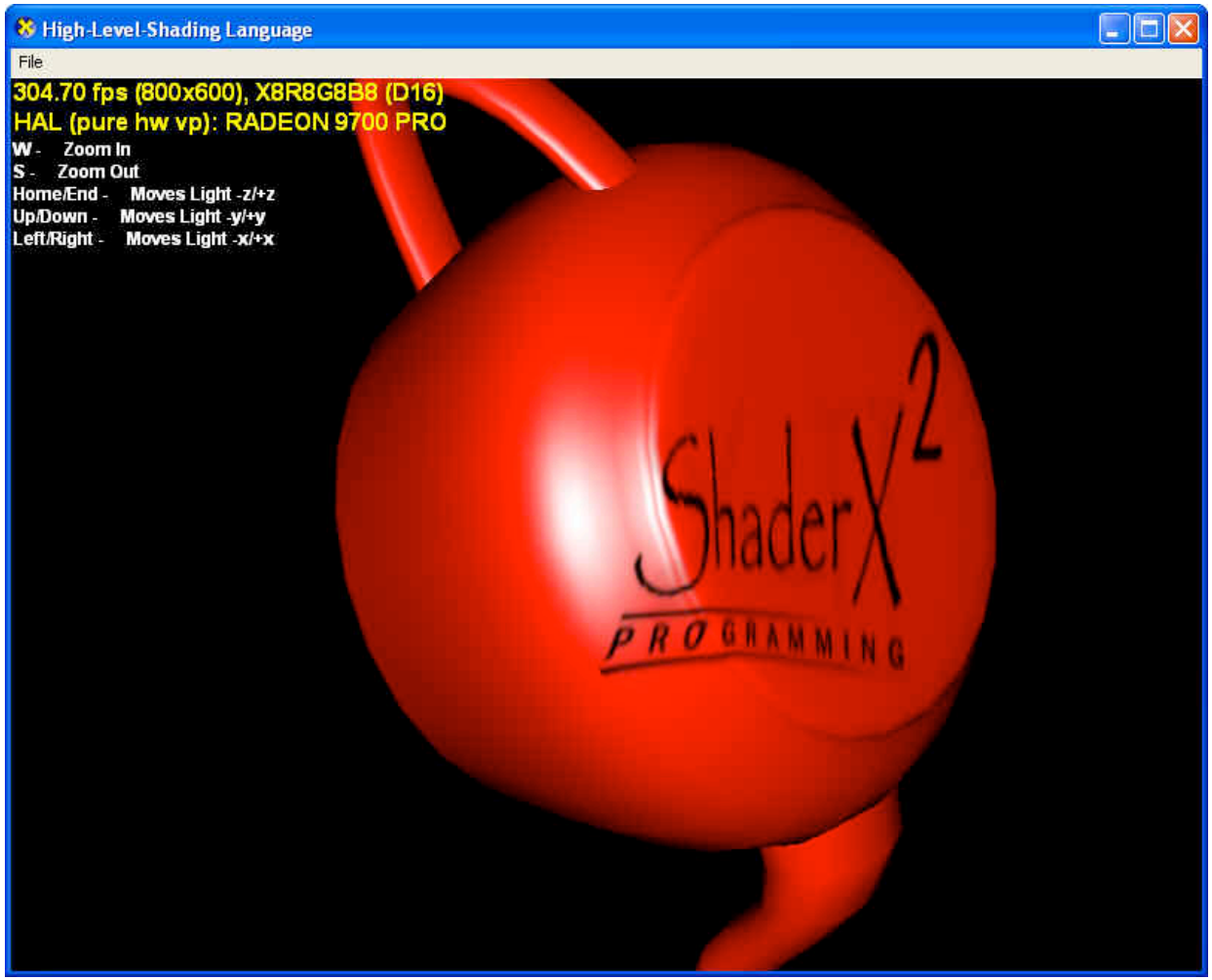
A HLSL Primer for Developers

By Wolfgang F. Engel (wolf@shaderx.com)

June 13th, 2003

Agenda

- HLSL by example: projective texturing
- Two tricks
  - Simulating blending operations on floating point render targets
  - Normal Map Compression


High-Level-Shading Language
File
304.70 fps (800x600), X8R8G8B8 (D16)
HAL (pure hw vp): RADEON 9700 PRO
W - Zoom In
S - Zoom Out
Home/End - Moves Light -z/+z
Up/Down - Moves Light -y/+y
Left/Right - Moves Light -x/+x
ShaderX2
PROGRAMMING

