1 LIBGDX Camara3D

UNIDADE 4: A cámara en 3D.

1.1 Sumario

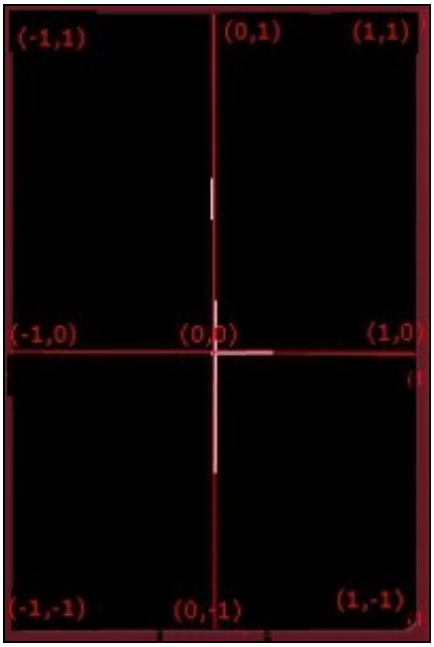
1.1.1 OrthographicCamera

Clase: http://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/OrthographicCamera.html

Información na wiki: https://github.com/libgdx/libgdx/wiki/Orthographic-camera

Agora mesmo estamos a ver unha Proxección Ortográfica. Neste tipo de proxección, non existe unha perspectiva dos obxectos que se visualizan na pantalla.

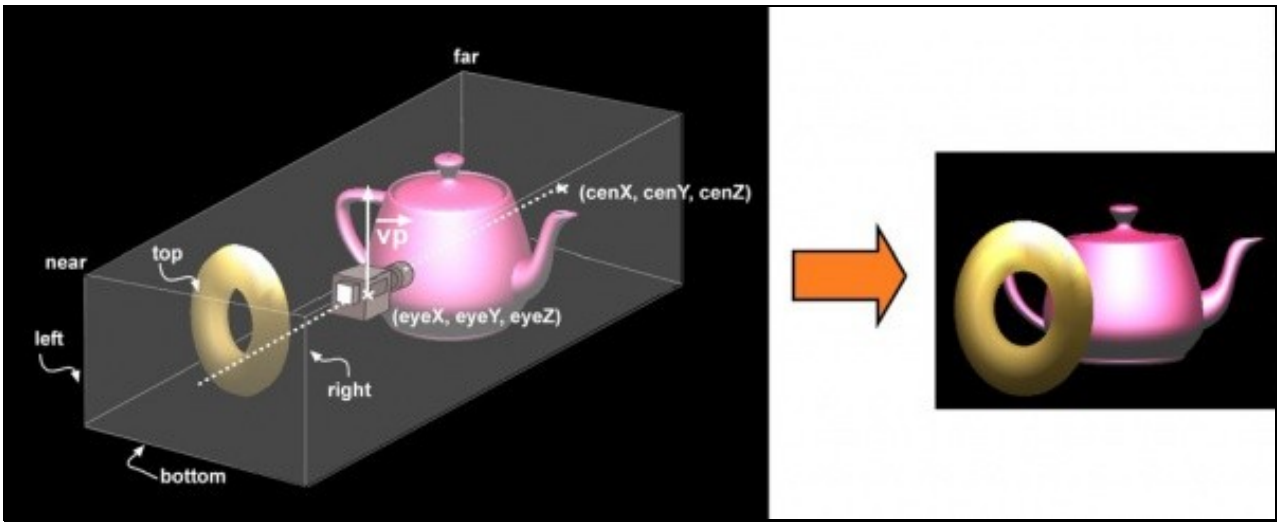
Dita cámara por defecto visualiza o seguinte espazo:



Sendo a coordenada Z a que sairía da pantalla, indo dende +1 (mirando cara nós, saíndo da pantalla) ata -1 (cara dentro da pantalla).

Podemos comprobar como se posicionamos o noso triángulo na coordenada Z = -1.1f xa non se visualiza (comprobádeo).

Do anterior podemos concluír que a cámara visualiza un volume:



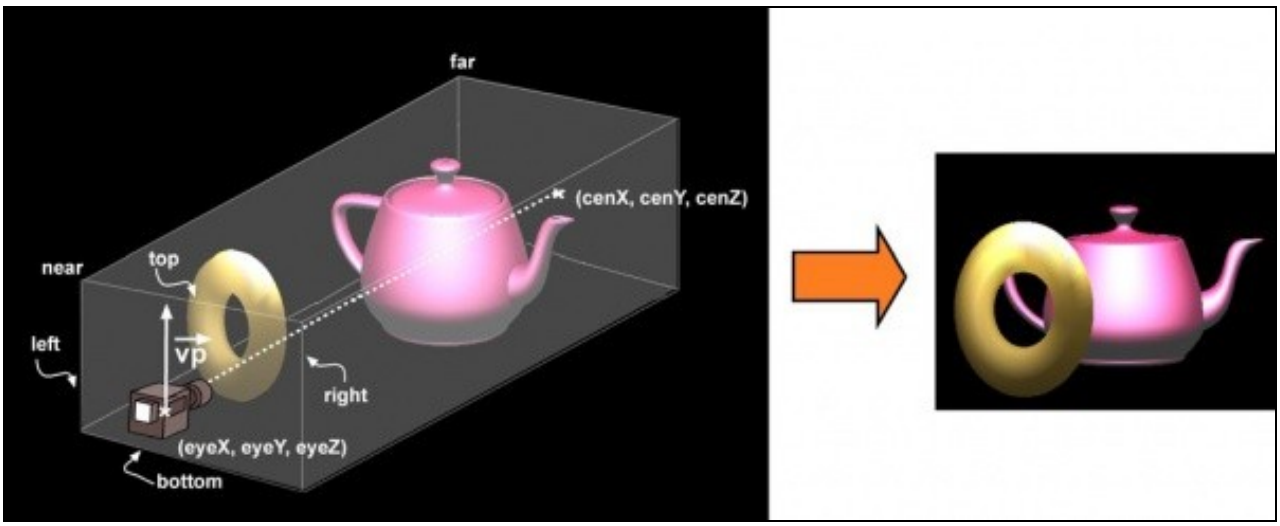
Imaxe obtida dos apuntes de Cristina Cañero Morales

Fixarse onde está a cámara e o resultado da proxección. Ó ter unha cámara ortográfica (en 2D) todo o que visualiza a cámara (estea diante ou detrás) se vai plasmar sobre un plano, levando os punto de forma paralela.

