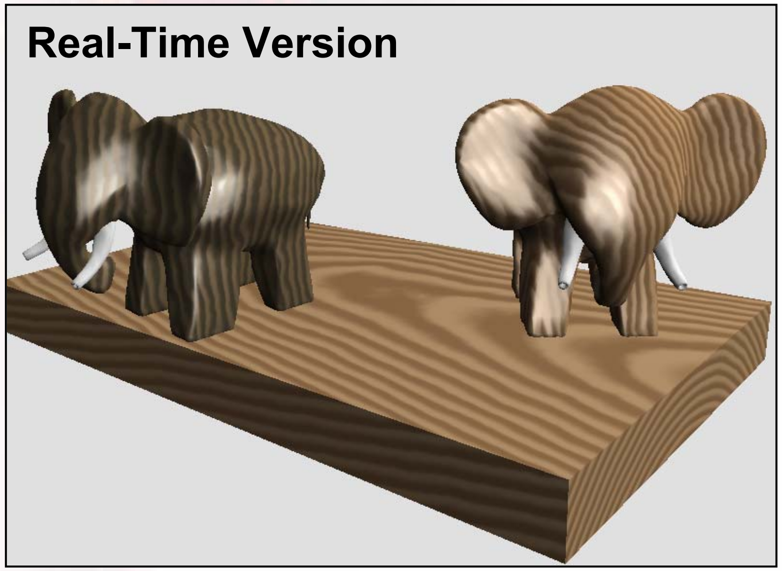
- Based on example in *Advanced RenderMan*
- Uses volume texture for noise, 1D texture for smooth pulse train and 2D texture for variable specular function
- My version has 6 intuitive parameters
  - Light Wood Color
  - Dark Wood
  - ring frequency
  - ring noise amplitude
  - trunk wobble frequency
  - trunk wobble amplitude

Non-Real-Time Version



Procedural Wood

- 35-instruction 2.0 pixel shader
- Samples noise map 6 times
- Phong shading



Step-by-step Approach

- Shader Space (P_{shade})
- Distance from trunk axis (*z*)
- Run through pulse train
- Add noise to P_{shade}
- Add noise as function of *z* to wobble

Wood Vertex Shader

```
dcl_position v0
dcl_normal   v3

def c40, 0.0f, 0.0f, 0.0f, 0.0f  // All zeroes
m4x4 oPos, v0, c[0]              // Transform position to clip space

m4x4 r0, v0, c[17]               // Transformed Pshade (using texture matrix 0)
mov oT0, r0
m4x4 oT1, v0, c[21]              // Transformed Pshade (using texture matrix 1)
m4x4 oT2, v0, c[25]              // Transformed Pshade (using texture matrix 2)

mov r1, c40
mul r1.x, r0.z, c29.x            // {freq*Pshade.z, 0, 0, 0}
mov oT3, r1                      // {freq*Pshade.z, 0, 0, 0} for 1D trunkWobble noise in x
mov r1, c40
mad r1.x, r0.z, c29.x, c29.y     // {freq*Pshade.z + 0.5, 0, 0, 0}
mov oT4, r1                      // {Pshade.z+0.5, 0, 0, 0} for 1D trunkWobble noise in y

m4x4 oT6, v0, c[4]               // Transform position to eye space
m3x3 oT7.xyz, v3, c[8]           // Transform normal to eye space
```

$P_{\textit{shade}}$

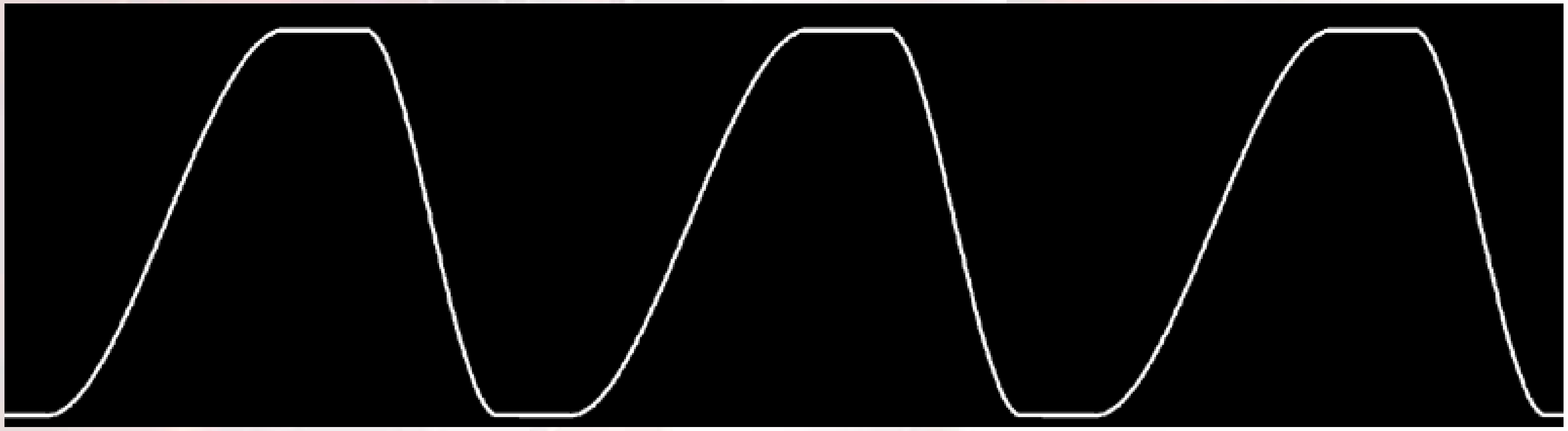
- For this app, P_{shade} is just world space
- The infinite virtual log runs along the *z* axis
- I make a few different transformed versions of P_{shade} in the vertex shader in order to turn scalar noise into color noise, as I'll show later

Distance from z axis

- $\text{sqrt}\,(P_{shade}.x^2 + P_{shade}.y^2) * \mathit{freq}$
- Pass this in to pulse train

Pulse Train

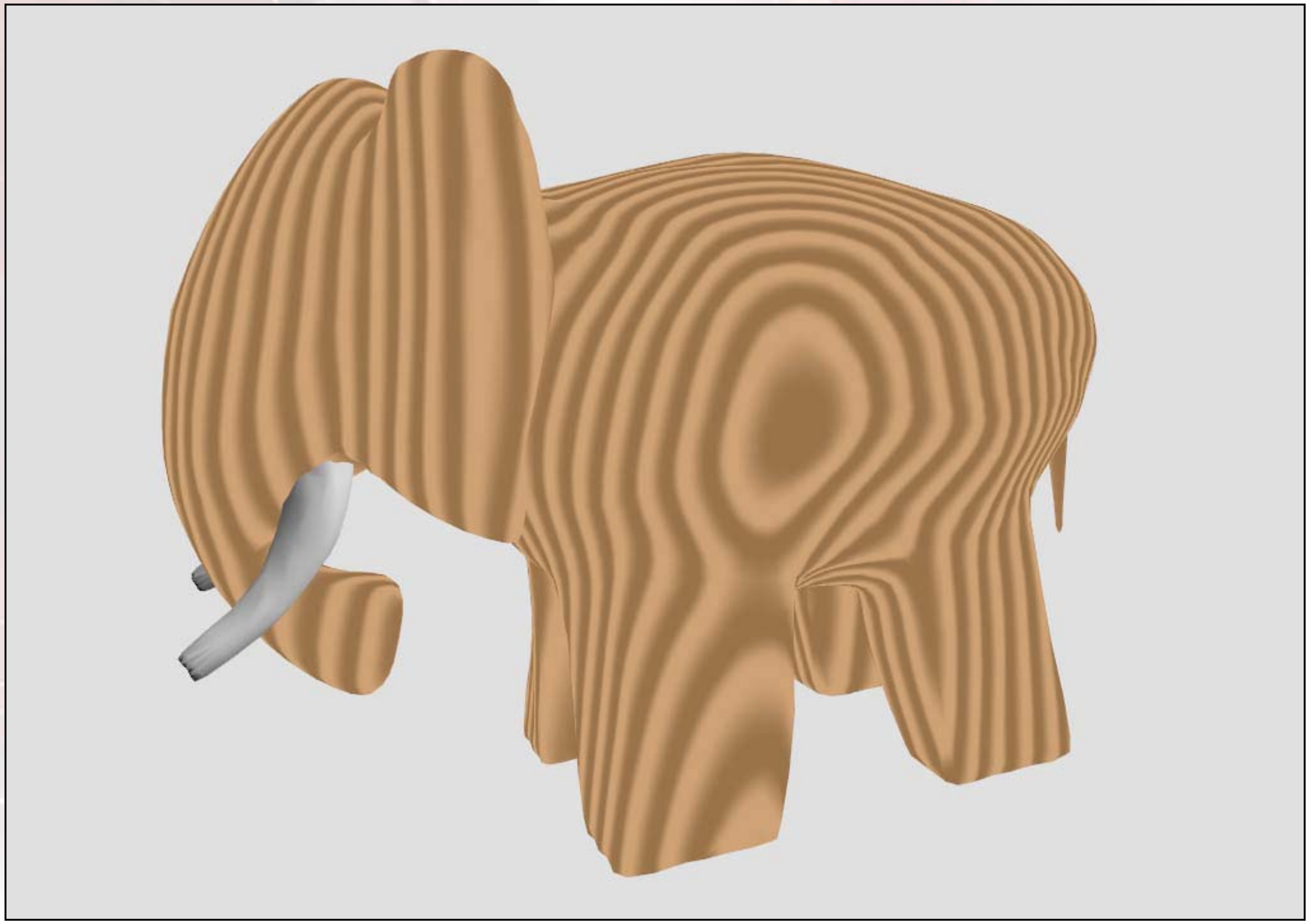
- Tuned to mimic the way colors mix in real wood
- One pulse stored in 1D texture which repeats



Concentric Rings

```
ps.2.0

def c0, 2.0f, -1.0f, 0.5f, 0.5f // scale, bias, half, X
def c1, 1.0f, 1.0f, 0.1f, 0.0f  // X, X, 0.1, zero
// c2: xyz == Light Wood Color, w == ringFreq
// c3: xyz == Dark Wood Color,  w == noise amplitude

dcl t0.xyzw                 // xyz == Pshade (shader-space position), w == X

dcl_2d     s1               // 1D smooth step function (blend factor in red, spec exp in green,…)

dp2add r0, t0, t0, c1.w     // x*x + y*y + 0
rsq r1.x, r0.x              // 1/sqrt(x*x + y*y)
mul r0, r1.x, r0.x          // sqrt(x*x + y*y)
mul r0, r0, c2.w            // sqrt(x*x + y*y) * freq

texld r0, r0, s1            // Sample from 1D pulse train texture

mov r1, c3
lrp r2, r0.x, c2, r1        // Blend between light and dark wood colors

mov oC0, r2
```

