1 LIBGDX Shader Program

UNIDADE 2: Creando o mundo

1.1 Sumario

- 1 Introdución
- 2 Onde gardar o Program Shader
    - 2.1 Arquivo externo
    - 2.2 Definición interna
- 3 Funcionamento do ShaderProgram
    - 3.1 Traballando coa posición
    - 3.2 Traballando coa cor
    - 3.3 Traballando coa textura
    - 3.4 Traballando coa cámara

1.1.1 Introdución

Información: https://github.com/mattdesl/lwjgl-basics/wiki/Shaders

A partires da versión OPEN GL 2.0 os gráficos renderízanse utilizando Shader´s. Os Shader son programas escritos en linguaxe parecida a C que se denomina GLSL e que permiten modificar as propiedades dos puntos que se van renderizar podendo facer efectos moi complexos.

O proceso de aplicar un Shader Program a unha figura divídese en dúas etapas:

- Vertex Shader: aplícanse operacións a cada un dos vértices da figura. Por exemplo, poderíamos modificar a posición e tamaño dun obxecto 3D.
- Fragment Shader: é a segunda etapa. A diferenza coa anterior é que se vai aplicar a cada fragmento da figura. Por simplificar imos identificar fragmento como pixel. Como podedes pensar o custe de GPU será maior que no caso anterior, por lo que non se debe abusar deste mecanismo. Por exemplo, poderíamos cambiar a cor de cada pixel da figura.

1.1.2 Onde gardar o Program Shader

O Shader Program pódese definir na propia clase ou ben nun arquivo externo.

1.1.2.1 Arquivo externo

- Temos que gardar o Vertex Shader nun arquivo con extensión .vert (non é obrigatorio).
- Temos que gardar o Fragment Shader nun arquivo con extensión .frag (non é obrigatorio).

Vexamos un exemplo sinxelo, no que simplemente asinamos a posición a cada vértice dun triángulo.

Preparación: Crear unha clase de nome UD4_4_ProgramShader, que derive da clase Game e sexa chamada pola clase principal das diferentes versións (desktop, android,...).

Creamos dous arquivos que colocaremos no cartafol assets da versión Android e terán de nome:

- vertex.vert

```
attribute vec4 a_position;
void main()
{
  gl_Position = a_position;
}
```

Comentarios do código:

- Liña 4: nesta liña copiamos a posición de cada vértice a un Vec4 interno de ShaderProgram, que será o dato que envía á GPU (Graphics Procesor Unit).

- Liña 1: definimos unha variable de tipo vec4(vector de 4 dimensións). Esta variable é un parámetro onde van chegar cada unha das posicións dos vértices definidos no Mesh. E vos estaredes preguntando, pero as posicións son Vector3(x,y,z) como pode ser que o almacene un Vector4 ? Pois debido a que internamente traballa con matrices de 4x4. O que fai é encher o último número cun 1.

  Tamén poderíamos poñer:

```
attribute vec3 a_position;
void main()
{
  gl_Position = vec4(a_position,1);
}
```

- fragment.frag

```
#ifdef GL_ES
  precision mediump float;
#endif
void main()
{
}
```

Agora mesmo non fai nada. As liñas 1 a 3 define a precisión dos vectores. Sempre se debe de poñer as liñas anteriores xa que esa é a precisión adecuada para traballar con texturas e posicións.

1.1.2.2 Definición interna

Tamén podemos definir o contido dos arquivos en forma de texto dentro do código da clase da seguinte forma:

Código da clase UD4_4_ProgramShader
Obxectivo: Definimos os contidos do Program Shader internamente.

```
@Override
public void create() {
// TODO Auto-generated method stub

// create shader program
String vertexShader = "attribute vec4 "+ ShaderProgram.POSITION_ATTRIBUTE +";\n"
+ "void main()                                    \n"
+ "{                                                \n"
+ "     gl_Position = "   + ShaderProgram.POSITION_ATTRIBUTE +";\n"
+"}         ;                                                 \n"
String fragmentShader = "#ifdef GL_ES   \n"
+ "precision mediump float;             \n"
+ "#endif                                           \n"
+ "void main()                                  \n"
+ "{                                                \n"
+ "}         ;                                      \n"
           }
```

