RADEON™ 9700 Shading

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

Introduction

In this supplement to Chapter 3 in the bound course notes, we will outline the shading capabilities of the new RADEON™ 9700 graphics processor and illustrate its power with a series of examples. After a brief overview of vertex shading on the RADEON™ 9700, we will spend the bulk of the time on the fragment (*aka* pixel) shader programming model. For much of this chapter, we will use Direct3D® syntax due to its brevity and readability, but all of this functionality (as shown in the section on homomorphic factorization of BRDFs) is available through multivendor and ATI-developed OpenGL extensions.

RADEON™ 9700 Vertex Shaders

The vertex shading model in the RADEON™ 9700 has advanced beyond the programming model found in previous graphics processors such as the RADEON™ 8500 by adding constant-based flow control capabilities. The motivation for this extension to the earlier programming models is to cut down on the number of shaders that must be created and managed by application developers. In previous vertex shader programming models such as the one found in DirectX® 8.1, developers generally had to create and use large numbers of shaders from fragments of shader code. This was necessary to handle the large numbers of permutations of drawing state (i.e. with/without environment mapping, with/without skinning, different numbers of and types of lights etc). The vertex shading model found in the RADEON™ 9700 enables developers to create a much smaller set of domain-specific über shaders which can contain constant-based loops, subroutines and branches to manage the permutation problem.

Vertex Shader Assembly Language

Like the vertex shading model in the previous generation of graphics processors, the RADEON™ 9700 can read from a bank of 4D float vectors as well as the input vertex components. A bank of read/write temps is available for intermediate computations and a bank of write-only output registers define the output from the vertex shader. Specific register counts for the RADEON™ 9700 are listed below.

Type	Name	Count
Float Constant	c[n]	256 read-only vectors
Temporary	r_n	12 read/write vectors
Input	v_n	16 read-only vectors

RADEON™ 9700 ALU Register Counts

In addition to the ALU-related inputs and temps above, the RADEON™ 9700 supports a set of constants and a counter to control the flow of execution as shown below.

Type	Name	Count
Integer Constant	I_n	16 read-only vectors
Address	A	1 read/write vector
Loop Counter	aL	1 scalar
Boolean Constant	B	16 read-only bits

RADEON™ 9700 Flow Control Register Counts

These constants are set through API calls and persist until set again. In this way, an application can program the control flow by setting a shader once and updating a few flow-control constants as appropriate. A vertex shader on the RADEON™ 9700 can be up to 256 instructions long, though more instructions than this may be executed due to looping, subroutines etc. The ALU instruction set supports the usual functionality such as ADD, DP3, DP4, EXP, FRAC, LOG, MAD, MAX, MIN, MOV, MUL, RCP, RSQ, SGE (set greater than or equal to) and SLT (set less than). The control flow instructions include CALL, LOOP, ENDLOOP, JUMP, JNZ, LABEL, REPEAT, ENDREPEAT and RETURN.

RADEON™ 9700 Pixel Shaders: First with Floating Point

The pixel shading model in the RADEON™ 9700 has also made a large step forward from previous generations, particularly due to the leap from fixed point internal computations to floating point. The new model goes beyond the models found in previous graphics processors such as the RADEON™ 8500 by adding support for floating point, an instruction set appropriate for per-pixel floating point operations and significantly greater program length.

Type	Name	Count
Vertex Color	v_n	2 read-only vectors
Temporary	r_n	12 read/write vectors
Constant	c_n	32 read-only vectors
Sampler	s_n	16 read-only
Texture Coordinate	t_n	8 read-only vectors

RADEON™ 9700 Pixel Shader Registers

These inputs and temporary registers may be operated on by a pixel shader with up to 64 ALU instructions and 32 texture instructions. The ALU instruction set has been extended to include instructions appropriate to floating point data such as reciprocal and reciprocal square root. The ALU instruction set is made up of instructions such as ADD, MOV, MUL, MAD, DP3, DP4, FRAC, RCP, RSQ, EXP, LOG and CMP.

The texture instructions of the RADEON™ 9700 are used to sample data from texture maps and to conditionally kill pixels altogether. The texture loading instructions include the plain TEXLD instruction, the projective TEXLDP instruction and the TEXLDBIAS instruction. The TEXLDBIAS instruction can be used to apply a per-pixel LOD bias to any texture map access. This can be used for a wide variety of applications including simulation of rough specular surfaces and general blurriness. We will show examples of this in the motion blur and car paint shaders later in these notes. The TEXKILL instruction can be used to conditionally kill pixels based on the signs of register components.

Samplers and Texture Coordinates

In the pixel shader programming model of the RADEON™ 9700, samplers and texture coordinates are completely decoupled. Texture coordinates are iterated across polygons and may be used to sample data through a sampler. At any one time, a given sampler is associated with a specific texture map in memory as well as a set of filtering state and texture coordinate clamping state. The RADEON™ 9700 can iterate up to 8 4D texture coordinates and has 16 samplers. Naturally, it is possible to sample from a single sampler multiple times in a given shader using different texture coordinates. This is common when performing image processing operations as we will show later in the High Dynamic Range bloom and image space outlining examples.

Pixel Shader Outputs

The pixel shading unit of the RADEON™ 9700 can output up to four colors to different render targets. The ability to output to multiple render targets simultaneously allows multiple intermediate values to be saved out between rendering passes and allows for implementation of G-buffer techniques [Saito and Takahashi 1990]. An image-space outlining technique, as described by Saito and Takahashi, using multiple simultaneous pixel shader outputs is shown later in these notes and in the SIGGRAPH 2002 sketch *Real-Time Image-Space Outlining for Non-Photorealistic Rendering*.

Now that we've given a brief introduction to the shader programming models, we'll discuss the common example shaders used by all presenters in this course: Bumped Cubic Environment Mapping, McCool's Homomorphic Factorization of BRDFs and Procedural Wood. Subsequently, we'll present additional shaders including high-dynamic range rendering, motion blur, image space outlining and two-tone car paint in order to illustrate usage of many of the new aspects of the RADEON™ 9700 pixel shader programming model.

The Common Example Shaders

All of the presenters in this course have been asked to implement a set of common example shaders to aid in understanding the differences between the programming models. In this section, we

will discuss our implementations of the common shaders using DirectX® 8.1's ps.1.4 programming model, the `ATI_fragment_shader` model and DirectX® 9's ps.2.0 programming model.

Bumped Cubic Environment Mapping

Cubic environment mapping has become a common environment mapping technique for reflective objects. Specifying a normal per-pixel in order to apply local detail or an *Appearance Preserving Simplification* technique [Cohen et al 1998] is becoming increasingly popular, even in conjunction with cubic environment mapping. In fact, the two-tone layered car paint example shown later in these notes uses a high-precision normal map derived from appearance preserving simplification. In this first common example shader, however, we will use a simple model and a basic ps.1.4 shader which does cubic environment mapping based on per-pixel normals from a normal map. A reflection vector is calculated per-pixel and is used to access a specular cubic environment map. This shader, introduced last year in the ATI Treasure Chest Demo, also does diffuse cube mapping and combines this with the specular environment map based on a per-pixel Fresnel term in one rendering pass [Brennan 02]. As it turns out, these additional effects basically come for free when implementing bumped cubic environment mapping, particularly in the RISC pixel shading models found in the RADEON™ 8000 and 9000 series GPUs.

The fundamental operation performed by this shader is transformation of a normal from tangent space into the space of the cube map to be sampled (world space, say). This operation can be performed in ps.1.4 with a series of 3 dot products with write masks to perform the 3D rotation which gives us the correct normal N. Subsequently, the interpolated eye vector E is reflected through this vector using the typical reflection operation:

$$R = 2N(N \cdot E) + E$$

or if the normal is not of unit length:

$$R = 2N(N \cdot E) + E(N \cdot N)$$

The reflection vector, R, is used to do a dependent read from a cubic environment map. Additionally, due to the RISC nature of ps.1.4, we can set aside the intermediate result, N, and use it to do a dependent read from a diffuse cube map. It is also possible to sample several other maps including

a base texture map and a colored gloss map. These terms plus a per-pixel Fresnel term can easily be composited together in one pass as shown in [Brennan 02].

A simple pixel shader which does only the diffuse and specular dependent cube map lookups is shown below.

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

 texld   r0, t0                  ; Look up normal map
 texcrd  r1.xyz, t4              ; Eye vector
 texcrd  r4.xyz, t1              ; 1st row of environment matrix
 texcrd  r2.xyz, t2              ; 2st row of environment matrix
 texcrd  r3.xyz, t3              ; 3rd row of environment matrix

 dp3     r4.x, r4, r0_bx2        ; N.x = 1st row of matrix multiply
 dp3     r4.y, r2, r0_bx2        ; N.y = 2nd row of matrix multiply
 dp3     r4.z, r3, r0_bx2        ; N.z = 3rd row of matrix multiply
 dp3_x2  r3.xyz, r4, r1          ; 2(N.Eye)
 mul     r3.xyz, r4, r3          ; 2N(N.Eye)
 dp3     r2.xyz, r4, r4          ; N.N
 mad     r2.xyz, -r1, r2, r3     ; 2N(N.Eye) - Eye(N.N)

 phase

 texld   r2, r2                  ; Sample cubic reflection map
 texld   r3, t0                  ; Sample base map with gloss in alpha
 texld   r4, r4                  ; Sample cubic diffuse map

 mul     r1.rgb, r3.a, r2        ; Specular = Gloss * Reflection
 mad     r0.rgb, r3, r4, r1      ; Base*Diffuse + Specular
+mov     r0.a, c0.a              ; Put 1.0 in alpha
```

Homomorphic Factorization BRDF

The second of the common example shaders is based upon techniques outlined in McCool et al's *Homomorphic Factorization of BRDFs for High-Performance Rendering* from SIGGRAPH 2001. On modern GPUs like the RADEON™ 8500 or RADEON™ 9700, the actual pixel shader used in this technique is very simple to execute in one pass. For most of the materials rendered by McCool's sample application, the equation to be evaluated is one of the following:

1. (((diffuse * tex0) * scale1) * tex1 * scale2) * tex2
2. (((diffuse * tex0) * scale1) * tex1 * scale2) * tex2 + tex3
3. (((diffuse * tex0) * scale1) * tex1 * scale2) * tex2) * tex3

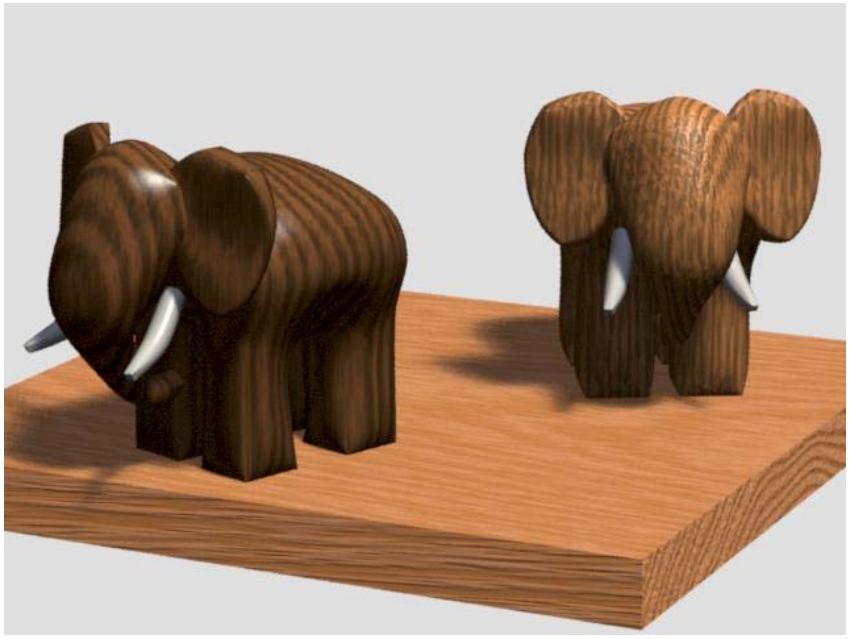
Using OpenGL `ATI_fragment_shader` notation, these equations can easily be evaluated:

```
glBeginFragmentShaderATI();

glSampleMapATI (GL_REG_0_ATI, GL_TEXTURE0_ARB, GL_SWIZZLE_STR_ATI); // Sample maps
glSampleMapATI (GL_REG_1_ATI, GL_TEXTURE1_ARB, GL_SWIZZLE_STR_ATI);
glSampleMapATI (GL_REG_2_ATI, GL_TEXTURE2_ARB, GL_SWIZZLE_STR_ATI);

if (param == PARAM_OHI_H)   // only sample the specular map if necessary
{
   if (sg_tex3Type == GL_TEXTURE_CUBE_MAP_ARB) {
      glSampleMapATI (GL_REG_3_ATI, GL_TEXTURE3_ARB, GL_SWIZZLE_STR_ATI); }
   else {
      glSampleMapATI (GL_REG_3_ATI, GL_TEXTURE3_ARB, GL_SWIZZLE_STQ_ATI); }
}

// r0 = diffuse * tex0 * scale1
glColorFragmentOp2ATI (GL_MUL_ATI, GL_REG_0_ATI,         GL_NONE, scale1,
                                   GL_PRIMARY_COLOR_EXT, GL_NONE, GL_NONE,
                                   GL_REG_0_ATI,         GL_NONE, GL_NONE);

// r0 = (diffuse * tex0 * scale1) * tex1 * scale2
glColorFragmentOp2ATI (GL_MUL_ATI, GL_REG_0_ATI, GL_NONE, scale2,
                                   GL_REG_1_ATI, GL_NONE, GL_NONE,
                                   GL_REG_0_ATI, GL_NONE, GL_NONE);

if (param == PARAM_OHI_H)  // do a MAD if specular map is used
{
   // r0 = ((diffuse * tex0 * scale1) * tex1 * scale2) * tex2 + tex3
   glColorFragmentOp3ATI (GL_MAD_ATI, GL_REG_0_ATI, GL_NONE, GL_NONE,
                                      GL_REG_0_ATI, GL_NONE, GL_NONE,
                                      GL_REG_2_ATI, GL_NONE, GL_NONE,
                                      GL_REG_3_ATI, GL_NONE, GL_NONE);
} else{
   // r0 = ((diffuse * tex0 * scale1) * tex1 * scale2) * tex2
   glColorFragmentOp2ATI (GL_MUL_ATI, GL_REG_0_ATI, GL_NONE, GL_NONE,
                                      GL_REG_0_ATI, GL_NONE, GL_NONE,
                                      GL_REG_2_ATI, GL_NONE, GL_NONE);}
glEndFragmentShaderATI();
```

Parameterized Volumetric Wood

Procedural generation of texture patterns is a popular technique for a variety of reasons including reduced storage space and customizability through intuitive parameters. A seminal example of this is the venerable wood shader from the *RenderMan Companion*.

In this section, we present a shader which procedurally generates wood in an attempt to mimic the real wood samples running down the left side of the page. The shader is based on the parametrized wood shader in Chapter 12.4 of *Advanced RenderMan* by Apodaca and Gritz.



Non-real-time Gumbos from *Advanced RenderMan*

The real-time version of this shader uses a single 128×128×128 luminance-only 3D noise texture and one 1D function texture to evaluate a pulse-train composed of *smoothstep()* functions, for a total of around 2 MB of texture memory. Of course, the power of this approach is that it's parametrized so that a variety of woods can be generated by tweaking a small number of intuitive input parameters to a single shader. Also, the sample application used here has been developed so that an artist can interactively position the geometry

anyplace in shader space (i.e. at any position in the virtual log) in order to obtain a particular look. The disadvantage of this wood implementation is the relatively high computational cost of executing the shader. The input parameters to the shader are two wood colors ($C_{lightwood}$ and $C_{darkwood}$), ring frequency (*freq*), noise amplitude (*amp*), trunk wobble frequency, trunk wobble amplitude, specular exponent scale and specular exponent bias.

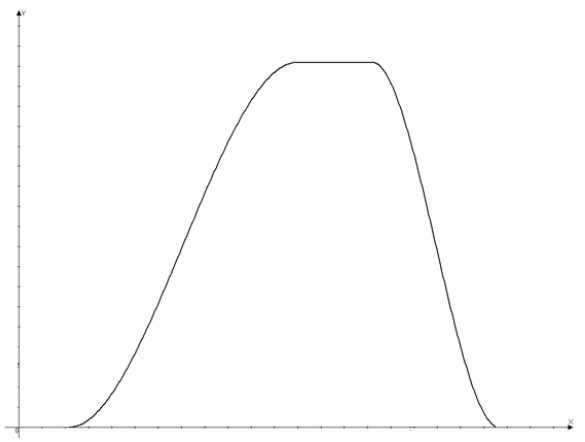
Computation occurs in *shader space* and a pixel's position in this space is denoted as P_{shade}. This allows us to position the volumetric wood relative to the object by transforming P_{shade} in the vertex shader. This is a similar technique to the per-pixel distance attenuation used in the RADEON™ 8500 Treasure Chest Demo which procedurally computes distance attenuation from the *light space position* interpolated across polygons [Vlachos 02]. In fact, for sampling noise into this wood shader, we will interpolate five 3D positions: P_{shade} and four simple shifts and scales of P_{shade}. This allows us to sample the same 3D scalar noise texture multiple times to get vector noise whose channels are reasonably uncorrelated.

Basic Concentric Rings

We will begin with a simple shader which defines cylindrical bands around the shader space *z* axis. The intermediate value *r* is computed as follows:

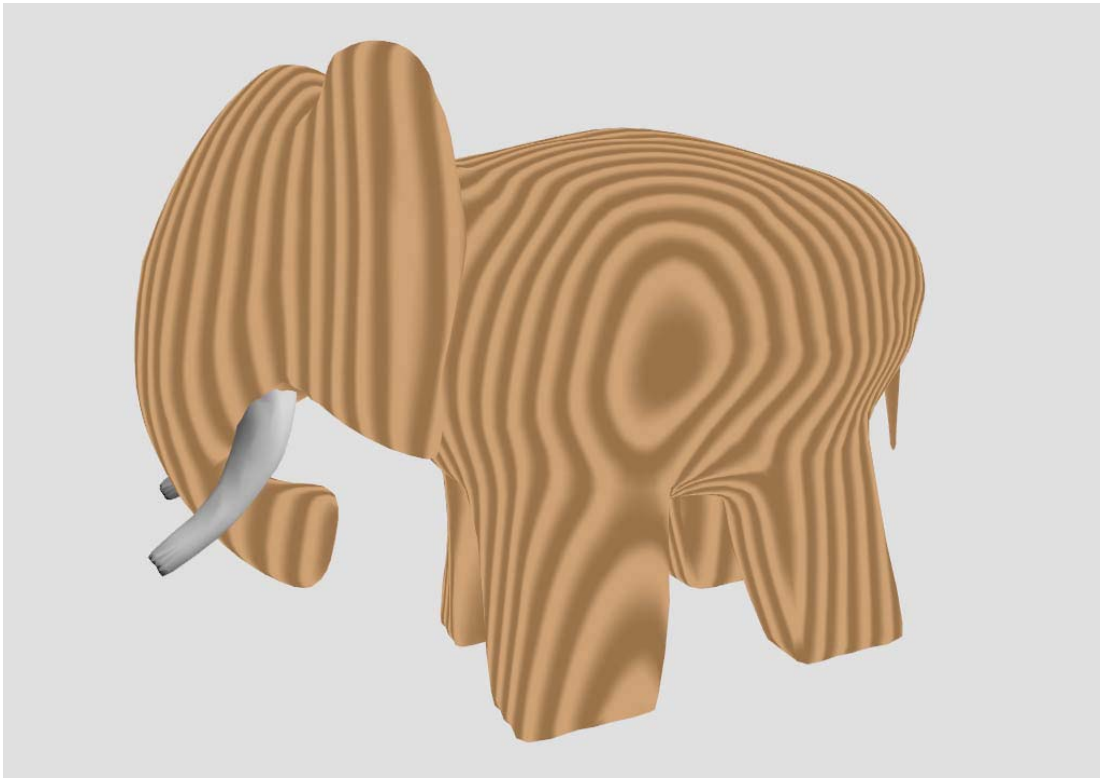
$$r = \sqrt{P_{shade_x}{}^2 + P_{shade_y}{}^2} * freq$$

This gives us the distance from the shader space *z* axis which is scaled by *freq*. We will use *r* as a texture coordinate into a 1D luminance texture which defines the shape of each ring as it radiates out from the shader space *z* axis. This 1D luminance texture is a smooth pulse train composed of *smoothstep*() functions which has been tuned to mimic the way that colors mix in wood. A single smooth pulse is shown below:



Smooth Pulse Used to Blend Wood Colors

Using the luminance that we get back from the smooth pulse function texture shown above as a blend factor between $C_{lightwood}$ and $C_{darkwood}$ gives us the following effect on the familiar Gumbo model.



Gumbo with procedural concentric rings

So far, our ps.2.0 shader code looks like the following

```
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

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

dcl_2d     s1              // 1D smooth step function

dp2add r0, t0, t0, 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
```

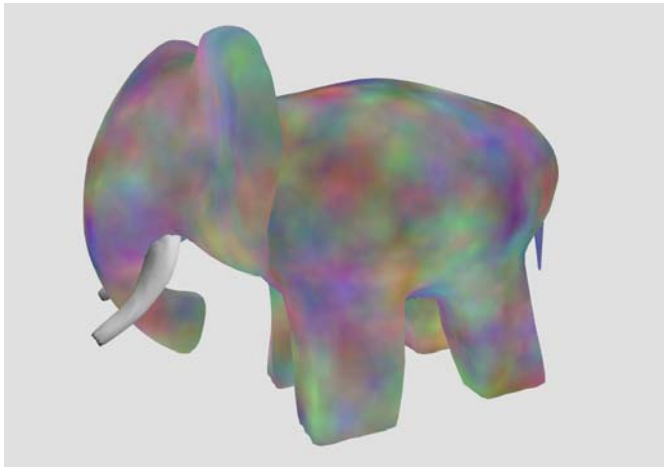
The final vertex shader, which generates the inputs to all of the wood pixel shaders is shown below.

