A Fixed Function Shader in HLSL

Pedro V. Sander
ATI Research

October 2003

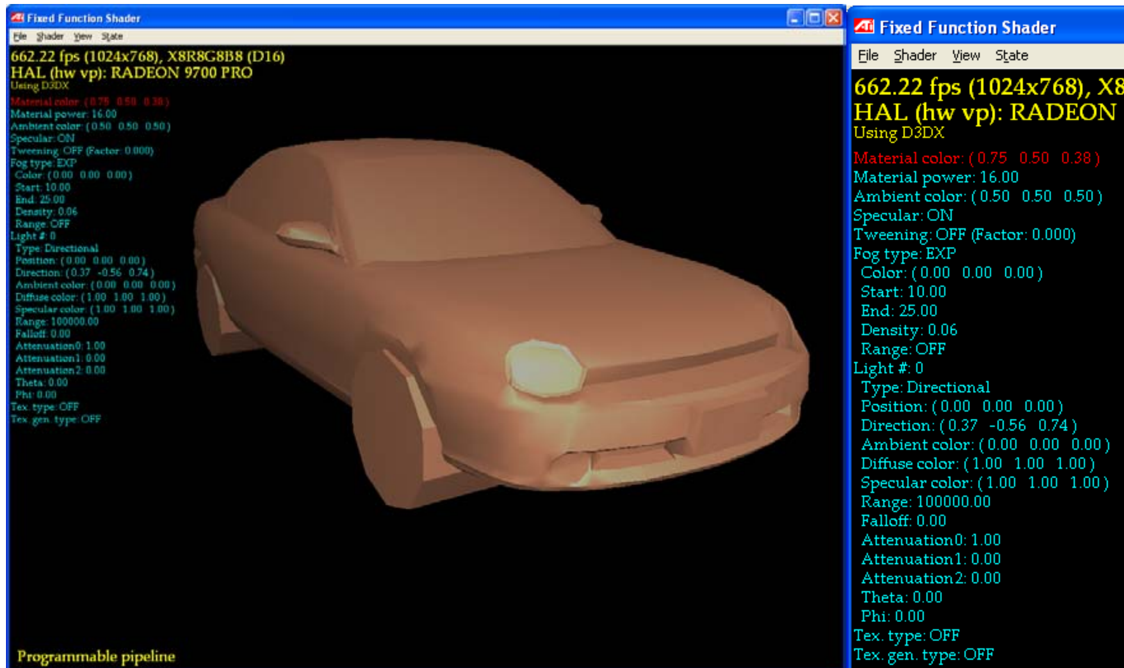


Figure 1 - A screenshot of the Fixed Function Shader program with a closeup of the state settings shown to the right.

Introduction

The High Level Shading Language (HLSL) introduced DirectX® 9 allows developers to write complex shaders in significantly less time than using earlier, low-level assembly languages. This document describes the implementation of parts of the fixed function pipeline using an HLSL vertex shader. It serves as a starting point for developers writing shaders that, in part, mimic portions of the fixed function pipeline. This white paper is also a valuable source on compiler issues and general information concerning HLSL programming. The HLSL compiler included in the DirectX 9 SDK Update (Summer 2003) is the first version available which supports compilation to asm flow-control instructions and is necessary for this analysis. Hence, if you wish to download our sample application and rebuild it, you will need this SDK or a later version.

The shader discussed in this document includes multiple directional, point and spot light sources, vertex fog, tweening, and automatic texture coordinate generation. In addition, a "diff" mode is available to compare the output of the "fixed function" shader with that of the fixed function code path. The source code for this project is publicly available on the ATI Developer Relations web site (http://ati.com/developer).

Application Overview

The FixedFuncShader application renders a mesh using either our HLSL shader or the standard DirectX 9 fixed function API. The code is written is C++ and HLSL. It is based on the CD3DApplication class provided with the Microsoft DirectX 9 SDK.

As shown in Figure 1, the state settings used to render the mesh are displayed on the upper-left corner of the window. The state can be changed by using the keyboard arrow keys, or by loading a state file (*.s) from disk.

State files (.s)*

State files can be loaded from the command line by specifying "`/s:file.s`". They can also be loaded from the State menu or by pressing F9. States can be saved to disk from the State menu or by pressing F10.

