Direct3D® High Level Shader Language Programming using RenderMonkey™ IDE

Natasha Tatarchuk
3D Applications Research Group
ATI Research, Inc.

Game Developer Conference, San Jose, CA, March 7th 2003
Outline

• RenderMonkey™ IDE overview

• Creating HLSL shaders using RenderMonkey™
  – High Level Shading Language
  – HLSL support in RenderMonkey™

• Advanced shaders discussion

• HLSL Tips and Tricks

• Conclusions
RenderMonkey Addresses the Needs of Game Developers

- Shaders are more than just assembly code

- Encapsulating shaders can be complex
  - Cannot deliver shader-based effects using a standard mechanism

- Solve problems for shader development
  - Designing software on emerging hardware is difficult
  - Lack of currently existing tools for full shader development
  - Need a collaboration tool for artists and programmers for shader creation
RenderMonkey Facilitates Efficient Shader Development

- Simplifies shader creation
- Fast prototyping and debugging of new graphics algorithms
- Helps you share graphics effects with other developers and artists
- Easy integration of effects into existing applications
- Quickly create new components using our flexible framework
- A communication tool between artists and developers
- Provides an open platform for ATI and ISV’s to develop and incorporate future shader tools
Program Your Shaders Using Our Intuitive, Convenient IDE

Editor Windows

Artist Editor
RenderMonkey Effect Data Organization

- Encapsulate all effect data in a single text file
- Each *Effect Workspace* consists of:
  - Effect Group(s)
    - Effect(s)
  - Pass(es)
    - Render State
    - Pixel Shader
    - Vertex Shader
    - Geometry
    - Textures
  - Variables and stream mapping nodes
    - Variables can live at any level of the workspace
High Level Shading Language

- Introduced in DirectX® 9.0, improved compiler shipping with DirectX® 9.1

- C-like language

- Supports Pixel and Vertex Shaders
  - Specify which target to compile to
    - vs_1_1 or vs_2_0
    - ps_1_1, ps_1_2, ps_1_3, ps_1_4 or ps_2_0
  - Applies optimizations to generate standard asm output which is passed to the API/DDI
Advantages of Implementing Shaders Using HLSL

- Develop your shaders quicker
- Better code maintainability and re-use
- Optimization
- Industry standard
  - Guaranteed support on cards from any vendor
Creating an HLSL-based Effect Using RenderMonkey

1. Setup all your effect parameters
   1. Variables
   2. Stream mapping
   3. Transforms
   4. Models and textures

2. Create HLSL vertex and pixel shaders

3. Link RenderMonkey parameter nodes to your HLSL shaders as necessary

4. Compile and explore!
Diffuse Lighting Effect

• Computes diffuse lighting based on the normal and the light vector

• Shader parameters:
  – ambient material color
  – diffuse material color
  – ambient coefficient
  – diffuse coefficient
  – light direction vector

• Assumes basic knowledge of how RenderMonkey interface works

• Distributed as Diffuse.xml workspace
HLSL Shader Editor Plug-in

- Allows incredible ease of shader creation

- Simple interface links RenderMonkey nodes to HLSL variables and samplers
  - Link vectors, colors, scalars and matrices to variable parameters
  - Link texture objects to samplers

- Control target and entry points for your shaders
Diffuse Lighting Effect: Vertex Shader

```c
struct VS_OUTPUT
{
    float4 Pos : POSITION;
    float3 Normal : TEXCOORD0;
    float3 Light : TEXCOORD1;
};

VS_OUTPUT main( float4 Pos : POSITION, float3 Normal : NORMAL )
{
    VS_OUTPUT Out = (VS_OUTPUT) 0;

    // Output transformed vertex position:
    Out.Pos = mul( view_proj_matrix, Pos );

    // Compute normal in view space:
    Out.Normal = normalize( mul( view_matrix, Normal ) );

    // Compute light vector in view space:
    Out.Light = normalize( lightDir );

    return Out;
}
```
Diffuse Lighting Effect: Pixel Shader

```c
float Ka;
float Kd;
float4 ambientColor;
float4 diffuseColor;

float4 main( float4 Diff : COLOR0,
             float3 Normal: TEXCOORD0,
             float3 Light : TEXCOORD1 ) : COLOR
{
    return Ka * ambientColor +
           Kd * diffuseColor * dot( Normal, Light );
}
```
Specular Lighting Effect

