High Level Shading with DirectX® 9 on RADEON 9700

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

- What’s new this hardware generation?
- General DirectX® 9 High Level Shading Language (HLSL) details
- D3DX Effects
- Sample applications (available on web)
  - High Dynamic Range Rendering
  - Depth of Field
  - Procedural Wood Effect
- Optimization Tips
What’s new this generation?

• Longer shaders at both vertex and pixel level
• Vertex Shading Flow Control
• Floating Point Pixels
• High precision throughout chip
• DirectX® 9 Standard High Level Shading Language
Why High Level Shading?

• All the usual advantages of high level languages
  – Faster, easier development
  – Code re-use
    • Within or between teams/companies
  – Optimization
• Industry standard which will run on cards from any vendor
What does a high level shader look like?

```c
float4 hls1_bluemarble (float3 P : TEXCOORD0,
                        float3 Peye : TEXCOORD1,
                        float3 Neye : TEXCOORD2) : COLOR
{
    float4 Ct, Ci;
    float3 NNeye;
    float f;
    float marble = -2.0f * snoise(P * psConst2.w) + 0.75f;

    NNeye = normalize(Neye);

    // Color spline lookup texture (gloss in alpha)
    Ct = tex1D (MarbleSplineSampler, marble);

    // Color from illumination
    Ci = Ct * (Ka * ambient() + Kd * soft_diffuse(NNeye, Peye)) +
        Ct.w * Ks * specular(NNeye, Peye, roughness);

    return Ci;
}
```
Declaring Samplers

// Declare texture
TEXTURE tMarbleSpline;

// Declare sampler
sampler MarbleSplineSampler = sampler_state {
    Texture = (tMarbleSpline);
    MinFilter = Linear;
    MagFilter = Linear;
    MipFilter = Linear;
    AddressU = Clamp;
    AddressV = Clamp;
    MaxAnisotropy = 16;
};
Functions

• No surprises here...

```c
float3 snoise (float3 x)
{
    return 2.0f * tex3D (NoiseSampler, x) - 1.0f;
}

float4 soft_diffuse(float3 Neye, float3 Peye)
{
    // Compute normalized vector from vertex to Leye
    float3 Leye = (psConst4 - Peye) / len(psConst4 - Peye);
    float NdotL = dot(Neye, Leye) * 0.5f + 0.5f;

    // N.L
    return float4 (NdotL, NdotL, NdotL, NdotL);
}
```
HLDSL Development Process

- Write shader in HLDSL
  - Standalone or using D3DX FX files
- Compile for specific target
  - ps_1_0, ps_1_1, ps_1_2, ps_1_3, ps_1_4 or ps_2_0
  - vs_1_1 or vs_2_0
  - Essentially cross compiling for the asm of one of these virtual machines
  - Get back code and constant block
  - Can manually manage constant block or let D3DX handle it
D3DX Effects

• Issue “compile” where technique is declared:

```c
technique technique_hlsl_wood
{
    pass P0
    {
        VertexShaderConstant[0] = <matWorldViewProj>;
        VertexShaderConstant[4] = <matWorldView>;
        VertexShaderConstant[8] = <matITWorldView>;
        VertexShaderConstant[12] = <matWorld>;
        VertexShaderConstant[17] = <matTex0>;
        VertexShaderConstant[21] = <matTex1>;
        VertexShaderConstant[25] = <matTex2>;
        VertexShaderConstant[29] = <vWoodVertexConstants>;

        VertexShader = <asm_wood_vs>;
        PixelShader = compile ps_2_0 hlsl_wood();

        CullMode = CCW;
    }
}
```
Constant and Sampler State

- All the usual D3DX mechanisms such as setting constants just work as before
  – Illustrated in the Procedural Wood sample
- Can declare a constant to map to a particular hardware register to minimize constant state setting:
  
  \[
  \text{matrix worldViewProj : register(c0);} \\
  \]

- Same for samplers:
  
  \[
  \text{sampler noiseSampler : register(s0);} \\
  \]
Some Applications using HLSL

- High Dynamic Range post-processing
  - Gaussian Blur
  - Vignetting and Tone Mapping
- Image Space Depth of Field
- Procedural Wood

