D3DX Effects and the DirectX 9 High-Level Shading Language

or *How I Learned to Stop Worrying and Love Shaders*

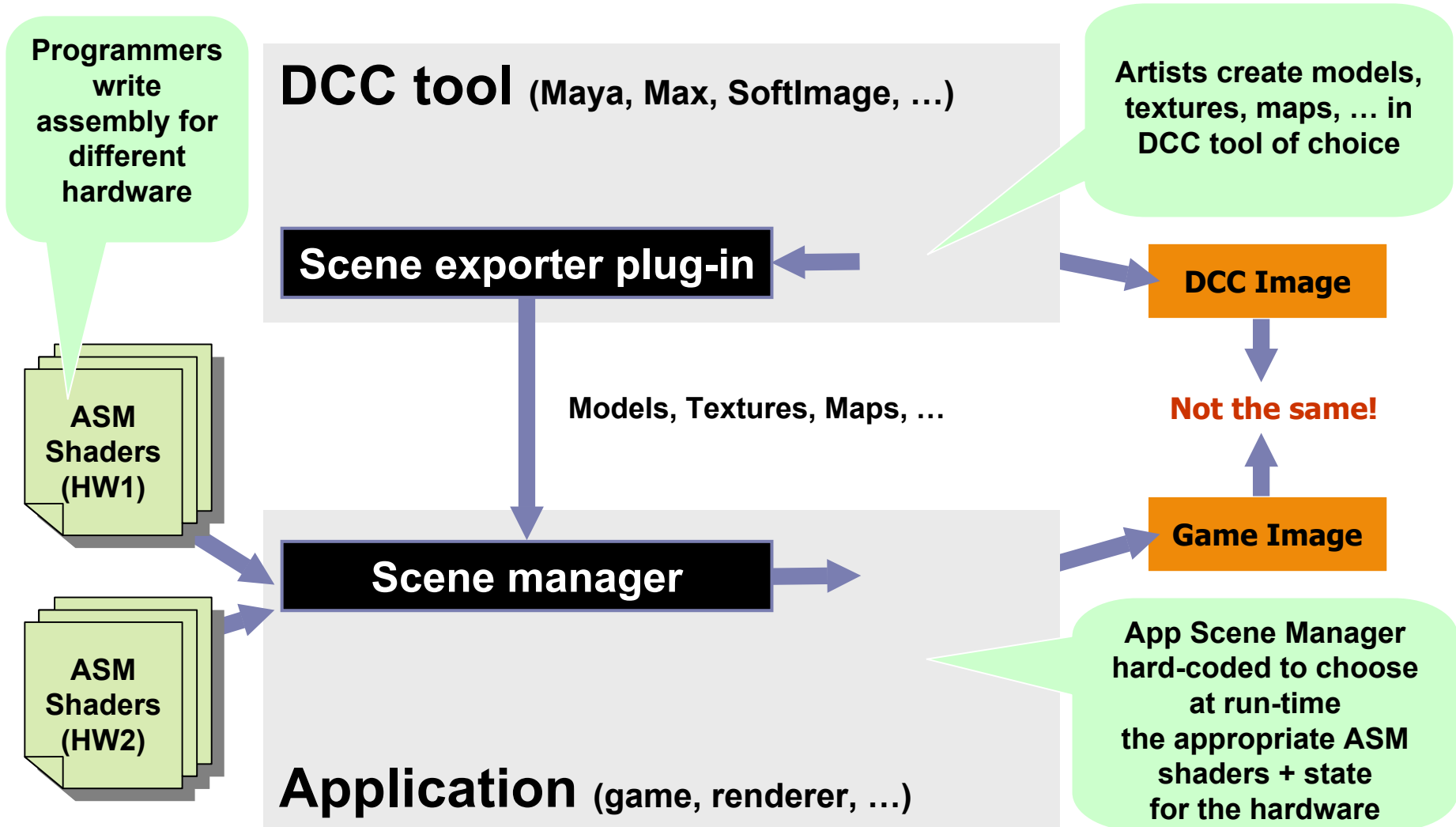Ashu Rege

ARege@nvidia.com

Guennadi Riguer

GRiguer@ati.com

Outline

- **Current Game Production Pipeline**
- **Why HLSL and FX**
- **FX-enabled Production Pipeline**
- **HLSL and FX Overview and Concepts**
- **FX Demos**
- **High-Level Shader Language**
  - **Language Constructs**
  - **Functions**
  - **Semantics**
- **PS 1.x Shaders In HLSL**
- **HLSL Shader Examples**
- **HLSL Optimizations**

Typical Production Pipeline

Programmers write assembly for different hardware

DCC tool (Maya, Max, SoftImage, …)

Artists create models, textures, maps, … in DCC tool of choice

Scene exporter plug-in

DCC Image

ASM Shaders (HW1)

Models, Textures, Maps, …

Not the same!

Scene manager

Game Image

ASM Shaders (HW2)

App Scene Manager hard-coded to choose at run-time the appropriate ASM shaders + state for the hardware

Application (game, renderer, …)

Typical Production Pipeline

- **Programmers write game engine/toolset and shaders**
  - **Develop a selection of assembly shaders for artists**
  - **Integrate shaders into engines**
  - **Combine various shaders + states to create effects**
  - **Develop different versions of shaders for different hardware targets**
  - ***Lather, rinse, repeat***

Typical Production Pipeline

- **Artists create content**
  - **Models, textures etc. created in DCC tools**
  - **Exported to engine or custom viewer for preview**
  - **Programmer-developed effects + Artist-created content visualized in engine or viewer**
  - **Identify improvements and problems**
  - *Lather, rinse, repeat*

What are the Problems?

