

Chapter 3

ATI

Jason L. Mitchell

Pixel Shading with DirectX 8.1 and the ATI RADEON™ 8500

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

Introduction

Programmable shaders are a powerful way to describe the interaction of surfaces with light, as evidenced by the success of programmable shading models like RenderMan and others. As graphics hardware evolves beyond the traditional “fixed function” pipeline, hardware designers are looking to programmable models to empower the next generation of real-time content. To allow content to interface with current programmable pixel shading hardware, we have designed the 1.4 pixel shader model (ps.1.4) exposed in DirectX 8.1 and supported by the ATI RADEON™ 8500. In these notes, we will outline the structure of the programming model and present some illustrative examples. In the companion notes distributed at SIGGRAPH, we will show implementations of the common example shaders used throughout this course (bumped cubic environment mapping, McCool BRDF and parameterized volumetric wood) as well as a new programming model which goes beyond ps.1.4. Soft copies of these notes and the supplemental material distributed at SIGGRAPH 2002 are available at <http://www.ati.com/developer>.

The ps.1.4 Programming Model

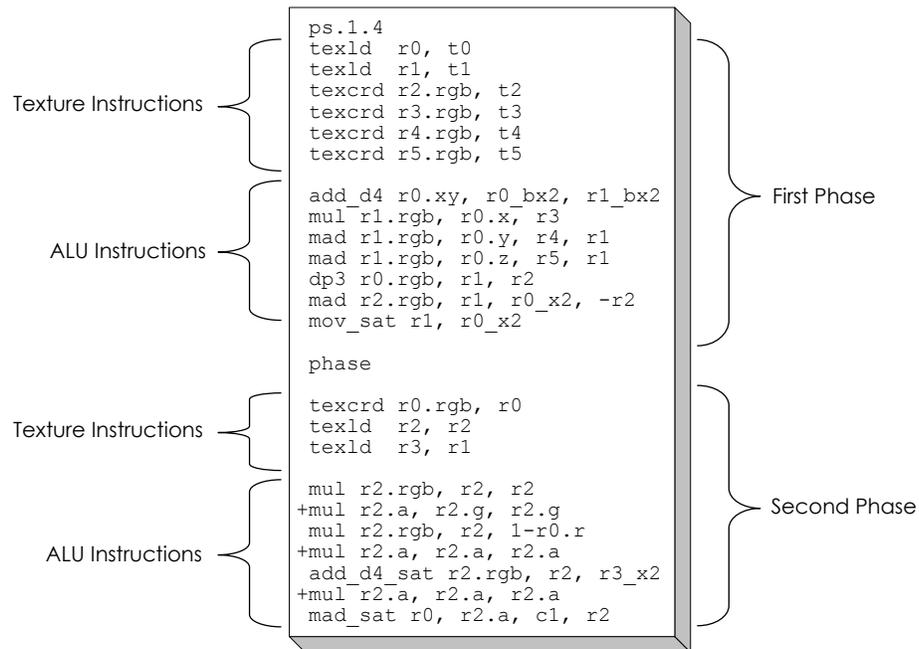
The 1.4 pixel shader programming model (ps.1.4), introduced in DirectX 8.1 in late 2001, advances the previously available programming model by applying a RISC approach. That is, the same micro operations which can be applied to colors can also be applied to texture addresses. This allows a wider variety of pixel shading affects to be achieved, as well as backward compatibility with previously available CISC models.

Inputs and Outputs

The pixel shader may take as inputs the data from interpolated texture coordinates, samples from texture maps, constant colors, the diffuse interpolator or the specular interpolator. There are six sets of texture coordinates (t_0-t_5), which may be used as extra interpolated data or as texture coordinates for sampling texture maps. There are six texture maps available in the ps.1.4 model and eight read-only constant registers (c_0-c_7). The low-precision diffuse (v_0) and specular (v_1) interpolators may also be used as arguments to ALU operations. There are six read-write temp registers (r_0-r_5) available in the ps.1.4 model. The contents of the r_0 temp register are considered the RGBA output of the pixel shader.

Shader Structure

A ps.1.4 shader may contain one or two *phases*, each of which begins with up to 6 texture instructions and ends with up to 8 ALU instructions. Each of the ALU instructions may be *co-issued*.



