1 LIBGDX Figuras 3D Mesh

UNIDADE 4: Figuras 3D. Mesh

1.1 Sumario

1.2 Introducion

No espazo en tres dimensións (no que nos movemos) os obxectos están definidos por tres dimensións espaciais (eixes X, Y, Z). Ata o de agora estabamos a visualizar gráficos que estaban definidos nos eixes x-y de tal forma que non tiñan profundidade (eixe z).

A clase que nos permite definir os puntos que conforman as figuras no espazo tridimensional denomínase Mesh.

Pero todo non vai ser tan sinxelo como isto...

1.3 Shader Program

Resulta que 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 se divide 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.

Un exemplo do que se pode facer cós Shader: https://www.youtube.com/watch?v=JkyGTZ0992s

Por que esta explicación...?

Porque cando no seguinte punto expliquemos como renderizar unha figura sinxela coma un triángulo, teremos que engadirlle un Shader Program. Loxicamente non tedes que aprender nada del, pero si é interesante que saibades porque aparece. É obrigatorio xa que estamos a utilizar OPEN GL ES 2.0.

O Shader Program que imos utilizar non vai facer ningunha modificación, só vai copiar a posición e a cor que traian cada un dos puntos da figura a debuxar.

Máis información en:

- Información dos ShaderProgram: https://github.com/libgdx/libgdx/wiki/Shaders#a-simple-shaderprogram
- Shader: http://www.opengl.org/wiki/Shader

1.4 Definindo unha figura en 3D. A clase Mesh

Preparación: Debedes crear un novo proxecto (ou utilizar un xa feito como base, tal como xa explicamos no seguinte enlace).

Crear unha clase de nome UD4_1_Triangulos3D, que derive da clase Game e sexa chamada pola clase principal das diferentes versións (desktop, android,...).

1.4.1 Posición

O primeiro que imos facer vai ser definir un triángulo cun obxecto da clase Mesh. O Mesh vainos permitir representar unha figura xeométrica en tres dimensións.

Código da clase UD4_1_Triangulos3D
Obxectivo: Definir un triángulo en 3D.

```
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.VertexAttribute;
import com.badlogic.gdx.graphics.glutils.ShaderProgram;

/**
 * Traballos coa clase Mesh para definir un triángulo en 3D
 * @author ANGEL
 */

public class UD4_1_Triangulos3D extends Game {

private Mesh triangulo1; // Definimos un obxecto pertencente á clase Mesh
private ShaderProgram shaderProgram;

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


// Creamos o ShaderProgram ===> NON NECESARIO MIRALO
String vertexShader = "attribute vec4 a_position;       \n"
+ "varying vec4 v_color;                \n"
+ "void main()                          \n"
                            + "{                                        \n"
+ "     gl_Position = a_position;       \n"
+"}                 ;                   \n"
String fragmentShader = "#ifdef  GL_ES          \n"
+     "precision mediump float; \n"
                            + "#endif                       \n"
+ "varying vec4 v_color;\n"
+ "void main()                   \n"
                            + "{                            \n"
                            + "   gl_FragColor = vec4(0,0,0,1); \n"
+ "}         ;                   \n"

// Creamos o ShaderProgram en base ós programas definidos anteriormente
shaderProgram = new ShaderProgram(vertexShader, fragmentShader);
if (shaderProgram.isCompiled() == false) {
Gdx.app.log("ShaderError", shaderProgram.getLog());
System.exit(0);
}

triangulo1 = new Mesh(true, 3, 3, VertexAttribute.Position());
triangulo1.setVertices(new float[] { -0.5f, -0.5f, 0,
  0.5f, -0.5f, 0,
  0, 0.5f, 0 });
triangulo1.setIndices(new short[] { 0, 1, 2 });

}
```

Analicemos o constructur empregado na creación do obxecto Mesh:

```
triangulo1 = new Mesh(true, 3, 3, VertexAttribute.Position());
```

- 1º parámetro (true): indicámoslle se é de clase ou non. Este parámetro é usado internamente por motivos de 'eficiencia' e se aplica á función glBufferData de OpenGL e normalmente sempre poñeremos true excepto no caso de usar animacións no que o mesh varía en cada frame (http://stackoverflow.com/questions/5402567/whats-glbufferdata-for-inopengl-es).
- 2º parámetro: é o número de vértices que ten a figura (no noso caso 3, un triángulo).
- 3º parámetro: é o número de índices. Os índices van a indicar en que orden se deben unir as liñas entre os vértices. No noso caso son tres índices.
- 4º parámetro: indicamos que información imos a enviarlle o mesh (pode ser a posición, a cor, a textura,...). No noso caso pasámoslle a posición dos vértices que conforman o triángulo no espazo x,y,z.. Cada vértice terá tres números (as coordenadas x,y,z)