- **Programmers are good at writing code**
  - **But writing and debugging long assembly shaders is a pain! (Hello Pixel Shader 2.0!)**
  - **Hard-coded assembly tedious to modify and maintain**
  - **Reuse of assembly shaders limited**

What are the Problems?

- **Artists are good at making stuff look cool**
  - **But not necessarily at writing assembly**
- **Writing shaders is a creative process – artist input is critical**
- **Back-and-forth time-consuming process**

What is the Solution?

- **High-level Shading Languages**
  - **High-level Shading Languages make shader creation accessible to everyone, especially artists**
  - **Eliminates painful authoring, debugging and maintenance of long assembly shaders**
  - **Many artists, especially from film studios, familiar with Renderman or variants**
  - **DirectX 9 HLSL provides syntax similar to Renderman for shader authoring**

HLSL Example

Assembly

...

dp3 r0, r0, r1

max r1.x, c5.x, r0.x

pow r0.x, r1.x, c4.x

mul r0, c3.x, r0.x

mov r1, c2

add r1, c1, r1

mad r0, c0.x, r1, r0

...

HLSL

...

float4 cSpec = pow(max(0, dot(Nf, H)), phongExp).xxx;

float4 cPlastic = Cd * (cAmbi + cDiff) + Cs * cSpec;

...

Simple Blinn-Phong shader expressed in both assembly and HLSL

We are Not Home Yet!

- **In an ideal world:**
  - **If the artist wants a cool new effect, (s)he should be able to create it or load it into the DCC package of their choice, tweak its parameters and view it in both the DCC tool and the engine**
  - **You want WYSIWYG across BOTH the DCC package and the game engine**
  - **Eliminate multiple iterations caused by differences in the effect in different environments**
  - **You want *one* mechanism to describe an entire effect for multiple hardware targets**
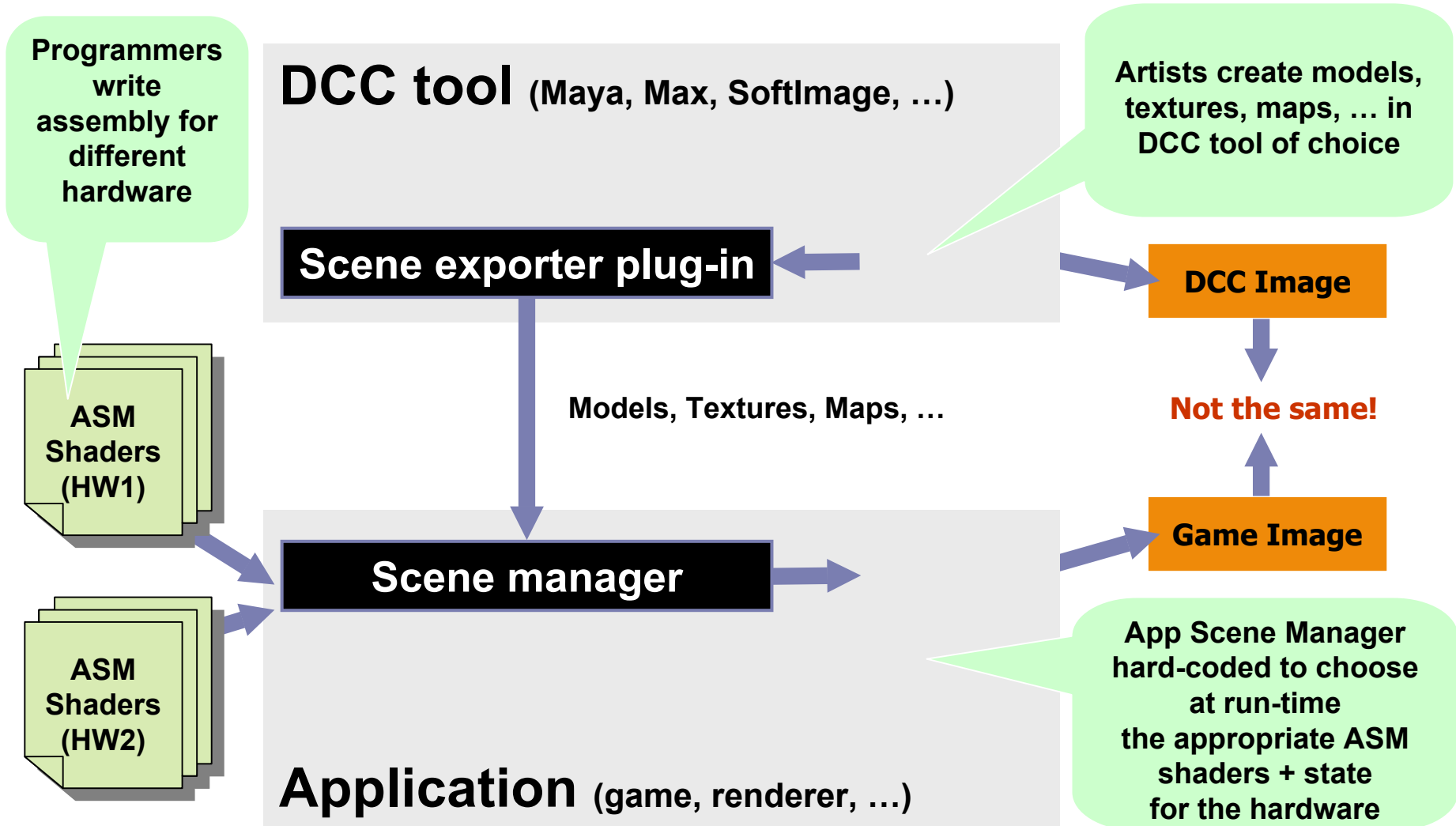
What are the Problems? – Part 2

