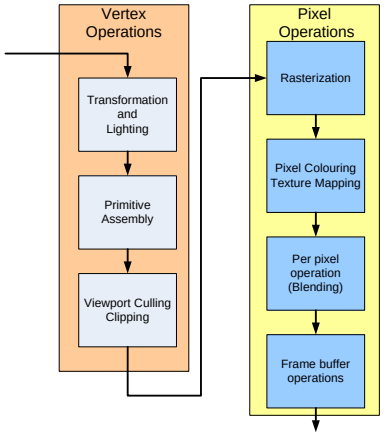
Shaders

Part II

Doron Nussbaum

Traditional Rendering Pipeline

- Traditional pipeline (older graphics cards) restricts developer to texture and the 3-term lighting model
- Pixel information was interpolated to vertices
- Shaders
  - Lift the restrictions
  - Allow developers to manipulate outcome
- Result
  - vertex shaders
  - Pixel/fragment shaders



Shader Support

- Test for shader availability (on the graphics card)
  - Different graphics cards may have different capabilities
- Result
  - Use the fixed (traditional) graphics pipeline
  - Use programmable graphics pipeline (shaders)
    - Different version of shaders

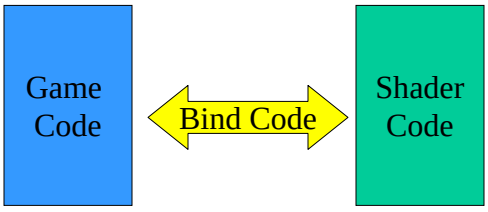

```
checkShaderSupport() {

        D3DCAPS9 caps;

        d3dDev->GetDeviceCaps(&caps);

        // check for shader support of version 3

        If (caps.VertexShaderVersion < D3DVS_VERSION(3,0)) {
             // no support for Vertex Shader version 3
             // do something – use fixed pipline, try version 2.0 etc.
        }

        // check for pixel Shader Support
        If (caps.PixelShaderVersion < D3DVS_VERSION(3,0)) {
             // no support for Vertex Shader version 3
             // do something – use fixed pipline, try version 2.0 etc.
        }
}
```

Shaders

- Stored in an external file
- Accessed at run time
  - "Independent" of compiled code
  - Compiled at run time
- Easy to replace and modify
  - Send a new copy to the client
  - Easy to upgrade/degrade the graphics presentation
- Executed on the graphics card

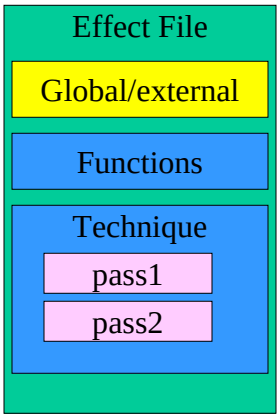
Shader Operation

- Three main steps
  - Write the shader code in an external file
  - Bind required shader code to game code
  - Determine at run time which parts of the code to execute



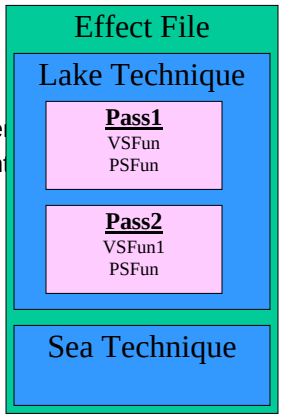
Shader Framework – Direct3D Effect

- The shader is encapsulated in Direct3D effect
- Want to determine at run time
  - What to execute (which function(s))
  - When to execute
  - Adapt the shaders tl
    - Code
    - Performance
    - capabilities



