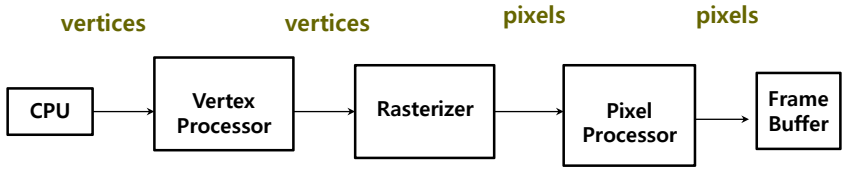
Shader

305890
Spring 2013
4/19/2013
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, 11
    - OpenGL Extensions
    - OpenGL Shading Language (GLSL)
    - Cg

Shader



Vertex Processor

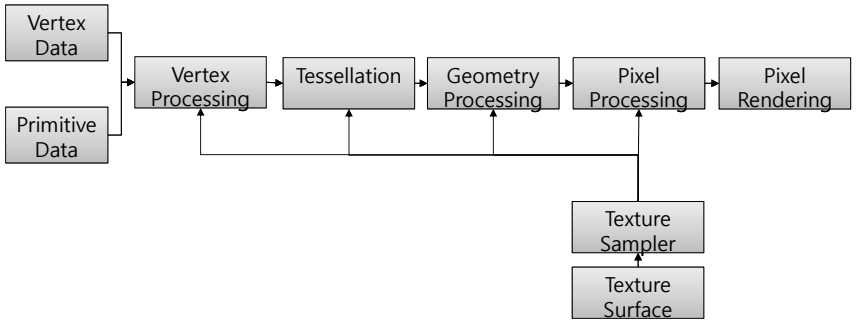
- Takes in **vertices**
  - Position attribute
  - Possibly color
  - State
- Produces
  - **Position** in clip coordinates
  - **Vertex color**

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**

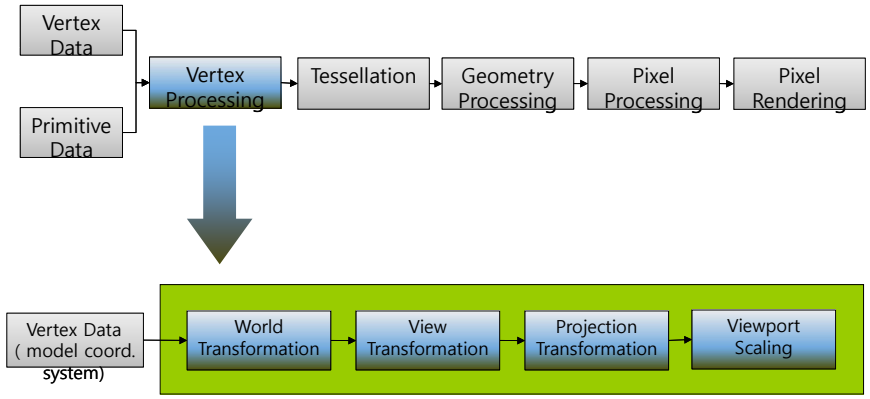
Shader

- Rendering Pipeline (DirectX11, OpenGL4.0)



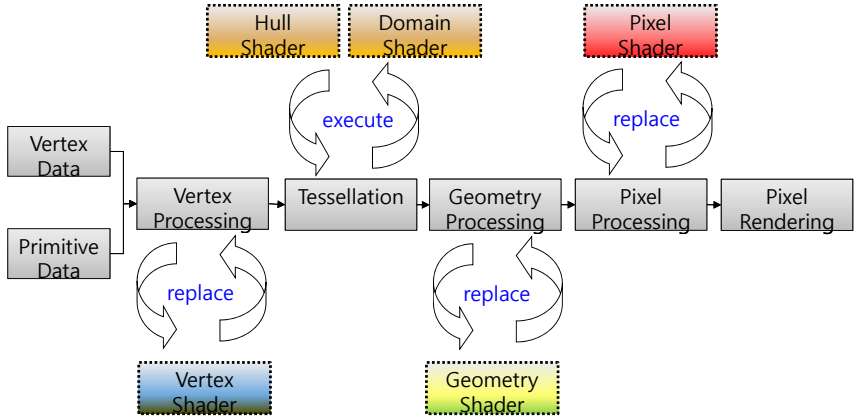
Shader

- Rendering Pipeline (DirectX11, OpenGL4.0)