- **HLSL shaders not the full answer…**
  - **Only describe *part* (vertex or pixel shader) of *one pass* of the entire effect**
  - **We also need the *shading context***
  - **An *effect* comprises more than just shaders**
    - **An entire collection of render states, texture states, …**
    - **A mechanism to express the same shading idea across different hardware and API's.**
  - **Everything else required to show it correctly in a game engine AND a DCC application such as MAX/Maya/SoftImage/…**
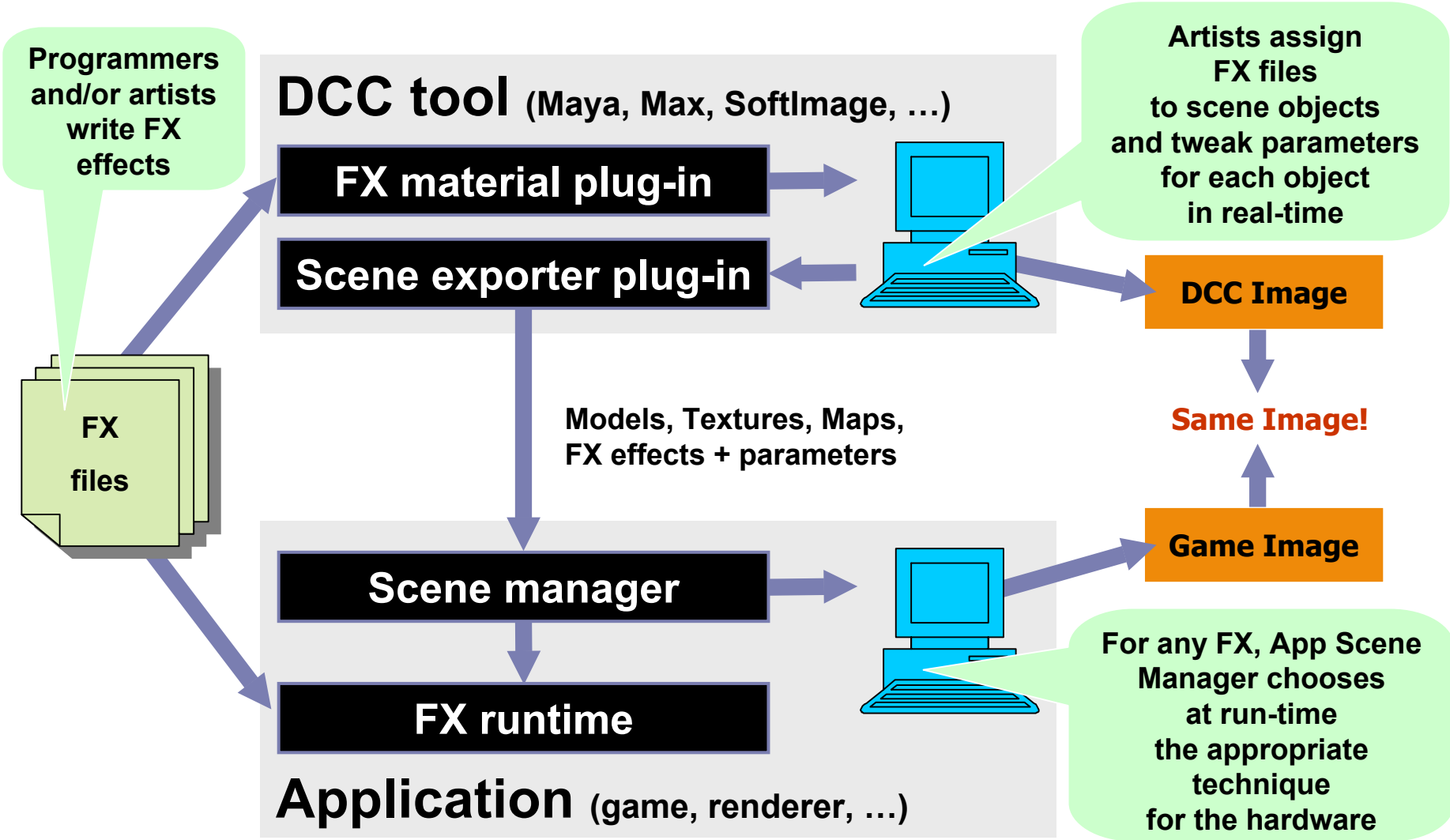
What is the Solution? – Part 2

- **FX file format and FX runtime API**
  - **Originally introduced in Dx8, extensively modified and extended in Dx9.**

- **Everthing we need**
  - **FX file *contains* HLSL and/or assembly vertex and pixel shaders**
  - **Parameters that can be exposed/tweaked**
  - **Fixed function state**
  - **Other render, texture, etc. states**
  - **Multiple implementations (*techniques*) for targeting different hardware**
  - **Techniques can be single or multi-pass**

Typical Production Pipeline

Programmers write assembly for different hardware

DCC tool (Maya, Max, SoftImage, …)

Artists create models, textures, maps, … in DCC tool of choice

Scene exporter plug-in

DCC Image

ASM Shaders (HW1)

Models, Textures, Maps, …

Not the same!

Game Image

Scene manager

ASM Shaders (HW2)

App Scene Manager hard-coded to choose at run-time the appropriate ASM shaders + state for the hardware

Application (game, renderer, …)

FX-Enabled Production Pipeline

Programmers and/or artists write FX effects

DCC tool (Maya, Max, SoftImage, …)

FX material plug-in

Scene exporter plug-in

Artists assign FX files to scene objects and tweak parameters for each object in real-time

DCC Image

FX files

Models, Textures, Maps, FX effects + parameters

Same Image!