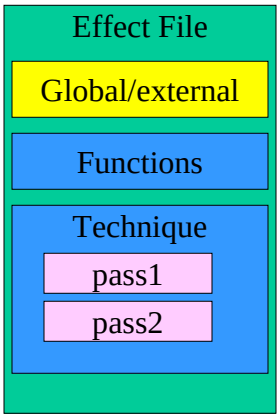
Direct3D Effect - Technique

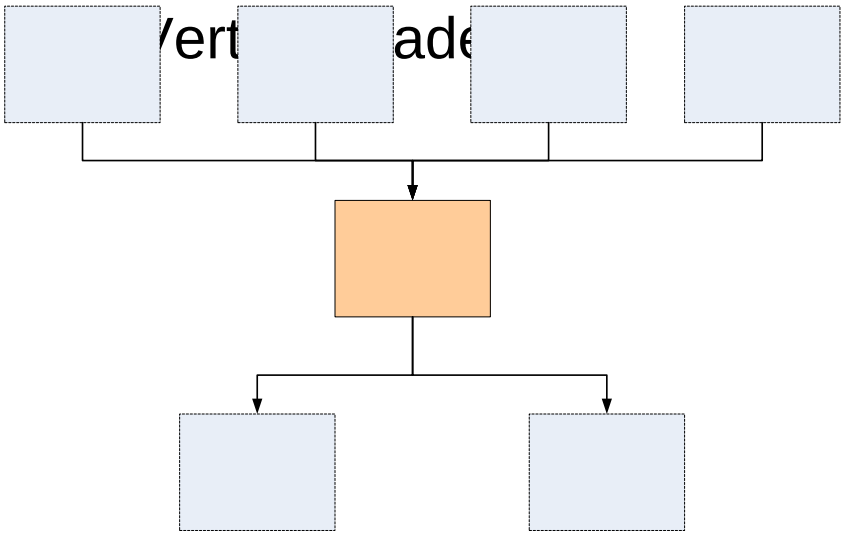
- Technique - The core of the effect
- Purpose
  - Determines how an object is rendered
  - Different objects → different techniques
- More than that
  - Use different techniques to support differe
  - Use different techniques to create different
    - Calm water, Waves, Whirlpool
- Techniques contain one or more passes
  - Each pass contains
    - Vertex shader function
    - Pixel shader function



Shader Framework – Direct3D Effect

- Variables
  - Global
  - Local
- Functions
  - Designated Special functions
    - Vertex shader
    - Pixel shader
  - Local functions





```
// transformation matrix from the program  (LAST)
uniform extern float4x4 WorldViewProjMat;

struct OutputVS
{
    float4 pos : POSITION0;
};

// vertex shader
struct OutputVS TransformFunVS(float3 pos : POSITION0)
{
      // initalize the output
      struct OutputVS v = {0.0,0.0,0.0,0.0};

      v.pos = mul(float4(pos,1.0f),WorldViewProjMat);


      // produce output
      return(v);
}

// Pixel Shader
float4 TransformFunPS() : COLOR
{
      float4 colour = {1.0,1.0,1.0,1.0};
      return(colour);
}
```



```
// transformation matrix from the program
uniform extern float4x4 WorldViewProjMat;

struct OutputVS
{
    float4 pos : POSITION0;
    float4 col : COLOR0;
};

// vertex shader
struct OutputVS TransformFunVS(float3 pos : POSITION0, float4 col : COLOR0)
{
      // initalize the output
      struct OutputVS v = {{0.0,0.0,0.0,0.0},{0.0,0.0,0.0,0.0}};

      v.pos = mul(float4(pos,1.0f),WorldViewProjMat);
      v.col = col;

      // produce output
      return(v);
}


// Pixel Shader
float4 TransformFunPS(float4 col : COLOR0) : COLOR
{
      return(col);
}
```

Semantic HLSL

- All input parameters must be “marked” in order to link/bind flow of data in the shaders
  - Position
  - Color
  - Normal
  - Texture coordinates

```
struct meshVertex {
    D3DXVECTOR3 pos;  // position
    DWORD color;          // vertex color
};
```

```
// vertex shader
struct OutputVS TransformFunVS(float3 pos : POSITION0, float4 col : COLOR0)
```