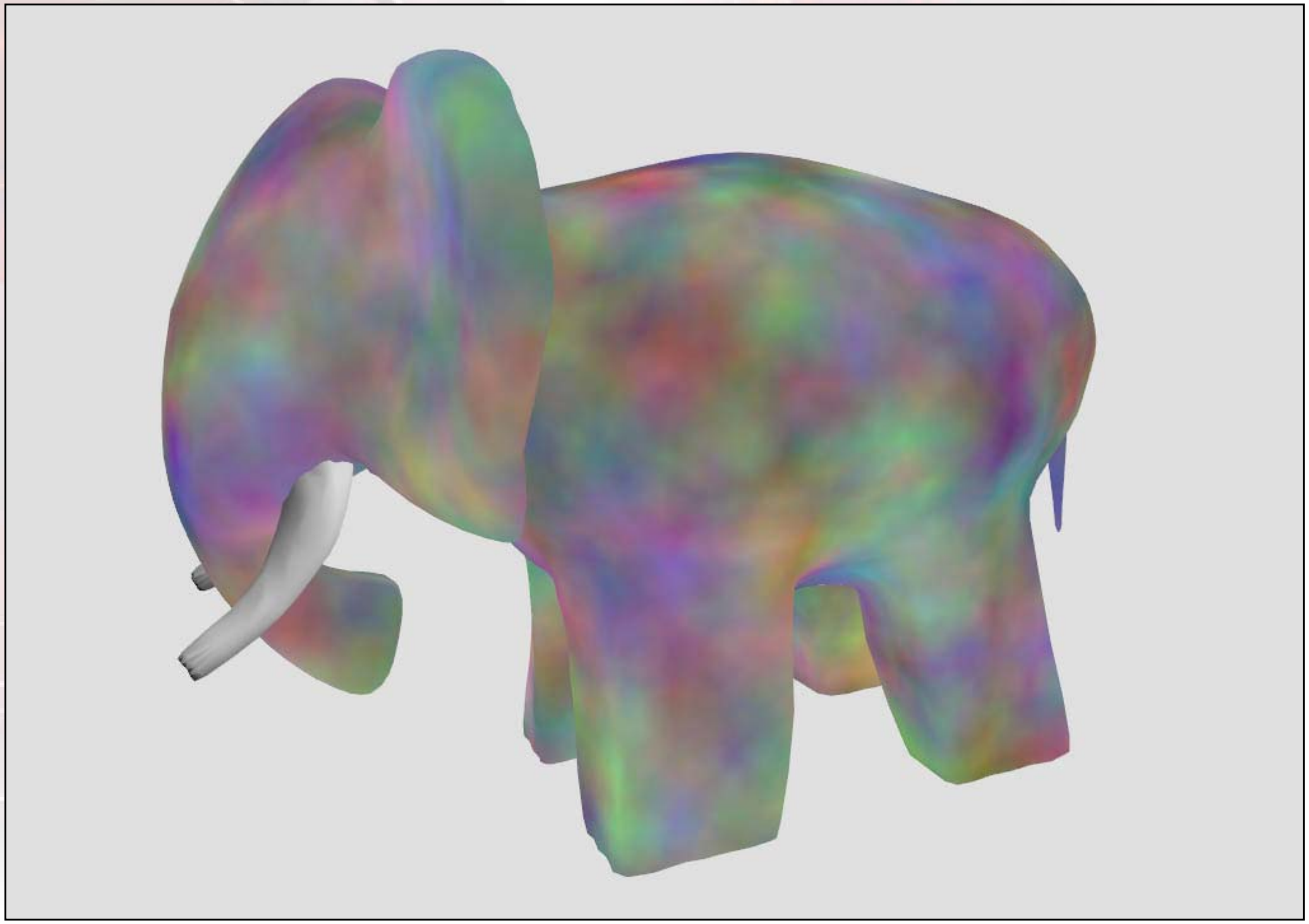
Concentric Rings



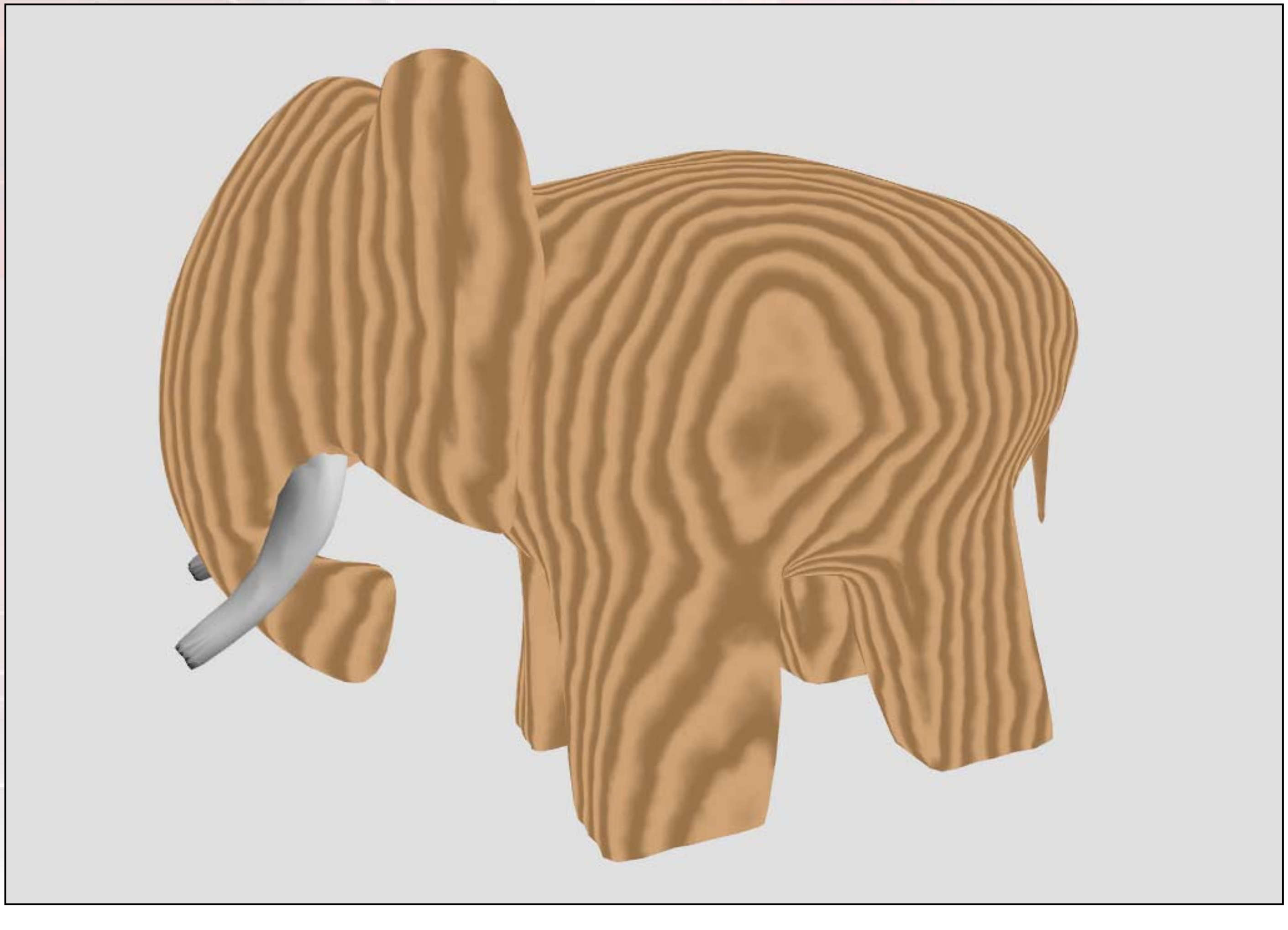
Noisy Rings

```
ps.2.0
def c0, 2.0f,-  10f, 0.5f, 0.5f // scale, bias, half, X
def c1, 1.0f, 1.0f, 0.1f, 0.0f  // X, X, 0.1, zero
// c2: xyz == Light Wood Color, w == ringFreq
// c3: xyz == Dark Wood Color,  w == noise amplitude
// c4: xyz == L_eye,  w == trunkWobbleAmplitude
dcl t0.xyzw                  // xyz == Pshade (shader  space position), w == X
dcl t1.xyzw                  // xyz == Perturbed Pshade, w == X
dcl t2.xyzw                  // xyz == Perturbed Pshade, w == X
dcl t3.xyzw                  // xyz == {Pshade.z, 0, 0}, w == X
dcl t4.xyzw                  // xyz == {Pshade.z + 0.5, 0, 0}, w == X
dcl_volume s0                // Luminance only Volume noise
dcl_2d     s1                // 1D smooth step function
texld r3,  t0, s0            // Sample dX from scalar noise at P_shade
texld r4,  t1, s0            // Sample dY from scalar noise at perturbed P_shade
texld r5,  t2, s0            // Sample dZ from scalar noise at perturbed P_shade
mov r3.y, r4.x               // Put dY in y
mov r3.z, r5.x               // Put dZ in z
mad r3, r3, c0.x, c0.y       // Put noise in-  1..+1 range
mad r7, c3.w, r3, t0         // Scale by amplitude and add to P_shade to warp domain
dp2add r0, r7, r7, c1.w      // x^2 + y^2 + 0
rsq r0, r0.x                 // 1/sqrt(x^2 + y^2)
rcp r0, r0.x                 // sqrt(x^2 + y^2)
mul r0, r0, c2.w             // sqrt(x^2 + y^2) * freq
texld r0, r0, s1             // Sample from 1D pulse train texture
mov r1, c3
lrp r2, r0.x, c2, r1         // Blend between light and dark wood colors
mov oC0, r2
```

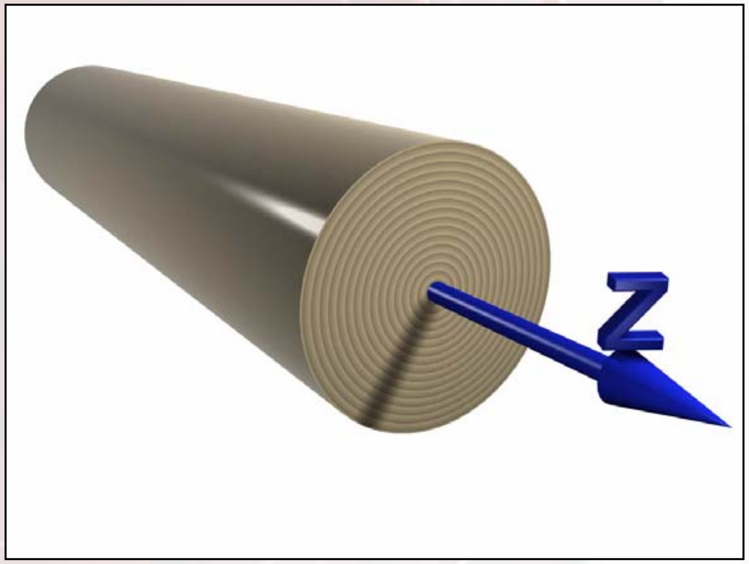
Colored Volume Noise



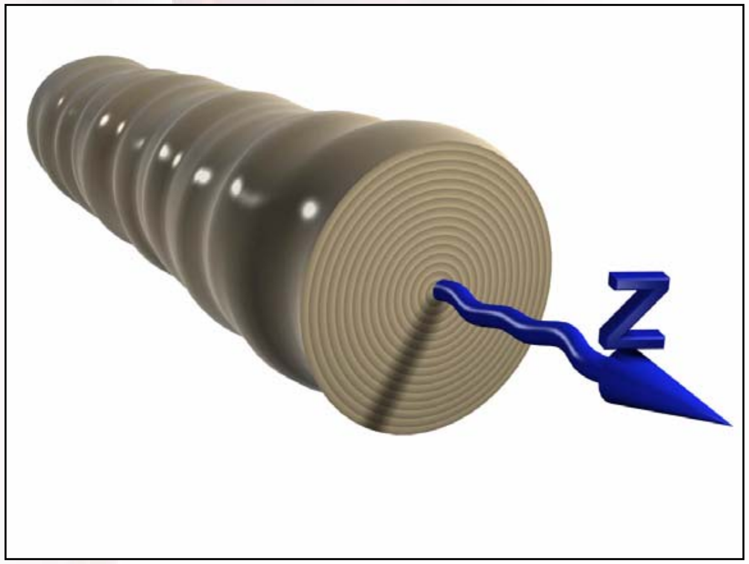
Noisy Rings



Trunk Wobble



Without Wobble

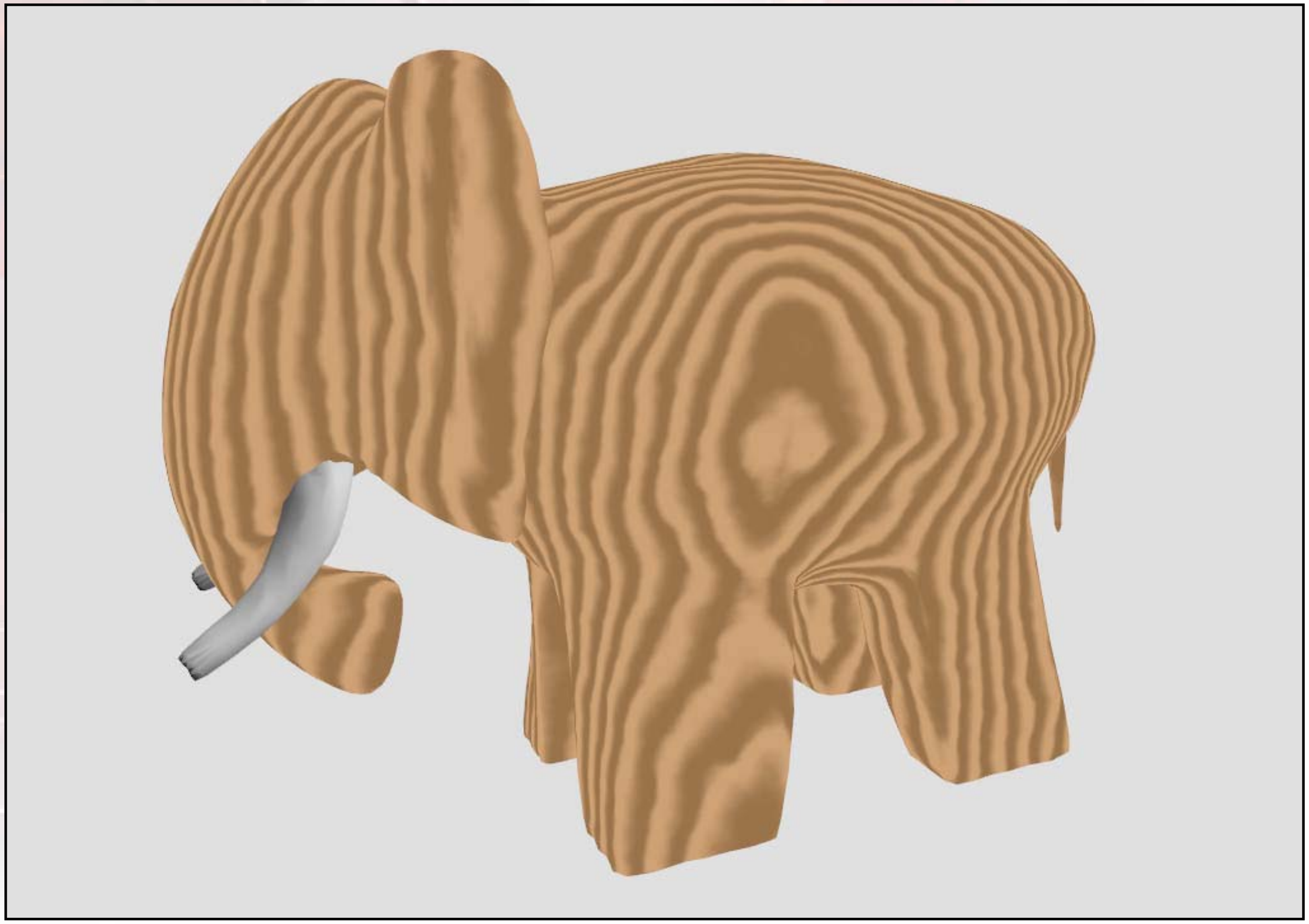


With Wobble

Noise and wobble

```
def c0, 2.0f, -1.0f, 0.5f, 0.5f // scale, bias, half, X
def c1, 1.0f, 1.0f, 0.1f, 0.0f  // X, X, 0.1, zero
// c2: xyz == Light Wood Color, w == ringFreq
// c3: xyz == Dark Wood Color,  w == noise amplitude
// c4: xyz == L_eye,  w == trunkWobbleAmplitude
dcl t0.xyzw                // xyz == Pshade (shader-space position), w == X
dcl t1.xyzw                // xyz == Perturbed Pshade, w == X
dcl t2.xyzw                // xyz == Perturbed Pshade, w == X
dcl t3.xyzw                // xyz == {Pshade.z, 0, 0}, w == X
dcl t4.xyzw                // xyz == {Pshade.z + 0.5, 0, 0}, w == X
dcl_volume s0              // Luminance-only Volume noise
dcl_2d     s1              // 1D smooth step function (blend factor in x, spec exp in y, ...)
texld r3,  t0, s0          // Sample dX from scalar volume noise texture at P_shade
texld r4,  t1, s0          // Sample dY from scalar volume noise texture at perturbed P_shade
texld r5,  t2, s0          // Sample dZ from scalar volume noise texture at perturbed P_shade
texld r6, t3, s0           // Sample trunkWobble.x from scalar noise at {P_shade, 0, 0}
texld r7, t4, s0           // Sample trunkWobble.y from scalar noise at {P_shade + 0.5, 0, 0}
mov r3.y, r4.x             // Put dY in y
mov r3.z, r5.x             // Put dZ in z
mov r6.y, r7.x             // Move to get {trunkWobble.x, trunkWobble.y, 0}
mad r6, r6, c0.x, c0.y     // Put {trunkWobble.x, trunkWobble.y, 0} in -1..+1 range
mad r3, r3, c0.x, c0.y     // Put noise in -1..+1 range
mad r7, c3.w, r3, t0       // Scale noise by amplitude and add to P_shade to warp the domain
mad r7, c4.w, r6, r7       // Scale {trunkWobble.x, trunkWobble.y, 0} by amplitude and add in
dp2add r0, r7, r7, c1.w    // x^2 + y^2 + 0
rsq r0, r0.x               // 1/sqrt(x^2 + y^2)
rcp r0, r0.x               // sqrt(x^2 + y^2)
mul r0, r0, c2.w           // sqrt(x^2 + y^2) * freq
texld r0, r0, s1           // Sample from 1D pulse train texture
mov r1, c3
lrp r2, r0.x, c2, r1       // Blend between light and dark wood colors
mov oC0, r2
```