Comentar no código que podemos facer uso das constantes da clase ShaderProgram para acceder ó nome dos atributos que van recoller os datos enviados polo Mesh de posición, cor e textura. No exemplo anterior, poñer:

```
"attribute vec4 "+ ShaderProgram.POSITION_ATTRIBUTE +";\n"
```

é equivalente a poñer:

```
"attribute vec4 a_position;\n"
```

1.1.3 Funcionamento do ShaderProgram

Unha vez decido onde vai ir o código analizaremos o funcionamento dos Shader´s.

Nota: A partires de aquí usaremos a opción de gardar o arquivo de forma externa en dous arquivos.

Como dixemos anteriormente o proceso de aplicar un shader consta de dous pasos. Nun primeiro paso se aplican transformacións ós vértices da figura e nun segundo paso se aplican transformacións a fragmentos (pixeles) da figura.

1.1.3.1 Traballando coa posición

Inicialmente nos temos que pasarlle ó programa información como pode ser a posición de cada vértice. Para pasarlle información usaremos a liña:

attribute vec4 a_position;

Coa palabra clave atribute estamos a definir un parámetro. Este parámetro (o da posición) ten un nome xa predefinido e un tipo (Vector4).

Cando definimos o Mesh da forma:

trianguloRed = new Mesh(true, 3, 3, VertexAttribute.Position());

Estamos a dicir que imos enviar como datos a información da posición ó parámetro a_position definido no programa vertex.vert.

O código completo:

Código da clase UD4_4_ProgramShader
Obxectivo: Ver como funciona o Shader Program.

```
import com.badlogic.gdx.Game;
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.Mesh;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.VertexAttribute;
import com.badlogic.gdx.graphics.glutils.ShaderProgram;

/**
 *  Funcionamento do ShaderProgram
 * @author ANGEL
 */

public class UD4_4_ProgramShader extends Game {

private Mesh trianguloRed;
private ShaderProgram shaderProgram;

private OrthographicCamera camara2d;

@Override
public void create() {
// TODO Auto-generated method stub

shaderProgram = new ShaderProgram(Gdx.files.internal("vertex.vert"), Gdx.files.internal("fragment.frag"));
if (shaderProgram.isCompiled() == false) {
Gdx.app.log("ShaderError", shaderProgram.getLog());
System.exit(0);
}

trianguloRed = new Mesh(true, 3, 3, VertexAttribute.Position());
trianguloRed.setVertices(new float[] {
-0.5f, -0.5f, 0f,
0.5f, -0.5f, 0f,
0f, 0.5f, 0f,
});
trianguloRed.setIndices(new short[] {0, 1, 2});


}
```

```
@Override
public void render() {

Gdx.gl20.glClearColor(1f, 1f, 1f, 1f);
Gdx.gl20.glClear(GL20.GL_COLOR_BUFFER_BIT|GL20.GL_DEPTH_BUFFER_BIT);


Gdx.gl20.glEnable(GL20.GL_DEPTH_TEST);

shaderProgram.begin();

trianguloRed.render(shaderProgram, GL20.GL_TRIANGLES,0,3);

shaderProgram.end();

Gdx.gl20.glDisable(GL20.GL_DEPTH_TEST);

}

@Override
public void dispose(){
shaderProgram.dispose();
trianguloRed.dispose();

}

}
```

Podemos consultar neste enlace as operacións que podemos facer sobre os tipos de datos da linguaxe GLSL (coa que definimos o Program Shader).

Por exemplo, podemos facer que o triángulo se estreita desta forma:

```
attribute vec3 a_position;
void main()
{
  vec4 v = vec4(a_position,1);
  v.x = v.x/4;
  gl_Position = v;
}
```

TAREFA 4.3 A FACER: Esta parte está asociada á realización dunha tarefa.

