Advanced Vertex and Pixel Shader Techniques

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

- **Vertex Shaders**
  - Quick Review
  - Vertex Local coordinate system
- **Pixel Shaders**
  - Unified Instruction set
  - Flexible dependent texture read
- **Basic examples**
  - Image Processing
  - 3D volume visualizations
- **Gallery of Advanced Shaders**
  - Per-pixel lighting
  - Per-pixel specular exponent
  - Bumpy Environment mapping
  - Per-pixel anisotropic lighting
  - Per-pixel fresnel
  - Reflection and Refraction
  - Multi-light shaders
  - Skin
  - Roiling Clouds
  - Iridescent materials
  - Rolling ocean waves
- **Tools**
  - ATILLA
  - ShadeLab
What about OpenGL?

- For this talk, we’ll use Direct3D terminology to remain internally consistent. But we still love OpenGL.
- ATI has led development of a multi-vendor extension called EXT_vertex_shader.
- Pixel shading operations of the RADEON™ 8500 are exposed via the ATI_fragment_shader extension.
- Refer to the ATI Developer Relations website for more details on these extensions.
What I assume

- You know that a vertex shader is a small program which processes vertex streams and executes on chip.
- You know that this temporarily replaces the fixed function API but that the fixed function API can be used if you want to use it.
- If you choose to write a vertex shader, you choose to implement all the processing you want (i.e. fixed function texgen or lighting are not in play).
- You know that pixel shaders are handled similarly, but execute at the pixel level.
- You’ve heard of a tangent vector.
- You have some idea of what a “dependent read” is.
Vertex Shader In’s and Out’s

- Inputs are vertex streams
- Several read-write temp registers
- Large constant store
- Output registers
  - Position, Fog, Point Size
  - Two Colors
  - Eight sets of tex coords
- Followed by clipping and triangle set up
- Vertex Shader outputs are available as Pixel Shader inputs (except for oPts)

**Note:** There are two oDn output registers and eight oTn output registers.
Pixel Shader In’s and Out’s

- Inputs are texture coordinates, constants, diffuse and specular
- Several read-write temps
- Output color and alpha in r0.rgb and r0.a
- Output depth is in r5.r if you use texdepth (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
Vertex Shader Registers

- $v_n$ – Vertex Components
  - 16 vectors read-only
- $a_n$ – Address register
  - 1 scalar
  - Write/use only
- $c[n]$ – Constants
  - At least 96 4D vectors
  - Read-only
- $r_n$ – Temp Registers
  - 12 4D vectors
  - Read/write
Simple Vertex Shader

Version ➔ vs.1.1

WVP in c0..c3 ➔

; Transform to clip space
dp4 oPos.x, v0, c0
dp4 oPos.y, v0, c1
dp4 oPos.z, v0, c2
dp4 oPos.w, v0, c3

; Write out a color
mov oD0, c4
Vertex Local Coordinate System

• Coordinate system at point on a surface
• Fundamental to advanced pixel shading, as you must put quantities like light directions and per-pixel surface normals into same coordinate frame to do any meaningful math.
• Basis of this space is 3 3D vectors: the object-space normal, tangent and their cross product
  • Taken as a matrix, these vectors represent a transformation (usually just a rotation) from object space to tangent space
  • The transpose is the inverse transform
• Tangent vector typically points along isoparametric line for a given set of 2D texture coordinates
Vertex Local Coordinate System

- For a Longitude-Latitude mapping, think of the normal vector as the local “Up,” the tangent vector as the local “East” and the binormal as the local “North.”
- Take a step back and you’ll see that “Up” varies depending upon where you are on the globe.
- For a given latitude, the binormal (North) is the same.
- The same is true of the tangent (East) along a line of longitude.
Using Vertex Local Coordinates

- Must provide tangent vector in your dataset along with vertex positions, normals etc.
- Can generate binormal in the vertex shader so that you don’t have to pass it in and waste valuable memory bandwidth.
- Possibly store the “sense” of the binormal for generality.
- Can transform L or H vectors to tangent space and pass them down to the raster stage.
- Can also pass whole 3x3 matrix down to the raster stage to allow the pixel shader to transform from tangent space to object space etc.
- Will show several examples in a few minutes.
Skinning and Deformation

• Vertex Local Coordinate System (aka Tangent Space) must remain consistent with transformations
  • Vertex Blending / Skinning
  • Procedural deformation
    • Driving sine waves across surface
    • Will show examples of this …
Pixel Shader In’s and Out’s