The shader shown above has two phases. The first phase uses six texture instructions (the maximum) and 7 ALU instructions. The second phase uses three texture instructions (two of which are *dependent reads*) and 4 ALU instructions (the first three of which are co-issued). The *phase* instruction marks the boundary between the phases.

ALU Instructions

The instruction set available for ALU operations is a fairly traditional set of arithmetic operations and comparators as listed below.

add d, s0, s1	$s0 + s1$
sub d, s0, s1	$s0 - s1$
mul d, s0, s1	$s0 * s1$
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	$d = (s2 \geq 0) ? s0 : s1$
dp3 d, s0, s1	$s0 \cdot s1$ replicated to d.rgba
dp4 d, s0, s1	$s0 \cdot s1$ replicated to d.rgba
bem d, s0, s1, s2	Macro for EMBM

The inputs to the ALU instructions may be any of the temporary registers ($r0-r5$) or constant registers ($c0-c7$). The diffuse interpolator ($v0$) and specular interpolator ($v1$) may be inputs to ALU instructions in the second phase of the shader.

Argument Modifiers

As shown in the sample shader on the previous page, arguments to ALU instructions may have modifications made to them prior to the operation of the ALU instruction. There are five argument modifiers which can be used to perform operations such as negation, inversion, scaling and conversion from the $[0..1]$ range to the $[-1..1]$ range.

rn_bias	Bias
$1 - rn$	Invert
$-rn$	Negate
rn_x2	Scale by 2
rn_bx2	Signed Scaling

Source Register Selectors

It is often useful to think of the individual components of an RGBA vector as independent scalars. With source register selectors, it is possible to extract these scalars from an argument register and replicate them across all channels of the argument. The four source register selectors are shown below.

.r	Replicate Red
.g	Replicate Green
.b	Replicate Blue
.a	Replicate Alpha

Arbitrary Write Masks

It is often desirable to write to only a subset of the channels of a destination register. In ps.1.4, destination write masks can be used in any combination as long as the masks are ordered r, g, b, a. This allows the shader to execute a sequence of ALU operations which write to different components of the same destination register. This is especially useful when computing texture coordinates to be used in dependent texture reads, as we will illustrate later.

Instruction Modifiers

In some cases, we wish to modify the result of an ALU instruction as it is written into the destination register. In the ps.1.4 model, we can use instruction modifiers to perform shifts and saturates on the results of ALU operations. There are six shift (multiplier or divider, depending on the direction of the shift) operations that we can perform. Additionally, ALU results may be explicitly saturated to the [0..1] range. Saturation and shifting may be performed on the same ALU instruction.

<i>instr_x2</i>	Multiply by 2
<i>instr_x4</i>	Multiply by 4
<i>instr_x8</i>	Multiply by 8
<i>instr_d2</i>	Divide by 2
<i>instr_d4</i>	Divide by 4
<i>instr_d8</i>	Divide by 8
<i>instr_sat</i>	Saturate (clamp from 0 and 1)

Co-Issue

Pairing or *co-issuing* of ps.1.4 instructions is indicated by a plus sign (+) preceding the second instruction of the pair. The first instruction of the pair is a vector instruction which may write to any or all of r, g and b of the destination register. The second instruction of the pair is a scalar which writes into the alpha channel of the destination register. As an example, consider the following instructions:

```
mul r0.rgb, t0, v0 // Component-wise multiply of the colors
+add r1.a, r1, c2 // Add an alpha component at the same time
```

The dot product instructions may not be executed in the alpha pipeline, as they are always vector instructions.

Texture Instructions

The two most common texture instructions are the `texcrd` and `texld` instructions. The `texcrd` instruction is used to specify that a given temporary register (`r0-r5`) is to contain interpolated data. The `texld` instruction uses the specified texture coordinates to sample data from a texture map into the destination register. For example, the following `texcrd` instruction causes `r0` to contain interpolated data from the 0th set of texture coordinates:

```
texcrd r0.rgb, t0
```

The following `texld` instruction causes `r1` to contain sampled data from the 1st texture using the 1st set of texture coordinates:

```
texld r1, t1
```

The following `texld` instruction causes `r2` to be loaded with sampled data from the 2nd texture using the contents of `r3` as texture coordinates:

```
texld r2, r3
```

Using the contents of a temporary register as texture coordinates (the second argument of a `texld` instruction) is the definition of a dependent read because these texture coordinates *depend* upon the earlier ALU ops used to compute them (in this case `r3`). Naturally, a dependent read can only be used at the top of the second phase.