```
dcl_position v0
dcl_normal   v3

def c40, 0.0f, 0.0f, 0.0f, 0.0f

m4x4 oPos, v0, c[0]           // Transform position to clip space

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

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

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

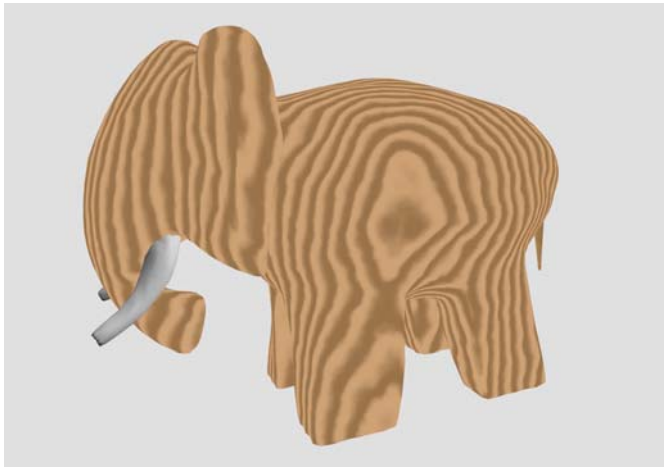
Incorporating Noise

The ring pattern that we've generated so far is obviously too regular for believable wood and needs to be made more noisy. Since we do not yet have native noise hardware, we will use a texture to look up the noise. In this example, we will sample a 3D texture which contains several octaves of

tilable signed scalar fractional Brownian motion noise. We will sample this map 3 times using shifted, negated and slightly scaled values of P_{shade} to construct reasonably uncorrelated vector fractional Brownian motion noise. This signed vector noise is added to P_{shade} prior to the computation of r above.



Vector fractional Brownian motion noise



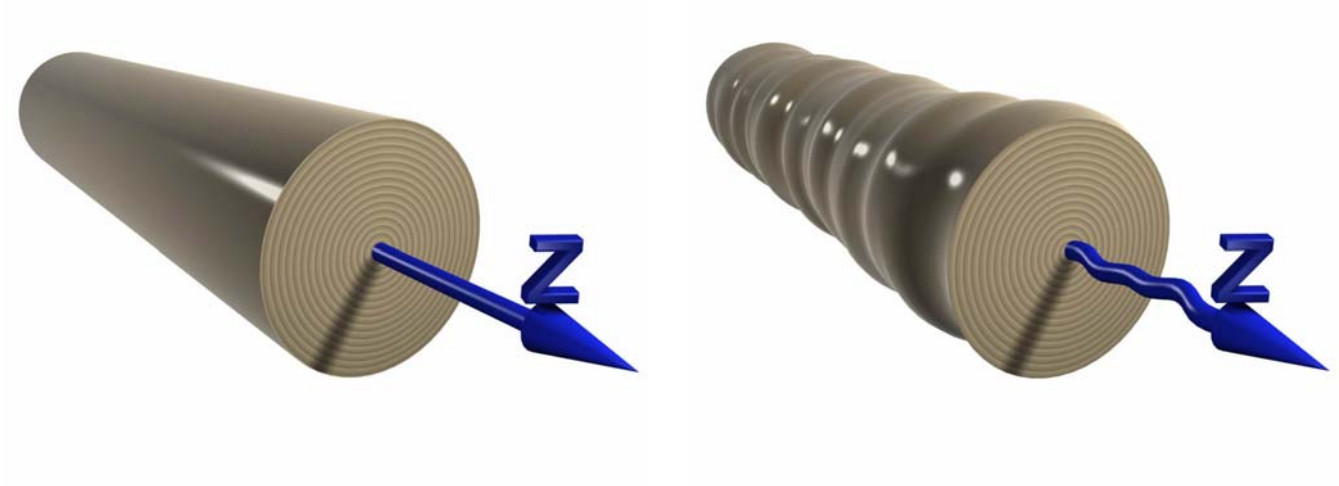
Gumbo with "noised up" rings

Incorporating this additional noise, our ps.2.0 shader code looks like the following

```
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
// 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_volume s0               // Luminance-only Volume noise
dcl_2d     s1               // 1D smooth step function

texld r3,  t0, s0           // Sample dX from scalar noise at Pshade
texld r4,  t1, s0           // Sample dY from scalar noise at perturbed Pshade
texld r5,  t2, s0           // Sample dZ from scalar noise at perturbed Pshade

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 Pshade to warp the 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
```

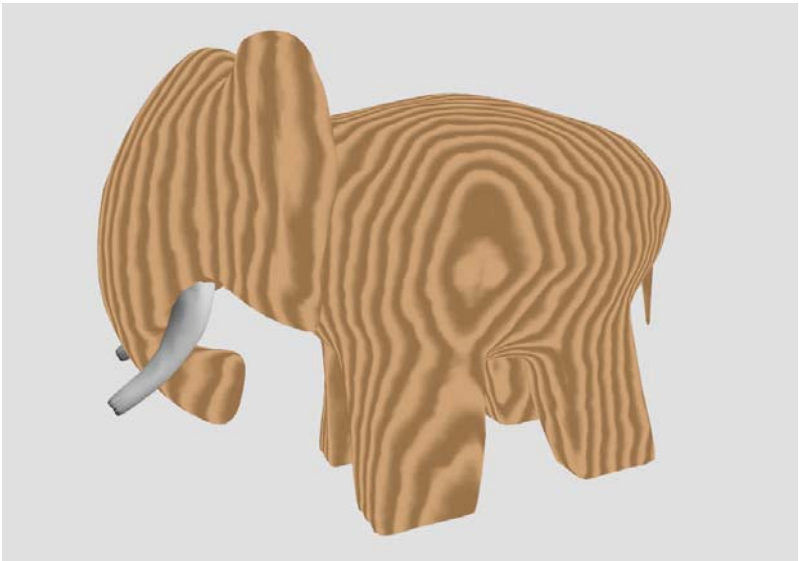
Trunk Wobble

This level of noise is definitely an improvement, but the rings are still all centered around the *z* axis. We can add noise to $P_{shade}.x$ and $P_{shade}.y$ as a function of *z* to cause the trunk to wobble as it varies in *z*. This effect alone essentially causes a wobbly tree trunk as shown below.



Tree trunk without wobble and with wobble in x and *y* as a function of z

To incorporate noisy trunk wobble into this shader, we sample the volume noise map at two different locations (which are functions of $P_{shade}.z$) along the $y = z = 0$ column of texels in our volume noise map. These two scalar values are added to x and y in the pixel shader, causing the rings to wobble as shown in the diagram above. The effect of this wobbling on our Gumbo dataset is shown below.



Gumbo with "noised up" rings and wobbly tree trunk

Incorporating the previous noise and the perturbations to make the trunk wobble, our ps.2.0 shader code looks like the following.

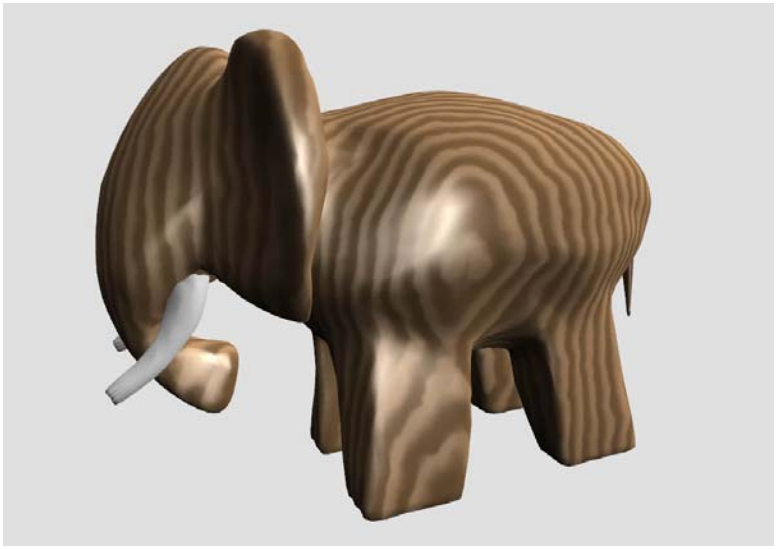
```
ps.2.0

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
// 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, specular exponent in y, ...)

texld r3,  t0, s0          // Sample dX from scalar volume noise texture at Pshade
texld r4,  t1, s0          // Sample dY from scalar volume noise texture at perturbed Pshade
texld r5,  t2, s0          // Sample dZ from scalar volume noise texture at perturbed Pshade

texld r6, t3, s0           // Sample trunkWobble.x from scalar volume noise at {Pshade.z, 0, 0}
texld r7, t4, s0           // Sample trunkWobble.y from scalar volume noise at {Pshade.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 Pshade 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
```

At this point, we have procedurally generated the albedo of the wood in a 2.0 pixel shader. Next we will Phong shade the surface using pixel shader instructions to compute *L* from the position of a single light source and perform several renormalization operations along the way—operations that were previously impossible to carry out in the pixel pipeline due to limited range and precision. With the floating point pixel pipeline of the RADEON™ 9700, true Phong shading is now possible in real-time.