- Sample code available at www.ati.com/developer
High Level Shading with DirectX® 9 on RADEON 9700

High Dynamic Range Rendering
HDR Rendering Process

Scene
Geometry
lit with
HDR Light
Probes

Image Space Operations

HDR
Scene

Bloom
Filter

Tone
Map

Displayable
Image
Frame Postprocessing

HDR Scene

One Pass Each

¼ Size Frame

Vertical Gaussian Filter

Horizontal Gaussian Filter

One Final Pass

Tone Map

High Level Shading with DirectX® 9 on RADEON 9700
Separable Gaussian Filter

- Some filters, such as a 2D Gaussian, are separable and can be implemented as successive passes of 1D filter.
- We will do this by rendering into temporary buffer, sampling a line or column of texels on each of two passes.
- One **center** tap, six **inner** taps and six **outer** taps.
- Sample 25 texels in a row or column using a layout as shown below:
Separable Gaussian Blur Part 1

```c4d
float4 hlsl_gaussian_x (float2 tapZero : TEXCOORD0, float2 tap12 : TEXCOORD1,
float3 tapMinus12 : TEXCOORD2, float2 tap34 : TEXCOORD3,
float2 tapMinus34 : TEXCOORD4, float3 tap56 : TEXCOORD5,
float3 tapMinus56 : TEXCOORD6 ) : COLOR
{
    float4 accum, Color[NUM_INNER_TAPS];
    Color[0] = tex2D (nearestImageSampler, tapZero); // sample 0
    Color[1] = tex2D (linearImageSampler,  tap12);    // samples 1, 2
    Color[2] = tex2D (linearImageSampler,  tapMinus12); // samples -1, -2
    Color[3] = tex2D (linearImageSampler,  tap34);    // samples 3, 4
    Color[4] = tex2D (linearImageSampler,  tapMinus34); // samples -3, -4
    Color[5] = tex2D (linearImageSampler,  tap56);    // samples 5, 6
    Color[6] = tex2D (linearImageSampler,  tapMinus56); // samples -5, -6

    accum = Color[0] * gTexelWeight[0];  // Weighted sum of samples
    accum += Color[1] * gTexelWeight[1];
    accum += Color[2] * gTexelWeight[1];
    accum += Color[3] * gTexelWeight[2];
    accum += Color[4] * gTexelWeight[2];
    accum += Color[5] * gTexelWeight[3];
    accum += Color[6] * gTexelWeight[3];

    ...
```
Separable Gaussian Blur Part 2

...  

```c
float2 outerTaps[NUM_OUTER_TAPS];
outerTaps[0] = tapZero * gTexelOffset[0]; // coord for samples  7,  8
outerTaps[1] = tapZero * -gTexelOffset[0]; // coord for samples -7, -8
outerTaps[2] = tapZero * gTexelOffset[1]; // coord for samples  9, 10
outerTaps[3] = tapZero * -gTexelOffset[1]; // coord for samples -9, -10
outerTaps[4] = tapZero * gTexelOffset[2]; // coord for samples  11, 12
outerTaps[5] = tapZero * -gTexelOffset[2]; // coord for samples -11, -12

// Sample the outer taps
for (int i=0; i<NUM_OUTER_TAPS; i++)
{
    Color[i] = tex2D (linearImageSampler, outerTaps[i]);
}

accum += Color[0] * gTexelWeight[4]; // Accumulate outer taps
accum += Color[1] * gTexelWeight[4];
accum += Color[2] * gTexelWeight[5];
accum += Color[3] * gTexelWeight[5];
accum += Color[4] * gTexelWeight[6];
accum += Color[5] * gTexelWeight[6];

return accum;
```
Tone Mapping

- Underexposed
- Very Underexposed
- Overexposed
- Good exposure
Tone Mapping Shader

```hlsl
float4 hls1_tone_map (float2 tc : TEXCOORD0) : COLOR
{
    float fExposureLevel = 32.0f;

    float4 original = tex2D (originalImageSampler, tc);
    float4 blur = tex2D (blurImageSampler, tc);

    float4 color = lerp (original, blur, 0.4f);

    tc -= 0.5f; // Put coords in -1/2 to 1/2 range

    // Square of distance from origin (center of screen)
    float vignette = 1 - dot(tc, tc);

    // Multiply by vignette to the fourth
    color = color * vignette * vignette * vignette * vignette;

    color *= fExposureLevel; // Apply simple exposure level
    return pow (color, 0.55f); // Apply gamma and return
}
```
Depth Of Field