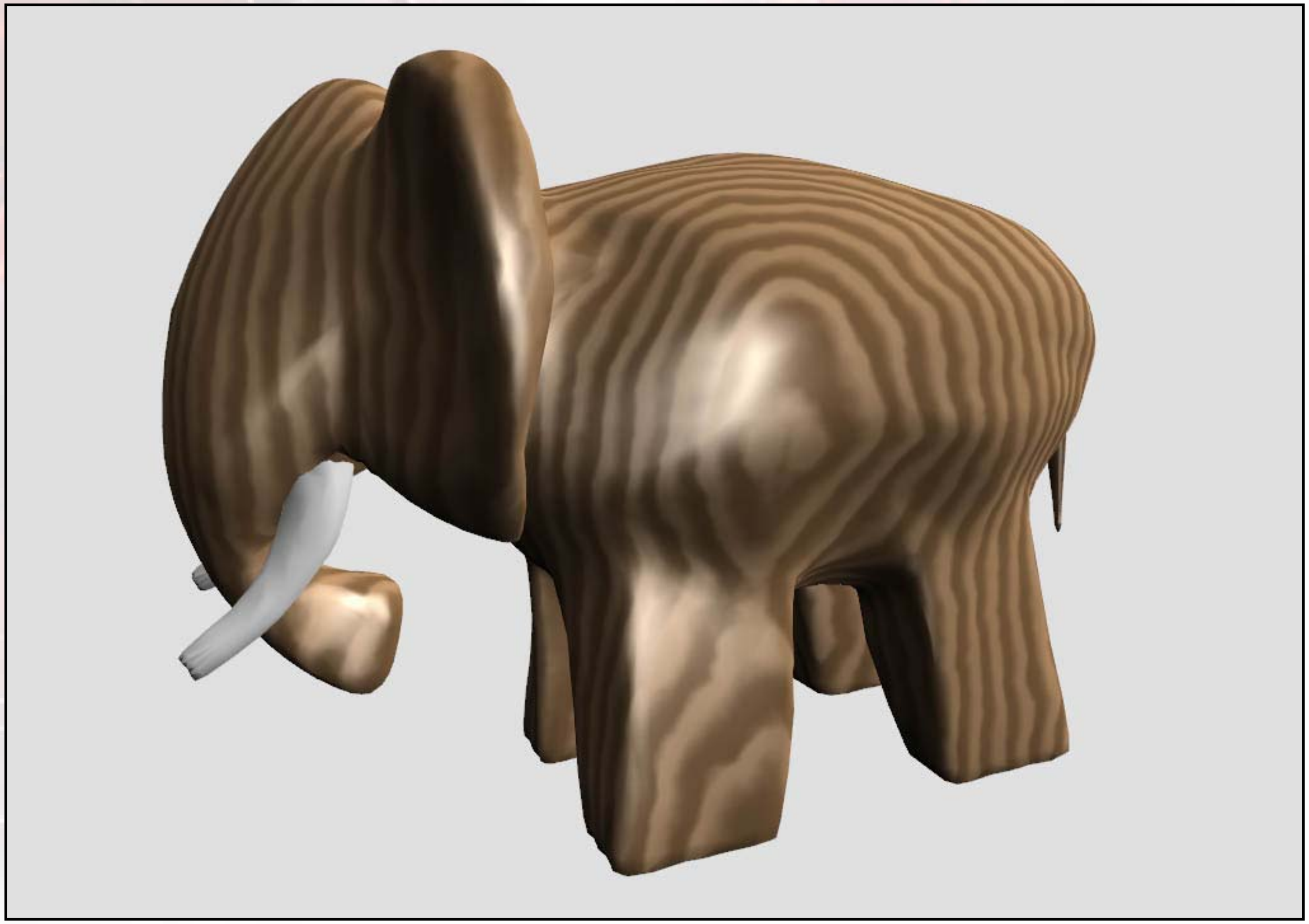
Noise and Wobble



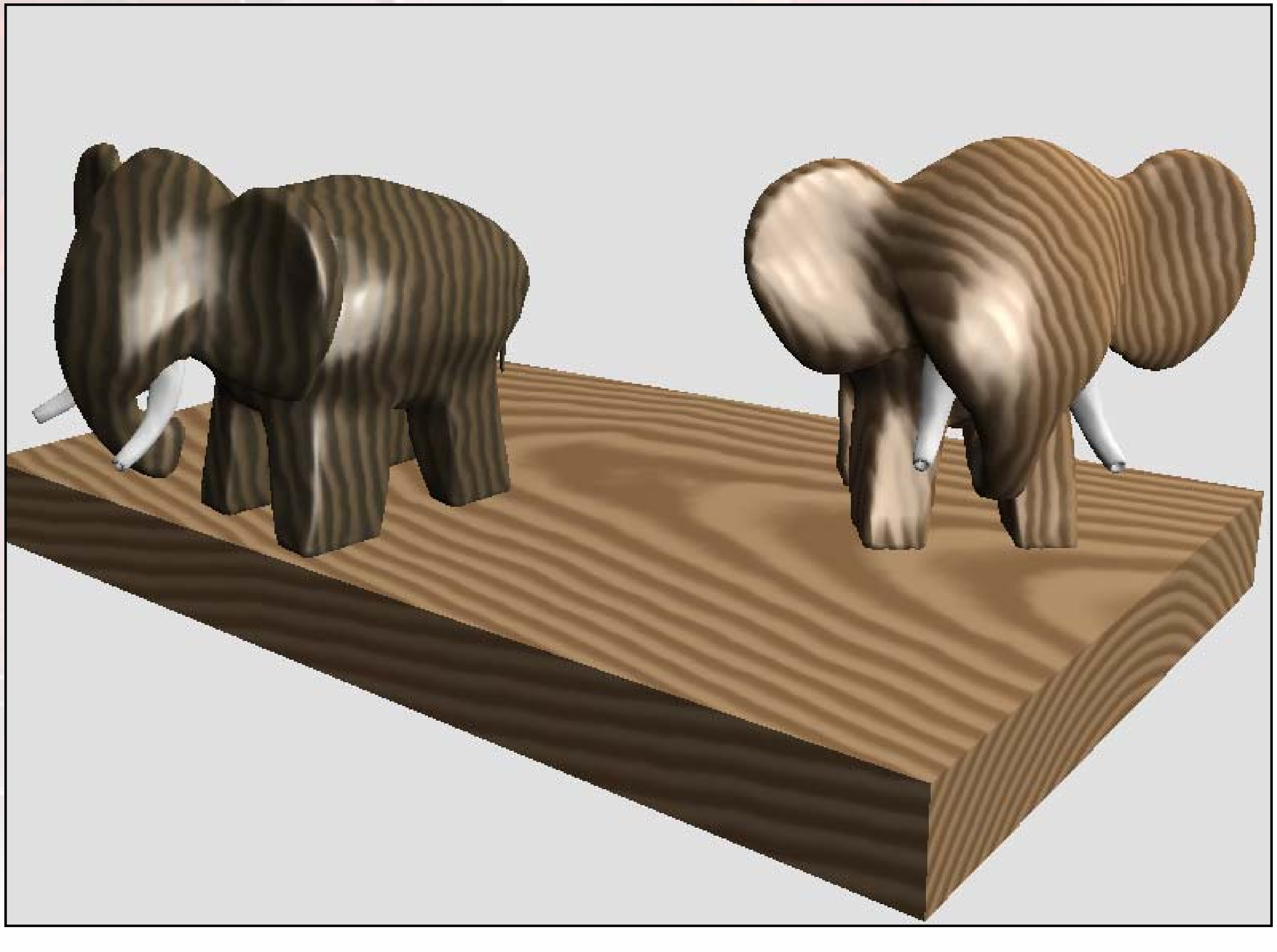
Full Shader

```
ps.2.0
def c0, 2.0f, -1.0f, 0.5f, 0.5f // scale, bias, half, X
def c1, 1.0f, 1.0f, 0.1f, 0.0f  // X, X, 0.1, zero
dcl t0.xyzw                 // xyz == Pshade (shader-space position), w == X
dcl t1.xyzw                 // xyz == Perturbed P_shade, w == X
dcl t2.xyzw                 // xyz == Perturbed P_shade, w == X
dcl t3.xyzw                 // xyz == {Pshade.z, 0, 0}, w == X
dcl t4.xyzw                 // xyz == {Pshade.z + 0.5, 0, 0}, w == X
dcl t6.xyzw                 // xyz == P_eye, w == X
dcl t7.xyzw                 // xyz == N_eye, w == X
dcl_volume s0               // Luminance-only Volume noise
dcl_2d     s1               // 1D smooth step function (blend factor in x, specular exponent in y, ...)
texld r3,  t0, s0           // Sample dX from scalar volume noise texture at P_shade
texld r4,  t1, s0           // Sample dY from scalar volume noise texture at perturbed P_shade
texld r5,  t2, s0           // Sample dZ from scalar volume noise texture at perturbed P_shade
texld r6, t3, s0            // Sample trunkWobble.x from scalar volume noise at {P_shade.z, 0, 0}
texld r7, t4, s0            // Sample trunkWobble.y from scalar volume noise at {P_shade.z + 0.5, 0, 0}
mov r3.y, r4.x              // Put dY in y
mov r3.z, r5.x              // Put dZ in z
mov r6.y, r7.x              // Move to get {trunkWobble.x, trunkWobble.y, 0}
mad r6, r6, c0.x, c0.y      // Put {trunkWobble.x, trunkWobble.y, 0} in -1..+1 range
mad r3, r3, c0.x, c0.y      // Put noise in -1..+1 range
mad r7, c3.w, r3, t0        // Scale noise by amplitude and add to P_shade to warp the domain
mad r7, c4.w, r6, r7        // Scale {trunkWobble.x, trunkWobble.y, 0} by amplitude and add in
dp2add r0, r7, r7, c1.w     // x^2 + y^2 + 0
rsq r0, r0.x                // 1/sqrt(x^2 + y^2)
rcp r0, r0.x                // sqrt(x^2 + y^2)
mul r0, r0, c2.w            // sqrt(x^2 + y^2) * freq
texld r0, r0, s1            // Sample from 1D pulse train texture
mov r1, c3
lrp r2, r0.x, c2, r1        // Blend between light and dark wood colors
sub r4, c4, t6              // Compute normalized vector from vertex to light in eye space  (L_eye)
dp3 r5.w, r4, r4            //
rsq r5.w, r5.w              //
mul r4, r4, r5.w            // L_eye
dp3 r6.w, t7, t7            // Normalize the interpolated normal
rsq r6.w, r6.w              //
mul r5, t7, r6.w            // N_eye
dp3 r3.w, t6, t6            // Compute normalized vector from the eye to the vertex
rsq r3.w, r3.w              //
mul r3, -t6, r3.w           // V_eye
add r6, r3, r5              // Compute Eye-Space HalfAngle (L_eye+V_eye)/|L_eye+V_eye|
dp3 r6.w, r6, r6
rsq r6.w, r6.w
mul r6, r6, r6.w            // H_eye
dp3_sat r6, r5, r6          // N.H
mad r0.z, r0.z, c5.z, c5.w  // scale and bias wood ring pulse to specular exponent range
pow r6, r6.x, r0.z          // (N.H)^k
dp3 r5, r4, r5              // Non-clamped N.L
mad_sat r5, r5, c0.z, c0.z  // "Half-Lambert" trick for more pleasing diffuse term
mul r6, r6, r0.y            // Gloss the highlight with the ramp texture
mad r2, r5, r2, r6          // N.L * procedural albedo + specular
mov oC0, r2
```

With Phong Shading



Final Scene



High Level Shading Language

- New in DirectX® 9
- C-like language
- Supports Pixel and Vertex Shaders
  - Specify which target to compile to
    - vs_1_1 or vs_2_0
    - ps_1_1, ps_1_2, ps_1_3, ps_1_4 or ps_2_0
  - Applies optimizations to generate standard asm output which is passed to the API/DDI

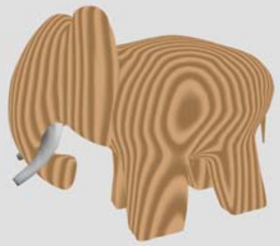
HLSL Usage

- D3DX routines
- Microsoft **`fxc`** Command Line Compiler
  - -Zi - enable debugging info
  - -Vd - disable shader validation
  - -Od - disable optimizations
  - -T *target* - target instruction set (default: vs_2_0)
    - Can be one of vs_1_1, vs_2_0, ps_1_1, ps_1_2, ps_1_3, ps_1_4 or ps_2_0
  - -E *name* - entrypoint name (default: main)
  - -Fc - output listing of generated code
  - -Fh - output header containing generated code
  - -D *id = text*   define macro
- Example usage:
  - **`fxc -T ps_2_0 -Fc -Vd -E hlsl_rings ProceduralWood.fxl`**

HLSL Versions of Wood Shaders

- `hlsl_rings()`
- `hlsl_noise()`
- `hlsl_noisy_rings()`
- `hlsl_noisy_wobble_rings()`
- `hlsl_wood()`
- `hlsl_ivory()`

hlsl_rings()



```
float4 hlsl_rings (float4 Pshade : TEXCOORD0) : COLOR
{
    float scaledDistFromZAxis = sqrt(dot(Pshade.xy, Pshade.xy))*psConst2.w;

    float4 blendFactor = tex2D (PulseTrainSampler, scaledDistFromZAxis.xx);

    return psConst2 * blendFactor.x + psConst3 * (1 - blendFactor.x);
}
```

asm generated for hlsl_rings()

```
ps_2_0
def c2, 1, 0, 0, 0
def c3, 0, 1, 0, 0
dcl t0.xy
dcl_2d s0
dp2add r0.w, t0, t0, c3.x
rsq r1.w, r0.w
mul r7.w, r1.w, r0.w
mul r2.w, c0.w, r7.w
mov r1.xy, r2.w
texld r9, r1, s0
add r9.w, -r9.x, c2.x
mul r4, c1, r9.w
mad r6, c0, r9.x, r4
mov oC0, r6
```

Output of HLSL compiler

9 ALU Instructions

```
ps_2_0
def c0, 2, -1, 0.5, 0.5
def c1, 1, 1, 0.1, 0
dcl t0.xyzw
dcl_2d s1
dp2add r0, t0, t0, c1.w
rsq r1.x, r0.x
mul r0, r1.x, r0.x
mul r0, r0, c2.w

texld r0, r0, s1
mov r1, c3
lrp r2, r0.x, c2, r1 ← 2 slots
mov oC0, r2
```

Handwritten asm

8 ALU Instructions

hlsl_noise()

```
float4 hlsl_noise (float3 Pshade0 : TEXCOORD0, float3 Pshade1 :
                   TEXCOORD1, float3 Pshade2 : TEXCOORD2) : COLOR
{
    float red;
    float green;
    float blue;

    red   = tex3D (NoiseSampler, Pshade0);
    green = tex3D (NoiseSampler, Pshade1);
    blue  = tex3D (NoiseSampler, Pshade2);

    return float4 (red, green, blue, 0.0f);
}
```

asm generated for hlsl_noise()

```
ps_2_0
def c0, 0, 0, 0, 0

dcl t0.xyz
dcl t1.xyz
dcl t2.xyz
dcl_volume s0
texld r0, t0, s0
mov r7.x, r0.x
texld r2, t1, s0
mov r7.y, r2.x
texld r9, t2, s0
mov r7.z, r9.x
mov r7.w, c0.x
mov oC0, r7
```

Output of HLSL compiler

5 ALU Instructions

```
ps_2_0
def c0, 2, -1, 0.5, 0.5
def c1, 1, 1, 0.1, 0
dcl t0.xyzw
dcl t1.xyzw
dcl t2.xyzw
dcl_volume s0
texld r3,  t0, s0
texld r4,  t1, s0
texld r5,  t2, s0
mov r3.y, r4.x
mov r3.z, r5.x
mov oC0, r3
```

Handwritten asm

3 ALU Instructions

hlsl_noisy_rings()

```
float4 hlsl_noisy_rings (float3 Pshade0  : TEXCOORD0,
                         float3 Pshade1  : TEXCOORD1,
                         float3 Pshade2  : TEXCOORD2) : COLOR
{
 float3 coloredNoise;

    // Construct colored noise from three samples
    coloredNoise.x = tex3D (NoiseSampler, Pshade0);
    coloredNoise.y = tex3D (NoiseSampler, Pshade1);
    coloredNoise.z = tex3D (NoiseSampler, Pshade2);

    // Make signed
    coloredNoise = coloredNoise * 2.0f - 1.0f;

    // Scale noise and add to Pshade
    float3 noisyPshade = Pshade0 + coloredNoise * psConst3.w;

    float scaledDistFromZAxis = sqrt(dot(noisyPshade.xy, noisyPshade.xy)) * psConst2.w;

    float4 blendFactor = tex2D (PulseTrainSampler, float2 (0.0f, scaledDistFromZAxis));

    return psConst2 * blendFactor.x + psConst3 * (1 - blendFactor.x);
}
```

asm generated for hlsl_noisy_rings()

```
ps_2_0
def c2, - 1, 1, 0, 0
def c3, 0, 1, 0, 0
dcl t0.xyz
dcl t1.xyz
dcl_volume s0
dcl_2d s1


texld r0, t0, s0
add r7.x, r0.x, r0.x
texld r2, t1, s0
add r7.y, r2.x, r2.x
add r9.xy, r7, c2.x
mad r11.xy, r9, c1.w, t0
dp2add r6.w, r11, r11, c3.x
rsq r2.w, r6.w
mul r1.w, r2.w, r6.w
mul r8.y, c0.w, r1.w
mov r8.x, c2.w
texld r3, r8, s1
add r3.w,-  r3.x, c2.y
mul r10, c1, r3.w
mad r0, c0, r3.x, r10
mov oC0, r0
```

HLSL generates 13 ALU Instructions

```
ps_2_0
def c0, 2.0f,-  10f, 0.5f, 0.5f
def c1, 1.0f, 1.0f, 0.1f, 0.0f
dcl t0.xyzw
dcl t1.xyzw
dcl t2.xyzw
dcl_volume s0
dcl_2d     s1
texld r3,  t0, s0
texld r4,  t1, s0
texld r5,  t2, s0
mov r3.y, r4.x
mov r3.z, r5.x
mad r3, r3, c0.x, c0.y
mad r7, c3.w, r3, t0
dp2add r0, r7, r7, c1.w
rsq r0, r0.x
rcp r0, r0.x
mul r0, r0, c2.w
texld r0, r0, s1
mov r1, c3
lrp r2, r0.x, c2, r1  ← 2 slots
mov oC0, r2
```