Introduction

- High-Level Shader Language (HLSL) is a language for programming GPU‘s
- Looks like **C**
- Example vertex and pixel shader for **projective texturing** (texture should appear to be projected onto the scene, as if from a slide projector)

```
struct VS_OUTPUTPROJTEX   // output structure
{
    float4 Pos : POSITION;
    float4 Tex : TEXCOORD0;
};

VS_OUTPUTPROJTEX VSProjTexture(float4 Pos : POSITION, float3 Normal : NORMAL)
{
    VS_OUTPUTPROJTEX Out = (VS_OUTPUTPROJTEX)0;
    Out.Pos = mul(Pos, matWorldViewProj);  // transform Position
    Out.Tex = mul(ProjTextureMatrix, Pos); // project texture coordinates

   return Out;
}

float4 PSProjTexture(float4 Tex: TEXCOORD0) : COLOR
{
   return tex2Dproj(ProjTexMapSampler, Tex);
}
```

Vertex Shader Explanation

```
struct VS_OUTPUTPROJTEX   // output structure
{
    float4 Pos : POSITION;
    float4 Tex : TEXCOORD0;
};
```

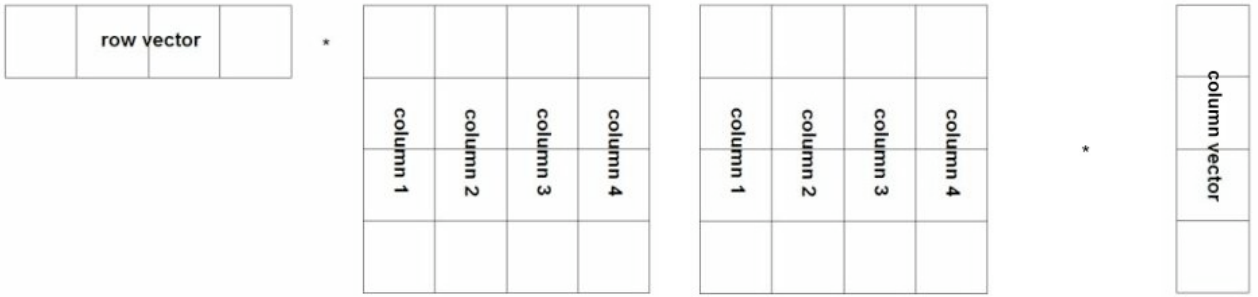
- **Output structure**: vertex shader outputs a position value and a texture coordinate to the pixel shader
- **Semantics** (f.e. : POSITION or : TEXCOORD0)
  - capital keyword + preceded by a colon (:)
  - Links shader input to the output of the previous stage of the graphics pipeline
  - In other words: helps the HLSL compiler to bind the data to the right hardware registers
  - Different Input and output semantics for vertex and pixel shaders
- **Data types**:
  - bool, float, float2, float3, float4, float4x4 (matrix) etc.
  - int (32-bit integer), half (16-bit float), double (64-bit float) (emulated if not supported)
  - No string type

Vertex Shader Explanation II

```
VS_OUTPUTPROJTEX VSProjTexture(float4 Pos : POSITION)
{
    VS_OUTPUTPROJTEX Out = (VS_OUTPUTPROJTEX)0;
    Out.Pos = mul(Pos, matWorldViewProj);  // transform Position
    Out.Tex = mul(ProjTextureMatrix, Pos); // project texture coordinates

   return Out;
}
```

- **Math Intrinsic functions** (converted to instructions by the HLSL compiler)
  - normalize(v) returns normalized vector v/length(v)
  - pow(x, y) returns x^y
  - saturate(x) clamps x to 0..1
  - mul(a, b) multiplies row-vector * matrix or matrix * column-vector (might help to save a transpose)



  - and many more
  - Math intrinsic functions work in vertex and pixel shaders

Pixel Shader

```
float4 PSProjTexture(float4 Tex: TEXCOORD0) : COLOR
{
    return tex2Dproj(ProjTexMapSampler, Tex);
}
```

- **Texture sampling intrinsics**: 16 tex* for 1D, 2D, 3D and cube map textures
  tex2Dproj(s, t): 2D projective texture lookup by dividing *t* (4D texture coordinate) through its last component
- **Output value:** float4 : COLOR
  Rendering into a multiple render target:

```
struct PS_OUTPUTPROJTEX
{
    float4 Col : COLOR;
    float4 Col1 : COLOR1;
};

PS_OUTPUTPROJTEX PSProjTexture(float4 Tex: TEXCOORD0)
{
    PS_OUTPUTPROJTEX Out = (PS_OUTPUTPROJTEX) 0;
    Out.Col =  tex2Dproj(ProjTexMapSampler, Tex);
    Out.Col1 =  tex2Dproj(ProjTexMapSampler, Tex);

    return Out;
}
```

