http://libgdx.badlogicgames.com/

Programación de xogos 2D/3D
Framework LIBGDX

PARTE 3D

Lugar: CE.FO.RE. FERROL
Docente: ANGEL DANIEL FERNÁNDEZ GONZÁLEZ

DO 2 Ó 6 DE SETEMBRO DO 2013

Manual de desenvolvemento de xogos 3D co motor de xogos LIBGDX

Imos desenvolver un xogo en 3D.
Para facelo, primeiro imos ver cales son as diferencias entre a cámara Ortográfica e a cámara de Perspectiva.

Tamén imos explorar como facer figuras xeométricas en 3D mediante o uso dunha nova clase denominada Mesh.

Un pouco de historia.

A librería gráfica que imos utilizar para desenvolver xogos 3D é a OPEN GL ES. Denomínase ES xa que é a usado en dispositivos móbiles coma teléfonos, tablets,....
Ten diferenzas con respecto a open gl usada nas aplicacións de computadores, pero os conceptos que imos aprender son os mesmos.

Podedes consultar as diferenzas entre OPEN GL e OPEN GL ES no seguinte enlace:

http://es.wikipedia.org/wiki/OpenGL_ES

Agora mesmo temos tres versións de OPEN GL: 1.0, 1.1, 2.0 e 3.0.

Nos imos desenvolver xogos baseados na versión 1.0/1.1, compatible con todos os dispositivos.

Empezando:

Creamos por tanto un novo proxecto en branco e engadimos unha clase de nome: Triangulos3D.java que derive da clase Game e sexa chamada pola clase principal das diferentes versións (desktop, android,...).

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.

O código será o seguinte:

```
private Mesh triangulo1;  // Definomos unha propiedade pertencente á clase Mesh

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

                triangulo1 = new Mesh(true, 3, 3,
                new VertexAttribute(Usage.Position, 3, "a_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, new VertexAttribute(Usage.Position, 3, "a_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-in-opengl-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), por iso indicamos o número 3 (**new** VertexAttribute(Usage.*Position*, **3,.....**))

  Creamos por tanto unha instancia do obxecto VertexAttribute onde indicamos que información vai levar cada vértice. Sempre ten que levar coma mínimo a posición x,y,z de cada vértice por iso poñemos Usage.Position.

  ```
  new VertexAttribute(Usage.Position, 3, "a_position")
  ```

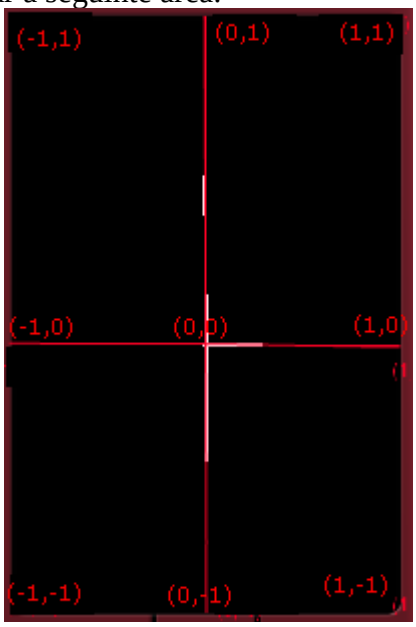
  O valor 'a_position' é un alias utilizado por Open Gl 2.0 para asociar cada vértice a unha sombra (non usado por nos).

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

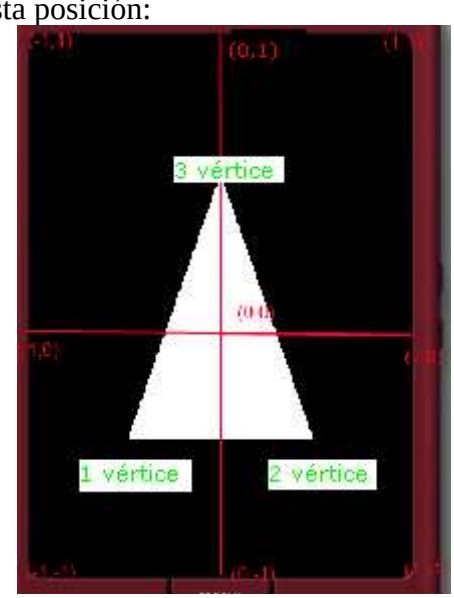
Nota: cabe sinalar que a cámara, por defecto, se sitúa na posición (0,0,0) mirando cara a coordenada z negativa:

```
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:

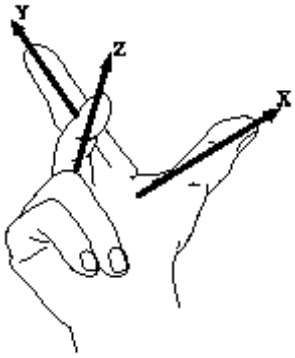


Polo tanto o triángulo estará nesta posición:

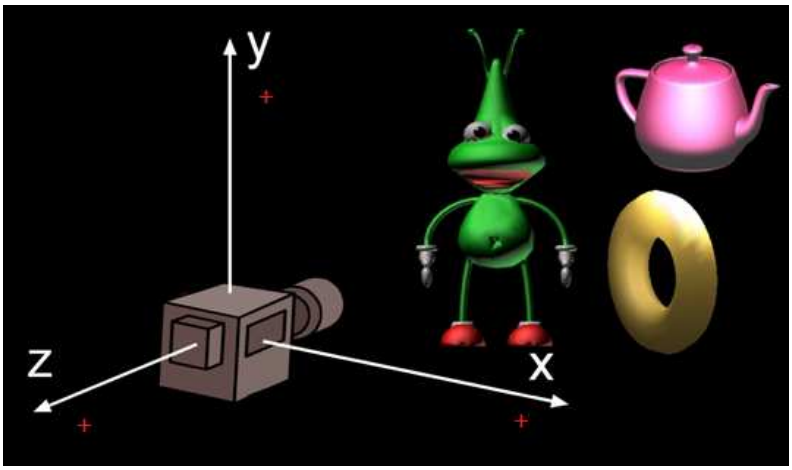


Nota: A coordenada Z sempre é 0 xa que estamos a debuxar un triángulo 'plano', non unha pirámide.

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.



A cámara 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.

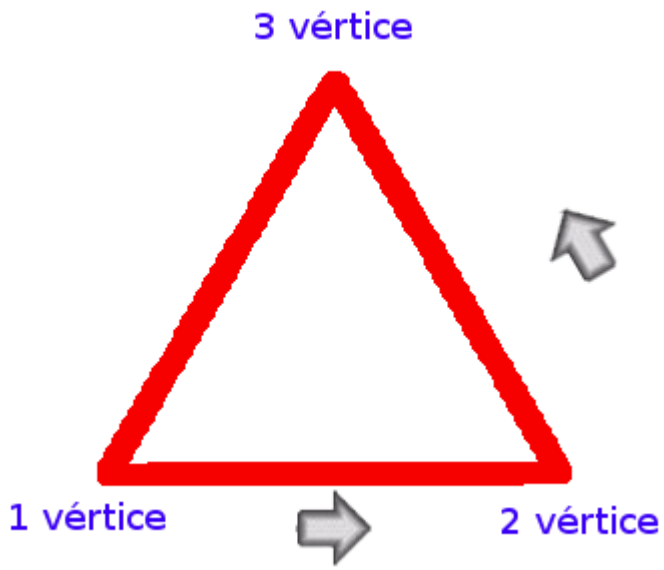


Seguimos co exemplo do triángulo:

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 1.0 normalmente suficiente para a maioría dos xogos 2D / 3D e sabemos que é compatible con todos os dispositivos.

Nota: máis información en http://www.khronos.org/opengles/

```
@Override
public void render() {

        Gdx.gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        triangulo1.render(GL10.GL_TRIANGLES, 0, 3);
}
```

A primeira liña indica que borre a pantalla. A cor por defecto é o negro, pero podemos modificala coa orden Gdx.gl.glClearColor.

```
Gdx.gl.glClearColor(1, 0, 0, 0); Exemplo para cambiar a cor de fondo
                  (red,green,blue,alfa): Parámetros do método glClearColor
```

A segunda liña

```
triangulo1.render(GL10.GL_TRIANGLES, 0, 3);
```

debuxa o triángulo, indicando que:

- Imos utilizar triángulos (os cadrados se poden forman con dous triángulos).
- O segundo parámetro indica o desprazamento dentro do buffer de vértices. Normalmente poñeremos 0 (é o array onde están definidos os vértices).
- O terceiro parámetro indica o nº de índices ou vértices a utilizar.

Seguimos. Agora imos modificar o triángulo para que se debuxe cunha cor.
Para isto temos que modificar a chamada ó constructor do Mesh para engadir a cada vértice a

información de cor que queiramos que teña:

```
triangulo1 = new Mesh(true, 3, 3,
                new VertexAttribute(Usage.Position, 3, "a_position"),
                new VertexAttribute(Usage.ColorPacked, 4, "a_color"));
```

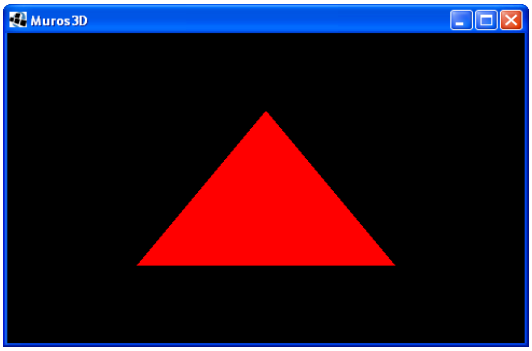
Como cada cor vai ter catro datos (Red,Green,Blue,Alfa), por iso indicamos un 4.

Nota: lembrar que alfa e o nivel de transparencia.

Agora temos que indicar a cor no mesmo sitio onde vai a información de posicionamento:

```
triangulo1.setVertices(new float[] { -0.5f, -0.5f, 0,Color.toFloatBits(255, 0, 0, 255),
              0.5f, -0.5f, 0,Color.toFloatBits(255, 0, 0, 255),
              0, 0.5f, 0 ,Color.toFloatBits(255, 0, 0, 255)}
              );
```

Como lle indicamos que na compoñente Red ten valor 255, estamos a pintar un triángulo vermello. O alfa polo de agora non fai nada, o veremos máis adiante.

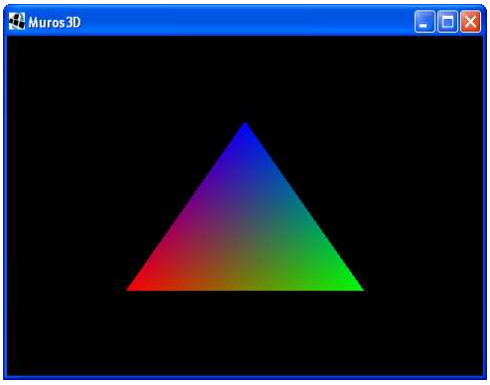


Nota: O parámetro alfa indica o nivel de transparencia que pode ir dende 0 (transparente) a 1 (opaco). No caso anterior estamos utilizando unha función que nos permite utilizar números entre 0 e 255, que será equivalente a usar valores dende 0 a 1.

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.

Por exemplo:

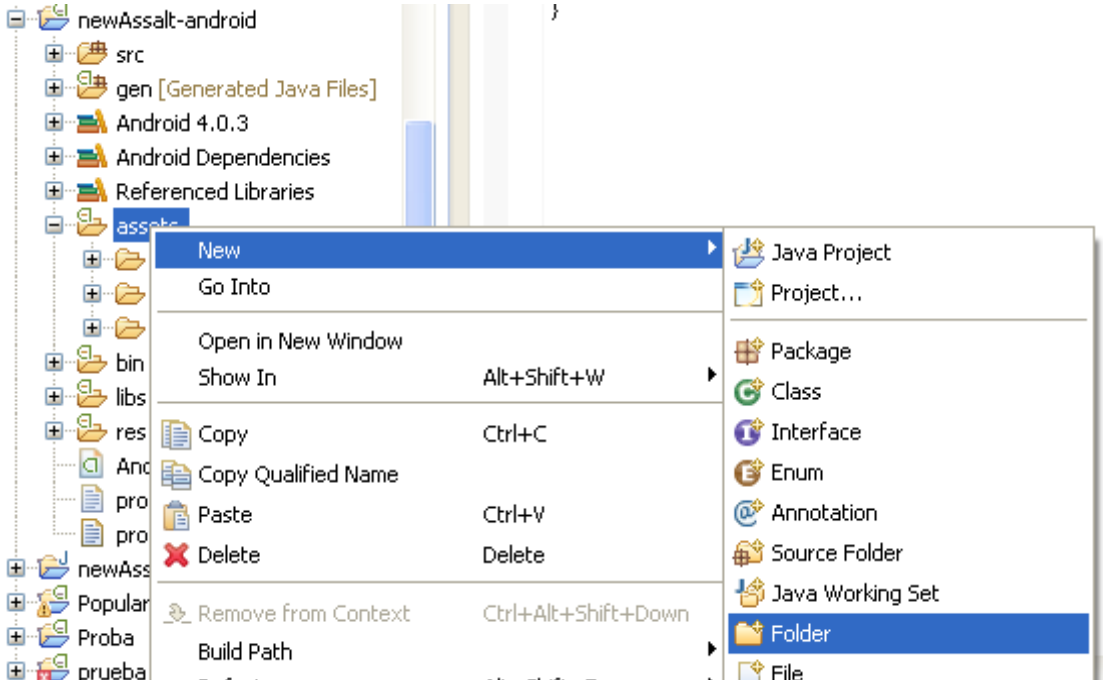
```
triangulo1.setVertices(new float[] { -0.5f, -0.5f, 0,Color.toFloatBits(255, 0, 0, 255),
              0.5f, -0.5f, 0,Color.toFloatBits(0, 255, 0, 255),
              0, 0.5f, 0 ,Color.toFloatBits(0, 0, 255, 255)}
);
```



Engadindo unha textura.

Para nos, a textura ven ser un debuxo (imaxe) que vai 'pegada' a unha figura xeométrica (no noso caso un triángulo).

Imos 'pegar' unha imaxe ó triángulo.
Creamos un cartafol na versión Android de nome /texturas dentro do cartafol assets:



Copiamos o arquivo de textura dende o DVD (/Proxectos Eclipse/Proxecto 3D planetas/0.-Explicación inicial (Triángulo-Cubo)/badlogic.jpg) ó cartafol assets/texturas/ da versión de Android.

Igual que fixemos antes coas cores, temos que informar ó obxecto Mesh que imos utilizar unha textura.

```
triangulo1 = new Mesh(true, 3, 3,
                new VertexAttribute(Usage.Position, 3, "a_position"),
                new VertexAttribute(Usage.ColorPacked, 4, "a_color"),
                new VertexAttribute(Usage.TextureCoordinates, 2, "a_texCoords"));
```

Con isto imos dicirlle que por cada vértice imos utilizar dúas coordenadas para indicar que parte da textura vai nese vértice. Temos que pensar que a textura vai debuxarse sobre unha superficie 2D (no noso caso un triángulo).

A textura sempre se debuxa sobre unha superficie de coordenadas entre 0-1. As coordenadas, cando falamos de texturas se denominan (u,v) que serían o equivalente a coordenadas (x,y).

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 **<u>coordenadas da textura</u>** (0,1) – (1,1) – (0,5,0) que irán en cada vértice do triángulo.

Para cargar a textura creamos unha propiedade Texture:

```
private Texture textura;
```

No método create cargamos a textura dende o arquivo e a asociamos á propiedade:

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

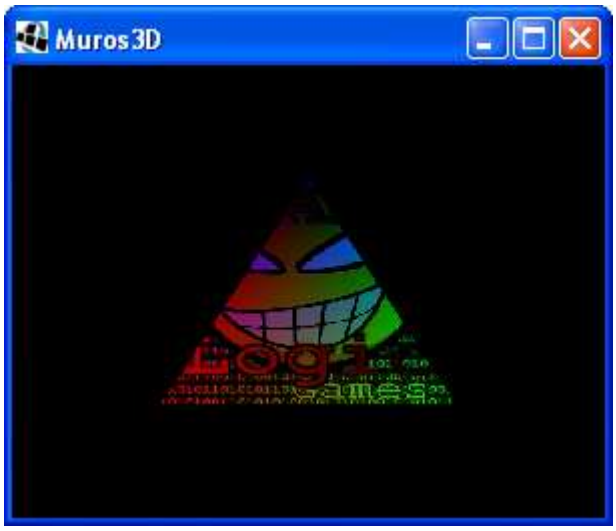
Agora temos que asinar a textura a cada vértice do triángulo:

```
triangulo1.setVertices(new float[] { -0.5f, -0.5f, 0,Color.toFloatBits(255, 0, 0, 255),0,1,
              0.5f, -0.5f, 0,Color.toFloatBits(0, 255, 0, 255),1,1,
              0, 0.5f, 0 ,Color.toFloatBits(0, 0, 255, 255),0.5f,0}
              );
```

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

```
public void render() {
       // Gdx.gl.glClearColor(1, 0, 0, 0); Exemplo para cambiar a cor de fondo
       Gdx.gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
       Gdx.gl.glEnable(GL10.GL_TEXTURE_2D);
       textura.bind();
       triangulo1.render(GL10.GL_TRIANGLES, 0, 3);
}
```

Despois de facer o bind, calquera cousa que se debuxe vai ter asociado a textura.