Handwritten asm is 12 instructions

hlsl_noisy_wobble_rings()

```
float4 hlsl_noisy_wobble_rings (float3 Pshade0  : TEXCOORD0, float3 Pshade1  : TEXCOORD1,
                                float3 Pshade2  : TEXCOORD2, float3 zWobble0 : TEXCOORD3,
                                float3 zWobble1 : TEXCOORD4) : COLOR
{
    float3 coloredNoise;
    float3 wobble;

    // Construct colored noise from three samples
    coloredNoise.x = tex3D (NoiseSampler, Pshade0);
    coloredNoise.y = tex3D (NoiseSampler, Pshade1);
    coloredNoise.z = tex3D (NoiseSampler, Pshade2);

    wobble.x = tex3D (NoiseSampler, zWobble0);
    wobble.y = tex3D (NoiseSampler, zWobble1);
    wobble.z = 0.5f;

    // Make signed
    coloredNoise = coloredNoise * 2.0f - 1.0f;
    wobble       = wobble       * 2.0f - 1.0f;

    // Scale noise and add to Pshade
    float3 noisyWobblyPshade = Pshade0 + coloredNoise * psConst3.w + wobble * psConst4.w ;

    float scaledDistFromZAxis = sqrt(dot(noisyWobblyPshade.xy, noisyWobblyPshade.xy)) * psConst2.w;

    // Lookup blend factor from pulse train
    float4 blendFactor = tex2D (PulseTrainSampler, float2 (0.0f, scaledDistFromZAxis));

    // Blend wood colors together
    return psConst2 * blendFactor.x + psConst3 * (1 - blendFactor.x);
}
```

asm generated for hlsl_noisy_wobble_rings ()

```
...
texld r0, t0, s0
add r7.x, r0.x, r0.x
texld r2, t1, s0
add r7.y, r2.x, r2.x
add r9.xy, r7, c3.x
mad r11.xy, r9, c1.w, t0
texld r6, t3, s0
add r11.z, r6.x, r6.x
texld r1, t4, s0
add r11.w, r1.x, r1.x
add r11.zw, r11, c3.x
mad r8.x, r11.z, c2.w, r11.x
mad r8.y, r11.w, c2.w, r11.y
dp2add r3.w, r8, r8, c4.x
rsq r1.w, r3.w
mul r10.w, r1.w, r3.w
mul r5.y, c0.w, r10.w
mov r5.x, c3.w
texld r0, r5, s1
add r0.w,-  r0.x, c3.y
mul r2, c1, r0.w
mad r9, c0, r0.x, r2
mov oC0, r9
```

HLSL generates 18 ALU Instructions

```
...
texld r3,  t0, s0
texld r4,  t1, s0
texld r5,  t2, s0
texld r6, t3, s0
texld r7, t4, s0
mov r3.y, r4.x
mov r3.z, r5.x
mov r6.y, r7.x
mad r6, r6, c0.x, c0.y
mad r3, r3, c0.x, c0.y
mad r7, c3.w, r3, t0
mad r7, c4.w, r6, r7
dp2add r0, r7, r7, c1.w
rsq r0, r0.x
rcp r0, r0.x
mul r0, r0, c2.w
texld r0, r0, s1
mov r1, c3
lrp r2, r0.x, c2, r1  ← 2 slots
mov oC0, r2
```

Handwritten asm is 15 instructions

hlsl_wood()

```
float4 hlsl_wood (float3 Pshade0  : TEXCOORD0, float3 Pshade1  : TEXCOORD1, float3 Pshade2  : TEXCOORD2,
           float3 zWobble0 : TEXCOORD3, float3 zWobble1 : TEXCOORD4, float3 Peye : TEXCOORD6, float3 Neye : TEXCOORD7) : COLOR
{
    float3 coloredNoise;
    float3 wobble;

    coloredNoise.x = tex3D (NoiseSampler, Pshade0); // Construct colored noise from three samples
    coloredNoise.y = tex3D (NoiseSampler, Pshade1);
    coloredNoise.z = tex3D (NoiseSampler, Pshade2);

    wobble.x = tex3D (NoiseSampler, zWobble0);
    wobble.y = tex3D (NoiseSampler, zWobble1);
    wobble.z = 0.5f;

    coloredNoise = coloredNoise * 2.0f - 1.0f; // Make signed
    wobble       = wobble       * 2.0f - 1.0f;

    // Scale noise and add to Pshade
    float3 noisyWobblyPshade = Pshade0 + coloredNoise * psConst3.w + wobble * psConst4.w ;

    float scaledDistFromZAxis = sqrt(dot(noisyWobblyPshade.xy, noisyWobblyPshade.xy)) * psConst2.w;

    // Lookup blend factor from pulse train
    float4 blendFactor = tex2D (PulseTrainSampler, float2 (0.0f, scaledDistFromZAxis));

    float3 albedo =  psConst2 * blendFactor.x + psConst3 * (1 - blendFactor.x); // Blend wood colors together

    // Compute normalized vector from vertex to light in eye space  (Leye)
    float3 Leye = (psConst4 - Peye) / len(psConst4 - Peye);

    Neye = Neye / len(Neye);                                              // Normalize interpolated normal
    float3 Veye = -(Peye / len(Peye));                                    // Compute Veye
    float3 Heye = (Leye + Veye) / len(Leye + Veye);                       // Compute half-angle
    float  NdotH = clamp(dot(Neye, Heye), 0.0f, 1.0f);                   // Compute N.H
    float  k = blendFactor.z;                                             // Scale and bias exponent from pulse train
    float  specular = tex2D (VariablSpecularSampler, float2 (NdotH, k)); // Evaluate (N.H)^k via dependent read
    float  NdotL = dot(Neye, Leye);                                       // N.L
    float  diffuse = NdotL * 0.5f + 0.5f;                                 // "Half-Lambert" technique for more pleasing diffuse
    float  gloss = blendFactor.y;                                         // gloss the specular term

    return diffuse * float4 (albedo.r, albedo.g, albedo.b, 0.0f) + specular * gloss;
}
```

asm generated for hlsl_wood ()

```
...
texld r0, t0, s0
add r7.x, r0.x, r0.x
texld r2, t1, s0
add r7.y, r2.x, r2.x
add r9.xy, r7, c3.x
mad r11.xy, r9, c1.w, t0
texld r6, t3, s0
add r11.z, r6.x, r6.x
texld r1, t4, s0
add r11.w, r1.x, r1.x
add r11.zw, r11, c3.x
mad r8.x, r11.z, c2.w, r11.x
mad r8.y, r11.w, c2.w, r11.y
dp2add r3.w, r8, r8, c4.x
rsq r2.w, r3.w
mul r10.w, r2.w, r3.w
mul r5.y, c0.w, r10.w
mov r5.x, c3.w
texld r0, r5, s1
add r7.xyz, c2, -t6
dp3 r0.w, r7, r7
rsq r0.w, r0.w
mul r9.xyz, r7, r0.w
dp3 r0.w, t6, t6
rsq r0.w, r0.w
mad r1.xyz, r0.w, -t6, r9
dp3 r9.w, r1, r1
rsq r1.w, r9.w
mul r11.xyz, r1, r1.w
dp3 r9.w, t7, t7
rsq r11.w, r9.w
mul r8.xyz, r11.w, t7
dp3_sat r3.x, r8, r11
mov r3.y, r0.z
texld r10, r3, s2
mul r8.w, r10.x, r0.y
add r9.w, -r0.x, c3.y
mul r5, c1, r9.w
mad r7, c0, r0.x, r5
dp3 r8.z, r8, r9
mad r8.z, r8.z, c3.z, c3.z
mad r6, r8.z, r7, r8.w
mov oC0, r6
```

```
...
texld r3,  t0, s0
texld r4,  t1, s0
texld r5,  t2, s0
texld r6, t3, s0
texld r7, t4, s0
mov r3.y, r4.x
mov r3.z, r5.x
mov r6.y, r7.x
mad r6, r6, c0.x, c0.y
mad r3, r3, c0.x, c0.y
mad r7, c3.w, r3, t0
mad r7, c4.w, r6, r7
dp2add r0, r7, r7, c1.w
rsq r0, r0.x
rcp r0, r0.x
mul r0, r0, c2.w
texld r0, r0, s1
mov r1, c3
lrp r2, r0.x, c2, r1  ← 2 slots
sub r4, c4, t6
dp3 r5.w, r4, r4
rsq r5.w, r5.w
mul r4, r4, r5.w
dp3 r6.w, t7, t7
rsq r6.w, r6.w
mul r5, t7, r6.w
dp3 r3.w, t6, t6
rsq r3.w, r3.w
mul r3, -t6, r3.w
add r6, r3, r4
dp3 r6.w, r6, r6
rsq r6.w, r6.w
mul r6, r6, r6.w
dp3_sat r6, r5, r6
mov r6.y, r0.z
texld r6, r6, s2
dp3 r5, r4, r5
mad_sat r5, r5, c0.z, c0.z
mul r6, r6, r0.y
mad r2, r5, r2, r6
mov oC0, r2
```

HLSL generates 37 ALU Instructions

Handwritten asm is 35 instructions

hlsl_ivory()

```
float4 hlsl_ivory (float3 Peye : TEXCOORD0, float3 Neye : TEXCOORD1) : COLOR
{
    // Compute normalized vector from vertex to light in eye space  (Leye)
    float3 Leye = (psConst4 - Peye) / len(psConst4 - Peye);

    // Normalize interpolated normal
    Neye = Neye / len(Neye);

    // Compute Veye
    float3 Veye = -(Peye / len(Peye));

    // Compute half-angle
    float3 Heye = (Leye + Veye) / len(Leye + Veye);

    // N.L
    float NdotL = dot(Neye, Leye);

    // "Half-Lambert" technique for more pleasing diffuse term
    float diffuse = NdotL * 0.5f + 0.5f;

    // Compute N.H
    float NdotH = clamp(dot(Neye, Heye), 0.0f, 1.0f);

    float NdotH_2  = NdotH    * NdotH;
    float NdotH_4  = NdotH_2  * NdotH_2;
    float NdotH_8  = NdotH_4  * NdotH_4;
    float NdotH_16 = NdotH_8  * NdotH_8;
    float NdotH_32 = NdotH_16 * NdotH_16;

    return NdotH_32 * NdotH_32 + diffuse;
}
```

asm generated for hlsl_ivory()

```
...
add r7.xyz, c0, - t0
dp3 r7.w, r7, r7
rsq r7.w, r7.w
mul r2.xyz, r7, r7.w
dp3 r2.w, t0, t0
rsq r2.w, r2.w
mad r11.xyz, r2.w,-  t0, r2
dp3 r2.w, r11, r11
rsq r2.w, r2.w
mul r6.xyz, r11, r2.w
dp3 r2.w, t1, t1
rsq r2.w, r2.w
mul r1.xyz, r2.w, t1
dp3_sat r2.w, r1, r6
mul r1.w, r2.w, r2.w
mul r2.w, r1.w, r1.w
mul r1.w, r2.w, r2.w
mul r2.w, r1.w, r1.w
mul r1.w, r2.w, r2.w
mul r2.w, r1.w, r1.w
dp3 r2.z, r1, r2
mad r2.z, r2.z, c1.x, c1.x
add r8.w, r2.w, r2.z
mov r0, r8.w
mov oC0, r0
```

HLSL generates 25 ALU Instructions

```
...
sub r4, c4, t0
dp3 r5.w, r4, r4
rsq r5.w, r5.w
mul r4, r4, r5.w
dp3 r6.w, t1, t1
rsq r6.w, r6.w
mul r5, t1, r6.w
dp3 r3.w, t0, t0
rsq r3.w, r3.w
mul r3, - t0, r3.w
add r6, r3, r4
dp3 r6.w, r6, r6
rsq r6.w, r6.w
mul r6, r6, r6.w
dp3 r7, r5, r4
mad_sat r7, r7, c0.z, c0.z
dp3_sat r6, r5, r6
mul r6, r6, r6
mul r6, r6, r6
mul r6, r6, r6
mul r6, r6, r6
mul r6, r6, r6
mad r2, r6, r6, r7
mov oC0, r2
```

Handwritten asm is 24 instructions

Use the HLSL!

- All the usual advantages of High Level Languages
  - Faster, easier development
  - Code re-use
  - Optimization
    - Current HLSL compiler is *very* good and getting better every day
- Industry standard which will run on any DirectX® shader chip
- Next rev of RenderMonkey™ will support DirectX® 9 HLSL

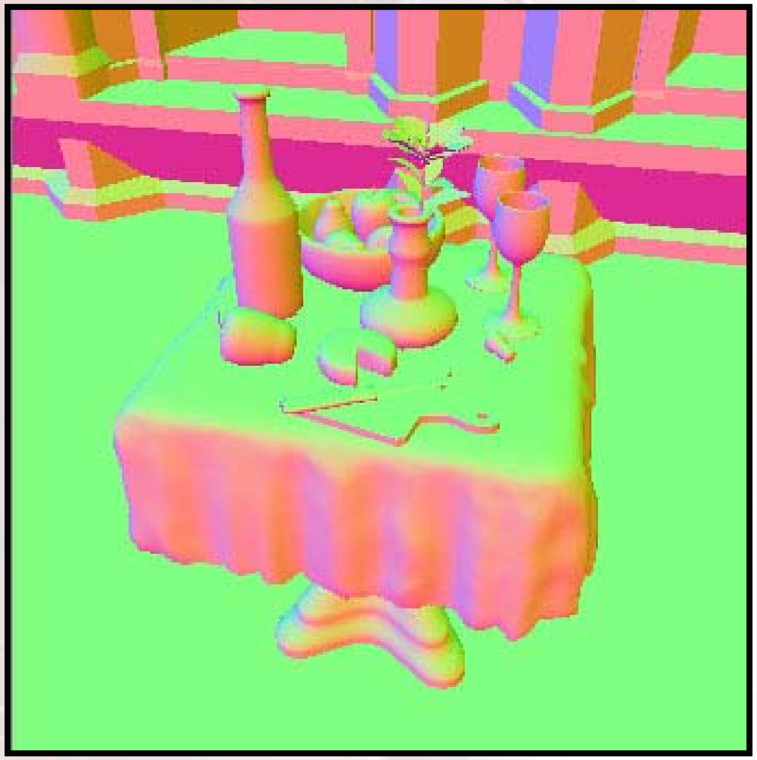
Image-Space Effects

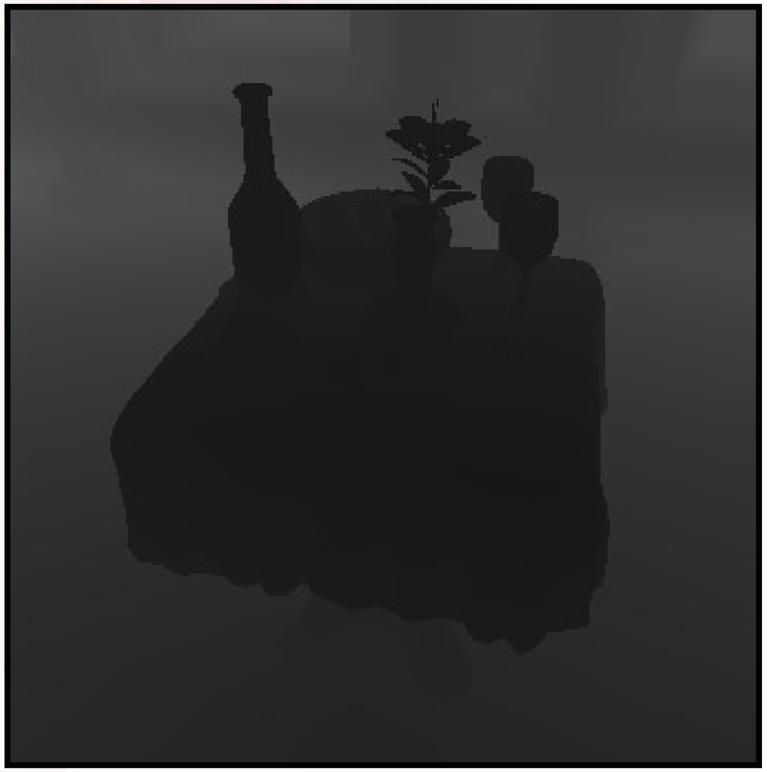
- Edge Detection for Cartoon outlining
- High Dynamic Range Rendering
- Depth of Field