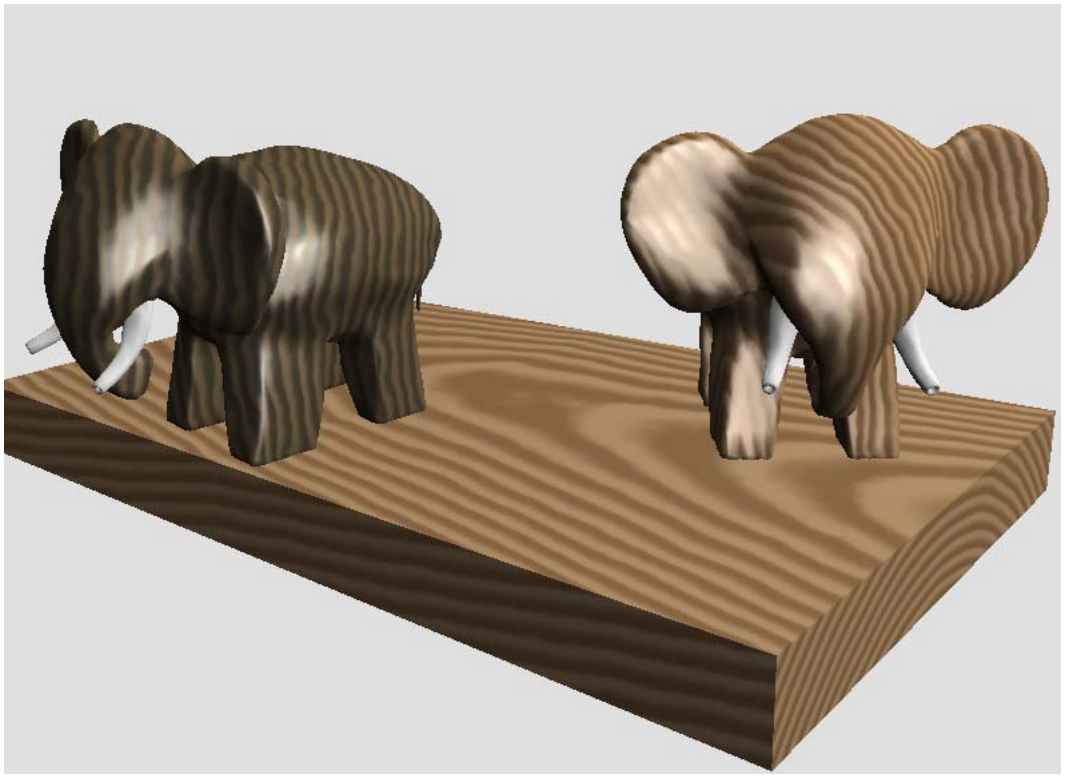
With the procedural albedo generation and Phong shading (where specular exponent is also a function of the smoothstep function, giving different specular exponents to the different wood rings), our final pixel shader looks like the following:

```
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
// c4: xyz == L_eye,  w == trunkWobbleAmplitude
// c5: {X, X, expScale, expBias}

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 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 Pshade 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
```

The final shader gives us the following result.



Final Shader on one Gumbo



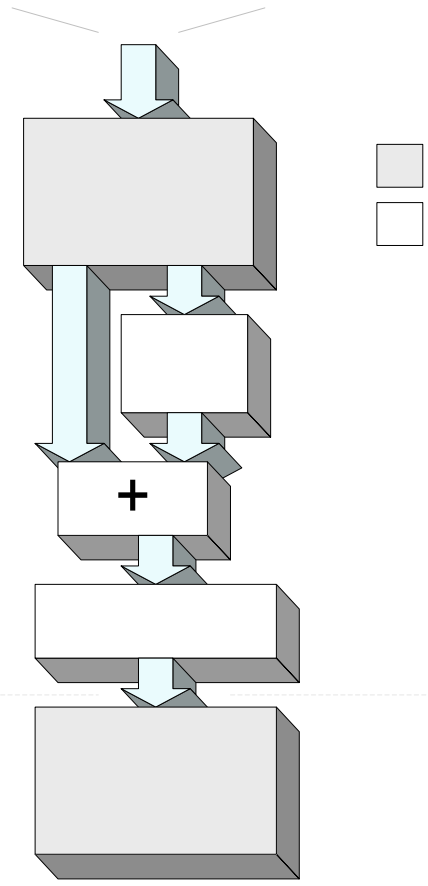
Final Shader on test scene

Additional Shaders

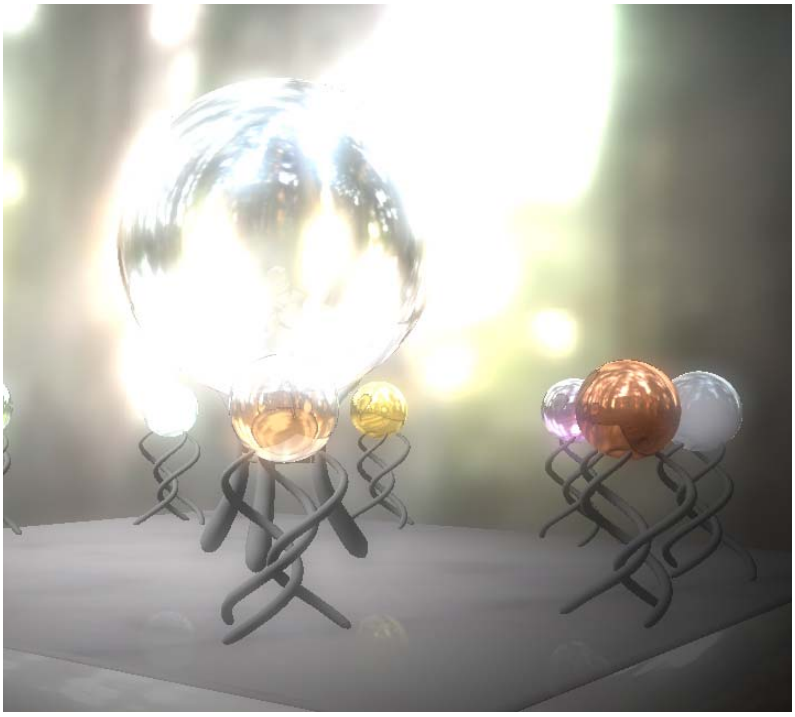
Now that we've covered the compulsory shaders for this course, I'll introduce some additional shaders to illustrate a variety of new applications enabled by the RADEON™ 9700. This diverse collection of shaders includes several steps of the High Dynamic Range Rendering process, motion blur, local reflections, a two-tone car paint model and image-space outlining.

High Dynamic Range Rendering

With the extended range and precision of the RADEON™ 9700, it is possible to do a variety of rendering operations in High Dynamic Range (HDR) space. In fact, we have implemented Paul Debevec's *Rendering With Natural Light* animation in real-time to illustrate the ability to do real-time image-based lighting in HDR space. The block diagram to the right shows the high-level approach to rendering that is used in the real-time and non-real-time renderings of the *Rendering with Natural Light* animation. The bulk of these operations need to be performed in High Dynamic Range Space in order to truly represent the wide range of radiances present in a real-world scene [Debevec et al 1998]. The objects in the scene, illuminated with HDR images, are first rendered into an HDR image of the scene. A bloom filter is applied to the image to give the image a soft look, simulating light scattering in the eye or the optics viewing the virtual scene. The blooms are added back to the HDR rendering of the scene. This sum is then vignetted and tone mapped down to a displayable range.



HDR Implementation on RADEON™ 9700

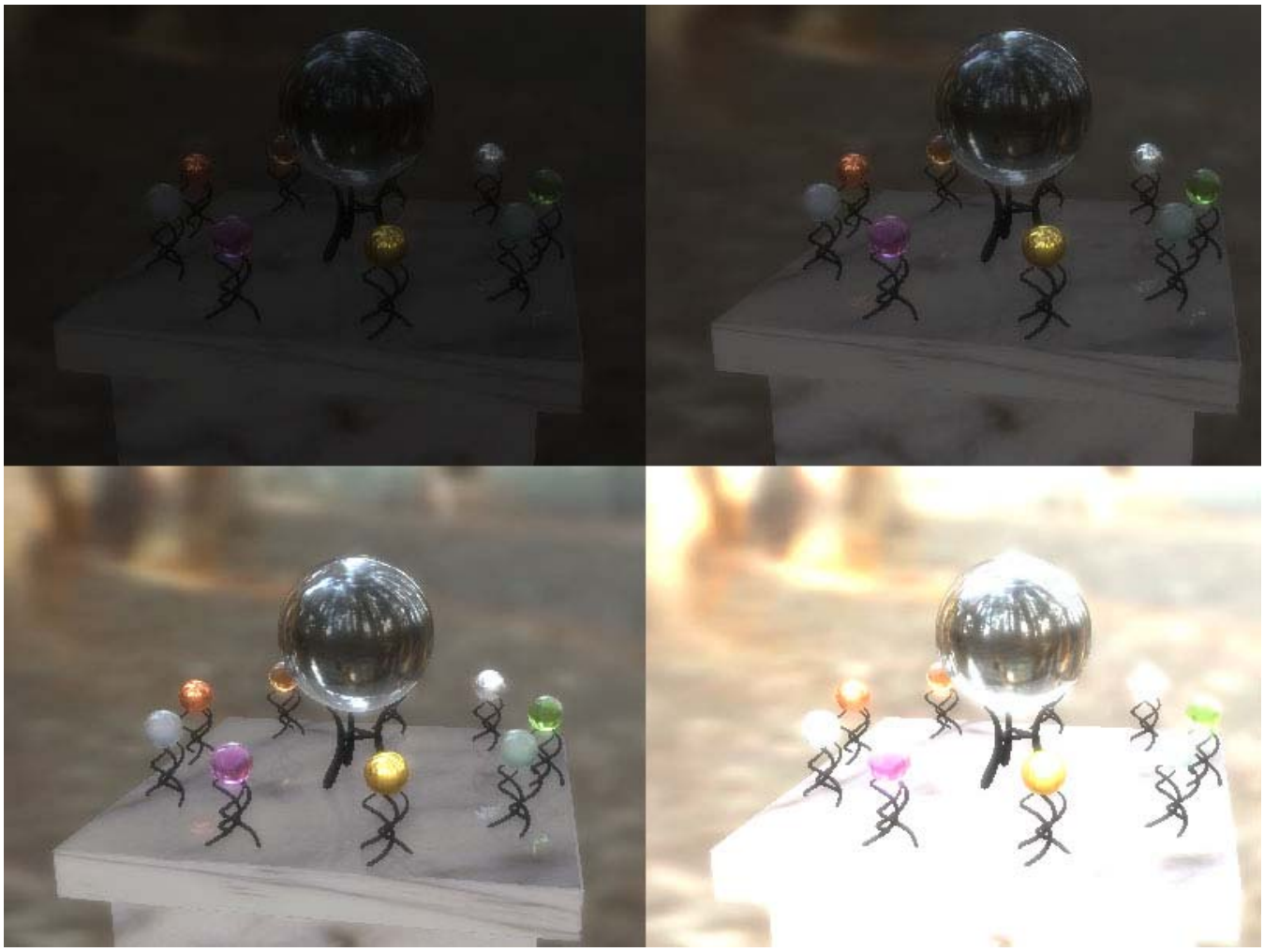


The image to the right is a typical frame from the real-time *Rendering with Natural Light* demo. The first step in rendering a frame of this animation is generation of the planar reflection map for the table top. The HDR planar reflection map is rendered to a 16-16-16-16 texture. Once we have the reflection map, the objects are rendered into the scene, using HDR diffuse, specular and rough specular illumination maps as appropriate for each material. The table top is textured with the planar reflection map generated earlier in the frame. The local reflections and refractions on the spheres in the scene are handled using a special shader which will be discussed later in these notes.