Game Image

Scene manager

FX runtime

Application (game, renderer, …)

For any FX, App Scene Manager chooses at run-time the appropriate technique for the hardware

Effect File Structure

- **An effect is made up of multiple rendering algorithms (*techniques*) each made up of one or more passes**

- **Effect File Structure:**

  Variable declarations

  Technique 1
    - **Pass 1**
    - **...**
    - **Pass n**

  ...

  Technique n
    - **Pass 1**
    - **...**
    - **Pass n**

FX Example – Diffuse Color

```
technique Simple                    A technique is made up of passes
{
    pass p0
    {                               An effect is made up of techniques

        Lighting = true;                    // Enable Lighting
        LightEnable[0] = true;              // Enable Light 0
        ZEnable = true;                     // Enable DepthTest
        ZWriteEnable = true;                // Enable Depth Buffer Writes

        // Assign diffuse color to be used
        ColorOp[0]   = SelectArg2;          A pass can set render states
        ColorArg1[0] = Texture;
        ColorArg2[0] = Diffuse;
        AlphaOp[0]   = SelectArg2;
        AlphaArg1[0] = Texture;
        AlphaArg2[0] = Diffuse;             ... texture stage states, ...
    } // end pass p0
} // end technique Simple
```

FX Example - Texturing

```
texture diffuseTexture : DiffuseMap;
technique SimpleTexture
{
    pass p0
    {
        ...
        Texture[0] = <diffuseTexture>;          // Assign texture to stage 0
        MinFilter[0] = Linear;                   // Set filter values...
        MagFilter[0] = Linear;
        MipFilter[0] = Linear;
        // Modulate texture with diffuse color
        ColorOp[0]   = Modulate;
        ColorArg1[0] = Texture;
        ColorArg2[0] = Diffuse;
        AlphaOp[0]   = SelectArg2;
        AlphaArg1[0] = Texture;
        AlphaArg2[0] = Diffuse;
    } // end pass p0
} // end technique SimpleTexture
```

You can declare variables outside a technique...

...and use them inside the passes of any technique

FX Example – HLSL functions

```
struct vertexIn {
    float4 Position : POSITION;         // position in object space
    float3 Normal : NORMAL;             // normal in object space
    float2 TexCoord : TEXCOORD0;
    float3 T : TEXCOORD1;               // tangent in object space
    float3 B : TEXCOORD2;               // binormal in object space
};

struct vertexOut {
    float4 Position : POSITION;         // position in clip space
    float4 TexCoord0 : TEXCOORD0;       // texcoords for diffuse map
    float4 TexCoord1 : TEXCOORD1;       // texcoords for normal map
    float4 LightVector : TEXCOORD2;     // interpolated light vector
};

vertexOut DiffuseBumpVS(vertexIn IN, uniform float4x4 WorldViewProj,
    uniform float4x4 WorldIMatrix, uniform float4 LightPos)
{
    vertexOut OUT;
    ...
    // transform position to clip space
    OUT.Position = mul(WorldViewProj, IN.Position);
    return OUT;
}
```

You can define struct types...

...and use them in HLSL functions

FX Example – HLSL functions

```
float4x4 WorldIMatrix : WorldI;  // World Inverse matrix
float4x4 wvpMatrix : WorldViewProjection;
float4 lightPos : Position
<
    string Object = "PointLight";
    string Space = "World";
> = { 100.0f, 100.0f, 100.0f, 0.0f };


technique DiffuseBump
{
    pass p0
    {

        ...
        Zenable = true;
        ZWriteEnable = true;
        CullMode = None;
        VertexShader = compile vs_1_1 DiffuseBumpVS(wvpMatrix,WorldIMatrix,lightPos);

        ...
    } // end pass p0
} // end technique DiffuseBump
```