Image Space Outlining for NPR

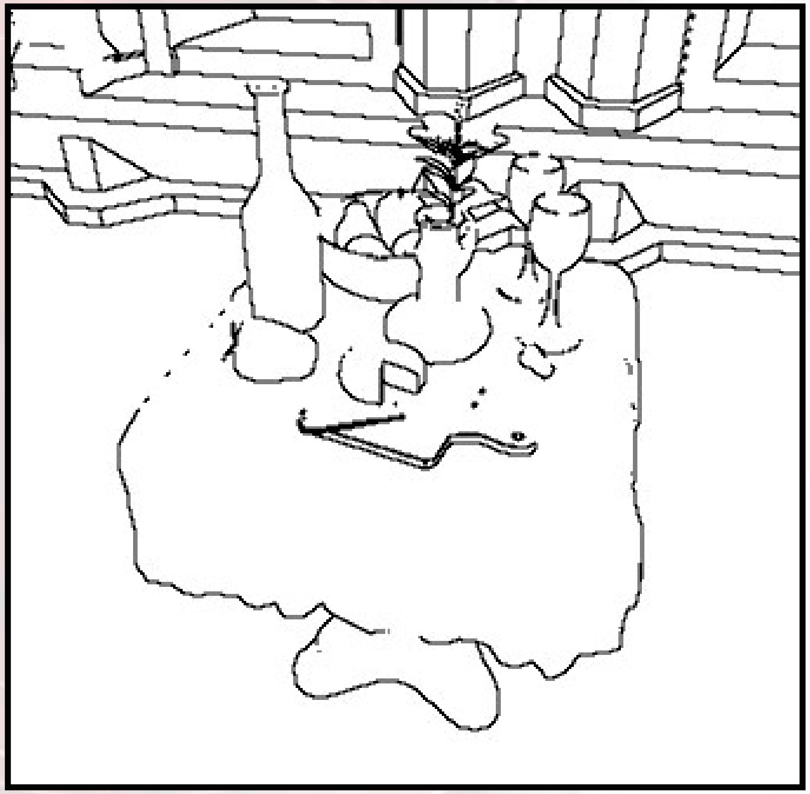
- Render alternate representation of scene into texture map
  - With the RADEON 9700, we’re able to render into up to four targets simultaneously, effectively implementing Saito and Takahashi’s G-buffer
- Run filter over image to detect edges
  - Implemented using pixel shading hardware

