Writing shaders for Gideros Mobile

A quick guide

Index

Preamble

As you may know, most of graphics hardware now allow processing of user code directly on the GPU. This means two things:

- you can relieve the CPU from performing some graphics-related computation by letting the GPU perform those computations and thus gain overall performance in your app
- you can achieve beautiful graphic effects by modifying the per-pixel rendering process

Those little pieces of code you can upload to the GPU are called shaders.

Gideros recently abandoned support for legacy rendering mode, also referred as fixed functionality rendering pipeline, and embraced the shaders system, i.e. programmable pipeline. The next step was to let Gideros users access the shaders system themselves, opening a broad range of new possibilities.

Shaders and Gideros

At the time we write those lines, Gideros internally uses five distinct shaders:

- the 'Basic' shader handle shapes with a constant color
- the 'Color' shader handle shapes with per-vertex colors (mostly used by Mesh sprite)
- the 'Texture' shader handle textured shapes (Bitmaps)
- the 'TextureColor' shader handle textured and per-vertex colored shapes
- and the 'Particle' shader deals with Box2D particle systems

The new shader API allows to replace the default shader used by Gideros with a custom one, on a sprite per sprite basis. As with most of Gideros API's this one is straight-forward: create a Shader object and assign it to one or several sprites.

```
local myShader=Shader.new(…shader parameters…)
mySprite:setShader(myShader)
```

That said, since Gideros will use your shader as if it was the standard one, you will have to make sure that your custom shader is compatible with the standard one, which essentially means that it takes the same input parameters. But let's start with a deeper insight about shaders.

Shaders types, Attributes and Uniforms

First thing: the rendering pipeline has several chained processing stages, which is why we call it a pipeline. A few stages are programmable, each requiring a particular type of program or shader. Gideros uses and let user access two of them: the vertex shader and the fragment shader (also called pixel shader).

Each shader process one or more stream of data, which we will call attributes as per OpenGL semantics, and further produce one of more stream of (processed) data.

Each shader also has access to one or more constants called uniforms. Those constants are settable at will before invoking the shader program, so that they are in fact only constants from the shader program point of view, and because their values can't be changed in the middle of the processing of a stream of data.

Vertex Shaders

The first processing stage that Gideros deals with is the Vertex shader. As its name implies, its purpose is to manipulate vertices for the geometry being rendered. No real big deal here, most of the work of a vertex shader is transforming geometry coordinates from sprite local space to actual window coordinates. Other input data is most of the time just passed along unmodified to the next stage.

Vertex shaders typically uses the following attributes:

- vertex coordinates (always)
- texture coordinates (if required)
- vertex color (if required)

Fragment Shaders

The fragment shader is the most interesting. It takes its inputs from the vertex shader's outputs but is applied to every pixel of the shape being rendered. Input data from the vertex shader is interpolated and then fed to the fragment shader. The final goal of the fragment shader is to produce the color of the pixel to actually draw.

Fragment shaders typically uses the following attributes:

- interpolated texture coordinates (if required)
- interpolated pixel color (if required)

Writing shaders programs

Plenty of literature is available to learn writing shaders, so we won't go into language details here.

Depending on your target platforms, you will need to write your shaders in one or two forms: GLSL for OpenGL based platforms (Windows desktop, Android, iOS), and HLSL for Direct3D based platforms (currently WinRT). As quick advice: if you need to write both, then start with the HLSL version which is in many ways more restrictive than the GLSL. It is far easier to port HLSL code to GLSL than the opposite way.

GLSL comes in two flavor: the regular language developed for desktop platforms, and the GLSL for embedded system, aka OpenGL ES 2.0. The latter is often more strict and needs additional precision modifiers. Gideros helps you to deal with those differences by removing precision modifiers if run on a desktop platform and setting the appropriate language version. That way you only need to write an OpenGL ES 2.0 compliant shader.

If you are a beginner, it is better to start with a working shader, from an example or from this documentation, and then modify it to suit your needs.