- Inputs are texture coordinates, constants, diffuse and specular
- Several read-write temps
- Output color and alpha in r0.rgb and r0.a
- Output depth is in r5.r if you use texdepth (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
Pixel Shader 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
  ```
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
- More instructions
- High Precision
- Range of at least -8 to +8
- Well along the road to DX9
1.4 Pixel Shader Structure

- **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

---

```plaintext
Texture Register File

```
t0

```
t1

```
t2

```
t3

```
t4

```
t5

```
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```
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 chips that don’t have clip planes
- **texdepth**
  - Substitute value for this pixel’s z!
texkill

- Another way to kill pixels
- If you’re just doing a clip plane, use a clip plane
  - As a fallback, use texkill for chips that don’t support user clip planes
- Pixels are killed based on the sign of the components of registers
• Substitute a register value for z
• Imaged based rendering
• Depth sprites
1.4 Pixel Shader ALU Instructions

- `add d, s0, s1` // sum
- `sub d, s0, s1` // difference
- `mul d, s0, s1` // modulate
- `mad d, s0, s1, s2` // \( s0 \times s1 + s2 \)
- `lrip d, s0, s1, s2` // \( s2 + s0 \times (s1-s2) \)
- `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 similar to `texbem`
Argument Modifiers

- **Negate** \(-r_n\)
- **Invert** \(1-r_n\)
  - Unsigned value in source is required
- **Bias (\_bias)**
  - Shifts value down by \(\frac{1}{2}\)
- **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 \texttt{D3DTOP\_DOTPRODUCT3} in \texttt{SetTSS()}.
- **Channel replication**
  - \(r_n.r, r_n.g, r_n.b\) or \(r_n.a\)
  - Useful for extracting scalars out of registers
  - Not just in alpha instructions like the .b in ps.1.2
Instruction Modifiers

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

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

• Any channels of the destination register may be masked during the write of the result
• Useful for computing different components of a texture coordinate for a dependent read
• Example:
  dp3 r0.r, t0, t4
  mov r0.g, t0.a
• We’ll show more examples of this
Range and Precision

- ps.1.4 range is at least -8 to +8
  - Determine with MaxPixelShaderValue
- Pay attention to precision when doing operations that may cause errors to build up
- Conversely, use your range when you need it rather than scale down and lose precision. Intermediate results of filter kernel computations are one case.
- Your texture coordinate interpolators are your high precision data sources. Use them.
- Sampling an 8 bit per channel texture (to normalize a vector, for example) gives you back a low precision result
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 illustrated in Alex Vlachos’s talk, following this one in the Waterfall demo.
- Syntax looks like this:
  texld r3, r3_dz or
  texld r3, r3_dw
- Useful for projective textures or just doing a divide.
- Used in the Rachel demo later on…
Examples: Image Filters

• Use on 2D images in general
• Use as post processing pass over 3D scenes rendered into textures
  • Luminance filter for Black and White effect
    • The film *Thirteen Days* does a crossfade to black and white with this technique several times for dramatic effect
  • Edge filters for non-photorealistic rendering
  • Glare filters for soft look (see *Fiat Lux* by Debevec)
  • Opportunity for you to customize your look
• Rendering to textures is fundamental. You need to get over your reluctance to render into textures.
  • Becomes especially interesting when we get to high dynamic range
Luminance Filter

- Different RGB recipes give different looks
  - Black and White TV (*Pleasantville*)
  - Black and White film (*Thirteen Days*)
  - Sepia
  - Run through arbitrary transfer function using a dependent read for “heat signature”
- A common recipe is \( \text{Lum} = 0.3r + 0.59g + 0.11b \)

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

Original Image

Luminance Image
Multitap Filters

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

- Roberts Cross Gradient Filters

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

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```

```
...  
```
Gradient Filter

Original Image

8 x Gradient Magnitude
ps.1.4
def c0, 0.2f, 0.2f, 0.2f, 1.0f
texld r0, t0 // Center Tap
texld r1, t1 // Down & Right
texld r2, t2 // Down & Left
texld r3, t3 // Up & Left
texld r4, t4 // Up & Right
add r0, r0, r1
add r2, r2, r3
add r0, r0, r2
add r0, r0, r4
mul r0, r0, c0
Five Tap Blur Filter

Original Image

Blurred Image
Sepia Transfer Function

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

1D Luminance to Sepia map
Sepia Transfer Function

Original Image

Sepia Tone Image
Heat Signature

1D Heat Signature Map
Heat Transfer Function

Heat input image

Heat Signature Image

Transfer Function
Volume Visualization

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

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

- With 1.4 pixel shaders, it is very natural to sample data from a 3D texture map and apply a transfer function via a dependent read.

- Transfer functions are usually 1D and are very cheap to update interactively.
Dynamic Transfer Functions

Scalar Data

Transfer Function Applied
Per-pixel N·L and Attenuation
Per-pixel N·L and Attenuation

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

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

phase

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

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

Constant specular power

Variable specular power
Variable Specular Power

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

- Base map with albedo in RGB and gloss in alpha
- Normal map with xyz in RGB and \(k\) in alpha
- \(N \cdot H \times k\) map
- Should also be able to apply a scale and a bias to the map and in the pixel shader to make better use of the resolution
Maps for per-pixel variation of k shader

- Albedo in RGB
- Gloss in alpha
- Normals in RGB
- k in alpha

N·H × k map

k = 120

k = 10
Variable Specular Power