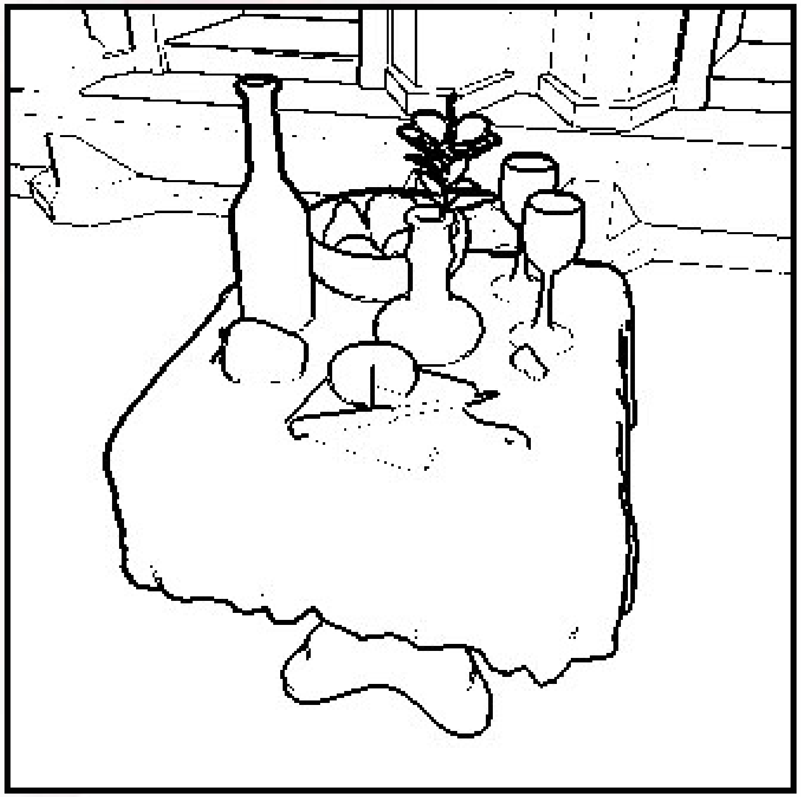
Normal and Depth


World Space Normal

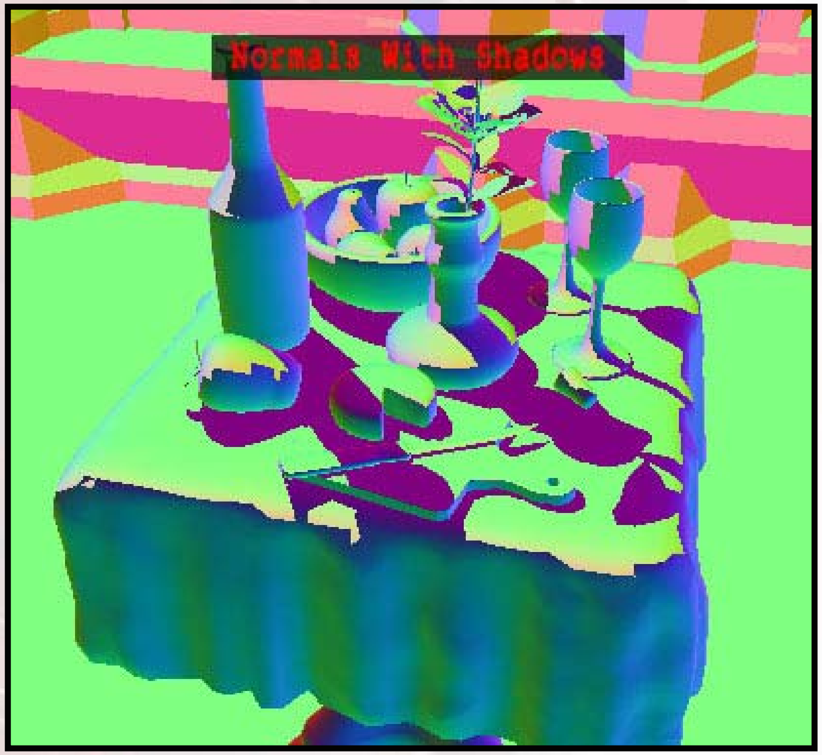

Eye Space Depth

Outlines

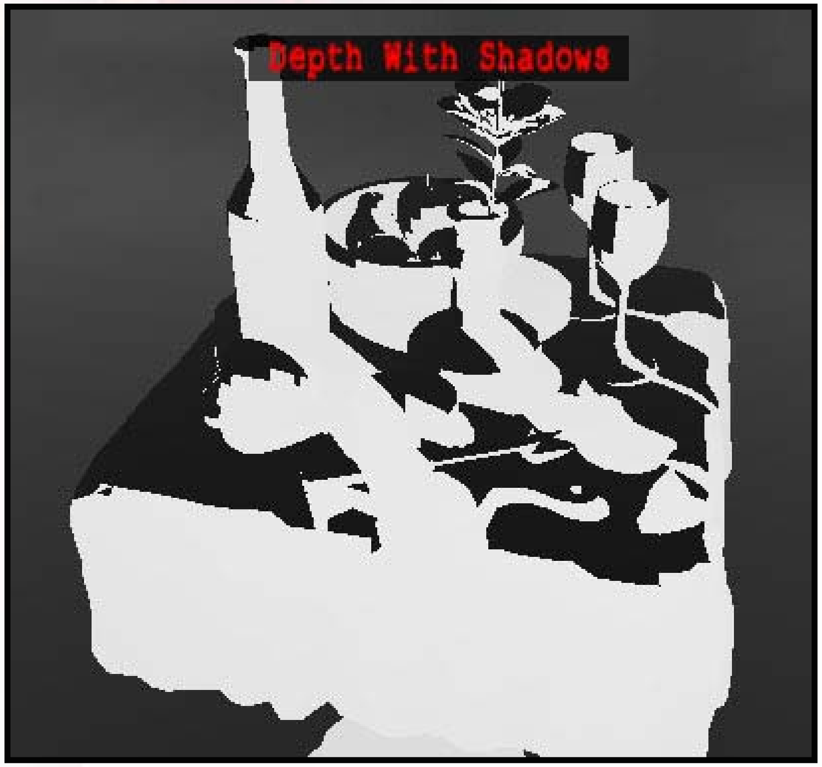

Normal Edges


Depth Edges

Normal and Depth Negated in Shadow

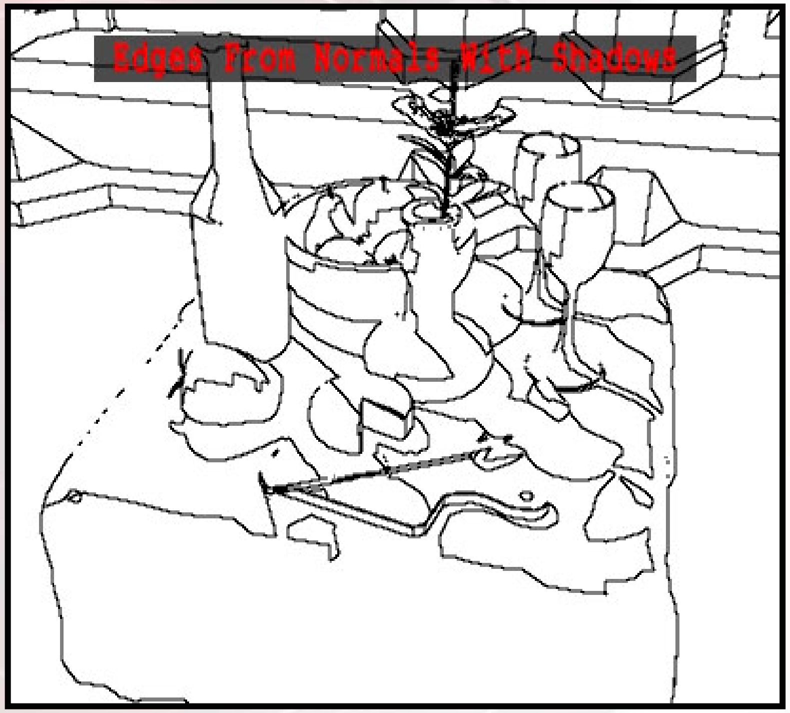


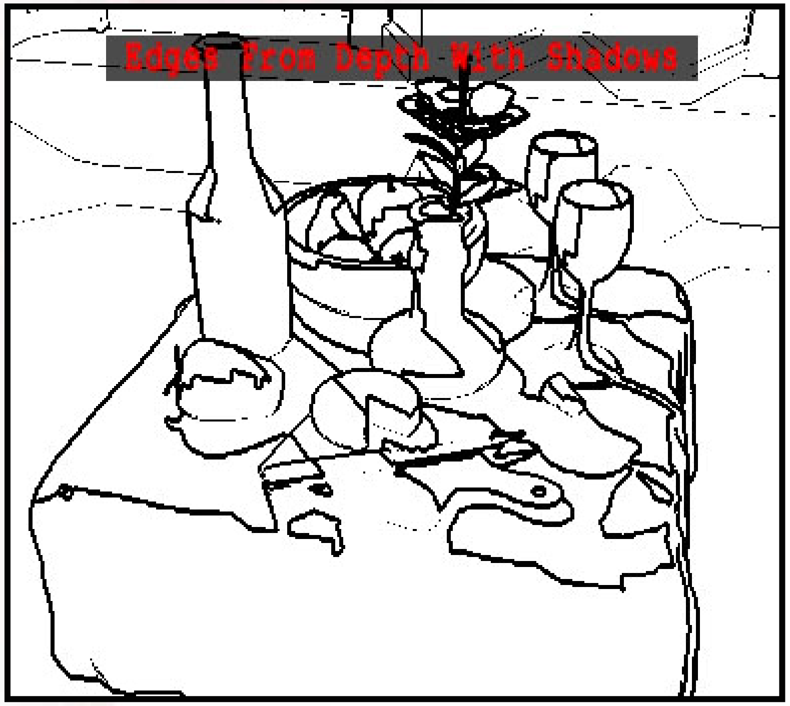
World Space Normal
Negated in Shadow



Eye Space Depth
Negated in Shadow

Normal and Depth Outlines

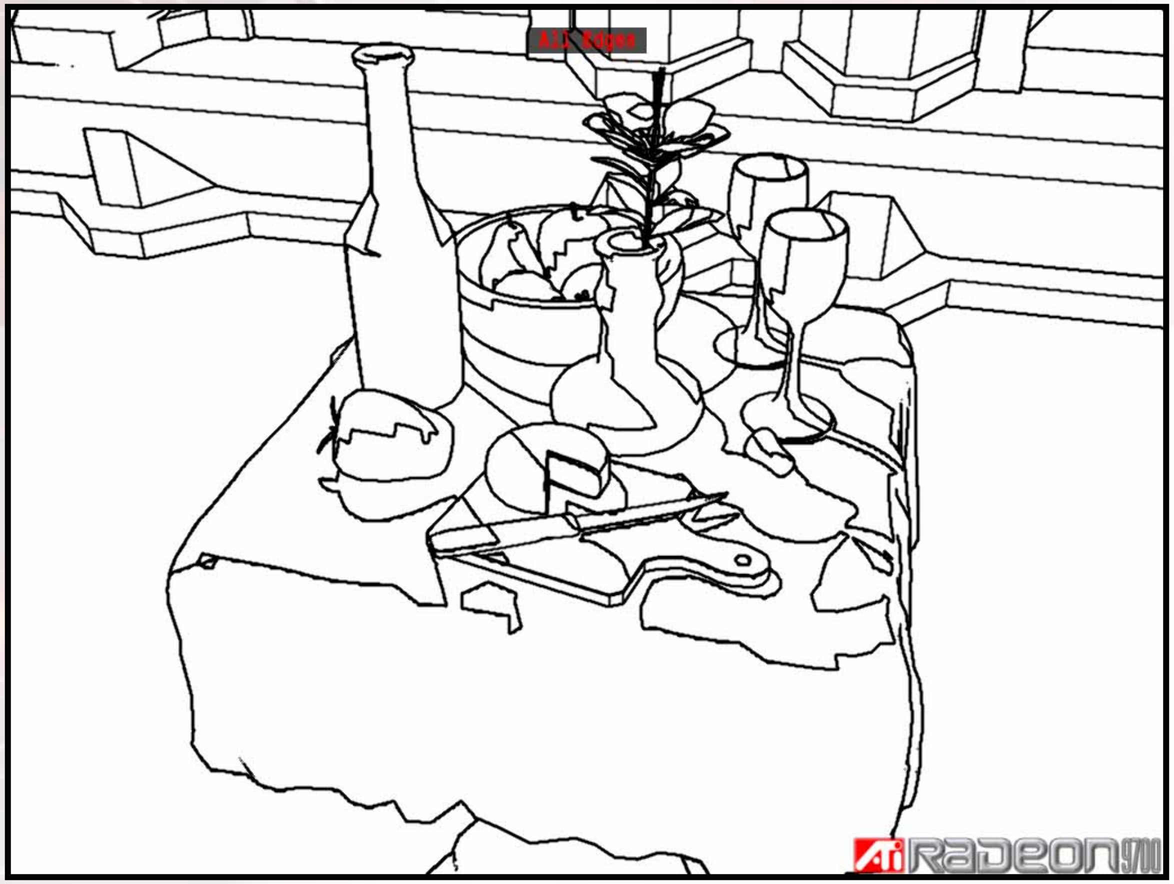




Edges from Normals

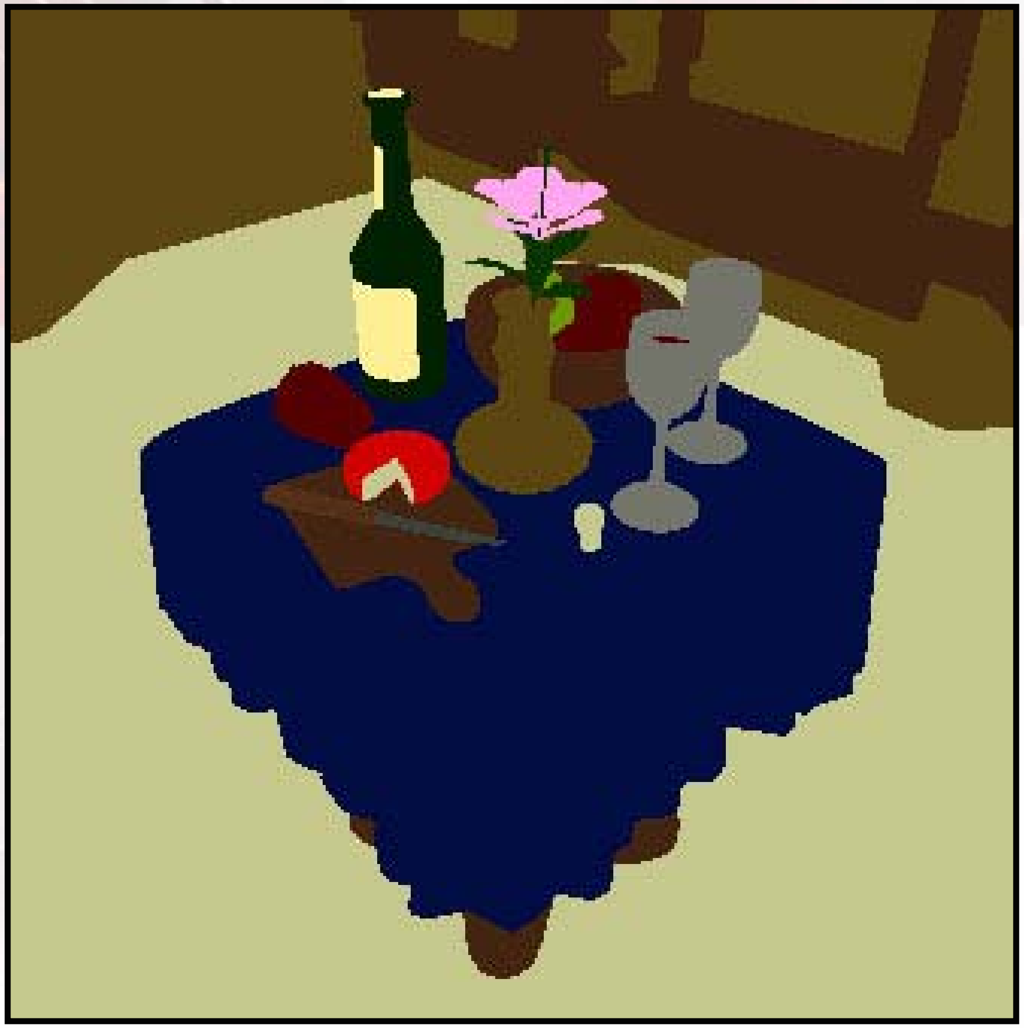
Edges from Depth

Object and Shadow Outlines

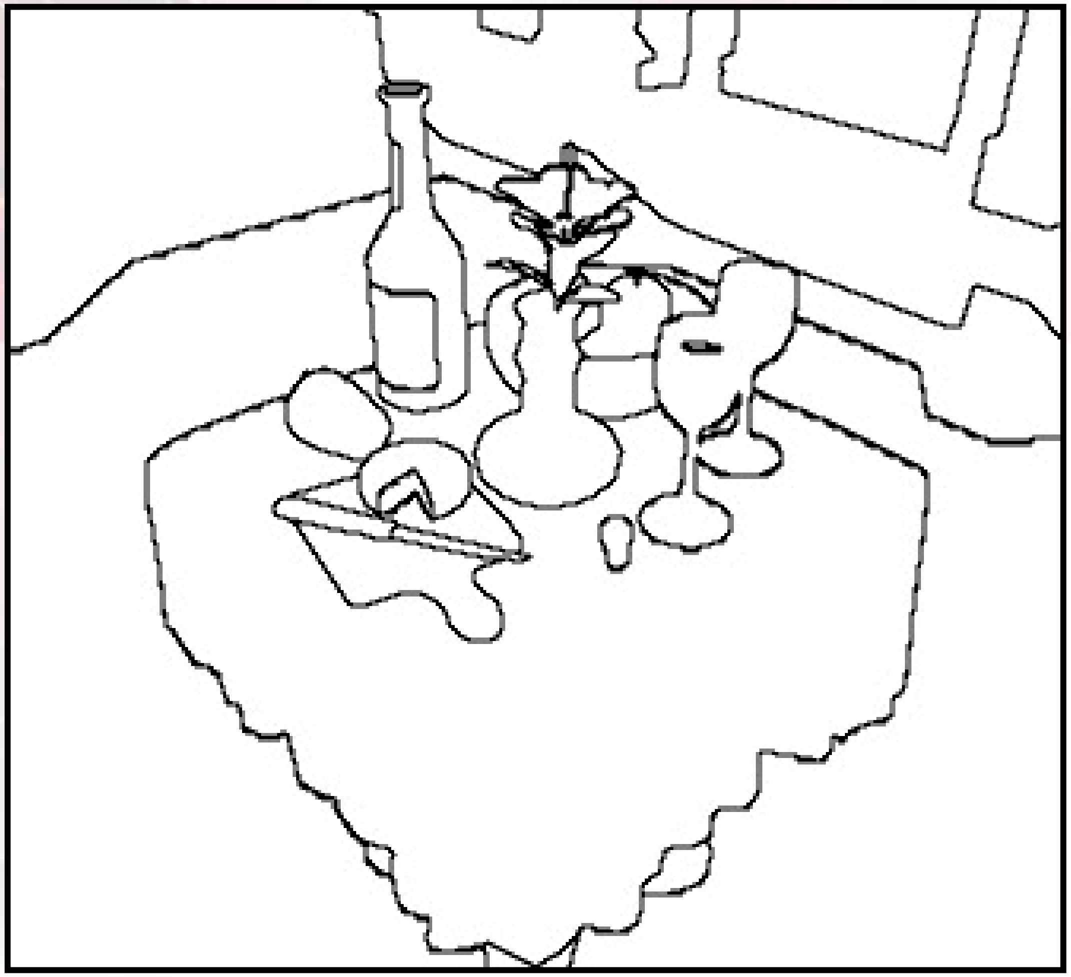


Outlines from selectively negated normals and depths

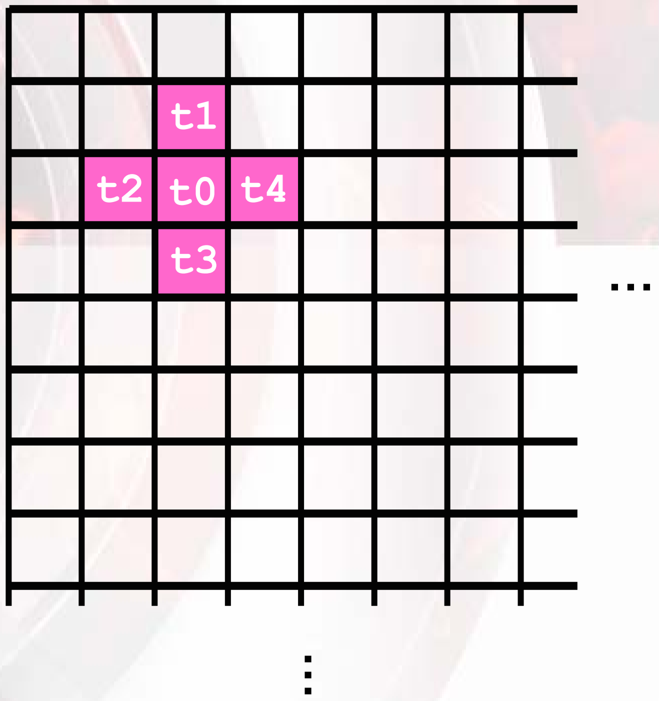
Texture Region IDs



Edges at Texture Region Boundaries



Edge Filter



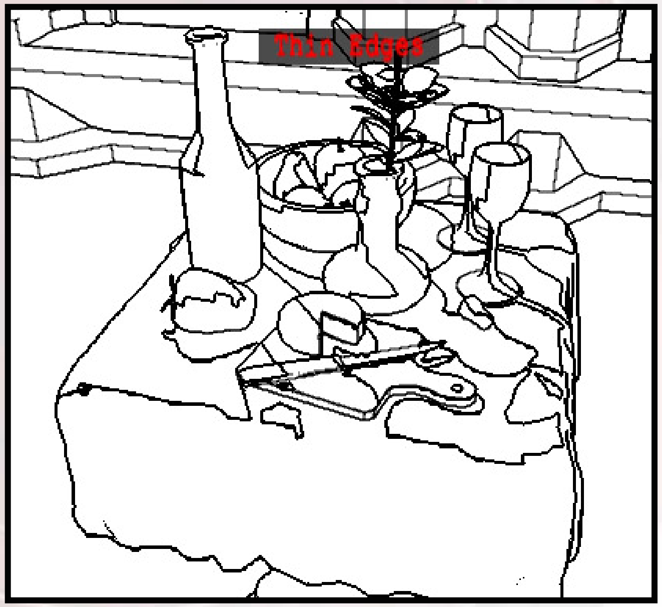
5-tap Filter

Edge Filter Code

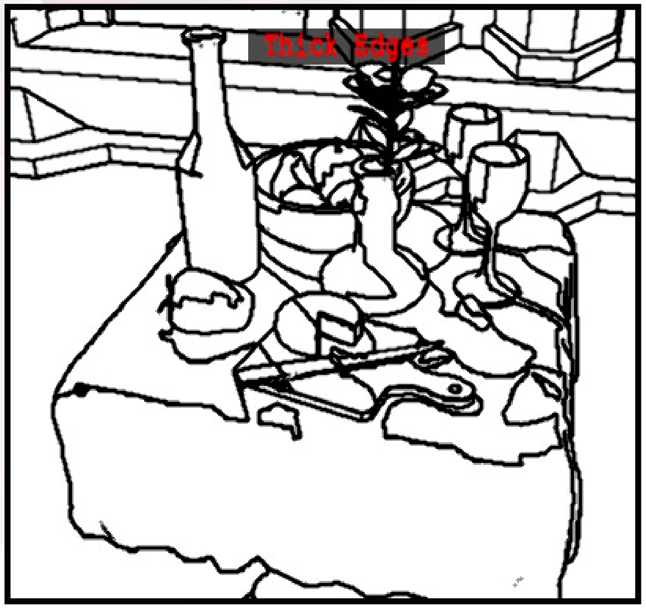
```
ps.2.0
def     c0,  0.0f, 0.80f, 0, 0
def     c3,  0, .5, 1, 2
def     c8,  0.0f, 0.0f, -0.01f, 0.0f
def     c9,  0.0f, -0.25f, 0.25f, 1.0f
def     c12, 0.0f, 0.01f, 0.0f, 0.0f
dcl_2d s0
dcl_2d s1
dcl     t0
dcl     t1
dcl     t2
dcl     t3
dcl     t4

// Sample the map five times
texld r0, t0, s0 // Center Tap
texld r1, t1, s0 // Down/Right
texld r2, t2, s0 // Down/Left
texld r3, t3, s0 // Up/Left
texld r4, t4, s0 // Up/Right

//--------------------------------
// NORMALS
//--------------------------------
mad r0.xyz, r0, c3.w, -c3.z
mad r1.xyz, r1, c3.w, -c3.z
mad r2.xyz, r2, c3.w, -c3.z
mad r3.xyz, r3, c3.w, -c3.z
mad r4.xyz, r4, c3.w, -c3.z

// Take dot products with center
dp3 r5.r, r0, r1
dp3 r5.g, r0, r2
dp3 r5.b, r0, r3
dp3 r5.a, r0, r4

// Subtract threshold
sub r5, r5, c0.g

// Make 0/1 based on threshold
cmp r5, r5, c1.g, c1.r

// detect any 1's
dp4_sat r11, r5, c3.z
mad_sat r11, r11, c1.b, c1.w
//----------------------------------
// Z
//----------------------------------
// Take four deltas
add r10.r, r0.a, -r1.a
add r10.g, r0.a, -r2.a
add r10.b, r0.a, -r3.a
add r10.a, r0.a, -r4.a
```

```
cmp r10, r10, r10, -r10         // Take absolute value
add r10, r10, c8.b              // Subtract threshold
cmp r10, r10, c1.r, c1.g        // Make black/white
dp4_sat r10, r10, c3.z          // Sum up detected pixels
mad_sat r10, r10, c1.b, c1.w    // Scale and bias result
mul r11, r11, r10               // Combine with previous

//-----------------------------------------------------
// TexIDs
//-----------------------------------------------------
// Sample the map five times
texld r0, t0, s1 // Center Tap
texld r1, t1, s1 // Down/Right
texld r2, t2, s1 // Down/Left
texld r3, t3, s1 // Up/Left
texld r4, t4, s1 // Up/Right

// Get differences in color
sub r1.rgb, r0, r1
sub r2.rgb, r0, r2
sub r3.rgb, r0, r3
sub r4.rgb, r0, r4

// Calculate magnitude of color differences
dp3 r1.r, r1, c3.z
dp3 r1.g, r2, c3.z
dp3 r1.b, r3, c3.z
dp3 r1.a, r4, c3.z

cmp r1, r1, r1, -r1             // Take absolute values
sub r1, r1, c12.g               // Subtract threshold
cmp r1, r1, c1.r, c1.g          // Make black/white
dp4_sat r10, r1, c3.z           // Total up edges
mad_sat r10, r10, c1.b, c1.w    // Scale and bias result
mul r11, r10, r11               // Combine with previous

// Output
mov oC0, r11
```

Morphology

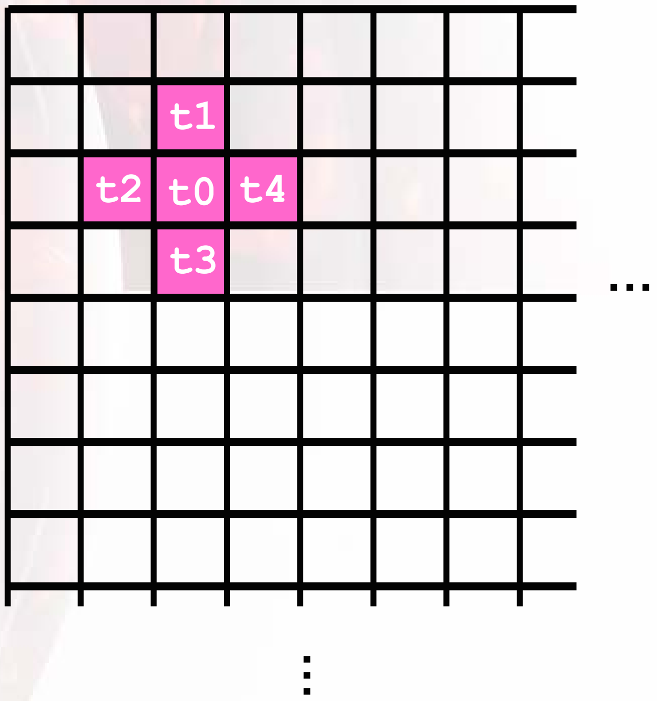


Dilate

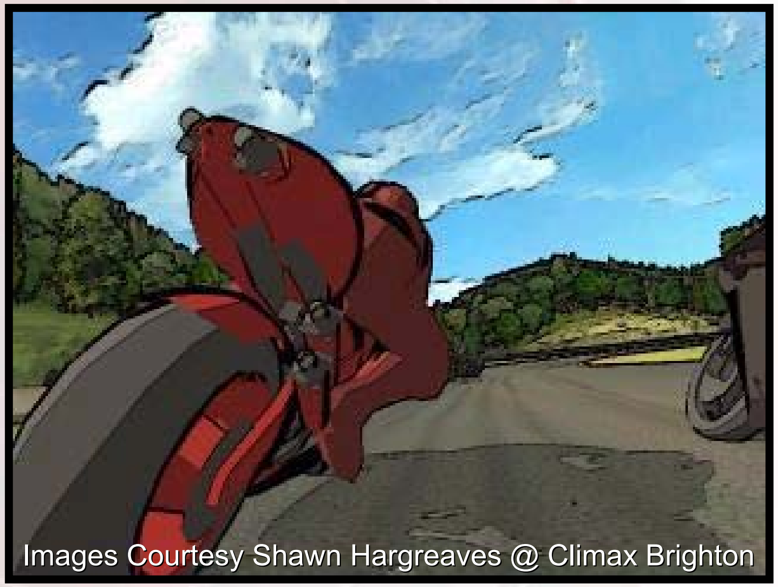


Dilation Shader

```
ps.2.0
def c0, 0, .5, 1, 2
def c1, 0.4f, -1, 5.0f, 0
dcl_2d s0
dcl     t0
dcl     t1
dcl     t2
dcl     t3
dcl     t4

// Sample the map five times
texld    r0, t0, s0 // Center Tap
texld    r1, t1, s0 // Up
texld    r2, t2, s0 // Left
texld    r3, t3, s0 // Down
texld    r4, t4, s0 // Right

// Sum the samples
add      r0, r0, r1
add      r1, r2, r3
add      r0, r0, r1
add      r0, r0, r4
mad_sat r0, r0.r, c1.r, c1.g  // Threshold
mov oC0, r0
```