Projective Texturing Issues

There are two notable issues with projective texturing

- **Back-projection artifacts**: projects texture on surfaces behind the projector
  Reason: undefined negative results from texture interpolators
  Solution: `Tex.w < 0.0 ? 0.0 : tex2Dproj(ProjTexMapSampler, Tex)`
- projective texture is applied to every triangle (front and back) that is within the projectors frustrum
  Reason: **No occlusion checks**
  Solution #1: Restrict projection to front facing triangles by comparing a dot product between the projectors direction and the normal to 0.0.

```
VS_OUTPUTPROJTEX VSProjTexture(float4 Pos : POSITION, float3 Normal : NORMAL)
{
    VS_OUTPUTPROJTEX Out = (VS_OUTPUTPROJTEX)0;
    Out.Pos = mul(Pos, matWorldViewProj);          // transform Position
    Out.Tex = mul(ProjTextureMatrix, Pos);         // project texture coordinates

    float4 PosWorld = normalize(mul(Pos, matWorld));        // vertex in world space
    Out.Norm = normalize(mul(Normal, matWorld));            // normal in world space
    Out.Proj = PosWorld - normalize(vecProjDir);       // projection vector in world space

   return Out;
}
float4 PSProjTexture(float4 Norm : TEXCOORD0, float4 Tex: TEXCOORD1, float4 Proj : TEXCOORD2)
    : COLOR
{
    if (dot(Proj, Norm) <= 0.0)
      return Tex.w < 0.0 ? 0.0 : tex2Dproj(ProjTexMapSampler, Tex);
    else
      return 1.0;
}
```

- **Solution #2**: **Depth Map** to clamp projection

Demo

HLSL Features from C

- HLSL provides **structures** and **arrays** incl. Multidimensional arrays
- Comments (//, /* */)
- All C‘s **arithmetic operators** (+, *, /, etc.)
- bool type with **bolean** and **relational operators** (||, &&, !, etc.)
- Increment/decrement (**++, --**)
- **Conditional expression** (? :)
- **Assigment expressions** (+=, etc.)
- C **comma** operator
- **User defined functions** (not: recursive functions)
- A subset of C‘s **flow constructs** (do, while, for, if, break, continue) Not: goto, switch
- **#define, #ifdef** etc.

Additional HLSL Features not in C

- Built-in **constructors**

```
float4 vec = float4(4.0, -2.0, 5.0, 3.0);
```

- **Swizzling**

```
float4 vec1 = float4(4.0, -2.0, 5.0, 3.0);
float2 vec2 = vec1.yx;                          // vec2 = (-2.0, 4.0)
float scalar = vec1.w;                          // scalar = 3.0
float3 vec3 = scalar.xxx;                       // vec3 = (3.0, 3.0, 3.0)
```

- **Write mask**

```
vec1.xw = vec3;                                 // vec1 = (3.0, -2.0, 5.0, 3.0)
```

- Matrices can be stored as **row_major** or **column_major**

```
row_major       half1x4 fh1By4;
column_major    half1x4 fh4By1;
row_major       half3x2 fh3By2;
column_major    half3x2 fh2By3;
```

Or as compiler pragmas

```
#pragma PACK_MATRIX (ROW_MAJOR)
#pragma PACK_MATRIX (COLUMN_MAJOR)
```

- The keyword **register** can be used to specify a specific register

```
float4 vDisplace : register (c0); // puts vDisplace constant into C0
```

- Restrict the usage of specific registers on **specific versions**

```
float4 vDisplace : register(ps_2_0, c10) // puts vDisplace into C10 for ps_2_0
```

C Features not Supported in HLSL

- No **pointers/bitwise operations**
- No **unions** and **function variables**
- No **classes**, **templates**, **operator overloading**, **exception handling**, and **namespaces**
- No **string** processing (no string type), **file i/o**, **memory allocation**
  Not the scope of HLSL

What‘s required for Compilation

- **Entry point**: name of vertex shader or pixel shader
- **Compile target**: vertex or pixel shader version