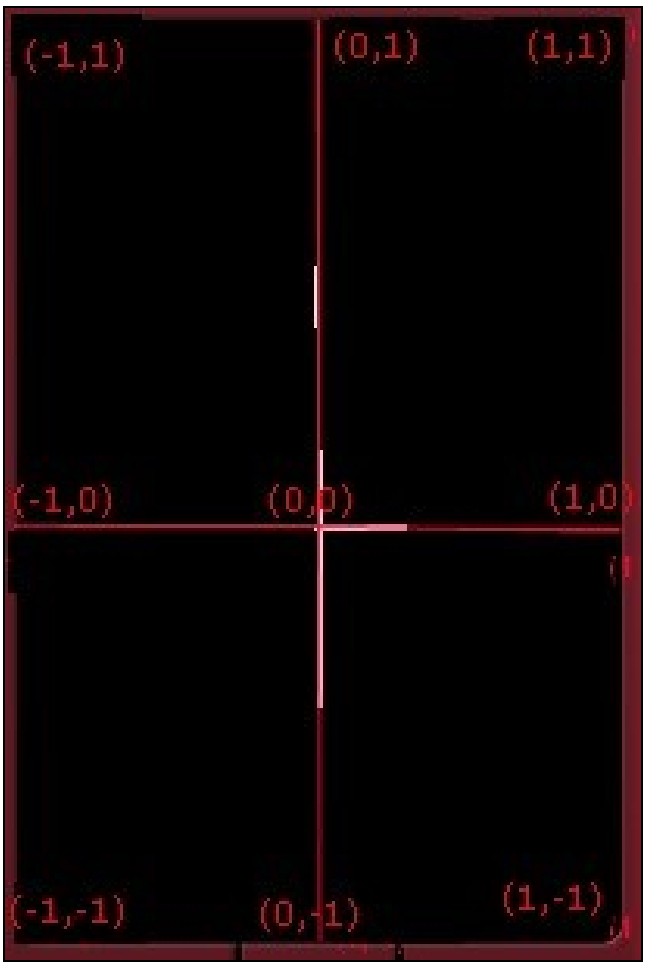
A continuación indicamos as coordenadas x,y,z do noso triángulo.

Nota: Cabe sinalar que a cámara, por defecto, se sitúa na posición (0,0,0) mirando cara a coordenada z negativa. É dicir, se a vosa cabeza é a cámara, os vosos ollos están a mirar agora mesmo cara ó monitor. Esa liña é o eixe Z que se vai afastando con valores Z negativos.

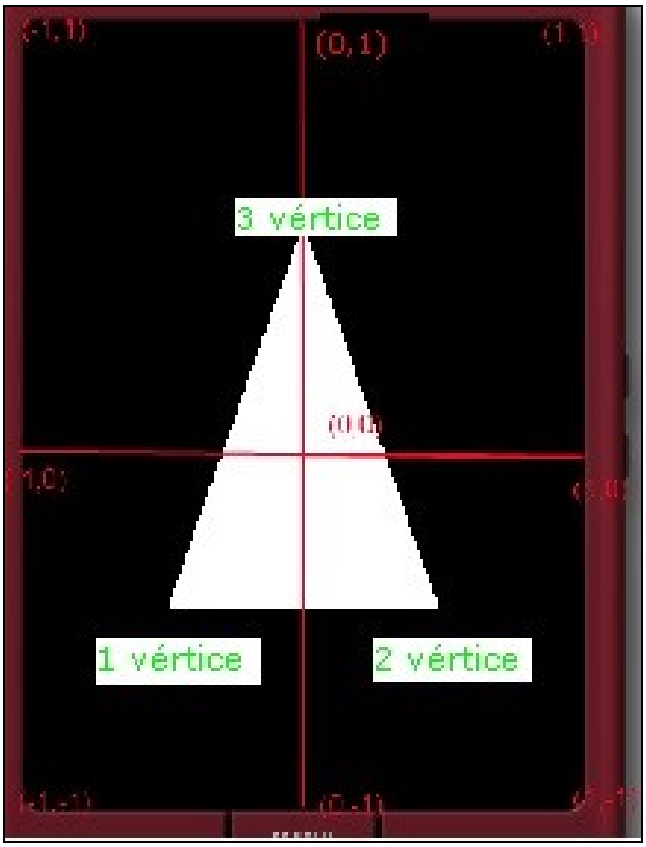
Definimos por tanto a posición dos vértices do triángulo:

```
triangulo1.setVertices(new float[] { -0.5f, -0.5f, 0,
                                      0.5f, -0.5f, 0,
                                      0, 0.5f, 0 });
```

Por defecto a cámara vai visualizar a seguinte área:

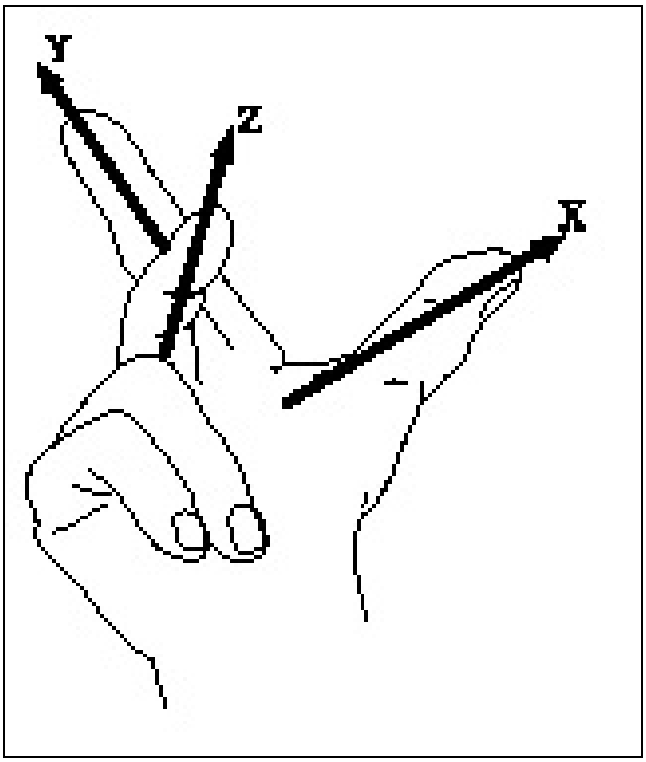


Polo tanto o triángulo estará nesta posición:



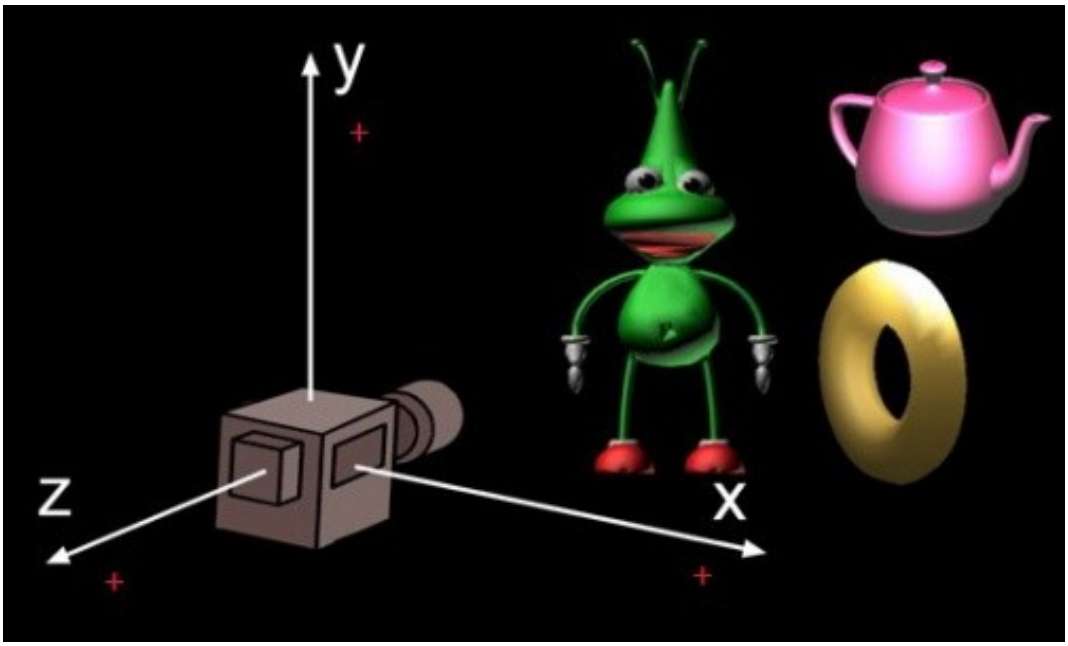
Nota: A coordenada Z sempre é 0 xa que estamos a debuxar un triángulo 'plano', non unha pirámide.

A forma en como a cámara interpreta o signo positivo ou negativo das coordenadas x,y,z é o que se coñece como regra da man dereita:



Neste gráfico os eixes indican a dirección dos valores positivos. O dedo medio apuntaría cara o teu corpo.

Cando definimos unha cámara, por defecto, mira cara ao z negativo (ven ser coma nos mirando cara o monitor, a nosa cabeza estaría situada na coordenada z=0 e o monitor nun z cun valor negativo). Por detrás da nosa cabeza o z aumenta. O X e Y serán igual ca sempre: X positivo cara a dereita e negativo cara á esquerda e Y positivo cara arriba e negativo cara abaixo.



