FLSL - A short introduction to the Flare3D Shading Language

What's a shader?

It's a program that runs on the GPU of your computer instead of running on the CPU like any other program does. This allows you to control every aspect of the drawing process of your scene: From moving vertices to create mesh effects to defining the final color of every single pixel of your final render. Simple transformations, lighting and shadows, post-processing, all that can be done using shaders benefitting from the GPU's highly parallelized architecture to do it blazingly fast.

In vanilla Stage3D, shaders are written in a language called AGAL (Adobe Graphics Assembly Language) which, as the name implies, is a low-level assembly language.

What's FLSL?

FLSL is the high-level shader programming language used by Flare3D. With some similarities with other popular shader languages like GLSL, HLSL and PixelBender 3D, FLSL lets you perform the pixel and vertex operations at the same time, simplifying the development process. Of course, having experience with any of these languages is useful but with some patience everyone should be able to learn FLSL by exploring all the documentation and examples.

There are two distinct types of FLSL objects: FLSLMaterial and FLSLFilter. A FLSLMaterial is a complete shader, ready to be compiled and executed in your Flare3D application. A FLSLFilter on the other hand, is a small part of a shader that can be combined with others to create complex shader effects. This makes it possible to maximize code reutilization and modularization.

Hello FLSL World

What does FLSL look like?

```
use namespace flare.transforms;
use namespace flare.filters;

technique main()
{
        sampler2D texture;

        output vertex = transform();
        output fragment = textureMap( texture );
}
```

This one's an extremely simple shader, but it truly shows that FLSL can be much simpler and shorter compared to other shader languages. The first thing we do is to tell the compiler what parts of the flare FLSL namespace we want to use. This namespace is the default one and includes the basic types and functions of the FLSL library. You can of course add your own namespaces with your custom functionality.

Once that's taken care of, we declare a sampler2D object to hold our texture information and we set up our output vertex and fragment information using a simple transformation for our vertices and using our texture sampler to create a texture map in our fragment.

Now, let’s try something a little more complex:

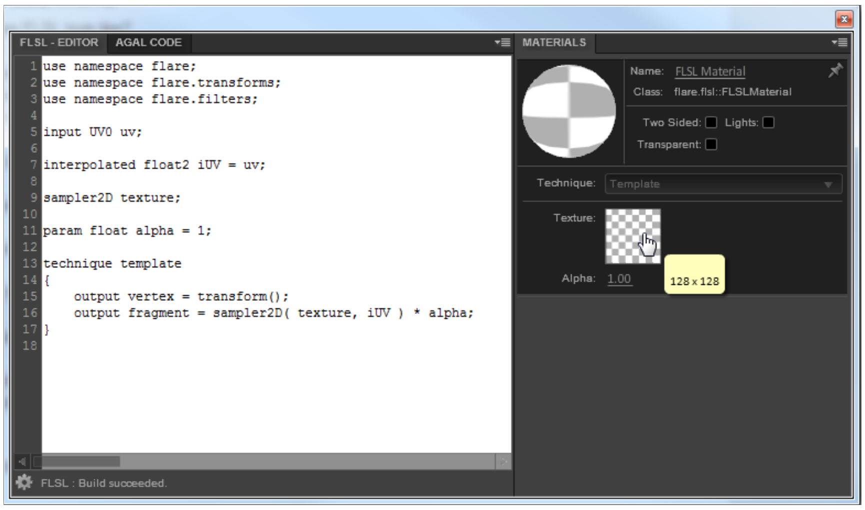
```
use namespace flare;
use namespace flare.transforms;
use namespace flare.filters;

input UV0 uv;

interpolated float2 iUV = uv;

sampler2D texture;

param float alpha = 1;

technique template
{
    output vertex = transform();
    output fragment = sampler2D( texture, iUV ) * alpha;
}
```

This is a fairly simple shader too, but it add some interesting features. It starts just like the previous one, importing all the namespaces we want to use.

After that, we start declaring some data to use later: We make a reference to the object UV0 channel and declare an interpolated variable to handle it.