Para determinar o tamaño do volume do que se ve, se ten que indicar o plano NEAR (preto) e o plano FAR (afastado).

O plano NEAR se determina por catro coordenadas (left, top, right, bottom) e despois indicamos dúas distancias (near / far). O tamaño do plano far é igual o tamaño do plano near.

Fixarse como neste tipo de perspectiva podemos ter un near que quede por detrás da cámara (como é o exemplo anterior). O que fai dita perspectiva será coller todo o que hai dentro deste volume e visualizalo de forma 'plana'.



Imaxe obtida dos apuntes de Cristina Cañero Morales

Nota: A todo o volume que se visualiza e o que se coñece coma **View Frustrum**.

Neste caso se visualiza o mesmo que no caso anterior xa que segue 'collendo' os dous obxectos. Imos crear dous triángulos (de cores vermello e verde) sen texturas e imos colocalos nas coordenadas Z -3 e -5 respectivamente, situando o triángulo verde na coordenada X dende a posición 0 ata a posición 1f e deixando o triángulo vermello na posición do exercicio anterior (-0,5 a +0,5 en X).

Preparación do exercicio:

Crear uhna nova clase de nome UD4_2_Camara2D que derive da clase Game e cambiade os diferentes proxectos para que executen dita clase.

Código da clase UD4_2_Camara2D
Obxectivo: Visualizar dous triángulos.

```
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.PerspectiveCamera;
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_2_Camara2D extends Game {

private Mesh trianguloRed,trianguloGreen;
private ShaderProgram shaderProgram;

private PerspectiveCamera camara3d;
private OrthographicCamera camara2d;



@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"
+ "uniform mat4 u_worldView;    \n"
+ "varying vec4 v_color;                \n"
+ "void main()                                   \n"
+ "{                                                     \n"
+ "     gl_Position = u_worldView *"   + ShaderProgram.POSITION_ATTRIBUTE +";\n"
+ " v_color = " + ShaderProgram.COLOR_ATTRIBUTE +";\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);
}

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

trianguloGreen = new Mesh(true, 3, 3, VertexAttribute.Position(), VertexAttribute.ColorUnpacked());
trianguloGreen.setVertices(new float[] { // É o verde, se ve detrás
0f, -0.5f, -5f, 0f,1f,0f,1f,
1.0f, -0.5f, -5f, 0f,1f,0f,1f,
0.5f, 0.5f, -5f, 0f,1f,0f,1f });
trianguloGreen.setIndices(new short[] {0, 1, 2});

}
```

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

}

}
```

Agora chega o momento de indicarlle a cámara cal é o seu volume de visualización. Isto o podemos facer cun obxecto cámara ou directamente no método render. Pero antes disto temos que falar doutro concepto que existe en OPEN GL ES que son as matrices. Todas as modificacións que se fan sobre os obxectos que se van visualizar (cambio de perspectiva, movemento da cámara, cambiar o tamaño do que se visualiza, rotar ou trasladar,...) se fai mediante unha serie de operacións matemáticas con matrices. Ditas matrices sofren modificacións e se aplican a cada un dos vértices que conforman a figura xeométrica que queremos debuxar.

Os tres tipos de matrices que existen son: matriz de proxección (GL_PROJECTION), matriz de modelado (GL_MODELVIEW) e matriz de textura (GL_TEXTURE). No caso de querer cambiar o tamaño de visualización o temos que facer sobre a matriz de proxección, mentres que o debuxado (render) dos obxectos o temos que facer na matriz de modelado.

Imos definir unha cámara ortográfica:

```
private OrthographicCamera camara2d;

......
camara2d = new OrthographicCamera(2f,2f);
camara2d.near=0.1f;
camara2d.far=100f;
camara2d.update();
```

Como vemos estamos a definir o seu ViewFrustum o área de visualización. A cámara vai ver 100 unidades de distancia cara a adiante (plano near e far) e o seu plano near (igual en tamaño ó plano far) terá un tamaño de 2x2 unidades.

IMPORTANTE: É necesario chamar ó método update cando se fagan modificacións sobre a cámara.

Cando chamamos ó método update estamos a actualizar a matriz de proxección da cámara. Podedes acceder a dita matriz escribindo: camara2d.projection

Ó mesmo tempo, a cámara se posiciona nun lugar por defecto. Este lugar é a metade do viewportwidth e viewportheight da definición. Así, no noso caso, a cámara estaría situada nas coordenadas:

X = 2f / 2 = 1f;
Y = 2f /2 = 1f;
Z = 0;

Cando movemos a cámara estamos a modificar a súa matriz de modelado. Ó chamar o método update esta matriz se actualiza. Podedes acceder a dita matriz escribindo: camara2d.view

Na cámara temos unha matriz que é a combinación das dúas matrices anteriores, e que se denomina **combined**. É esta matriz a que temos que aplicar a cada un dos puntos dos obxectos do noso mundo para que se rendericen adecuadamente.

Na versión OPEN GL ES 2.0 temos que pasarlle dita matriz como parámetro ó Shader Program para que a aplique a cada punto.

Seguindo o noso exemplo:

```
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;
```

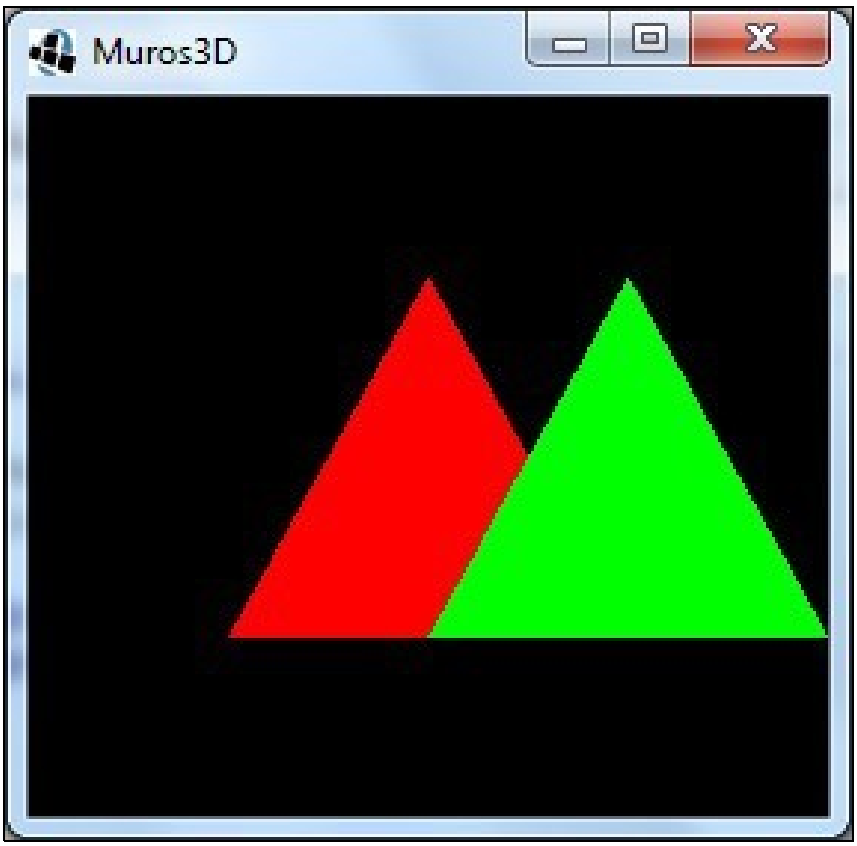
```
/**
 * Uso da cámara orthographic e perspective
 * @author ANGEL
 */

public class UD4_2_Camara2D extends Game {

private Mesh trianguloRed,trianguloGreen;
private ShaderProgram shaderProgram;

private OrthographicCamera camara2d;

@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"
+ "uniform mat4 u_worldView;    \n"
+ "varying vec4 v_color;                \n"
+ "void main()                                   \n"
+ "{                                                  \n"
+ "     gl_Position = u_worldView *"   + ShaderProgram.POSITION_ATTRIBUTE +";\n"
+ " v_color = " + ShaderProgram.COLOR_ATTRIBUTE +";\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);
}

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

trianguloGreen = new Mesh(true, 3, 3, VertexAttribute.Position(), VertexAttribute.ColorUnpacked());
trianguloGreen.setVertices(new float[] { // É o verde, se ve detrás
0f, -0.5f, -5f, 0f,1f,0f,1f,
1.0f, -0.5f, -5f, 0f,1f,0f,1f,
0.5f, 0.5f, -5f, 0f,1f,0f,1f });
trianguloGreen.setIndices(new short[] {0, 1, 2});

// Definimos os parámetros da cámara
camara2d = new OrthographicCamera(2f,2f);
camara2d.near=0.1f;
camara2d.far=100f;
camara2d.update();
}




@Override
public void render() {

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

```
shaderProgram.begin();
shaderProgram.setUniformMatrix("u_worldView", camara2d.combined);
trianguloRed.render(shaderProgram, GL20.GL_TRIANGLES,0,3);
trianguloGreen.render(shaderProgram, GL20.GL_TRIANGLES,0,3);

shaderProgram.end();

}

@Override
public void resize (int width,int height){
}

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

}

}
```

Analicemos o código:

- Liñas 19 e 67-71: Definimos a cámara ortográfica, o seu viewport e os planos far e near.
- Liña 84: Pasamos ó Shader Program a matriz combinada (Model - View) da cámara para que a aplique a cada punto do obxecto Mesh.
- Liñas 85-86: Debuxamos os dous triángulos.

Se executamos este código dará como resultado isto:



TAREFA 4.2 A FACER: Esta parte está asociada á realización dunha tarefa.

Se queremos podemos modificar a posición da cámara, polo que os triángulos parecen que se moven, pero será a cámara quen o faga.

Se posicionamos a cámara nun principio e non se move, teríamos que facelo no constructor, ou onde teñamos o código que define os planos far e near da cámara.

No noso caso imos facelo no render, xa que imos facer que se mova a cámara ata que os triángulos deixen de estar dentro do viewfrustrum da cámara.

Código da clase UD4_2_Camara2D
Obxectivo: Mover a cámara.

```
private float tempo=0;
        ..............

@Override
public void render() {

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

tempo += Gdx.graphics.getDeltaTime();
camara2d.position.sub(0,0,tempo*0.2f);
camara2d.update();

shaderProgram.begin();
shaderProgram.setUniformMatrix("u_worldView", camara2d.combined);

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

shaderProgram.end();

}
```

Se executades o código veredes que ó cabo dun rato os triángulos desaparecen. Sabedes por que non van afastándose pouco a pouco da cámara ?

Porque estamos utilizando unha cámara 2D e nesta cámara non hai perspectiva. Os obxectos se 'aplastan' contra a parede indepedentemente da distancia.

1.1.2 Algoritmo Z-Buffer

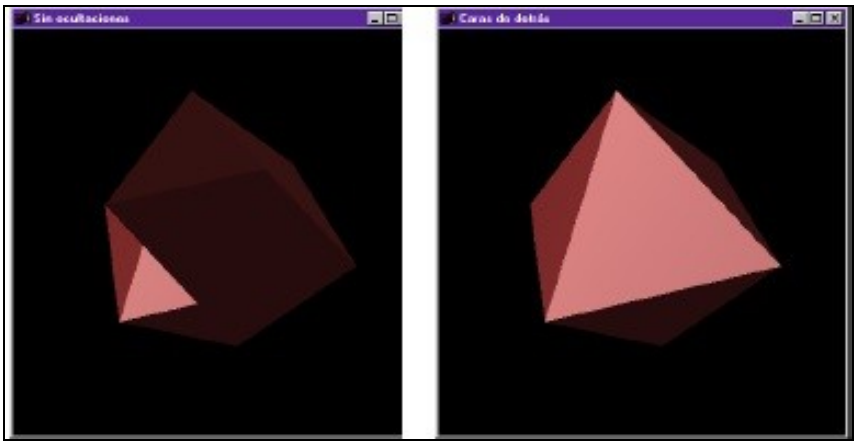
Información na wiki: http://es.wikipedia.org/wiki/Z-Buffer

Se vos fixades no exercicio anterior o triángulo verde (coordenada z=-5) está máis afastado que o triángulo vermello (coordenada z=-3). Sen embargo se debuxa o verde por enriba.

Isto é debido a que agora mesmo OPEN GL non ten en conta a coordenada Z para determinar que obxecto se debuxa antes ou despois.

