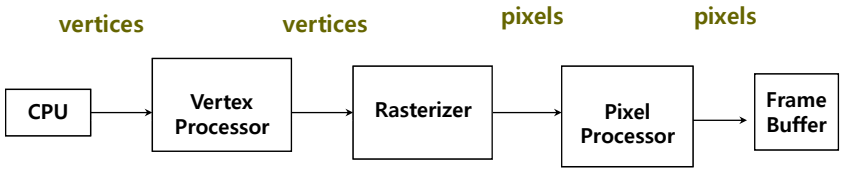
DirectX Shader
- HLSL

305890
Spring 2010
4/16/2010
Kyoung Shin Park

Overview

- Recent major advance in real time graphics is ***programmable pipeline***
  - First introduced by Nvidia GeForce3
  - Supported by high-end commodity cards
    - NVIDIA, ATI, 3D Labs
  - Software Support
    - DirectX8, 9, 10
    - OpenGL Extensions
    - OpenGL Shading Language (GLSL)
    - Cg

Shader



Vertex Processor

- Takes in vertices
  - Position attribute
  - Possibly color
  - State
- Produces
  - Position in clip coordinates
  - Vertex color

Pixel Processor

- Takes in output of rasterizer (pixels)
  - Vertex values have been interpolated over primitive by rasterizer
- Produces a pixel
  - Color
  - Textures
  - Possibly depth
  - Alpha

Sample: Transform.fx

```
//=============================================================================
//
// File: transform.fx
//
// Basic FX that simply transforms geometry from local space to
// homogeneous clip space, and draws the geometry in wireframe mode.
//
//=============================================================================

// Effect parameter (a combined world, view, and projection transformation matrix)
uniform extern float4x4 gWVP;

// Define a vertex shader output structure; that is, a structure that defines the data
// we output from the vertex shader. Here, we only output a 4D vector in homogeneous
// clip space.  The semantic ": POSITION0" tells Direct3D that the data returned in this
// data member is a vertex position.
struct OutputVS
{
    float4 posH : POSITION0;
};
```

Sample: Transform.fx

```
// Define the vertex shader program.  The parameter posL corresponds to a data
// member in the vertex structure. Specifically, it corresponds to the data member
// in the vertex structure with usage D3DDECLUSAGE_POSITION and index 0
// (as specified by the vertex declaration).
OutputVS TransformVS(float3 posL : POSITION0)
{
    // Zero out our output.
    OutputVS outVS = (OutputVS)0;

    // Transform to homogeneous clip space.
    outVS.posH = mul(float4(posL, 1.0f), gWVP);  // vector-matrix multiplication

    // Done--return the output.
    return outVS;
}
```

Sample: Transform.fx

```
// Define the pixel shader program.  Just return a 4D color vector (i.e., first component
// red, second component green, third component blue, fourth component alpha).
// Here we specify black to color the lines black.
float4 TransformPS() : COLOR
{
    return float4(0.0f, 0.0f, 0.0f, 1.0f);
}
// entry point
technique TransformTech
{
    pass P0
    {
        // Specify the vertex and pixel shader associated with this pass.
        vertexShader = compile vs_2_0 TransformVS();
        pixelShader  = compile ps_2_0 TransformPS();
        // Specify the render/device states associated with this pass.
        FillMode = Wireframe;
    }
}
```

Compiling a Shader

- Creating an Effect

```
HRESULT D3DXCreateEffectFromFile (
        LPDIRECT3DDEVICE9 pDevice,
        LPCSTR pSrcFile,
        CONST D3DXMACRO* pDefines, // set as null
        LPD3DXINCLUDE pInclude,    // set as null
        DWORD Flags,
        LPD3DXEFFECTPOOL pPool,
        LPD3DXEFFECT* ppEffect,
        LPD3DXBUFFER *ppCompilationErrors
);
```