1.1.3.2 Traballando coa cor

O proceso para traballar coa cor é o mesmo que no caso da posición.

A nome do parámetro a definir no arquivo vertex.vert é **a_color** ou se estamos a definir o Program Shader internamente poderíamos utilizar a constante: **ShaderProgram.COLOR_ATTRIBUTE**.

```
attribute vec4 a_color;
```

Nota: Fixarxe como a cor é un vector4 xa que enviamos RGBA.

O parámetro temos que definilo no vertex.vert pero o valor da cor non se vai utilizar neste arquivo se non que ten que pasarse ó Fragment Shader.

Para pasalo temos que definir unha variable de tipo varying. O que facemos é pasarlle o valor (a cor) ó parámetro a_color, despois gardaremos dito valor na variable de tipo varying v_color e dende o Fragment Program (o arquivo fragment.frag) podemos acceder a dito valor.

- Arquivo vertex.vert

```
attribute vec3 a_position;
attribute vec4 a_color;
varying vec4 v_color;
void main()
{
  gl_Position =  vec4(a_position,1);
  v_color = a_color;
}
```

Agora modificaremos o arquivo fragment.frag e recoller o dato e pasalo á variable **gl_FragColor**, que é o que se vai pasar á GPU.

- Arquivo fragment.frag

```
#ifdef GL_ES
  precision mediump float;
#endif
varying vec4 v_color;
void main()
{
  gl_FragColor = v_color;
}
```

Por último modificamos o Mesh para enviarlle os datos da cor:

```
package com.plategaxogo3d.exemplos;

import com.badlogic.gdx.Game;
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.Mesh;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.VertexAttribute;
import com.badlogic.gdx.graphics.glutils.ShaderProgram;

/**
 * Funcionamento do ShaderProgram
 * @author ANGEL
 */

public class UD4_4_ProgramShader extends Game {

private Mesh trianguloRed;
private ShaderProgram shaderProgram;

private OrthographicCamera camara2d;

@Override
public void create() {
// TODO Auto-generated method stub

shaderProgram = new ShaderProgram(Gdx.files.internal("vertex.vert"), Gdx.files.internal("fragment.frag"));
if (shaderProgram.isCompiled() == false) {
Gdx.app.log("ShaderError", shaderProgram.getLog());
System.exit(0);
}

trianguloRed = new Mesh(true, 3, 3, VertexAttribute.Position(), VertexAttribute.ColorUnpacked());
trianguloRed.setVertices(new float[] {
-0.5f, -0.5f, 0f, 1f, 0f, 0f, 1f,
0.5f, -0.5f, 0f, 1f, 0f, 0f, 1f,
0f, 0.5f, 0f, 1f, 0f, 0f, 1f
});
trianguloRed.setIndices(new short[] {0, 1, 2});


}
```

```
@Override
public void render() {

Gdx.gl20.glClearColor(0f, 0f, 0f, 1f);
Gdx.gl20.glClear(GL20.GL_COLOR_BUFFER_BIT|GL20.GL_DEPTH_BUFFER_BIT);


Gdx.gl20.glEnable(GL20.GL_DEPTH_TEST);

shaderProgram.begin();

trianguloRed.render(shaderProgram, GL20.GL_TRIANGLES,0,3);

shaderProgram.end();

Gdx.gl20.glDisable(GL20.GL_DEPTH_TEST);

}

@Override
public void dispose(){
shaderProgram.dispose();
trianguloRed.dispose();

}

}
```

1.1.3.3 Traballando coa textura

Xa vimos anteriormente como asociar unha textura a un Mesh.

Imos explicar un pouco en profundidade o proceso.

Preparación: Deberemos copiar o arquivo seguinte ó cartafol assets do proxecto Android. Se o proxecto está xerado coa ferramenta de Libgdx xa deberedes ter este gráfico.

O proceso é moi parecido a como fixemos coa cor, pero cunhas pequenas diferenzas.