Version	Inst. Slots	Constant Count
vs_1_1	128 at least	96 cap'd (4)
vs_2_0	256	cap'd (4)
vs_2_x	256	cap'd (4)
vs_2_sw	unlimited	8192
vs_3_0	cap'd (1)	cap'd (4)
ps_1_1 - ps_1_3	12	8
ps_1_4	28 (in two phases)	8
ps_2_0	96	32
ps_2_x	cap'd (2)	32
ps_2_sw	unlimited	8192
ps_3_0	cap'd (3)	224
ps_3_sw	unlimited	8192

(1) D3DCAPS9.MaxVertexShader30InstructionSlots
(2) D3DCAPS9.D3DPSHADERCAPS2_0.NumInstructionSlots
(3) D3DCAPS9.MaxPixelShader30InstructionSlots
(4) D3DCAPS9.MaxVertexShaderConst

Example **command line**:

- fxc.exe (HLSL compiler) /T ps_2_0 (compile **T**arget) /E PSProjTexture (**E**ntrypoint) /Fc test.txt (output assembly file) $(InputName).fx (name of input file)

- **Effect file framework** helps managing shaders and helps to provide fallback pathes: Recommended !

Trick 1: Blending on Float RTs

- Trick by Francesco Carucci, „Simulating Blending Operations on Floating Point Render Targets“, ShaderX^2 – Shader Tips & Tricks, August 2003, Wordware Ltd.
- Simulating blending operations on floating point render targets
- How does it work:
  - Pass 1: Setup floating point texture as render target and then render into it
  - Pass 2:
    - send vertex position in camera space from the vertex shader through the interpolators to the pixel shader
    - transform vertex position into screen space in the pixel shader
    - Blend fp texture with the result of this pass
- Reading in the input texture in the pixel shader (Pass 2)
  - send vertex position in camera space from the vertex shader through the interpolators to the pixel shader

```
VS_OUTPUTPROJTEX VSProjTexture(float4 Pos : POSITION, float3 Normal : NORMAL)
{
   VS_OUTPUTPROJTEX Out = (VS_OUTPUTPROJTEX)0;
   float4 Position = mul(Pos, matWorldViewProj);       // transform Position
   Out.Tex = mul(ProjTextureMatrix, Pos);    // project texture coordinates
   Out.Pos = Position;

   Out.Tex2 = Position;                      // provide vertex position in camera space
  return Out;
}
```

Trick: Blending on Float RTs II

– transform vertex position into screen space in the pixel shader

```
float4 tex2DRect(sampler2D s, float4 position)          // user-defined function
{
   float2 tc;
   tc.x = (position.x / position.w) * 0.5 + 0.5;
   tc.y = (-position.y / position.w) * 0.5 + 0.5;

   return tex2D(s, tc);
}
```

– Blend values:

```
// ------------------------------------------------------------
// Pixel Shader (input channels):output channel
// ------------------------------------------------------------
float4 PSProjTexture(float4 Tex: TEXCOORD0, float4 Position : TEXCOORD1) : COLOR
{
   float4 TexProj = tex2Dproj(ProjTexMapSampler, Tex);
   float4 TexRenderMap = tex2DRect(RenderMapSampler, Position);
  return TexRenderMap * TexProj;                        // blend
}
```

Trick 1: Blending on Float RTs III

- Does not support blending modes with a destination color computed in the same pass (f.e. particle system)
- Watch out for color saturation -> map to 0..1 with a logarithmic function
- Fillrate: 64-bit or 128-bit render targets burn fillrate.

Trick 2: Normal Map Compression

- Trick by Jakub Klarowicz, „Normal Map Compression “, ShaderX^2 – Shader Tips & Tricks, August 2003, Wordware Ltd.
- Geometry details in bump maps (ATI Normal Mapper or NVIDIA Melody) need high resolution maps
  -> big chunk of memory
- How does it work:
  - Use all four color channels of DXT5 format
  - One channel of the normal map (R, G or B) is copied into alpha channel A and then it is cleared (filled with zero values) (f.e. with Adobe Photoshop)
  - The alpha channel is copied back into the color channel in the pixel shader
- Why does it work:
  - Alpha channel is quantized separately from the RGB channels
  - Alpha channel is quantized with an increased accuracy
  - Clearing one color channel leads to a higher precision in quantization of the color channels
- Pixel Shader source:

```
…
 float4 bumpNormal = 2 * (tex2D(BumpMapSampler, Tex) - 0.5); // bump map
 bumpNormal.g = bumpNormal.a;           // move alpha into green color channel
…
```

Trick 2: Normal Map Compression II

- Does not work well for **model space normal maps** as for tangent space maps -> vectors in model space normal maps vary much
- **Rough normal maps** still look bad when compressed