```
ps.1.4
						
	
texld r1, t0  ; Normal

texld r2, t1  ; Normalized Tangent Space L vector

texcrd r3.rgb, t2  ; Tangent Space Halfangle vector

dp3_sat r5.xyz, r1_bx2, r2_bx2 ; N·L

dp3_sat r2.xyz, r1_bx2, r3 ; N·H

mov r2.y, r1.a  ; K = Specular Exponent

phase

texld r0, t0  ; Base

texld r3, r2  ; Specular NHxK map

add r4.rgb, r5, c7  ; += ambient

mul r0.rgb, r0, r4  ; base * (ambient + N·L)

+mul_x2 r0.a, r0.a, r3.a  ; Gloss map * specular

add r0.rgb, r0, r0.a  ; (base*(ambient + N·L)) +

; (Gloss*Highlight)
```

Dependent Read
Anisotropic lighting

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

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

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

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

- These two dot products can be computed per-pixel with the `texm3x2*` instructions or just two `dp3s` in `ps.1.4`

- Use this 2D tex coord to index into special map to evaluate above functions

- At GDC 2001, we showed this limited to per-pixel tangents in the plane of the polygon

- Here, we orthogonalize the tangents with respect to the per-pixel normal inside the pixel shader
Per-pixel anisotropic lighting

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

Diffuse in RGB

Specular in Alpha

V·T

L·T

-1.0 +1.0

-1.0 +1.0

+1.0 -1.0

+1.0 -1.0
Anisotropic Lighting
Example: Brushed Metal
ps.1.4

def c0, 0.5f, 0.5f, 0.0f, 1.0f
text r0, t0 ; Contains direction of anisotropy in tangent space
texcrd r2.rgb, t1 ; light vector
texcrd r3.rgb, t2 ; view vector
text s4, t0 ; normal map

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

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

; Diffuse Light Term
dp3 Sat r4.rgb, r4_bx2, r2 ; N.L
phase
text r2, r5 ; Anisotropic lighting function lookup
text s3, t0 ; gloss map
mul r4.rgb, r3, r4.b ;basemap * N.L
mad r0.rgb, r3, r2.a, r4 ;+= glossmap * specular
mad r0.rgb, r3, c7, r0 ;+= ambient * basemap
Anisotropic Lighting Example: Human Hair

- Direction of anisotropy map is used to light the hair

Highlights computed in pixel shader
Bumpy Environment Mapping

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

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

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

texld  r0, t0 ; Look up normal map
texld  r1, t4 ; Eye vector through normalizer cube map
texcrd r4.rgb, t1 ; 1st row of environment matrix
texcrd r2.rgb, t2 ; 2nd row of environment matrix
texcrd r3.rgb, t3 ; 3rd row of environment matrix
texcrd r5.rgb, t5 ; World space L (Unit length is light's range)
dp3    r4.r, r4, r0_bx2 ; 1st row of matrix multiply
dp3    r4.g, r2, r0_bx2 ; 2nd row of matrix multiply
dp3    r4.b, r3, r0_bx2 ; 3rd row of matrix multiply
dp3_x2 r3.rgb, r4, r1_bx2 ; 2(N·Eye)
mul    r3.rgb, r4, r3 ; 2N(N·Eye)
dp3    r2.rgb, r4, r4 ; N·N
mad    r2.rgb, -r1_bx2, r2, r3 ; 2N(N·Eye) - Eye(N·N)
phase
texld  r2, r2 ; Sample cubic reflection map
texld  r3, t0 ; Sample base map
texld  r4, r4 ; Sample cubic diffuse map
texld  r5, t0 ; Sample gloss map
mul    r1.rgb, r5, r2 ; Specular = Gloss * Reflection
mad    r0.rgb, r3, r4_x2, r1 ; Base * Diffuse + Specular
Per-Pixel Fresnel

Per-Pixel Diffuse + Per-Pixel Bumped Environment map \times Per-Pixel Fresnel = Result
Reflection and Refraction Shader

Normal used to compute reflection and refraction rays in one pass
Reflection and Refraction

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

- Updated version which takes into account the distance from the object center will be available on our website shortly...
Multi-light Shaders

Four Diffuse Per-Pixel Lights in one Pass
4-light Shader