Semantic HLSL

```
struct meshVertex {
    D3DXVECTOR3 pos;  // position
    DWORD color;          // vertex color
};



D3DVERTEXELEMENT9 decl[] =
{{0,0,D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0},

 {0,12,D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_COLOR, 0},
D3DDECL_END()};




// vertex shader
struct OutputVS TransformFunVS(float3 pos : POSITION0,
                               float4 col : COLOR0)
```

```
struct OutputVS
{
    float4 pos : POSITION0;
    float4 col : COLOR0;
    float2 texSky          :  TEXCOORD0
    float2 unUsedTex     :  TEXCOORD1

};


// vertex shader
struct OutputVS TransformFunVS(float3 pos : POSITION0, float4 col : COLOR0,
                                        float2 texSky :  TEXCOORD0, float2 unUsedTex :  TEXCOORD1)
{
            // initalize the output
            struct OutputVS v = {{0.0,0.0,0.0,0.0},{0.0,0.0,0.0,0.0}};
            …
            return(v);

}

// Pixel Shader
float4 TransformFunPS(float4 col : COLOR0,
                              float2 texSky :  TEXCOORD0,
                             float2 unUsedTex :  TEXCOORD1) : COLOR
…
```


Typical VS semantic

Input	Description	Type
COLOR[n]	Diffuse and specular color	float4
NORMAL[n]	Normal vector	float4
POSITION[n]	Vertex position in object space.	float4
PSIZE[n]	Point size	float
TEXCOORD[n]	Texture coordinates	float4

Output	Description	Type
COLOR[n]	Diffuse or specular color	float4
FOG	Vertex fog	float
POSITION[n]	Position of a vertex in homogenous space. Compute position in screen-space by dividing (x,y,z) by w. Every vertex shader must write out a parameter with this semantic.	float4
TEXCOORD[n]	Texture coordinates	float4

```
struct OutputVS
{
    float4 pos : POSITION0;
    float4 col : COLOR0;
    float2 texSky          :  TEXCOORD0
    float2 unUsedTex     :  TEXCOORD1

};

// vertex shader
struct OutputVS TransformFunVS(float3 pos : POSITION0, float4 col : COLOR0,
                                        float2 texSky :  TEXCOORD0, float2 unUsedTex :  TEXCOORD1)
…

// Pixel Shader
float4 TransformFunPS(float4 col : COLOR0,
                              float2 texSky :  TEXCOORD0,
                              float2 unUsedTex :  TEXCOORD1) : COLOR
…

technique TransformationTechnique
{
            pass P1 {
                        vertexshader = compile vs_3_0 TransformFunVS();
                        pixelshader = compile ps_3_0 TransformFunPS();

                        FillMode = WireFrame;  // see possible states for shader (documentation)
            }
}
```

Technique

- Defined by
  - *technique* identifier e.g., technique FirstTech

```
technique TransformationTechnique
{

            pass pass-identifier {...}
            pass pass-identifier {...}
            pass pass-identifier {...}
}
```

Steps

- Preparation
  - Create an effect
  - Create a handle for external effect parameters
  - Set the effect parameters
- Apply the effect
  - Determine the technique
  - Set the effect parameters
  - Begin the effect
  - Draw primitives
  - End the effect

Preparation Steps

- Create an effect
  - Create an effect interface
  - Load the effect from the file
  - Parse the effect

```
ID3DXEffect *myEffect = NULL;

ID3DXBuffer *errBuf = NULL;

Rc  = D3DXCreateEffectFromFile( d3dDev, "simpleShaders.fx", NULL, NULL,
D3DXSHADER_DEBUG, NULL, &myEffect, &errBuf);

If (rc != D3D_OK)
      if (errBuf != NULL) {
            MessageBox(0, (char *) errBuf->GetBufferPointer(), 0,0);
            errBuf->Release();
            errBug = NULL;
      }
}
```


HRESULT D3DXCreateEffectFromFile(
LPDIRECT3DDEVICE9 pDevice - Pointer to the device that will create the effect.

LPCSTR pSrcFile - Pointer to the filename.

*D3DXMACRO *pDefines* - Optional NULL-terminated array of preprocessor macro definitions.

LPD3DXINCLUDE pInclude – Optional interface pointer – use NULL

long Flags – optional flags for compiling the shaders. Use 0 for no flags
important flags: D3DXSHADER_DEBUG,

LPD3DXEFFECTPOOL pPool - Pointer to a ID3DXEffectPool object to used for shared parameters across a number of effects files. Use NULL (no parameters are shared)

ID3DXEffect **ppEffect* - returns a pointer to a compiled effect interface ID3DXEffect.