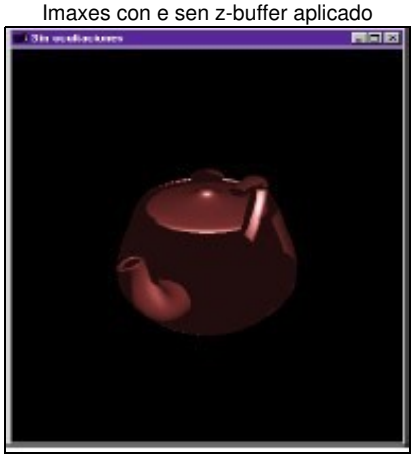
Para solucionalo en OPELGL temos dúas opcións:

- Algoritmo das caras de atrás: consiste en ocultar as caras que non se debuxarían porque formarían parte da parte traseira do obxecto:



- Algoritmo do Z-Buffer: consiste en ter gardada nunha matriz a posición Z de cada punto de tal forma que cando se debuxa cada punto se comproba o Z do novo punto e se verifica que non exista outro punto no mesmo sitio cun Z máis grande (iría diante).

- Imaxes con e sen z-buffer aplicado
-



Sen Z-Buffer.

-

Con Z-Buffer aplicado.

Nos imos utilizar dita técnica. Para isto temos que:

- Dicirlle que cando borre a pantalla tamén borre os datos do Z-buffer:

```
public void render(){
   Gdx.gl20.glClear(GL20.GL_COLOR_BUFFER_BIT|GL20.GL_DEPTH_BUFFER_BIT);
   ..............
```

- Antes de debuxar debemos habilitar dito modo:

```
Gdx.gl20.glEnable(GL20.GL_DEPTH_TEST);
```

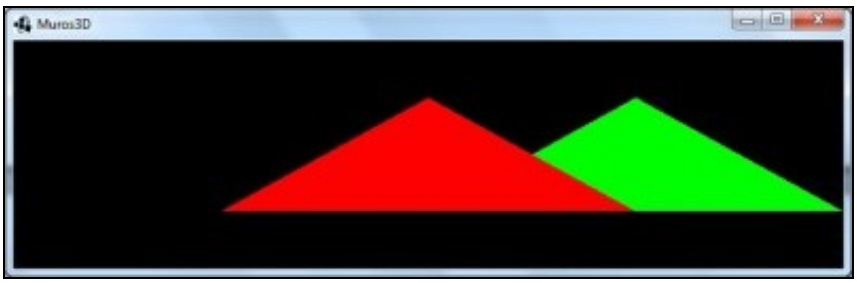
- Debuxamos e deshabilitamos o modo:

```
shaderProgram.begin();
shaderProgram.setUniformMatrix("u_worldView", camara2d.combined);

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

shaderProgram.end();
Gdx.gl20.glDisable(GL20.GL_DEPTH_TEST);
```

1.1.3 Relación de aspecto

Sobre este punto xa falamos no desenvolvemento dos xogo 2D.

Fixarse que pasa se alongamos a pantalla (sería equivalente a executar a nosa aplicación nun dispositivo móbil cunha resolución maior en ancho):



Como vemos os triángulos se alongan. Isto é debido a que o ViewPort que definimos sempre ten 2f de ancho, polo que se aumenta a resolución os triángulos se teñen que adaptar á nova medida. Para evitalo, unha posible solución sería sobreescribir o método resize para que aumente o tamaño do width do viewport en función do ancho da pantalla (relación de aspecto):

Nota: quitamos todo o código do constructor da clase referido á cámara.

public void create() { ..... ~~camara2d = new OrthographicCamera(2f,2f); camara2d.near=0.1f; camara2d.far=100f; camara2d.update();~~

No método resize:

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

float aspectRatio = (float) width / (float) height;
camara2d = new OrthographicCamera();
camara2d.viewportWidth = 2f*aspectRatio;
camara2d.viewportHeight = 2f;
camara2d.near=0.1f;
camara2d.far=10f;
camara2d.update();


}
```

O resultado agora é este:

O que estamos a facer é ampliar o width da cámara para que manteña a proporción de aspecto có height.

Dependendo do xogo esta pode ser unha opción ou non, xa que os usuarios que tiveran una resolución maior poderían ver máis partes do xogo (ven máis ancho).

Veremos máis adiante que na cámara 3D, non importa o tamaño do width-height, no senso que o máximo que imos ter cando o obxecto se achegue á cámara será un valor baixo, de 1f x 1f ou 1fx2f. **Só imos establecer a relación de aspecto**.

1.1.4 Transparencia

A transparencia ven indicado polo valor alfa nos datos que enviamos de cor en cada un dos vértices.

Nota: Lembrade que un valor 1 é opaco e un valor 0 é transparente.

Para que funcione a transparencia temos que facer o seguinte:

- Todos os obxectos con transparencias teñen que estar renderizados dende o máis distante á cámara ata o máis preto, nesa orde.
- Todos os obxectos opacos teñen que estar renderizados primeiro e non importa a orde.

No noso caso temos que renderizar primeiro a triángulo verde (é o máis distante), despois o vermello.

Ademais temos que activar o Blending (transparencia) en Open GL dunha forma moi parecido a como activamos o Z-Buffer:

```
Gdx.gl20.glEnable(GL20.GL_BLEND);     //ACTIVAMOS
Gdx.gl20.glBlendFunc(GL20.GL_SRC_ALPHA , GL20.GL_ONE_MINUS_SRC_ALPHA) ;

Gdx.gl20.glDisable(GL20.GL_BLEND) ;  // DESACTIVAMOS
```

Modificamos o exercicio UD4_2_Camara2D para engadir a transparencia. Fixarse como seguindo as normas primeiro imos debuxar o triángulo verde (que ten alfa = 1) e despois o vermello que ten as transparencias (alfa=0.5f)

Código da clase UD4_2_Camara2D
Obxectivo: Engadir transparencias ó triángulo vermello.