O código do Mesh será o seguinte:

```
trianguloRed = new Mesh(true, 3, 3, VertexAttribute.Position(), VertexAttribute.ColorUnpacked(),VertexAttribute.TexCoords(0));
```

Esta liña indica que imos enviar os datos da textura, cun identificador de 0 (TexCoords**(0)**).

Ese 0 vai pasarse como parámetro ó programa do vertex.vert, na variable a_texCoord**X**. Polo tanto, teremos que poñer algo como isto:

- Programa vertex.vert

```
attribute vec3 a_position;
attribute vec4 a_color;
attribute vec2 a_texCoord0;
varying vec4 v_color;
varying vec2 v_textCoord;
void main()
{
  gl_Position =  vec4(a_position,1);
  v_color = a_color;
  v_textCoord = a_texCoord0;
}
```

- Liña 3: Fixarse como as coordenadas da textura é un Vector2.
- Liña 5: Definimos unha variable de tipo varying para pasarlle o valor ó programa Fragment (fragment.frag).
- Liña 10: Gardamos o valor na variable v_textCoord.

Ata aquí o código da clase UD4_4_ProgramShader é o seguinte:

Código da clase UD4_4_ProgramShader
Obxectivo: Debuxar unha textura utilizando un ShaderProgram.

```
public class UD4_4_ProgramShader extends Game {

private Mesh trianguloRed;
private ShaderProgram shaderProgram;

private Texture textura;

@Override
public void create() {
// TODO Auto-generated method stub

shaderProgram = new ShaderProgram(Gdx.files.internal("vertex.vert"), Gdx.files.internal("fragment.frag"));
if (shaderProgram.isCompiled() == false) {
Gdx.app.log("ShaderError", shaderProgram.getLog());
System.exit(0);
}

trianguloRed = new Mesh(true, 3, 3, VertexAttribute.Position(), VertexAttribute.ColorUnpacked(),VertexAttribute.TexCoords(0));
trianguloRed.setVertices(new float[] {
-0.5f, -0.5f, 0f, 1f, 0f, 0f, 1f, 0f, 1f,
0.5f, -0.5f, 0f, 1f, 0f, 0f, 1f, 1f, 1f,
0f, 0.5f, 0f, 1f, 0f, 0f, 1f, 0.5f,0f
});
trianguloRed.setIndices(new short[] {0, 1, 2});

FileHandle imageFileHandle = Gdx.files.internal("badlogic.jpg");
textura = new Texture(imageFileHandle);
}
        ..............
}
```

Imos analizar agora a parte render desta clase.

Para asociar unha textura a un obxecto temos que chamar ó método bind da textura. Ó facelo, indicamos un número que vai indicar á qué número de textura de OPEN GL vai asociarse. Por defecto é 0 (se non podemos nada).

O código sería este:

```
textura.bind(0);
```

Agora temos que buscar a maneira de 'pasarlle' esta textura o programa Fragment, xa que é aquí onde asignamos a cor e a textura. A idea é que colla punto a punto (en texturas falamos de fragmentos) da textura e a debuxe no sitio que lle corresponda de acordo á posición indicada na figura.

Como facendo o bind xa estamos a cargar a textura en OPEN GL ES, imos a enviarlle o programa o índice utilizado, para que dende o programa Fragment accede a dita textura.

A forma de pasarlle o índice é así:

```
shaderProgram.begin();
shaderProgram.setUniformi("u_texture", 0);
trianguloRed.render(shaderProgram, GL20.GL_TRIANGLES,0,3);

shaderProgram.end();
```

- Liña 2: pasamos á variable u_texture ó valor 0 que se corresponde co índice empregado na textura con bind.

Agora modificamos o programa vertex.vert para que recolla o índice e acceda á textura:

- Programa fragment.frag

```
#ifdef GL_ES
  precision mediump float;
#endif
varying vec4 v_color;
varying vec2 v_textCoord;
uniform sampler2D u_texture;
void main()
{
  vec4 texColor = texture2D(u_texture, v_textCoord);
  gl_FragColor = v_color*texColor;
}
```

- Liña 6: O índice é utilizado para acceder á textura correspondente (sampler2D é un tipo de dato co que accedemos á textura).
- Liña 9: Accedemos a un punto da textura.
- Liña 10: Mandamos a textura a debuxar. Multiplicamos pola cor para tintar a textura. Se non queremos que a cor teña efecto sobre a textura podemos quitar v_color da multiplicación.