  - **pDevice** – device to be associated with the created ID3DXEffect obj
  - **pScrFile** – a fx file that contains the effect source code
  - **Flags** – optional compiling flag
  - **ppEffect** – returns a pointer to an ID3DXEffect interface
  - **ppCompilationErrors** – returns a pointer to an ID3DXBuffer that contains a string of error codes

Compiling a Shader

- Obtaining handles to effect parameters
  - **D3DXHANDLE ID3DXEffect::GetParameterByName**

```
D3DXHANDLE ID3DXEffect::GetParameterByName (
        D3DXHANDLE hParent,
        LPCSTR pName
);
```

  - **hParent** – scope of variable – parent structure
  - **pName** – name of variable

Compiling a Shader

- Setting effect parameters
  - **ID3DXEffect::SetXXX** method

```
HRESULT ID3DXEffect::SetXXX (
        D3DXHANDLE hParameter,
        XXX value
);
```

  - **hParameter** – a handle to parameter
  - **value** – value

- Getting effect parameters
  - **ID3DXEffect::GetXXX** method

Set Effect Parameters

SetBool	bool b = true; SetBool(handle, b);
SetBoolArray	bool b[3] = {true, false, true}; SetBoolArray(handle, b, 3);
SetFloat	float f = 3.14f; SetFloat(handle, f);
SetFloatArray	float f[2] = {1.0f, 2.0f}; SetFloatArray(handle, f, 2);
SetInt	int x = 4; SetInt(handle, x);
SetIntArray	int x[4] = {1, 2, 3, 4} SetIntArray(handle, x, 4);
SetMatrix	D3DXMATRIX M(...); SetMatrix(handle, &M);
SetMatrixArray	D3DXMATRIX M[4]; // ... Initialize matrices SetMatrixArray(handle, M, 4);
SetMatrixPointerArray	D3DXMATRIX *M[4]; // ... Allocate and initialize matrix pointers SetMatrixPointerArray(handle, M, 4);

Set Effect Parameters

SetMatrixTranspose

```
D3DXMATRIX M(...);
D3DXMatrixTranspose(&M, &M);
SetMatrixTranspose(handle, &M);
```

SetMatrixTransposeArray

```
D3DXMATRIX M[4];
// ... Initialize matrices and transpose them
SetMatrixTransposeArray(handle, M, 4);
```

SetMatrixTransposePointerArray

```
D3DXMATRIX *M[4];
// ... Allocate, initialize matrix pointers and transpose them
SetMatrixTransposePointerArray(handle, M, 4);
```

SetVector

```
D3DXVECTOR4 v(1.0f, 2.0f, 3.0f, 4.0f);
SetVector(handle, &v);
```

SetVectorArray

```
D3DXVECTOR4 v[3];
// ... Initialize vectors
SetVectorArray(handle, v, 3);
```

SetValue

```
D3DXMATRIX M(...);
SetValue(handle, (void*)&M, sizeof(M));
```

Compiling a Shader

- Begin/End an effect
  - **ID3DXEffect::Begin(UINT * pPasses, DWORD flags)**

```
HRESULT ID3DXEffect::Begin(
        UINT* pPasses, DWORD flags
);
```

    - **pPasses** – returns the number of passes in the currently active technique
    - **flags** – 0 (instructs the effect to save the current device states)

  - **ID3DXEffect::End()**

Sample: Check for Shader Support

```
//
// vertify if the graphics card running the demo supports the vertex and pixel shader version we use
//
bool MeshDemo::checkDeviceCaps()
{
    D3DCAPS9 caps;
    HR(gd3dDevice->GetDeviceCaps(&caps));

    // Check for vertex shader version 2.0 support.
    if( caps.VertexShaderVersion < D3DVS_VERSION(2, 0) )
        return false;

    // Check for pixel shader version 2.0 support.
    if( caps.PixelShaderVersion < D3DPS_VERSION(2, 0) )
        return false;

    return true;
}
```