ID3DXBuffer **ppCompilationErrors* – Returns a pointer to a buffer containing a listing of compile errors

Return Values
Success - D3D_OK.
Failure - D3DERR_INVALIDCALL, D3DXERR_INVALIDDATA. E_OUTOFMEMORY

Preparation Steps

- Create a handle for external effect parameters
    - Bind the name from the file with a handle in the game program
- Set the effect parameters
  - Links a specific program variable with a parameter in the shader

```
// declare the handle
D3DXHANDLE ID3DXXEffect::GetParameterByName(

D3DXHANDLE parent,                    // scope of variable – parent structure

LPCSTR nNULL,                         // name of variable in the fx file
)


HRESULT SetMatrix(
D3DXHANDLE hParameter          // handle identifier

D3DXATRIX* pMatrix             // Pointer to a nontransposed matrix
)
```

Other Set Functions

- ID3DXBaseEffect::SetXXXX(D3DXHANDLE hParamater, XXXX var)
  - XXXX – BOOL, float, int,
- ID3DXBaseEffect::SetXXXX(D3DXHANDLE hParamater, XXXX *var)
  - XXXX –vector (4D), D3DXMATRIX
- ID3DXBaseEffect::SetXXXXArray(D3DXHANDLE hParamater, XXXX *var, UINT count)
  - XXXX – BOOL, float, int, vector (4D), D3DXMARTIX

// Set the value of an arbitrary parameter or annotation, including simple types, structs, arrays, strings, shaders and textures.

- ID3DXBaseEffect::SetValue(D3DXHANDLE *hParameter*, void *buf, unsigned int bufSize)

Preparation Steps (last)

C++ Code

```
D3DXMASTRIX combinedMat ;


// declare the handle
D3DXHANDLE *hWorldViewProjMat = NULL;

// Bind the handle to a name – the name is declared in the .fx file
hWorldViewProjMat = myEffect->GetParameterByName(NULL, “WorldViewProjMat”);
.
.
.
// set the matrix in the code
myEffect->SetMatrix(hWorldViewProjMat, &combinedMat);
```

FX File

```
uniform extern float4x4 WorldViewProjMat;

.
.
```


Steps

- Preparation
  - Create an effect
    - Create an effect interface
    - Load the effect from the file
    - Parse the effect
  - Create a handle for external effect parameters
    - Bind the name from the file with a handle in the game program
  - Set the effect parameters
    - Links a specific program variable with a parameter in the shader

Steps

- Apply the effect
  - Obtain a handle to the technique
    - D3DXHANDLE hT = myEffect->GetTechnicqueByName(“firstTech”);
  - Activate the desired technique
    - myEffect->SetTechnique(hT);
  - Set the effect parameters
  - Begin the effect
  - For each pass in the technique
    - Draw primitives
  - End the effect

Steps

- Apply the effect
  - Obtain a handle to the technique
    - D3DXHANDLE hT = myEffect->GetTechnicqueByName(“firstTech”);
  - Activate the desired technique
  - Set the effect parameters
  - Begin the effect
  - For each pass in the technique
    - Draw primitives
  - End the effect

```
// set the technique
rc = pMeshEffect->SetTechnique(hTechnique);

// set the global matrix
pMeshEffect->SetMatrix(hWorldViewProjMat, &(worldMat*viewMat*projMat));

// set the time
fTime = time;
pMeshEffect->SetFloat(hTime, fTime);

// start the effect
pMeshEffect->Begin(&numPasses, 0); // 0 means save & restore device states

// start the technique passess
for (i = 0; i < numPasses; i++) {
        pMeshEffect->BeginPass(i);

        // draw the mesh
        md3dDev->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0,
                                      this->numVtx, 0, this->numTriangles);

        pMeshEffect->EndPass();
}

pMeshEffect->End();
```

Steps

- Apply the effect
  - Obtain a handle to the technique
  - Activate the desired technique
  - Set the effect parameters
  - Begin the effect
  - For each pass in the technique
    - Draw primitives
  - End the effect

ID3DXEffect::Begin sets the active technique.

ID3DXEffect::BeginPass sets the active pass.

ID3DXEffect::CommitChanges updates changes to any set calls in the pass. This should be called before any draw call.

ID3DXEffect::EndPass ends a pass.

ID3DXEffect::End ends the active technique.