O código completo da clase:

Código da clase UD4_4_ProgramShader
Obxectivo: Mandar unha textura ó ShaderProgram

```
import com.badlogic.gdx.Game;
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.files.FileHandle;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.Mesh;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.VertexAttribute;
import com.badlogic.gdx.graphics.glutils.ShaderProgram;

/**
 * Funcionamento do ShaderProgram
 * @author ANGEL
 */

public class UD4_4_ProgramShader extends Game {

private Mesh trianguloRed;
private ShaderProgram shaderProgram;

private Texture textura;

@Override
public void create() {
// TODO Auto-generated method stub

shaderProgram = new ShaderProgram(Gdx.files.internal("vertex.vert"), Gdx.files.internal("fragment.frag"));
if (shaderProgram.isCompiled() == false) {
Gdx.app.log("ShaderError", shaderProgram.getLog());
System.exit(0);
}

trianguloRed = new Mesh(true, 3, 3, VertexAttribute.Position(), VertexAttribute.ColorUnpacked(),VertexAttribute.TexCoords(0));
```

```
trianguloRed.setVertices(new float[] {
-0.5f, -0.5f, 0f, 1f, 0f, 0f, 1f, 0f, 1f,
0.5f, -0.5f, 0f, 1f, 0f, 0f, 1f, 1f, 1f,
0f, 0.5f, 0f, 1f, 0f, 0f, 1f, 0.5f,0f
});
trianguloRed.setIndices(new short[] {0, 1, 2});

FileHandle imageFileHandle = Gdx.files.internal("badlogic.jpg");
textura = new Texture(imageFileHandle);
}


@Override
public void render() {

Gdx.gl20.glClearColor(0f, 0f, 0f, 1f);
Gdx.gl20.glClear(GL20.GL_COLOR_BUFFER_BIT|GL20.GL_DEPTH_BUFFER_BIT);


Gdx.gl20.glEnable(GL20.GL_DEPTH_TEST);

textura.bind(0);
shaderProgram.begin();
shaderProgram.setUniformi("u_texture", 0);
trianguloRed.render(shaderProgram, GL20.GL_TRIANGLES,0,3);

shaderProgram.end();

Gdx.gl20.glDisable(GL20.GL_DEPTH_TEST);

}

@Override
public void resize (int width,int height){
// Definimos os parámetros da cámara

}

@Override
public void dispose(){
shaderProgram.dispose();
trianguloRed.dispose();

}

}
```

1.1.3.4 Traballando coa cámara

Como último paso temos que pasarlle a matriz da cámara para que axuste cada punto da figura e os visualice no lugar correcto en función da cámara.

Lembrar que cando vimos as cámaras comentamos que para posicionar correctamente os puntos, usamos as matrices de proxección e modelado. Unha matriz servía para definir os planos far, near e viewportwidth / viewportheight (a matriz de proxección) e a outra matriz (modelado) servía para mover a cámara, rotala,...non é que servira para iso, quero dicir que cando movemos a cámara para unha posición, todos as figuras da nosa escena se teñen que debuxar de acordo á posición da cámara.

Polo tanto é necesario enviarlle as dúas matrices ó programa Vertex.vert, xa que é aquí onde posicionamos os puntos da nosa figura.

Lembrar que a matriz de proxección e modelado se combinan para dar a matriz combinada a cal temos acceso a través de método combined das cámaras.

Como pasamos dita matriz ó programa Vertex.vert ?

- Programa vertex.vert:

```
attribute vec3 a_position;
attribute vec4 a_color;
```

```
attribute vec2 a_texCoord0;
varying vec4 v_color;
varying vec2 v_textCoord;
uniform mat4 u_worldView;
void main()
{
  gl_Position =  u_worldView *vec4(a_position,1);
  v_color = a_color;
  v_textCoord = a_texCoord0;
}
```

- Liña 6: definimos a matriz de 4x4 como de tipo uniform (vaise pasar como parámetro dende a clase).
- Liña 9: multiplicamos a posición da figura Mesh pola matriz da cámara.