When states are saved to disk, the mesh filename and camera information are also stored. When loading a state file at a later time, one may want to also load the mesh that is associated with that state as well as the viewpoint at the time the state was saved. This can be accomplished by selecting "Load state w/ mesh" from the State menu or by pressing Alt-F9. This can be very useful when analyzing any discrepancies between rendering done with the HLSL shader (which you can feel free to modify) and the fixed function API.

Mesh files (.x)*

Mesh files can be loaded from the command line by specifying "`/x:file.x`". They can also be loaded from the File menu or by pressing F11. For tweening to work, the current mesh filename must end with "1.x". There must also be a file ending with "2.x" in the same directory (*e.g.*, the dolphin1.x and dolphin2.x files provided with the accompanying application).

Display modes

There are four display modes which can be selected from the View menu:

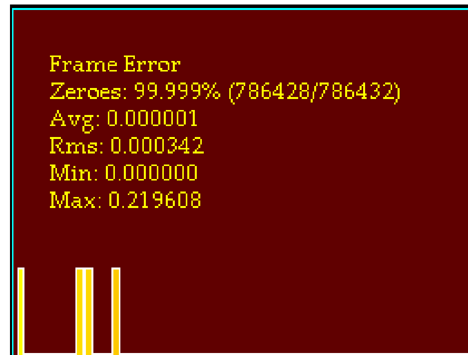
- **Programmable pipeline**: Uses our HLSL shader.
- **Fixed function pipeline**: Uses the fixed function code path.
- **Both**: Displays both of the above side-by-side.
- **Diff**: Displays a color-coded pixel "diff" between the programmable and fixed function pipelines.

Diff mode details

If diff mode is enabled, a pixel shader computes the distance between the color vectors of the output of both the programmable and fixed function pipeline.

If the difference is zero, the shader outputs a black pixel. Otherwise, if the difference is smaller than or equal 1/255.f then the shader outputs a green pixel. If the difference is larger than 1/255.f, then the shader interpolates between yellow and red (red meaning higher error).

Due to what we believe are numerical precision issues (e.g., dword vs. float4), we often get small errors in our shader. The "green" errors can be turned on and off from the View menu or by pressing D.



In diff mode, a histogram of *all non-zero pixels* is displayed on the lower-right of the window. The percentage of zeroes, the average (avg), root-mean-square (rms), min and max are also displayed. The histogram to the right shows that there were four non-zero samples, the max being approximately 0.22.

The FixedFuncShader.fx effects file

D3DX Effects files encapsulate vertex shaders, pixel shaders, global variables and techniques in a single source. Techniques specify the rendering state and which shaders should be used in each pass. The main application source code simply sets the technique. After that, all primitives that are sent to the hardware are rendered using these settings.

We will now first step through our effects file, with additional comments where necessary. Subsequently, we will discuss other issues related to our HLSL shader.

First, the data structures that hold the state data are defined. These global variables are placed in the constant store and are accessed by the vertex shader. Note that the `bSpecular` variable is explicitly stored in a bool register. Since we will compile this shader using vs_2_0, this causes the compiler to perform static flow control for `if(bSpecular)` instructions. The same applies to the `bTweening` and `bFogRange` variables.

```
#define PI  3.14f

//this file contains light, fog, and texture types
#include "FixedFuncShader.fxh"

// Structs and variables with default values

float4 vMaterialColor = float4(192.f/255.f, 128.f/255.f,
                               96.f/255.f, 1.f);
float fMaterialPower = 16.f;

float4 vAmbientColor = float4(128.f/255.f, 128.f/255.f,
                              128.f/255.f, 1.f);

bool bSpecular : register(b0) = false;
```

```
//tweening settings
bool bTweening : register(b2) = false;
float fTweenFactor = 0.f;
```

The fog settings are defined next:

```
//fog settings
int iFogType = FOG_TYPE_NONE;
float4 vFogColor = float4(0.0f, 0.0f, 0.0f, 0.0f);
float fFogStart = 10.f;
float fFogEnd = 25.f;
float fFogDensity = .02f;
bool bFogRange : register(b4) = false;
```

All three fog types are supported (linear, exp, and exp2). To enable fog, iTexFogType can be set to one of:

```
//from FixedFuncShader.fxh
#define FOG_TYPE_NONE           0
#define FOG_TYPE_EXP            1
#define FOG_TYPE_EXP2           2
#define FOG_TYPE_LINEAR         3
```