HLSL function invocation

Specify target: vs_1_1, ps_1_1, ps_2_0, …
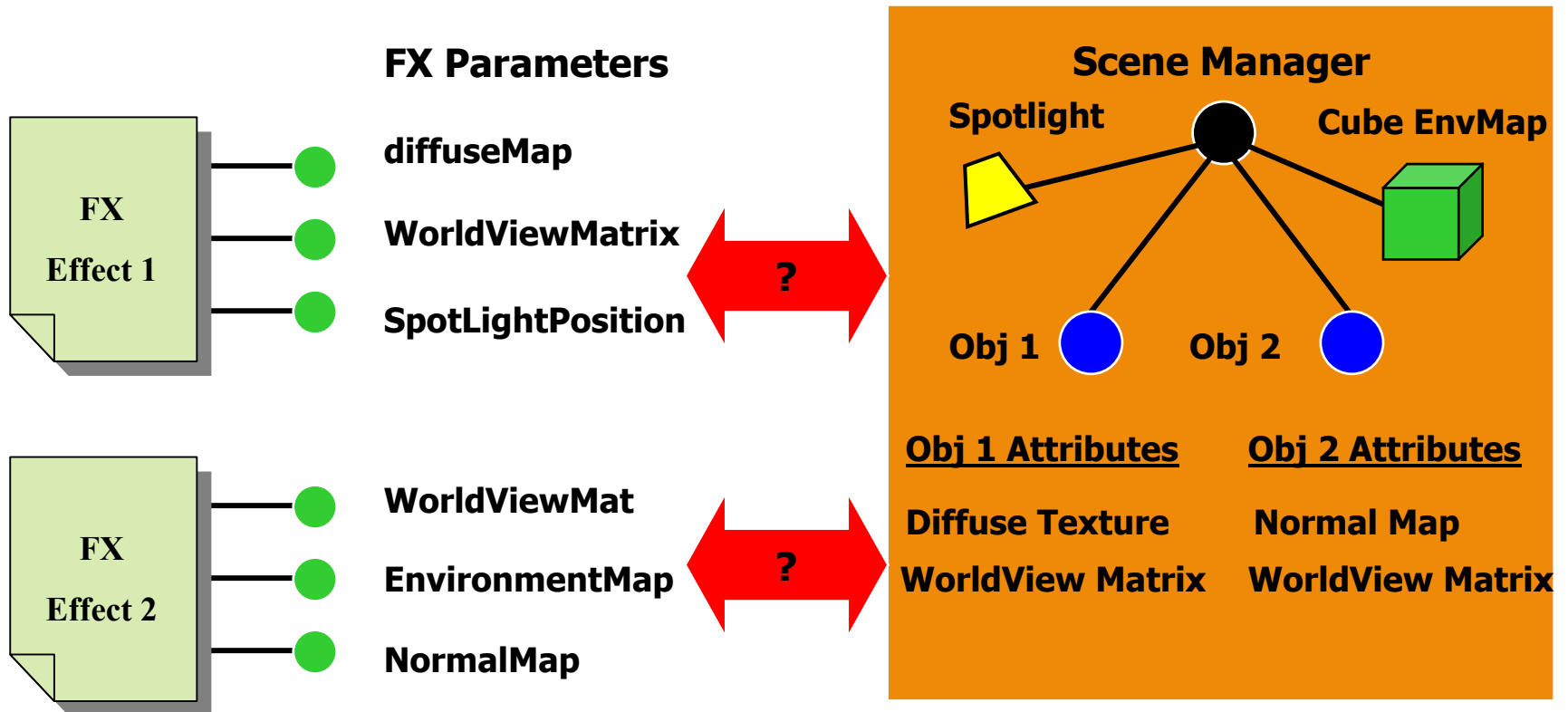
FX Example – Assembly Shaders

```
technique DiffuseBump
{
    pass p0
    {
        ...
        Zenable = true;
        ZWriteEnable = true;
        CullMode = None;
        VertexShaderConstant[4] = <wvpMatrix>;
        VertexShader =
        asm
        {
            vs_1_1
            ...
            m4x4 oPos, v0, c4              // pos in screen space.
        };
    } // end pass p0
} // end technique SimpleTexture
```

Old skool assembly shader

Making Connections

FX Parameters

FX Effect 1

- diffuseMap
- WorldViewMatrix
- SpotLightPosition

?

Scene Manager

Spotlight

Cube EnvMap

Obj 1   **Obj 2**

FX Effect 2

- WorldViewMat
- EnvironmentMap
- NormalMap

?

Obj 1 Attributes	Obj 2 Attributes
Diffuse Texture	Normal Map
WorldView Matrix	WorldView Matrix

How do we connect parameters in the FX files to the scenes in the game?

Semantics

```
texture diffuseTexture : DiffuseMap;
```

Semantics

```
float4 spotlight1Direction : Direction
<
    string Object = "SpotLight";
    string Space = "DeviceLightSpace";
> = {1.0f, 0.0f, 0.0f, 0.0f};
```

Semantics

```
texture diffuseTexture : DiffuseMap;
```

- **Each variable can optionally have a semantic**
- **Semantics are essentially user-defined strings**
- **Semantics provide a 'meaning' to the variable**
- **Application can query a variable for its semantic**
- **Application can use semantic to set appropriate value for a variable**

Annotations

```
texture normalizationCubeMap
<
    string File = "normalize.dds";                  Annotations
>;

float reflStrength
<
    string gui = "slider";
    float uimin = 0.1;
    float uimax = 1.0;
    float uistep = 0.05;
    string Desc = "Edge reflection";
    float min = 0.1;
    float max = 1.0;
> = 1.0;
```

Annotations

```
texture normalizationCubeMap
<
    string File = "normalize.dds";
>;
```

- **Each variable can optionally have multiple annotations**
- **Annotations are essentially user-defined strings**
- **Annotations provide more information about the variable to the application**
- **Application can query a variable for its annotations**
- **Application can use annotations to set appropriate value for a variable**

Annotating Techniques

```
technique BumpyShinyHiQuality
<
    float quality = 5.0;
    float performance = 1.0;
    ...
>;
{
    pass p0 { ... }
}

technique BumpyShinyHiPerf
<
    float quality = 1.0;
    float performance = 5.0;
    ...
>;
{ ... }
```

Annotations for techniques

- **Annotations can be used to identify characteristics of technique and other info for the application**

Annotating Passes

```
technique multiPassGlow
{
    pass p0
    <
            bool renderToTexture = true;
            float widthScale = 0.25;
            float heightScale = 0.25;
            ...
    >
    {
            Zenable = true;
            ...
    }
}
```

Annotations for passes

- **Annotations can be used to identify requirements for each pass and other info for the application such as render to texture**

Automatic Parameter Discovery

- **Semantics and annotations provide powerful mechanism for automating parameter discovery**
- **What we want: Write the application once and use any FX effect file without recompiling the app**
- **Use semantics and annotations to create a common language for your engine and FX effects**
- **Initial effort to write the parameter discovery code in your app, after that all debugging is in the FX files!**

Using FX in Your Application

- **Load effect**
- **Validate technique for hardware**
- **Detect parameters for technique**
- **Render Loop (for each object in scene):**
  - **Set technique for the object**
  - **Set parameter values for technique**
  - **For each pass in technique**
    - **Set state for pass**
    - **Draw object**