```
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;

/**
 * Uso da cámara orthographic e perspective
 * @author ANGEL
 */

public class UD4_2_Camara2D extends Game {

private Mesh trianguloRed,trianguloGreen;
private ShaderProgram shaderProgram;

private OrthographicCamera camara2d;

@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"
+ "uniform mat4 u_worldView;    \n"
+ "varying vec4 v_color;                \n"
+ "void main()                                   \n"
+ "{                                                 \n"
+ "     gl_Position = u_worldView *"   + ShaderProgram.POSITION_ATTRIBUTE +";\n"
+ " v_color = " + ShaderProgram.COLOR_ATTRIBUTE +";\n"
```

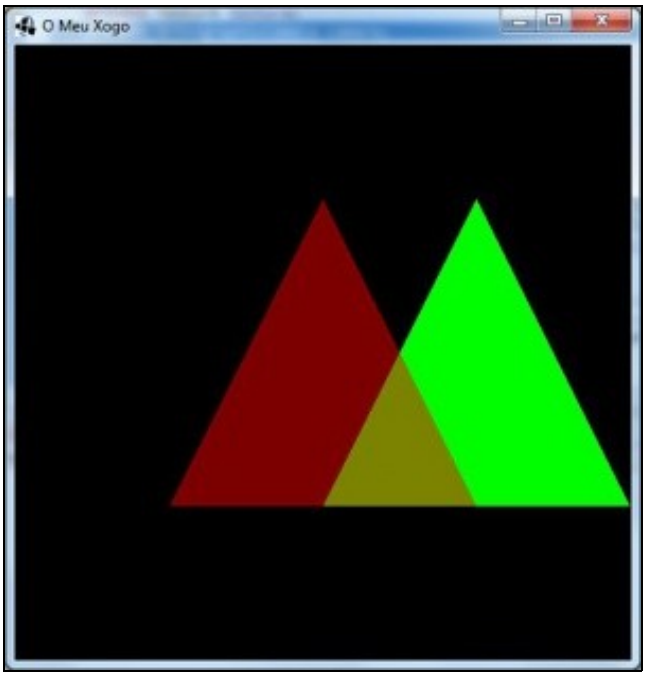
```
+"}         ;                                                       \n"
String fragmentShader = "#ifdef GL_ES   \n"
+ "precision mediump float;             \n"
+ "#endif                                                   \n"
+ "varying vec4 v_color;                \n"
+ "void main()                                      \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);
}

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

trianguloGreen = new Mesh(true, 3, 3, VertexAttribute.Position(), VertexAttribute.ColorUnpacked());
trianguloGreen.setVertices(new float[] { // É o verde, se ve detrás
0f, -0.5f, -5f, 0f,1f,0f,1f,
1.0f, -0.5f, -5f, 0f,1f,0f,1f,
0.5f, 0.5f, -5f, 0f,1f,0f,1f });
trianguloGreen.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);
Gdx.gl20.glEnable(GL20.GL_BLEND);
Gdx.gl20.glBlendFunc(GL20.GL_SRC_ALPHA , GL20.GL_ONE_MINUS_SRC_ALPHA) ;


shaderProgram.begin();
shaderProgram.setUniformMatrix("u_worldView", camara2d.combined);

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

shaderProgram.end();
Gdx.gl20.glDisable(GL20.GL_BLEND) ;

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;
camara2d = new OrthographicCamera();
camara2d.viewportWidth = 2f*aspectRatio;
camara2d.viewportHeight = 2f;
camara2d.near=0.1f;
camara2d.far=10f;
camara2d.update();
```