Frame Postprocessing

After the 16-16-16-16 HDR image of the scene is rendered, a set of three Gaussian filters covering a region of 50×50 pixels are applied to the image. This is achieved in three steps since the Gaussian is separable. The first step is downsizing the image to ¼ of its original size (½ in x and ½ in y). Three Gaussians with $\sigma=2$, $\sigma=6$ and $\sigma=14$ are applied in two more steps. The first of these performs the first pass of the separable blurs on the ¼ size image. It computes an appropriately weighted sum of a column of 25 pixels centered on the current pixel and saves the results to a temporary buffer. This temporary buffer is then processed to apply the filter in the horizontal direction. In this same pass, the Gaussians are added back to the original image, tone mapped and vignetted to create the final image which is rendered to an 8-8-8-8 surface that can be read by the DAC.

The tone mapping process allows us to simulate different exposure levels by simply applying a different curve to the rendered HDR image. Four examples are shown below.



High Dynamic Range scene at four different exposure levels

All four of these images were generated by applying different tone-mapping curves to the same real-time rendered HDR image.

Local Reflections and Refractions

In the real-time *Rendering with Natural Light* demo, it is necessary to compute local and non-local reflections when rendering the balls in the scene. This is done with two key techniques.

First, a separate cube map is stored for each of the 10 spheres in the scene. These cubemaps are rendered once during a preprocessing step from the center of the corresponding sphere and contain a mask channel which masks the local (other objects in the scene) and non-local (distant environment) parts of the scene. This mask channel is used to isolate local and non-local contributions to the illumination of each sphere as it is rendered into the HDR scene.

Second, some creative reflection and refraction calculations are done when computing the cube map coordinates for the local environment map. For the bottom quarter of the sphere, surface normals are used to index the cube map while pure reflection vectors are used on the top quarter. For the middle section of the sphere, a blend between the normals and reflection vectors is used. This is necessary to ensure that the spheres appear to be in contact with the pedestals upon which they are resting in the scene. These non-standard vectors are used to access the local environment map while "standard"

reflection vectors are used to access the distant HDR light probe. The mask channel of the local reflection cube map is used to select between the two cube maps. The pixel shader for the opaque reflective (non-refractive) metal balls is shown below.

```
ps.2.0
dcl_cube s0
dcl_cube s1
dcl t0                              // normal
dcl t1                              // view vector

dp3     r2.a, t0, t0
rsq     r2.a, r2.a
mul     r2.rgb, t0, r2.a            // normalized normal

dp3     r3.a, t1, t1
rsq     r3.a, r3.a
mul     r3.rgb, t1, r3.a            // normalized view vector

dp3     r4.a, r2, r3                // N.V
mul     r4.rgb, r4.a, r2            // N(N.V)
mad     r4.rgb, r4, c0.w, -r3       // R = 2N(N.V)-V

mad_sat r2.a, -r2.y, c4.r, c4.g // curve on sphere (y up)
lrp     r5, r2.a, r2, r4            // lerp between N and R

texld r0, r4, s0
texld r1, r5, s1

mul r0.rgb, r0, r0.a                // Decompress hdr env map
mul r0.rgb, r0, c2.r                // Brighten hdr env map (reflection)

mul      r1.a, r1.a, r1.a
lrp      r2.rgb, r1.a, r1, r0

mul       r0.rgb, r2, c1            // color hdr env map
mov       r0.a, r0.a
mad       r0, r0, c7.r, c7.g        // attenuation and additive factor

mov       oC0, r0
```

Pixel shader for opaque reflective (non-refractive) metal balls

The transparent spheres in the scene use slightly different shaders which contain refractive terms. Additionally, the rough specular spheres use a per-object LOD bias to sample from a blurrier environment map as well as an image-based diffuse term generated from the original light probe in HDRShop.

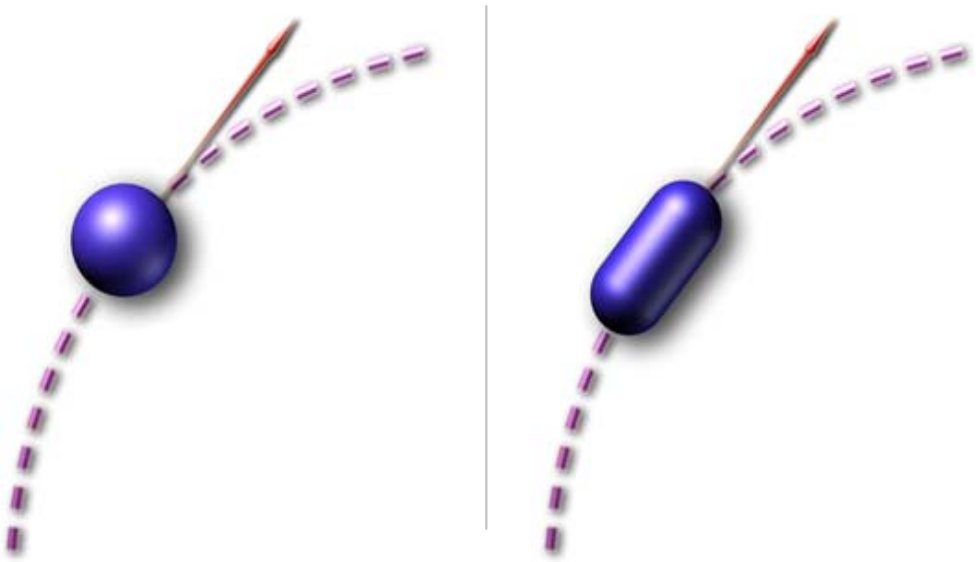
Motion Blur on Balls in Real-Time Animusic Demo

To demonstrate many of the features of the RADEON™ 9700, we implemented a real-time version of the Animusic *Pipe Dream* Animation shown in the SIGGRAPH 2001 Electronic Theater. One simple but powerful shader that we wrote for this real-time demo is the motion blur shader used on the numerous flying balls in the scene. Rather than draw the moving balls several times at different points in space and time as one might do with an accumulation buffer [Haeberli and Akeley 1990], we chose to draw each ball once, distorting its appearance to make it appear as if it were in motion during the frame "exposure." This technique is an extension to the approach taken by [Wloka and Zeleznik 1996]. The previous work eliminates the obvious discrete renderings of the ball which would be inevitable in an accumulation buffer approach, while our shader also accounts for blurring of the object and computing a more accurate approximation to its contribution to the scene over time. To achieve the look we wanted, we used a vertex shader and a pixel shader to distort both the shape and shading of the balls.

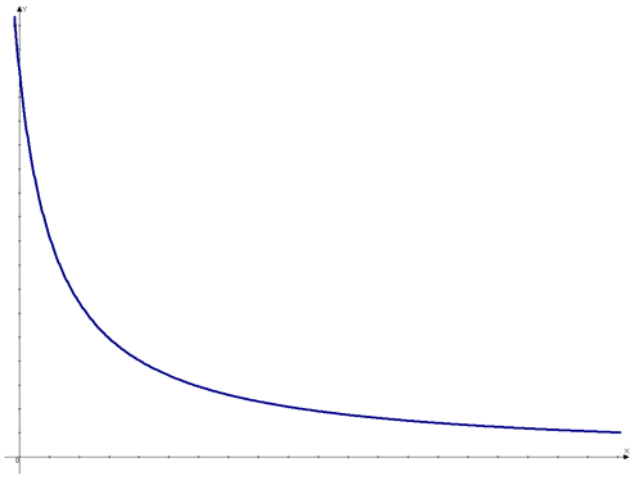
Distorting the Shape

In the diagram below, we show how the ball's shape is distorted by the vertex shader in the direction tangent to the path of motion at the time instant centered in the current finite shutter time. Each ball is modeled as a "capsule" (two hemispheres with a connecting cylinder) which is aligned with the direction of motion. The vertex shader stretches the ball tangent to the direction of motion as shown below. The vertices of the front half of the capsule (those whose normals have a positive dot product with the direction of motion) are pushed in the direction of motion and the vertices of the back half capsule (those whose normals have a negative dot product with the direction of motion) are pulled in the

opposite direction. Naturally, the total amount of stretching is the amount of distance the ball moved since the previous frame.



The capsule geometry is stretched in the vertex shader as a function of distance traveled, d, measured in ball diameter units. The vertex shader also computes a blurriness factor from $1 / (1 + d)$, shown in the curve below. This factor is interpolated across the polygons and used by the pixel shader to determine how much to blur the shading of the balls.



Blurriness factor as a function of distance traveled

As you can see from the curve, if the ball hasn't moved at all, the blurriness factor is one. The blurriness factor falls asymptotically to zero from there. The relevant snippet of vertex shader code, which computes the blurriness factor, is as follows.

```
. . .
// Calculate a 0 to 1 blurriness factor.
mul  r9.y, r9.x, c19.x // /= BallSize => Distance traveled in ball lengths.
add  r9.z, r9.y, c0.z  // += 1 => coverage in ball lengths.
rcp  r9.z, r9.z        // 1/(1+DistanceTraveled/BallSize) => Alpha calculated from speed
sub  r9.z, c0.z, r9.z  // blur factor used for blurriness of ball
mov oT0.w, r9.z        // export blurriness
. . .
```

Blurriness Computation from Motion Blur Vertex Shader

Distorting the Shading

In addition to merely stretching the capsule geometry along the tangent to the path of the ball's motion, the shading of the object is affected by the blurriness factor computed above. There are a number of factors which contribute to the ball color including two specular highlights and an environment map, all of which are blurred as a function of the ball's motion during the frame. In the case of the two specular highlights on each ball, the specular exponent and its intensity are lowered. This serves to spread out the highlight and make it appear to be blurred in the direction of the ball's motion. In essence, the goal is to spread the energy radiating from the specular highlight among the pixels that the specular highlight would move across during a finite frame exposure time. In the case of the environment map, we use the `texldb` pixel shader instruction, which applies a per-pixel bias to selectively sample the smaller mip levels. This blurs the environment map term.

```
ps.2.0
def c0, 0.0, 0.5, 1.0, 2 // common constants
dcl t0.xyzw   // Normal.   Blurriness in alpha
dcl t1.xyz    // View vector
dcl t2.xyz    // Light vector 1
dcl t3.xyz    // Light vector 2
dcl v0.rgb    // ambient lighting
dcl_cube s0   // environment map

// Normalize normal
dp3     r0.w,   t0, t0          // Normal length squared
rsq     r0.w,   r0.w            // 1 / Normal length
mul     r0.xyz, t0, r0.w        // Normalized normal

// Do reflections
dp3     r2.w, t1, r0            // V·N
mul     r2.w, r2.w, c0.w        // 2V
```