define saturate(r1, r2, r3)
    return r1 * (r2 / r3)
define mul_x2(r1, r2, r3)
    return r1 * r2 * r3
define phase(r1, r2, r3)
    return r1

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

vs.1.1
// Figure out tween constants
sub r9.x, v0.w, c32.x  // 1-tween  v0.w=1  v0.w is
// used in order to avoid
// 2 const regs error

mov r9.y, c32.x
// Compute the tweened position
mul r2, v0, r9.xxxx
mad r2, v14, r9.yyyy, r2

mul r3, v3, r9.xxxx    // Compute the tweened normal
mad r3, v15, r9.yyyy, r3

mov r9, v1        // Compute fourth weight

// Multiply input position by matrix 0
m4x4 r0, r2, c12
mul r1, r0, r9.x

// Multiply input position by matrix 1 and sum
m4x4 r0, r2, c16
mad r1, r0, r9.y, r1

// Multiply input position by matrix 2 and sum
m4x4 r0, r2, c20
mad r1, r0, r9.z, r1

// Multiply input position by matrix 3 and sum
m4x4 r0, r2, c24
mad r1, r0, r9.w, r1

// Multiply by the projection and pass it along
m4x4 oPos, r1, c8

// Skin the normal (z-axis for tangent space)
m3x3 r0, r3, c12
mul r4, r0, r9.x
m3x3 r0, r3, c16
mad r4, r0, r9.y, r4
m3x3 r0, r3, c20
mad r4, r0, r9.z, r4
m3x3 r0, r3, c24
mad r4, r0, r9.w, r4

// Skin the tangent (x-axis for tangent space)
m3x3 r0, v8, c12
mul r2, r0, r9.x
m3x3 r0, v8, c16
mad r2, r0, r9.y, r2
m3x3 r0, v8, c20
mad r2, r0, r9.z, r2
m3x3 r0, v8, c24
mad r2, r0, r9.w, r2

// Skinned bi-normal (y-axis for tangent space)
mul r4, r3, r4.yzxw, r2.zxyw
mad r3, r4.zxyw, -r2.yzxw, r3

// Compute light vector 0
sub r6, c28, r1
dp3 r11.x, r6, r6
rsq r11.y, r11.x
mul r5, r6, r11.y
// Compute the view vector
sub    r8, c2, r1
dp3    r11.x, r8, r8
rsq    r11.y, r11.x
mul    r7, r8, r11.y

// Transform light vector 0 into tangent space
m3x3   r6, r5, r2

// Transform the view vector into tangent space
m3x3   r8, r7, r2

// Halfway vector L+(0.985*V) (numeric fixup to prevent zero vector)
mad     r9, r8, c3.x, r6

// View vector in diffuse interp
mad     oD0, r8, c0.y, c0.y

// Reflect view vector around normal
dp3    r7.w, r7, r4  // V.N
add    r7.w, r7.w, r7.w  // V.N
mad     r7, r7.w, r4, -r7  // 2N(N.V)−V

// Pass along texture coordinates
mov oT0, v7
mad oT1, r7, c0.y, c0.y  // object space
mov oT2, r6
mul oT3, r9, c0.y  // view vector

// Compute light vector 1
sub    r6, c30, r1
dp3    r11.x, r6, r6
rsq    r11.y, r11.x
mul    r5, r6, r11.y

// Transform light vector 1 into tangent space
m3x3   r6, r5, r2

// Halfway vector L+(0.985*V)
// (numeric fixup to prevent zero vector)
mad     r9, r8, c3.x, r6

// Transform light vector 1 into tangent space
m3x3   r6, r5, r2

mov oT4, r6
mul oT5, r9, c0.y
Rachel Skin Pixel Shader

ps.1.4

texld r0, t0
		// fetch a second time to get spec map to use as gloss map

texld r1, t0
		// base map

texld r2, t2
		// tangent space L0

texld r3, t4
		// tangent space L1

texld r4, r4_dz
		// ((N.H)^2 /(H.H))^k @= |N.H|^k

texld r5, r5_dz
		// ((N.H)^2 /(H.H))^k @= |N.H|^k

dp3_sat r2.r, r2_bx2, r0_bx2
		// (N.L0)
+mul r2.a, r0.a, r4.r
		// f(k) * |N.H0|^k <- Gloss specular highlight 0

dp3_sat r3.r, r3_bx2, r0_bx2
		// (N.L1)
+mul r3.a, r0.a, r5.r
		// f(k) * |N.H1|^k <- Gloss specular highlight 1

mul r0.rgb, r2.a, c2
		// Id0*f(k)*|N.H0|^k

mad_x2 r0.rgb, r3.a, c3, r0
		// Id0*f(k)*|N.H0|^k + Id1*f(k)*|N.H1|^k

mad r2.rgb, r2.r, c2, c1
		// Ia + Id0*(N.L)

mad r2.rgb, r3.r, c3, r2
		// Ia + Id0*(N.L) + Id1*(N.L)

mul r0.rgb, r0, c4
		// spec strength * (Id0*f(k)*|N.H0|^k + Id1*f(k)*|N.H1|^k)

mad_x2_sat r0.rgb, r2, r1, r0
		// base(Ia + Id0*(N.L) + Id1*(N.L))
		// + Id0*f(k)*|N.H0|^k + Id1*f(k)*|N.H1|^k

+mov r0.a, c0.z
Iridescent Shader

Iridescent Materials

From Pixar’s *A Bug’s Life*

1.4 Pixel Shader
Iridescent Pixel Shader

**ps.1.4**