Fixarse como parte da textura se perde, xa que a textura é un cadrado e estamos a debuxar un triángulo.

Resume:

- Vimos como debuxar unha figura xeométrica usando a clase Mesh.
- Asociamos unha cor e unha textura a dita figura.

Polo de agora imos deixar o tema dos mesh e imos comprender como funciona a cámara en 3D (PerspectiveCamera).

Exercicio:

Facer un cadrado de tamaño 2x2 e asinarlle a textura anterior.

Solución:

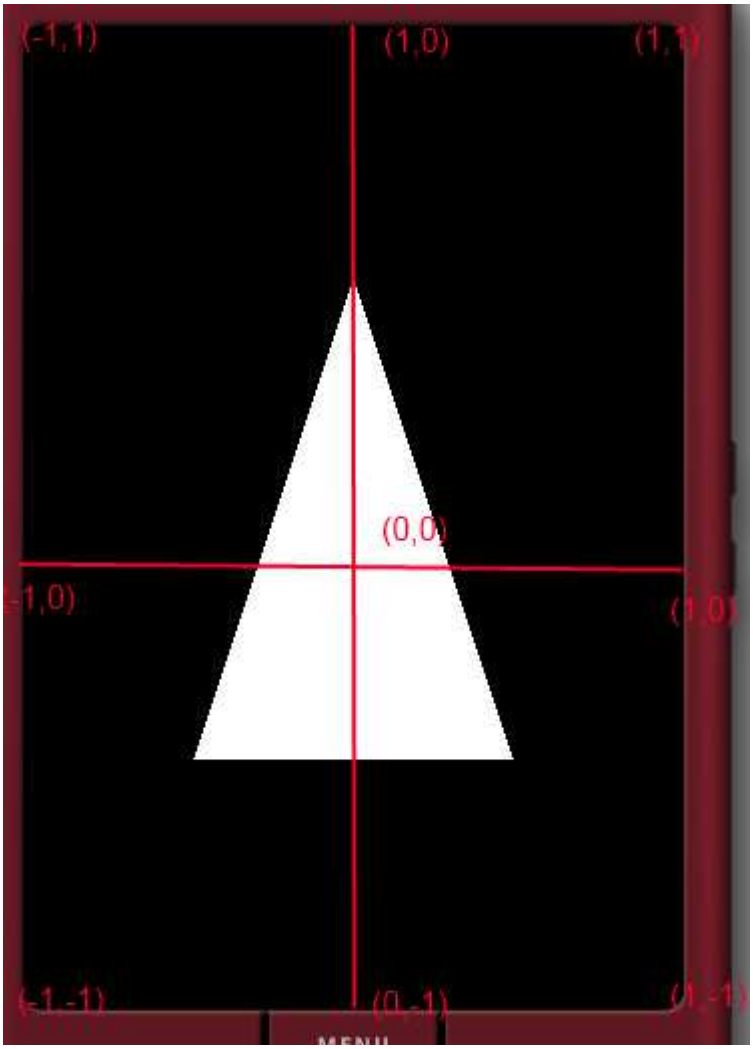
```
public class Cadrado3D extends Game {

	private Texture textura;
	private PerspectiveCamera camara;

	private Mesh cadrado;

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

		cadrado = new Mesh(true,4,6,
				new VertexAttribute(Usage.Position,3,"a_position"),
				new VertexAttribute(Usage.TextureCoordinates,2,"a_textcoords")
		);
		cadrado.setVertices(new float[] {-1f,-1f,-3f,0,1,
                                                 1f,-1f,-3f,1,1,
                                                 -1f,1f,-3f,0,0,
                                                 1f,1f,-3f,1,0
```

```
                                                                        });
                cadrado.setIndices(new short[]{0,1,2,1,2,3});

                FileHandle imagefile = Gdx.files.internal("texturas/badlogic.jpg");
                textura = new Texture(imagefile);



        } // fin do create

        public void resize(int width, int height) {

        float aspectRatio = (float) width / (float) height;

        camara = new PerspectiveCamera();
                camara.viewportWidth = 2f*aspectRatio;
                camara.viewportHeight = 2f;
                camara.near=0.1f;
                camara.far=10f;
                camara.update();


    }

        @Override
        public void render(){
                //Gdx.gl.glClearColor(1,0,0,0);
                GL10 gl = Gdx.gl10;

                gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);

                gl.glMatrixMode(GL10.GL_PROJECTION);
                gl.glLoadIdentity();
                camara.apply(gl);

                gl.glMatrixMode(GL10.GL_MODELVIEW);
                gl.glLoadIdentity();

                Gdx.gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
                Gdx.gl.glEnable(GL10.GL_TEXTURE_2D);

                textura.bind();
                cadrado.render(GL10.GL_TRIANGLES,0,6);

                Gdx.gl.glDisable(GL10.GL_TEXTURE_2D);

        }

}
```

A CAMARA:

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

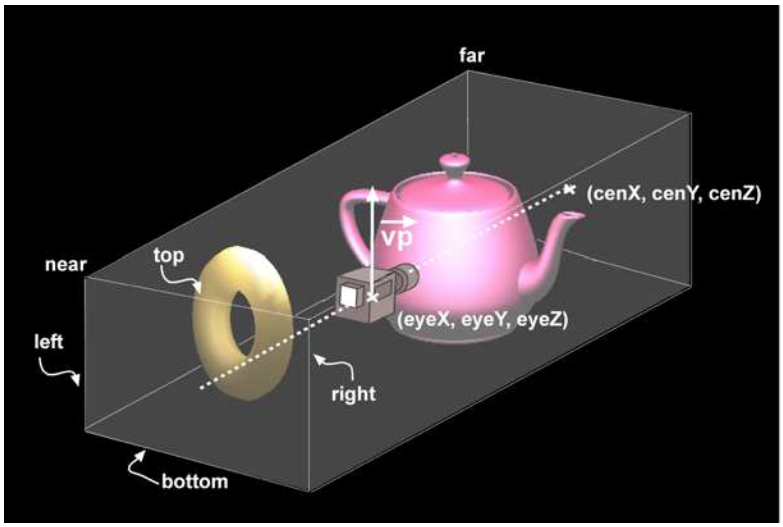
Dita cámara por defecto visualiza o seguinte espazo:

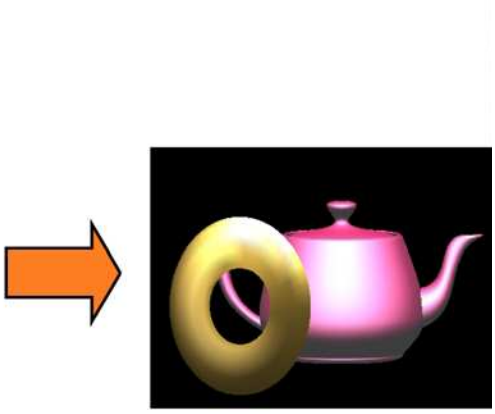


Sendo a coordenada Z a que sairía da pantalla, indo dende +1 (mirando cara nós, saíndo da pantalla) ata -1 (cara dentro da pantalla).

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:





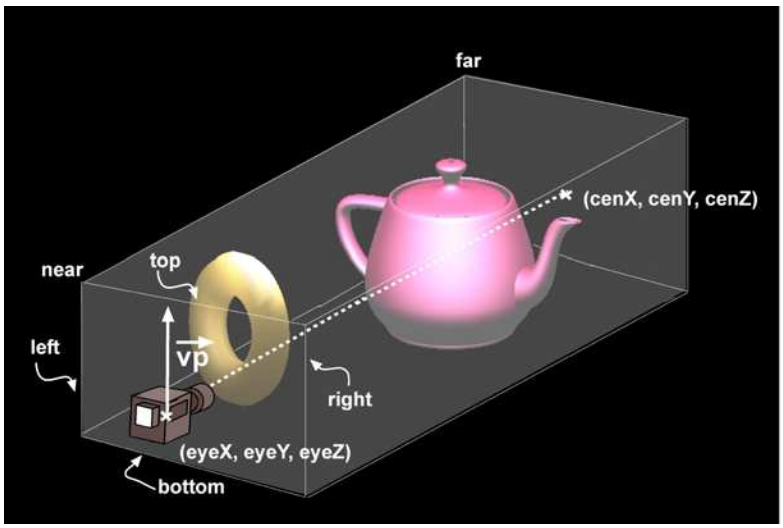
Fixarse na posición da cámara

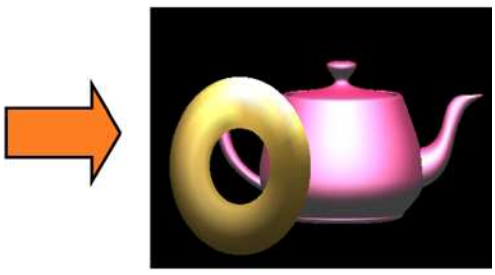
Resultado da Proxección ortográfica

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'.





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).

Nota: non podedes visualizar nada xa que a cámara só chega ata a coordenada Z=-1.

```
private Mesh triangulored,trianguloverde;

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


	triangulored = new Mesh(true, 3, 3,
			new VertexAttribute(Usage.Position, 3, "a1_position"),
			new VertexAttribute(Usage.ColorPacked, 4, "a1_color"));

	triangulored.setVertices(new float[] {
			-0.5f, -0.5f, -3f, Color.toFloatBits(255, 0, 0, 255),
			0.5f, -0.5f, -3f, Color.toFloatBits(255, 0, 0, 255),
			0f, 0.5f, -3f, Color.toFloatBits(255, 0, 0, 255)
			 });

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


	trianguloverde = new Mesh(true, 3, 3,
			new VertexAttribute(Usage.Position, 3, "a2_position"),
			new VertexAttribute(Usage.ColorPacked, 4, "a2_color"));

	trianguloverde.setVertices(new float[] {  // É o verde, se ve detrás
			0f, -0.5f, -5f, Color.toFloatBits(0, 255, 0, 120),
			1.0f, -0.5f, -5f, Color.toFloatBits(0, 255, 0, 120),
			0.5f, 0.5f, -5f, Color.toFloatBits(0, 255, 0, 120) });

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


}

@Override
public void render() {

	Gdx.gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
	triangulored.render(GL10.GL_TRIANGLES, 0, 3);
	trianguloverde.render(GL10.GL_TRIANGLES, 0, 3);
}
```

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.

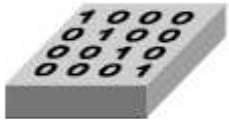
Para elixir o tipo de matriz temos que indicalo así:

```
GL10 gl = Gdx.gl10;

gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
gl.glMatrixMode(GL10.GL_PROJECTION);
```

Nota: estamos traballando coa versión de OPEN GL ES 1.0 xa que é soportada por todos os dispositivos. Se queremos saber se podemos usar outra versión distinta podemos facer uso da función:

```
Gdx.graphics.isGL11Available() ou  Gdx.graphics.isGL20Available()
```

Agora temos que inicializar a matriz (é o que se coñece coma matriz de identidade):



O facemos coa orden:

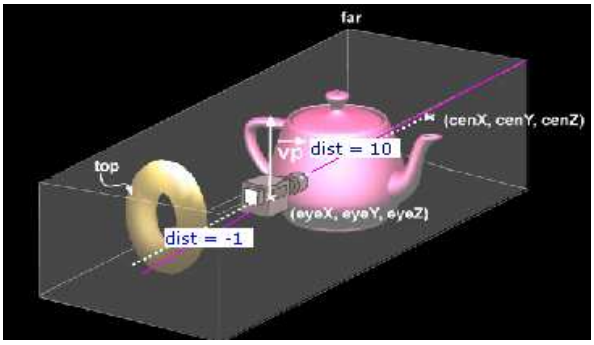
```
gl.glLoadIdentity();
```

Despois temos que definir o View Frustrum (volume de visualización) coar orde glOrthof:

```
gl.glOrthof(-1f  ,  1f   ,  -1f   ,  1f ,    -1f,       10f);
            left , right , bottom , top ,  dist_near , dist_far)
```

Con esta orde se modifica a matriz de proxección para que aplica as operacións necesarias sobre os vértices para ter a proxección indicada.

Parece un pouco lioso xa que os parámetros dist_near e dist_far son distancias con respecto a posición da cámara, de tal forma que o valor negativo supón ir detrás da cámara e un valor positivo supón ir diante da cámara. Despois, iso traducido as coordenadas x,y,z vai depender da posición da cámara (no noso caso posición 0,0,0 por defecto) e polo tanto o volume de visualización vai dende Z = +1 ata Z=-10 na coordenada Z (lembrar que a Z positiva é a que sae da pantalla => regra da man dereita)



Agora temos que cambiar de vista é elixir a matriz ModelView e inicializala para debuxar:

```
gl.glMatrixMode(GL10.GL_MODELVIEW);
gl.glLoadIdentity();
triangulored.render(GL10.GL_TRIANGLES, 0, 3);
trianguloverde.render(GL10.GL_TRIANGLES, 0, 3);
```

Quedando o método así:

```
@Override
public void render() {
	GL10 gl = Gdx.gl10;

	gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
	gl.glMatrixMode(GL10.GL_PROJECTION);
	gl.glLoadIdentity();
	gl.glOrthof(-1f, 1f, -1f, 1f, -1f, 10f);


	gl.glMatrixMode(GL10.GL_MODELVIEW);
	gl.glLoadIdentity();
	triangulored.render(GL10.GL_TRIANGLES, 0, 3);
	trianguloverde.render(GL10.GL_TRIANGLES, 0, 3);
}
```

Nota: Estamos modificando directamente a matriz de proxección coa orden glOrthof, pero podemos facer uso dun obxecto OrthographicCamera para facer o mesmo como veremos a continuación.

Exercicios:

Modificar o View Frustrum para que só apareza o triángulo vermello.

Outra forma de facer o mesmo é utilizando un obxecto da clase `OrthographicCamera`. Para isto creamos un obxecto pertencente a esta clase e no método create engadimos o seguinte código:

```
private OrthographicCamera camara;

@Override
public void create() {
......

        camara = new OrthographicCamera();
        camara.viewportWidth = 2f;
        camara.viewportHeight = 2f;
        camara.near=0.1f;
        camara.far=10f;
        camara.update();
}
```

Fixarse que no constructor da clase OrthographicCamera podemos enviarlle o tamaño do Viewport. O ViewPort ven ser o tamaño que ten a pantalla e no caso anterior se corresponde coa definición do plano Near (é o que definíamos no exercicio anterior coas coordenadas Left-Bottom-Right-Top).

Lembrar que se poñemos isto:

```
        camara = new OrthographicCamera(2f,2f);
```

estaremos situando a cámara na posición (1f,1f) automaticamente (a metade do tamaño x,y pasado no constructor).

Nese caso, se queremos situar á cámara na posición (0,0,0) teríamos que facer a chamada o método:

```
        camara.position.set(x,y,z)
```

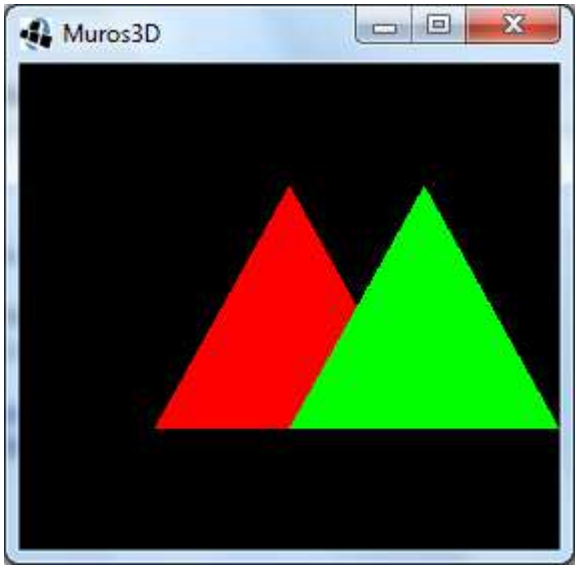
Despois definimos a distancia dos planos Near e Far e obrigatoriamente temos que chamar ó método update.

MOI IMPORTANTE: é necesario chamar o método update() da cámara ante calquera modificación porque se non, non actualiza o View Frustrum.

Despois temos que modificar o método render para que a matriz GL_PROJECTION se modifique coas opcións da cámara:

```
@Override
public void render(){
        GL10 gl = Gdx.gl10;

        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        //gl.glOrthof(-1f, 1f, -1f, 1f, 0f, 10f); // Xa non é necesario
        camara.apply(gl);  // Aplicamos as propiedades modificadas no create

        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();

        triangulo1.render(GL10.GL_TRIANGLES,0,3);
```

```
            triangulo2.render(GL10.GL_TRIANGLES,0,3);
        }
```



Como vemos o resultado é o mesmo.

Nota: Lembra que calquera modificación sobre o tamaño do View Frustrum e os planos Far e Near ten que actualizarse chamando o método update da cámara e aplicar dita cámara a matriz de proxección (`camara.apply(gl);`).