Then, texture coordinate generation settings are defined.

```
//texture coordinate generation settings
int iTexType = TEX_TYPE_NONE;
int iTexGenType = TEXGEN_TYPE_NONE;
```

`iTexType` can either be set to `TEX_TYPE_NONE`, which disables textures, or `TEX_TYPE_CUBEMAP`, which loads a simple cube map texture. `iTexGenType` can either be set to one of the following:

```
//from FixedFuncShader.fxh
#define TEXGEN_TYPE_NONE                          0
#define TEXGEN_TYPE_CAMERASPACENORMAL             1
#define TEXGEN_TYPE_CAMERASPACEPOSITION           2
#define TEXGEN_TYPE_CAMERASPACEREFLECTIONVECTOR   3
```

`TEXGEN_TYPE_NONE` disables texture coordinate generation and simply passes the input texture coordinates on to the rasterizer. The other three modes are the texture coordinate generation modes supported by the fixed function API.

Next comes the light data. All light data is encapsulated in a single structure. This structure is analogous to the `D3DLIGHT9` light structure of DirectX. It contains all the data necessary for directional, point, and spot lights. By default, the compiler places all data members of this structure in individual constant float4 registers, with each data member taking up 128 bytes (4 floats). This is true even of the scalar float members of the structure. We have tightly packed the attenuation and spot light terms. One can imagine

packing this structure even tighter if one wants to sacrifice readability for additional float constant store space.

```
struct CLight
{
   int iType;
   float3 vPos;
   float3 vDir;
   float4 vAmbient;
   float4 vDiffuse;
   float4 vSpecular;
   float  fRange;
   float3 vAttenuation; //1, D, D^2;
   float3 vSpot;        //cos(theta/2), cos(phi/2), falloff
};
```

The `iType` of the light can be one of:

```
//from FixedFuncShader.fxh
#define LIGHT_TYPE_NONE          0
#define LIGHT_TYPE_POINT         1
#define LIGHT_TYPE_SPOT          2
#define LIGHT_TYPE_DIRECTIONAL   3
```

Then, the additional light settings are declared. These settings are used by the for loops that loop over the lights. `iLightDirIni` is the index to the first directional light in the light array. `iLightDirNum` is the number of directional lights (all lights of the same type are placed consecutively in the array). Point and spot light variables are defined similarly.

```
//initial and range of directional, point and spot lights within the
//light array
int iLightDirIni;
int iLightDirNum;
int iLightPointIni;
int iLightPointNum;
int iLightSpotIni;
int iLightSpotNum;
```

Five lights are initialized. The number of lights is configurable and can be changed in the FixedFuncShader.fxh file. The number of lights that this shader can operate on is limited only by the size of the constant store space due to the use of looping. (The limit for our shader is 24 lights due to the other variables that are also in the constant store.)

```
CLight lights[5] = {                        //NUM_LIGHTS == 5
   {
      LIGHT_TYPE_DIRECTIONAL,               //type
      float3(0.0f, 0.0f, 0.0f),             //position
      float3(2.0f,-3.0f, 4.0f),             //direction
      float4(0.0f, 0.0f, 0.0f, 0.0f),       //ambient
      float4(1.0f, 1.0f, 1.0f, 1.0f),       //diffuse
      float4(1.0f, 1.0f, 1.0f, 1.0f),       //specular
      1000.f,                               //range
      float3(1.f, 0.f, 0.f),                //attenuation
```

```
      float3(.999f, .996f, 1)                //spot (theta=PI/50,
                                             //phi=PI/20)
   },
   //Note: The remaining four lights go here (see source code).
};
```

Next, the transformation matrices are declared. These are initialized from the application source code as described in the "Interfacing with the effects file" section.

```
//transformation matrices
float4x4 matWorldViewProj  : WORLDVIEWPROJ;
float4x4 matWorldView      : WORLDVIEW;
float4x4 matWorld          : WORLD;
float4x4 matWorldViewIT;
float4x4 matViewIT;
```