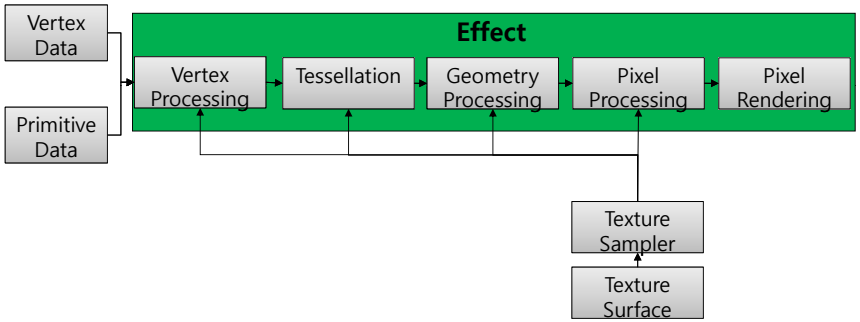
Shader

- Rendering Pipeline (DirectX11, OpenGL4.0)



Effect

- What is an Effect?



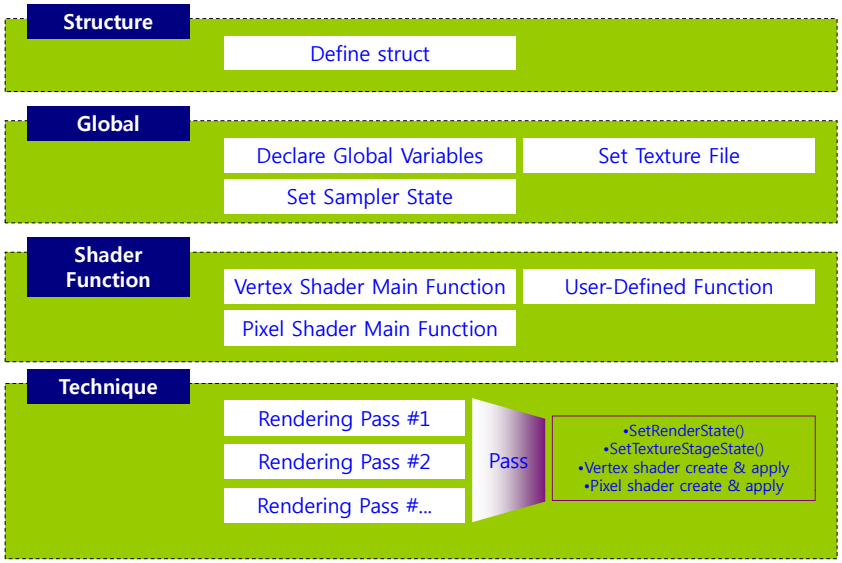
Effect

- Shader State
  - Set shader creation, delete, compile, rendering
  - Set constant, state
- Texture & Sampler State
  - Set texture file & stage
  - Set sampler object & state
- Etc
  - Set transformation, light, material, rendering options
- Shared parameters
  - Can share global variables, annotation parameters among different effects

Effect

- Multi-pass Rendering Option **<Technique>**
  - Can set different rendering styles using several **Pass**
  - Technique
    - Global variables
    - Rendering states
    - Texture stage states, Sampler states
    - Shader states

Effect Code Format



XNA Configurable Effect

- XNA 4.0 introduces configurable effect classes
  - **BasicEffect**
    - Use BasicEffect for transformation and basic lighting for Blinn-Phong shading. You have the option of adding up to three more directional lights, fog, and a texture. User one light for the fastest performance and use three lights for more interesting 3D lighting.
  - **DualTextureEffect**
    - Use DualTextureEffect to add more sophisticated lighting with a light map using two textures with two independent sets of texture coordinates.
  - **AlphaTestEffect**
    - Use AlphaTestEffect to use alpha blending (e.g. Billboards and Imposters). The effect uses a **CompareFunction** to compare the alpha value for a pixel against the **ReferenceAlpha** value to determine whether to draw the pixel.