Agora imaxinemos que queremos modificar a posición da cámara (por defecto na posición 0,0,0). Temos que facelo na matriz de Modelado:

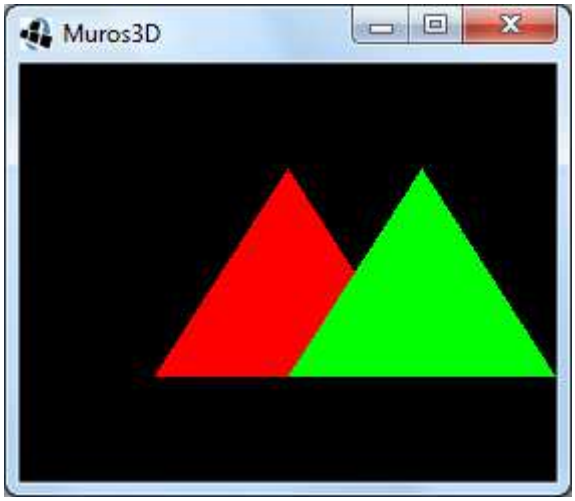
```
            gl.glMatrixMode(GL10.GL_MODELVIEW);
            gl.glLoadIdentity();
            camara.position.set(1,0,0);
            camara.update();
            camara.apply(gl);
```

Exercicio: Modificar a posición da cámara para que só apareza o triángulo verde.

Agora, antes de empezar a ver a cámara de Perspectiva (utilizada para xogos tridimensionais) imos a arranxar outra cousa.

Se vos fixades, os triángulos non teñen en conta a dimensión Z, no sentido que o triángulo vermello que está situado na coordenada Z = -3 debería debuxarse sempre por enriba do triángulo verde situado na coordenada Z = -5.

Sen embargo isto non se cumpre, xa que depende da orde no que se debuxen así aparecen:

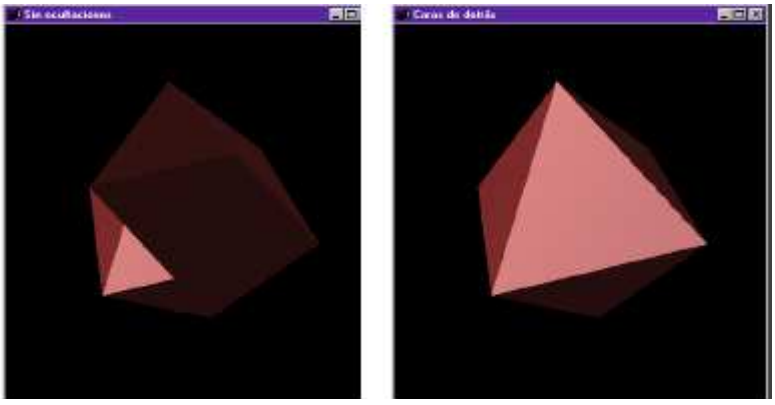



```
triangulloverde.render(GL10.GL_TRIANGLES,0,3);
triangulored.render(GL10.GL_TRIANGLES,0,3);
```

O triángulo verde está situado en Z=-5 (é o de cor verde). Sen embargo aparece por enriba.

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:



sen ocultar ocultando

- 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).

Sen Z-buffer

Con Z-buffer

Nos imos utilizar dita técnica. Para isto temos que:

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

```
public void render(){
        GL10 gl = Gdx.gl10;

        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
```

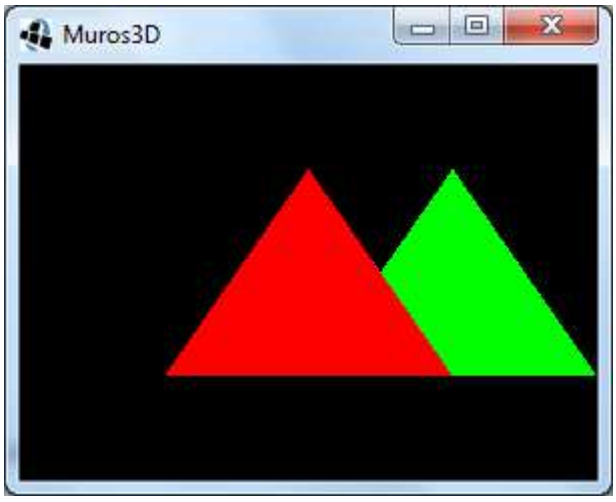
- Antes de debuxar debemos habilitar dito modo:

```
        gl.glEnable(GL10.GL_DEPTH_TEST);
```

- Debuxamos e deshabilitamos o modo:

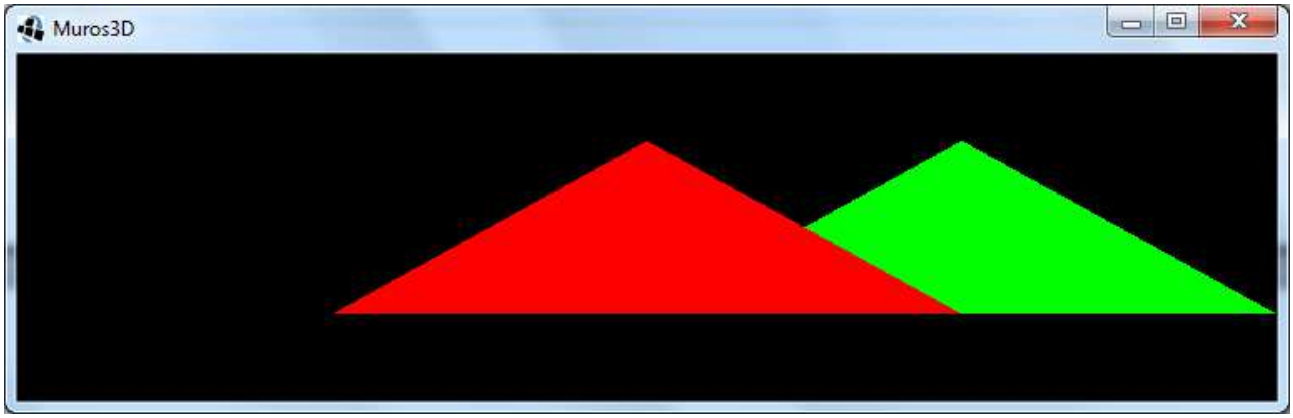
```
        triangulo1.render(GL10.GL_TRIANGLES,0,3);
        triangulo2.render(GL10.GL_TRIANGLES,0,3);

        gl.glDisable(GL10.GL_DEPTH_TEST);
```

Resultado:



Mantendo o aspecto dos triángulos:

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):

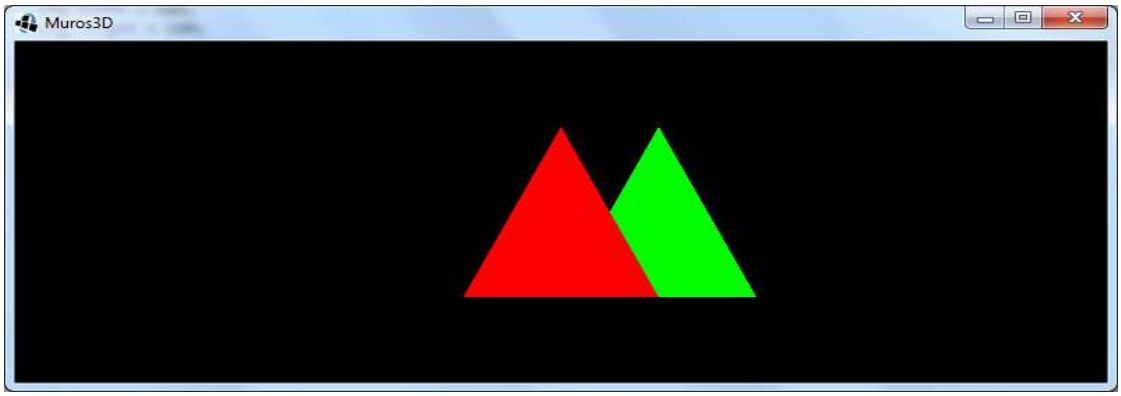
Nota: quitamos todo o código do constructor da clase referido á cámara.

```
public void create() {
......

        camara = new OrthographicCamera();
        camara.viewportWidth = 2f;
        camara.viewportHeight = 2f;
        camara.near=0.1f;
        camara.far=10f;
        camara.update();
}
```

No método resize:

```
public void resize(int width, int height) {

        float aspectRatio = (float) width / (float) height;

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

O resultado agora é este:



Nota: o tamaño que visualiza a cámara é de 2f x 'relacion_de_aspecto'.
Nada impide modificalo para que teña un alto de 60,100,...ou as unidades que nos inventemos do noso mundo. Teremos que axustar o tamaño dos obxectos en relación co tamaño do noso mundo. 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).

Transparencia:

Modificamos a transparencia do triángulo vermello, o situado máis cerca (na posición Z=-3), ó valor 125 nos tres vértices:

```
triangulo1.setVertices(new float[] {-0.5f,-0.5f,-3f,Color.toFloatBits(255, 0, 0, 125),0,1,
            0.5f,-0.5f,-3f,Color.toFloatBits(255, 0, 0, 125),1,1,
            0f,0.5f,-3f,Color.toFloatBits(255, 0, 0, 125),0.5f,0
                                                                    });
```

Nota: lembrade que un valor 1 (255 no noso exemplo) é opaco e un valor 0 é transparente.

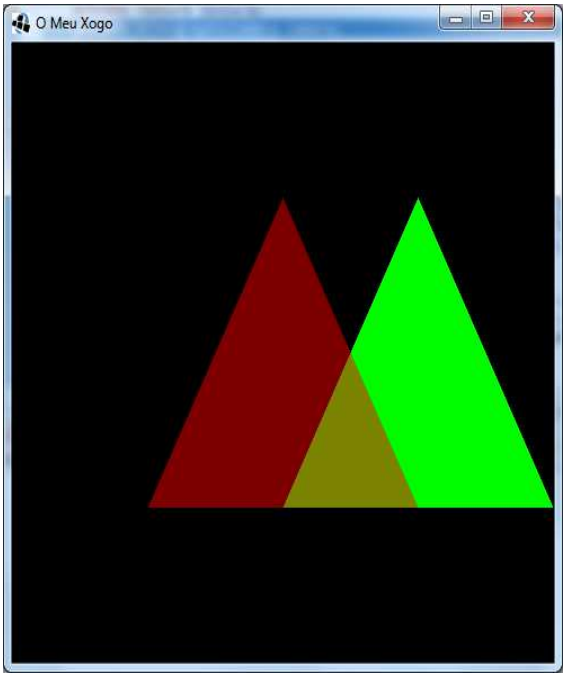
Veremos que non fai nada. Para que funcione a transparencia temos que facer o seguinte:

- 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. Ademáis temos que activar o Blending en Open GL dunha forma moi parecido a como activamos o Z-Buffer:

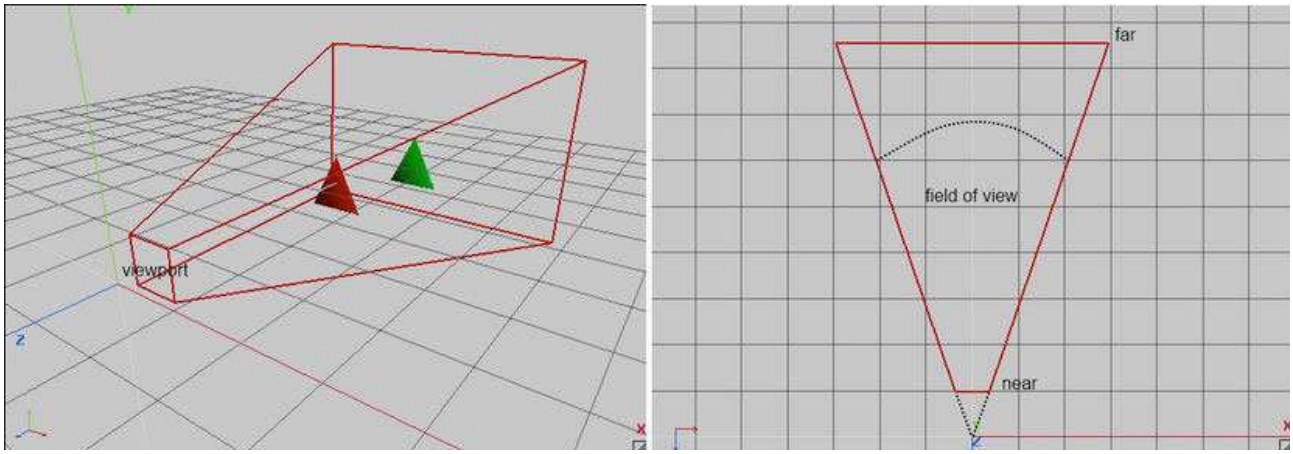
```
@Override
public void render(){
        GL10 gl = Gdx.gl10;

        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);

        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();

        camara.apply(gl);

        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();

        gl.glEnable(GL10.GL_DEPTH_TEST);
        gl.glEnable(GL10.GL_BLEND);
        gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);

        triangulo2.render(GL10.GL_TRIANGLES,0,3);  // CAMBIAMOS A ORDE
        triangulo1.render(GL10.GL_TRIANGLES,0,3);

        gl.glDisable(GL10.GL_BLEND);
        gl.glDisable(GL10.GL_DEPTH_TEST);

}
```

O resultado:

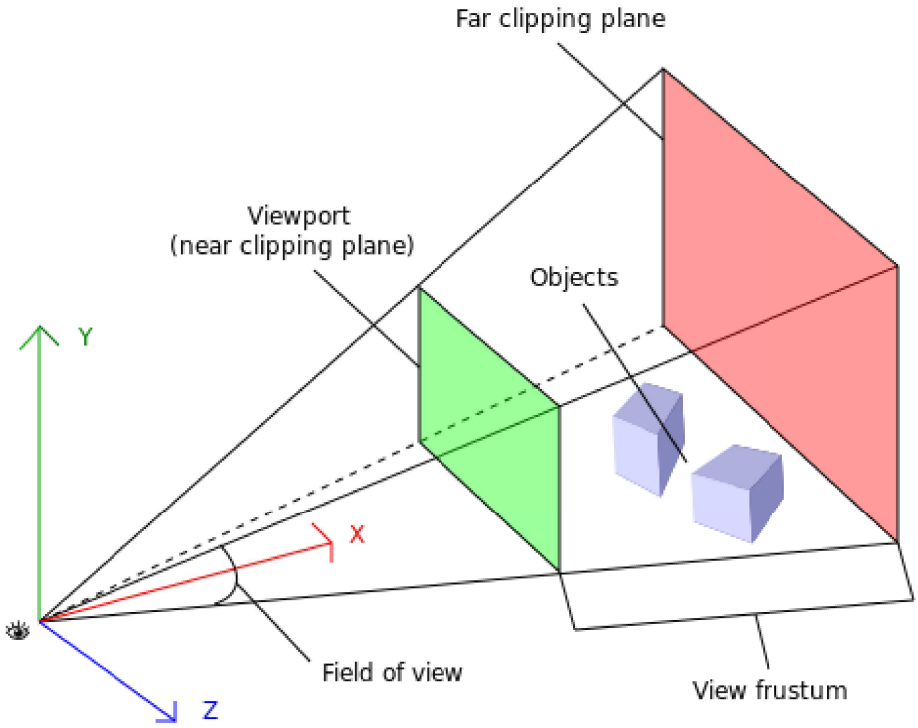


A cámara en perspectiva.

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).

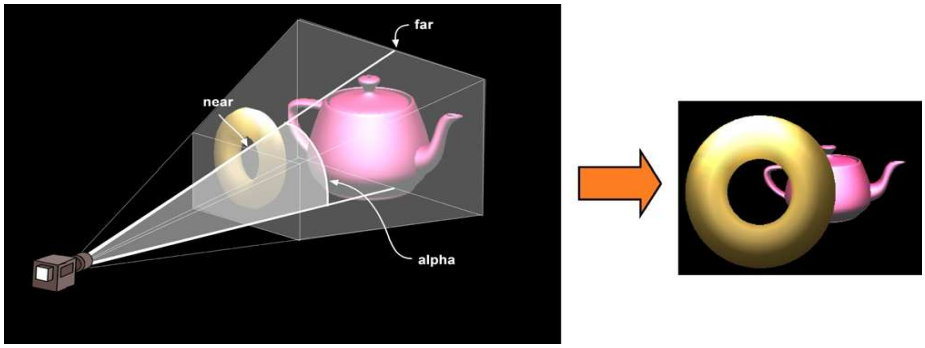


Temos que definir unha cámara que proporcione unha proxección en perspectiva fronte a proxección ortogonal.

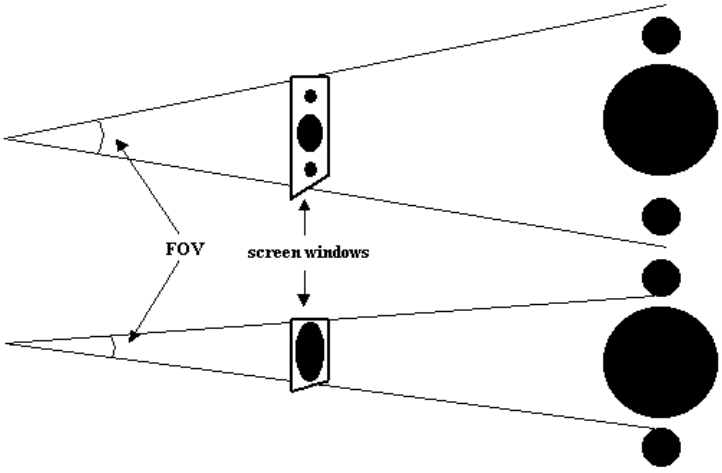
Para definir dita perspectiva o podemos facer como o fixemos no caso de cámara ortogonal (definindo o tamaño frontal do plano e establecendo a distancia dos planos near e far.