```plaintext
texld r0, t0                        // dot 3 map
texld r1, t0                        // base map
texcrd r2.rgb, t2                   // tan H
texld r4, t1                        // normalize L
texld r5, t3                        // normalize V

dp3_sat     r4.g, r0_bx2, r4_bx2    // (N.L)
dp3_sat     r4.b, r0_bx2, r5_bx2    // (N.V) <- Fresnel Term
mad_sat     r4.r, r4.b, c0.g, c0.g  // compress range of N.V into 0 to 1
dp3_sat     r3.b, r2, r2            // (H.H)
dp3_sat     r3.r, r0_bx2, r2        // (N.H)
mul_d2      r3.g, r3.b, c0.r        // .5*k*((H.H))
mul         r3.r, r3.r, r3.r        // (N.H)^2

phase

texld r1, t0                    // Base map
texld r2, r3_dz                 // Attenuated (n.h)^k map
texld r3, r4                    // Iridescent map 1D tex map lookup
texcrd r4.rgb, r4               // 0.5*N.V+0.5, N.L, N.V

mul_sat    r0.rgb, r2, r3            // N.L * diffuse + ambient
mad         r5, r4.g, c2, c1        // diffamb= basemap * (N.L * diffuse + amb)
mul         r5.rgb, r1, r5            // output= diffamb+specular*specular color
mad_sat     r0.rgb, r0, c3, r5      // fresnel term
```

---

**Dependent Reads**
Roiling Clouds Shader
Cloud Pixel Shader

```
ps.1.4
texcrd r0.rgb, t0
texld r1, t1
texcrd r2.rgb, t2
texld r3, t3

mad r0.rgb, c1, r1_bx2, r0 // distort base map coords // by cloud coords
mad r2.rgb, c3, r3_bx2, r2 // distort base map coords // by cloud coords