- Built upon previous example
- Computes diffuse and specular lighting
- Parameters:
  - Previous example’s parameters
  - Specular contribution and power coefficients
  - Specular material color

Steps:
1. Compute View vector
2. Compute Reflection vector
3. Compute specular component: $k_s \cdot \text{specular\_color} \cdot (R \cdot V)^n$
Specular Lighting: Vertex Shader

```c
struct VS_OUTPUT {
    float4 Pos : POSITION;
    float3 Normal : TEXCOORD0;
    float3 Light : TEXCOORD1;
    float3 View : TEXCOORD2;
};

VS_OUTPUT main( float4 Pos: POSITION, float3 Norm: NORMAL ) {
    VS_OUTPUT Out = (VS_OUTPUT) 0;

    Out.Pos = mul( view_proj_matrix, Pos ); // Transformed position
    Out.Normal = normalize( mul(view_matrix, Norm) ); // Normal
    Out.Light = normalize(-lightDir); // Light vector

    float3 Pview = mul( view_matrix, Pos ); // Compute view position
    Out.View = -normalize( Pview ); // Compute view vector

    return Out;
}
```

Specular Lighting: Pixel Shader

```c
float4 ambient;
float4 diffuse;
float4 specular;
float Ka;
float Ks;
float Kd;
float N;

float4 main( float4 Diff  : COLOR0,    float3 Normal: TEXCOORD0,
            float3 Light : TEXCOORD1, float3 View  : TEXCOORD2 )
            : COLOR
{
    // Compute the reflection vector:
    float3 vReflect = normalize(2*dot(Normal, Light)*Normal - Light);

    // Final color is composed of ambient, diffuse and specular
    // contributions:
    float4 FinalColor = Ka * ambient +
        Kd * diffuse * dot( Normal, Light ) +
        Ks * specular * pow( max( dot( vReflect, View), 0), N ) ;

    return FinalColor;
}
```
Advanced Shader Examples

- Marble shader
- Anisotropic metal shader
- Translucent iridescent shader
Veined Marble Shader
Veined Marble Shader

• Based on the marble RenderMan shader developed by Larry Gritz

• Designed to emulate marble material
  – Marble is formed from turbulent mixing of different bands of rock

• Creates solid marble texturing with distinct veins
  – Traditional approaches generate noise on CPU
  – This example uses volume texture for noise generation for faster results
RenderMonkey
Veined Marble Example

- Intuitive parameters control the final look:
  - ambient / diffuse / specular material colors and coefficients;
  - contributing light color and global ambient;
  - noise amplitude and frequency
  - vein turbulence sharpness and frequency

- HLSL_stones.xml
  - Veined marble Effect
Veined Marble Shader
Algorithm: Basic Steps

1. Generate noise value
2. Determine vein turbulence using noise
3. Blend vein color with base solid material
4. Shade using a diffusely reflected light
5. Add specular highlights
Noise Generation

- Use volumetric noise texture
- Sample the texture using interpolated object position modified by noise frequency
  - Prevents marble from “swimming”

```cpp
float4 noise (float4 x)
{
    return tex3D (noise_volume, x);
}

// Signed noise generation //
float4 snoise (float4 x)
{
    return 2.0f * tex3D (noise_volume, x) - 1.0f;
}
```
Vein Turbulence

- Compute vein turbulence based on modified Perlin algorithm:
  - Accumulate multiple octaves of noise with magnitude decreasing along with frequency - $\frac{1}{f}$ noise
  - Results in visual impression of turbulent fluid flow
  - Fades out the highest frequency turbulence components
Diffusely Reflective Marble

- Blend between vein color and base solid color using turbulence coefficient
- Multiply resulting color by diffusely reflected light

\[ \text{lerp} ( \quad ) = \]
Diffusely Reflective Marble

Blending between two colors using a blending factor:

```cpp
float4 mix(float4 a, float4 b, float blendFactor)
{
    // a * blendFactor + b * (1-blendFactor)
    return lerp(b, a, blendFactor);
}
```