Real World Example
MotoGP from Climax Brighton



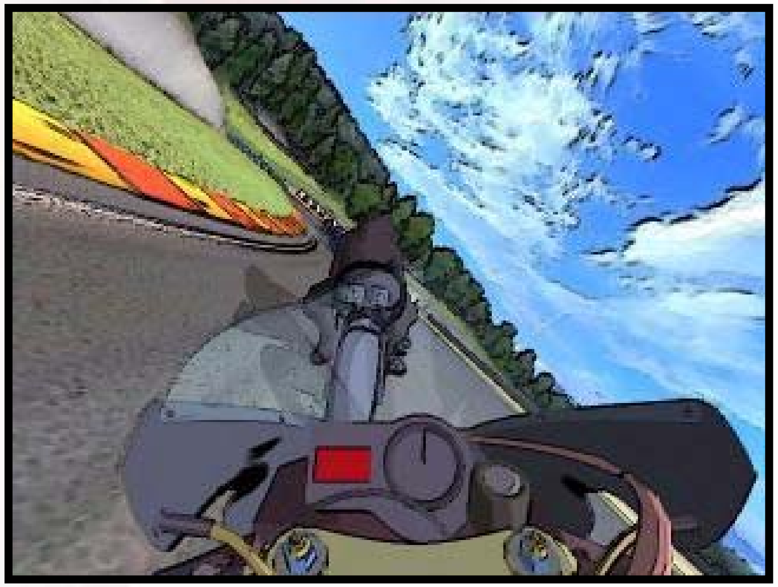
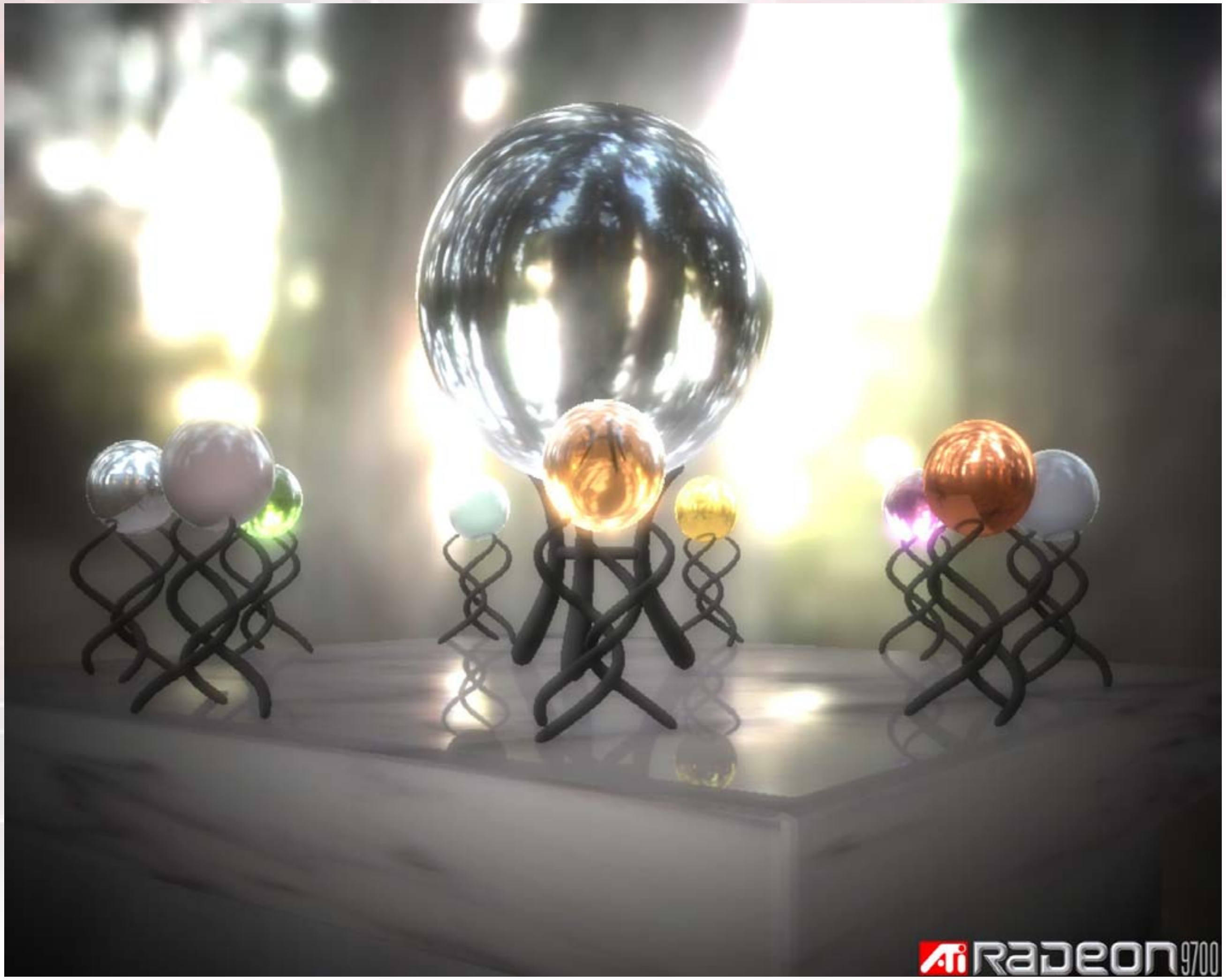
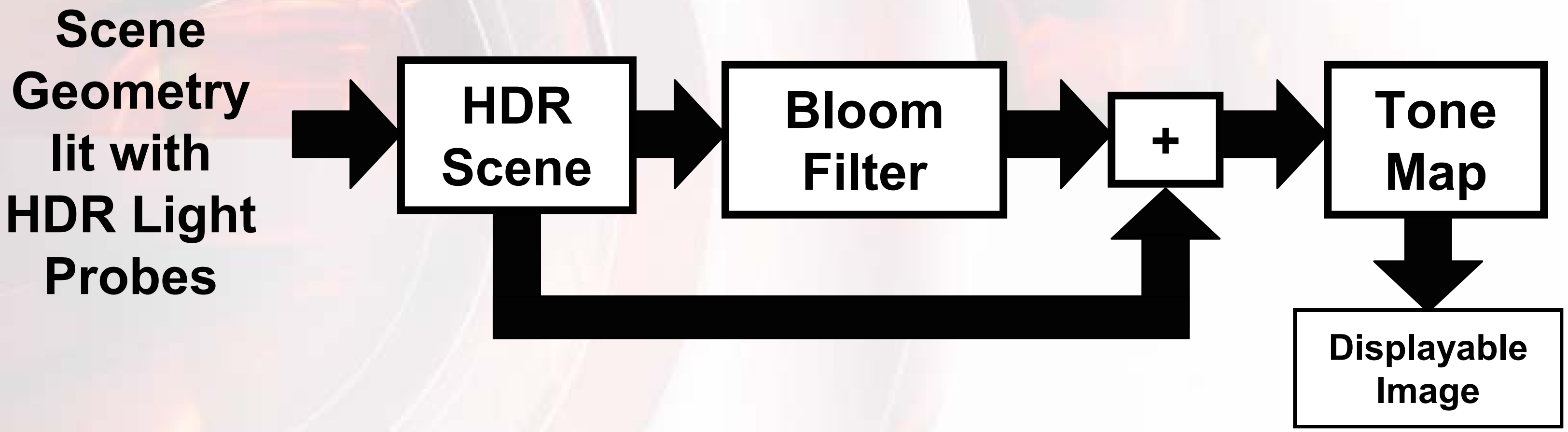


Image space outlining over toon shading

High Dynamic Range Rendering



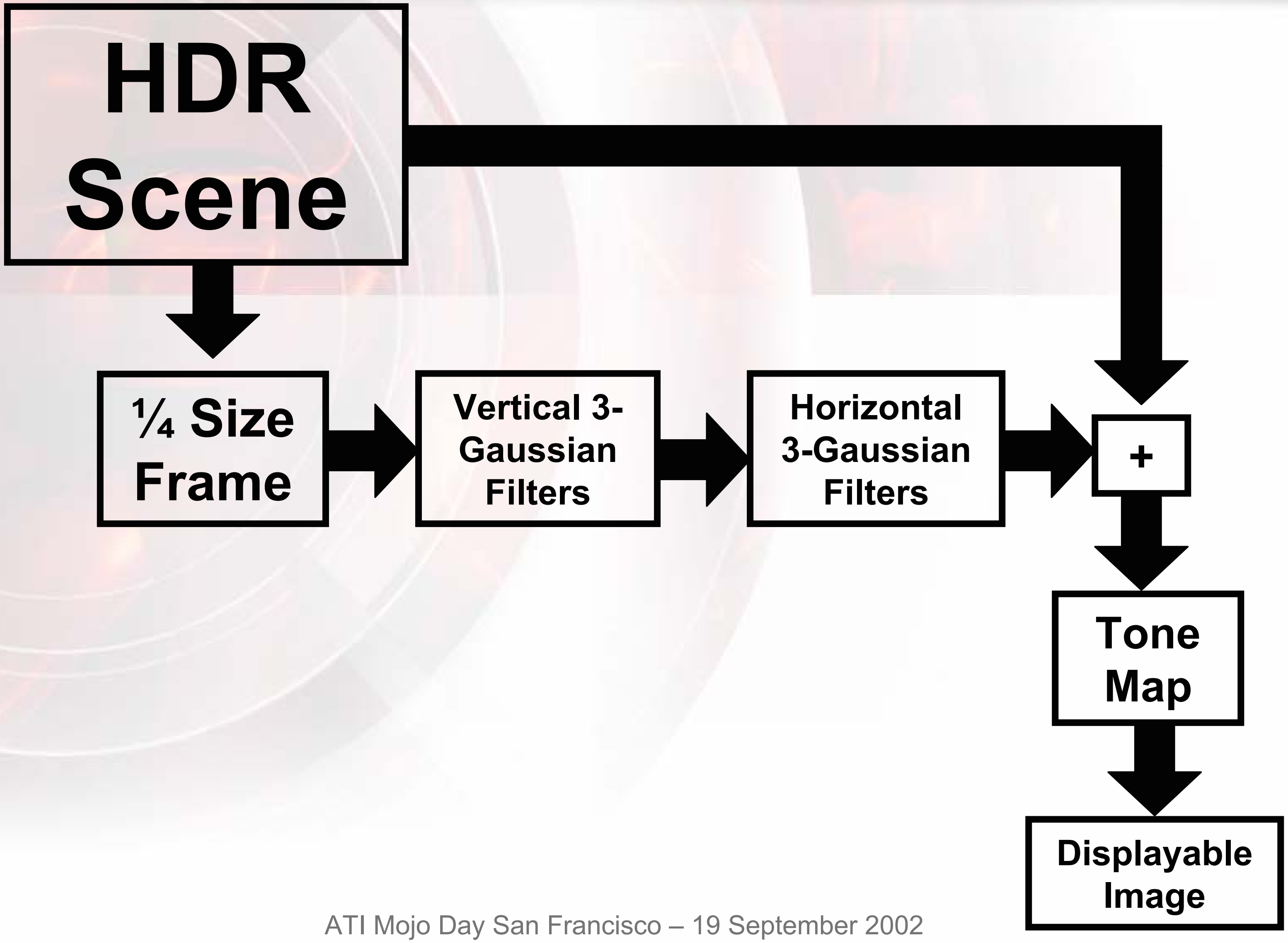
HDR Rendering Process



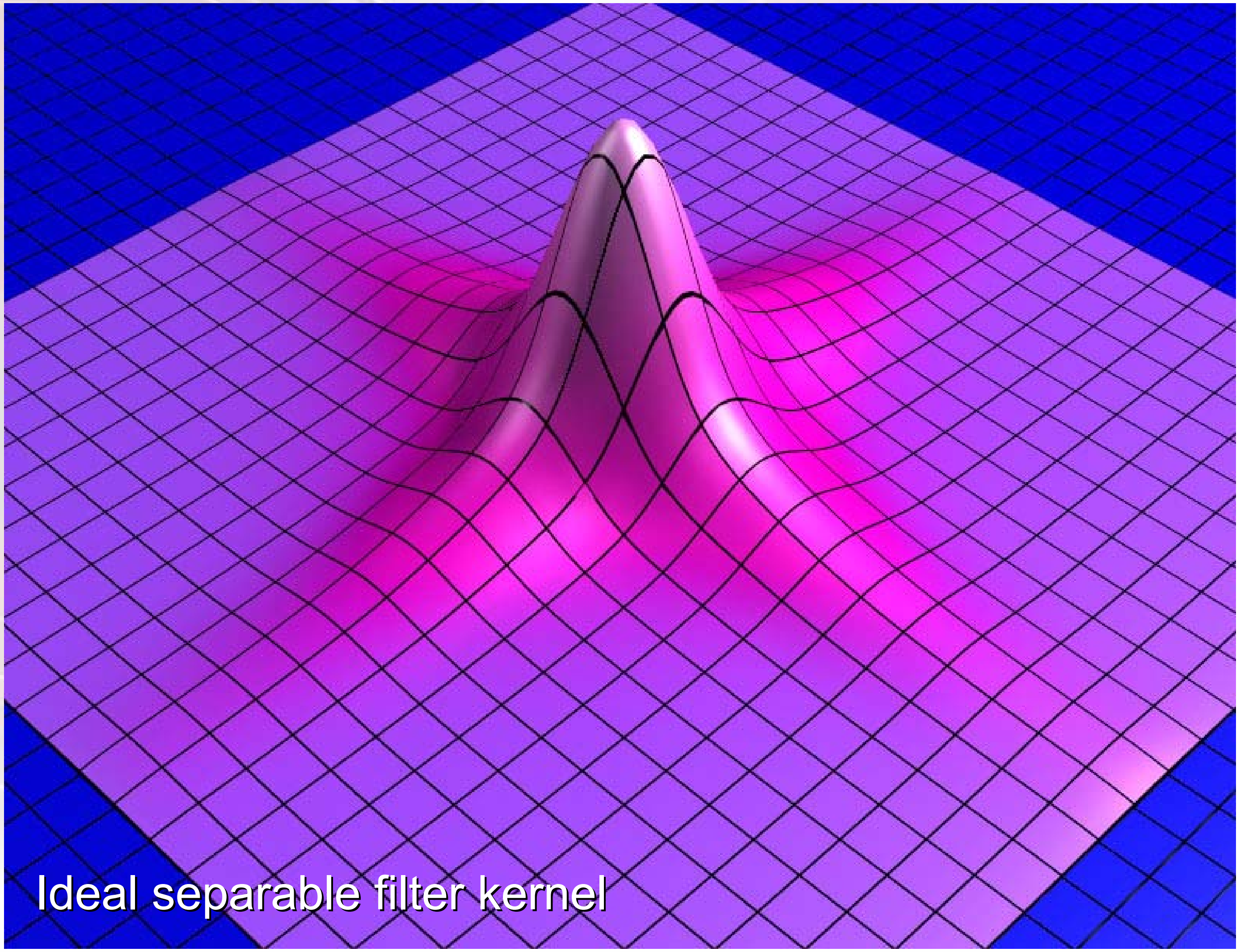
Rendering the Scene

- Render reflected scene into HDR planar reflection map for table top
- HDR light probe for distant environment
- HDR environment maps for local reflections from balls on pedestals
- Postprocess to get glows
- Tone map to displayable image

