TextureFX basics

1/Introduction

TextureFX in DirectX11 follows a new system which embeds multipass rendering, embedded renderers and which are now spreadable.

It introduces a new set of semantics and annotations which allows to create complex effects using a single file.

Altough it uses an integrated Vertex Shader (which renders a fulle screen quad under the hood), Usage of compute shaders is also possible in some extent.

Since their syntax are different, Texture FX uses the extension .tfx instead of .fx

2/My first TextureFX

So to start we will do the basic infamous Invert Shader.

To start, we declare the Pixel Shader layout, a Sampler and a Texture

```
struct vs2ps { float4 pos: SV_POSITION; float2 uv: TEXCOORD0; };

SamplerState linSamp : IMMUTABLE
{
	Filter = MIN_MAG_MIP_LINEAR;
	AddressU = Clamp;
	AddressV = Clamp;
};

Texture2D tex : INITIAL;
```

Pixel shader input is static and sends general data for a full screen quad.

The semantic INITIAL is assigned to the texture, by default a Texture FX creates a Texture In and a Texture Out pin, so using initial means we want to use Texture In instead of creating a new pin.

Also please note that the output texture will automatically inherit size and format from it's input, so no need for Info node.

Now here is our pixel shader:

```
float4 PS_Invert(vs2ps input): SV_Target
{
    float4 res = tex.Sample(linSamp,input.uv);
	res.rgb = 1.0 -res.rgb;
	return res;
}
```

As simple as it used to be.

To finish, we finally create the technique:

```
technique10 Invert
{
	pass P0
	{
		SetPixelShader( CompileShader( ps_4_0, PS_Invert() ) );
	}
}
```

Please note that we omit Vertex Shader, since it's automatically created and bound under the hood.

That's it, now we have a spreadable invert shader.

3/Creating several Passes

For our previous sample, we only needed a single pass to process our texture. But most post processing shaders require several passes.

So now let's say we want to multiply the inverted image by a constant color after the first pass. We can create another tfx file for it, but we can also do it in the same shader.

So first we need to create a color pin and a pixel shader for it (in the same file)

```
float4 cMul <bool color=true; >;


float4 PS_Mul(vs2ps input): SV_Target
{
    float4 res = tex.Sample(linSamp,input.uv);
       res.rgb *= cMul;
       return res;
}
```

Easy so far. Since we said we want to multiply after invert, we modify the Technique like this:

```
technique10 Processor
{
       pass PInvert
       {
              SetPixelShader( CompileShader( ps_4_0, PS_Invert() ) );
       }
       pass PMultiply
       {
              SetPixelShader( CompileShader( ps_4_0, PS_Mul() ) );
       }
}
```

I renamed the technique to processor, since now we don't do inversion only.

We also need a little amendment to our Texture declaration, which becomes:

```
Texture2D tex : PREVIOUS;
```

The PREVIOUS semantic instructs the shader to use the following logic:

- On the first pass, binds the Texture from the Texture In pin
- On any subsequent passes, use the result from the previous pass.

4/Rebinding intermediate results

Now we have done an Invert, and then we multiplied by a color.

Next step is now let's blend the Inverted Texture (result from first pass), with the multiplied Texture (result from the second pass).

We can access the multiplied texture using PREVIOUS semantic as before, but we now need access to the result from the first pass.

To do that, we can use the PASSRESULT[n] semantic, which allows to rebind the result of a specific pass.

So we create a new texture declaration like this:

```
Texture2D texinv : PASSRESULT0;
```

This will instruct the Texture FX engine to keep track of the result from the first Pass since we want to reuse it.

Now blend is as simple as that:

```
float alpha;

float4 PS_Blend(vs2ps input): SV_Target
{
    float4 res = tex.Sample(linSamp,input.uv);
       float4 res2 = texinv.Sample(linSamp,input.uv);
       res.rgb = lerp(res.rgb,res2.rgb,alpha);
       return res;
}
```

And we of course modify our technique:

```
technique10 Processor
{
       pass PInvert
       {
              SetPixelShader( CompileShader( ps_4_0, PS_Invert() ) );
       }
       pass PMultiply
       {
              SetPixelShader( CompileShader( ps_4_0, PS_Mul() ) );
       }
       pass PBlend
       {
              SetPixelShader( CompileShader( ps_4_0, PS_Blend() ) );
       }
}
```

That's about it!

5/Changing Formats and Generating Mips

Now we will create a new shader, and we want to do the following:

-Convert to grayscale

-Threshold the initial texture to have a threshold on it's average luminance (eg if color luma > average luma, return initial color, else return black).

So first we convert to grayscale, usual trivial operation:

```
Texture2D tex : INITIAL;

const float3 lumcoeff : IMMUTABLE = float3(0.2125,0.7154,0.0721);

float PS_Gray(vs2ps input): SV_Target
{
    float3 col = tex.Sample(linSamp,input.uv).rgb;
       return dot(col,lumcoeff);
}
```

Now we need to declare our pass, since we will want average, we also need to generate mips.

As we want a gray value, we will also output to a single channel texture.

```
technique10 Processor
{
       pass PGrayScale <bool mips=true; string format="R8_UNorm"; >
       {
              SetPixelShader( CompileShader( ps_4_0, PS_Gray() ) );
       }
}
```

Pretty self explanatory I suppose)))

Now we need to apply our threshold:

```
Texture2D graytex : PREVIOUS;

float4 PS_Threshold(vs2ps input): SV_Target
{
       float avg = graytex.SampleLevel(linSamp,0.0f,100).r;
       float luma = graytex.Sample(linSamp,input.uv).r;
       float4 col = tex.Sample(linSamp,input.uv);
       return luma >= avg ? col : 0;
}
```

So we sample a high level mipmap (to make sure we get highest value, it will clamp).

Since we already processed the luminance calculation, we can also sample this texture.

Next we get back the initial color (since tex variable was bound with the INITIAL semantic).

And to finish we do our technique:

```
technique10 Processor
{
       pass PGrayScale <bool mips=true; string format="R8_UNorm"; >
       {
              SetPixelShader( CompileShader( ps_4_0, PS_Gray() ) );
       }
       pass PThreshold < string format="R8G8B8A8_UNorm"; >
       {
              SetPixelShader( CompileShader( ps_4_0, PS_Threshold() ) );
       }
}
```

Please note than since Previous pass is grayscale, we make sure that output is 4 channels again.

6/ Resizing and default sizes

Now let's say we actually only want a texture generator (eg: no input texture).

We still need to define a size and a format for it.

We saw that format is simple (Pass result).

For size, we also have the Default Size pin (visible in inpector only by default).

So if texture in is connected, we inherit size, otherwise we use the one from the pin.

Now let's say we want also a simple relative resize shader.

Pixel shader is simple, just passtrough with sampling:

```
float4 PS(vs2ps input): SV_Target
{
    return tex.Sample(linSamp,input.uv);
}
```

Now let's say we want to downscale by 2, we can just annotate the pass like this:

```
technique10 Processor
{
	pass PResize < float scale=0.5f; >
	{
		SetPixelShader( CompileShader( ps_4_0, PS() ) );
	}
}
```

This will automatically instruct the processor to multiply the size by the factor provided.

That's it for TextureFX basics