Computing diffuse material color:

```cpp
// Get the blended color (pre-diffuse):
float4 Ct = mix (vein_color, base_color, turbulenceFactor);

// Compute diffuse and ambient contributions:
float4 DiffuseMarble = light_color *
    (Ct * ( Ka*ambient + Kd * soft_diffuse(Neye, Peye))
```
Shiny Marble

- Add specular highlights to solid material
  - Apply standard specular illumination formula, use $pow$ intrinsic function to compute the result
  - OR instead of computing full $pow$, only raise to $n = 32$ by using multiplications
    - Faster and leaner shader
Anisotropic Metal Shader
Anisotropic Metal Shader

- Simulates properties of anisotropic surfaces
  - Brushed metal surfaces (brass, silver, etc.)
  - CDs (iridescence component would be added)

- Models light scattering due to elongated features of brushed metal surfaces
  - Based on Heidrich / Seidel approach for computing anisotropic lighting

- Underlying material is modeled as a solid volume of metallic rings
  - Similar to modeling procedural wood
RenderMonkey Anisotropic Metal Example

- Parameters allow tweaking the look of brushed metal for the effect
  - Ring volume center, direction and scale
    - Control the direction and placement of ring “trunk” volume and size of the rings
  - 2 lights: positions and color
    - For anisotropic lighting computation
  - Ambient lighting color and contribution coefficient
  - Ring color
  - Material ambient color

- Workspace:
  - HLSL_Anisotropic.xml
Anisotropic Metal Shader Algorithm: Basic Steps

1. Determine the coordinates for vertex along the rings “trunk” volume to figure out direction of anisotropy

2. Compute anisotropic lighting contribution for two lights in the scene

3. Sample ring gradient and combine it with lighting contribution to achieve the final brushed metal look
Determining Direction of Anisotropy

Using input position, the ring volume texture coordinate is determined in texture space and propagated to the pixel shader.

```cpp
VS_OUTPUT main( float4 inPos: POSITION, float4 inNormal: NORMAL )
{
  VS_OUTPUT o = (VS_OUTPUT)0;

  // Texture space coordinates:
  float3 texPos = ((float3)inPos) - vRingCenter;
  float3 texSpacePos = texPos * vRingScale + vRingDirection;
  texSpacePos = mul(texSpacePos, (float3x3)texture_space_matrix);
  o.TexPos.xyz = texSpacePos;

  // Compute direction of anisotropy:
  float3 dirAniso = cross(inNormal, normalize(texPos));

  // Propagate direction of anisotropy:
  o.DirAniso = dirAniso * 0.5 + (float3)0.5;
  ...
```

Using input position, the ring volume texture coordinate is determined in texture space and propagated to the pixel shader.

Ring trunk center, direction and scale determine the placement of the ring's trunk volume.

Compute direction of anisotropy for the rings and propagate it to the pixel shader for later use.
Computing Anisotropic Lighting

- Compute and propagate the normal, light vectors for both lights and the view vector (vertex shader)
- In the pixel shader, compute contribution of each light using the light vector and direction of anisotropy
- Adjust for self shadowing

Light 1: $\rightarrow$ Light 2: $\rightarrow$ Result:

Game Developer Conference, San Jose, CA, March 7th 2003
In the vertex shader, propagate the normal vector to pixel shader.

- Compute camera position and the view vector.
- Compute light direction vectors for both contributing lights.

```cpp
// Output the normal vector (w component contains ambient factor)
o.Normal = float4( inNormal * .5 + (float3).5, fAmbient );

// Compute camera position:
float4 vCameraPosition = mul( inv_view_matrix, float4(0,0,0,1) );

// Calculate view vector:
o.View = normalize( vCameraPosition - mul( mWorld, inPos ) );

// Compute light direction vectors:
light1Position = mul( inv_view_matrix, light1Position );
light2Position = mul( inv_view_matrix, light2Position );
o.Light1 = normalize (light1Position - inPos );
o.Light2 = normalize (light2Position - inPos );
```
Computing Anisotropic Lighting: Pixel Shader

