HLSL Shaders

Drikus Kleefsman

February 16, 2010

Abstract

Keywords: shader, lighting, FX Composer, HLSL, Effect file, pixel shader, vertex shader, sass, semantics

An Effectfile contains a vertex shader and a pixel shader. Both are programs written for a GPU (Graphics Processing Unit). Here we present some short examples to get started using FX Composer as a tool.

1 Introduction

An Effectfile contains a vertex shader and a pixel shader. Both are programs written for a GPU (Graphics Processing Unit). The program using a shader has to fill a vertexbuffer and a buffer that tells which vertices make primitives like points, lines and triangles. The vertices in the vertexbuffer normally contain also data as color, normal and texture coordinates. Also some global variables for the shader programs must be set by the program using it. The WorldViewProjection matrix is almost always a global variable that has to be set. But also the EyePosition will be necessary if you want good lighting and shading. The vertex shader has as input some globals and the vertex buffer. The output will normally contain the computed coordinates of the vertices, of normals and also color and texture information will be passed. The next step on the graphics processor will be the construction of primitives and the rasterization. The output of this proces is a large number of fragements (having a pixel-position). And for each of the fragments the pixel shader will run.

Here we present some short examples to get started using FX Composer as a tool.

Section 2 describes the features of Fx Composer (NVdia). Section 3 is a first example of an effect file. Section 4 describes a simple effect for lighting (alleen ambient + diffuse). Section 5 describes how to use a texture. Section 6 describes a simple animation.

2 Fx Composer

Fx Composer 2.5 van NVdia is a freeware programma. It has a friendly user interface for making effect files. On the right left is a panel where you can find project information and materials infromation. In the middle is an editor for creating and editing new effects and on the right side you get the possibility to apply your effects to a scene (figuur 1). On the right is also a properties window that can be use to tweak global values. The central window for editing has a tab to load models (most common formats are supported) and a tab to load existing shaders in the NVdia shader library.

A Fx Composer project contains materials. The materials do not have a file of their own, but share a file with a .dae extension. Think of a material as an effect together with the settings for the global variables of that effect.

At the NVdia site you can download a book on writing Cg effect files. It is a very good book but now a bit outdated (it is written in 2003). You will not find readings on software semantics and attributes in this book. But take a look, it is really good.