The `texkill` instruction can be used to kill pixels based upon results computed in a pixel shader. This is similar to alpha-testing, but more general in that multiple conditions may be tested with the `texkill` instruction. Multiple `texkill` instructions may appear in a single shader.

The final texture instruction is the `texdepth` instruction, which causes the current pixel's `z` to be replaced with the contents of a given register component. This instruction can be used to implement `z`-sprites, `z`-correct bump mapping and other effects. Naturally, only one `texdepth` instruction may be present in a given pixel shader.

Texture Projection

Any `texld` instruction may be modified to express a projected texture access. This includes projective *dependent* reads, which are fundamental to doing reflection and refraction mapping of things like water surfaces. Syntax looks like this:

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

Projective loads are useful for projective textures like refraction maps or for doing a divide, as we will show later in the skin shader [Vlachos02].

Example ps.1.4 Shaders

Now that we have introduced the structure and syntax of 1.4 pixel shaders, we will illustrate their usage in a variety of practical applications.

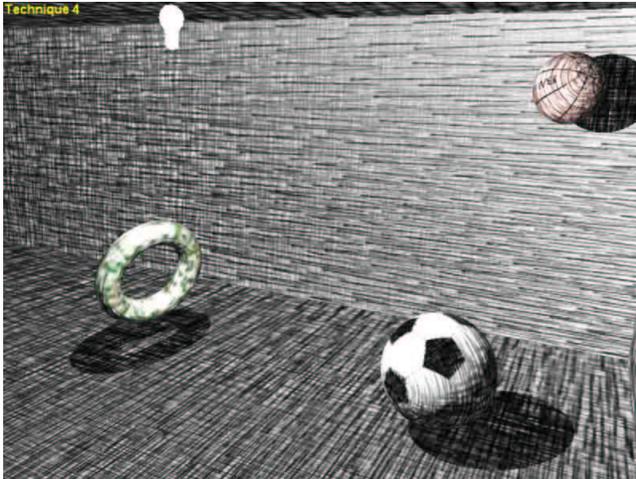
Real-Time Hatching

The first application of the 1.4 pixel shading model that we will illustrate is the Real-Time Hatching technique shown at SIGGRAPH last year [Praun01]. The general goal of this pixel shader is to compute the linear combination of 6 channels of a Tonal Art Map (TAM). The coefficients defining this linear combination have been computed in the vertex shader as a function of $N \cdot L$ with respect to a given light source and are stored in the `r`, `g` and `b` components of the 1st and 2nd texture coordinates.

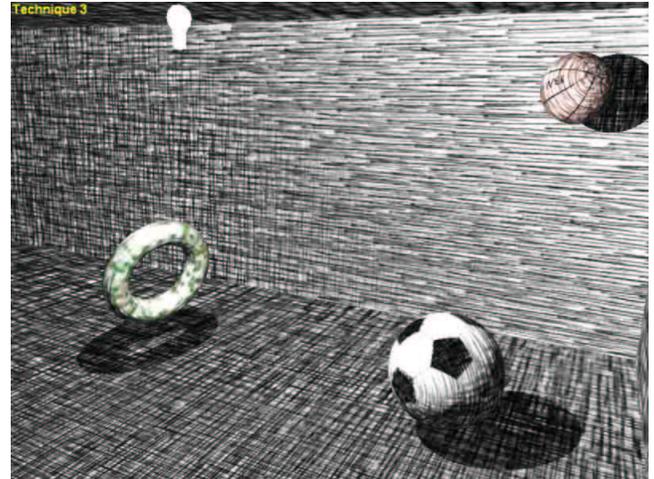
```
ps.1.4
texld r0, t0           ; sample the first three channels of the TAM
texld r1, t0           ; sample the second three channels of the TAM
texcrd r2.rgb, t1.xyz  ; get the 123 TAM weights and place in register 2
texcrd r3.rgb, t2.xyz  ; get the 456 TAM weights and place in register 3
dp3_sat r0, 1-r0, r2   ; dot the reg0 (TAM values) with reg2 (TAM weights)
add_sat r1, 1-r1, r3   ; dot the reg1 (TAM values) with reg3 (TAM weights)
add_sat r0, r0, r1     ; add reg 0 and reg1
mov_sat r0, 1-r0      ; complement and saturate
```

Real-Time Hatching with Per-Vertex TAM weights

One side effect of this approach is inaccurate lighting due to the fact that the TAM weights are computed at the vertices and interpolated. This can cause artifacts when the light source is close to a large polygon. The two-polygon wall in the image on the left side of the figure below seems to have its hatches grayed out as it transitions from the top right corner of near white, to the other corners which are near black. The wall in the image on the right shows the effect of per-pixel TAM weights, correctly transitioning between the intermediate hatching levels across the polygon.