You can avoid common compatibility pitfalls by reading this: https://www.opengl.org/wiki/GLSL_:_common_mistakes

Defining shaders from lua

Shader definition

The new API allows the creation of Shader objects from lua. The 'Shader.new()' constructor takes five arguments:

- The path and name for the vertex shader without its extension. Gideros will search the assets for a file with the supplied name, automatically adding the extension relevant for the target platform: .glsl for OpenGL, .cso or .hlsl for DirectX.
- The path and name for the fragment shader without its extension. Same remark as above applies too.
- A set of numerical flags or 0 if none. See description below.
- An array of uniforms/constants descriptors
- An array of attributes descriptors

Flags

A single flag is currently defined.

Shader.FLAG_NO_DEFAULT_HEADER : Suppress the automatic addition of a default header for GLSL programs.

The default header folder desktop OpenGL is:

```
#version 120
#define highp
#define mediump
#define lowp
```

And for OpenGL ES 2.0:

```
#version 100
#define GLES2
```

Uniform descriptors

Each uniform descriptor entry is a lua table with the following fields:

- name: Name of the uniform. Must match the name used in GLSL program
- type: Type of data this uniform holds. Must be one of Shader.CFLOAT,Shader.CFLOAT4,Shader.CINT,Shader.CMATRIX or Shader.CTEXTURE
- mult: The number of elements if this uniform is an array
- vertex: Boolean which must be set to true if the uniform is defined in the vertex shader, and false if it is defined in the fragment shader
- sys: Indicate that this uniform has a special meaning and should be set by Gideros when appropriate. Current possible values are:
    - Shader.SYS_WVP: this uniform of type CMATRIX will hold the World/View/Projection matrix.
    - Shader.SYS_COLOR: this uniform of type CFLOAT4 will hold the constant color to be used to draw the shape

- Shader.SYS_WORLD: this uniform of type CMATRIX will hold the World transform matrix.
- Shader.SYS_WIT: this uniform of type CMATRIX will hold the World Inverse Transpose matrix.
- Shader.SYS_TEXTUREINFO: this uniform of type CFLOAT4 will hold information about the texture used in a particle shader.
- Shader.SYS_PARTICLESIZE: the uniform of type CFLOAT will hold the particle size to use in a particle shader.

The uniforms must be declared in the order they appear in the constant block of the HLSL version of the shader. If shaders are only GLSL, then the order is not relevant.

Attributes descriptor

Each attribute descriptor entry is a lua table with the following fields:

- name: the name of the attribute/vertex data stream. Must match the name used in GLSL and HLSL code
- type: data type used from the host point of view. One of Shader.DFLOAT, Shader.DBYTE, Shader.DUBYTE, Shader.DSHORT, Shader.DUSHORT or Shader.DINT.
- mult: number of components of the input vector
- slot: index of the input slot (HLSL only)
- offset: offset within the input slot (HLSL only, should be 0)

The first three attributes have fixed meaning meaning. Those are, in order:

- The vertex coordinate stream. Type must be Shader.DFLOAT and mult should be 3.
- The per-vertex color. Type must be Shader.DUBYTE and mult should be 4.
- The texture coordinates. Type must be Shader.DFLOAT and mult should be 2.

If one of these fixed attributes is not used by the custom shader program, it should still be defined with a mult value of 0 to serve as a placeholder.

Changing uniforms/constants

In order to change the value of a uniform from lua, use the setConstant function.

It takes three arguments:

- The uniform name
- The type if data to set (one of the Shader.Cxxx constants)
- And the actual data to set, either as a table or as multiple arguments

Associating a shader to a sprite

Sprite API has a new call to deal with that: Sprite:setShader(shader) tells Gideros to use the specified shader for rendering the sprite. Setting back the shader to nil actually revert to the default shader.