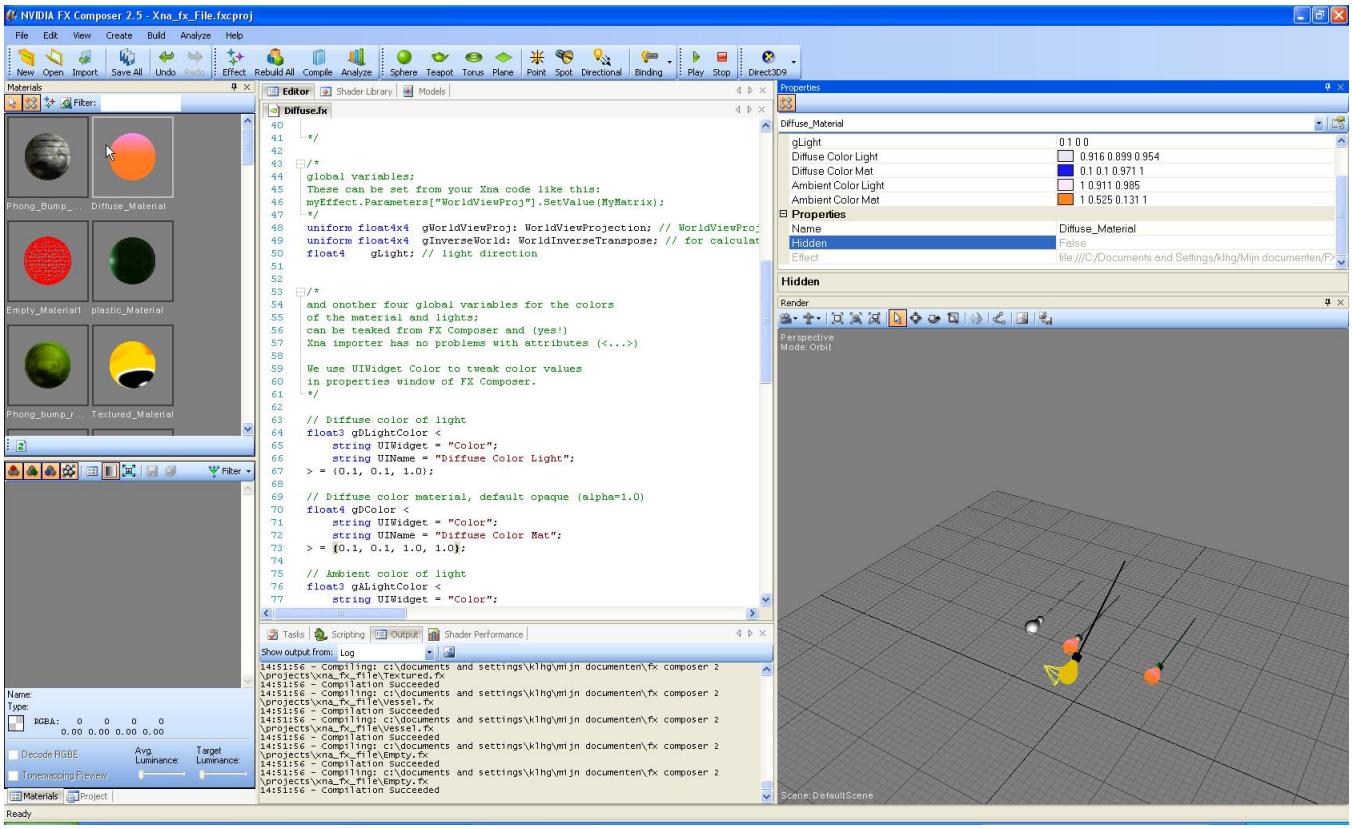

Figure 1: User interface Fx Composer 2.5

3 First example

In this first example you see a vertex shader and two pixel shaders. Fx Composer lets you choose which technique you want to use by rightclicking on the material in the Materials panel. If you are using this shader in your own program you will also have to tell which technique to use. Here one technique colors everything green and the other colors everithing yellow.

Also the use of semantics is shown. POSITION is an example of a hardware semantic. WorldViewProjection is an example of a software semantic. The expression

return mul(float4(pos.xyz, 1.0), WorldViewProj);

shows how to multiply with matrices and how to manipulate vectors using the .xyz notation. Here a new float4 is created using the xyz in the input and assigning 1.0 to the .w component.

```
/*

A first example of a shader.
Made with FX Composer 2.5.
Xna_fx_File.fxcproj.
author: Drikus Kleefsman

keywords: material classic

date: 2010-januari

*/

/*
Global variables;
These can be set from your Xna code like this:
myEffect.Parameters["WorldViewProj"].SetValue(MyMatrix);
*/
float4x4 WorldViewProj : WorldViewProjection;


/*
The only vertex shader.
Runs once for every vertex.
Input is parameter pos, a float3, with semantic POSITION.
Input-semantics are used for automatic assignment of values.
Here it is the position of the vertex for which the
shader runs.
Result is float4 with semantic POSITION. Here POSITION is
the output-semantic.
*/
float4 mainVS(float3 pos : POSITION) : POSITION {
return mul(float4(pos.xyz, 1.0), WorldViewProj);
}

/*
The first pixel shader.
```

```
Runs once for every fragment.
Output for every fragment a float4 with semantic COLOR.
red=1.0, green=1.0: results in color yellow.
*/
float4 mainYellowPS() : COLOR {
return float4(1.0, 1.0, 0.0, 1.0);
}

/*
The second pixel shader.
Output for every fragment a float4 with
output-semantic COLOR.
green=1.0: results in color green.
*/
float4 mainGreenPS() : COLOR {
return float4(0.0, 1.0, 0.0, 1.0);
}

/*
A shader can have many techniques. Here we have 2
Which technique is used can be set in your Xna code

Every Technique can have many passes, here there is
only 1 pass in every technique.

VertexShader and PixelShader are defined in HLSL and
musted be spelled exactly this way. In the first
technique the function mainVS() defined above
is compiled against the 3.0 specification and in
the second technique against the 1.1 specification.
*/
technique techniqueYellow {
pass p0 {
CullMode = None;
VertexShader = compile vs_3_0 mainVS();
PixelShader = compile ps_3_0 mainYellowPS();
}
}

technique techniqueGreen {
pass p0 {
CullMode = None;
VertexShader = compile vs_1_1 mainVS();
PixelShader = compile ps_2_0 mainGreenPS();
}
}
```

4 Ambient and diffuse lighting

The next example shows the use of attributes. Fx Composer can use the software semantics and the attributes that can be found in the DirectX Sass specification (in Fx Composer: look at Help — DirectX Sas guide). An

attribute can be used to make it possible to set global variables in the properties window.