ATI Bacteria Screensaver
Depth Of Field

- Important part of photo-realistic rendering
- Computer graphics uses a pinhole camera model
- Real cameras use lenses with finite dimensions
- See Potmesil and Chakravarty 1981 for a good discussion
Camera Models

- Pinhole lens lets only a single ray through
- In thin lens model if image plane isn’t in focal plane, multiple rays contribute to the image
- Intersection of rays with image plane approximated by circle
Real-time Depth Of Field Implementation On Radeon 9700

• Use MRT to output multiple data – color, depth and “blurriness” for DOF post-processing

• Use pixel shaders for post-processing
  – Use post-processing to blur the image
  – Use variable size filter kernel to approximate circle of confusion
  – Take measures to prevent sharp foreground objects from “leaking” onto background
Depth Of Field Using MRT

Pixel Pipeline Output

- Depth and “blurriness” in 16-bit FP format
- Blurriness computed as function of distance from focal plane
Circle Of Confusion Filter Kernel

- Vary kernel size based on the "blurriness" factor.
Elimination Of “Leaking”

• Conventional post-processing blur techniques cause “leaking” of sharp foreground objects onto blurry backgrounds

• Depth compare the samples and discard ones that can contribute to background “leaking”
Semantic Depth Of Field

- Semantic depth of field – sharpness of objects controlled by “relevance”, not just depth
- Easy to accommodate with our technique
  – “Blurriness” is separate from depth
- Can be used in game menus or creatively in real-time cinematics to focus on relevant scene elements
Depth Of Field Shader

```csh
float4 hlsl_depth_of_field_loop (float2 centerTap : TEXCOORD0) : COLOR
{
    float2 tap[NUM_DOF_TAPS];
    float4 Color[NUM_DOF_TAPS];
    float2 Depth[NUM_DOF_TAPS];

    // Fetch center samples from depth and focus maps
    float4 CenterColor = tex2D (ColorSampler, centerTap);
    float2 CenterFocus = tex2D (DoFSampler, centerTap);
    float fTotalContribution = 1.0f;
    float fContribution;
    float fCoCSize = CenterFocus.y * gMaxCoC; // Scale the Circle of Confusion

    for (int i=0; i<NUM_DOF_TAPS; i++) // Run through all of the taps
    {
        // Compute tap locations relative to center tap
        tap[i] = fCoCSize * gTapOffset[i] + centerTap;
        Color[i] = tex2D (ColorSampler, tap[i]);
        Depth[i] = tex2D (DoFSampler, tap[i]);

        // Compute tap's contribution to final color
        fContribution = (Depth[i].x > CenterFocus.x) ? CenterFocus.y : Depth[i].y;
        CenterColor += fContribution * Color[i];
        fTotalContribution += fContribution;
    }

    float4 FinalColor = CenterColor / fTotalContribution; // Normalize
    return FinalColor;
}
```
Procedural Wood

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

- Shader Space ($P_{shade}$)
- Distance from trunk axis ($z$)
- Run through pulse train
- Add noise to $P_{shade}$
P \text{ shade}

- For this app, \( P_{\text{shade}} \) is just world space.
- The infinite virtual log runs along the z axis.
- I make a few different transformed versions of \( P_{\text{shade}} \) in the vertex shader in order to turn scalar noise into color noise.
Pulse Train

- $\sqrt{P_{\text{shade}}.x^2 + P_{\text{shade}}.y^2} \times \text{freq}$
- Pass this in to pulse train
- Tuned to mimic the way colors mix in real wood
- One pulse stored in 1D texture which tiles:
hlsl_rings()