```
private PerspectiveCamera camara;

public void resize(int width, int height) {

        float aspectRatio = (float) width / (float) height;

        camara = new PerspectiveCamera();
        camara.viewportWidth = 2f*aspectRatio;
        camara.viewportHeight = 2f;
        camara.near=0.1f;
        camara.far=10f;
        camara.update();
}
```

Por defecto establece un ángulo alfa de 67 grados, que se pode establecer ou ben no constructor ou utilizando o atributo fieldOfView.



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,...).

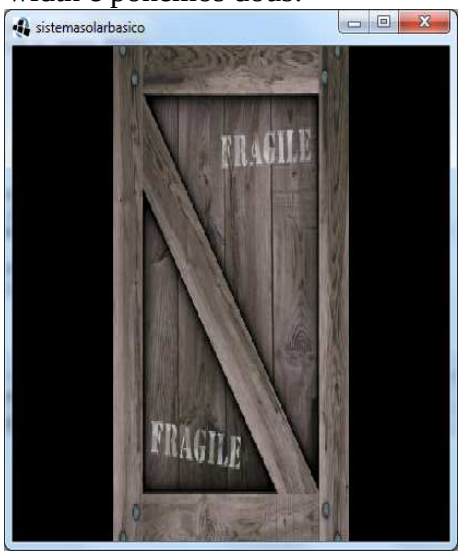
```
cfg.width = 400;
cfg.height = 400;
```

Situamos a cámara na posición (0f,0f,1.2f).
Definimos a cámara co 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:

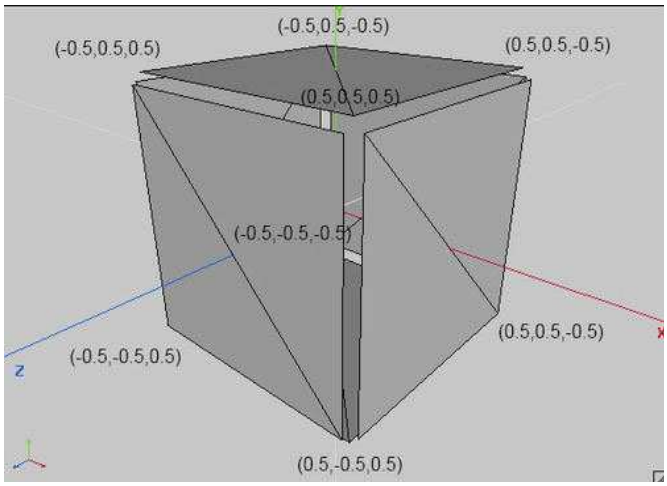
```
camara = new PerspectiveCamera(67, width*aspectRatio ,height);
```

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

Creando cubos animados.

Imos crear unha nova clase Cubo3D.java e imos copiar a textura dunha caixa (no DVD /Proxectos Eclipse/Proxecto 3D planetas/0.- Explicación inicial (Triángulo-Cubo)/crate.png) no cartafol Assets/texturas/ do proxecto de Android.

Imos crear un obxecto Mesh onde imos definir os vértices que conforman os cubos de acordo co seguinte esquema:



Para facelo imos a utilizar os conceptos aprendidos cos triángulos, pero facendo algunha modificación.

Cando fixemos o proxecto dos triángulos definimos dous grupos de vértices, un para cada triángulo. Pero tamén podemos definir os dous triángulos nun mesmo grupo de vértices da seguinte forma:

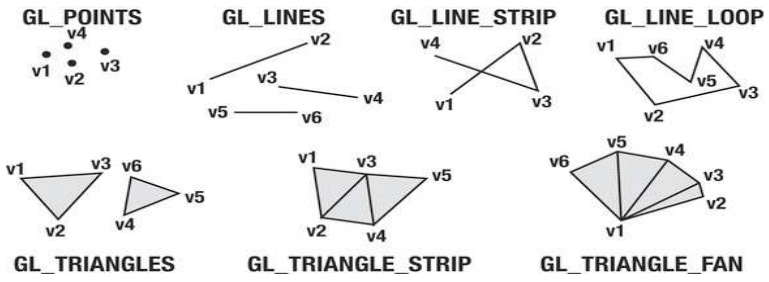
```
public class Triangulos3D extends Game {

	private Mesh triangulos;
	private Texture textura;
	//private OrthographicCamera camara;
	private PerspectiveCamera camara;

	@Override
	public void create() {
		// TODO Auto-generated method stub
		triangulos = new Mesh(true,6,6,
				new VertexAttribute(Usage.Position,3,"a_position"),
				new VertexAttribute(Usage.ColorPacked,4,"a_color")
				);

		triangulos.setVertices(new float[] {-0.5f,-0.5f,-3f,Color.toFloatBits(255, 0, 0, 255),
				0.5f,-0.5f,-3f,Color.toFloatBits(255, 0, 0, 255),
				0f,0.5f,-3f,Color.toFloatBits(255, 0, 0, 255),
				0f,-0.5f,-5f,Color.toFloatBits(0, 255, 0, 125),
				1f,-0.5f,-5f,Color.toFloatBits(0, 255, 0, 125),
				0.5f,0.5f,-5f,Color.toFloatBits(0, 255, 0, 125)
				});
		triangulos.setIndices(new short[]{0,1,2,3,4,5});

	} // fin do create

	public void resize(int width, int height) {

	float aspectRatio = (float) width / (float) height;
```

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


    }

        @Override
        public void render(){
                //Gdx.gl.glClearColor(1,0,0,0);
                GL10 gl = Gdx.gl10;

                gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);

                gl.glMatrixMode(GL10.GL_PROJECTION);
                gl.glLoadIdentity();
                //gl.glOrthof(-1f, 1f, -1f, 1f, 0f, 10f);
                camara.apply(gl);

                gl.glMatrixMode(GL10.GL_MODELVIEW);
                gl.glLoadIdentity();

                gl.glEnable(GL10.GL_DEPTH_TEST);

                triangulos.render(GL10.GL_TRIANGLES,0,3);
                triangulos.render(GL10.GL_TRIANGLES,3,3);

                gl.glDisable(GL10.GL_DEPTH_TEST);

        }

}
```

Está marcado en negro os cambios que fixemos. Agora os vértices están xuntos. Cando debuxamos (render) informamos que imos debuxar un triángulo que ten a información dos vértices no array de vértices. Un dos triángulos empeza na posición 0 do array e ten tres vértices, mentres que o triángulo2 empeza na posición 3 (0,1,2 => triángulo1) e ten tres vértices.

A orde de debuxo dos vértices ven indicado polo array de índices (triángulo1 => 0,1,2 ; triángulo2 => 3,4,5).

Nota: Poderíamos aproveitar os triángulos que comparten lados e aforrarnos puntos no array de vértices.



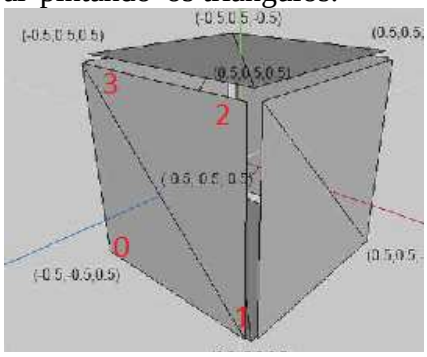
Volvendo ó Cubo, vemos que cada lado do cubo vai necesitar 4 vértices, por 6 caras = 24 vértices. Por outra banda cada triángulo necesita 3 índices. Se cada cara ten dous triángulos isto da:

3 índices por triángulo x 2 triángulos por cara x 6 caras = 36 índices.

Polo tanto para crear o cubo temos que facer:

```
float[] vertices = { -0.5f, -0.5f,  0.5f, 0, 1,
        0.5f, -0.5f,  0.5f, 1, 1,
        0.5f,  0.5f,  0.5f, 1, 0,
       -0.5f,  0.5f,  0.5f, 0, 0,

        0.5f, -0.5f,  0.5f, 0, 1,
        0.5f, -0.5f, -0.5f, 1, 1,
        0.5f,  0.5f, -0.5f, 1, 0,
        0.5f,  0.5f,  0.5f, 0, 0,

        0.5f, -0.5f, -0.5f, 0, 1,
       -0.5f, -0.5f, -0.5f, 1, 1,
       -0.5f,  0.5f, -0.5f, 1, 0,
        0.5f,  0.5f, -0.5f, 0, 0,

       -0.5f, -0.5f, -0.5f, 0, 1,
       -0.5f, -0.5f,  0.5f, 1, 1,
       -0.5f,  0.5f,  0.5f, 1, 0,
       -0.5f,  0.5f, -0.5f, 0, 0,

       -0.5f,  0.5f,  0.5f, 0, 1,
        0.5f,  0.5f,  0.5f, 1, 1,
        0.5f,  0.5f, -0.5f, 1, 0,
       -0.5f,  0.5f, -0.5f, 0, 0,

       -0.5f, -0.5f,  0.5f, 0, 1,
        0.5f, -0.5f,  0.5f, 1, 1,
        0.5f, -0.5f, -0.5f, 1, 0,
       -0.5f, -0.5f, -0.5f, 0, 0
};

    short[] indices = { 0, 1, 3, 1, 2, 3, // Os dous triángulos dunha das caras => mirar gráfico seguinte
              4, 5, 7, 5, 6, 7,
              8, 9, 11, 9, 10, 11,
              12, 13, 15, 13, 14, 15,
              16, 17, 19, 17, 18, 19,
              20, 21, 23, 21, 22, 23,
    };

    private Mesh cubo;
```

Os dúos números do final fan referencia á textura que vai levar cada cara (xa o explicamos cos triángulos).
Fixarse como o array de índices vai 'pintando' os triángulos:

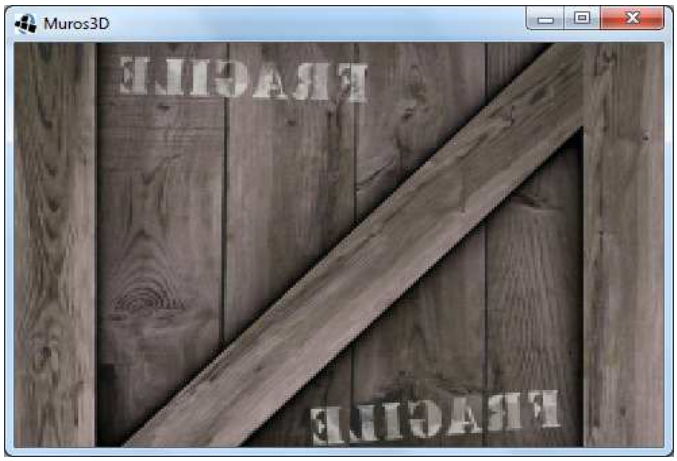


A maiores do cubo creamos o resto de propiedades necesarias coma a textura, método resize e

render:

```
        PerspectiveCamera camara;
        Texture cuboTexture;
        FPSLogger fps;


        @Override
        public void create() {


                // TODO Auto-generated method stub
                cubo = new Mesh(true,24,36,
                                new VertexAttribute(Usage.Position, 3, "a_position"),
                                new VertexAttribute(Usage.TextureCoordinates, 2, "a_texCoords"));

                cubo.setVertices(vertices,0,vertices.length);
                cubo.setIndices(indices,0,indices.length);

                FileHandle imageFileHandle = Gdx.files.internal("texturas/crate.png");
                cuboTexture = new Texture(imageFileHandle);

                fps=new FPSLogger();

        }

    @Override
    public void render() {
        GL10 gl = Gdx.gl10;
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);

        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();

        camara.apply(gl);

        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();

        gl.glEnable(GL10.GL_DEPTH_TEST);
        gl.glEnable(GL10.GL_TEXTURE_2D);

        cuboTexture.bind();
        cubo.render(GL10.GL_TRIANGLES, 0, 36);

        gl.glDisable(GL10.GL_TEXTURE_2D);
        gl.glDisable(GL10.GL_DEPTH_TEST);

        fps.log();
    }

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

        float aspectRatio = (float) width / (float) height;
        camara = new PerspectiveCamera(67, 2f * aspectRatio, 2f);
        camara.far=10;
        camara.near=0.1f;
        camara.update();
    }
}
```

Fixarse como se facemos a proba de visualizar da como resultado isto:



Isto é así xa que a cámara está situada na posición 0,0,0, e o cubo na posición z entre -0.5 e 0.5.

O lóxico sería mover a cámara. Pero temos que movela non no método resize, se non que temos que aplicar dito movemento á matriz de modelado (ModelView). Se o facemos no constructor estaremos movendo a cámara, pero o cubo se debuxará igual diante dela, xa que lembrade que o que definimos no cubo non son as súas coordenadas no mundo, se non o seu tamaño nun mundo 3D.

Polo tanto facemos a seguinte modificación no método render:

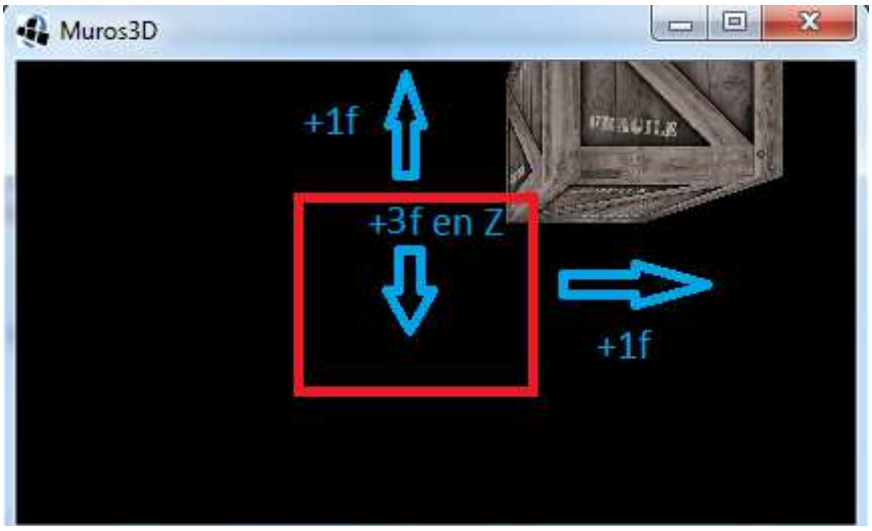
```
        ..................
gl.glMatrixMode(GL10.GL_MODELVIEW);
gl.glLoadIdentity();

gl.glEnable(GL10.GL_DEPTH_TEST);
gl.glEnable(GL10.GL_TEXTURE_2D);

camara.position.set(0f,0f,5f);
camara.update();
camara.apply(gl);

cuboTexture.bind();
cubo.render(GL10.GL_TRIANGLES, 0, 36);
        ..................
```

Nota: lembrade que o signo positivo da Z é 'saíndo' da pantalla.

Agora imaxinade que queremos facer algún tipo de modificación sobre o cubo (escalado, movemento ou rotación).

Para facelo temos que utilizar as funcións de open gl: glTranslate, glRotate, glScale.
Se queremos mover o cubo 1 unidade en x,y e 3 en z usaremos glTranslate:

```
camara.position.set(0f,0f,5f);
camara.update();
camara.apply(gl);

gl.glTranslatef(+1f, +1f, +3f);

cuboTexture.bind();
cubo.render(GL10.GL_TRIANGLES, 0, 36);
```



Nota: nas funcións de rotación (`gl.glRotatef(angulo, X, Y, Z))`, se poñemos 1 na coordenada se aplica a rotación a esa coordenada e se poñemos 0 non se aplica. Por exemplo: `gl.glRotatef(35, 1, 0, 0)` rotaría 35 grados o eixe X.

Exercicio:

- Facer que o cubo mida de ancho o dobre que de alto.
- Rotar o cubo 35 grados no eixe Y.

Tamén podemos facer que o cubo rote de forma continua. Para facelo creamos unha variable angulo de tipo float e a incrementamos ata 360 (360 grados), aplicando dito ángulo á matriz de modelado:

```
        private float angulo=0f

Método render:
        ...........
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();

        gl.glEnable(GL10.GL_DEPTH_TEST);
        gl.glEnable(GL10.GL_TEXTURE_2D);

        camara.position.set(0f,0f,5f);
        camara.update();
        camara.apply(gl);

        angulo += Gdx.graphics.getDeltaTime()*20f;
        gl.glRotatef(angulo, 1, 1, 1);
        if (angulo>=360) angulo = 0;
        cuboTexture.bind();
        cubo.render(GL10.GL_TRIANGLES, 0, 36);
        ............
```

Isto dará como resultado o cubo rotando nos tres eixes.

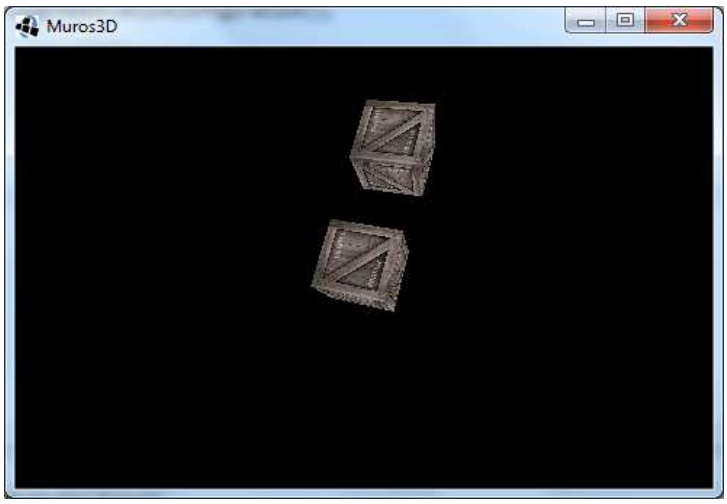
A pila de matrices.