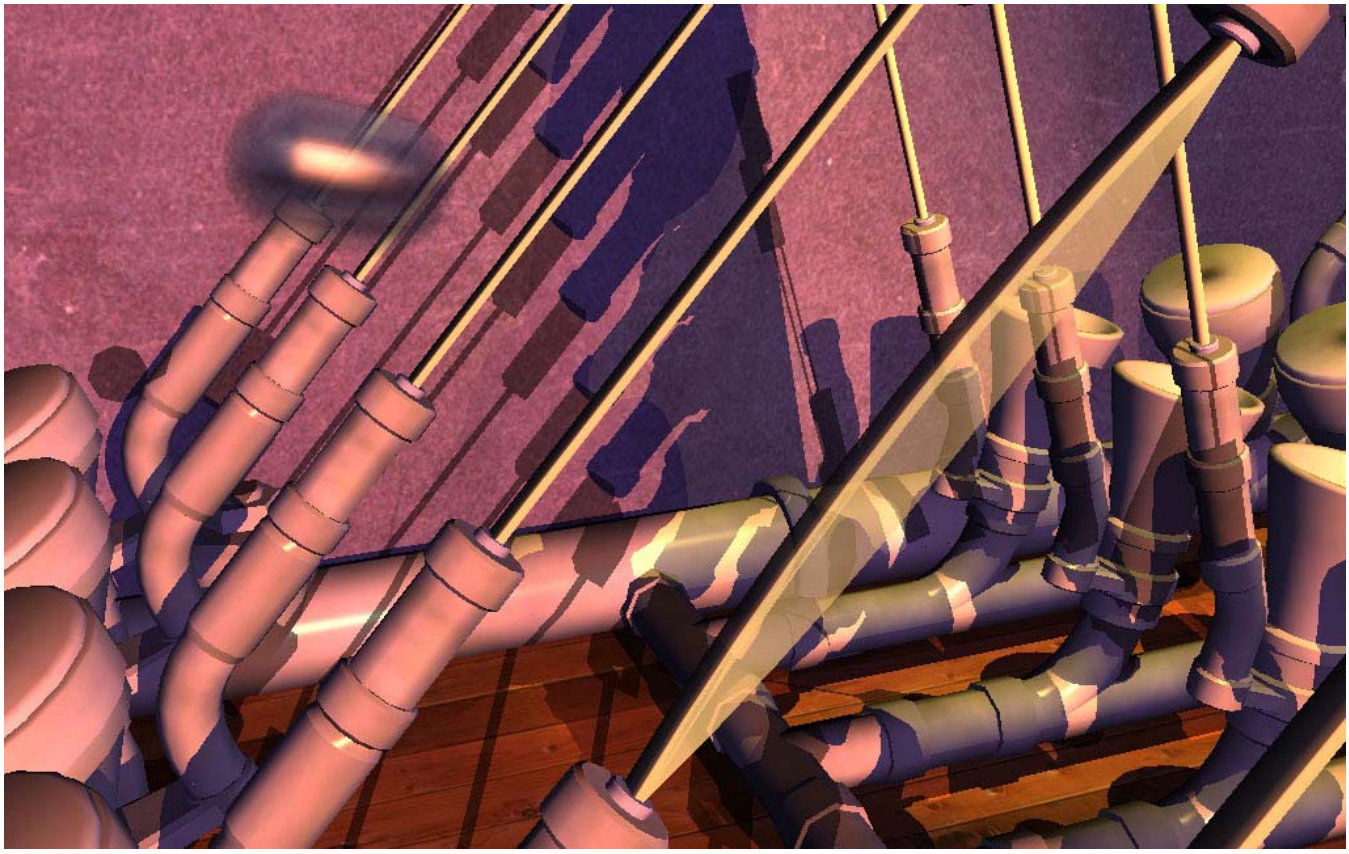
```
mad     r1.xyz, r2.w, r0, -t1    // W=(2V·N)N-V reflected view vector
dp3     r1.w,   r1, r1           // length squared
rsq     r1.w,   r1.w             // 1 / Normal length
mul     r1.xyz, r1, r1.w         // Normalized reflection


// Do blur calculation
mad_sat r2.x, -c0.w, t0.w, c0.w // Clamp(2-2*ext)
mul_sat r2.y, c0.w, t0.w        // Clamp(2*ext)
dp3_sat r2.z, r0, t1            // N·V
sub     r2.z, c0.z, r2.z        // 1-N·V
mad     r2.w, -r2.y, r2.z, c0.z // 1-Clamp(2*ext)*(1-N·V)
mul     r2.w, r2.w, r2.x        // Clamp(2-2*ext) * Clamp(1-2*ext)*(1-N·V)

// Blur texture
mul     r1.w, t0.w, c1.y        // blurAmount * textureBlur
texldb  r5, r1, s0              // Sample reflections
mad     r6.rgb, r5, c2, v0      // Reflection * reflection color + ambient

// Calc specular blur
mul     r3.x, c1.z, t0.w        // Blur amount reduced specular exponent
add     r3.y, r3.x, c7.x        // Add into base specular exp.
max     r4.z, r3.y, c0.z        // Clamp to 1
mul     r3.w, t0.w, c1.w        // Specular dimming.
mad     r3.w, -r3.w, c1.x, c1.x // SpecIntensity-SpecIntensity*specular dim.

// Light 1
dp3_sat r3.x, r1, t2            // R·L
pow     r3.y, r3.x, r4.z        // R·L^k
mul     r3.z, r3.y, r3.w        // Dim specular
mul     r7.rgb, c3, r3.z        // *= LightColor * Falloff

// Light 2
dp3_sat r3.x, r1, t3            // R·L
pow     r3.y, r3.x, r4.z        // R·L^k
mul     r3.z, r3.y, r3.w        // Dim specular
mad     r7.rgb, c4, r3.z, r7    // *= LightColor * Falloff

// Combine
add     r0.rgb, r6, r7          // Diffuse + Specular
mov     r0.a, r2.w              // Copy blur to alpha
mul     r0.rgb, r0, r2.w        // Premultiply Src * SrcAlpha

mov     oC0, r0                 // Output
```

Motion Blurred Ball Pixel Shader

In the last few instructions of the pixel shader, the diffuse and specular components of illumination are combined. Because the specular contribution can be greater than one, we perform part of the frame-buffer compositing operation (Src * SrcAlpha) in the pixel shader before the colors are clamped to the zero to one range. Each pixel is composited with the frame buffer with a src + srcAlpha * Dest blend. Doing the Src * SrcAlpha premultiplication in the pixel shader gives a more accurate result since it happens prior to pixel shader output color saturation.

Motion Blur on Strings in Real-Time Animusic Pipe Dream Demo

The motion blur on the strings in the real-time Animusic Pipe Dream is considerably simpler than it is for the balls, but the effect is still quite convincing. Essentially, the goal is to render a string moving rapidly back and forth at a frequency much higher than the frame rate and the simulated finite frame exposure time. For a given frame of our rendering, a moving string has a given amplitude. That string is motion blurred by drawing two instances of it. One of the instances of the plucked string is drawn opaque with both of the long sides of the string geometry bowed in the same direction. The amount of bowing is equal to the amplitude of the string vibration during that frame. The other instance of the string geometry is drawn semitransparent with the long sides of the string geometry bowed away from each other, covering the full range of motion of the vibrating string for that frame. The transparency of this second string instance is proportional to the amplitude. The effect of this motion blur is shown below.



A Motion Blurred Ball and String from the Real-Time Animusic Pipe Dream Demo

Two-tone Layered Car Paint Model

The next shader that we'll cover is an interesting pixel shader effect simulating two-tone car paint. The car model shown here has a relatively modest number of polygons, but uses a normal map generated from an appearance preserving simplification algorithm [Cohen et al 1998]. Due to the pixel shader operations performed and the subtle gradients across areas such as the hood, a 16 bit per channel normal map is necessary.

Normal Map Decompression

The first step in this pixel shader is normal decompression. Since the normals are stored in surface local coordinates (*aka* tangent space), we can assume that the *z* component of the normals will be positive. Thus, we can store *x* and *y* in two channels of a 16-16 texture map and derive *z* in the pixel shader from +sqrt ($1 - x^2 - y^2$). This gives us much higher precision than a traditional 8-8-8-8 normal map (even 10 or 11 bits per channel is not enough for this particular shader) for the same memory footprint.

Base Color

The normal decompression described above is performed on a surface normal map which is generated from an appearance preserving simplification process (N) and a high frequency normalized vector noise map (N_n) which is repeated across the surface. These two normals are used to compute two

perturbed normals which are used simulate the two-toned nature of the paint as well as the microflake suspended in an inner coat of the paint. These normals, N_s and N_{ss} are computed as follows:

$$N_s = (aN_n + bN) / |aN_n + bN| \text{ , where } a << b$$
$$N_{ss} = (cN_n + dN) / |cN_n + dN| \text{ , where } c = d$$

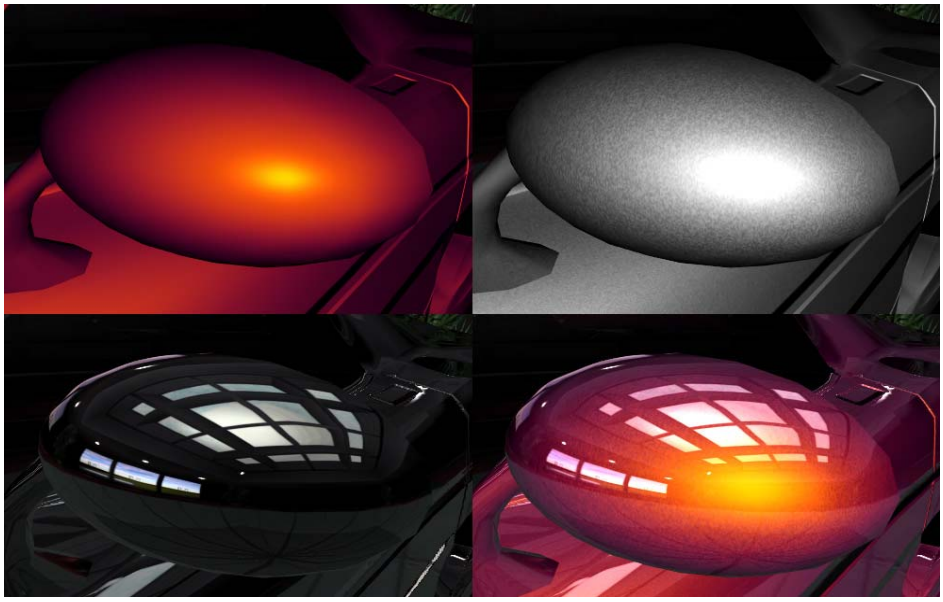
The coefficients *a*, *b*, *c* and *d* above are constant input parameters to the pixel shader which essentially determine the distributions of the perturbed normals which in turn determine the width of the region in which the microflake is readily visible. These two normals are dotted with the view vector and used as parameters in the following polynomial, which determines the color of the base coat and strength of the microflake term:

$$c_0(N_s \cdot V) + c_1(N_s \cdot V)^2 + c_2(N_s \cdot V)^4 + c_3(N_{ss} \cdot V)^{16}$$

Essentially, the first three terms of this polynomial perform the blend between the two tones of the paint and the fourth adds an extra layer of sparkling from the microflake.