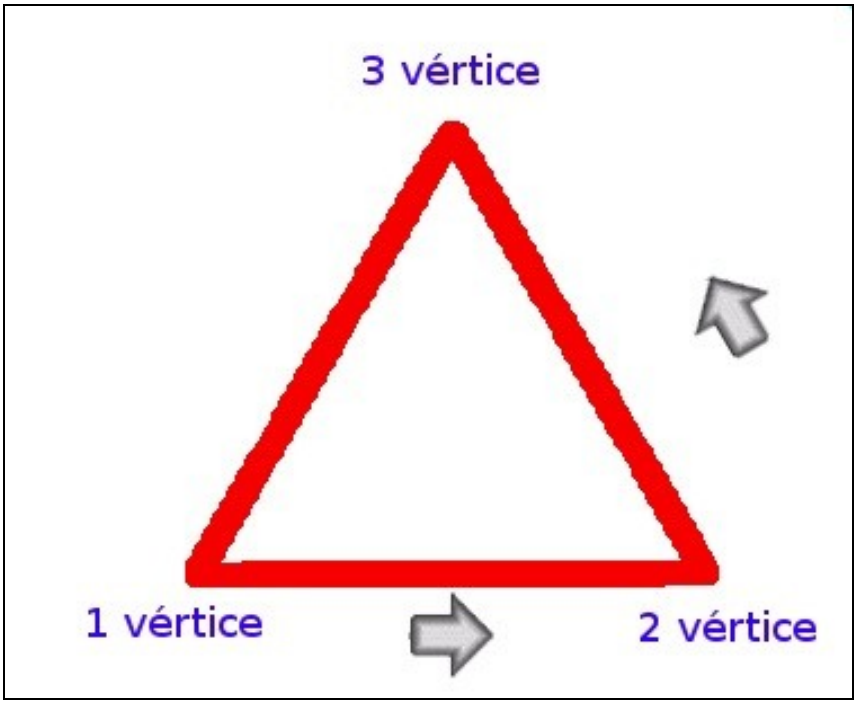
Imaxe obtida dos apuntes de Cristina Cañero Morales

Seguimos co exemplo do triángulo:

O seguinte é indicarlle a orde que ten que seguir polos vértices para debuxar a figura. Os vértices empezan a numerarse polo 0.

```
triangulo1.setIndices(new short[] { 0, 1, 2 });
```

No noso caso non importaría a orden xa que sempre debuxamos un triángulo:



Agora temos que sobreescribir o método render para debuxar o triangulo1. Para facelo imos utilizar a versión de OpenGL ES 2.0.

```
@Override
public void render() {

Gdx.gl20.glClearColor(1f, 1f, 1f, 1f);
Gdx.gl20.glClear(GL20.GL_COLOR_BUFFER_BIT);
shaderProgram.begin();
triangulo1.render(shaderProgram, GL20.GL_TRIANGLES,0,3);
```

```
shaderProgram.end();

}
```

Analicemos agora o código:

- Liña 4: Indicamos a cor que queremos que teñan por defecto cada un dos puntos que se envían á GPU no formato (RGBA). Xa visto neste enlace.
- Liña 5: Esta liña borra o buffer onde se garda a información de cada punto que se vai enviar á GPU.
- Liñas 6 e 8: Todo que se queira enviar á GPU aplicándolle o ShaderProgram definido terá que estar ente o begin - end.
- Liña 7: Indicamos que queremos renderizar o Mesh. Parámetros:

    shaderProgram: Aplica o Shader Program definido.
    GL20.GL_TRIANGLES: Imos utilizar triángulos (os cadrados se poden forman con dous

triángulos).

    Valor 0: O terceiro parámetro indica o desprazamento dentro do buffer de vértices. Normalmente poñeremos 0 (é o array onde están definidos os vértices).
    Valor 3: O carto parámetro indica o nº de índices ou vértices a utilizar.

Finalmente liberaremos a memoria.

```
@Override
public void dispose(){
shaderProgram.dispose();
triangulo1.dispose();
}
```

Código da clase UD4_1_Triangulos3D
Obxectivo: Visualizar unha figura 3D usando a clase Mesh.

```
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.VertexAttribute;
import com.badlogic.gdx.graphics.glutils.ShaderProgram;

/**
 * Traballos coa clase Mesh para definir un triángulo en 3D
 * @author ANGEL
 */

public class UD4_1_Triangulos3D extends Game {

private Mesh triangulo1; // Definimos un obxecto pertencente á clase Mesh
private ShaderProgram shaderProgram;

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

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

```
+ "void main()                                    \n"
+ "{                                              \n"
                                + "   gl_FragColor = vec4(0,0,0,1); \n"
+ "}         ;                                    \n"