Using FX – The FX API

```cpp
LPD3DXBUFFER pError = NULL;
D3DXCreateEffectFromFile(m_pd3dDevice, _T("simple.fx"),
    NULL, NULL, 0, NULL, &m_pEffect, &pError);
SAFE_RELEASE(pError);


. . .
UINT iPass, cPasses;
m_pEffect->SetTechnique("Simple");

m_pEffect->SetVector("var1", v);

m_pEffect->Begin(&cPasses, 0);
for (iPass = 0; iPass < cPasses; iPass++)
{
   m_pEffect->Pass(iPass);
   m_pMesh->Draw();
}
m_pEffect->End();
```

High-Level Shader Language (HLSL)

- **C like language with special shader constructs**

- **All the usual advantages of high level languages**
  - **Faster, easier development**
  - **Code re-use**
  - **Optimization**

- **Industry standard which will run on cards from any vendor**

HLSL Data Types

- **Scalar data types**
  - **bool – TRUE or FALSE**
  - **int – 32-bit signed integer**
  - **half – 16-bit floating point value**
  - **float – 32-bit floating point value**
  - **double – 64-bit floating point value**
- **Support of various types not guarantied and depends on hardware implementation**

HLSL Data Types

- **Vector types**
  - **By default all 4 components are floats**
  - **Various types already predefined**
    ```
    vector vVar;
    vector<float,4> vVar;
    float4 vVar;
    int2 vVar1;
    ```

- **Access to vector components**
  ```
  vVar.x
  vVar[0]
  ```

HLSL Data Types

- ## Matrix types
  - ### By default all 4x4 elements are floats
  - ### Various types already predefined
    ```
    matrix mVar;
    matrix<float,4,4> mVar;
    float4x4 vVar;
    int2x3 vVar1;
    ```

- ## Access to matrix elements
    ```
    mVar._m00,    mVar._11,    mVar[0][0]
    mVar._m00_m01_m02_m03,    mVar[0]
    ```

- ## Can control matrix orientation
    ```
    #pragma pack_matrix (row_major);
    ```

HLSL Data Types

- **Arrays supported**

```
float2 offs2D[5];
```

- **Structs supported**

```
struct VERTEX
{
    float3 Pos;
    float3 Norm;
    float2 TexCoord;
}
```

Type Casts

- **Floats can be promoted by replication**
  `vVec3+0.5` ⬄ `vVec3+`**`float3`**`(0.5,0.5,0.5)`

- **Vectors and matrices can be downcast**
  - **Picking from left/upper subset**

- **Structures can be cast to and from scalars, vectors, matrices and other structures**

Operators

	Operators
Arithmetic	-, +, *, /, %
Prefix/postfix	++, --
Boolean	&&, ||, ?:
Unary	!, -, +
Comparison	<, >, <=, >=, ==, !=
Assignment	=, -=, +=, *=, /=
Cast	(*type*)
Comma	,
Structure member	.
Array member	[*i*]

Operators

- **Matrix-vector multiplication is defined as intrinsic function**
- **Modulo operator (%) works with integers and floats**
- **Vector comparisons work on per-component basis**
- **Be careful with integer math**
  - **HLSL emulates integers if not natively supported, so rounding might produce different results**

Constructors

- **Constructors can be used to create and initialize objects like in C++**
  - **Works with all HLSL types**

    ```
    float3 v3 = float3(0,0,0);
    float4 v4 = float4(0,0,0,0);
    float4 v4 = float4(v3,0);
    ```

- **Can initialize members of the structues**
  - **Casting occurs if number of components doesn't match**

Flow Control

- ## Branching
  ```
  if (expr) then statement [else statement]
  ```

- ## Loops
  ```
  do statement while (expr);
  while (expr) statement;
  for (expr1;expr2;expr3) statement
  ```

- ## Loops are supported in all models
  - ### Unrolled if necessary

Functions

- **C like functions**
  - **HLSL does type checking**
  - **Prototypes are supported**
  - **Arguments passed by value**
- **Recursion not supported**
- **Function can use special semantics**
- **Large number of predefined functions**
  - **Simplify development**
  - **Highly optimized implementation**
  - **Can be overloaded**

Functions

- **Function parameters can have initializers**
- **Functions can return multiple values**

```
float4 foo( in float v,
            out float a,
            inout float b = 0.5 )
{
    a = v * b;
    b = v / 2;

    return (a + b);
}
```

Intrinsic Functions

Various math functions

	VS 1.1	VS 2.0	PS 1.1	PS 1.4	PS 2.0
degrees, lerp, radians, saturate	X	X	X	X	X
abs, clamp, isfinite, isnan, max, min, sign	X	X		X	X
acos, asin, atan, atan2, ceil, cos, cosh, exp, exp2, floor, fmod, frac, frexp, isinf, ldexp, log, log2, log10, modf, pow, round, rsqrt, sin, sincos, sinh, smoothstep, sqrt, step, tan, tanh	X	X			X

Intrinsic Functions

Vector functions

	VS 1.1	VS 2.0	PS 1.1	PS 1.4	PS 2.0
dot, reflect	X	X	X	X	X
any, cross, faceforward	X	X		X	X
distance, length, lit, normalize, refract	X	X			X

Matrix functions

	VS 1.1	VS 2.0	PS 1.1	PS 1.4	PS 2.0
mul, transpose	X	X	X	X	X
determinant	X	X			X

Intrinsic Functions

Texturing functions

	VS 1.1	VS 2.0	PS 1.1	PS 1.4	PS 2.0
tex1D, tex2D, tex3D, texCube			X	X	X
tex1Dproj, tex2Dproj, tex3Dproj, texCUBEproj, tex1Dbias, tex2Dbias, tex3Dbias, texCUBEbias				X	X
clip			X	X	X

Miscellaneous functions

	VS 1.1	VS 2.0	PS 1.1	PS 1.4	PS 2.0
D3DCOLORtoUBYTE4	X	X			?