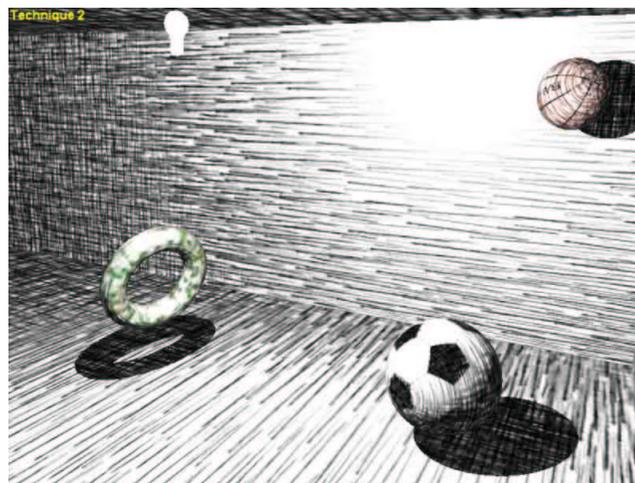


Per-Vertex TAM Weights



Per-Pixel TAM Weights

Another dramatic improvement that can be made to the hatching shader is inclusion of a per-pixel distance attenuation term as shown below.



Per-Pixel Distance Attenuation and TAM weight computation

The improved shader interpolates $N \cdot L$, modulates it with per-pixel distance attenuation and uses this scalar as a texture coordinate to look up the per-pixel TAM weights. The two 1D RGB function textures used to look up the TAM weights based on $N \cdot L$ are shown here:



Two 1D RGB textures used to determine Per-Pixel TAM Weights from $N \cdot L$

After computing the 6-term linear combination of hatching patterns in the TAM as before, the color is tinted to match a base texture map color.

```

ps.1.4
def c0, 1.00f, 1.00f, 1.00f, 1.00f
def c1, 0.30f, 0.59f, 0.11f, 0.00f ; RGB to luminance conversion weights

texcrd r1.rgb, t2 ; N·L
texld r4, t3 ; Intensity map looked up from light space position
texld r5, t0 ; Base Texture

mul_x2 r4, r4.r, r1.r ; N·L * attenuation
add r4, r4, c2 ; += ambient
dp3 r3, r5, c1 ; Intensity of base map
mul r5, r4, r5 ; Modulate base map by light
mul r4, r4, r3 ; Modulate light by base map intensity

phase

texld r0, t1 ; sample the first three channels of the TAM
texld r1, t1 ; sample the second three channels of the TAM
texld r2, r4 ; Get weights for 123
texld r3, r4 ; Get weights for 456

dp3_sat r0, 1-r0, r2 ; dot the reg0 (TAM values) with reg2 (TAM weights)
dp3_sat r1, 1-r1, r3 ; dot the reg1 (TAM values) with reg3 (TAM weights)
add_sat r0, r0, r1 ; add reg0 and reg1
mul r0.rgb, 1-r5, r0 ; Color hatches with base texture
mov_sat r0, 1-r0 ; complement and saturate

```

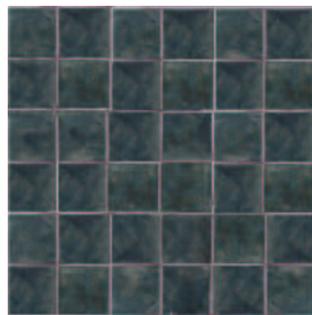
Real-Time Hatching with Per-Pixel TAM weights, distance attenuation and color tinting