Supoñamos agora que queremos dibuxar outro cubo noutra posición (en X+2).
Para isto teremos que facer unha traslación no eixe X, pero temos un problema:

Método render:

```
.....
angulo += Gdx.graphics.getDeltaTime()*30f;
gl.glRotatef(angulo, 1, 1, 1);
if (angulo>=360) angulo = 0;

cuboTexture.bind();
cubo.render(GL10.GL_TRIANGLES, 0, 36);

gl.glTranslatef(2,0,0);
cubo.render(GL10.GL_TRIANGLES, 0, 36);
.....
```

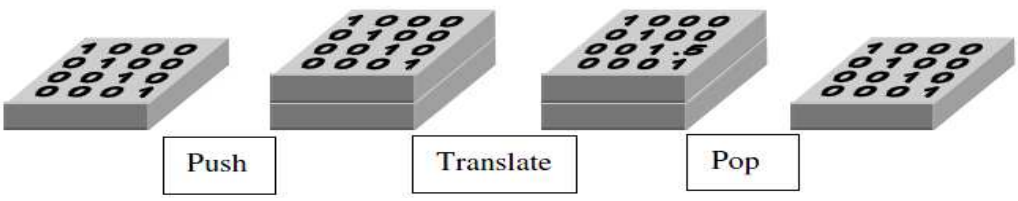
Os dous cubos se moven á vez:



Isto se debe a que a matriz que se modifica (coa rotación) se aplica ós dous cubos, polo que se facemos unha rotación con respecto o punto 0,0,0 e despois desprazamos o cubo, as súas coordenadas x,y,z estarán cambiadas pola rotación, e é por iso polo que fai un movemento de traslación.

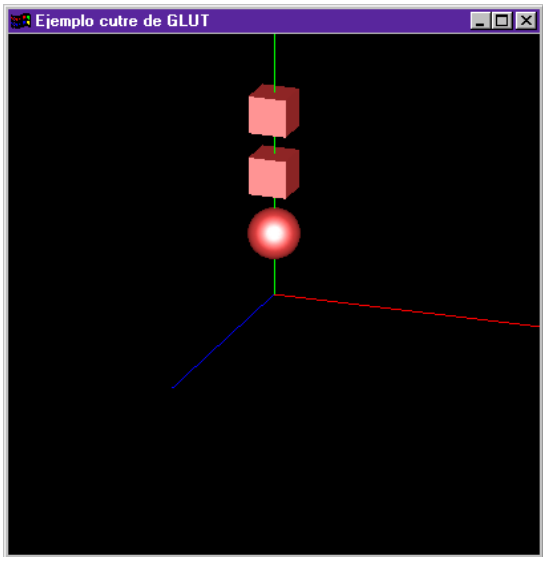
Teríamos que recuperar a matriz 'antes' de facer a rotación para que se aplique ó segundo cubo. Isto se fai cunha estrutura de pila de tipo LIFO (Last In First Out) de ata 10 matrices de tipo ModelView e dúas de tipo Projection. Normalmente usaremos só as de ModelView.

Os procedementos son: **glPushMatrix**() e **glPopMatrix**() (sobre o obxecto da clase GLXX).

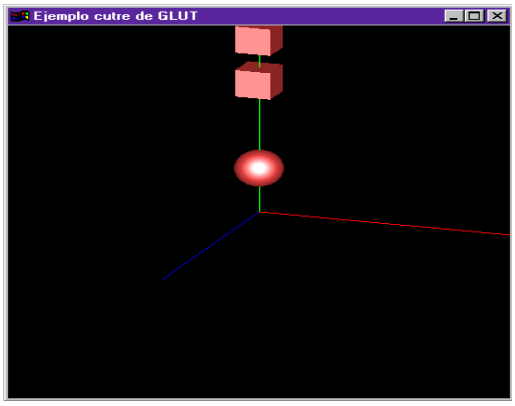


Vexamos outros exemplos feitos en open gl:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glPushMatrix();
glTranslatef(0.0f, 0.25f, 0.0f);
glutSolidSphere(0.1, 30, 30);  // Dibuxa unha esfera en open gl
glPopMatrix();
glPushMatrix();
glTranslatef(0.0f, 0.5f, 0.0f);
glutSolidCube(0.15);  // Dibuxa un cubo en open gl
glTranslatef(0.0f, 0.25f, 0.0f);
glutSolidCube(0.15); //  Dibuxa un cubo en open gl
glPopMatrix();
```




```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glPushMatrix();
glTranslatef(0.0f, 0.25f, 0.0f);
glutSolidSphere(0.1, 30, 30);
glTranslatef(0.0f, 0.5f, 0.0f);
glutSolidCube(0.15);
glTranslatef(0.0f, 0.25f, 0.0f);
glutSolidCube(0.15);
glPopMatrix();
```



Como vemos nos exemplos, as operacións que facemos sobre a matriz de modelado son acumulativas, quere dicir que se a modifico e fago unha translación debuxo algo e despois fago unha rotación e debuxo algo, a segunda figura terá a translación e a rotación.

Exercicio:

Aplicando estes conceptos fai que o segundo cubo non rote e permaneza á dereita do primeiro que sí rota.

Solución:

Método render():

```
        ..................
gl.glMatrixMode(GL10.GL_MODELVIEW);
gl.glLoadIdentity();

gl.glEnable(GL10.GL_DEPTH_TEST);
gl.glEnable(GL10.GL_TEXTURE_2D);

camara.position.set(0f,0f,5f);
camara.update();
camara.apply(gl);

gl.glPushMatrix(); // Gardamos a matriz unha vez movida a cámara.

angulo += Gdx.graphics.getDeltaTime()*30f;
gl.glRotatef(angulo, 1, 1, 1);
if (angulo>=360) angulo = 0;

cuboTexture.bind();
cubo.render(GL10.GL_TRIANGLES, 0, 36);

gl.glPopMatrix();  // Recuperamos a matriz por tanto xa non está rotada

gl.glTranslatef(2,0,0);
cubo.render(GL10.GL_TRIANGLES, 0, 36);

gl.glDisable(GL10.GL_TEXTURE_2D);
gl.glDisable(GL10.GL_DEPTH_TEST);
        ..................
```

Exercicio: Facer que o segundo cubo tamén rote como o primeiro.

Solución;

Primeiro debemos trasladar o cubo e despois rotalo:

```
.....................
camara.position.set(0f,0f,5f);
camara.update();
camara.apply(gl);

gl.glPushMatrix();

angulo += Gdx.graphics.getDeltaTime()*30f;
gl.glRotatef(angulo, 1, 1, 1);

cuboTexture.bind();
cubo.render(GL10.GL_TRIANGLES, 0, 36);

gl.glPopMatrix();
gl.glTranslatef(2,0,0);
gl.glRotatef(angulo, 1, 1, 1);

cubo.render(GL10.GL_TRIANGLES, 0, 36);

gl.glDisable(GL10.GL_TEXTURE_2D);
gl.glDisable(GL10.GL_DEPTH_TEST);

if (angulo>=360) angulo = 0;
.................
```

Exercicio: Crea outro cubo que estea a esquerda e máis adiante (cara fora de pantalla) do cubo central e que teña a metade de tamaño.

Exercicio: Crea tres cubos por enriba do central. O primeiro ten que estar a dous unidades por enriba. O segundo a dous unidades á dereita do primeiro e o terceiro a dous unidades a esquerda do primeiro.

Solución:

```
public void render(){
        //Gdx.gl.glClearColor(1,0,0,0);
        GL10 gl = Gdx.gl10;

        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        camara.apply(gl);

        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();

        gl.glEnable(GL10.GL_DEPTH_TEST);
        gl.glEnable(GL10.GL_TEXTURE_2D);

        camara.position.set(0,0,10);
        camara.update();
        camara.apply(gl);

        gl.glPushMatrix();

        angulorotacion += 20f*Gdx.graphics.getDeltaTime();
        gl.glRotatef(angulorotacion, 1, 1, 1);
        if (angulorotacion>=360)
                angulorotacion = 0;

        textura.bind();
        cubo.render(GL10.GL_TRIANGLES,0,36);

        gl.glPopMatrix();
        gl.glPushMatrix();

        gl.glTranslatef(2f, 0, 0);
        gl.glScalef(0.5f, 0.5f, 0.5f);
        cubo.render(GL10.GL_TRIANGLES,0,36);

        gl.glPopMatrix();
        gl.glPushMatrix();

        gl.glTranslatef(-2f, 0, 0);
        gl.glRotatef(angulorotacion, 1, 0, 0);
        gl.glScalef(1.5f, 1.5f, 1.5f);
        cubo.render(GL10.GL_TRIANGLES,0,36);

        gl.glPopMatrix();  // ORIXINAL
        gl.glPushMatrix();  // A GARDAMOS POR SE A NECESITARAMOS PARA MAIS ADIANTE
        gl.glTranslatef(0, 2f, 0);
        cubo.render(GL10.GL_TRIANGLES,0,36);

        gl.glPushMatrix();
        gl.glTranslatef(2f, 0, 0);
        cubo.render(GL10.GL_TRIANGLES,0,36);

        gl.glPopMatrix();
        gl.glTranslatef(-2f, 0, 0);
        cubo.render(GL10.GL_TRIANGLES,0,36);

        gl.glPopMatrix();  // E A ORIXINAL

        gl.glDisable(GL10.GL_TEXTURE_2D);
        gl.glDisable(GL10.GL_DEPTH_TEST);

}
```

Cabe sinalar que temos outros métodos para manexar a posición da cámara, coma son o método **lookAt(x,y,z)** que indica cara onde está mirando a cámara no noso mundo. Dito método se ten que aplicar á cámara, facer o update e aplicar os cambios á matriz ModelView.
Tamén temos os métodos `camara.rotateAround(point, axis, angle)` que rota a cámara arredor dun punto nun dos eixes, ou `camara.direction.set(x,y,y)` que modifica a dirección cara onde está mirando a cámara.

NOTA: se a cámara non vai modificarse no seu tamaño (viewport) e ángulo de visión (field of view) pódese quitar todo o referido á matriz de proxeción do método render e poñelo no resize.

Carga de arquivos gráficos: obj.

Loxicamente os nosos modelos 3D non vai ser creados 'manualmente', se non utilizando programas gráficos.

Algún deles:

Blender: http://www.blender.org/
Moi completo pero tamén bastante complexo.
Wings3D: http://www.wings3d.com
Bastante máis sinxelo.

Nos imos usar o wings3d.
Se atopa no DVD (/SOFTWARE/3D/wings-1.4.1.exe).

En canto os formatos gráficos que podemos cargar están, entre outros:

- obj: https://en.wikipedia.org/wiki/Wavefront_.obj_file
- md2: https://en.wikipedia.org/wiki/MD2_%28file_format%29
- fbx: http://en.wikipedia.org/wiki/FBX
- g3dj: Json
- g3db: Binario.

Para pasar dun formato a outro podemos utilizar unha ferramenta: fbx-conv.
Podemos ver o código fonte en: https://github.com/libgdx/fbx-conv
e podemos descargar a versión compilada en: http://libgdx.badlogicgames.com/fbx-conv/

Se atopa no DVD (/SOFTWARE/3D/fbx-conv).

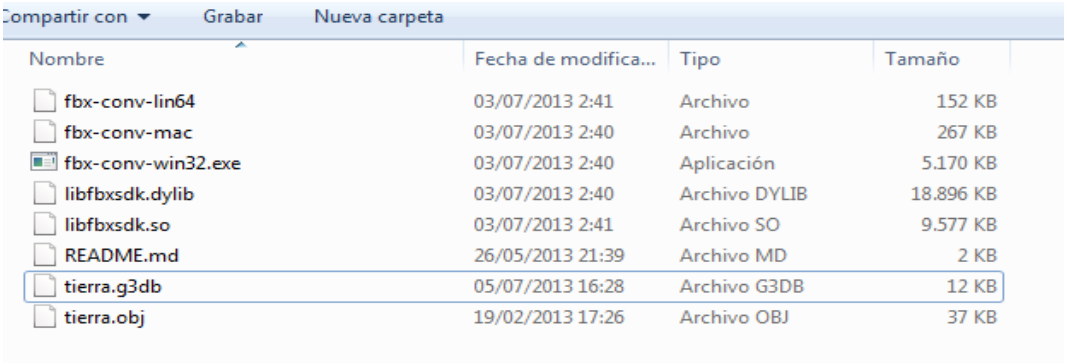
No caso de windows teremos que ir cunha consola ó cartafol onde copiamos os executables descargados e escribir:

```
fbx-conv-win32.exe -o g3db tierra.obj
```

sendo 'tierra.obj' o arquivo a converter que está copiado no mesmo cartafol.

Neste caso crearíamos un modelo de nome tierra.g3db

NOTA: Sempre que queiramos acelerar a carga dos modelos deberemos de pasalos a binario.



Podemos ver como a versión binaria ocupa menos que o arquivo obj (e ademais se carga máis rápido).

Para cargala teríamos que utilizar un AssetManager da forma:

```
AssetManager am = new AssetManager();
am.load("texturas/tierra.g3db", Model.class);

am.finishLoading();
Model a = am.get("texturas/tierra.g3db");
mesh_planetas = a.meshes.get(0);
```

Fixarse como a carga dun arquivo gráfico se garda nunha clase Model. Esta clase está, ou pode estar, composta por varios meshe´s, de aí que utilicemos o método get para obter o único mesh que sabemos que ten o modelo.

Podemos buscar por Internet diferentes sitios onde atopar modelos gratuítos para descargar:

http://www.hongkiat.com/blog/60-excellent-free-3d-model-websites/
http://thefree3dmodels.com/

No VIDEO que hai no DVD (/Videos/) vai explicado como facer un modelo e asociarlle unha textura.

No noso proxecto non imos a usar o assetmanager e cargaremos os modelos directamente facendo uso da clase ObjLoader.

XOGO DO SISTEMA SOLAR.

Co que acabamos de ver imos a desenrolar un xogo en 3D.

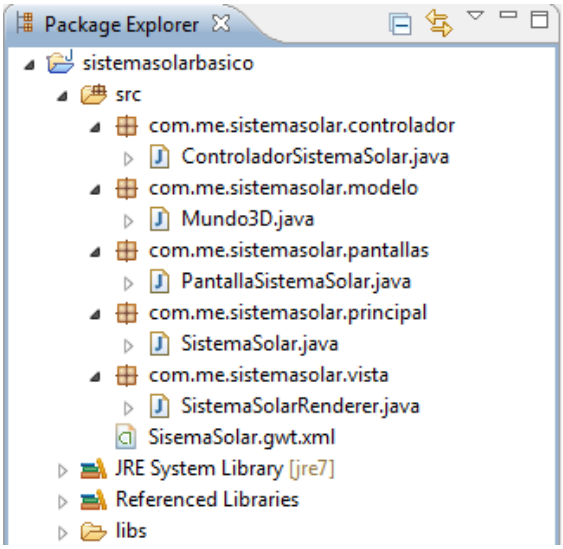
Un nome: CATASTROFE.
A historia: O gran capitán KIRK ten que pilotar a nave espacial Mini-q no sistema solar e impedir que os meteoritos teledirixidos cheguen á TERRA.
Categoría: casual

Imos desenrolar o xogo para unha resolución das máis utilizadas, de 480x800. Podemos poñer esta mesma na versión Desktop para ver como queda.

Só faremos a pantalla do xogo, quedando como exercicio o desenrolo do resto das pantallas.

Importaremos dende o DVD o proxecto 'sistemasolarbasico' (/Proxectos Eclipse/Proxecto 3D planetas/1.-inicial/).

Teremos por tanto a seguinte estrutura:



O primeiro será pensar a capa modelo.
O noso mundo non vai ter un 'tamañó' xa que estamos nun espazo 3D. Podemos limitar o que vai visualizar a cámara co plano far e near, pero isto xa o temos feito na clase SistemaSolarRenderer no paquete vista.

Imos comezar creando o Sol e os planetas. Todos van ter a mesma información, polo que creamos unha clase Planeta dentro do paquete modelo.
Todos os planetas, incluído o sol, van ter unha posición no espazo 3D, unha velocidade de rotación e unha velocidade de translación. Relacionado coa velocidade teremos que gardar o ángulo de rotación e translación de cada planeta para debuxalo adecuadamente.

Ademais, como imos usar o mesmo obj para todos os planetas (e este ten un tamaño de dous unidades no espazo 3D) teremos que escalar os planetas aplicando a operación de opengl glScale e polo tanto teremos que gardar dita información.

Como todos os planetas van ter uns datos diferentes, podemos facer que o constructor por defecto sexa o que teña a información do sol e poñeremos unha escala de 1000.

Para identificar a cada planeta podemos mandarlle o nome do mesmo no constructor.

Arquivo Planeta.java