Sample: Create an Effect & Obtain Handles

```
//
// Create an effect & Obtain handles
//
ID3DXEffect * mFX = 0;

Void MeshDemo::buildFx()
{
    // create the FX from a .fx file
    ID3DXBuffer * errors      = 0;
    HR(D3DXCreateEffectFromFile(
        gd3dDevice, "transform.fx",
        0, 0, D3DXSHADER_DEBUG, 0, &mFX, &errors));

    // output any error messages
    if( errors )
        MessageBox(0, (char*)errors->GetBufferPointer(), 0, 0);

    // obtain a handle to an effect
    mhTech = mFx->GetTechniqueByName("TransformTech");

    // obtain a handle to effect parameters
    mhWVP = mFx->GetParameterByName(0, "gWVP");
}
```

Sample: Activate an Effect & Set Effect Parameters

```
void MeshDemo::drawScene()
{
    HR(gd3dDevice->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, 0xffffffff, 1.0f, 0));
    HR(gd3dDevice->BeginScene());
    // 중간 생략 ...

    // set up the rendering FX
    HR(mFX->SetTechnique(mhTech));
    // Beding passes
    UINT numPasses = 0;
    HR(mFX->Begin(&numPasses, 0));
    for (UINT i = 0; I < numPasses; i++) {
        HR(mFX->BeginPass(i));
        HR(mFX->SetMatrix(mhWVP, &(mView*mProj)));
        HR(mFX->CommitChanges());      // for change to take effect
        HR(gd3dDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0, mNumGridVertices,
                                            0, mNumGridTriangles);
        drawCylinders();
        drawSpheres();
        HR(mFX->EndPass());
    }
    HR(mFX->End());

    HR(gd3dDevice->EndScene());
    HR(gd3dDevice->Present(0, 0, 0, 0));
}
```

Variable Types

- Scalar
- Vector
- Matrix
- Arrays
- Structures
- **typedef** keyword
- Variable Prefixes

Scalar

- **bool** – boolean
- **int** – 32-bit integer
- **half** – 16-bit floating point
- **float** – 32-bit floating point
- **double** – 64-bit floating point

Vector

- **vector** – **float** type 4D vector
- **vector<T,n>** – **T** type **nD** vector
  - **n** – 1~4 integer
  - **T** – scalar type
  - E.g. **vector<double,2> vec2;**
- Vector element access

```
vec[i] = 2.0f;
vec.x = vec.r = 1.0f;
vec.y = vec.g = 2.0f;
vec.z = vec.b = 3.0f;
vec.w = vec.a = 4.0f;
```

Vector

- Pre-defined vector type

```
float2 vec2;
float3 vec3;
float4 vec4;
```

- Vector "copy" operation
  - *swizzles* (순서에 구애 받지 않고 복사를 수행)

```
vector u1 = {1.0f, 2.0f, 3.0f, 4.0f};
vector v1 = {0.0f, 0.0f, 5.0f, 6.0f};

v1 = u1.xyyw;    // v1 = {1.0f, 2.0f, 2.0f, 4.0f};

vector u2 = {1.0f, 2.0f, 3.0f, 4.0f};
vector v2 = {0.0f, 0.0f, 5.0f, 6.0f};

v2.xy = u2;      // v2 = {1.0f, 2.0f, 5.0f, 6.0f};
```

Matrix

- **matrix** – 4x4 **float** type matrix
- **matrix<T,m,n>** – **m** x **n** **T** type matrix
  - **m**, **n** – 1~4 integer
  - **T** – scalar type
  - e.g. `matrix<int,2,2> m2x2;`
- Other methods to define **m** x **n** matrix

```
float2x2 mat2x2;
float3x3 mat3x3;
float4x4 mat4x4;
float2x4 mat2x4;

int2x2 i2x2;
int3x3 i3x3;
int4x4 i4x4;
```

Matrix

- Matrix element access
  - ij^{th} element : `M[i][j] = value;`
  - 1-based matrix :

```
M._11 = M._12 = M._13 = M._14 = 0.0f;
M._21 = M._22 = M._23 = M._24 = 0.0f;
M._31 = M._32 = M._33 = M._34 = 0.0f;
M._41 = M._42 = M._43 = M._44 = 0.0f;
```