// Creamos o ShaderProgram en base ós programas definidos anteriormente
shaderProgram = new ShaderProgram(vertexShader, fragmentShader);
if (shaderProgram.isCompiled() == false) {
Gdx.app.log("ShaderError", shaderProgram.getLog());
System.exit(0);
}

triangulo1 = new Mesh(true, 3, 3, VertexAttribute.Position());
triangulo1.setVertices(new float[] { -0.5f, -0.5f, 0,
  0.5f, -0.5f, 0,
          0, 0.5f, 0 });
triangulo1.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);
shaderProgram.begin();
triangulo1.render(shaderProgram, GL20.GL_TRIANGLES,0,3);
shaderProgram.end();

}

@Override
public void dispose(){
shaderProgram.dispose();
triangulo1.dispose();
}

}
```

1.4.2 Cor

Agora imos modificar a clase Mesh para indicarlle que imos a enviar información acerca da cor que van ter cada un dos vértices da figura Mesh.

Para facelo imos ter que modificar o ShaderProgram para que lle pase a cada 'fragmento' a cor definida na clase Mesh.

Código da clase UD4_1_Triangulos3D
Obxectivo: Definir unha cor en cada vértice da figura 3D.

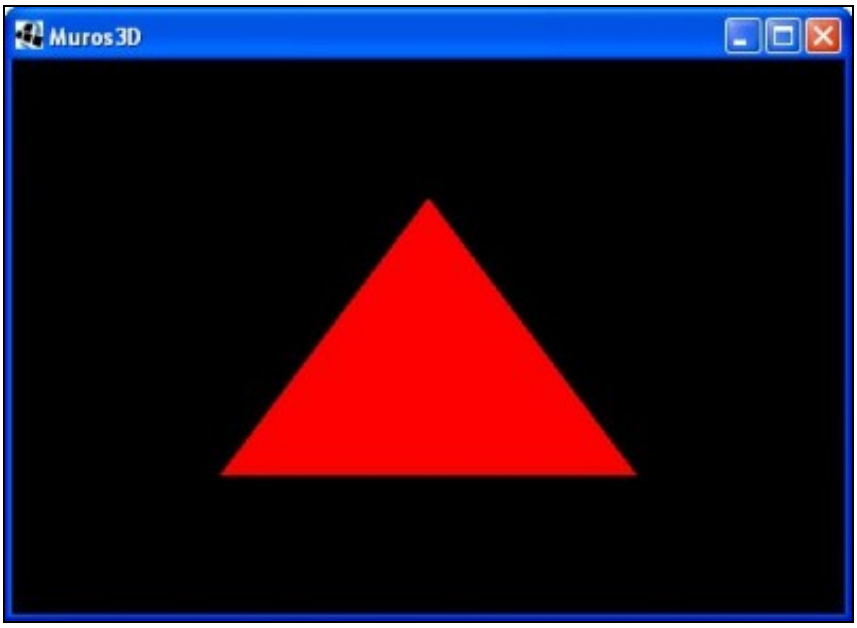
```
@Override
public void create() {
// TODO Auto-generated method stub

// create shader program
String vertexShader = "attribute vec4 a_position;       \n"
+ "attribute vec4 a_color;              \n"
+ "varying vec4 v_color;                \n"
+ "void main()                                  \n"
+ "{                                            \n"
+ "     gl_Position = a_position;       \n"
+ " v_color = a_color; \n"
+"}         ;                                                   \n"
String fragmentShader = "#ifdef GL_ES   \n"
+ "precision mediump float;             \n"
+ "#endif                                       \n"
+ "varying vec4 v_color;                \n"
+ "void main()                                  \n"
+ "{                                            \n"
+ "     gl_FragColor = v_color; \n"
+ "}         ;                                   \n"

// Creamos o ShaderProgram en base ós programas definidos anteriormente
```

```
shaderProgram = new ShaderProgram(vertexShader, fragmentShader);
if (shaderProgram.isCompiled() == false) {
Gdx.app.log("ShaderError", shaderProgram.getLog());
System.exit(0);
}

// CON COR
triangulo1 = new Mesh(false, 3, 3, VertexAttribute.Position(),
                                                    VertexAttribute.ColorUnpacked());
triangulo1.setVertices(new float[] {-0.5f, -0.5f, 0, 1, 0, 0, 1,
 0.5f, -0.5f, 0, 1, 0, 0, 1,
 0, 0.5f, 0, 1, 0, 0, 1});
triangulo1.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);
shaderProgram.begin();
triangulo1.render(shaderProgram, GL20.GL_TRIANGLES,0,3);
shaderProgram.end();