An example explained

Example shaders code

Vertex shader, GLSL version (vShader.glsl)

```
attribute highp vec3 vVertex;
attribute mediump vec2 vTexCoord;
uniform highp mat4 vMatrix;
varying mediump vec2 fTexCoord;

void main() {
  vec4 vertex = vec4(vVertex,1.0);
  gl_Position = vMatrix*vertex;
  fTexCoord=vTexCoord;
}
```

Vertex shader, HLSL version (vShader.hlsl)

```
struct VOut
{
	float4 position : SV_POSITION;
	float2 texcoord : TEXCOORD;
};

cbuffer cbv : register(b0)
{
	float4x4 vMatrix;
};

VOut VShader(float4 position : vVertex, float2 texcoord : vTexCoord)
{
	VOut output;

	position.w = 1.0f;

	output.position = mul(vMatrix, position);
	output.texcoord = texcoord;

	return output;
}
```

Fragment shader, GLSL version (fShader.glsl)

```
uniform lowp vec4 fColor;
uniform lowp sampler2D fTexture;
uniform int fRad;
uniform mediump vec4 fTexelSize;
varying mediump vec2 fTexCoord;

void main() {
 lowp vec4 frag=vec4(0,0,0,0);
 int ext=2*fRad+1;
 mediump vec2 tc=fTexCoord-fTexelSize.xy*fRad;
 for (int v=0;v<ext;v++)
 {
      frag=frag+texture2D(fTexture, tc);
    tc+=fTexelSize.xy;
 }
 frag=frag/ext;
 if (frag.a==0.0) discard;
 gl_FragColor = frag;
}
```

Fragment shader, HLSL version (fShader.hlsl)

```
Texture2D myTexture : register(t0);
SamplerState samLinear : register(s0);

cbuffer cbp : register(b1)
{
      float4 fColor;
      float4 fTexelSize;
      int fRad;
};

float4  PShader(float4  position  :  SV_POSITION,  float2  texcoord  :
TEXCOORD) : SV_TARGET
{
      float4 frag=float4(0,0,0,0);
      int ext=(2*fRad+1);
      float2 tc=texcoord-fTexelSize.xy*fRad;
      [unroll(21)]
      for (int v=0;v<ext;v++)
      {
            frag=frag+myTexture.Sample(samLinear, tc)/ext;
            tc+=fTexelSize.xy;
      }
      if (frag.a == 0.0) discard;
      return frag;
}
```

Lua code

Shader definition

```
local shader=Shader.new("vShader","fShader",0,
{
{name="vMatrix",type=Shader.CMATRIX,sys=Shader.SYS_WVP,vertex=true},
{name="fColor",type=Shader.CFLOAT4,sys=Shader.SYS_COLOR,vertex=false
},
{name="fTexture",type=Shader.CTEXTURE,vertex=false},
{name="fTexelSize",type=Shader.CFLOAT4,vertex=false},
{name="fRad",type=Shader.CINT,vertex=false},
},
{
{name="vVertex",type=Shader.DFLOAT,mult=3,slot=0,offset=0},
{name="vColor",type=Shader.DUBYTE,mult=4,slot=1,offset=0},
{name="vTexCoord",type=Shader.DFLOAT,mult=2,slot=2,offset=0},
});
```

The code above defines a shader constiting of vShader.{glsl/hlsl} vertex shader and fShader.{glsl/hlsl} fragment shader. No special flag is given.

Five uniforms are also defined, named vMatrix,fColor,fTexture,fTexelSize and fRad. Only the first one is associated to the vertex program (you can check that in the respective programs seen earlier).

Furthermore, we ask Gideros to take responsibility of setting vMatrix to the Matrix/View/Projection matrix (SYS_WVP), and fColor to the fixed color of the shape being rendered (SYS_COLOR)

fTexture is actually a texture (Shader.CTEXTURE), and declared appropriately in GLSL or HLSL.

fTexelSize and fRad have not specific meaning, and will be set by the lua code.

The three standard attributes are also defined.

Changing uniforms

```
shader:setConstant("fRad",Shader.CINT,1,0) --Initial blur level
shader:setConstant("fTexelSize",Shader.CFLOAT4,1,{1/texw,1/texh,0,0}
) --Initial texel size
```

Applying to a sprite

```
spr:setShader(shader)
```