HLSL Shader Semantics

- **Function arguments and results might be semantically bound to shader inputs/outputs**
  - **I.e. POSITION, TEXCOORD1, COLOR0**
  - **Meaningful only at top level**
- **Constants can be bound to registers**

```
matrix worldViewProj : register(c0);
```

- **Samplers can also be bound**

```
sampler noiseSampler : register(s0);
```

Texture Sampler Declarations

- **Textures and samplers have to be declared**
  - Sampler configuration can be provided for D3DX Effects use

```
texture tMarbleSpline;
sampler MarbleSplineSampler = sampler_state
{
    Texture = (tMarbleSpline);
    MinFilter = Linear;
    MagFilter = Linear;
    MipFilter = Linear;
    AddressU  = Clamp;
    AddressV  = Clamp;
};


sampler SimpleSampler;
```

Shader Model Support in HLSL

- **Right now supports:**
  - **VS 1.1**
  - **VS 2.0**
  - **PS 1.1-1.3**
  - **PS 1.4**
  - **PS 2.0**
- **Support for other models will be added in the near future**
- **PS 1.x can be tricky to use to get the most out of shaders**

PS 1.x Shaders With HLSL

- **HLSL supports PS 1.1-1.4, but there are some nuances**
- **HLSL supports almost all capabilities of each shader model (including modifiers)**
- **Functionality of course is limited by shader model**
- **Knowledge of assembly is helpful**

Support Of Modifiers In HLSL Pixel Shaders

- **Compiler recognizes and uses instruction and argument modifiers**

```
a = b*(c*2-1);            // mul r0, r0, r1_bx2
a = dot(b,(c-0.5f)*2);    // dp4 r0, r0, r1_bx2
a = b*(c-0.5f);           // mul r0, r0, r1_bias
a = dot(b,c*2);           // dp4 r0, r0, r1_x2
a = b*(1-c);              // mul r0, r0, 1-r1
a = -b*c;                 // mul r0, -r0, r1
a = 2*b*c;                // mul_x2 r0, r0, r1
a = (b+c)*4;              // add_x4 r0, r0, r1
a = (b+c)/8;              // add_d8 r0, r0, r1
a = saturate(b+c);        // add_sat r0, r0, r1
a = clamp(b+c,0,1);       // add_sat r0, r0, r1
```

PS 1.1-1.3 Shaders With HLSL

- **Range of computed values should be −1..+1**
- **Texture coordinates available for computations should be 0..1**
- **Texture coordinates are tied to samplers**
- **Dependent texture read is limited**
- **In most cases access to .W texture coordinate is not permitted**
- **Result masks and argument swizzles are not reasonable**

Per-Pixel Diffuse Lighting in HLSL (PS 1.1 Model)

```hlsl
sampler normalMap: register(s0);
sampler diffuseCubeMap: register(s3);
float4 vAmbient;
float4 vDiffuse;

float4 main( float2 TexCoord : TEXCOORD0,
             float3 EnvXform[3] : TEXCOORD
{
    float3 N = tex2D(normalMap, TexCoord);

    float3 Nworld;
    Nworld.x = dot(N*2-1, EnvXform[0]);
    Nworld.y = dot(N*2-1, EnvXform[1]);
    Nworld.z = dot(N*2-1, EnvXform[2]);

    float4 diffuse = texCUBE(diffuseCubeMap, Nworld);
    return (diffuse * vDiffuse + vAmbient);
}
```

Per-Pixel Diffuse Lighting in HLSL (PS 1.1 Model)

- **Compiler recognizes normal transformation, dependent cube map lookup and translates into appropriate instructions with modifiers**

```
ps_1_1
tex t0
texm3x3pad t1, t0_bx2
texm3x3pad t2, t0_bx2
texm3x3tex t3, t0_bx2   // Dependent texture read

mad r0, t3, c1, c0
```

Per-Pixel Specular Lighting in HLSL (PS 1.1 Model)

```hlsl
sampler normalMap: register(s0);
sampler specularCubeMap: register(s3);
float4 vAmbient;
float4 vSpecular;

float4 main( float4 diffuse : COLOR,
             float2 TexCoord : TEXCOORD0,
             float4 EnvXform[3] : TEXCOORD1 ) : COLOR
{
    float3 N = tex2D(normalMap, TexCoord);
    float3 Nworld;
    Nworld.x = dot(N*2-1, EnvXform[0]);
    Nworld.y = dot(N*2-1, EnvXform[1]);
    Nworld.z = dot(N*2-1, EnvXform[2]);

    float3 Eye;
    Eye.x = EnvXform[0].w;
    Eye.y = EnvXform[1].w;
    Eye.z = EnvXform[2].w;

    float3 R = 2 * dot(Nworld, Eye) * Nworld – Eye * dot(Nworld, Nworld);
    float4 specular = texCUBE(specularCubeMap, R);
    return (specular * vSpecular + diffuse + vAmbient);
}
```

Per-Pixel Specular Lighting in HLSL (PS 1.1 Model)

- **Compiler correctly identifies all operations and translates them into modifiers and even texm3x3vspec instruction!**

```
ps_1_1
tex t0
texm3x3pad t1, t0_bx2
texm3x3pad t2, t0_bx2
texm3x3vspec t3, t0_bx2   // Dependent texture read

mad r0, t3, c1, v0
add r0, r0, c0
```