}
```

Analicemos o código:

- Liña 7: Esta variable recibe a cor que ven do Mesh (definido en cada un dos vértices).
- Liña 8: Como o dato da cor temos que pasalo ao FragmentProgram utilizamos unha variable de tipo variant.
- Liñas 12 e 20: modificamos o Shader Program para que amose a cor definida no mesh.
- Liñas 31-32: indicamos que o mesh terá como atributos a posición e a cor. Cada cor está definida por catro número no formato RGBA cun float de valor 0 a 1.
- Liñas 33-35: indicamos a posición e cor de cada vértice do triángulo. Poñemos como cor a cor vermella (1,0,0,0).
- Liña 42: Modificamos a cor de fondo polo negro para que se vexa mellor.

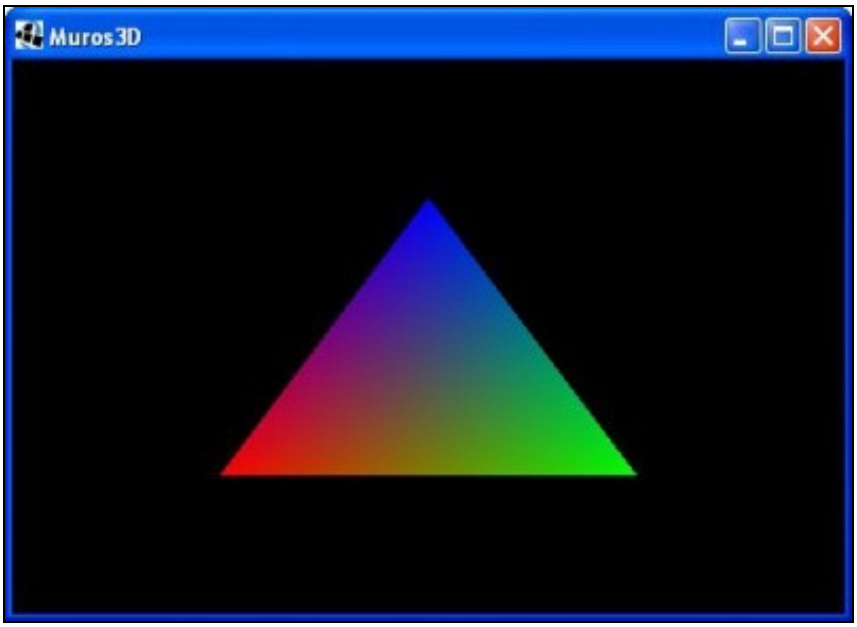
Se executades o programa teremos como resultado o seguinte:



Nota: O parámetro alfa indica o nivel de transparencia que pode ir dende 0 (transparente) a 1 (opaco).

Cada cor está asociada a cada un dos vértices. Se poñemos valores diferentes en cada vértice fai un varrido de cor dende un valor a outro.

```
triangulo1.setVertices(new float[] {-0.5f, -0.5f, 0, 1, 0, 0, 1,
 0.5f, -0.5f, 0, 0, 0, 1, 1,
 0, 0.5f, 0, 0, 1, 0, 1});
```



1.4.3 Texturas

Información sobre o manexo de texturas:

- https://github.com/mattdesl/lwjgl-basics/wiki/ShaderLesson1
- https://github.com/mattdesl/lwjgl-basics/wiki/LibGDX-Meshes-Lesson-1
- http://ogldev.atspace.co.uk/www/tutorial16/tutorial16.html

- https://github.com/mattdesl/lwjgl-basics/wiki
- https://github.com/libgdx/libgdx/blob/master/tests/gdx-tests/src/com/badlogic/gdx/tests/MeshShaderTest.java

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.

Unha textura é unha imaxe que imos 'pegar' sobre un gráfico. Xa traballamos coas texturas no xogo 2D.

Imaxinade que tedes unha parede e un cartel. O cartel é a textura e a parede a unha parte da figura en 3D (Mesh).

Normalmente as texturas teñen formas cadradas ou rectangulares. Xa veremos como 'adaptamos' esas formas ás formas dun modelo 3D...

Cando falamos de texturas, as unidades nas que se miden denomínanse **U** e **V**, que serían ó equivalente ás coordenadas X / Y. Ditas coordenadas U / V teñen uns valores que se moven entre 0 e 1, como amosamos na seguinte figura:

Se queremos que dita textura se debuxe sobre un triángulo, temos que indicar as coordenadas na que se ten que debuxar a parte da textura que queiramos:

No noso caso serían as **coordenadas da textura (0,1) ? (1,1) ? (0.5f,0)** que irán en cada vértice do triángulo.

Fixarse como parte da textura vaise perder xa que a textura ten unha forma cadrada. Xa veremos máis adiante como podemos formar un cadrado con dous triángulos e desa forma a textura vai cadrar cua figura do Mesh.

A nivel de código, teremos que facer as seguintes modificacións:

Modificamos o Shader Program:

```
String vertexShader = "attribute vec4 "+ ShaderProgram.POSITION_ATTRIBUTE +";\n"
+ "attribute vec4 " + ShaderProgram.COLOR_ATTRIBUTE +"; \n"
+ "attribute vec2 " + ShaderProgram.TEXCOORD_ATTRIBUTE + "0;\n"
+ "varying vec4 v_color;                \n"
+ "varying vec2 v_textCoord;            \n"
+ "void main()                                  \n"
+ "{                                                    \n"
+ "     gl_Position = "   + ShaderProgram.POSITION_ATTRIBUTE +";\n"
```

```
+ " v_color = " + ShaderProgram.COLOR_ATTRIBUTE +";\n"
+ " v_textCoord = " + ShaderProgram.TEXCOORD_ATTRIBUTE+ "0;      \n"
+"}         ;                                                    \n"
String fragmentShader = "#ifdef GL_ES   \n"
+ "precision mediump float;             \n"
+ "#endif                                                  \n"
+ "varying vec4 v_color;                \n"
+ "varying vec2 v_textCoord;    \n"
+ "uniform sampler2D u_texture; \n"
+ "void main()                                    \n"
+ "{                                                       \n"
+ " vec4 texColor = texture2D(u_texture, v_textCoord); \n"
+ " gl_FragColor = texColor*v_color; \n"
+ "}         ;                                             \n"
```

Non vos preocupedes se non entendedes esta parte, xa falaremos dos Shader´s no seguinte punto. Igual que no caso das cores, temos que recoller as coordenadas das texturas (que veremos a continuación) e pasalas ó fragmentShader. Dende aquí accederemos á textura a visualizar para que cada punto se debuxe con dita textura.

Agora modificamos a definición do Mesh para indicarlle que imos a enviarlle as coordenadas da textura:'

```
triangulo1 = new Mesh(false, 3, 3, VertexAttribute.Position(),
                    VertexAttribute.ColorUnpacked(),VertexAttribute.TexCoords(0));
```

E agora pasamos os datos do Mesh:

```
triangulo1.setVertices(new float[] {-0.5f, -0.5f, 0, 1, 0, 0, 1, 0, 1,
 0.5f, -0.5f, 0, 0, 0, 1, 1, 1, 1,
 0, 0.5f, 0, 0, 1, 0, 1, 0.5f,0});
```

Lembrar que os datos da textura son os dous últimos. Estamos a indicar onde debería ir cada vértice do triángulo na textura.

- VÉRTICE 1: X=> -0.5f,Y=>-0.5f,Z=>0,RED=>1,GREEN=>0,BLUE=>0,ALFA=>1,**TEXTURA_U=>0,TEXTURA_V=>1**
- VÉRTICE 2: X=> 0.5f,Y=>-0.5f,Z=>0,RED=>0,GREEN=>0,BLUE=>1,ALFA=>1,**TEXTURA_U=>1,TEXTURA_V=>1**
- VÉRTICE 3: X=> 0,Y=>0.5f,Z=>0,RED=>0,GREEN=>1,BLUE=>0,ALFA=>1,**TEXTURA_U=>0.5f,TEXTURA_V=>0**

Agora temos que cargar a textura. Isto xa o fixemos no xogo 2D. Poderíamos utilizar a clase AssetManager. Para cargar a textura creamos unha propiedade Texture:

```
private Texture textura;
```

No método create cargamos a textura dende o arquivo e a asociamos ó obxecto:

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

Agora temos que debuxar a textura. Para facelo temos que habilitar o uso de texturas e 'vincular' a textura ó Mesh:

```
@Override
public void render() {

Gdx.gl20.glClearColor(0f, 0f, 0f, 1f);
Gdx.gl20.glClear(GL20.GL_COLOR_BUFFER_BIT);
Gdx.gl20.glEnable(GL20.GL_TEXTURE_2D);
textura.bind(0);
shaderProgram.begin();
shaderProgram.setUniformi("u_texture", 0);
triangulo1.render(shaderProgram, GL20.GL_TRIANGLES,0,3);
shaderProgram.end();