Then, the output structures of the vertex shaders and its helper functions are declared. Note that specular color is output separately from diffuse. This is important to match the fixed function pipeline; diffuse and specular are saturated independently prior to vertex fog processing.

```
struct VS_OUTPUT
{
   float4 Pos            : POSITION;
   float4 Color          : COLOR0;
   float4 ColorSpec      : COLOR1;
   float4 Tex0           : TEXCOORD0;
   float  Fog            : FOG;
};

struct COLOR_PAIR
{
   float4 Color          : COLOR0;
   float4 ColorSpec      : COLOR1;
};
```

Next, all the functions for light computation are defined.

```
//-----------------------------------------------------------------------
// Name: DoDirLight()
// Desc: Directional light computation
//-----------------------------------------------------------------------
COLOR_PAIR DoDirLight(float3 N, float3 V, int i)
{
   COLOR_PAIR Out;
   float3 L = mul((float3x3)matViewIT, -normalize(lights[i].vDir));
   float NdotL = dot(N, L);
   Out.Color = lights[i].vAmbient;
   Out.ColorSpec = 0;
   if(NdotL > 0.f)
   {
      //compute diffuse color
      Out.Color += NdotL * lights[i].vDiffuse;
```

```
      //add specular component
      if(bSpecular)
      {
         float3 H = normalize(L + V);   //half vector
         Out.ColorSpec = pow(max(0, dot(H, N)), fMaterialPower) *
                         lights[i].vSpecular;
      }
   }
   return Out;
}

//-----------------------------------------------------------------------
// Name: DoPointLight()
// Desc: Point light computation
//-----------------------------------------------------------------------
COLOR_PAIR DoPointLight(float4 vPosition, float3 N, float3 V, int i)
{
   float3 L = mul((float3x3)matViewIT, normalize((lights[i].vPos-
               (float3)mul(matWorld,vPosition))));
   COLOR_PAIR Out;
   float NdotL = dot(N, L);
   Out.Color = lights[i].vAmbient;
   Out.ColorSpec = 0;
   float fAtten = 1.f;
   if(NdotL >= 0.f)
   {
      //compute diffuse color
      Out.Color += NdotL * lights[i].vDiffuse;

      //add specular component
      if(bSpecular)
      {
         float3 H = normalize(L + V);   //half vector
         Out.ColorSpec = pow(max(0, dot(H, N)), fMaterialPower) *
                         lights[i].vSpecular;
      }

      float LD = length(lights[i].vPos-
                  (float3)mul(matWorld,vPosition));
      if(LD > lights[i].fRange)
      {
         fAtten = 0.f;
      }
      else
      {
         fAtten *= 1.f/(lights[i].vAttenuation.x +
                   lights[i].vAttenuation.y*LD +
                   lights[i].vAttenuation.z*LD*LD);
      }
      Out.Color *= fAtten;
      Out.ColorSpec *= fAtten;
   }
   return Out;
}

//-----------------------------------------------------------------------
// Name: DoSpotLight()
```

```
// Desc: Spot light computation
//-----------------------------------------------------------------------
COLOR_PAIR DoSpotLight(float4 vPosition, float3 N, float3 V, int i)
{
   float3 L = mul((float3x3)matViewIT, normalize((lights[i].vPos-
                  (float3)mul(matWorld,vPosition))));
   COLOR_PAIR Out;
   float NdotL = dot(N, L);
   Out.Color = lights[i].vAmbient;
   Out.ColorSpec = 0;
   float fAttenSpot = 1.f;
   if(NdotL >= 0.f)
   {
      //compute diffuse color
      Out.Color += NdotL * lights[i].vDiffuse;

      //add specular component
      if(bSpecular)
      {
         float3 H = normalize(L + V);   //half vector
         Out.ColorSpec = pow(max(0, dot(H, N)), fMaterialPower) *
                         lights[i].vSpecular;
      }

      float LD = length(lights[i].vPos-
                        (float3)mul(matWorld,vPosition));
      if(LD > lights[i].fRange)
      {
         fAttenSpot = 0.f;
      }
      else
      {
         fAttenSpot *= 1.f/(lights[i].vAttenuation.x +
                       lights[i].vAttenuation.y*LD +
                       lights[i].vAttenuation.z*LD*LD);
      }

      //spot cone computation
      float3 L2 = mul((float3x3)matViewIT, -normalize(lights[i].vDir));
      float rho = dot(L, L2);
      fAttenSpot *= pow(saturate((rho - lights[i].vSpot.y)/
                        (lights[i].vSpot.x - lights[i].vSpot.y)),
                        lights[i].vSpot.z);

      Out.Color *= fAttenSpot;
      Out.ColorSpec *= fAttenSpot;
   }
   return Out;
}
```