```
public class Planeta {

	public static enum NOMES_PLANETAS
{SOL,MERCURIO,VENUS,TIERRA,MARTE,JUPITER,SATURNO,URANO,NEPTUNO,PLUTON};


	private Vector3 posicion;
	private float angulo_rotacion=0,angulo_traslacion=0;
	private float velocidaderotacion=0,velocidadetraslacion=0;
	private Vector3 escala = new Vector3();
	private NOMES_PLANETAS nome_planeta;


	public Planeta(){
		posicion = new Vector3(0,0,0);
		angulo_rotacion=0;
		angulo_traslacion=0;
		velocidaderotacion=0.3f;
		velocidadetraslacion=0f;
		escala.set(1000f,1000f,1000f);
		nome_planeta=NOMES_PLANETAS.SOL;
	}

	public Planeta(float x, float y, float z,float velocidaderotacion,float velocidadetraslacion,
NOMES_PLANETAS nomeplaneta, float escala){
		posicion = new Vector3(x,y,z);

		this.velocidadetraslacion=velocidadetraslacion;
		this.velocidaderotacion=velocidaderotacion;

		this.escala.set(escala,escala,escala);
		this.nome_planeta=nomeplaneta;

	}

	public Vector3 getPosicion() {
		return posicion;
	}

	public float getAngulo_rotacion() {
		return angulo_rotacion;
	}

	public float getAngulo_traslacion() {
		return angulo_traslacion;
	}

	public float getVelocidaderotacion() {
		return velocidaderotacion;
	}

	public float getVelocidadetraslacion() {
		return velocidadetraslacion;
	}

	public Vector3 getEscala() {
```

```
		return escala;
	}

	public NOMES_PLANETAS getNome_planeta() {
		return nome_planeta;
	}


}
```

O seguinte será cargar os modelos e texturas que necesitamos.
Os dos planetas se poden descargar dende aquí: http://planetpixelemporium.com/sun.html
Nos xa o temos feito e están no cartafol assets do proxecto copiado.

Por un lado temos que cargar as texturas e por outro os obxectos Mesh que imos a usar.

As texturas as imos gardar nun HashMap utilizando as propiedade enum da clase Planeta.

Arquivo AssetsXogo.java

```
public class AssetsXogo {

	public static HashMap<Planeta.NOMES_PLANETAS,Texture> textura_planetas = new
HashMap<Planeta.NOMES_PLANETAS,Texture>();
	public static Mesh mesh_planetas;

	public static void cargarGraficos(){
		FileHandle imageFileHandle = Gdx.files.internal("texturas/sun.jpg");
		textura_planetas.put(Planeta.NOMES_PLANETAS.SOL, new Texture(imageFileHandle));
		imageFileHandle = Gdx.files.internal("texturas/mercurio.jpg");
		textura_planetas.put(Planeta.NOMES_PLANETAS.MERCURIO, new Texture(imageFileHandle));
		imageFileHandle = Gdx.files.internal("texturas/venus.jpg");
		textura_planetas.put(Planeta.NOMES_PLANETAS.VENUS, new Texture(imageFileHandle));
		imageFileHandle = Gdx.files.internal("texturas/tierra.jpg");
		textura_planetas.put(Planeta.NOMES_PLANETAS.TIERRA, new Texture(imageFileHandle));
		imageFileHandle = Gdx.files.internal("texturas/marte.jpg");
		textura_planetas.put(Planeta.NOMES_PLANETAS.MARTE, new Texture(imageFileHandle));
		imageFileHandle = Gdx.files.internal("texturas/jupiter.jpg");
		textura_planetas.put(Planeta.NOMES_PLANETAS.JUPITER, new Texture(imageFileHandle));
		imageFileHandle = Gdx.files.internal("texturas/saturno.jpg");
		textura_planetas.put(Planeta.NOMES_PLANETAS.SATURNO, new Texture(imageFileHandle));
		imageFileHandle = Gdx.files.internal("texturas/urano.jpg");
		textura_planetas.put(Planeta.NOMES_PLANETAS.URANO, new Texture(imageFileHandle));
		imageFileHandle = Gdx.files.internal("texturas/neptuno.jpg");
		textura_planetas.put(Planeta.NOMES_PLANETAS.NEPTUNO, new Texture(imageFileHandle));
		imageFileHandle = Gdx.files.internal("texturas/pluton.jpg");
		textura_planetas.put(Planeta.NOMES_PLANETAS.PLUTON, new Texture(imageFileHandle));

		ObjLoader loader = new ObjLoader();
		Model model = loader.loadModel(Gdx.files.internal("texturas/tierra.obj"));
		mesh_planetas = model.meshes.get(0);

	}

	public static void liberarGraficos(){
		mesh_planetas.dispose();
		for (Texture textura : textura_planetas.values()){
			textura.dispose();
		}
	}

}
```

Exercicio:

Modificar a clase SistemaSolar do paquete principal para que cargue e libere as texturas e chame á pantalla 'PantallaSistemaSolar'.

Solución:

```
public class SistemaSolar extends Game{

	PantallaSistemaSolar pantallagame;
	public static final String LOG = SistemaSolar.class.getSimpleName();

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

		AssetsXogo.cargarGraficos();
		pantallagame = new PantallaSistemaSolar();
		setScreen(pantallagame);
	}


	public static void imprimirLog(String mensaxe){
		Gdx.app.log(SistemaSolar.LOG,mensaxe);
	}

	@Override
	public void dispose(){
		pantallagame.dispose();
		AssetsXogo.liberarGraficos();
	}
}
```

Agora imos crear o noso mundo. Vai estar formado por nove planetas e o sol.
Podemos considera-lo sol como un planeta e poñer a todos dentro dun array.

Exercicio: face-lo anterior e inicializar os planetas.

NOTA:
Loxicamente o tamaño e velocidades non van ser reais, pero sí tedes que ter en conta o seguinte. O tamaño dos obxectos 3D son relativos os demais obxectos. É dicir, se poñedes unha escala a TERRA de 1f non podedes poñer unha escala o sol de 2f xa que estariades a dicir que o sol só e dous veces maior que a Terra.

O importante por tanto é manter a escala dos obxectos. Lembrade que cando debuxamos o obxecto 3D que representa os planetas e o sol tiña un tamaño de 2 cadrados na cuadrícula 3D.

Os datos dos planetas postos por min son:

```
Planeta(1500,0,0,10f,3f,Planeta.NOMES_PLANETAS.MERCURIO,3f)
Planeta(2000,0,0,5f,1f,Planeta.NOMES_PLANETAS.VENUS,9f)
Planeta(2550,0,0,10f,0.9f,Planeta.NOMES_PLANETAS.TIERRA,10f)
Planeta(3850,0,0,8f,0.7f,Planeta.NOMES_PLANETAS.MARTE,5f)
Planeta(4250,0,0,6f,0.5f,Planeta.NOMES_PLANETAS.JUPITER,110f)
Planeta(5650,0,0,10f,0.3f,Planeta.NOMES_PLANETAS.SATURNO,9f)
Planeta(6000,0,0,12f,0.2f,Planeta.NOMES_PLANETAS.URANO,39f)
Planeta(7200,0,0,8f,0.1f,Planeta.NOMES_PLANETAS.NEPTUNO,38f)
Planeta(8500,0,0,4f,0.05f,Planeta.NOMES_PLANETAS.PLUTON,2f)
```

Solución:

Arquivo Mundo3D.java

```
public class Mundo3D {

	/**
	 * Os planetas e o sol do sistema solar
	 */
	private Array<Planeta> planetas = new Array<Planeta>();

	public Mundo3D(){
		inicializarPlanetas();
	}


	/**
	 * Crea os planetas no array
	 */
	private void inicializarPlanetas(){
		planetas.add(new Planeta());	// O SOL

		planetas.add(new Planeta(1500,0,0,10f,3f,Planeta.NOMES_PLANETAS.MERCURIO,3f));
		planetas.add(new Planeta(2000,0,0,5f,1f,Planeta.NOMES_PLANETAS.VENUS,9f));
		planetas.add(new Planeta(2550,0,0,10f,0.9f,Planeta.NOMES_PLANETAS.TIERRA,10f));
		planetas.add(new Planeta(3850,0,0,8f,0.7f,Planeta.NOMES_PLANETAS.MARTE,5f));
		planetas.add(new Planeta(4250,0,0,6f,0.5f,Planeta.NOMES_PLANETAS.JUPITER,110f));
		planetas.add(new Planeta(5650,0,0,10f,0.3f,Planeta.NOMES_PLANETAS.SATURNO,9f));
		planetas.add(new Planeta(6000,0,0,12f,0.2f,Planeta.NOMES_PLANETAS.URANO,39f));
		planetas.add(new Planeta(7200,0,0,8f,0.1f,Planeta.NOMES_PLANETAS.NEPTUNO,38f));
		planetas.add(new Planeta(8500,0,0,4f,0.05f,Planeta.NOMES_PLANETAS.PLUTON,2f));
	}

	/**
	 * Devolve o array cos planetas e o sol do sistema solar
	 * @return the planetas
	 */
	public Array<Planeta> getPlanetas() {
		return planetas;
	}
}
```

Agora temos que crear un obxecto da clase Mundo3D dentro da clase PantallaSistemaSolar (a que deriva de Screen) e chamar os constructores das clase Renderer e Controler pasándolle dito obxecto.

Arquivo PantallaSistemaSolar.java

```
	private Mundo3D mundo3d;

	@Override
	public void show() {
		// TODO Auto-generated method stub
		mundo3d = new Mundo3D();
		xogorender = new SistemaSolarRenderer(mundo3d);
		controladorSistemaSolar3D = new ControladorSistemaSolar(mundo3d);

	}
```

Modificamos constructores de SistemaSolarRenderer e ControladorSistemaSolar e aproveitamos para instanciar as cámaras, o spritebatch e a clase GL11 na clase Renderer.

```
	private Mundo3D mundo3d;
	public SistemaSolarRenderer(Mundo3D mundo3d) {

		this.mundo3d = mundo3d;
		gl = Gdx.gl11;
		camara2d = new OrthographicCamera();
```

```
        camara3d = new PerspectiveCamera();

        sprite = new SpriteBatch();
}

private Mundo3D mundo3d;
public ControladorSistemaSolar(Mundo3D mundo3d){
        this.mundo3d = mundo3d;
}
```

Exercicio: Agora imos debuxar os planetas e o sol.

Pista: teremos que percorrer o array de planetas gardando a matriz de proxección para cada un deles.

Solución;

Arquivo SistemaSolarRenderer.java

```
public void render(){
        gl.glClear(GL11.GL_COLOR_BUFFER_BIT | GL11.GL_DEPTH_BUFFER_BIT);
        gl.glEnable(GL11.GL_DEPTH_TEST);
        gl.glEnable(GL11.GL_TEXTURE_2D);


        gl.glMatrixMode(GL11.GL_MODELVIEW);
        gl.glLoadIdentity();
        camara3d.position.set(400,10,50);
        camara3d.lookAt(0,0,0);

        camara3d.update();
        camara3d.apply(gl);

        amosarPlanetaseSol();


}

private void amosarPlanetaseSol(){
        for(Planeta planeta : mundo3d.getPlanetas()){
                gl.glPushMatrix();

                gl.glRotatef(planeta.getAngulo_traslacion(),0,1,0);
                gl.glTranslatef(planeta.getPosicion().x,
                                planeta.getPosicion().y,planeta.getPosicion().z);
                gl.glScalef(planeta.getEscala().x, planeta.getEscala().y, planeta.getEscala().z);
                gl.glRotatef(planeta.getAngulo_rotacion(),0,1,0);
                AssetsXogo.textura_planetas.get(planeta.getNome_planeta()).bind();
                AssetsXogo.mesh_planetas.render(GL11.GL_TRIANGLES);

                gl.glPopMatrix();
        }
```

Agora imos facer que os planetas se movan. O primeiro será crear o método update da clase Planeta para que modifique os seus ángulos de rotación e translación. Lembrar que a posición sempre é a mesma e se moven polo seu ángulo (lembrar exercicios dos cubos).

Arquivo Planeta.java

```
/**
 * Actualiza os ángulos dos planetas en función das súas velocidades
 * @param delta
 */
public void update(float delta){

        angulo_rotacion+=velocidaderotacion*delta;
        if (angulo_rotacion>=360) angulo_rotacion=0;

        angulo_traslacion+=velocidadetraslacion*delta;
        if (angulo_traslacion>=360) angulo_traslacion=0;
}
```

Chamamos a dito procedemento dende o controlador (para cada planeta do array).

Arquivo ControladorSistemaSolar.java

```
/**
 * Actualiza os ángulos dos planetas.
 * @param delta
 */
public void actualizarPlanetas(float delta){
        for(Planeta planeta : mundo3d.getPlanetas()){
                planeta.update(delta);
        }
}



public void update(float delta) {

        actualizarPlanetas(delta);
        processInput();
}
```

A NOSA NAVE ESPACIAL.

Agora imos cargar a nave espacial.
No proxecto podemos usar dous diferentes, o transbordador espacial da NASA (se pode descargar da web) ou unha nave que usou Mario Zechner (a persoa que desenrolou o motor libgdx) no seu libro 'Beginning Android 4 Games'.

Nota: os dous arquivos están no cartafol assets/texturas do proxecto base importado: ship.obj e transbordador.obj

Como sempre teremos que cargar o modelo e a textura no AssetManager. Imos darlle o nome de 'mesh_nave'.

Exercico: facer o anterior.

Solución:

Arquivo AssetManager.java

```
public static Texture textura_nave;
public static Mesh mesh_nave;

public static void cargarGraficos(){
        ……….
        imageFileHandle = Gdx.files.internal("texturas/ship.png");
        textura_nave = new Texture(imageFileHandle);
        model = loader.loadModel(Gdx.files.internal("texturas/ship.obj"));
        mesh_nave = model.meshes.get(0);
}

public static void liberarGraficos(){
        mesh_planetas.dispose();
        mesh_nave.dispose();
        for (Texture textura : textura_planetas.values()){
                textura.dispose();
        }
        textura_nave.dispose();
}
```

Agora temos que crear o modelo correspondente á nave espacial pero tendo en conta o seguinte.

Para mover a nave temos que rotar a nave e despois darlle unha velocidade.

A rotación pode ser en tres eixes pero nos a imos limitar a un: eixe Y. Desta forma a nave poderá rotar no sentido esquerda-dereita. O control arriba-abaixo subirá a nave no eixe Y.

NOTA: Para saber cal sería o movemento do obxecto rotando en cada un dos eixes podedes facer a proba de mover a cabeza en torno ó eixe correspondente.

> Se movemos a cabeza de arriba – abaixo estaremos rotándoa no eixe X e polo tanto a nave se move arriba-abaixo.
> Se movemos a cabeza de esquerda – dereita estaremos rotándoa no eixe Y e polo tanto a nave se move esquerda-dereita.
> Se movemos a cabeza de de ombreiro-ombreiro estaremos rotándoa no eixe Z e polo tanto a nave se inclina de esquerda-dereita.

A velocidade de rotación sempre será a mesma e polo tanto o que temos que facer é 'activar' o eixe na que se vai rotar en función do control da pantalla pulsado (no noso caso o eixe Y = Vector3(0,1,0))

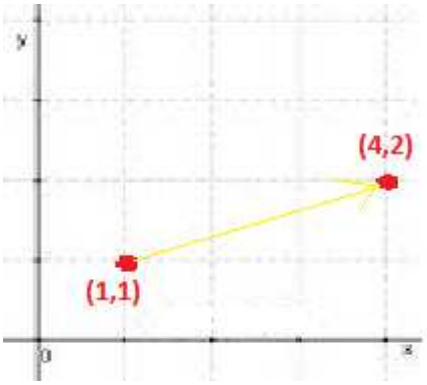
Cousas a ter claro:

- Imos rotar a posición da nave e esa rotación queda 'gardada', de tal forma que a idea é ter un ángulo de rotación que se vai incrementado a medida que nos movemos coa nave (premer control esquerda-dereita).
- Teremos que manexar un vector 'dirección' que vai ser o encargado de indicar por onde vai ir a nave cando acelere e este vector vaise rotar en función do ángulo de rotación (explicado a continuación).
- Cando debuxamos a nave teremos que rotar a matriz de opengl cos mesmos grados que o ángulo de rotación que temos gardado para que a nave rote gráficamente.

Aclaración: que é o vector de dirección ?

Imos velo en 2D sendo igual en 3D pero cun eixe máis.

Imaxinemos que estamos na coordenada (1,1) e queremos dirixirnos ata a coordenada (4,2).
O que teríamos que facer é calcular cal é o vector de dirección, é dicir, que números (x,y) sumados a (1,1) nos levan ata o (4,2).



O máis lóxico é restar o punto de destino menos o punto de orixe, desta forma:

Vector Dirección = Punto_Destino – Punto_Orixe = (4,2)-(1,1) = (3,1).

En libgdx (con 3 coordenadas usamos un Vector3 e con dous Vector2):

Punto Orixe = Posición da nave = Vector3 posicion
Punto Destino = Posición do sol (por exemplo) = Vector3 (0,0,0)

```
direccion = posicionsol.cpy().sub(posicion);
```

Sendo dirección e posicionsol vector3 xa instanciados.

Fixarse como usamos unha copia do vector de posición do sol xa que o método sub modifica o vector orixinal. Se queremos aumentar o rendemento (xa que estamos a facer new´s ó usar a función cpy) teríamos un vector temporal xa creado previamente (no constructor) e copiaríamos o contido

do vector que indica a posición do sol da forma:

```
temporal.set(posicionsol);
direccion = temporal.sub(posicion)
```

Agora poderíamos chegar ó punto de destino dun único salto, xa que na primeira iteración sumaríamos e xa chegaríamos. Como o que queremos é que pouco a pouco vaia chegando podemos facer uso do métodor nor() que normaliza o vector e fai que o seu valor sexa o vector unidade (para nos terá un valor <=1) pero os seus valores farán que ó sumalos á posición se vaia achegando ó punto de destino.

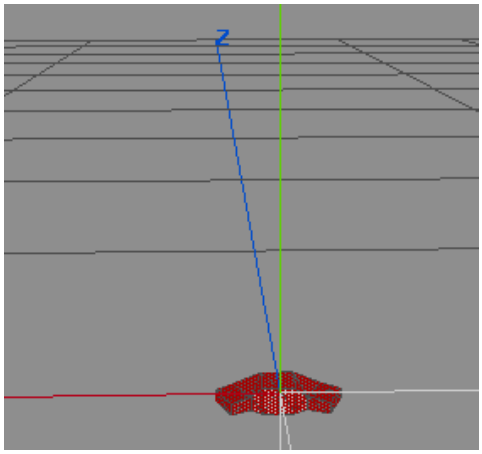
```
direccion.nor();
```

Agora para mover a nave ó punto de destino teríamos que facer no método update da nave:

```
posicion.add(direccion.cpy().scl(velocidade*delta));
```

Igual que no caso anterior podemos usar o mesmo vector temporal para asinarlle antes a dirección. O método scl multiplica o vector por un escalar (no exemplo velocidade*delta).

No noso caso non temos un punto de destino xa que a nave vai rotar.
Inicialmente o vector de dirección é (0,0,1) xa que é como está situada a nave no programa de deseño Wings3d:



A nave mira ó eixe Z positivo

Cando premamos o control de esquerda-dereita teremos que rotar (método rotate do vector dirección) có ángulo gardado á nave.

Polo tanto a información que necesitamos no modelo que representa a nave é:

- Unha posición (vector3).
- Un ángulo de rotación en cada un dos eixes (vector3).
- Unha velocidade de translación ata un valor máximo (constante)
- Unha velocidade de rotación (constante).
- Unha velocidade de ascenso-descenso (constante).
- Métodos get de todo.
- Unha propiedade de tipo enum (nome estado) cos métodos get e set e que indica os posibles estados da nave. Existe un estado IDLE que vainos servir para o movemento do campo de estrelas visto posteriormente e vai indicar que non está xirando a nave.

Arquivo Nave.java en paquete 'modelo'.