> *A note on interpolated variables: This ones can be tricky to understand at first, but once you “get” them you’ll find that they’re very useful. You can think of interpolated variables as variables that vary their contents automatically through time. In this particular example, our iUV variable will always contain the current UV values for any given vertex.*

We also set up two other variables: A sampler2D to hold the texture data and another one to hold the alpha opacity value of our shader. If you check the material panel in Flare3D, you should see something like this:



```
use namespace flare;
use namespace flare.transforms;
use namespace flare.filters;

input UV0 uv;

interpolated float2 iUV = uv;

sampler2D texture;

param float alpha = 1;

technique template
{
    output vertex = transform();
    output fragment = sampler2D( texture, iUV ) * alpha;
}

```

Name: FLSL Material
Class: flare.flsl::FLSLMaterial
Two Sided: ☐ Lights: ☐
Transparent: ☐
Technique: Template
Texture:
Alpha: 1.00
128 x 128

FLSL : Build succeeded.

As you can see, the texture sampler and the alpha parameter were picked up by the editor and you can now customize their values directly in the editor without having to resort to writing code. Just drag and drop a JPEG or PNG file in the texture picker and your shader will use it and you’ll be able to see it inside the editor on real time.

Next, the main technique entry: Here we should output both our vertex and fragment information. In this example we're not making any changes to the vertex information, so a simple transform() call will just output it as it was received. For the fragment (pixel) information we basically get the current pixel information by using our texture and UV data and then multiply that for the alpha value received. By default our material will be fully opaque, but you can always change the value either in the code or in the editor and see how it affects the material.

Let's delve a little deeper into the shader world and take a little more "hands-on" approach to the same shader:

```
sampler2D texture;

param float alpha = 1;

float4 transform()
{
    param WORLD_VIEW_PROJ worldViewProj;
    input POSITION position;
    return position.xyzw * worldViewProj;
}

float4 textureMap( sampler2D texture )
{
    input UV0 uv;
    interpolated float2 iUV = uv;
    return sampler2D( texture, iUV )
}

technique template
{
    output vertex = transform();
    output fragment = textureMap( texture ) * alpha;
}
```

Again we declare a couple variables to hold our texture and opacity information, but this time, instead of using the helper functions contained in the flare namespace, we'll do things by hand.

Here we replicate the transform() function we used previously to actually see how it works: It simply creates a basic transform by multiplying the vertex position by the world view projection matrix.

We then create a textureMap function that receives a texture and returns the final pixel information for the current vertex. The rest is similar to the previous example: it'll take the texture and UV information to get the correct pixel information.

You can notice that we're declaring all the variables directly into our functions even though they bring information directly from the engine. This makes it very easy to use the same functions in different parts at the same time even if they need external information like the world view projection.

Finally, we create the main technique that uses our two functions to output correct vertex and fragment values.

The AS3 side

Now, this is all fine and dandy, but how do we use this from our AS3 project? The first thing is to compile our shader file. For this, must have Flare3D.exe installed correctly so it'll associate itself with FLSL (*.flsl) files. Just double click your shader file and if there aren't any errors, the FLSL compiler should generate a compiled version of your shader on the same folder of your original FLSL file. Compiled FLSL files carry the *.flsl.compiled extension.

Now let's take a look at the code to see how to load this into our project:

```
package
{
	import flare.basic.*;
	import flare.flsl.*;
	import flare.primitives.*;
	import flash.display.*;

	public class Main extends Sprite
	{
		[Embed(source = "../bin/assets/basic.flsl.compiled", mimeType = "application/
octet-stream")] private var flslMaterialAsset:Class;

		private var _scene:Scene3D;
		private var _cube:Cube;
		private var _flslMaterial:FLSLMaterial;

		public function Main():void
		{
			_scene = new Scene3D( this );
			_scene.camera.setPosition( 10, 20, -30 );
			_scene.camera.lookAt( 0, 0, 0 );

_flslMaterial = new FLSLMaterial( "FLSL Material", new flslMaterialAsset );
			_cube = new Cube( "cube", 10, 10, 10, 1, _flslMaterial );
			_scene.addChild( _cube );
		}
	}
}
```

As you can see, we only have to embed the compiled file, create a new FLSLMaterial object instance and feed it the shader bytecode we embedded. Once that's done, we can assign our FLSLMaterial to any Mesh3D object we want, just like we would do with a standard Material3D or Shader3D material.