phase
texld r0, r0
texld r2, r2

mul_sat r1.rgb, r0_x2, c2 // cloud 0, saturate out color
+mul r1.a, r0.r, c2.a
mul_sat r3.rgb, r2_x2, c4
1rp r0.rgb, r2.r, r3, r1
```
Iridescent Shader with procedural displacement
Inputs to iridescent shader with procedural displacement

- Four different waves (easy due to SIMD)
- Sin displaces position
- Cos displaces normal and tangent

<table>
<thead>
<tr>
<th>v0</th>
<th>Vertex Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>v3</td>
<td>Vertex Normal</td>
</tr>
<tr>
<td>v7</td>
<td>Vertex Texture Data u, v</td>
</tr>
<tr>
<td>v8</td>
<td>Vertex Tangent (v direction)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c0</th>
<th>{ 0.0, 0.5, 1.0, 2.0}</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>{ 4.0, 0.5\pi, \pi, 2\pi}</td>
</tr>
<tr>
<td>c2</td>
<td>{1, -1/3!, 1/5!, -1/7!} //for sin</td>
</tr>
<tr>
<td>c3</td>
<td>{1/2!, -1/4!, 1/6!, -1/8!} //for cos</td>
</tr>
<tr>
<td>c4-7</td>
<td>Composite World-View-Projection Matrix</td>
</tr>
<tr>
<td>c8</td>
<td>ModelSpace Camera Position</td>
</tr>
<tr>
<td>c9</td>
<td>ModelSpace Light Position</td>
</tr>
<tr>
<td>c10</td>
<td>{fixup factor for taylor series imprecision, }</td>
</tr>
<tr>
<td>c11</td>
<td>{waveHeight0, waveHeight1, waveHeight2, waveHeight3}</td>
</tr>
<tr>
<td>c12</td>
<td>{waveOffset0, waveOffset1, waveOffset2, waveOffset3}</td>
</tr>
<tr>
<td>c13</td>
<td>{waveSpeed0, waveSpeed1, waveSpeed2, waveSpeed3}</td>
</tr>
<tr>
<td>c14</td>
<td>{waveDirX0, waveDirX1, waveDirX2, waveDirX3}</td>
</tr>
<tr>
<td>c15</td>
<td>{waveDirY0, waveDirY1, waveDirY2, waveDirY3}</td>
</tr>
<tr>
<td>c16</td>
<td>{time, sin(time)}</td>
</tr>
<tr>
<td>c17</td>
<td>{basetexcoord distortion x0, y0, x1, y1}</td>
</tr>
</tbody>
</table>
Iridescent shader with procedural displacement

```
vs.1.1
mul r0, c14, v7.x  // use tex coords as inputs to sinusoidal warp
mad r0, c15, v7.y, r0  // use tex coords as inputs to sinusoidal warp
mov r1, c16.x  // time...
add r0, r1, c13, r0  // add scaled time to move bumps according to freq
add r0, r0, c12
frc r0.xy, r0  // take frac of all 4 components
frc r1.xy, r0.zwzw
mov r0.zw, r1.xyxy
mul r0, r0, c10.x  // multiply by fixup factor (due to inaccuracy of taylor series)
sub r0, r0, c0.y  // subtract .5
mul r0, r0, c1.w  // mult tex coords by 2π
// coords range from(-π to π)
mul r5, r0, r0  // (wave vec)^2
mul r1, r5, r0  // (wave vec)^3
mul r2, r6, r0  // (wave vec)^4
mul r3, r7, r0  // (wave vec)^5
mul r4, r8, r0  // (wave vec)^6
mul r5, r4, r0  // (wave vec)^7
mul r6, r5, r0  // (wave vec)^8
mad r4, r1, c2.y, r0  // (wave vec) - ((wave vec)^3)/3!
mad r4, r2, c2.z, r4  // + ((wave vec)^5)/5!
mad r4, r3, c2.w, r4  // - ((wave vec)^7)/7!
```
Iridescent shader with procedural displacement cont.

```
dp3 r10.x, r4, r4
rsq r10.y, r10.x
mul r4, r4, r10.y // normalize tangent
mul r3, r4.zyxw, r5.zxyw
mad r3, r4.zxyw, -r5.zyxw, r3 // xprod to find binormal
sub r1, c9, r0 // light vector
sub r2, c8, r0 // view vector
dp3 r10.x, r1, r1 // normalize light vector
rsq r10.y, r10.x
mul r1, r1, r10.y
dp3 r6.x, r1, r3
dp3 r6.y, r1, r4
dp3 r6.z, r1, r5 // put light vector in tangent space
dp3 r10.x, r2, r2
rsq r10.y, r10.x
mul r2, r2, r10.y // normalized view vector
dp3 r7.x, r2, r3
dp3 r7.y, r2, r4
dp3 r7.z, r2, r5 // put view vector in tangent space
mad oD0, r3, c0.y, c0.y
mov oT0, v7 // distorted tex coord 0
mov r0, c16.x
mul r0, c18, r0
frc r0.xy, r0 // frc of incoming time
mov r0, v7
mad r1, r9, c17.xyxy, r0
mov oT1, r1 // distorted tex coord 1
mov oT2, r6 // light vector
mov oT3, r7 // view vector
add r8, r6, r7
mul oT4, r8, c0.y // halfway vector
```
Pixel Shader