This time there is also the software semantic WorldInverseTranspose. This is used to transform normals. A normal will have a different transform than World if World is not a rotation. If some scaling is also part of the transformation you will have to use this matrix. By writing

```
float4 gDColor <
    string UIWidget = "Color";
    string UIName = "Diffuse Color Mat";
> = {0.1, 0.1, 1.0, 1.0};
```

you let a new property Diffuse Color Mat show up in the properties window and you can adjust the value using a color-widget. In the property window the other globals like gWorldViewProj, WorldInverseTranspose will appera too. If you do'nt want that use the attribute

```
string UIWidget = "None";
```

. The general syntax for a variable declaration is:

```
type name : semantic <annotation_list> = value ;
```

.

```
/*

Ambient and Diffuse light
Made with FX Composer 2.5.
Xna_fx_File.fxcproj.
author: Drikus Kleefsman

in this file the following lighting model is used

1. I(ntensity) = IAmbient + IDuffuse;

// AColor = ambient color of material
// ALightColor = ambientcolor of light
2. IAmbient = AmbientLightColor * aColor;

// DColor = diffuse color of material
// DColorLight = diffuse color of light
// L = direction Light
// N = normal on material
// N.L inproduct of N and L
3. IDiffuse = DiffuseLightColor * (DColor * N.L);

it is assumed that the material has a fixed color and intensity
(no change in input vertex shader of colors)

the algorithm is similar to Lambert shading. Other shading algorithms:
Gouraud
```

```
Phong
Blinn
Oren-Nayar
Cook-Torrance
Ward anisotropic
Retrieved from "http://en.wikipedia.org/wiki/List_of_common_shading_algorithms"



keywords: material classic Lambert

date: 19 januari 2010

*/

/*
global variables;
These can be set from your Xna code like this:
myEffect.Parameters["WorldViewProj"].SetValue(MyMatrix);
*/
uniform float4x4  gWorldViewProj: WorldViewProjection; // WorldViewProjection FX semantic
uniform float4x4  gInverseWorld: WorldInverseTranspose; // for calculating normals
float4    gLight; // light direction


/*
and onother four global variables for the colors
of the material and lights;
can be teaked from FX Composer and (yes!)
Xna importer has no problems with attributes (<...>)

We use UIWidget Color to tweak color values
in properties window of FX Composer.
*/

// Diffuse color of light
float3 gDLightColor <
    string UIWidget = "Color";
    string UIName = "Diffuse Color Light";
> = {0.1, 0.1, 1.0};

// Diffuse color material, default opaque (alpha=1.0)
float4 gDColor <
    string UIWidget = "Color";
    string UIName = "Diffuse Color Mat";
> = {0.1, 0.1, 1.0, 1.0};

// Ambient color of light
float3 gALightColor <
    string UIWidget = "Color";
    string UIName = "Ambient Color Light";
> = {1.0, 0.1, 0.1};
```

```
// Ambient color material, default opaque (alpha=1.0)
float4 gAColor <
    string UIWidget = "Color";
    string UIName = "Ambient Color Mat";
> = {1.0, 0.1, 0.1, 1.0};

/*
Here we use an OUT structure. It holds the values that
the Vertex shader outputs to the Pixel shader.
*/
struct OUT {
float4 Pos:  POSITION;  // stores transformed position
float3 L:    TEXCOORD0; // stores normalized light direction
float3 N:    TEXCOORD1; // stores transformed normal
};


/*
The vertex shader has the name mainVS.
It expects as input a float3 position an a float3 normal.
*/
OUT mainVS(float3 pos : POSITION, float3 N: NORMAL ) {
OUT Out = (OUT) 0;
Out.Pos = mul( float4(pos.xyz, 1.0) , gWorldViewProj);

// in case L is not normalized
Out.L = normalize(gLight);

// transform normals with InverseWorld
// normalize them
Out.N = normalize(mul(N, gInverseWorld));

return Out;
}



/*
The pixel shader has the name mainPS.
The input are two normalised values in TEXCOORD0 an TEXCOORD1.
*/
float4 mainPS(float3 L: TEXCOORD0, float3 N: TEXCOORD1) : COLOR {
//saturate clamps values within range [0,1]
return  float4(gALightColor * gAColor.rgb, gAColor.a) +
float4(gDLightColor * (gDColor * saturate(dot(L, N))).rgb, gDColor.a);
}

/*
Give name to technique.
Use vs_2_0 and ps_2_0, no fancy things happen
in this shader, just basics.
```

```
px_1_x no longer supported in FX Composer
*/
technique DiffuseLight
{
pass P0
{
VertexShader = compile vs_2_0 mainVS();
PixelShader = compile ps_2_0 mainPS();
}
}
```