IMPORTANTE: Hai que ter ven claro que as figuras en 3D se debuxan todas no punto (0,0,0). A partires dese punto se moven de acordo a súa posición (o veremos no seguinte punto) e unha vez movidas se aplica a cámara para sacar unha 'foto' do que se ve. Lembrar que todos son operacións matemáticas feitas con matrices.

Código da clase UD4_4_ProgramShader
Obxectivo: Aplicar a matriz combinada dunha cámara ó Shader Program.

```
import com.badlogic.gdx.Game;
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.files.FileHandle;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.Mesh;
import com.badlogic.gdx.graphics.PerspectiveCamera;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.VertexAttribute;
import com.badlogic.gdx.graphics.glutils.ShaderProgram;

/**
 * Funcionamento do ShaderProgram
 * @author ANGEL
 */

public class UD4_4_ProgramShader extends Game {

private Mesh trianguloRed;
private ShaderProgram shaderProgram;

private Texture textura;
private PerspectiveCamera camara3d;

@Override
public void create() {
// TODO Auto-generated method stub

shaderProgram = new ShaderProgram(Gdx.files.internal("vertex.vert"), Gdx.files.internal("fragment.frag"));
if (shaderProgram.isCompiled() == false) {
Gdx.app.log("ShaderError", shaderProgram.getLog());
System.exit(0);
}

trianguloRed = new Mesh(true, 3, 3, VertexAttribute.Position(), VertexAttribute.ColorUnpacked(),VertexAttribute.TexCoords(0));
trianguloRed.setVertices(new float[] {
-0.5f, -0.5f, 0f, 1f, 0f, 0f, 1f, 0f, 1f,
0.5f, -0.5f, 0f, 1f, 0f, 0f, 1f, 1f, 1f,
0f, 0.5f, 0f, 1f, 0f, 0f, 1f, 0.5f,0f
});
trianguloRed.setIndices(new short[] {0, 1, 2});

FileHandle imageFileHandle = Gdx.files.internal("badlogic.jpg");
textura = new Texture(imageFileHandle);

camara3d = new PerspectiveCamera();
}


@Override
public void render() {
```

```
Gdx.gl20.glClearColor(0f, 0f, 0f, 1f);
Gdx.gl20.glClear(GL20.GL_COLOR_BUFFER_BIT|GL20.GL_DEPTH_BUFFER_BIT);


Gdx.gl20.glEnable(GL20.GL_DEPTH_TEST);

shaderProgram.begin();
shaderProgram.setUniformMatrix("u_worldView", camara3d.combined);

textura.bind(0);
shaderProgram.setUniformi("u_texture", 0);

trianguloRed.render(shaderProgram, GL20.GL_TRIANGLES,0,3);

shaderProgram.end();

Gdx.gl20.glDisable(GL20.GL_DEPTH_TEST);

}

@Override
public void resize (int width,int height){
// Definimos os parámetros da cámara
float aspectRatio = (float) width / (float) height;
camara3d.viewportWidth = 1f*aspectRatio;
camara3d.viewportHeight = 1f;
camara3d.far=1000f;
camara3d.near=0.1f;
camara3d.lookAt(0,0,0);
camara3d.position.set(0f,0f,3f);
camara3d.update();
}

@Override
public void dispose(){
shaderProgram.dispose();
trianguloRed.dispose();

}

}
```

- Liña 22: Definimos a cámara.
- Liña 45: Instanciamos o obxecto.
- Liñas 73-83: Actualizamos os valores da cámara. Importante chamar ó método update.
- Liña 82: Mandamos a matriz ó programa vertex.vert.

TAREFA 4.4 A FACER: Esta parte está asociada á realización dunha tarefa.

-- Ángel D. Fernández González -- (2014).