Then, the vertex shader is defined. It first checks whether tweening is enabled. If so, it properly blends the positions based upon the tween factor.

```
//-----------------------------------------------------------------------
// Name: vs_main()
// Desc: The vertex shader
```

```
//-----------------------------------------------------------------------
VS_OUTPUT vs_main (float4 vPosition  : POSITION0,
                           float4 vPosition2 : POSITION1,
                           float3 vNormal    : NORMAL0,
                           float3 vNormal2   : NORMAL1,
                           float2 tc         : TEXCOORD0)
{
   VS_OUTPUT Out = (VS_OUTPUT) 0;

   if(bTweening)
   {
      vPosition = (1.f-fTweenFactor) * vPosition +
                  fTweenFactor * vPosition2;
      vNormal   = (1.f-fTweenFactor) * normalize(vNormal) +
                  fTweenFactor * normalize(vNormal2);
   }
```

After that, it normalizes the normal, and converts vectors to view space, which is necessary for proper light, fog and texture coordinate computation. Strictly speaking, this normalization is optional and would be controlled by the `D3DRS_NORMALIZENORMALS` render state in the fixed function case, but we have chosen to hard code this normalization for simplicity.

```
   vNormal = normalize(vNormal);
   Out.Pos = mul(matWorldViewProj, vPosition);

   float3 P = mul(matWorldView, vPosition); //position in view space
   float3 N = mul((float3x3)matWorldViewIT, vNormal); //normal in view
   float3 V = -normalize(P); //viewer
```

Next, the shader performs texture coordinate generation. Note that since static flow control is only available on boolean constants, all code paths must be taken. Here, to minimize the number of instructions, the texture coordinates are computed by linearly interpolating all the possible settings.

```
   //automatic texture coordinate generation
   Out.Tex0 = float4((2.f * dot(V,N) * N - V) *
            (iTexGenType == TEXGEN_TYPE_CAMERASPACEREFLECTIONVECTOR)
            + N * (iTexGenType == TEXGEN_TYPE_CAMERASPACENORMAL)
            + P * (iTexGenType == TEXGEN_TYPE_CAMERASPACEPOSITION), 0);
   Out.Tex0.xy += tc * (iTexGenType == TEXGEN_TYPE_NONE);
```

Finally, the light computation is performed. One loop is used for each light type (directional, point, and spot). Point and spot lights can be easily combined in order to reduce the total shader instruction count (only if the light is a spot light, the spot cone factor would be multiplied by the resulting color). In fact, prior to optimization, we had to merge these two light types in one loop to stay within the 256 instruction limit.

```
   //directional lights
   for(int i = 0; i < iLightDirNum; i++)
   {
      COLOR_PAIR ColOut = DoDirLight(N, V, i+iLightDirIni);
```

```
      Out.Color += ColOut.Color;
      Out.ColorSpec += ColOut.ColorSpec;
   }

   //point lights
   for(int i = 0; i < iLightPointNum; i++)
   {
      COLOR_PAIR ColOut = DoPointLight(vPosition, N, V,
                                        i+iLightPointIni);
      Out.Color += ColOut.Color;
      Out.ColorSpec += ColOut.ColorSpec;
   }

   //spot lights
   for(int i = 0; i < iLightSpotNum; i++)
   {
      COLOR_PAIR ColOut = DoSpotLight(vPosition, N, V,
                                       i+iLightSpotIni);
      Out.Color += ColOut.Color;
      Out.ColorSpec += ColOut.ColorSpec;
   }
```

Next, the light is multiplied by the material color and saturated. Note that while pixel shaders have a saturate operation, vertex shaders do not. As a result, if we only need to check against one side, it is better to just perform the min or max operation, as it saves one instruction.

```
   //apply material color
   Out.Color *= vMaterialColor;
   Out.ColorSpec *= vMaterialColor;

   //saturate
   Out.Color = min(1, Out.Color);
   Out.ColorSpec = min(1, Out.ColorSpec);
```