Per-pixel Variable Specular power

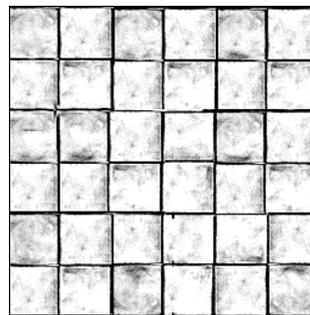
In the preceding example, we have illustrated the ability to migrate one type of per-vertex computation (TAM weight calculation) to the pixel level in order to improve rendering quality. We will now show how to implement per-pixel material properties (in this case, specular exponent) by using arbitrary register write masks and dependent texture reads in ps.1.4. We will use three different texture maps in this shader:

1. Albedo / Gloss map
2. Normal / k map
3. $N \cdot H \times k$ map (function look up)

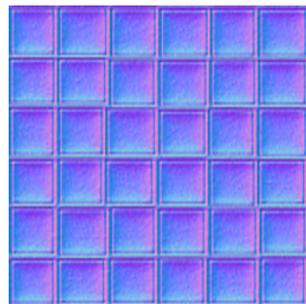
The first two of these maps are shown below. The images on the left are the RGB channels of the maps and the images on the right are the alpha channels. In the first map, we store albedo and gloss for the tile material. The second map stores the x , y and z components of the tangent-space normal in RGB and the specular exponent (k) in alpha. Note that the artist has given each tile in this texture map a different specular exponent to simulate neighboring tiles of disparate material properties. Being able to simply paint the quantity k into a texture map channel is both convenient and empowering to an artist.



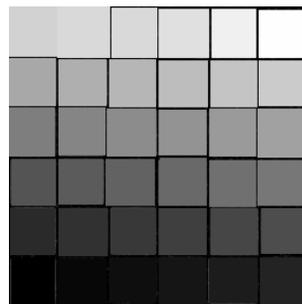
Albedo in RGB



Gloss in alpha

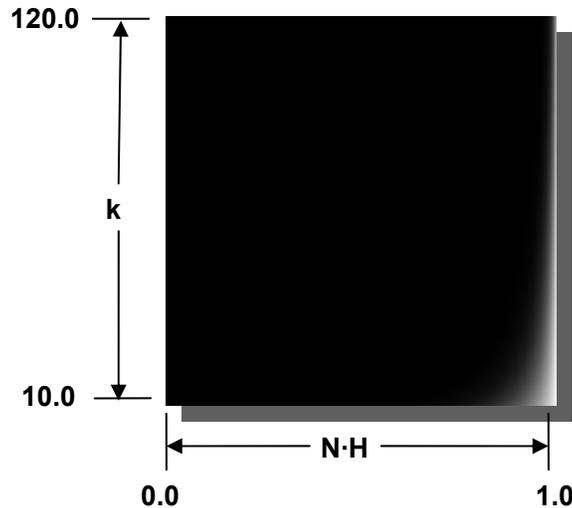


Normals in RGB

 k in alpha

Material maps for per-pixel specular exponent shader

The third texture we will use in this shader is a function lookup which will be used to raise $N \cdot H$ to the k^{th} power via a dependent texture read. Each row of this 2D texture can be thought of as an exponential function which is selected by the alpha channel of the Normal / k map shown above. In this way, we are able to select different specular exponents for different regions within the same texture map. For our purposes, we have found a dynamic range of 10 to 120 is reasonable for k :



