Advanced Pixel Shading Techniques

Jason L. Mitchell
JasonM@ati.com
Outline

• DirectX 8.1 Pixel Shader Architecture (ps.1.4)
  • Inputs and Outputs
  • Unified Instruction set
  • Flexible dependent texture read
  • Projective Dependent Reads

• Gallery of Shaders
  • Image Processing
    • Popular new trend. The “lens flare” of 2002 - 2003?
    • Image-space outlining for NPR
  • Polynomial Texture Maps from HP
  • Refraction
  • Skin
  • Dynamic Fur – Doing physics with the rasterizer!

• Tools from ATI
  • FurGen
  • ShadeLab

• Looking Forward: DX9 ps.2.0
What about OpenGL?

• For this talk, we’ll use Direct3D terminology to remain internally consistent. But we still love OpenGL!

• In fact, ATI is the newest permanent OpenGL Architectural Review Board (ARB) member

• Pixel shading operations of the RADEON™ 8500 are exposed via the ATI_fragment_shader extension.
What is a Pixel Shader?

- A pixel shader is a small program which processes pixels and executes on the Graphics Processing Unit.
- An application programmer writes pixel shaders in a specialized assembly language and downloads them onto the Graphics Processor during rendering.
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 $\alpha$) 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 ps.2.0
### 1.4 Pixel Shader Structure

#### Texture Register File

<table>
<thead>
<tr>
<th>Register</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>texld t4, t5</td>
</tr>
<tr>
<td></td>
<td>dp3 t0.r, t0, t4</td>
</tr>
<tr>
<td></td>
<td>dp3 t0.g, t1, t4</td>
</tr>
<tr>
<td></td>
<td>dp3 t0.b, t2, t4</td>
</tr>
<tr>
<td></td>
<td>dp3_x2 t2.rgb, t0, t3</td>
</tr>
<tr>
<td></td>
<td>mul t2.rgb, t0, t2</td>
</tr>
<tr>
<td></td>
<td>dp3 t1.rgb, t0, t0</td>
</tr>
<tr>
<td></td>
<td>mad t1.rgb, -t3, t1, t2</td>
</tr>
<tr>
<td>t1</td>
<td>phase</td>
</tr>
<tr>
<td></td>
<td>texld t0, t0</td>
</tr>
<tr>
<td></td>
<td>texld t1, t1</td>
</tr>
<tr>
<td></td>
<td>texld t2, t5</td>
</tr>
<tr>
<td>t2</td>
<td>mul t0, t0, t2</td>
</tr>
<tr>
<td></td>
<td>mad t0, t0, t2.a, t1</td>
</tr>
</tbody>
</table>

- **Optional Sampling**
  - Up to 6 textures
- **Address Shader**
  - Up to 8 instructions
- **Optional Sampling**
  - Up to 6 textures
  - Can be dependent reads
- **Color Shader**
  - Up to 8 instructions
1.4 Texture Instructions

Mostly just data routing. Not ALU operations per se

- texld
  - Samples data into a register from a texture
- texcrd
  - Moves high precision signed data into a temp register ($r_n$)
  - Higher precision than v0 or v1
- texkill
  - Kills pixels based on sign of register components
  - Fallback for chips that don’t have clip planes
- texdepth
  - Substitute value for this pixel’s z!
1.4 Pixel Shader ALU Instructions

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

- **Negate** \(-r_n\)
- **Invert** \(1-r_n\)
  - Unsigned value in source is required
- **Bias** (_bias)
  - Shifts value down by \(\frac{1}{2}\)
- **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 `D3DTOP_DOTPRODUCT3` in `SetTSS()`
- **Channel replication**
  - \(r_n.r, r_n.g, r_n.b\) or \(r_n.a\)
  - Useful for extracting scalars out of registers
  - Not just in alpha instructions like the .b in ps.1.2
Instruction Modifiers

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

\_sat may be used alone or combined with one of the other modifiers. i.e. mad\_d8\_sat
Write Masks

• Any channels of the destination register may be masked during the write of the result

• Useful for computing different components of a texture coordinate for a dependent read

• Example:
  
  \text{dp3 } r0.r, t0, t4
  
  \text{mov } r0.g, t0.a
Projective Textures

• You can do texture projection on any texld instruction.

• This includes projective dependent reads, which are fundamental to doing reflection and refraction mapping of things like water surfaces. This is used in the Nature and Rachel demos.

• Syntax looks like this:
  
  texld r3, r3_dz or
  texld r3, r3_dw

• Useful for projective textures like the refraction map in the nature demo or just doing a divide.
Frame Post Processing: Image Filters in Pixel Shaders

- Use on 2D images in general
- Use as post processing pass over 3D scenes
  - Opportunity for you to customize your look
  - 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, *ICO* on PS2, Halo on XBox)
  - Refraction Mapping (see *Jak and Daxter* on PS2)
  - Check out the XBox game *Wreckless: The Yakuza Missions* for some extreme examples of 3D scene post-processing
- Rendering to textures is fundamental
- Becomes especially interesting when we get to high dynamic range (tone mapping)
- See Dan Baker’s notes from the DX Dev Day
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 \( \text{Lum} = .3r + .59g + .11b \)

```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
Sepia Transfer Function

```
ps.1.4
def c0, 0.30f, 0.59f, 0.11f, 1.0f
texld r0, t0
dp3 r0, r0, c0  // Convert to Luminance
phase
texld r5, r0    // Dependent read from 1D Sepia map
mov r0, r5
```

1D Luminance to Sepia map
Sepia Transfer Function

Original Image  Sepia Tone 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
Roberts Cross Gradient Filters