Finally, fog is computed using the same linear interpolation approach of the texture computation.

```
   //apply fog
   if(bFogRange)
      d = length(P);
   else
      d = P.z;
   Out.Fog = 1.f * (vfog.iType == FOG_TYPE_NONE)
             + 1.f/exp(d * vfog.fDensity)
               * (vfog.iType == FOG_TYPE_EXP)
             + 1.f/exp(pow(d * vfog.fDensity, 2))
               * (vfog.iType == FOG_TYPE_EXP2)
             + saturate((vfog.fEnd - d)/(vfog.fEnd - vfog.fStart))
               * (vfog.iType == FOG_TYPE_LINEAR);
   return Out;
}
```

Next, we describe the techniques used for rendering. Two techniques are defined. The first technique follows the fixed function code path and just sets the state variables appropriately (no vertex shaders or pixel shaders are used).

```
// Techniques

//the technique to set the state for the fixed function shader

technique basic
{
   pass P0
   {
      AMBIENT = (vAmbientColor);
      SPECULARENABLE = (bSpecular);
      FOGENABLE = (vfog.iType != FOG_TYPE_NONE);
      FOGCOLOR = (vfog.vColor);
   }
}
```

The second technique uses our HLSL “fixed function” vertex shader. It does not need to set all the state variables, since the shader itself queries those. It just needs to set the specular and fog settings, so that the specular color and fog components output from the shader are used.

```
//technique for the programmable shader (simply sets the vertex shader)
technique basic_with_shader
{
   pass P0
   {
      SPECULARENABLE = (bSpecular);
      FOGENABLE = (vfog.iType != FOG_TYPE_NONE);
      FOGCOLOR = (vfog.vColor);
      VertexShader = compile vs_2_0 vs_main();
   }
}
```

When the app is in diff mode, two render-to-texture passes are used (one for each of the above techniques). Then, on a third pass, a quad is rendered filling the entire back buffer using a pixel shader that computes the difference between the two textures. The texture and sampler settings are omitted here. Please refer to the code for further detail. The diff pixel shader and technique are shown below.

```
//-----------------------------------------------------------------------
// Name: ps_diff()
// Desc: Pixel shader for the diff mode
//       Tiny errors: green. Larger errors: yellow to red.
//-----------------------------------------------------------------------
float4 ps_diff (float2 tcBase : TEXCOORD0) : COLOR
{
   float E = length(tex2D(DiffSampler1, tcBase)
                    - tex2D(DiffSampler2, tcBase))/sqrt(3);
   float4 C = float4(0.f,0.f,0.f,E);
```

```
   if(E > 0.f)
   {
      if(E <= 1.f/255.f)
      {
         if(bDiffSensitivity)
         {
            C = float4(0.f,1.f,0.f,E);
         }
      }
      else
      {
         C = lerp(float4(1.f,1.f,0.f,E), float4(1.f,0.f,0.f,E),E);
      }
   }
   return C;
}

//technique for the diff mode
technique technique_diff
{
   pass P0
   {
      PixelShader = compile ps_2_0 ps_diff();
   }
}
```

Additional HLSL Issues

Aside from the issues described above, several other design and technical issues arose in the process of building this shader. We will describe some of these issues next.

Vertex Shader 2.0

The main purpose of this shader is to mimic the standard DirectX 9 fixed function vertex pipeline using static flow control. vs_2_0 is the first version that supports if-else-endif and loop instructions. More details on these instructions can be found below on their respective sections.

In developing this shader, we attempted to mimic a large subset of the fixed function vertex pipeline within a single shader. vs_2_0 supports 256 instructions, twice that of vs_1_1, and is sufficient to implement all types of lights, fog, texture coordinate generation, and tweening. Aside from that, it also has significantly more constant store space, which is necessary for applications that require a lot of lights.

Having said that, parts of this shader can, however, be stripped out and compiled for vs_1_1 (without static flow control) in order to support legacy hardware.

Boolean registers

vs_2_0 supports 16 constant boolean registers. In order to place a bool variable in the boolean register, the variable must be declared as follows:

```
bool bSpecular : register(b0) = false;
```

where b0 can be replaced by b1, b2,…, b15.

Unfortunately, indexing into boolean registers (by using an int or float index) is not supported by the programming model. Thus, one cannot use an array of bools to specify whether a light is on or off and index into it inside a loop. In order to address this problem, our shader uses separate loops for the different types of lights. We will discuss this in further detail in the "Loops" section below.