  - 0-based matrix :

```
M._m00 = M._m01 = M._m02 = M._m03 = 0.0f;
M._m10 = M._m11 = M._m12 = M._m13 = 0.0f;
M._m20 = M._m21 = M._m22 = M._m23 = 0.0f;
M._m30 = M._m31 = M._m32 = M._m33 = 0.0f;
```

  - i^{th} -row vector :

```
vector ithRow = M[i];
```

Scalar, Vector, Matrix Initialization

- Vector

```
vector u = {0.6f, 0.3f, 1.0f, 1.0f};
vector v = {1.0f, 5.0f, 0.2f, 1.0f};

vector u = vector(0.6f, 0.3f, 1.0f, 1.0f);
vector v = vector(1.0f, 5.0f, 0.2f, 1.0f);
```

- Matrix

```
float2x2 f2x2 = float2x2(1.0f, 2.0f, 3.0f, 4.0f);
int2x2 m = {1, 2, 3, 4};
```

- Scalar

```
int n = int(5);
int a = {5};
float3 x = float3(0.0f, 0.0f, 0.0f);
```

Array and Structure

- Array (C-style)

```
float M[4][4];
half p[4];
vector v[12];
```

- Structure (C-style)

```
struct MyStruct {
    matrix T;
    vector n;
    float f;
    int x;
    bool b;
};


MyStruct s;       // instantiate
s.f = 5.0f;       // member access
```

typedef keyword and Variable Prefixes

- **typedef** keyword (C/C++-style)

```
typedef vector<float,3> point;
typedef const float CFLOAT;
typedef float point2[2];
```

  - **vector<float,3> myPoint;** => **point myPoint;**
- Variable prefixes

static	- 전역 변수: 셰이더 외부에서 변수 접근할 수 없음 - 지역 변수: C++에서의 정적 지역 변수와 같은 의미
uniform	셰이더 외부(응용 프로그램)에서 초기화됨
extern	셰이더 외부에서 변수 접근 가능 (static이 아닌 전역 변수는 디폴트로 extern)
shared	다수의 효과에 공유될 변수 (→ 이펙트 프레임웍 chapter 19)
volatile	자주 수정될 변수 (→ 이펙트 프레임웍 chapter 19)
const	C++에서와 같은 의미

Keyword

- Keywords in HLSL

asm	bool	compile	const	decl	do
double	else	extern	false	float	for
half	if	in	inline	inout	int
matrix	out	pass	pixelshader	return	sampler
shared	static	string	struct	technique	texture
true	typedef	uniform	vector	vertexshader	void
volatile	while				

- Other reserved keywords

auto	break	case	catch	char	class
const_cast	continue	default	delete	dynamic_cast	enum
explicit	friend	goto	long	mutable	namespace
new	operator	private	protected	public	register
reinterpret_cast	short	signed	sizeof	static_cast	switch
template	this	throw	try	typename	union
unsigned	using	virtual			

Statements

- Statement (C/C++-style)
  - **return** :

```
return (expression);
```

  - **if / if...else** :

```
if( condition )
{
    statement(s);
}
```

```
if( condition )
{
    statement(s);
}
else
{
    statement(s);
}
```

  - **for** :

```
for(initial; condition; increment)
{
    statement(s);
}
```

  - **while / do...while** :

```
while( condition )
{
    statement(s);
}
```

```
do
{
    statement(s);
}while( condition );
```

Type Casting

- Type casting
  - C/C++-style
  - **float f = 5.0f;**
    matrix m = (matrix)f;
- Reference: DirectX SDK document

Operators

- Operators (C/C++-style)

[]	.	>	<	<=	>=
!=	==	!	&&	!!	?:
+	+=	-	-=	*	*=
/	/=	%	%=	++	--
=	()	,			

- % operator
  - Can be used in both integer and floating point value
  - Must have the same positive/negative in left and right operands (e.g., -5 % -2 or 5%2)