- Procedural concentric rings in DirectX® 9 HLSL

```cpp
float4 hlsl_rings (float4 Pshade : TEXCOORD0) : COLOR
{
    float scaledDistFromZAxis = sqrt(dot(Pshade.xy, Pshade.xy))*psConst2.w;
    float4 blendFactor = tex1D (PulseTrainSampler, scaledDistFromZAxis);
    return psConst2 * blendFactor.x + psConst3 * (1 - blendFactor.x);
}
```
hlsl_noise() (float3 Pshade0 : TEXCOORD0,
               float3 Pshade1 : TEXCOORD1,
               float3 Pshade2 : TEXCOORD2):COLOR
{
    float4 coloredNoise;

    coloredNoise   = tex3D (NoiseSampler, Pshade0);
    coloredNoise.g = tex3D (NoiseSampler, Pshade1);
    coloredNoise.b = tex3D (NoiseSampler, Pshade2);

    return coloredNoise;
}
hlsl_noisy_rings()

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

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

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

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

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

    float blendFactor = tex1D (PulseTrainSampler, scaledDistFromZAxis);

    return psConst2 * blendFactor.x + psConst3 * (1 - blendFactor.x);
}
```
hlsl_ivory(float3 Peye : TEXCOORD0, float3 Neye : TEXCOORD1) : COLOR
{
    // Compute normalized vector from vertex to light in eye space
    float3 Leye = (psConst4 - Peye) / len(psConst4 - Peye);
    Neye = Neye / len(Neye); // Normalize normal
    float3 Veye = -(Peye / len(Peye)); // Compute Veye
    float3 Heye = (Leye + Veye) / len(Leye + Veye); // Compute halfway vector
    float NdotL = dot(Neye, Leye); // N.L
    float diffuse = NdotL * 0.5f + 0.5f; // "Half-Lambert"
    float NdotH = clamp(dot(Neye, Heye), 0.0f, 1.0f); // N.H
    float NdotH_2 = NdotH * NdotH;
    float NdotH_4 = NdotH_2 * NdotH_2;
    float NdotH_8 = NdotH_4 * NdotH_4;
    float NdotH_16 = NdotH_8 * NdotH_8;
    float NdotH_32 = NdotH_16 * NdotH_16;
    return NdotH_32 * NdotH_32 + diffuse;
}
asm generated for hls1_ivory()

...  
add r7.xyz, -t0, c0  
dp3 r7.w, r7, r7  
rsq r7.w, r7.w  
mul r2.xyz, r7, r7.w  
dp3 r2.w, t0, t0  
rsq r2.w, r2.w  
mad r11.xyz, r2.w, -t0, r2  
dp3 r2.w, r11, r11  
rsq r2.w, r2.w  
mul r6.xyz, r11, r2.w  
dp3 r2.w, t1, t1  
rsq r2.w, r2.w  
mul r1.xyz, r2.w, t1  
dp3_sat r2.w, r1, r6  
mul r1.w, r2.w, r2.w  
mul r2.w, r1.w, r1.w  
mul r1.w, r2.w, r2.w  
mul r2.w, r1.w, r1.w  
mul r1.w, r2.w, r2.w  
dp3 r8.w, r1, r2  
mad r10.w, r8.w, c1.x, c1.x  
mad r5, r1.w, r1.w, r10.w  
mov oC0, r5

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

HLSL generates 23 ALU Instructions!  Handwritten asm is 24 instructions
In all of the cases we’ve tested, the Microsoft High Level Shading Language compiler generates assembly output as good or better than hand-tuned assembly. This assumes well-written HLSL. Use the fxc command line utility to analyze HLSL compiler output.
Optimization Tips

• Use `float4` for variable holding your final result
  – Don’t do `return float4 (o.r, o.g, o.b, 1.0f);` at the end of the shader

• Use `float`, `float3` and `float4` as appropriate
  – Vectorize where you can, but proper use of scalar operations is helpful to compiler and hardware

• Use `tex1D()` where it makes sense
  – There are no 1D textures in Direct3D®, but this can save you an instruction if you know the sampled texture will be $1 \times n$

• Use intrinsics rather than reinvent them
  – Can be helpful to hardware
Summary

• DirectX® 9 High Level Shading Language
• D3DX Effects
• Sample applications (available on web)
  – High Dynamic Range Rendering
  – Depth of Field
  – Procedural Wood Effect
• Optimization Tips
ATI Developer Resources

- For the sample applications used here and for general developer information, visit the ATI Developer Relations website:
  - www.ati.com/developer

- RenderMonkey™