```
        }

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

        }

        }
```

Comentemos o código:

- Liña 52: Cambiamos o valor alfa da cor en cada vértice do triángulo vermello a 0.5f.
- Liñas 78-79: Habilitamos as transparencias.
- Liñas 78-79: Renderizamos primeiro os opacos e despois os transparentes.
- Liña 89: Deshabilitamos as transparencias.

O resultado:

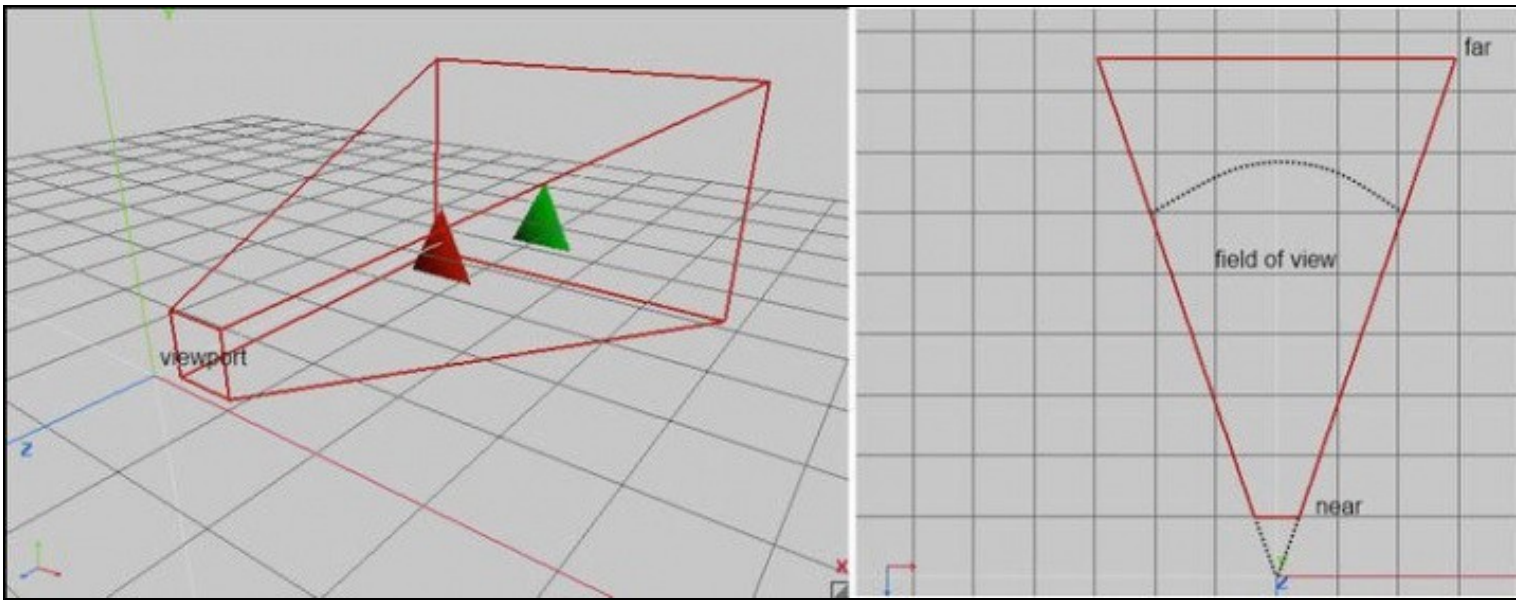


1.1.5 PerspectiveCamera

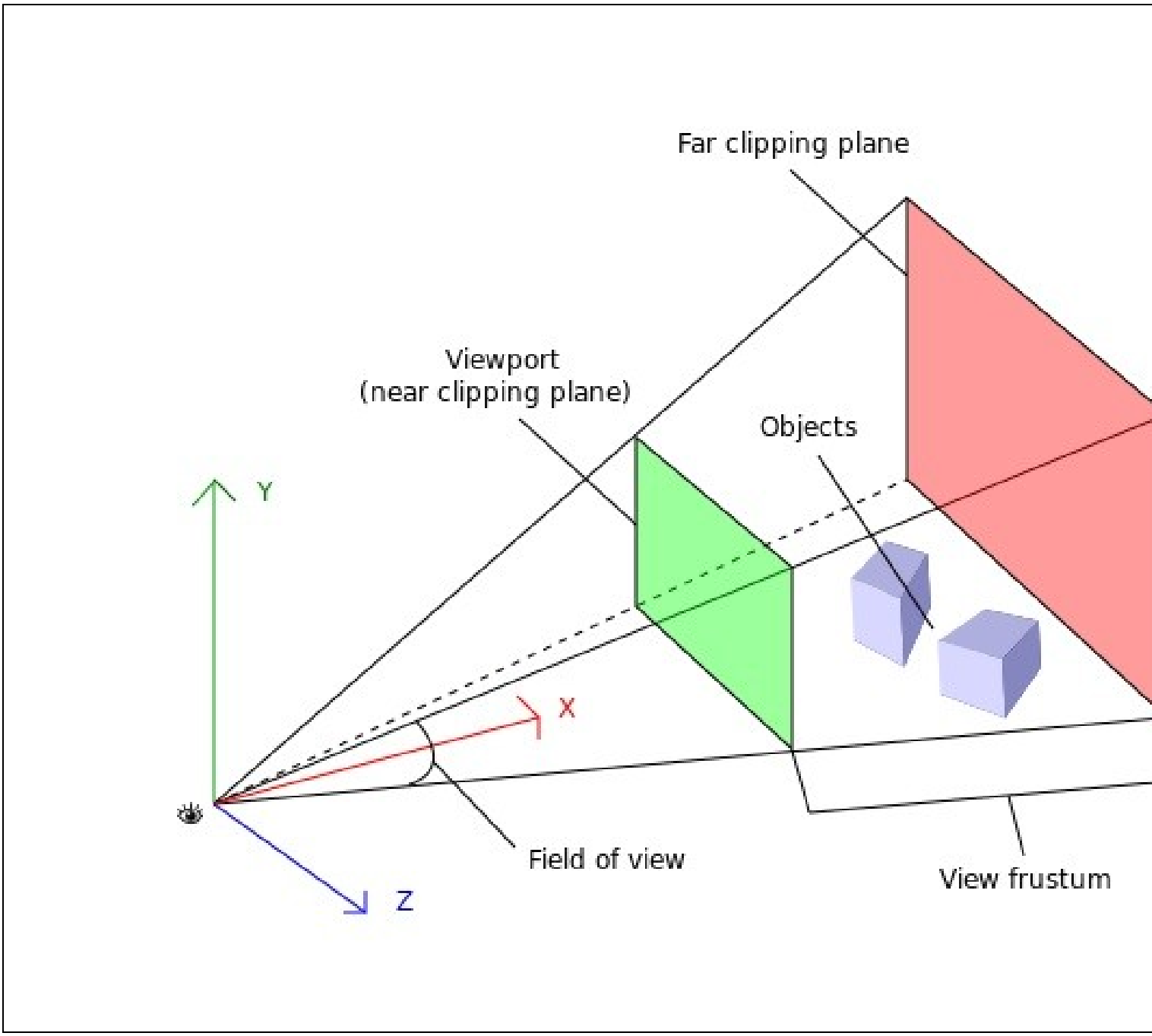
Clase: PerspectiveCamera.

Información: http://www.badlogicgames.com/wordpress/?p=1550

Ata o de agora estamos utilizando unha visión ortogonal. Agora imos utilizar unha visión en perspectiva polo que os obxectos distantes veranse máis pequenos que os pretos.



Os conceptos son os mesmos aprendidos para a cámara orthogonal. Temos un ViewFrustrum que ven a ser o volume que se pode visualizar, definido polo plano near e far. A diferenza é que agora cando vexamos a imaxe en pantalla teremos unha 'perspectiva' (por exemplo, os obxectos máis pretos veranse máis grandes).



Fixarse como agora os raios non van paralelos, se non que parten dun ollo (a cámara).

Agora o plano far e plano near non teñen o mesmo tamaño, xa que este último vaise agrandando a medida que nos afastamos da cámara.

Preparación do exercicio:

Crear unha nova clase de nome UD4_3_Camara3D que derive da clase Game e cambiade os diferentes proxectos para que executen dita clase.

Código da clase UD4_3_Camara3D
Obxectivo: Visualizar dous triángulos utilizando unha cámara en perspectiva.

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

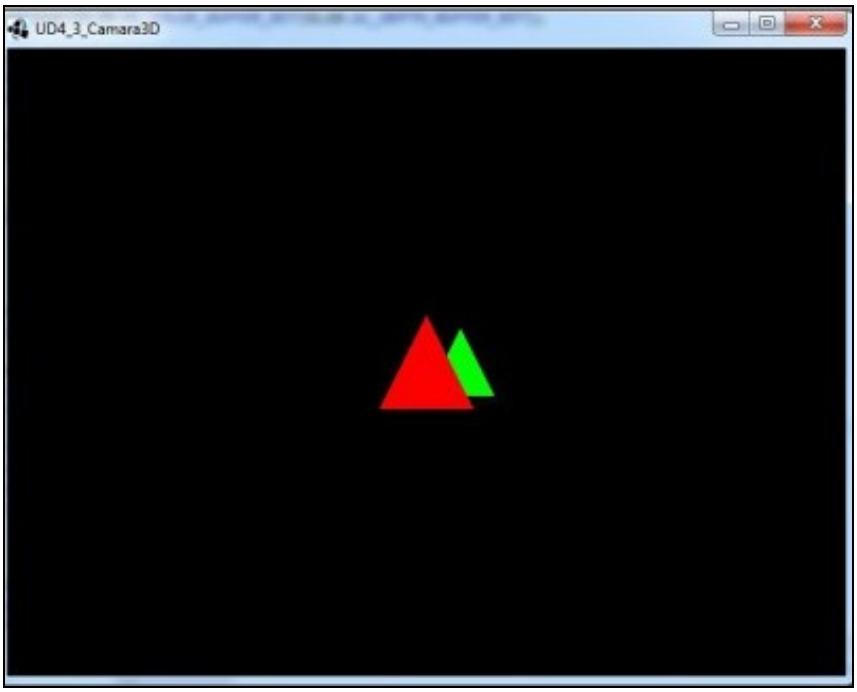
```
import com.badlogic.gdx.math.Matrix4;