This will just load the FLSL code and set it as material, but it's also possible to set certain values during runtime. For example, changing a texture sampler by code is as simple as doing:

```
// myTexture is a Texture3D object reference
_flslMaterial.params.texture.value = myTexture;
```

The general rule to access this data is to use FLSLMaterial.params.[FLSL variable name].value. In the previous example we used "texture" because that's how we had named our sampler2D before.

One more example: Instead of creating a FLSLMaterial, let's try creating a FLSLFilter and then adding that to a standard Shader3D. For the FLSL code, let's create a simple filter that animates a scrolling texture:

```
use namespace flare;
use namespace flare.transforms;
use namespace flare.filters;

sampler2D texture;
param TIME time;

technique template
{
        output fragment = textureMap( texture, float2(1), time.xy );
}
```

You probably already noticed, but as you can see, there's no need to specify a vertex output when creating filters. In our fragment output, we use a textureMap call with parameters, the actual reference is as follows:

```
float4 textureMap( sampler2D texture, float2 repeat, float2 offset );
```

TIME is a float4 parameter provided by FLSL and each position of the float4 contains a different time value in ranges from 0 to 1: x -> 1 second, y -> ¼ second, z -> ⅛ second and w -> 1/16 second. Since we're passing this values into the offset parameter of textureMap, the texture will appear to be scrolling. But before that, we have to set up everything in AS3:

```
package
{
        import flare.basic.*;
        import flare.core.*;
        import flare.flsl.*;
        import flare.materials.*;
        import flare.materials.filters.*;
        import flare.primitives.*;
        import flash.display.*;

        public class Main extends Sprite
        {
                [Embed(source = "../bin/assets/flsl_filter.flsl.compiled", mimeType
="application/octet-stream")] private var flslFilterAsset:Class;
                [Embed(source = "../bin/assets/akmat1.png")] private var textureAsset:Class;

                private var _scene:Scene3D;
                private var _cube:Cube;
                private var _flslFilter:FLSLFilter;
                private var _texture:Texture3D;
                private var _shader:Shader3D;

                public function Main():void
                {
                        _scene = new Viewer3D( this );
                        _scene.camera.setPosition( 10, 20, -30 );
                        _scene.camera.lookAt( 0, 0, 0 );

                        _flslFilter = new FLSLFilter( new flslFilterAsset, BlendMode.NORMAL );
                        _flslFilter.params.texture.value = new Texture3D( new textureAsset );

                        _shader = new Shader3D( "", [ _flslFilter, new ColorFilter( 0xff0000,
0.5,BlendMode.ADD ) ] );

                        _cube = new Cube( "cube", 10, 10, 10, 1, _shader );
                        _scene.addChild( _cube );
                }
        }
}
```

We create a simple Flare3D scene with a Cube mesh and then we set up a standard Shader3D material. We load a texture by hand and then we use that in our FLSLFilter sampler, just like we would do in a FLSLMaterial. The actual shader is very simple and only contains our scrolling texture filter and a separate ColorFilter with additive blending. The good thing of this FLSLFilter is that you can use it in multiple materials in a very modular way.

FLSL Function Reference

Top level

Float constuctors

float4 float4(float a)
float4 float4(float a, float b, float c, float d)
float4 float4(float2 a, float b, float c)
float4 float4(float a, float2 b, float c)
float4 float4(float a, float b, float2 c)
float4 float4(float2 a, float2 b)
float4 float4(float3 a, float b)
float4 float4(float a, float3 b)
float3 float3(float a, float b, float c)
float3 float3(float2 a, float b)
float3 float3(float a, float2 b)
float3 float3(float a)
float2 float2(float a, float b)
float2 float2(float a)

Constants constructor

float const1(float a)
float2 const2(float a, float b)
float3 const3(float a, float b, float c)
float4 const4(float a, float b, float c, float d)

Matrix Multiplication

float3 mul3x3(float3 a, float3 b)
float3 mul4x3(float4 a, float4 b)
float3 mul4x3(float4 a, float4x4 b)
float3 mul4x3(float4 a, float4x3 b)
float3 mul3x3(float3 a, float4x4 b)
float3 mul3x3(float3 a, float3x3 b)
float4 mul4x4(float4 a, float4x4 b)