5 Using a texture

The next example uses a texture. There is a type texture to introduce a texture in your program. Access to the texture is by means of a sampler. In the pixel shader the function tex2D is used to get the color at the given texture coordinates.

```
/*

Ambient and Diffuse light
Made with FX Composer 2.5.
project: Xna_fx_File.fxcproj.
effect : Textured.fx
author : Drikus Kleefsman

in this file a texture is applied to an object



keywords: material classic texture

date: 19 januari 2010

*/

/*
global variables;
These can be set from your Xna code like this:
myEffect.Parameters["WorldViewProj"].SetValue(MyMatrix);
*/
float4x4  gWorldViewProjection: WorldViewProjection;

/*
a texture and a corresponding sampler.
*/
texture gTexture<
    string ResourceName = "default_bump_normal.dds";
    string UIName =  "Nice Texture";
    string ResourceType = "2D";
>;
```

```
sampler gColoredTextureSampler = sampler_state {
texture = <gTexture> ;
magfilter = LINEAR;
minfilter = LINEAR;
mipfilter=LINEAR;
AddressU = mirror;
AddressV = mirror;
};

/*
the input and output stuctures for the vertex shader.
The input gets for every vertex a u,v pair as texture coordinates
The vertex shader passes the u,v values to the pixel shader
*/
struct VertexIn {
float4 position : POSITION;
float2 textureCoordinates : TEXCOORD0;
};
struct VertexOut {
float4 Position : POSITION;
float2 textureCoordinates : TEXCOORD0;
};

/*
the vertex shader
*/
VertexOut mainVS(VertexIn input) {
VertexOut output = (VertexOut)0;

output.Position = mul(input.position, gWorldViewProjection);
output.textureCoordinates = input.textureCoordinates;
return output;
}

/*
the pixel shader.
uses the function tex2D to lookup the color at (u,v) in the texture.
*/
float4 mainPS(float2 textureCoordinates: TEXCOORD0) : COLOR {
float4 output = tex2D(gColoredTextureSampler, textureCoordinates);
return output;
}


technique textured {
pass p0 {
CullMode = None;
VertexShader = compile vs_2_0 mainVS();
PixelShader = compile ps_2_0 mainPS();
}
}
```

6 An animation

In Fx Composer introduce a new effect based on Phong shading. This will serve as a basis for creating an animation. To make the playback of an animation possible add the following not tweakable global variable with software semantic TIME. Also introduce the globals gFrequency and gScaleFactor. I have put these variables between the ViewInverse matrix and Point Lamp 0:

```
float4x4 ViewIXf : ViewInverse < string UIWidget="None"; >;


float gTime : TIME < string UIWidget = "None"; >;


//// TWEAKABLE PARAMETERS ////////////////////

/// Animation /////////

// pulse frequency
float gFrequency = 1.0f;
float gScaleFactor = 0.0f;


/// Point Lamp 0 ////////////
```

Then add two extra lines to the vertex shader:

```
  float4 Po = float4(IN.Position.xyz,1);

  Po = Po + 0.5 * gScaleFactor * (sin(gTime)+1) * IN.Normal;
  Po = float4(Po.xyz, 1);

  float3 Pw = mul(Po,WorldXf).xyz;
```

The position in object space gets an extra displacement in the direction of the normal. gTime is the representation of time and so sin(gTime)+1 varies between 0 an 2. In the properties window set the values for gFrequency and gScaleFactor. Apply this material to a model and press the Play button in the user interface. You will see the object pulsating.

Do'nt forget the second line (Po = float4(Po.xyz, 1);) because in the first line also the w-component of Po will be multiplied giving an unexpected result.

You can also add particles to your scene. Go to the assets window (if not visible click on View — Assets). Right-click Particle emitters and add your favorite psrticle system. Drag it to the render window. In the editor you will see the code for your effect. Again press Play to see your effect. Many properties can be set by clicking on the particle emitter in the asset window.

7 Next

There are many nice examples in the effect library. Load them from Fx Composer and look at the effect code. And you have a User Guide at your disposal. See: Help — User Guide. It has info on visual styles, the example projects in the install and on scripting in Python. Look in the Sas guide (Help — DirectX Sas Guide) if you want more information on scripting. Scripting gives you the power to use more passes in a shaders Technique. It uses an annotation Script in a technique as in the following example:

```
technique Main <

string Script =

"Pass=MakeShadow;"

"Pass=HBlur;"

"Pass=VBlur;"

"RenderColorTarget0=;"

"RenderDepthStencilTarget=;"

"ClearSetColor=gClearColor;"

"ClearSetDepth=gClearDepth;"

"Clear=Color;"

"Clear=Depth;"

"Pass=useBakedLighting;";

> {

// rest of technique and pass declarations.
```

It is possible to tell Fx Composer to use another buffer then the back buffer. So you can pretty much make any preprossessing or postprocessing effect you want. You can also play animations. And there's much more to effect files. For special effects, look in the GPU Gems.

References

[1] NVdia, "Samantics and attributes (sass)", http://developer.nvidia.com/object/using_sas.html/

[2] Microsoft, "HLSL", http://msdn.microsoft.com/en-us/library/ee418149(VS.85).aspx