```
ps.1.4

texld r0, t0

texcrd r1.rgb, t4 // halfway vector
texld r4, t3 // fetch view vector from normalizer cube map

dp3_sat r2, r0_bx2, r4_bx2 // fresnel term \(N.V\)
mad_sat r2, r2, c0.g, c0.g // compress range of \(N.V\) into 0 to 1

dp3_sat r3.b, r1, r1 // \((H.H)\)
dp3_sat r3.a, r0_bx2, r1 // \((N.H)\)

mul r3.g, r3.b, c0.g // \(k \cdot (H.H)\)
mul r3.r, r3.a, r3.a // \((N.H)^2\)

phase
texcrd r0.rgb, r0 // pass through bump map
texld r1, t0 // base map

texld r2, r2 // fetch rainbow hue map
texld r3, r3_dz // fetch \(N.H^k\) map
texld r4, t2 // fetch light vector

dp3_sat r0, r0_bx2, r4_bx2 // \((N.L)\)
mad r0, r0, c2, c1 // \(I_a+I_d \cdot (N.L)\)
mul r0, r1, r0 // \(K_d(I_a+I_d \cdot (N.L))\)
mul r1, r3, r2 // multiply spec by iridescence \(O(N.V)\)
mad_x2_sat r0, r1, c3, r0 // \(K_d(I_a+I_d \cdot (N.L)) \cdot I_s \cdot O(N.V) \cdot (N.H)^k\)
```
Ocean Water
Ocean Water Vertex Shader Inputs and Constants

- **v0** - Vertex Position
- **v3** - Vertex Normal
- **v7** - Vertex Texture Data u,v
- **v8** - Vertex Tangent (v direction)

- **c0** - { 0.0, 0.5, 1.0, 2.0}
- **c1** - { 4.0, 0.5π, π, 2π}
- **c2** - {1, -1/3!, 1/5!, -1/7!} //for sin
- **c3** - {1/2!, -1/4!, 1/6!, -1/8!} //for cos
- **c4-7** - Composite World-View-Projection Matrix
- **c8** - ModelSpace Camera Position
- **c9** - ModelSpace Light Position
- **c10** - {fixup factor for taylor series imprecision, }
- **c11** - {waveHeight0, waveHeight1, waveHeight2, waveHeight3}
- **c12** - {waveOffset0, waveOffset1, waveOffset2, waveOffset3}
- **c13** - {waveSpeed0, waveSpeed1, waveSpeed2, waveSpeed3}
- **c14** - {waveDirX0, waveDirX1, waveDirX2, waveDirX3}
- **c15** - {waveDirY0, waveDirY1, waveDirY2, waveDirY3}
- **c16** - {time, sin(time)}
- **c17** - {basetexcoord distortion x0, y0, x1, y1}
### Ocean Water Vertex Shader

**vs.1.1**

```glsl
mul r0, c14, v7.x  // use tex coords as inputs
    // use sinusoidal warp
mad r0, c15, v7.y, r0  // use tex coords as inputs
    // to sinusoidal warp
mov r1, c16.x  // time...
    // add scaled time to move bumps according to freq
add r0, r0, c12
frc r0.xy, r0.zwzw
mov r0.zw, r1.xyxy
mul r0, r0, c10.x  // multiply by fixup factor
    // coords range from (-π to π)
sub r0, r0, c0.y  // subtract .5
mul r0, r0, c1.w  // mult tex coords by 2π
mul r5, r0, r0  // (wave vec)^2
mul r1, r5, r0  // (wave vec)^3
mul r6, r1, r0  // (wave vec)^4
mul r2, r6, r0  // (wave vec)^5
mul r7, r2, r0  // (wave vec)^6
mul r3, r7, r0  // (wave vec)^7
mul r8, r3, r0  // (wave vec)^8
mad r4, r1, c2.y, r0  // (wave vec) –
    // ((wave vec)^3)/3!
mad r4, r2, c2.z, r4  // + ((wave vec)^5)/5!
mad r4, r3, c2.w, r4  // – ((wave vec)^7)/7!
mov r0, c0.z  // 1
mad r5, r5, c3.x, r0  // –(wave vec)^2/2!
mad r5, r6, c3.y, r5  // + (wave vec)^4/4!
mad r5, r7, c3.z, r5  // – (wave vec)^6/6!
mad r5, r8, c3.w, r5  // + (wave vec)^8/8!
```

```glsl
dp4 r0, r4, c11  // multiply wave heights by waves
mul r0.xyz, v3, r0  // multiply wave magnitude at this vertex by normal
add r0.xyz, r0, v0  // add to position
mov r0.w, c0.z  // homogenous component
```

```glsl
m4x4 oPos, r0, c4  // ObjSpacePos * WVP
mul r1, r5, c11  // cos * wave height
dp4 r11.x, -r1, c14  // normal x offset
dp4 r11.yzw, -r1, c15  // normal y offset and tangent offset
mov r5, v3
mad r5.xy, r11, c10.y, r5
    // warped normal move nx, ny according to cos*wavedir*wave_height
mov r4, v8
mad r4.z, -r11.x, c10.y, r4.z  // warped T
sub r1, c9, r0  // light vector
sub r2, c8, r0  // view vector
dp3 r10.x, r4, r4  // normalize light vector
rsq r10.y, r10.x
mul r1, r1, r10.y
```