Standard AGAL functions

void kill(float src)
float4 rcp(float4 a)
float3 rcp(float3 a)
float2 rcp(float2 a)
float rcp(float a)
float4 min(float4 a, float4 b)
float3 min(float3 a, float3 b)
float2 min(float2 a, float2 b)
float min(float a, float b)
float4 min(float4 a, float b)
float3 min(float3 a, float b)
float2 min(float2 a, float b)
float4 max(float4 a, float4 b)
float3 max(float3 a, float3 b)
float2 max(float2 a, float2 b)
float max(float a, float b)
float4 max(float4 a, float b)
float3 max(float3 a, float b)
float2 max(float2 a, float b)
float4 frac(float4 a)
float3 frac(float3 a)
float2 frac(float2 a)
float frac(float a)
float4 sqrt(float4 a)
float3 sqrt(float3 a)
float2 sqrt(float2 a)
float sqrt(float a)
float4 rsq(float4 a)
float3 rsq(float3 a)
float2 rsq(float2 a)
float rsq(float a)
float4 pow(float4 a, float4 b)
float3 pow(float3 a, float3 b)
float2 pow(float2 a, float2 b)
float4 pow(float4 a, float b)
float3 pow(float3 a, float b)
float2 pow(float2 a, float b)
float pow(float a, float b)
float4 log(float4 a)

```
float3 log( float3 a )
float2 log( float2 a )
float  log( float  a )
float4 exp( float4 a )
float3 exp( float3 a )
float2 exp( float2 a )
float  exp( float  a )
float3 normalize( float3 a )
float2 normalize( float2 a )
float4 sin( float4 a )
float3 sin( float3 a )
float2 sin( float2 a )
float  sin( float  a )
float4 cos( float4 a )
float3 cos( float3 a )
float2 cos( float2 a )
float  cos( float  a )
float3 cross( float3 a, float3 b )
float2 cross( float2 a, float2 b )
float  dot( float2 a, float2 b )
float  dot( float3 a, float3 b )
float  dot( float4 a, float4 b )
float4 abs( float4 a )
float3 abs( float3 a )
float2 abs( float2 a )
float  abs( float  a )
float4 neg( float4 a )
float3 neg( float3 a )
float2 neg( float2 a )
float  neg( float  a )
float4 saturate( float4 a )
float3 saturate( float3 a )
float2 saturate( float2 a )
float  saturate( float  a )
float4 sampler2D( sampler2D texture, float2 uv )
float4 samplerCube( samplerCube texture, float3 normal )
```

Extended FLSL functions

```
float4 floor( float4 a );
float3 floor( float3 a );
float2 floor( float2 a );
float  floor( float  a );
float4 ceil( float4 a );
float3 ceil( float3 a );
float2 ceil( float2 a );
float  ceil( float  a );
float4 clamp( float4 a, float4 min, float4 max );
float3 clamp( float3 a, float3 min, float3 max );
float2 clamp( float2 a, float2 min, float2 max );
float  clamp( float  a, float  min, float  max );
float4 clamp( float4 a, float  min, float  max );
```

float3 clamp(float3 a, float min, float max);
float2 clamp(float2 a, float min, float max);
float3 reflect(float3 v, float3 normal);
float3 refract(float3 v, float3 normal, float eta);
float lerp(float a, float b, float w);
float2 lerp(float2 a, float2 b, float w);
float3 lerp(float3 a, float3 b, float w);
float4 lerp(float4 a, float4 b, float w);
float2 lerp(float2 a, float2 b, float2 w);
float3 lerp(float3 a, float3 b, float3 w);
float4 lerp(float4 a, float4 b, float4 w);
float smoothstep(float a, float b, float x);
float2 smoothstep(float2 a, float2 b, float2 x);
float3 smoothstep(float3 a, float3 b, float3 x);
float4 smoothstep(float4 a, float4 b, float4 x) ;
float length(float4 a);
float length(float3 a);
float length(float2 a);
float distance(float4 a, float4 b);
float distance(float3 a, float3 b);
float distance(float2 a, float2 b);
float interpolate(float value);
float2 interpolate(float2 value);
float3 interpolate(float3 value);
float4 interpolate(float4 value);