Clear Coat Paint Layer and Rough Specular Trim

The final step in rendering the painted areas of the car is inclusion of the clear coat through addition of an environment map as shown below.



Two-tone, microflake, clear coat and final lighting on side rear-view mirror

There is one interesting aspect of the clear coat term which the decision to store the environment map in an RGBScale form to simulate high dynamic range in a low memory footprint. The alpha channel of the texture, shown on the right below, represents $1/16^{th}$ of the true range of the data while the RGB, shown on the left below, represents the normalized color. In the pixel shader, the alpha channel and RGB channels are multiplied together and multiplied by eight to reconstruct a cheap form of HDR reflectance. This is multiplied by a subtle Fresnel term before being added to the lighting terms described above.

RGB and Scale channels of top face of HDR cubic environment map

Trim areas of the car such as the black hatch area on the back and the grooves around the door are separated out by monochrome base map. The monochrome map knocks out the metal shading for those pixels of the car surface and also determines a per-pixel LOD bias to use when accessing the environment map. This allows us to cheaply simulate more rough surfaces in trim areas. We also chose to convert the color from the environment map to a luminance value in the pixel shader because we liked the look. A rough-specular area is shown on the following page.

Rough specular on black hatch and other trim

The full pixel shader for the car paint and trim is shown below.

```
PsConst 2 (8.0, 0.0, -0.5, 1.0)    // hdr map boost, hdr_alpha_no_glow, fresnel scale, fresnel bias
PsConst 3 (-4.0, 4.0, 12.9, 0.50) // trim miplod scale, trim miplod bias, trim env scale, trim env bias
PsConst 4 (0.10, 1.00, 16.0, 0.0) // sparkle str (color), sparkle str (glistens), exp for shimmering, 0
PsConst 9 (0.500, 100.0, 0.0, 0.0)// hdr_alpha_no_glow, hdr_glow_boost

PsConst 5 (0.4, 0.0, 0.35, 0.0)    // car color 0
PsConst 6 (0.6, 0.0, 0.00, 0.0)    // car color 1
PsConst 7 (0.0, 0.35, -0.35, 0.0) // car color 2
PsConst 8 (0.5, 0.5, 0.0, 0.0)     // car sparkle color

ps.2.0
def c0, 0.0, 0.5, 1.0, 2.0
def c1, 0.0, 0.0, 1.0, 0.0
dcl_2d s0   // base map
dcl_2d s1   // normal map
dcl_cube s2 // cube env map
dcl_2d s3   // sparkle map (color shift)
dcl_2d s4   // sparkle map (sparkles)

dcl t0      // tex coords
dcl t1      // tanx
dcl t2      // tany
dcl t3      // tanz
dcl t4      // view vector
dcl t5      // sparkle map tex coords

texld r0, t0, s1  // fetch from normal map
texld r5, t0, s0  // fetch from grayscale gloss map
texld r8, t5, s3  // fetch from sparkle map (color shift)
texld r9, t5, s4  // fetch from sparkle map (sparkles)

dp2add r0.a, r0, r0, -c0.z     // 1 - X^2 + Y^2
rsq r0.a, r0.a                 // 1 / sqrt(1 - X^2 + Y^2)
rcp r0.b, r0.a                 // z = sqrt(1 - X^2 + Y^2)

mad r3, r8, c0.w, -c0.z        // bx2 on sparkle map (color shift)
```

```
mad r6, r3, c4.r, r0          // hack to add sparkle map to normal map

mad r3, r9, c0.w, -c0.z       // bx2 on sparkle map (sparkles)
mad r7, r3, c4.g, r0          // hack to add sparkle map to normal map

mad r1.a, r5.r, c3.x, c3.y    // scale and bias for miplod bias based on alpha

dp3  r4.a, t4, t4             // view vector mag squared
rsq  r4.a, r4.a               // 1/ view vector mag squared
mul  r4, t4, r4.a             // normalized view vector V

// reflection vector for env map
mul r2.rgb, r0.x, t1
mad r2.rgb, r0.y, t2, r2
mad r2.rgb, r0.z, t3, r2      // transform bump map normal into world space
dp3  r2.a, r2, r2             // bump mag squared
rsq  r2.a, r2.a               // 1 / view vector mag squared
mul  r2.rgb, r2, r2.a         // normalized view vector bump

dp3_sat r2.a, r2, r4          // Nn·V
mul r3, r2, c0.w              // 2Nn
mad r1.rgb, r2.a, r3, -r4     // Rn = 2Nn(Nn·V)-V  (non sparkle reflection vector)

// sparkles for color map
mul r10.rgb, r6.x, t1
mad r10.rgb, r6.y, t2, r10
mad r10.rgb, r6.z, t3, r10    // transform bump map normal into world space
dp3  r10.a, r10, r10          // bump mag squared
rsq  r10.a, r10.a             // 1 / view vector mag squared
mul  r10.rgb, r10, r10.a      // normalized view vector bump

dp3_sat r6.a, r10, r4         // Ns·V

// sparkles for env map
mul r10.rgb, r7.x, t1
mad r10.rgb, r7.y, t2, r2
mad r10.rgb, r7.z, t3, r2     // transform bump map normal into world space

dp3  r10.a, r10, r10          // bump mag squared
rsq  r10.a, r10.a             // 1 / view vector mag squared
mul  r10.rgb, r10, r10.a      // normalized view vector bump

dp3_sat r7.a, r10, r4         // Ns.V

texldb r0, r1, s2             // fetch from env map

mul r0.rgb, r0, r0.a          // RGBScale HDR expansion

mov_sat r3, r0
dp3 r3, r3, c3.w              // scale env map intensisty for trim (use dp3 to add channels together)
                              // grayscale conversion is a trick to make trim seem blacker

mul r0.rgb, r0, c2.r          // HDR brightening for painted regions

mov r4.a, r6.a
mul r4.rgb, r4.a, c5          // (Ns·V) * Car Color 0
mul r4.a, r4.a, r4.a
mad r4.rgb, r4.a, c6, r4      // (Ns·V)^2 * Car Color 1
mul r4.a, r4.a, r4.a
mad r4.rgb, r4.a, c7, r4      // (Ns·V)^4 * Car Color 2
pow r4.a, r7.a, c4.b

mad r4.rgb, r4.a, c8, r4      // (Nss.V)^16 * sparkle color

mad r1.a, r2.a, c2.z, c2.w    //  1.0 - 0.5 * N·V
mad r6.rgb, r0, r1.a, r4      // env map * fresnel + N·V * Car Color

lrp r6.rgb, r5.r, r6, r3      // lerp between car paint, and trim effect
sub r6.a, r0.a, c9.x
mul r6.a, r6.a, c9.y

mov oC0, r6
```

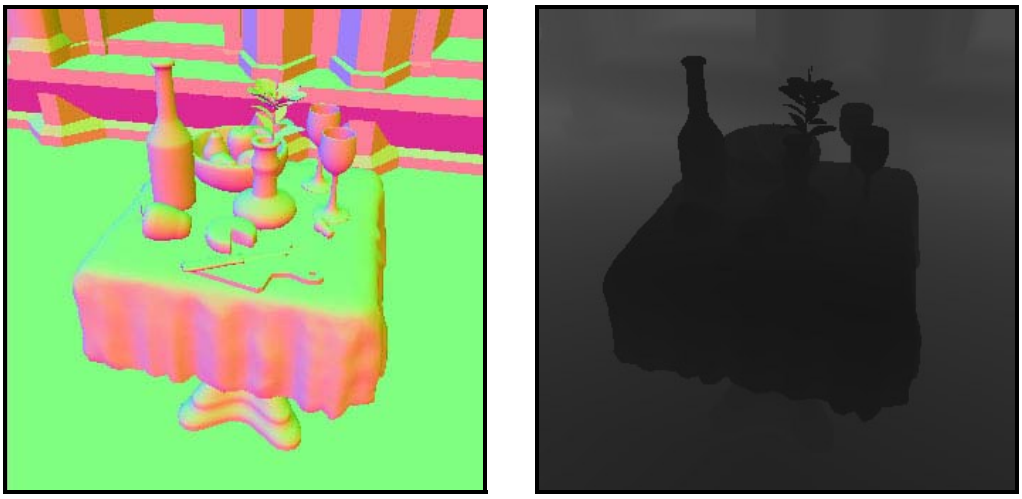
Car Paint – (a) Two-tone albedo, (b) microflake, (c) clearcoat and (d) all layers

Image-Space Outlining

In Non-Photorealistic Rendering (NPR), outlines at object silhouettes, shadow edges and texture boundaries are important visual cues which have previously been difficult to generate in real-time. In the SIGGRAPH 2002 sketch *Real-Time Image-Space Outlining for Non-Photorealistic Rendering*, we present an image-space technique which uses pixel shading hardware to generate these three classes of outlines in real time. In all three cases, we render alternate representations of the desired scene into texture maps which are subsequently processed by pixel shaders to find discontinuities corresponding to outlines. The outlines are then composited with the shaded scene.

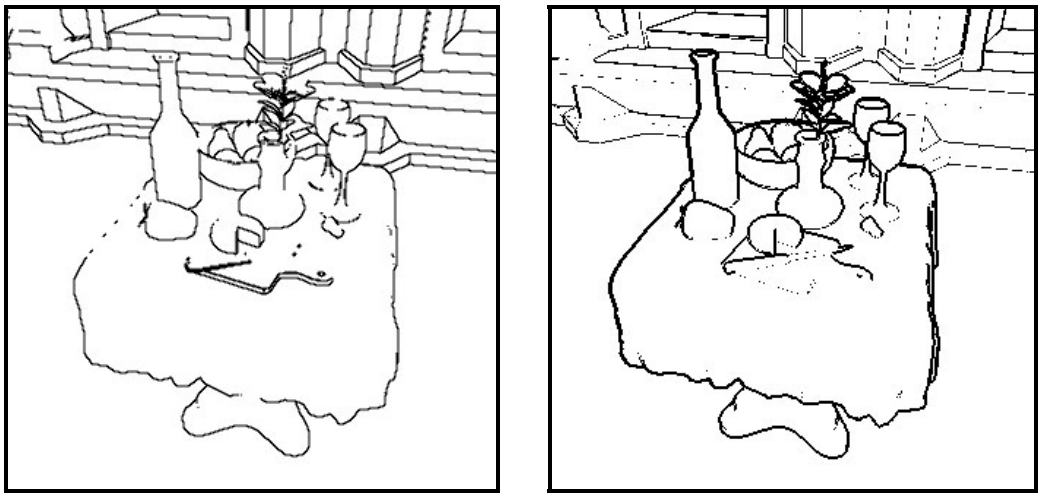
Outlining Silhouettes and Creases

For the cases of silhouette and crease outlining, we render the full scene's world-space normals and eye-space depths into an RGBA (nx_{world}, ny_{world}, nz_{world}, $depth_{eye}$) texture using a vertex shader to populate the color channels [Decaudin1996; Saito and Takahashi 1990].