Per-Pixel Anisotropic Lighting in HLSL (PS 1.1 Model)

```hlsl
sampler anisoDirMap: register(s0);
sampler baseMap: register(s1);
sampler anisoLookup: register(s3);
float4 vAmbient;

float4 main( float2 AnisoTexCoord : TEXCOORD0,
             float2 BaseTexCoord : TEXCOORD1,
             float3 Ltan : TEXCOORD2,
             float3 Vtan : TEXCOORD3 ) : COLOR
{
    float3 anisoDir = tex2D(anisoDirMap, AnisoTexCoord);
    float4 baseTex = tex2D(baseMap, BaseTexCoord);

    float2 v;
    v.x = dot(anisoDir, Ltan);
    v.y = dot(anisoDir, Vtan);

    float glossMap = baseTex.a;
    float4 aniso = tex2D(anisoLookup, v);
    return (baseTex * (aniso + vAmbient) + aniso.a * glossMap);
}
```

Per-Pixel Anisotropic Lighting in HLSL (PS 1.1 Model)

- **Again compiler correctly recognizes and translates dependent texture read operation**

```
ps_1_1
tex t0
tex t1
texm3x2pad t2, t0
texm3x2tex t3, t0      // Dependent texture read

add r0, t3, c0
mul r1.w, t3.w, t1.w
mad r0, r0, t1, r1.w
```

PS 1.4 Shaders With HLSL

- **Range of computed values should be –8..+8, consts –1..+1**
- **Texture coordinates available for computations should be –8..+8**
- **One level of dependent texture read**
- **In most cases access to .W texture coordinate is not permitted**
- **Right now projective textures don't work**
- **Argument swizzles are not reasonable, but channel replication is fine**

Ghost Shader in HLSL (PS 1.4 Model)

```hlsl
sampler normMap;
sampler normCubeMap;
sampler lookupMap;
float3 ghostColor;

float4 main( float2 texCoord : TEXCOORD0,
             float3 Eye : TEXCOORD1,
             float3 envXform[3] : TEXCOORD2 ) : COLOR
{
    float3 N = tex2D(normMap, texCoord) * 2 - 1;
    Eye = texCUBE(normCubeMap, Eye) * 2 - 1;

    float3 NN;
    NN.x = dot(N, envXform[0]);
    NN.y = dot(N, envXform[1]);
    NN.z = dot(N, envXform[2]);

    float NdotE = dot(NN, Eye);
    float ghost = tex1D(lookupMap, NdotE);
    return float4(ghostColor * ghost, ghost / 2);
}
```

Ghost Shader in HLSL (PS 1.4 Model)

- **Compiler identifies proper modifiers and uses phase with dependent texture read operation**

```
ps_1_4
texcrd r0.xyz, t2
texld r1, t0
texld r2, t1
texcrd r3.xyz, t3
texcrd r4.xyz, t4
dp3 r0.x, r1_bx2, r0
dp3 r0.y, r1_bx2, r3
dp3 r0.z, r1_bx2, r4
dp3 r0.xy, r0, r2_bx2
phase
texld r0, r0
mul r1.xyz, r0.x, c0
+mov r1.w, r0.x
mov r0, r1
```

Granite Shader

Granite – Vertex Shader

```
float4x4 view_matrix;
float4x4 view_proj_matrix: register(c0);

struct VS_OUTPUT
{
   float4 Pos: POSITION;
   float3 P     : TEXCOORD0;
   float3 Peye : TEXCOORD1;
   float3 Neye : TEXCOORD2;
};

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

   Out.P = Pos;
   Out.Peye = mul(view_matrix, Pos);
   Out.Neye = mul(view_matrix, Norm);
   return Out;
}
```

Granite – Pixel Shader (PS 2.0 Model) 1/3

```
float power = 10;
float4 Ks;
float freq = 0.1;
float4 light_pos;
float4 Kd;
float4 Ka;
sampler noise_volume;

#define OCTAVES 6

float noise(float3 p)
{
    float n = 0;
    float k = 1;
    for(int i = 0; i < OCTAVES; i++)
    {
        n += tex3D(noise_volume, p * k) / k;
        k *= 2;
    }

    return n / 2.08;
}
```

Granite – Pixel Shader (PS 2.0 Model) 2/3

```
float4 stone(float seed)
{
    float4 color1 = float4(0.68, 0.55, 0.5, 1.0);
    float4 color2 = float4(0.6, 0.47, 0.4, 1.0);
    float4 color3 = float4(0.55, 0.45, 0.45, 1.0);
    float4 color4 = float4(0.1, 0.1, 0.1, 1.0);
    float b1 = 0.2;
    float b2 = 0.3;

    if (seed < b1)
    {
        return lerp(color1, color2, smoothstep(0, b1, seed));
    }
    else if (seed < b2)
    {
        return lerp(color2, color3, smoothstep(b1, b2, seed));
    }
    else
    {
        return lerp(color3, color4, smoothstep(b2, 1.0, seed));
    }
}
```

Granite – Pixel Shader (PS 2.0 Model) 3/3

```
float4 main( float3 P: TEXCOORD0,
             float3 Neye: TEXCOORD1,
             float3 Peye: TEXCOORD2 ) : COLOR
{
    float3 N = normalize(Neye);
    float3 L = normalize(light_pos - Peye);
    float3 V = -normalize(Peye);
    float3 H = normalize(V + L);

    float4 base = stone(saturate(noise(P * freq) * 4 - 1.5));

    float4 Cd = saturate(dot(N, L)) * Kd * base;
    float4 Cs = pow(dot(N, H), power) * Ks;

    return saturate((Cd + Ka) * base + Cs);
}
```

HLSL Shader Optimizations

- **Vectorize if possible**
  - **Use swizzles when necessary**
- **Use proper types**
  - **Use float, float3, float4 as appropriate**
- **Use tex1D for 1D textures**
- **Use intrinsic functions**
- **Don't use dot() to extract vector components**
  - **Use swizzles instead**

Go Forth and Shade!

- **Questions?**