Trick 2: Normal Map Compression III

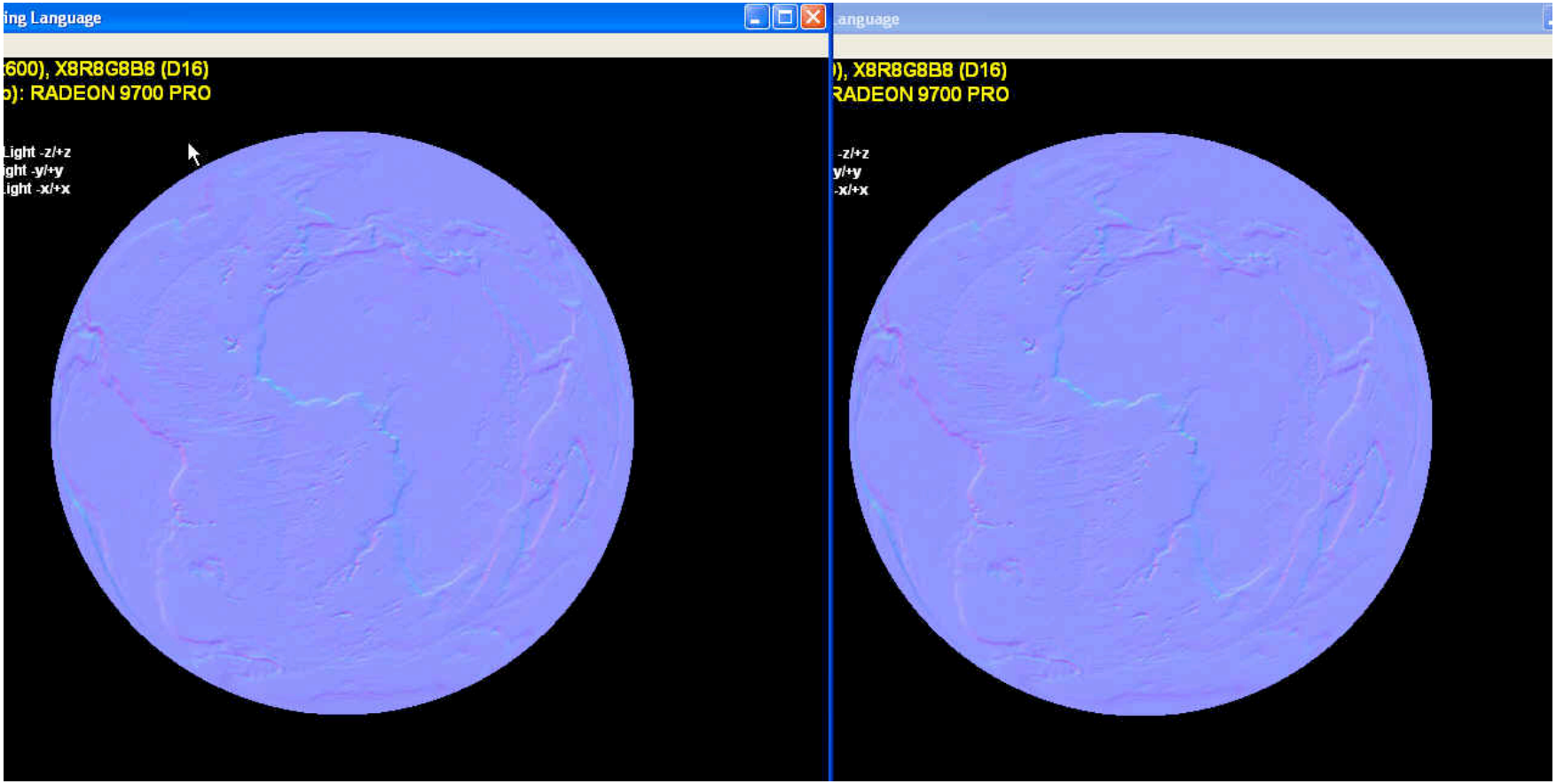


Figure: Left DXT5 compressed (2049 kb) | Right: uncompressed (8183 kb)

Wrap Up

- HLSL ~ C with modifications for Graphics
- HLSL compiler optimizes for hardware from different vendors
- Use it … don‘t invest time to learn assembly

Further Reading

- Jason L. Mitchell, Craig Peeper, „Introduction to HLSL“, ShaderX^2 – Shader Introduction & Tutorial, August 2003, Wordware
- Cass Everitt, „Projective Texturing“, NVIDIA developer web-site.

Thank you