Flow control

In vs_2_0, if-else-endif static flow control instructions are evaluated and the code path is determined prior to execution. Thus, static branching instructions are essentially free. Referring back to the HLSL code on pages 6 and 7, the DoDirLight() routine conditionally computes specular illumination based on a static branch. Such HLSL code results in the following asm code.

```
if b0
  mad r3.xyz, r7, -r7.w, r0
  nrm r0.xyz, r3  // DoDirLight::H<0,1,2>
  dp3 r0.x, r0, r8
  max r4.w, r0.x, c76.x
  pow r3.w, r4.w, c64.x
  mul r4, r3.w, c5[a0.w]  // DoDirLight::Out<4,5,6,7>
else
  mov r4, c76.x  // DoDirLight::Out<4,5,6,7>
endif
```

As you can see, this code computes the specular color only if register b0 is true. If b0 is false, only one instruction is executed.

Such static flow control instructions can only be performed on constant boolean registers. This can be a significant drawback since it prevents any flow control from being done inside the loop if the loop invariant is used to index the boolean array. If you have a large boolean array that is indexed inside a loop that is too large to unroll while staying within the 256 instruction limit, the only solution is to use float storage and give up doing flow control on that variable. In that case, since there is no dynamic flow control in vs_2_0, the shader would take all code paths and linearly interpolate the results. While this significantly degrades performance, it might be necessary to stay within the instruction count limit.

Since there is no dynamic flow control, code optimization strategies are significantly different from those performed on code that runs on the CPU. For instance, the following linear fog computation code:

```
if(d <= fFogStart)
   fog = 1.f;
```

```
else if(d >= fFogEnd)
   fog = 0.f;
else
   fog = (fFogEnd - d)/(fFogEnd - fFogStart);
```

can be replaced by

```
fog = saturate((fFogEnd - d)/(fFogEnd - fFogStart));
```

since the more expensive computation with two subtractions and one division would have to be performed anyway, regardless of the value of *d*. The above optimization prevents lerping of the if-elseif-else expressions, thus resulting in three fewer instructions.

Below is another code example.

```
if(floatVar == 1)
   x = a;
else
   x = 0;
```

If `floatVar` can only hold a value of 0 or 1, the best way to do flow control based on `floatVar`, is to do the lerping on the HLSL code. The above code can be replaced by

```
x = floatVar * a;
```

Also note that checking for a non-zero value (storing 1 in `r0.w` if the value in `c0.x` is positive) compiles to

```
mul r0.w, c0.x, c0.x
slt r0.w, -r0.w, r0.w
```

while checking for 1 compiles to

```
mov r0.w, c0.x
add r0.w, r0.w, c1.x   //c1.x == -1
mul r0.w, r0.w, r0.w
sge r0.w, -r0.w, r0.w
```

So, `if(floatVar)` would also be more efficient than `if(floatVar == 1)`.

Loops

Since we cannot perform static flow control to determine the light types of each of our N lights inside a loop, our solution is to use different loops for each light type. This solution ensures that, for each light, only the computation relevant to that particular light type is performed.

The D3DX HLSL compiler has some restrictions on the types of for loops which will result in asm flow-control instructions. Specifically, they must be of the form `for(i = 0; i < n; i++)` in order to generate the desired asm instruction sequence:

```
rep i0
  //loop instructions
endrep
```

where `i0` is an integer register specifying the number of times to go through the loop. The loop counter is initialized before the `rep` instruction and incremented before the `endrep` instruction.

Since loops must be specified in that form, we have an `iLightDirNum` instead of a `lightDirEnd` to specify the bounds of the light array:

```
for(int i = 0; i < iLightDirNum; i++)
{
   COLOR_PAIR ColOut = DoDirLight(N, V, i+iLightDirIni);
   Out.Color += ColOut.Color;
   Out.ColorSpec += ColOut.ColorSpec;
}
```

Interfacing with the Effects file

Effects interface creation

The ID3DXEffect interface is used to query and modify the state variables in the effects file. An instance of this interface is created via a call to **`D3DXCreateEffectFromFile`**`()`, passing the FixedFuncShader.fx file as a parameter:

```
D3DXCreateEffectFromFile(m_pd3dDevice, "Effects\\FixedFuncShader.fx",
                         NULL, NULL, D3DXSHADER_DEBUG, NULL,
                         &m_pEffect,&pBufferErrors);
```