StrandPair StrandLight(float3 normal, float3 light, float3 view, float3 dirAniso)
{
    StrandPair o;
    float LdA = dot(light, dirAniso);
    float VdA = dot(view, dirAniso);
    float2 fnLookup = tex2D(strand, float2(LdA, VdA) * 0.5 + (float2)0.5);
    float spec     = fnLookup.y * fnLookup.y;
    float diff     = fnLookup.x;
    float selfShadow = saturate(dot(normal, light));
    o.Diffuse = diff * selfShadow;
    o.Specular = spec * selfShadow;
    return o;
}

float4 main( PS_INPUT i ) : COLOR
{
    ... 
    float3 color = 0;
    for (int l = 0; l < 2; l++)
    {
        StrandPair strand = StrandLight(normal, light[l], view, dirAniso);
        color += (strand.Diffuse + strand.Specular) * lightColor;
    }
    ...
Assembling The Final Look

- Ring color is determined by combining a sampled gradient texture and user-specified ring material color
  - Sample gradient texture by using previously computed ring volume coordinates
- Ring ambient material color is added to the result
- Final color is assembled from the anisotropic lighting contribution modulated by shadowing, final ring color and scene ambient color

Game Developer Conference, San Jose, CA, March 7th 2003
Example Of Anisotropic Metal Shader
Translucent Iridescent Shader: Butterfly Wings
Translucent Iridescent Shader: Butterfly Wings

• Simulates translucency of delicate butterfly wings
  – Wings glow from scattered reflected light
  – Similar to the effect of softly backlit rice paper

• Displays subtle iridescent lighting
  – Similar to rainbow pattern on the surface of soap bubbles

• Combines gloss, opacity and normal maps for a multi-layered final look
  – Gloss map contributes to satiny highlights
  – Opacity map allows portions of wings to be transparent
  – Normal map is used to give wings a bump-mapped look
RenderMonkey Butterfly Wings Shader Example

• Parameters that contribute to the translucency and iridescence look:
  – Light position and scene ambient color
  – Translucency coefficient
  – Gloss scale and bias
  – Scale and bias for speed of iridescence change

• Workspace:
  – HLSL_IridescentButterly.xml
Translucent Iridescent Shader Algorithm: Basic Steps

1. Compute light, view and halfway vectors in tangent space
2. Load base texture map, gloss map, opacity map and normal map
3. Compute diffusely reflected light
4. Compute scattered illumination contribution
5. Adjust for transparency of wings
6. Compute iridescence contribution
7. Add gloss highlights
8. Assemble final color
Translucent Iridescent Shader: Vertex Shader

```c
... // Propagate input texture coordinates:
Out.Tex = Tex;

// Define tangent space matrix:
float3x3 mTangentSpace;
mTangentSpace[0] = Tangent;
mTangentSpace[1] = Binormal;
mTangentSpace[2] = Normal;

// Compute the light vector (object space):
float3 vLight = normalize( mul( inv_view_matrix, lightPos ) - Pos );

// Output light vector in tangent space:
Out.Light = mul( mTangentSpace, vLight );

// Compute the view vector (object space):
float3 vView = normalize( mul( inv_view_matrix, float4(0,0,0,1)) - Pos );

// Output view vector in tangent space:
Out.View = mul( mTangentSpace, vView );

// Compute the half angle vector (in tangent space):
Out.Half = mul( mTangentSpace, normalize( vView + vLight ) );

return Out;
```
Translucent Iridescent Shader: Loading Information

float3 vNormal, baseColor;
float fGloss, fTranslucency;

// Load normal and gloss map:
float4(vNormal, fGloss) = tex2D(bump_glossMap, Tex);

// Load base and opacity map:
float4(baseColor, fTranslucency) = tex2D(base_opacityMap, Tex);

Load normal from a normal map and gloss value from a gloss map (combined in one texture map)
Load base texture color and alpha value from combined base and opacity texture map.
Diffuse Illumination For Translucency

\[
\text{float3 scatteredIllumination} = \text{saturate}(\text{dot}(-v\text{Normal}, \text{Light})) \times \text{fTranslucency} \times \text{translucencyCoeff};
\]

\[
\text{float3 diffuseContribution} = \text{saturate}(\text{dot}(v\text{Normal}, \text{Light})) + \text{ambient};
\]

\[
\text{baseColor} \ast= \text{scatteredIllumination} + \text{diffuseContribution};
\]

Light scattered on the butterfly wings is computed based on the negative normal (for scattering off the surface), light vector and translucency coefficient and value for the given pixel.