Normals and Depths of Still Life Scene

This texture is subsequently processed using a discontinuity filter implemented as a pixel shader and composited over the shaded image.

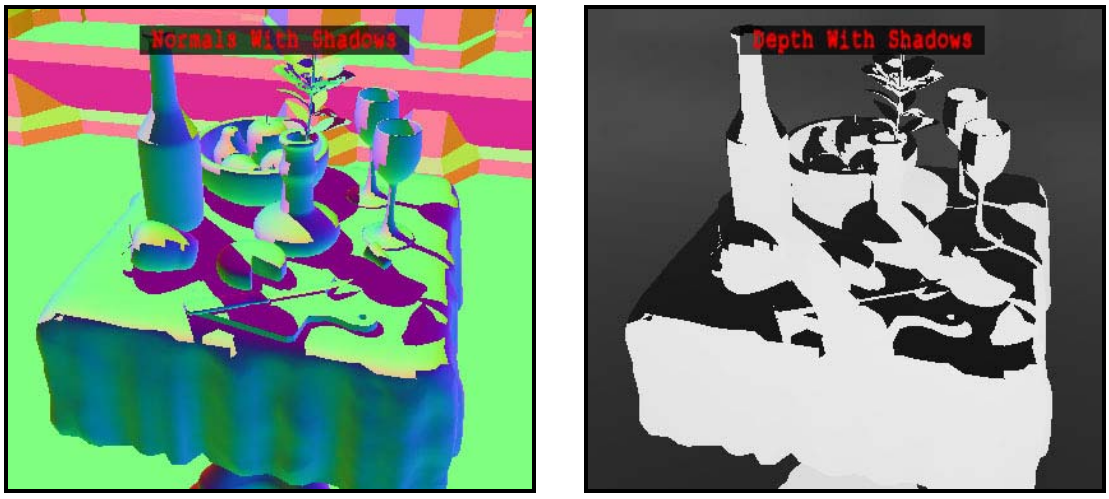


Edges from Normals and Depths Shown Independently

With this technique, creases and silhouettes both produce discontinuities in the image. Note that we are not processing the contents of the depth buffer, but just happen to be generating a texture channel that has a semantic meaning that is the same as the depth buffer.

Outlining Shadows

To outline shadow edges, we employ the same renderable texture and image filtering technique used for silhouettes and creases above. In the case of shadows, we use geometric shadow volumes and the stencil buffer to flag pixels which are in shadow with respect to a single light source. For these pixels, we negate the world-space normal before it is written to the renderable texture. This creates additional discontinuities in the $\{nx_{world}, ny_{world}, nz_{world}, depth_{eye}\}$ image which align with the shadow boundaries as shown below. Multiple lights can be handled with multiple $\{nx_{world}, ny_{world}, nz_{world}, depth_{eye}\}$ images and composited with the shaded image.



Normals and Depths Negated in Shadowed Pixels

Outlining Texture Boundaries

Texture boundaries lie between areas of an object which are textured differently but which do not generate a geometric discontinuity. To handle this case, we use a technique similar to priority shadow buffers [Hourcade 85]. The regions of different texture are colored with IDs which are suitably different from adjoining regions as shown below.

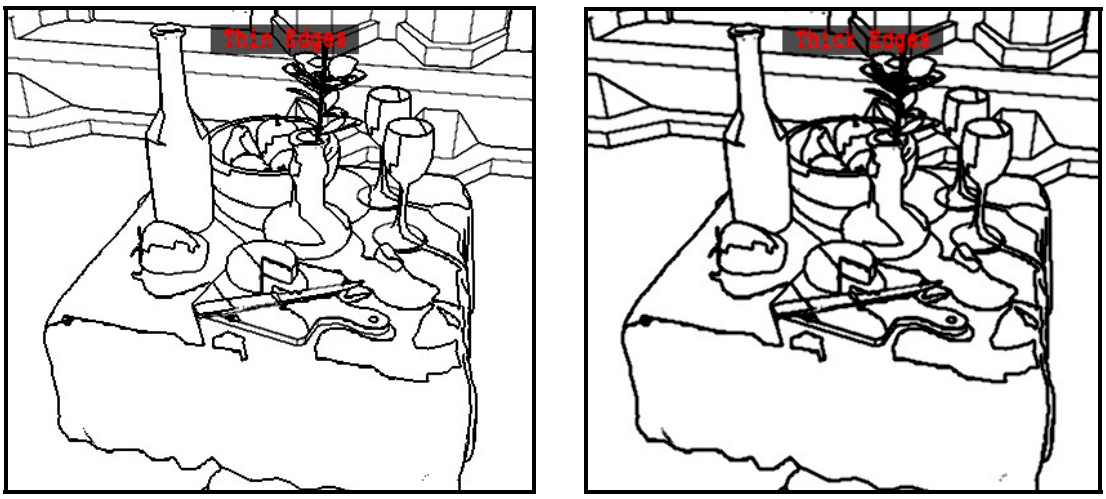
Region IDs and resulting discontinuities

This creates a discontinuity in the ID image. We have found that applying an ID map of insufficient resolution can lead to edge detection issues due to filtering of the ID texture. Specifically,

extreme magnification and bilinear filtering of an ID texture can lead to poor detection of the texture boundary edges. To combat this, we have used relatively large textures with linear-mip-nearest filtering.

Enhancing Outlines with Morphology

To further allow us to customize the look of a scene, we can choose to dilate or erode the edge image prior to compositing it with the shaded scene. This is done with dilation or erosion pixel shaders [Mitchell 02].



Original thin edges and the application of a dilation operator to thicken them

Optimization

It is possible to reduce the cost of compositing the edge images with the shaded scene using alpha testing to mask out non-edge pixels. Since there are typically a large number of non-edge pixels in a scene, as seen in the edge images above, this can be a significant memory bandwidth savings. This optimization can also be applied when dilating the edge images. Additionally, the ability to simultaneously output separate colors to multiple render targets enables us to optimize the initial rendering of the G-Buffer data. For example, we could output a scene's world-space normal vectors to one RGBA buffer while simultaneously outputting scalar screen-space depths to a 32-bit floating point buffer and RGBA IDs to a third buffer. These three buffers can be subsequently post-processed to extract edges that can be composited with a shaded image.

Pixel Shader Code

With DirectX® 9 2.0 pixel shaders, we can sample and filter the the normal, depth and texture ID images in one pass as shown below.

```
ps.2.0
def    c0,  0.0f, 0.80f, 0, 0          // normal thresholds
def    c3,  0, .5, 1, 2
def    c8,  0.0f, 0.0f, -0.01f, 0.0f   // Depth thresholds
def    c9,  0.0f, -0.25f, 0.25f, 1.0f
def    c12, 0.0f, 0.01f, 0.0f, 0.0f    // TexID Thresholds
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 (Signed result -1 to 1)
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 black/white 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 // Scale and bias result

// Z
add r10.r, r0.a, -r1.a     // Take four deltas
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
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
```

Conclusion

In summary, we have outlined the shading capabilities of the new RADEON™ 9700 graphics processor and illustrated its power with a series of examples. After a brief overview of the vertex and pixel shading programming models on the RADEON™ 9700, we have presented a detailed description of a variety of shaders including the required shiny bumpy, Homomorphic BRDF and procedural wood shaders. In addition to these compulsory shaders, we have presented a discussion of high dynamic range rendering, motion blur, two-tone paint and image-space cartoon outlining shaders.

Acknowledgments

Thanks to my colleagues at ATI for their valuable input, art and demo help, particularly John Isidoro, Chris Brennan, Eli Turner, Scott LeBlanc, Alex Vlachos and Dave Gosselin. Thanks to Paul Debevec and his team for the use of their HDR light probes and the *Rendering with Natural Light* dataset. Thanks also to Wayne Lytle and the crew at Animusic for their help with the real-time version of their *Pipe Dream* animation.

References

- [Apodaca and Gritz 01] Anthony A. Apodaca and Larry Gritz, *Advanced RenderMan: Creating CGI for Motion Pictures*, Morgan Kaufman, 2001
- [Brennan 02] Brennan, Chris. "Diffuse Cube Mapping." In *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, Wolfgang Engel editor, Wordware
- [Cohen et al 98] Cohen, Jonathan, Marc Olano, and Dinesh Manocha. *Appearance-Preserving Simplfication*. Proceedings of SIGGRAPH 98 (Orlando, Florida, July 19-24, 1998). In *Computer Graphics*, Annual Conference Series, ACM SIGGRAPH, 1998.
- [Debevec et al 98] Debevec, Paul. *Rendering Synthetic Objects Into Real Scenes: Bridging Traditional and Image-Based Graphics With Global Illumination and High Dynamic Range Photography*, Proceedings of SIGGRAPH 98 pp. 189-198 (July 1998, Orlando, Florida). See also: http://www.debevec.org
- [Decaudin 96] Decaudin, P.. *Cartoon-looking rendering of 3D-scenes*. Technical Report INRIA 2919, Université de Technologie de Compiègne, France. 1996
- [Haeberli and Akeley 1990] Paul E. Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. SIGGRAPH 1990, pages 309-318, August 1990.
- [Hourcade 85] Hourcade, J. C. and Nicolas, A.. *Algorithms for Antialiased Cast Shadows*. In Computer and Graphics, 9, 3, 259-265. 1985
- [Mitchell 02] Mitchell, J. L. 2002. Image Processing with Direct3D Pixel Shaders. In *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, Wolfgang Engel editor, Wordware
- [Saito and Takahashi 90] Saito, T and Takahashi, T. *Comprehensible Rendering of 3-D Shapes*. In Proceedings of SIGGRAPH 1990, ACM Press / ACM SIGGRAPH, New York. 197-206
- [Vlachos 02] Alex Vlachos, John Isidoro and Chris Oat "Textures as Lookup Tables for Per-Pixel Lighting Computations" in *Game Programming Gems III*, 2002.
- [Wloka and Zeleznik 1996] Wloka, M. and Zeleznik, R.C. "Interactive Real-Time Motion Blur", *Visual Computer,* Springer Verlag, 1996.

Resources

- ATI Developer Relations: http://www.ati.com/developer
- McCool's sample application:
  - http://www.cgl.uwaterloo.ca/Projects/rendering/Papers/#homomorphic
- For more on Animusic, see http://www.animusic.com