/**
 * Uso da cámara orthographic e perspective
 * @author ANGEL
 */

public class UD4_3_Camara3D extends Game {

private Mesh trianguloRed,trianguloGreen;
private ShaderProgram shaderProgram;

private PerspectiveCamera camara3d;

@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"
+ "uniform mat4 u_worldView;    \n"
+ "varying vec4 v_color;                \n"
+ "void main()                                   \n"
+ "{                                                     \n"
+ "     gl_Position = u_worldView *"   + ShaderProgram.POSITION_ATTRIBUTE +";\n"
+ " v_color = " + ShaderProgram.COLOR_ATTRIBUTE +";\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);
}

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

trianguloGreen = new Mesh(true, 3, 3, VertexAttribute.Position(), VertexAttribute.ColorUnpacked());
trianguloGreen.setVertices(new float[] { // É o verde, se ve detrás
0f, -0.5f, -5f, 0f,1f,0f,1f,
1.0f, -0.5f, -5f, 0f,1f,0f,1f,
0.5f, 0.5f, -5f, 0f,1f,0f,1f });
trianguloGreen.setIndices(new short[] {0, 1, 2});

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);
```

```
trianguloRed.render(shaderProgram, GL20.GL_TRIANGLES,0,3);
trianguloGreen.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 = 2f*aspectRatio;
camara3d.viewportHeight = 2f;

camara3d.far=100f;
camara3d.near=0.1f;
camara3d.position.set(0,0,2f);
camara3d.update();
}

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

}

}
```

Analicemos o código:

- Liña 21: Definimos o obxecto camara3d.
- Liña 68: Instanciamos a cámara 3D. O facemos aquí para optimizar o código, xa que poderíamos levalo ó método resize, pero dito método se chama dúas veces ó iniciar ó xogo e se hai un cambio na resolución volve a chamarse. Como neste momento non sabemos a resolución usamos o constructor sen parámetros.
- Liña 80: Pasamos á matriz combinada ó Shader Program.
- Liñas 93-100: Definimos o plano far, near e o viewportwidth - viewportheight. Tamén posicionamos a cámara para que mire cara ós triángulos.

O resultado:



Como vemos os triángulos se visualizan con perspectiva....

Un dos atributos que podemos modificar na cámara 3D é o que se coñece como Field of view (FOV).

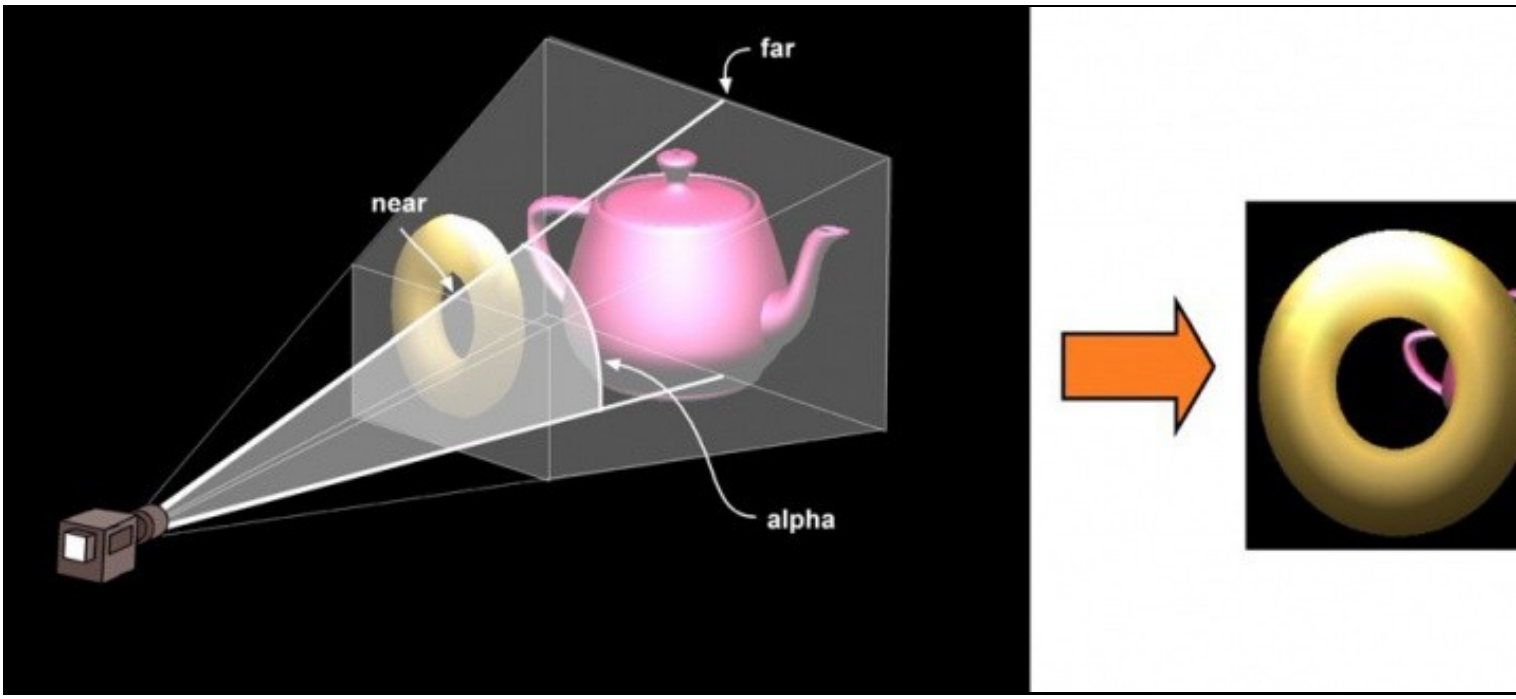
Este dato o podemos dar directamente no constructor da forma:

```
camara3d = new PerspectiveCamera(fov,viewportWidth,viewportHeight)
```

Ou accedendo á propiedade da forma:

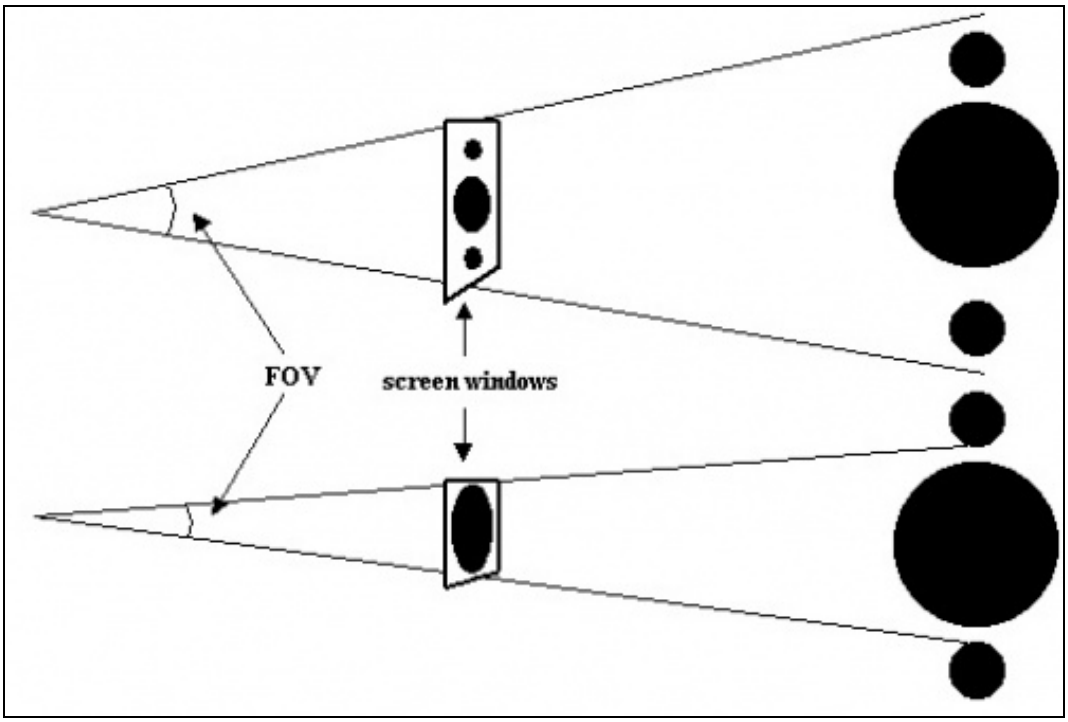
```
camara3d.fieldOfView=valor
```

Por defecto establece un ángulo alfa de 67 grados.



Imaxe obtida dos apuntes de Cristina Cañero Morales

O ángulo alfa se coñece coma Field Of View (FOV) e representa o ángulo de visión da cámara sobre a proxección dos obxectos. Queda máis claro no seguinte gráfico:



Como podemos observar, o que se visualizaría na pantalla (screen window) variará en función deste ángulo, podendo producir un certo efecto de 'zoom' dependendo do ángulo utilizado.

MOI IMPORTANTE: Na cámara en perspectiva ten que cumplirse que 0 < near < far.

NOTA: O viewport width/height só serve para establecer a relación de aspecto. Lembrar que estamos traballando en 3D e polo tanto non limitamos o ancho e alto do que vemos.

Por exemplo, se temos un cubo situado na posición (0,0,0) de tamaño 1f. O tamaño da pantalla onde se vai visualizar ten unha relación de aspecto de 1 (por exemplo 100x100 ou 200x200,...).

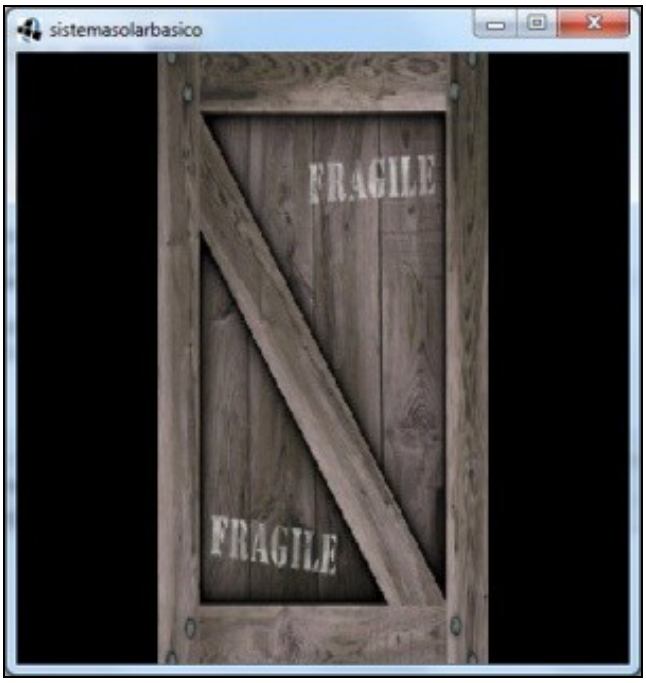
```
cfg.width = 400;
cfg.height = 400;
```

Situamos a cámara na posición (0f,0f,1.2f). Definimos a cámara có viewport width e height a 1:

```
camara = new PerspectiveCamera(67, 1f , 1f);
```

Teríamos este resultado:

Se agora cambiamos o viewport width e poñemos dous:



O cubo se deforma xa que coa cámara estamos a dicirlle que pode debuxar, para cubrir todo o ancho da pantalla, un obxecto con dúas unidades de ancho (2f) e o cubo ten unha polo que para adaptalo a ese ancho 'o deforma'. Se queremos que o cubo sempre se vexa ben teremos que adaptar o ancho da pantalla en función da relación de aspecto (ancho e alto da resolución de pantalla).

```
float aspectRatio = (float) width / (float) height;
camara = new PerspectiveCamera(67, 1f*aspectRatio , 1f);
```

Se o deixamos así o ancho se adapta e o cubo sempre vaise debuxar ben:

Normalmente ponse:

```
camara = new PerspectiveCamera(67, width*aspectRatio ,height);
```

Aínda que o efecto é o mesmo (lembrar que non estamos definindo o tamaño do que se ve, como no caso da cámara ortográfica, se non a relación de aspecto). **ESTAMOS EN 3D.**

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