Compute diffusely reflected light using the bump-mapped normal and ambient contribution.

Combine diffuse and scattered light with base texture.
Adding Opacity to Butterfly Wings

Resulted color is modulated by the opacity value to add transparency to the wings:

```c
// Premultiply alpha blend to avoid clamping the highlights:
baseColor *= fOpacity;
```
Making Butterfly Wings Iridescent

Iridescence is a view-dependent effect. Scale and bias gradient map index to make iridescence change quicker across the wings.

```c
// Compute index into the iridescence gradient map, which consists of N*V coefficient
float fGradientIndex = dot(vNormal, View) * iridescence_speed_scale + iridescence_speed_bias;

// Load the iridescence value from the gradient map:
float4 iridescence = tex1D(gradientMap, fGradientIndex);
```

Resulting iridescence image:
Assembling Final Color

// Compute glossy highlights using values from gloss map:
float fGlossValue = fGloss * ( saturate( dot( vNormal, Half )) *
gloss_scale + gloss_bias );

// Assemble the final color for the wings
baseColor += fGlossValue * iridescence;
Example of Translucent Iridescent Shader
Tips To Improve Your HLSL Output

• Pay attention to your loops
  – Flow control doesn’t come for free in DirectX® 9.0, basic flow control is added in the next DX release
  – Loops are unrolled ⇒ slower shader execution time

• Use HLSL intrinsic functions
  – Helps hardware to optimize your shaders

• Use float, float3, and float4 as appropriate
  – Stuff your values into a vector when needed
  – However, proper use of scalar operations helps both the compiler and the hardware
  – Neat example:

```cpp
float3 vNormal;
float fOpacity;
float4( vNormal, fOpacity ) = tex2D( normalAndOpacityMap, Tex );
```
Generate Better PS_1_x Code Through Good HLSL Use

- **Signed Scale (_bx2)**
  
  \[
  \text{value} \times 2 - 1
  \]
  generates \_bx2 modifier on the source register

- **Times 2, 4, 8, Divide by 2, 4, 8**
  
  1. Source register modifiers: for \(n = 2, 4, 8, 0.5, 0.25, 0.125\), multiplying the source value by \(n\) will generate the modifier.
     
     Example: \[
     \text{return Value*2}
     \] generates \_x2 modifier on the source reg.

  2. Destination modifiers:
     
     a. for \(X = 2, 4, 8, 0.5, 0.25, 0.125\), declare \(\text{static const float } N = X;\)

     b. Use that variable to generate a destination modifier:

        Example: \[
        \text{static const float } N = 2;
        \text{float4 } f = (\text{Value1} + \text{Value2}) \times N;
        \]
        generates \[
        \text{add}_x2 \ r0,v0, v1
        \]
More Tips To Generate Better PS_1_x Code Through Good HLSL Use

• Complement (1 – x)
  \[ 1 - \text{value} \] generates 1 – rX modifier on the source register, where rX is the source register.

• Saturate ( _sat )
  – Use either \texttt{saturate()} or \texttt{clamp()} intrinsics
  – _sat can be combined with other modifiers

• Negate ( - X )
  – Simplest to use: \[ \text{return } -\text{Color}; \] generates \[ \text{mov } r0, -c0 \]

• Bias (_bias)
  \[ \text{value} \times 0.5f \] generates _bias modifier on the source register
  – Note that _bias cannot be used in ps_1_1/2/3 unless the source is in [0,1] range
Summary

• Use HLSL for your shader development
  – All the usual advantages of High Level Languages
    • Faster, easier development
    • Code re-use
    • Optimization
      – Current HLSL compiler is very good and getting better every day
      – Industry standard which will run on any DirectX® shader chip

• Use RenderMonkey for your shader prototyping and development
  – Powerful tool with intuitive interface
  – Prototype your shaders and then fully explore them like never before until you find the look you like
ATI Developer Resources

• Visit the ATI Developer Relations website to view this and other presentations:
  – www.ati.com/developer

• Download RenderMonkey™:
  – Full documentation available
  – All shaders discussed during this talk are included in the installer

• View *High Level Shading With DirectX 9® on ATI’s RADEON™ 9700 Series* Net Seminar:
  – http://www.ati.com/developer