```
public class Nave {
	public static enum Keys {XIRARANDO_ESQUERDA, XIRARANDO_DEREITA,SUBINDO,BAIXANDO,
ACELERANDO,FRENANDO,PARADA,IDLE};
	/**
	 * Indica se a nave está rotando ou parada. Usado polo campo de estrelas de fondo
	 */
	private Keys estado;

	/**
	 * Posición da nave
	 */
	private Vector3 posicion;

	/**
	 * Angulo de rotación do vector de dirección
	 */
	private Vector3 angulo_rotacion;

	/**
	 * Velocidade da nave
	 */
	private float velocidade;

	/**
	 * Velocidade máxima á que se pode mover a nave.
	 */
	private final float MAX_VELOCIDADE=200f;
	/**
	 * Velocidade de rotación
	 */
	private final float VELOCIDADE_ROTACION=25f;

	/**
	 * Dirección da nave
	 */
	private Vector3 direccion;

	/**
	 * Máxima velocidade de asncensión-descenso
	 */
	private final float VELOCIDADE_ASCENSION=100f;

	/**
	 * Vector para non usar cpy()
	 */
	private Vector3 vectortemp;


	public Nave(){
		posicion = new Vector3(2590,0,0);
		angulo_rotacion = new Vector3(0,0,0);
		velocidade=0f;
		estado = Keys.PARADA;
		direccion = new Vector3(0,0,1);
		vectortemp = new Vector3();
	}


	public Vector3 getPosicion() {
		return posicion;
	}
	public Vector3 getDireccion() {
		return direccion;
	}


	public Keys getEstado() {
		return estado;
	}
```

```
	public void setEstado(Keys estado) {
		this.estado = estado;
	}


	public Vector3 getAngulo_rotacion() {
		return angulo_rotacion;
	}
	public float getVelocidade() {
		return velocidade;
	}


}
```

Fixarse que poño como velocidade máxima 200 unidades. Por qué ?
Se mirades o código dos planetas podedes ver como o máis afastado está colocado na posición 8500:

```
		planetas.add(new Planeta(8500,0,0,4f,0.05f,Planeta.NOMES_PLANETAS.PLUTON,2f));
```

Isto quere dicir que o tamaño do sistema solar sería 8500x2 = 17.000 unidades.
A unha velocidade de 200 unidades por segundo tardaría 17000/200=85 segundos=1' 25" en chegar de extremo a extremo.

Imos engadir á nave ó noso mundo.

Arquivo Mundo3D.java

```
	private Nave nave;

	public Mundo3D(){
		inicializarPlanetas();
		nave = new Nave();
	}

	public Nave getNave() {
		return nave;
	}
```

E imos debuxar a nave.

Temos que ter unha referencia á nave dende a clase SistemaSolarRenderer e aproveitamos para definir un vector de uso temporal.

Arquivo SistemaSolarRenderer.java

```
	private Nave nave;
	private Vector3 vector3temp;    // Vector temporal

	public SistemaSolarRenderer(Mundo3D mundo3d) {

		this.mundo3d = mundo3d;
		nave = mundo3d.getNave();

		gl = Gdx.gl11;
		camara2d = new OrthographicCamera();
		camara3d = new PerspectiveCamera();

		vector3temp = new Vector3();

	}
```

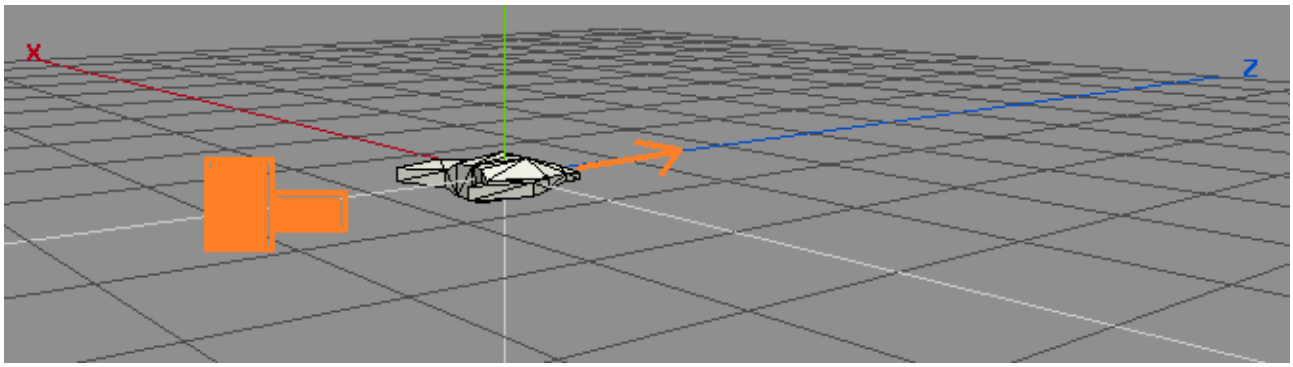
Agora debemos debuxar á nave.
Lembrade que debemos de rotar a matriz opengl có ángulo de rotación correspondente.

Arquivo SistemaSolarRenderer.java

```
private void amosarNave(){
        gl.glPushMatrix();

        gl.glTranslatef(nave.getPosicion().x, nave.getPosicion().y, nave.getPosicion().z);
        gl.glRotatef(nave.getAngulo_rotacion().y, 0, 1, 0);

        AssetsXogo.textura_nave.bind();
        AssetsXogo.mesh_nave.render(GL11.GL_TRIANGLES);

        gl.glPopMatrix();

}
```

Agora é necesario chamar ó método amosarNave dende o método render, pero tendo en conta que debemos de situar a cámara xusto detrás da nave.

Para facelo podemos utilizar o vector de dirección que vai indicar cara onde vai a nave.
A cámara debería estar na posición da nave RESTANDO o vector de dirección.



É dicir:

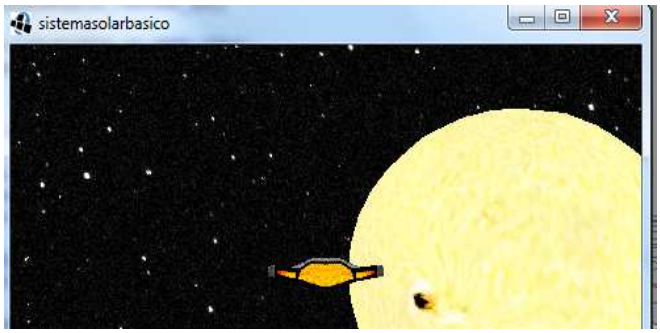
```
vector3temp.set(nave.getPosicion());
camara3d.position.set(vector3temp.sub(nave.getDireccion().cpy().scl(3)));
```

O multiplicamos por 3 xa que o vector de dirección sempre é menor ou igual a 1 e quedaría moi cerca da nave.

Pero isto non é todo, xa que a cámara non só ten unha posición se non que tamén ten cara onde mira, é dicir, unha dirección.
A dirección da cámara será a posición da nave (punto final) menos a posición da cámara (punto inicial):

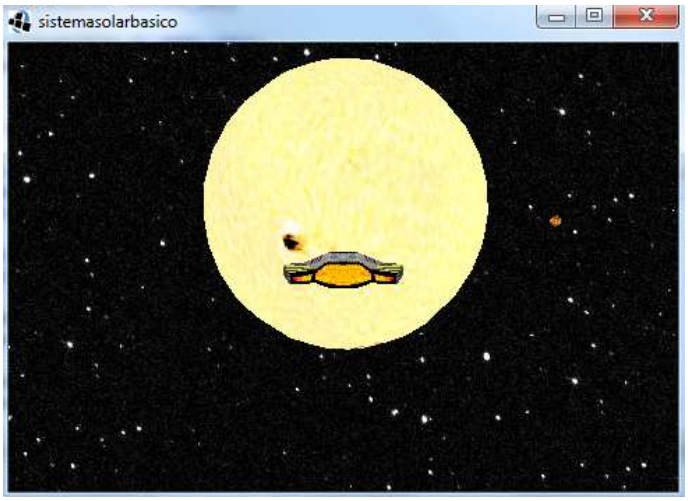
```
vector3temp.set(nave.getPosicion());
camara3d.direction.set(vector3temp.sub(camara3d.position));
```

Pero se probamos a ver como queda:



Vemos que estamos xusto detrás. Imos facer que a cámara estea un pouco por enriba da nave (sumarlle no eixe y):

```
camara3d.position.set(vector3temp.sub(nave.getDireccion().cpy().scl(3)).add(0,0.5f,0));
```



Se o poñemos todo xunto:

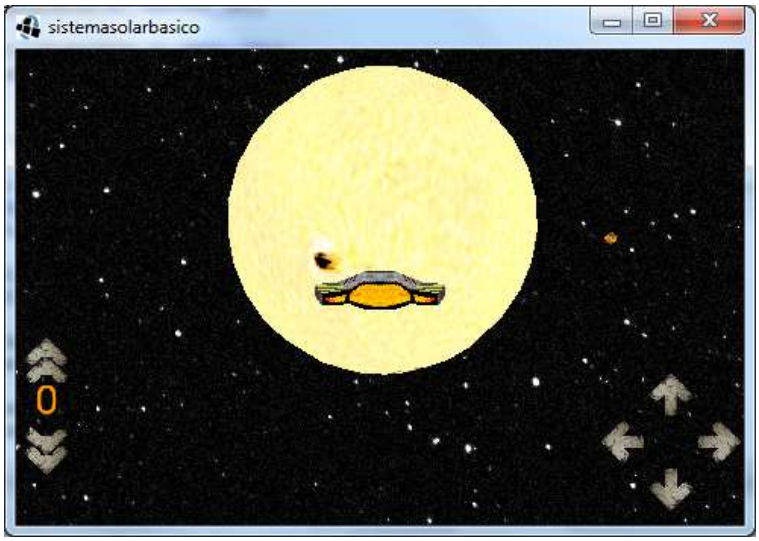
```
public void render(){
        gl.glClear(GL11.GL_COLOR_BUFFER_BIT | GL11.GL_DEPTH_BUFFER_BIT);


        gl.glEnable(GL11.GL_DEPTH_TEST);
        gl.glEnable(GL11.GL_TEXTURE_2D);

        gl.glMatrixMode(GL11.GL_MODELVIEW);
        gl.glLoadIdentity();


        vector3temp.set(nave.getPosicion());

        camara3d.position.set(vector3temp.sub(nave.getDireccion().cpy().scl(3)).add(0,0.5f,0));
        vector3temp.set(nave.getPosicion());
         camara3d.direction.set(vector3temp.sub(camara3d.position));

        camara3d.update();
        camara3d.apply(gl);

        amosarPlanetaseSol();
        amosarNave();

        gl.glDisable(GL11.GL_DEPTH_TEST);
        gl.glDisable(GL11.GL_TEXTURE_2D);
}
```

Probar o que se ve....

OS CONTROIS DA NAVE:

Agora falta controlar o movemento da nave.
Imos colocar catro gráficos de controis na pantalla e dous de velocidade (coa cámara orthográfica) .
Os gráficos están no cartafol 'controis' do proxecto base copiado.

Imos agrupalos na parte dereita da pantalla os de movemento e na parte esquerda os de acelerar e frear. Os cargaremos nun enummap de texturas e os referenciaremos cunha variable enum.



Como para controlar cando un control foi premido, é necesario gardar a posición e o tamaño dos mesmos, creamos no paquete Modelo unha clase de nome Controis. Imos utilizar a clase Rectangle para gardar as coordenadas de cada control.

Exercicio: crear dita clase.

A posición dos controis van depender do tamaño da cámara 2D usada na clase SistemaSolarRenderer, xa que esta varía para manter o aspect ratio (método resize).

Arquivo Controis.java

```
public class Controis {

	public static enum CONTROIS {ARRIBA,ABAIXO,ESQUERDA,DEREITA,ACELERAR,FRENAR};
	public final static float TAMAÑO = 40;

	public static EnumMap <CONTROIS,Rectangle> tamaño_controis = new EnumMap
<CONTROIS,Rectangle>(CONTROIS.class);

	public static void asinarTamañoControis(){


		OrthographicCamera cam2d=SistemaSolarRenderer.getCamara2d();

		tamaño_controis.put(CONTROIS.ABAIXO,new Rectangle(cam2d.viewportWidth-TAMAÑO*2,10,TAMAÑO,TAMAÑO));
		tamaño_controis.put(CONTROIS.ARRIBA,new Rectangle(cam2d.viewportWidth-
TAMAÑO*2,TAMAÑO*2+10,TAMAÑO,TAMAÑO));
		tamaño_controis.put(CONTROIS.DEREITA,new Rectangle(cam2d.viewportWidth-TAMAÑO,TAMAÑO+10,TAMAÑO,TAMAÑO));
		tamaño_controis.put(CONTROIS.ESQUERDA,new Rectangle(cam2d.viewportWidth-
```

```
TAMAÑO*3,TAMAÑO+10,TAMAÑO,TAMAÑO));

                tamaño_controis.put(CONTROIS.ACELERAR,new Rectangle(5,TAMAÑO*3,TAMAÑO,TAMAÑO));
                tamaño_controis.put(CONTROIS.FRENAR,new Rectangle(5,TAMAÑO,TAMAÑO,TAMAÑO));

        };

}
```

Cargamos os controis nas texturas correspondentes.

NOTA: Neste exercicio non estamos utilizando os MapAtlas pero sería moito mellor o seu uso. Desta forma poderíamos cargar un só dos controis e usar unha TextureRegion para debuxalo (chamando o método draw da clase SpriteBatch), podendo rotala e así debuxar o resto de controis utilizando o mesmo gráfico.

Arquivo AssetsXogo.java

```
        public static EnumMap<Controis.CONTROIS, Texture>texturas_controis = new EnumMap<Controis.CONTROIS,
Texture>(Controis.CONTROIS.class);      // Pon o ENUM como key

        public static void cargarGraficos(){
                ……….
                // CARGAR CONTROIS
                imageFileHandle = Gdx.files.internal("texturas/controis/arriba.png");
                texturas_controis.put(Controis.CONTROIS.ARRIBA, new Texture(imageFileHandle));
                imageFileHandle = Gdx.files.internal("texturas/controis/abaixo.png");
                texturas_controis.put(Controis.CONTROIS.ABAIXO, new Texture(imageFileHandle));
                imageFileHandle = Gdx.files.internal("texturas/controis/esquerda.png");
                texturas_controis.put(Controis.CONTROIS.ESQUERDA, new Texture(imageFileHandle));
                imageFileHandle = Gdx.files.internal("texturas/controis/dereita.png");
                texturas_controis.put(Controis.CONTROIS.DEREITA, new Texture(imageFileHandle));
                imageFileHandle = Gdx.files.internal("texturas/controis/acelerar.png");
                texturas_controis.put(Controis.CONTROIS.ACELERAR, new Texture(imageFileHandle));
                imageFileHandle = Gdx.files.internal("texturas/controis/frenar.png");
                texturas_controis.put(Controis.CONTROIS.FRENAR, new Texture(imageFileHandle));
        }

        public static void liberarGraficos(){
                mesh_planetas.dispose();
                mesh_nave.dispose();
                for (Texture textura : textura_planetas.values()){
                        textura.dispose();
                }
                textura_nave.dispose();

                for(Texture control : texturas_controis.values()){
                        control.dispose();
                }
        }
```

Lembrade que imos facer uso dun SpriteBatch.

Chamaremos ó método que asina os tamaños dende a clase SistemaSolarRenderer no método resize xa que é aí onde sabemos o tamaño da cámara (sempre despois de chamar a update da cámara).

Arquivo SistemaSolarRenderer.java