Function look-up map for per-pixel specular exponent shader

In the shader code below, we sample the tangent space normal from the first map and dot this quantity with interpolated L and H vectors. $N \cdot H$ is stored in the red channel of $r2$ and the specular exponent is moved into the green channel using write masks. This 2D texture coordinate is then used to access the function look-up map shown above via a dependent read. The instructions in the second phase composite the results into a final color.

```

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

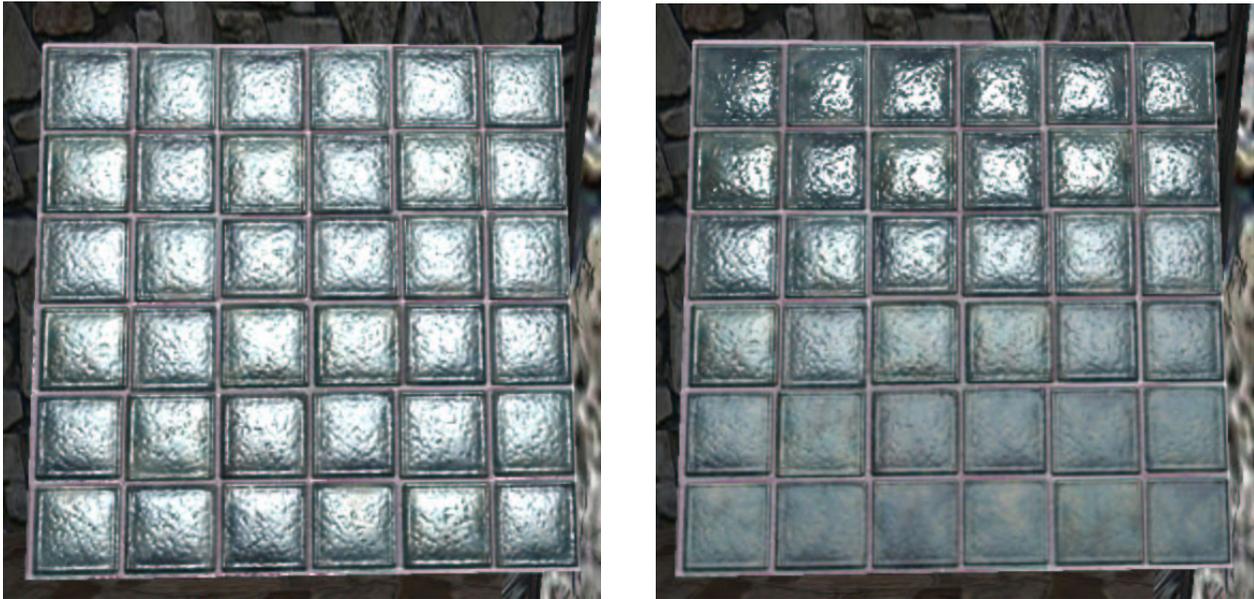
dp3_sat r5.xyz, r1_bx2, r2_bx2 ; N.L
dp3_sat r2.xyz, r1_bx2, r3     ; N.H
mov     r2.y, r1.a          ; K = Specular Exponent

phase
texld  r0, t0           ; Base
texld  r3, r2           ; Specular NHxK map

add     r4.rgb, r5, c7    ; += ambient
mul     r0.rgb, r0, r4    ; base * (ambient + N.L)
+mul_x2 r0.a, r0.a, r3.b  ; Gloss map * specular highlight
add     r0.rgb, r0, r0.a  ; (base*(ambient+N.L)) + (Gloss*Highlight)

```

Output from this shader is shown on the right side of the figure below. The left side shows the result of using the same normal map and a constant specular exponent for the whole object. The image on the right shows how different materials can be represented with the same map by migrating material calculations to the pixel level.



Constant specular power and per-pixel specular power using ps.1.4

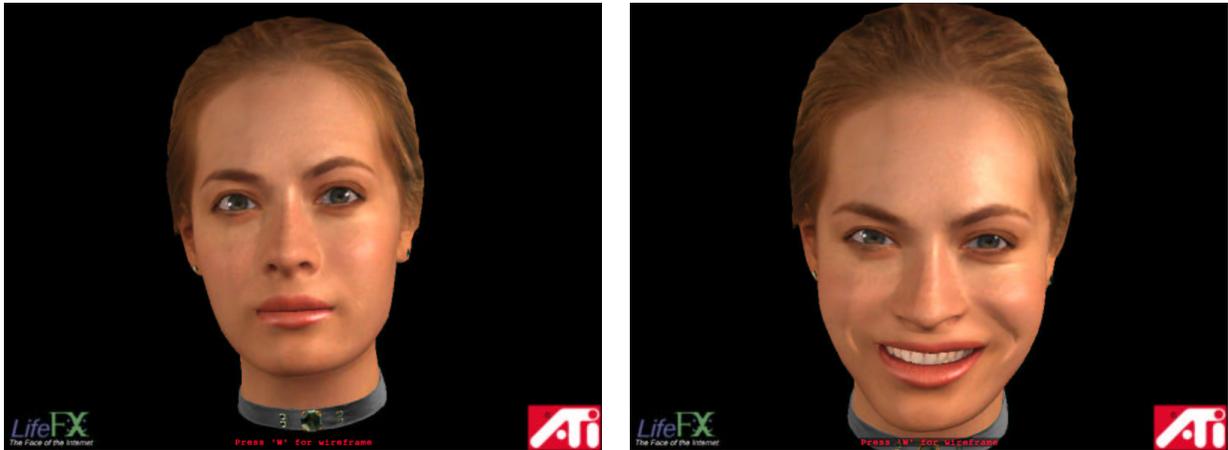
Human Skin

The skin shader used in the Rachel demo uses nearly the maximum number of instructions to implement per-pixel diffuse and specular illumination for two lights. The shader computes the following equation to calculate the lighting per-pixel.