Querying and modifying effects file variables

At startup, the application queries the default values from the effects file. At runtime, the effects file global variables are set whenever they are modified using the UI. To modify the values of a global variable in the effects file, the **`SetValue`**`()` member function is used. For instance, the following code sets the value of the ambient color on the shader.

```
pEffect->SetValue("vAmbientColor", (void*)&m_vAmbientColor,
                  sizeof(D3DCOLORVALUE));
```

Similarly, the following code queries that value.

```
pEffect->GetValue("vAmbientColor", (void*)&m_vAmbientColor,
                  sizeof(D3DCOLORVALUE));
```

There are also specific functions for different data types, such as **GetFloat()** and **GetInt()**. However, **GetValue()** and **SetValue()** work for all cases I came across, except for boolean variables and matrices, which must be accessed using their specific functions:

```
pEffect->SetBool("bTweening", &m_bTweening);
pEffect->GetBool("bTweening", &m_bTweening);

pEffect->SetMatrix("bTweening", &m_bTweening);
pEffect->GetMatrix("bTweening", &m_bTweening);
```

However, since matrices are stored in transposed form, it is easier to use:

```
pEffect->SetMatrixTranspose("matView", &m_matView);
pEffect->GetMatrixTranspose("matView", &m_matView);
```

Then, your HLSL code can do standard right-multiplication of vectors (*e.g.*, `x = Ab`).

The `CSceneState` class

The main application class owns an instance of the **CSceneState** class, which interfaces with the effects file. This class receives a pointer to the **ID3DXEffect** interface and queries/sets the global variables in the effects file that are used by the HLSL shader. This is achieved by calling the member functions **InitFromEffects()** and **WriteToEffects()**, which in turn calls the **GetValue()** and **SetValue()** functions described above for all the variables. Each global variable in the effects file has a counterpart in this class:

```
//three lights
int m_iLightType[3];
D3DLIGHT9 m_light[3];

//fog settings
CVertexFog m_vfog;

//texture coordinate settings for 3 stages
CTexture m_tex[3];

//misc settings
bool m_bSpecular;
bool m_bVertexColor;
bool m_bTweening;
D3DCOLORVALUE m_vAmbientColor;
D3DCOLORVALUE m_vMaterialColor;
float m_fMaterialPower;
```

The **CVertexFog** and **CTexture** structures are defined as:

```
struct CVertexFog
{
   int iType;
```

```
    D3DCOLORVALUE vColor;
    float fStart;
    float fEnd;
    float fDensity;
    bool bRange;
};

struct CTexture
{
    int iTexType;
    int iTexGenType;
    int iTexTransType;
    bool bTexTransProjected;
    D3DXMATRIXA16 matTexTransform;
};
```

Rendering

During rendering, the technique specified in the effects file is set using a call to **`SetTechnique`**`()`. Both code paths (our HLSL shader and the fixed function pipeline) use a single pass for rendering the scene, thus the rendering code is as follows:

```
m_pEffect->SetTechnique(m_pEffect->GetTechniqueByName("basic"));

UINT cPasses;
m_pEffect->Begin(&cPasses, 0);
m_pEffect->Pass(0);

//render code goes here

m_pEffect->End();
```

If the technique specifies multiple passes, they can be rendered by calling the **`Pass`**`()` member function prior to the draw calls.

Render and Texture States

Below is a list of render and texture states supported by our shader.

Lighting:

```
D3DRS_LIGHTING
D3DRS_AMBIENT
```

Fog:

```
D3DRS_FOGVERTEXMODE
D3DRS_FOGCOLOR
D3DRS_FOGSTART
D3DRS_FOGEND
D3DRS_FOGDENSITY
D3DRS_RANGEFOGENABLE
```

Tweening:

```
D3DRS_VERTEXBLEND
D3DRS_TWEENFACTOR
```

Texture coordinate generation:

```
D3DTSS_TEXCOORDINDEX
```

Acknowledgements

I would like to thank Jason Mitchell and other members of ATI's 3D Application Research Group for comments and suggestions. Thanks also to Craig Peeper and Loren McQuade of Microsoft for information on the HLSL compiler.