```
mul r3, r4.zyxw, r5.zxyw
mad r3, r4.zxyw, -r5.yzxw, r3
    // xprod to find binormal
sub r1, c9, r0
sub r2, c8, r0
```

```
dp3 r10.x, r1, r1  // normalize light vector
rsq r10.y, r10.x
mul r1, r1, r10.y
```
Ocean Water Vertex Shader

dp3  r6.x, r1, r3
dp3  r6.y, r1, r4
dp3  r6.z, r1, r5          // put light vector in tangent space
dp3  r10.x, r2, r2
rsq  r10.y, r10.x
mul  r2, r2, r10.y         // normalized view vectordp3  r7.x, r2, r3
dp3  r7.y, r2, r4
dp3  r7.z, r2, r5          // put view vector in tangent spacemad  oD0, r3, c0.y, c0.y
mov  r0, c16.x
mul  r0, c18, r0
frc  r0.xy, r0             // frc of incoming time
add  r0, r0, v7            // add scaled time to tex coords
mad  r1, r11, c17.xyxy, r0
mov  oT0, r1              // distorted tex coord 0
mad  r1, r11, c17.zwzw, r0
add  r1, c0.yy, r1         // add 0.5 for offset
mov  oT1, r1              // distorted tex coord 1
mov  oT2, r6
mov  oT3, r7
Pixel Shader

- Minimize per-pixel hit (uses 1D textures)

```glsl
ps.1.4

texld r0, t0
texld r1, t1
texcrt r3.rgb, t2
texcrt r4.rgb, t3

add_d2 r2, r0_bx2, r1_bx2       // average of 2 bump maps
dp3_sat r0, r2, r4               // fresnel term (N.V)
mad  r5.rgb, r0_x2, r2, -r4      // R= 2N(N.V)−V
dp3_sat r1, r3, r5               // (L.R)=1 when reflection vector
                                  // points into sun
phase
texcrt r0.rgb, r0
texld r2, r1
texld r3, r0

mad_x8 r0, 1-r0.r, 1-r0.r, -c0.a // approx fresnel term 1−((N.V))^8
mad_sat r0.rgb, r2, r0, r3      // mul spec by fresnel & add fresnel map
```
Developing Shaders

- Assembly language is fun and all, but it rapidly gets old and unmaintainable
- Higher level language is needed
- Research labs are tackling this (see Stanford SIGGRAPH 2001 paper by Proudfoot et al)
- In the meantime, we can at least use cpp, macros and a simple linker
- The ATI Library and Linking Assembler (ATILLA) allows you to reuse and maintain code, not to mention link with 3rd party routines.
- Presented this at Meltdown. Will release tools and library this fall.
Developing Pixel Shaders

- The good news is that since shaders are a programming language they’re easier to edit and debug on the fly than complex multitexturing setups.
- Use tools like ATI’s ShadeLab to play around with them interactively.
- As Pixel Shaders grow in complexity comparable to today’s vertex shaders, we’ll need to address the language issues like we are at the vertex shader level today. ATILLA will migrate to also supporting pixel shaders in DirectX 9.
ShadeLab

Advanced Vertex and Pixel Shader Techniques
The Road to DirectX 9

- Look for more of the same in vs.2.x
  - Longer programs
  - More constant store
- ps.1.4 is a good preparation for how to think about DX9 pixel shaders
  - Unified instruction set
  - Higher precision
  - Vectors, not colors
  - Flexible dependent texture reads
Summary

- Vertex Shaders
  - Quick Review
  - Vertex Local coordinate system
- Pixel Shaders
  - Unified Instruction set
  - Flexible dependent texture read
- Basic examples
  - Image Processing
  - 3D volume visualizations
- Gallery of Advanced Shaders
  - Per-pixel lighting
  - Per-pixel specular exponent
  - Bumpy Environment mapping
  - Per-pixel anisotropic lighting
  - Per-pixel fresnel
  - Reflection and Refraction
  - Multi-light shaders
  - Skin
  - Roiling Clouds
  - Iridescent materials
  - Rolling ocean waves
- Tools
  - ATILLA
  - ShadeLab
Questions

P.S. Stick around for the next talk to see more of these shaders in action and learn about ATI’s *sushi* graphics engine.