Operators

- Vector & Matrix operator overloading

```
vector u = { 1.0f, 0.0f, -3.0f, 1.0f};
vector v = {-4.0f, 2.0f,  1.0f, 0.0f};

vector sum = u + v;       // sum=(-3.0f,2.0f,-2.0f,1.0f)
sum++;                    // sum=(-2.0f,3.0f,-1.0f,2.0f)

vector u1 = { 1.0f, 0.0f, -3.0f, 1.0f};
vector v1 = {-4.0f, 2.0f,  1.0f, 0.0f};

vector p = u1 * v1;       // p=(-4.0f,0.0f,-3.0f,0.0f)
vector u2 = { 1.0f, 0.0f, -3.0f, 1.0f};
vector v2 = {-4.0f, 0.0f,  1.0f, 1.0f};

vector b = (u2 == v2);    // b=(false,true,false,true)
```

Operators

- Type upcasting
  - Left and right operands are different types,
    - **int x; half y; (x + y);**
      ➔ **int x** upcast to **half** type
  - Left and right operands size are different,
    - **float x; float3 y; (x + y);**
      ➔ **float x** upcast to **float3** type
  - When type casting is not defined,
    - **float2** cannot be upcast to **float3**

User-defined Functions

- HLSL functions
  - C/C++-style
  - Call-by-value parameter
  - No recursive calls
  - "inline" functions
- Keyword **in**, **out**, **inout**

```
bool foo(in const bool b, out int r1, inout float r2)
{
   if( b )
   {
         r1 = 5;
   }
   else
   {
         r1 = 1;
   }
   r2 = r2 * r2 * r2;
   return true;
}
```

User-defined Functions

- Keyword **in**, **out**, **inout**
  - **in** – when the functon is entered, in parameter is copied to the parameter (default)

```
float square(in float x)
{
   return x * x;
}
```

equal

```
float square(float x)
{
   return x * x;
}
```

  - **out** – when it returns, it copys parameter to out parameter

```
void square(in float x, out float y)
{
   y = x * x;
}
```

  - **inout** – can be used in both **in** and **out**

```
void square(inout float x)
{
   x = x * x;
}
```

Predefined Functions

- HLSL predefined functions

abs(x)	ceil(x)	clamp(x,a,b)	cos(x)
cross(u,v)	degrees(x)	determinant(M)	distance(u,v)
dot(u,v)	floor(x)	length(v)	lerp(u,v,t)
log(x)	log10(x)	log2(x)	max(x,y)
min(x,y)	mul(M,N)	normalize(v)	pow(b,n)
radians(x)	reflect(v,n)	refract(v,n,eta)	rsqrt(x)
saturate(x)	sin(x)	sincos(in x,out s,out c)	sqrt(x)
tan(x)	transpose(M)		

  - Function Overloading
    - **abs**(x) – function overloading for all scalar types
    - **cross**(u,v) – function overloading for all 3D vector
    - **lerp**(u,v,t) – function overloading for all scalars and 2D, 3D, 4D vector

Predefined Functions

```
float x = sin(1.0f);          // sine of 1.0f radian
float y = sqrt(4.0f);         // square root of 4

vector u = {1.0f,  2.0f, -3.0f, 0.0f};
vector v = {3.0f, -1.0f,  0.0f, 2.0f};
float s = dot(u,v);           // dot product of u and v

float3 i = {1.0f, 0.0f, 0.0f};
float3 j = {0.0f, 1.0f, 0.0f};
float3 k = cross(i,j);        // cross product of u and v

matrix<float,2,2> M = {1.0f, 2.0f, 3.0f, 4.0f};
matrix<float,2,2> T = transpose(M);      // transpose of M
```

```
float3 v = {0.0f, 0.0f, 0.0f};
v = cos(v);       // v = (cos(v.x),cos(v.y),cos(v.z))
```