}
```

Imos analizar un pouco o código:

- Liña 6: habilitamos o uso de texturas.
- Liña 7: en OPEN GL temos un conxunto de textura a utilizar. O que fai o bind é asociar a textura que cargamos a unha das texturas dispoñibles en OPEN GL. Concretamente á número 0. A cero é a textura por defecto polo que cando chamamos ó método bind poderíamos non enviarlle ningún numero.
- Liña 9: isto xa o veremos cando expliquemos o uso de Shader´s. Con esta liña estamos a 'pasar' a textura 0 (que está cargada na liña 7) ó fragmentShader para que cada punto poida coller o punto da textura que lle corresponda.

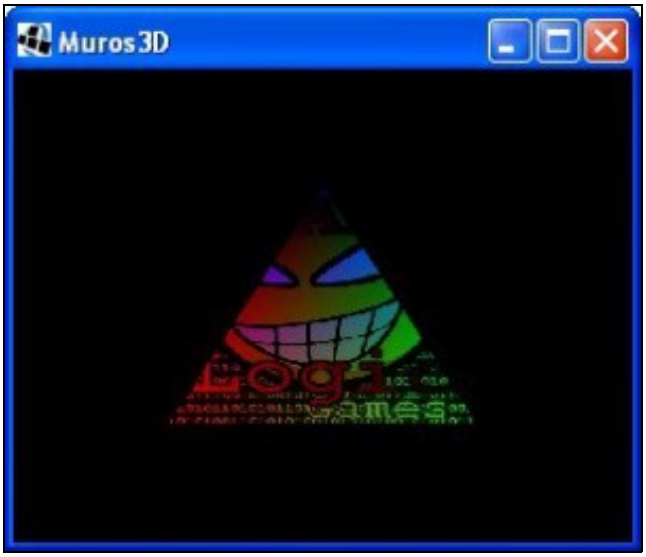
O código completo:

Código da clase UD4_1_Triangulos3D
Obxectivo: Vincular unha textura a un Mesh.

```
package com.plategaxogo3d.exemplos;

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;

/**
 * Traballos coa clase Mesh para definir un triángulo en 3D
 * @author ANGEL
 */

public class UD4_1_Triangulos3D extends Game {

private Mesh triangulo1; // Definimos un obxecto pertencente á clase Mesh
private ShaderProgram shaderProgram;
private Texture textura;
private Texture prueba;

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

// create shader program
String vertexShader = "attribute vec4 "+ ShaderProgram.POSITION_ATTRIBUTE +";\n"
+ "attribute vec4 " + ShaderProgram.COLOR_ATTRIBUTE +"; \n"
+ "attribute vec2 " + ShaderProgram.TEXCOORD_ATTRIBUTE + "0;\n"
+ "varying vec4 v_color;                \n"
+ "varying vec2 v_textCoord;            \n"
+ "void main()                                  \n"
+ "{                                                    \n"
+ "     gl_Position = "   + ShaderProgram.POSITION_ATTRIBUTE +";\n"
+ " v_color = " + ShaderProgram.COLOR_ATTRIBUTE +";\n"
+ " v_textCoord = " + ShaderProgram.TEXCOORD_ATTRIBUTE+ "0;     \n"
+"}         ;                                                   \n"
String fragmentShader = "#ifdef GL_ES   \n"
+ "precision mediump float;             \n"
+ "#endif                                               \n"
+ "varying vec4 v_color;                \n"
+ "varying vec2 v_textCoord;    \n"
+ "uniform sampler2D u_texture; \n"
+ "void main()                                  \n"
+ "{                                                    \n"
+ " vec4 texColor = texture2D(u_texture, v_textCoord); \n"
+ " gl_FragColor = texColor*v_color; \n"
+ "}         ;                                          \n"

// Creamos o ShaderProgram en base ós programas definidos anteriormente
shaderProgram = new ShaderProgram(vertexShader, fragmentShader);
if (shaderProgram.isCompiled() == false) {
Gdx.app.log("ShaderError", shaderProgram.getLog());
System.exit(0);
}

// CON COR
```

```
triangulo1 = new Mesh(false, 3, 3, VertexAttribute.Position(), VertexAttribute.ColorUnpacked(),VertexAttribute.TexCoords(0));
triangulo1.setVertices(new float[] {-0.5f, -0.5f, 0, 1, 0, 0, 1, 0, 1,
 0.5f, -0.5f, 0, 0, 0, 1, 1, 1, 1,
 0, 0.5f, 0, 0, 1, 0, 1, 0.5f,0});
triangulo1.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);
Gdx.gl20.glEnable(GL20.GL_TEXTURE_2D);

textura.bind();
shaderProgram.begin();
shaderProgram.setUniformi("u_texture", 0);
triangulo1.render(shaderProgram, GL20.GL_TRIANGLES,0,3);
shaderProgram.end();
}

@Override
public void dispose(){
shaderProgram.dispose();
triangulo1.dispose();
}

}
```

Ó executar o código dará como resultado o seguinte:



Fixarse como a cor se aplica sobre a textura para darlle diferentes tonalidades. Isto é así porque no fragmentShader temos a seguinte liña: gl_FragColor = texColor*v_color; Se queremos que a cor non modifique a textura teríamos que poñer: gl_FragColor = texColor;

TAREFA 4.1 A FACER: Esta parte está asociada á realización dunha tarefa. En caso de decidir facer o xogo proposto non será necesario facela.

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