XNA Configurable Effect

- XNA 4.0 introduces configurable effect classes
  - **SkinnedEffect**
    - Use SkinnedEffect to animate a character. This effect uses bones and weights to transform a mesh (an object is made up of several meshes). Simply set up a set of bones for a model when you create content, and then transform the bones during the render loop.
  - **EnvironmentMapEffect**
    - Use EnvironmentMapEffect to generate fast, specular highlights that add skinniness to an object. The effect uses two textures, a base texture with the texture detail and a cubemap whose six sides reflect the environment onto the object.
    - The effect uses **EnvironmentMapAmount** to control the amount of the environment map to add to the object. Also it uses **FresnelFactor** to control how much the edge of an object reflects specular lighting.

XNA Custom Effect

- A programmable effect is created from vertex and pixel shaders written in the High Level Shading Language (HLSL) and is completely customizable.
- Here are the steps involved with the custom effect
  - Design the **shaders** using **HLSL**. An effect usuablly one vertex and one pixel shader.
  - Create a **technique** that invokes the shaders; a technique determines which shaders are used. The technique and the shaders are usually stored in **an effect (.fx) file**.
  - Create an **effect object** in your game using the Effect class. This usually means loading the effect file with the Content Pipeline.
  - Initialize the **effect parameters**.
  - Render the effect by **applying the effect** to the device and rendering the scene.

XNA Effect Class

- XNA Framework includes a Effect class that is used to load and compile the shaders.
  - Create the shader (e.g.: transform.fx)
  - Put the shaderfile in "Contents"
  - Create an instance of the Effect class:
    - **Effect effect**
  - Initiate the instance of the Effect class
    - effect = Content.Load<Effect>("transform");
  - Select what technique you want to use
    - effect.CurrentTechnique = effect.Techniques["TransformTechnique"];
  - Set a shader parameter
    - effect.Parameters["World"].SetValue(world);
  - Pass different parameters to the shader; Apply it
  - Draw the scene/object

Set/Get Effect Parameters

- Setting effect parameter
  - **Effect.Parameters["pair.key"].SetValue("pair.value")** method

```
Matrix world, view, projection;
effect.Parameters["World"].SetValue(world);
effect.Parameters["View"].SetValue(view);
effect.Parameters["Projection"].SetValue(projection);
```

- Getting effect parameters
  - **Effect.Parameters[" "].GetValueXXX()** method

Set/Get Effect Parameters

Set Boolean

```
bool isFilled = true;
currentEffect.Parameters["isFilled"].SetValue(isFilled);
```

Set Float

```
float angle = 3.14f;
currentEffect.Parameters["angle"].SetValue(angle);
```

Set Float4

```
Vector4 v = new Vector4(1.0f, 1.0f, 0.0f, 1.0f);
currentEffect.Parameters["x"].SetValue(v);
```

Set Matrix

```
Matrix world(...);
currentEffect.Parameters["world"].SetValue(world);
```

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 parameters (world, view, and projection transformation matrix)
float4x4 World;
float4x4 View;
float4x4 Projection;

// 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 XNA that the data returned in this
// data member is a vertex position.
struct VertexShaderInput
{
    float4 Position : POSITION0;
};
```

Sample: Transform.fx

```
struct VertexShaderOutput
{
    float4 Position : POSITION0;
};
// 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).
VertexShaderOutput VertexShaderFunction(VertexShaderInput input)
{
    VertexShaderOutput output;
    // Transform to homogeneous clip space.
    float4 worldPosition = mul(input.Position, World);
    float4 viewPosition = mul(worldPosition, View);
    output.Position = mul(viewPosition, Projection);
    return output;
}
```

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 red.
float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0
{
    return float4(1.0f, 0.0f, 0.0f, 1.0f);      // red color
}
// entry point (technique)
technique TransformTechnique
{
    pass P0             // rendering pass
    {
        // Specify the vertex and pixel shader associated with this pass.
        vertexShader = compile vs_1_1 VertexShaderFunction();
        pixelShader  = compile ps_1_1 PixelShaderFunction();
        // Specify the render/device states associated with this pass.
        FillMode = Wireframe;
    }
}
```

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;

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))
```