```
p.s.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
```
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
• Outlines of objects are an important element of Non Photorealistic Rendering (NPR)
• Geometric approaches require some access to the model geometry and don’t necessarily scale well as a result. *Jet Set Radio Future*, for example, appears to use a geometric approach to outlining and you can see how low-poly their characters are.
• Image space approaches scale better and work well with higher-order surfaces
Image Space Outlining for NPR

- World Space Normals
- Eye Space Depth
- Edge Detect → Outlines
- Dilate → Thicker Outlines

Advanced Pixel Shading Techniques
Composite Outlines over Shaded Scene
Composite Outlines over Shaded Scene
Variable Specular Power

Constant specular power vs. 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
Maps for per-pixel variation of $k$ shader

N·H × $k$ map

Albedo in RGB

Gloss in alpha

Normals in RGB

$k = 120$

$k = 10$
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 NH×K map
add r4.rgb, r5, c7 ; += ambient
mul r0.rgb, r0, r4 ; base * (ambient + N·L))
+mul_x2 r0.a, r0.a, r3.a ; Gloss map * specular
add r0.rgb, r0, r0.a ; (base*(ambient + N·L)) +

; (Gloss*Highlight)
```
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

```assembly
	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 × Per-Pixel Fresnel = Result
Polynomial Texture Maps

- Published at SIGGRAPH 2001
- Images of surface are acquired from one position using various lighting directions
- Can be applied to virtual surfaces using the same tools.

PTM algorithms provided courtesy of Hewlett-Packard. HP retains all rights to the algorithms and code.
Polynomial Texture Maps

- \[ L(u,v;l_u,l_v) = a_0(u,v)l_u^2 + a_1(u,v)l_v^2 + a_2(u,v)l_u l_v + a_3(u,v)l_u + a_4(u,v)l_v + a_5(u,v) \]
  where \((l_u, l_v)\) are projections of the normalized light vector into the local texture coordinate system \((u,v)\) and \(L\) is the resultant surface luminance at that coordinate.

- \(a_0\) to \(a_5\) are fit to the (real or virtual) photographic data and are stored in the PTM

PTM algorithms provided courtesy of Hewlett-Packard. HP retains all rights to the algorithms and code.
Polynomial Texture Maps

- Accurate filtering
  - Unlike normal maps
- Self-shadowing

PTM algorithms provided courtesy of Hewlett-Packard. HP retains all rights to the algorithms and code.
RefRACTIVE STAINED GLASS

Normal used to compute refraction rays
Rachel
Rachel Skin Pixel Shader

ps.1.4

texld r0, t0

texcrd r1.xyz, t3  // tangent space H0

texcrd r2.xyz, t5  // tangent space H1

dp3_sat r4.r, r0_bx2, r1  // (N.H0)

dp3_sat r4.b, r1, r1  // (H0.H0)

mul_sat r4.g, r4.b, c0.a  // c0.a*(H0.H0)

mul r4.r, r4.r, r4.r  // (N.H0)^2

dp3_sat r5.r, r0_bx2, r2  // (N.H1)

dp3_sat r5.b, r2, r2  // (H1.H1)

mul_sat r5.g, r5.b, c0.a  // c0.a*(H1.H1)

mul r5.r, r5.r, r5.r  // (N.H1)^2

phase

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)

+m ul 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)

+m ul 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))

+mov r0.a, c0.z
Using Pixel Shaders to Perform Physics during Fur Rendering

- Maps normal and force textures onto object
- Render to/from these textures to perform physics using pixel shader
ATI RADEON™ 8500 Fur Demo by Tomohide Kano

• Models effect of gravity and inertia on fur using math done in a pixel shader
• Drawn entirely with “shells”
• OpenGL Demo, with source code, available on ATI Developer Relations Website: www.ati.com/developer

Copyright © 2002, Tomohide Kano
Tools from ATI

- **ShadeLab**
  - Pixel shader editor
  - Quickly experiment with ideas and check syntax
- **FurGen**
  - Fur rendering tool with a wide variety of customization parameters
  - Dynamically generates textures necessary for rendering fur according to user settings
ShadeLab

- Pixel shader editor
- Quickly experiment with and debug shaders
- Check syntax
- Choose from a variety of texture coordinate options
FurGen Fur Generation Utility

- User can tweak fur parameters such as length, curliness, color etc.
- Renders shells and fins
- Tangent map specifies tangent direction for anisotropic lighting
- Uses 1.4 Pixel Shaders for rendering
FurGen Fur Generation Utility

- Tangent Map
- Density Channel
- Bald-spots with albedo alpha
The Road to ps.2.0

- ps.1.4 is a good preparation for how to think about ps.2.0 pixel shaders
  - Unified instruction set
  - Floating point pixel pipeline
    - Think vectors, not colors
    - rcp, rsq etc
  - 16 textures
  - 64 ALU ops, 32 texture ops
  - Flexible dependent texture reads
    - Up to four levels of dependency
Summary

• DirectX 8.1 Pixel Shader Architecture (ps.1.4)
  • Inputs and Outputs
  • Unified Instruction set
  • Flexible dependent texture read
  • Projective Dependent Reads

• Gallery of Shaders
  • Image Processing
    • Popular new trend. The “lens flare” of 2002 - 2003?
    • Image-space outlining for NPR
  • Polynomial Texture Maps from HP
  • Refraction
  • Skin
  • Dynamic Fur – Doing physics with the rasterizer!

• Tools from ATI
  • ShadeLab
  • FurGen

• Looking Forward: DX9 ps.2.0
References

• DirectX 8.1 SDK
• ATI DevRel Website www.ati.com/developer
• Alex Vlachos - Designing a Game’s Shader Library for Current & Next Generation Hardware
  • Today at 4pm
• Arcot Preetham  Nathaniel Hoffman - Rendering Outdoor Light Scattering in Real-Time
  • Today at 4pm
• Come by the booth!
Questions