flare.blendMode namespace

float4 mix(float4 a, float4 b, float4 value);
float4 mix(float4 a, float4 b, float value);
float4 base(float4 a, float4 b);
float4 normal(float4 a, float4 b);
float4 add(float4 a, float4 b);
float4 subtract(float4 a, float4 b);
float4 multiply(float4 a, float4 b);
float4 divide(float4 a, float4 b);
float4 lighten(float4 a, float4 b);
float4 screen(float4 a, float4 b);
float4 darken(float4 a, float4 b);
float4 difference(float4 a, float4 b);
float4 invert(float4 a);
float4 invert(float4 a, float4 b);
float4 alpha(float4 a, float4 b);
float4 erase(float4 a, float4 b);
float4 overlay(float4 a, float4 b);
float4 hardlight(float4 a, float4 b);
float4 average(float4 a, float4 b);
float4 source(float4 a, float4 b);
float4 dest(float4 a, float4 b);

flare.transforms namespace

float4 transform()

float4 skin1();
float4 skin2();
float4 skin3();
float4 skin4();

flare.filters namespace

float4 textureMap(sampler2D texture)
float4 textureMap(sampler2D texture, float2 repeat)
float4 textureMap(sampler2D texture, float repeat)
float4 textureMap(sampler2D texture, float2 repeat, float2 offset)

Matrix Semantics

WORLD_VIEW_PROJ float4x4;
WORLD_VIEW float4x4;
VIEW_PROJ float4x4;
PROJ float4x4;
WORLD float4x4;
IWORLD float4x4;
VIEW float4x4;
CAMERA float4x4;

Input Semantics

POSITION float3;
NORMAL float3;
TANGENT float3;
BITANGENT float3;
SKIN_WEIGHTS float2;
SKIN_INDICES float2;
PARTICLE float4;
UV0 float2;
UV1 float2;
UV2 float2;
UV3 float2;
COLOR0 float3;
COLOR1 float3;
COLOR2 float3;
TARGET_POSITION float3;
TARGET_NORMAL float3;

Default Semantics

TIME float4;
COS_TIME float4;
SIN_TIME float4;
MOUSE float4;
CAM_POS float3;
VIEWPORT float4;

NEAR_FAR float4;
BONES float4;
TARGET_TEXTURE sampler2D;

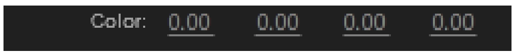
Defined Constants

PI 3.141592653589793;
PI2 6.283185307179586;

FORMAT_RGBA 0;
FORMAT_COMPRESSED 1;
FORMAT_COMPRESSED_ALPHA 2;
WRAP_CLAMP 0;
WRAP_REPEAT 1;
MIP_NONE 0;
MIP_NEAREST 1;
MIP_LINEAR 2;
FILTER_NEAREST 0;
FILTER_LINEAR 1;

Flare Editor metadata

It's possible to add metadata directly into your FLSL that will be picked up by Flare Editor. For example, if we declare a float4 variable called "color", the editor will display it like this:

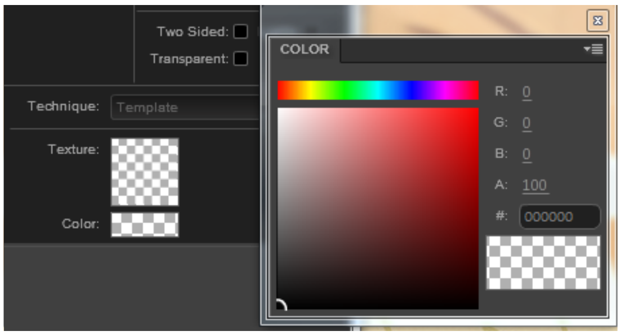


Color: 0.00 0.00 0.00 0.00

If we declare it with some metadata like this:

```
param float4 color <ui = "color">;
```

It will look like this:



The UI control will change into a fully working color picker. On the other hand, if you don't want the parameter to be available from the editor, it's possible to set it like this:

```
param float4 color <ui = "none">;
```

And then it won't appear on the Material panel.
You can also force the order of the parameter using the metadata order = "2" for example.

There are other types of metadata for float values like min, max that defines the range of allowed values for that parameter.

What’s next?

We have included a few more interesting ones in our flare3d 2.5.13 beta package in the “examples/flsl” folder. Of course, you can always check our wiki, or the brand new FLSL section of our forums.