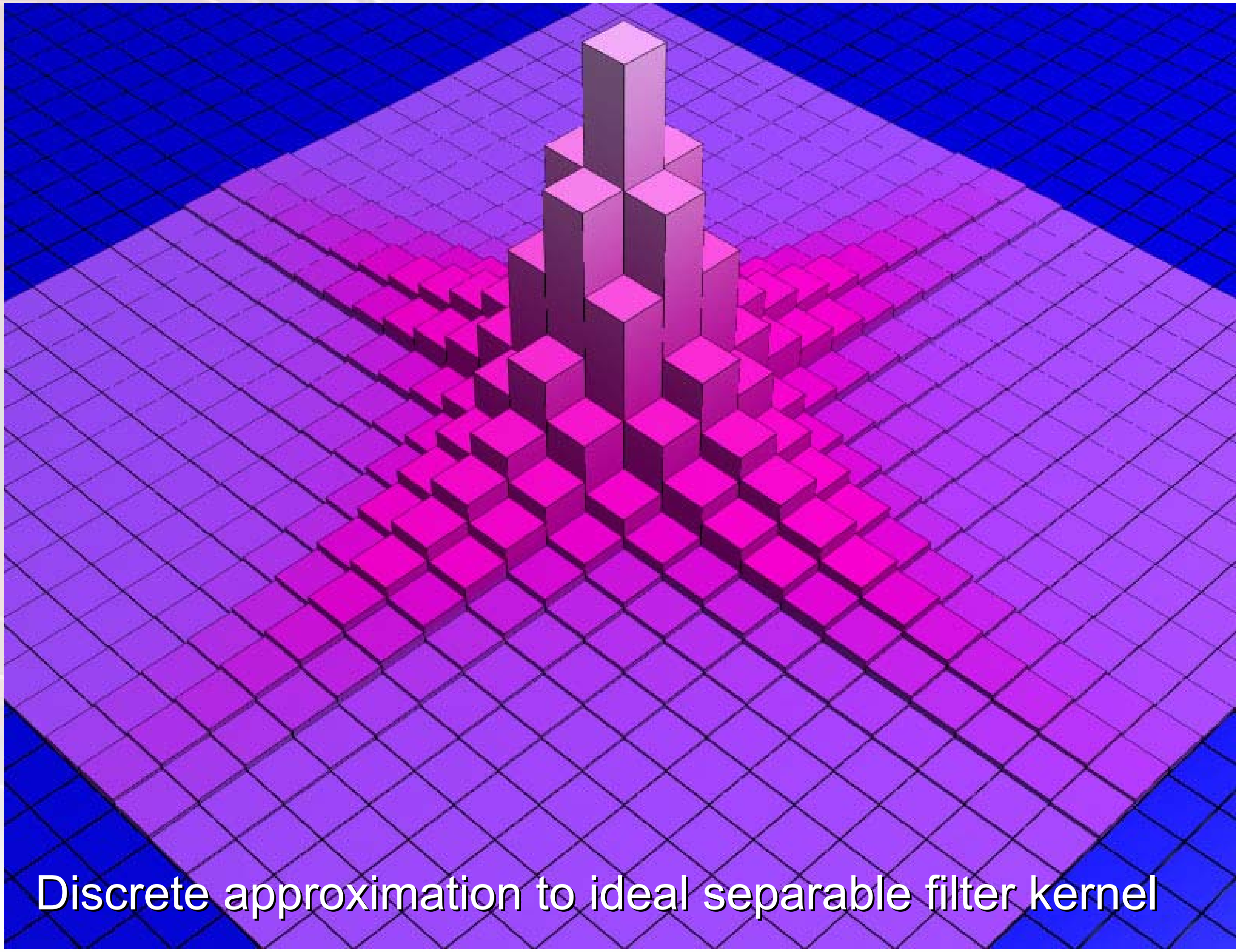
Frame Postprocessing



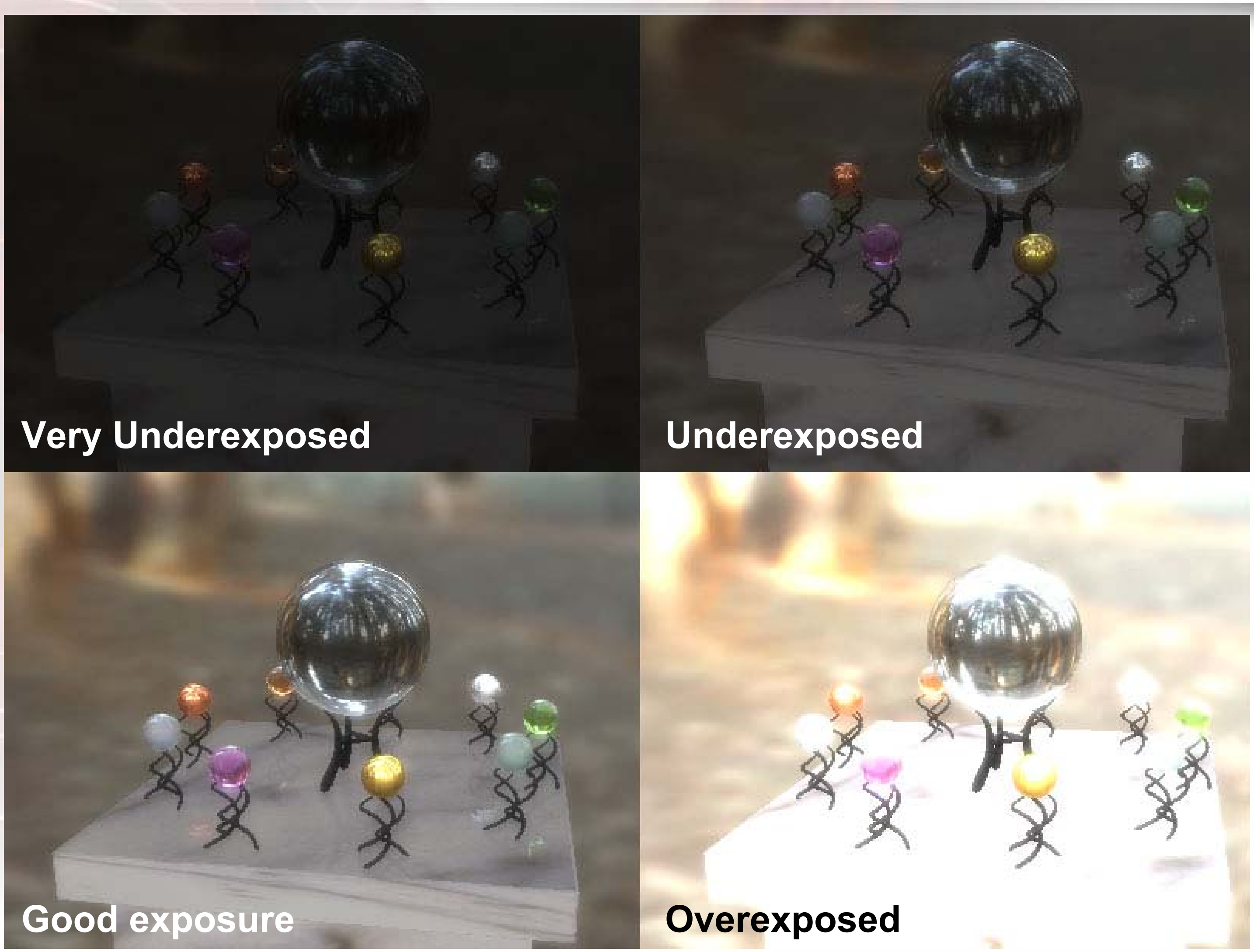
Approximate Sum of Three Gaussians



Approximate Sum of Three Gaussians


Discrete approximation to ideal separable filter kernel

Tone Mapping



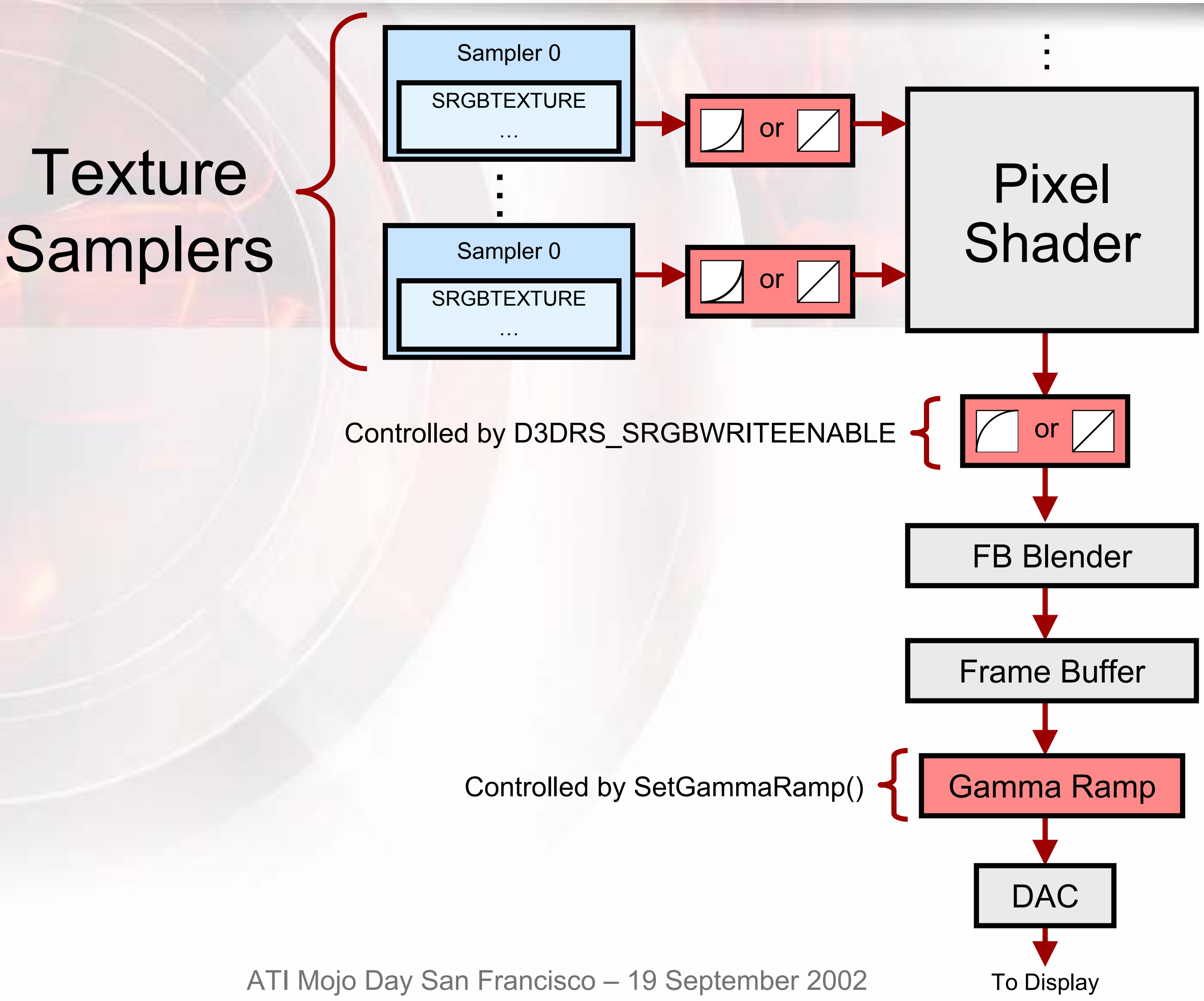
sRGB

- Standard encoding with $\gamma = 2.2$
- Dedicates bits to low-intensity values, where they're needed perceptually
- Colors can have γ curves applied at three points in the pipeline
  - Texture read ($\gamma = 2.2$ or linear)
  - Write to frame buffer ($\gamma = 2.2$ or linear)
  - Read from frame buffer into DAC (arbitrary table)

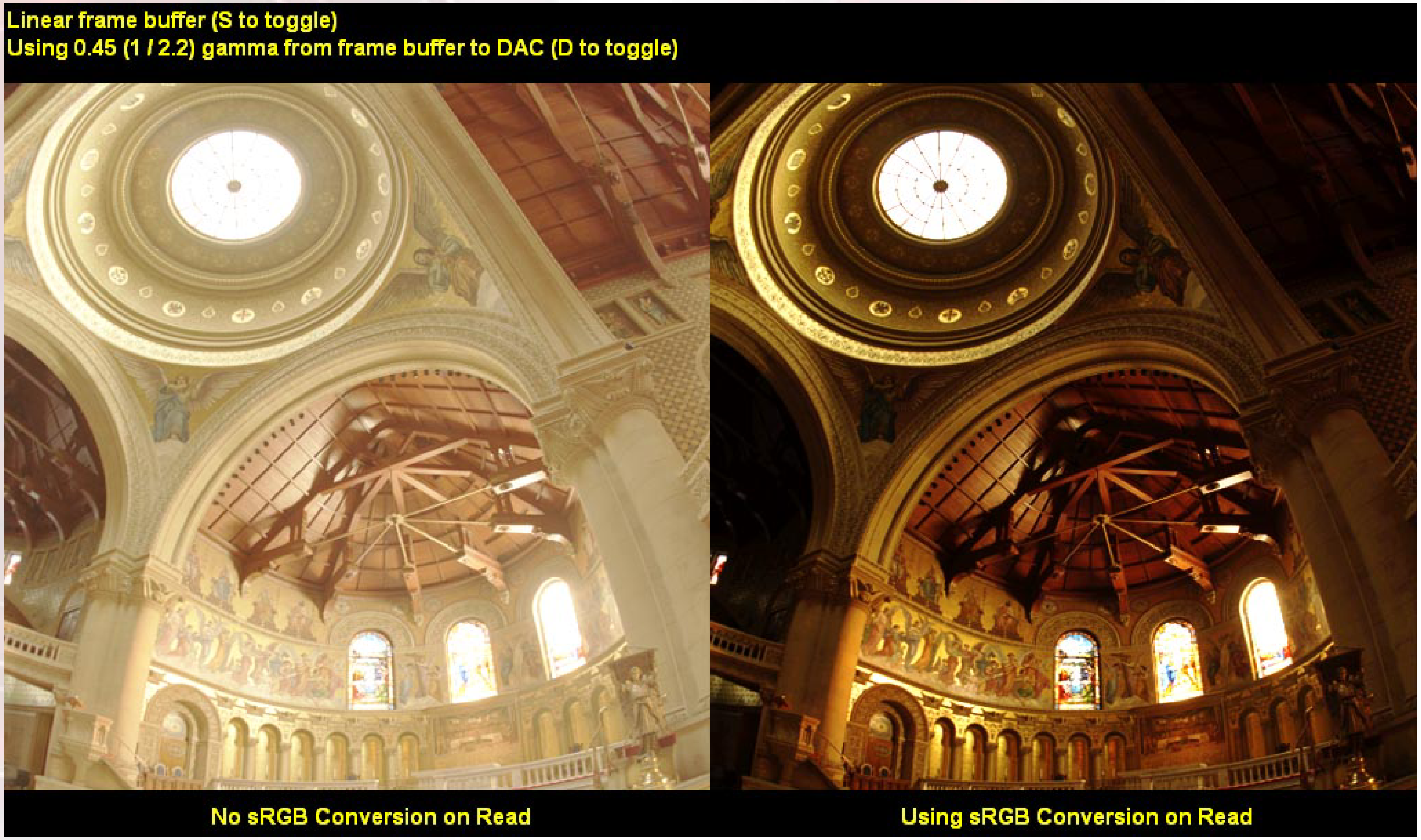
sRGB in DirectX® 9

- Application of sRGB conversion on texture read is associated with texture sampler state D3DSAMP_SRGBTEXTURE
  - If TRUE, art should be stored in 2.2γ and will be converted to linear on texture read
  - If FALSE, art is assumed to be linear already and no conversion is performed on texture read
- Application of sRGB conversion on frame buffer write is controlled with D3DRS_SRGBWRITEENABLE render state
  - If TRUE, frame buffer is considered to be 2.2γ and colors are converted from linear pixel pipeline
  - If FALSE, no conversion takes place

sRGB and Gamma in DirectX® 9



sRGB Sample Application


Linear frame buffer (S to toggle)
Using 0.45 (1 / 2.2) gamma from frame buffer to DAC (D to toggle)

No sRGB Conversion on Read

Using sRGB Conversion on Read

Conclusion

- DirectX® 8.1 1.4 Pixel Shader Review
  - Depth sprites
- DirectX® 9 2.0 Pixel Shaders
  - Procedural wood in assembly
  - High Level Shading Language
    - Review wood shaders using HLSL
    - Analyze resulting assembly code
  - Image-Space Operations
    - Edge detection for cartoon outlining
    - HDR
  - sRGB and Gamma