$$I_{RGB} = C_{base}(I_a + I_{d0}(\mathbf{N} \cdot \mathbf{L}_0) + I_{d1}(\mathbf{N} \cdot \mathbf{L}_1)) + gI_s(I_{d0} |\mathbf{N} \cdot \mathbf{H}_0|^k + I_{d1} |\mathbf{N} \cdot \mathbf{H}_1|^k)$$

- where
- C_{base} is the base color sampled from a texture map
 - I_a is the light source ambient coefficient
 - I_{dn} are light source diffuse coefficients
 - I_s is the light source specular coefficient
 - \mathbf{N} is the normal to the surface
 - \mathbf{L} is the light vector
 - \mathbf{H} is the halfway vector
 - g is the gloss factor
 - k is the specular exponent.

A per-pixel variable specular exponent similar to the preceding example is used in this shader but is further improved by using a dependent projective texture fetch as a way to perform a division [Vlachos02].



```

ps.1.4
texld r0, t0
texcrd r1.xyz, t3
texcrd r2.xyz, t5
dp3_sat r4.r, r0_bx2, r1
dp3_sat r4.b, r1, r1
mul_sat r4.g, r4.b, c0.a
mul r4.r, r4.r, r4.r
dp3_sat r5.r, r0_bx2, r2
dp3_sat r5.b, r2, r2
mul_sat r5.g, r5.b, c0.a
mul r5.r, r5.r, r5.r
phase
texld r0, t0
texld r1, t0
texld r2, t2
texld r3, t4
texld r4, r4_dz
texld r5, r5_dz
dp3_sat r2.r, r2_bx2, r0_bx2
+mul r2.a, r0.a, r4.r
dp3_sat r3.r, r3_bx2, r0_bx2
+mul r3.a, r0.a, r5.r
mul r0.rgb, r2.a, c2
mad_x2 r0.rgb, r3.a, c3, r0
mad r2.rgb, r2.r, c2, c1
mad r2.rgb, r3.r, c3, r2
mul r0.rgb, r0, c4
mad_x2_sat r0.rgb, r2, r1, r0
+mov r0.a, c0.z
// tangent space H0
// tangent space H1
// N·H0
// H0·H0
// c0.a*(H0·H0)
// (N·H0)2
// N·H1
// H1·H1
// c0.a*(H1·H1)
// (N·H1)2
// fetch again to get spec map to use as gloss
// Cbase
// tangent space L0
// tangent space L1
// ((N·H)2 / (H·H))k @= |N·H|k
// ((N·H)2 / (H·H))k @= |N·H|k
// N·L0
// g * |N·H0|k <- Gloss specular highlight 0
// N·L1
// g * |N·H1|k <- Gloss specular highlight 1
// Id0*g*|N·H0|k
// Id0*g*|N·H0|k + Id1*g*|N·H1|k
// Ia + Id0*(N·L)
// Ia + Id0*(N·L) + Id1*(N·L)
// Is * (Id0*g*|N·H0|k + Id1*g*|N·H1|k)
// Cbase * (Ia + Id0*(N·L) + Id1*(N·L))
// + Id0*g*|N·H0|k + Id1*g*|N·H1|k

```

Conclusion

We've outlined the behavior of the 1.4 pixel shading model which is available in DirectX 8.1 and is implemented by the ATI RADEON™ 8500. Three key examples have been presented to illustrate the properties of this programming model and the effects that can be achieved today on commodity graphics hardware.

References

[Praun01] Emil Praun, Hugues Hoppe, Matthew Webb and Adam Finkelstein, "Real-Time Hatching." *Proceedings of SIGGRAPH 2001*, pages 579-584.

[Vlachos02] Alex Vlachos, John Isidoro and Christopher Oat, "Textures as Lookup Tables for Per-Pixel Lighting Computations," in *Game Programming Gems 3*, Dante Treglia editor, 2002