```
public void resize(int width, int height) {
        ..................
        CAMARA2D_ANCHO=400*aspectRatio;
        CAMARA2D_ALTO=400;

        camara2d.setToOrtho(false, CAMARA2D_ANCHO, CAMARA2D_ALTO);
        camara2d.update();
        Controis.asinarTamañoControis();
        ................
}
```

Agora só falta debuxar os controis na mesma clase anterior.
Como son debuxos 2D podemos deshabilitar o GL_DEPTH_TEST e o GL_TEXTURE_2D no método render.

Exercicio: facer o anterior

Solución:

Arquivo SistemaSolarRenderer.java

```
/**
 * Dibuxa os controis da nave.
 */
private void dibuxarControis(){

        Rectangle control = Controis.tamaño_controis.get(Controis.CONTROIS.ABAIXO);
        sprite.draw(AssetsXogo.texturas_controis.get(Controis.CONTROIS.ABAIXO),control.x,control.y,control.width,control.height);
        control = Controis.tamaño_controis.get(Controis.CONTROIS.ARRIBA);
        sprite.draw(AssetsXogo.texturas_controis.get(Controis.CONTROIS.ARRIBA),control.x,control.y,control.width,control.height);
        control = Controis.tamaño_controis.get(Controis.CONTROIS.ESQUERDA);
        sprite.draw(AssetsXogo.texturas_controis.get(Controis.CONTROIS.ESQUERDA),control.x,control.y,control.width,control.height);
        control = Controis.tamaño_controis.get(Controis.CONTROIS.DEREITA);
        sprite.draw(AssetsXogo.texturas_controis.get(Controis.CONTROIS.DEREITA),control.x,control.y,control.width,control.height);

        control = Controis.tamaño_controis.get(Controis.CONTROIS.ACELERAR);
        sprite.draw(AssetsXogo.texturas_controis.get(Controis.CONTROIS.ACELERAR),control.x,control.y,control.width,control.height);
        control = Controis.tamaño_controis.get(Controis.CONTROIS.FRENAR);
        sprite.draw(AssetsXogo.texturas_controis.get(Controis.CONTROIS.FRENAR),control.x,control.y,control.width,control.height);

}

public void render(){
        ..................

        amosarNave();

        gl.glDisable(GL11.GL_DEPTH_TEST);
        gl.glDisable(GL11.GL_TEXTURE_2D);


        // PARTE 2D
        sprite.begin();
        dibuxarControis();
        sprite.end();
}
```

Agora imos facer que cando se preme o acelerador a nave avance e o revés co freo. A velocidade máxima estará indicado pola constante MAX_VELOCIDADE da clase Nave.
Os métodos se chaman aumentarVelocidade e disminuirVelocidade.

Exercicio: facer o anterior.

Solución:

Debemos de crear dous métodos na clase Nave para aumentar e diminuír a velocidade, controlando que non pase do máximo e que non sexa menor que a velocidade máxima en negativo (pode ir para atrás). Creamos o método 'update(float delta)' para que actualice a posición en función da velocidade.

Arquivo Nave.java

```
	/**
	 * Aumenta a velocidade da nave ata a constante MAX_VELOCIDADE
	 */
	public void aumentarVelocidade(){
		if (velocidade<MAX_VELOCIDADE) velocidade+=0.1;
		estado = Keys.ACELERANDO;
	}
	/**
	 * Disminue a velocidade da nave.
	 */
	public void disminuirVelocidade(){
		if(velocidade>-MAX_VELOCIDADE) velocidade-=0.1;
		estado = Keys.FRENANDO;
	}

	/**
	 * Actualiza a posición da nave.
	 * @param delta
	 */
	public void update(float delta){

		vectortemp.set(direccion);
		posicion.add(vectortemp.scl(velocidade*delta));
	}
```

Agora debemos capturar o evento de 'pulsar' sobre o control. Iso o faremos coa clase Intersector facendo o unproyect das coordenadas reais á nosa cámara orthográfica (como fixemos no proxecto do pescador).

Arquivo PantallaSistemaSolar.java

```
	@Override
	public boolean touchDown(int screenX, int screenY, int pointer, int button) {
		// TODO Auto-generated method stub
		Vector3 coord = new Vector3(screenX,screenY,0);
		SistemaSolarRenderer.getCamara2d().unproject(coord);	// Coord pasa a estar no sistema de coordenadas da cámara
		Circle pulsa = new Circle(coord.x,coord.y,Controis.TAMAÑO/4);

		if (Intersector.overlaps(pulsa,Controis.tamaño_controis.get(Controis.CONTROIS.ACELERAR))){

	controladorSistemaSolar3D.presionaTecla(ControladorSistemaSolar.Keys.AUMENTAR_VELOCIDADE);
		}
		if (Intersector.overlaps(pulsa,Controis.tamaño_controis.get(Controis.CONTROIS.FRENAR))){

	controladorSistemaSolar3D.presionaTecla(ControladorSistemaSolar.Keys.DISMINUIR_VELOCIDADE);
		}


		return false;
	}

	@Override
	public boolean touchUp(int screenX, int screenY, int pointer, int button) {
		// TODO Auto-generated method stub
```

```
                Vector3 coord = new Vector3(screenX,screenY,0);
                SistemaSolarRenderer.getCamara2d().unproject(coord);     // Coord pasa a estar no sistema de
coordenadas da cámara
                Circle pulsa = new Circle(coord.x,coord.y,Controis.TAMAÑO/4);

                if (Intersector.overlaps(pulsa,Controis.tamaño_controis.get(Controis.CONTROIS.ACELERAR))){

        controladorSistemaSolar3D.liberaTecla(ControladorSistemaSolar.Keys.AUMENTAR_VELOCIDADE);
                }
                if (Intersector.overlaps(pulsa,Controis.tamaño_controis.get(Controis.CONTROIS.FRENAR))){

        controladorSistemaSolar3D.liberaTecla(ControladorSistemaSolar.Keys.DISMINUIR_VELOCIDADE);
                }

                return false;
        }
```

Agora debemos modificar a clase controladora para que actualice a posición da nave e modifique á velocidade.

Arquivo ControladorSistemaSolar.java

```
        /**
         * Actualiza a posición da nave
         * @param delta
         */
        public void actualizarNave(float delta){
                mundo3d.getNave().update(delta);
        }


        public void update(float delta) {

                actualizarPlanetas(delta);
                actualizarNave(delta);
                processInput();
        }

        private void processInput(){
                if (keys.get(Keys.AUMENTAR_VELOCIDADE)){
                        mundo3d.getNave().aumentarVelocidade();
                }else if (keys.get(Keys.DISMINUIR_VELOCIDADE)){
                        mundo3d.getNave().disminuirVelocidade();
                }

        }
```

Como mellora, imos imprimir a velocidade da nave e imos facer que se se preme sobre o número a velocidade quede a cero. Para a nos a velocidade é outro control máis.

Xa temos no proxecto base as fontes de texto no cartafol 'fontes'. Modificamos o assetmanager para cargala.

Arquivo AssetsXogo.java

```
        /**
         * Fonte a usar polo xogo na pantalla de xogo
         */
        public static BitmapFont bitmapfont;

        public static void cargarGraficos(){
                ..................
```

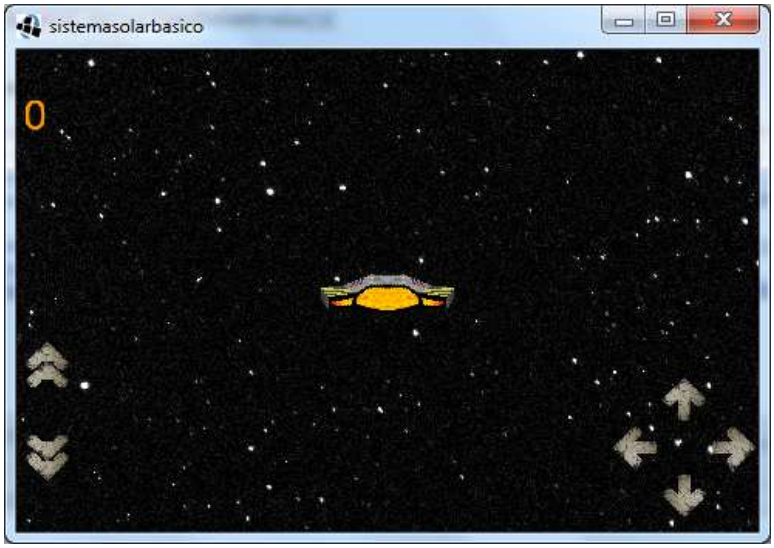
```
                // AS FONTES
                imageFileHandle = Gdx.files.internal("fontes/fonte.fnt");
                bitmapfont = new BitmapFont(imageFileHandle, false);
        }

        public static void liberarGraficos(){
                ................

                bitmapfont.dispose();
        }
```

Arquivo Controis.java

```
        public static enum CONTROIS {ARRIBA,ABAIXO,ESQUERDA,DEREITA,ACELERAR,FRENAR,PARAR};
        public static void asinarTamañoControis(){
                ..............
                tamaño_controis.put(CONTROIS.PARAR,new Rectangle(5,cam2d.viewportHeight-
TAMAÑO,TAMAÑO,TAMAÑO)); // Cando se dibuxa hai que aumentar en TAMAÑO o y xa que o sistema de coordenadas
empeza arriba

        };
```

O que está a dicir é que nas letras (bitmapfont) o punto (0,0) é a parte superior esquerda e se debuxa cara abaixo. Polo tanto para controlar a pulsación está ben.

Se debuxamos nas coordenadas indicadas polo control de parar, o número quedaría demasiado abaixo:



Polo tanto cando debuxamos temos que sumarlle o tamaño do control para que quede na parte superior.

Arquivo SistemaSolarRenderer.java

```
        private void dibuxarControis(){
                ..............
                // Velocidade
                control = Controis.tamaño_controis.get(Controis.CONTROIS.PARAR);
                AssetsXogo.bitmapfont.draw(sprite, String.valueOf((int)nave.getVelocidade()), control.x,
control.y+Controis.TAMAÑO);

        }
```

Arquivo Nave.java

```
/**
 * Para a nave
 */
public void parar(){
        velocidade=0;
        estado = Keys.PARADA;
}
```

Arquivo PantallaSistemaSolar.java

```
@Override
public boolean touchDown(int screenX, int screenY, int pointer, int button) {
        ..............
        if (Intersector.overlaps(pulsa,Controis.tamaño_controis.get(Controis.CONTROIS.PARAR))){
                controladorSistemaSolar3D.presionaTecla(ControladorSistemaSolar.Keys.PARAR);
        }


@Override
public boolean touchUp(int screenX, int screenY, int pointer, int button) {
        ..............
        if (Intersector.overlaps(pulsa,Controis.tamaño_controis.get(Controis.CONTROIS.PARAR))){
                controladorSistemaSolar3D.liberaTecla(ControladorSistemaSolar.Keys.PARAR);
        }
```

Arquivo ControladorSistemaSolar.java

```
private void processInput(){
        ..............
        if (keys.get(Keys.PARAR)){
                mundo3d.getNave().parar();
        }
}
```

O movemento de rotación da nave.

Creamos na clase Nave un procedemento que vai rotar a posición da nave. Levará como parámetro o estado no que se atopa a nave (propiedade estado). En función do estado rotaremos (esquerda-dereita) ou moveremos (arriba-abaixo) a nave.
Rotar a nave quere dicir que teremos que rotar o vector de dirección có ángulo que imos ter que gardar.

Arquivo Nave.java

```
/**
 * Rota a nave
 */
public void rotar(Keys xiro){

	estado=xiro;
	if (Keys.XIRARANDO_ESQUERDA==xiro) {
		angulo_rotacion.y += +VELOCIDADE_ROTACION*Gdx.graphics.getDeltaTime();
	}
	if (Keys.XIRARANDO_DEREITA==xiro) {
		angulo_rotacion.y += -VELOCIDADE_ROTACION*Gdx.graphics.getDeltaTime();
	}
	if (Keys.SUBINDO==xiro) {
		posicion.y+=VELOCIDADE_ASCENSION*Gdx.graphics.getDeltaTime();
	}
	if (Keys.BAIXANDO==xiro) {
		posicion.y-=VELOCIDADE_ASCENSION*Gdx.graphics.getDeltaTime();
	}

	direccion.set(0,0,1);
	direccion.rotate(angulo_rotacion.y, 0,1,0);
	direccion.nor();
}
```

Exercicio: modificar as clases PantallaSistemaSolar e ControladorSistemaSolar para que rote a nave (informen á clase controlador que ten que rotar-mover a nave).

Solución:

Arquivo PantallaSistemaSolar.java

```
@Override
public boolean touchDown(int screenX, int screenY, int pointer, int button) {
	.................
	if (Intersector.overlaps(pulsa,Controis.tamaño_controis.get(Controis.CONTROIS.ESQUERDA))){
		controladorSistemaSolar3D.presionaTecla(ControladorSistemaSolar.Keys.XIRAR_ESQUERDA);
	}
	if (Intersector.overlaps(pulsa,Controis.tamaño_controis.get(Controis.CONTROIS.DEREITA))){
		controladorSistemaSolar3D.presionaTecla(ControladorSistemaSolar.Keys.XIRAR_DEREITA);
	}
	if (Intersector.overlaps(pulsa,Controis.tamaño_controis.get(Controis.CONTROIS.ARRIBA))){
		controladorSistemaSolar3D.presionaTecla(ControladorSistemaSolar.Keys.XIRAR_ARRIBA);
	}
	if (Intersector.overlaps(pulsa,Controis.tamaño_controis.get(Controis.CONTROIS.ABAIXO))){
		controladorSistemaSolar3D.presionaTecla(ControladorSistemaSolar.Keys.XIRAR_ABAIXO);
	}

	return false;
}
@Override
public boolean touchUp(int screenX, int screenY, int pointer, int button) {
	// TODO Auto-generated method stub
	.....................
	if (Intersector.overlaps(pulsa,Controis.tamaño_controis.get(Controis.CONTROIS.ESQUERDA))){
```

```
                controladorSistemaSolar3D.liberaTecla(ControladorSistemaSolar.Keys.XIRAR_ESQUERDA);
            }
            if (Intersector.overlaps(pulsa,Controis.tamaño_controis.get(Controis.CONTROIS.DEREITA))){
                controladorSistemaSolar3D.liberaTecla(ControladorSistemaSolar.Keys.XIRAR_DEREITA);
            }
            if (Intersector.overlaps(pulsa,Controis.tamaño_controis.get(Controis.CONTROIS.ARRIBA))){
                controladorSistemaSolar3D.liberaTecla(ControladorSistemaSolar.Keys.XIRAR_ARRIBA);
            }
            if (Intersector.overlaps(pulsa,Controis.tamaño_controis.get(Controis.CONTROIS.ABAIXO))){
                controladorSistemaSolar3D.liberaTecla(ControladorSistemaSolar.Keys.XIRAR_ABAIXO);
            }

            return false;
        }
```

Arquivo ControladorSistemaSolar.java

```
        private void processInput(){
            if (keys.get(Keys.AUMENTAR_VELOCIDADE)){
                nave.aumentarVelocidade();
            }else if (keys.get(Keys.DISMINUIR_VELOCIDADE)){
                nave.disminuirVelocidade();
            }

            if (keys.get(Keys.XIRAR_ABAIXO)){
                nave.rotar(Nave.Keys.BAIXANDO);
            } else if (keys.get(Keys.XIRAR_ARRIBA)){
                nave.rotar(Nave.Keys.SUBINDO);
            } else if (keys.get(Keys.XIRAR_DEREITA)){
                nave.rotar(Nave.Keys.XIRARANDO_DEREITA);
            } else if (keys.get(Keys.XIRAR_ESQUERDA)){
                nave.rotar(Nave.Keys.XIRARANDO_ESQUERDA);
            }

            if ((!keys.get(Keys.XIRAR_ABAIXO)) && (!keys.get(Keys.XIRAR_ARRIBA)) && (!
keys.get(Keys.XIRAR_DEREITA)) && (!keys.get(Keys.XIRAR_ESQUERDA)))
                nave.setEstado(Nave.Keys.IDLE);

            if (keys.get(Keys.PARAR)){
                mundo3d.getNave().parar();
            }

        }
```

Nota: necesitamos o estado IDLE para que cando deixamos de premer o control de rotar a nave deixe de rotar.

O fondo de estrelas.

Para dar máis realismo ó xogo imos poñer un fondo de estrelas que se mova, tanto cando a nave estea en movemento como cando estea rotando, ascendendo ou descendendo.

Existen varias aproximacións para facer isto.
A idea é ter unha textura de fondo que se repita de forma indefinida.
No repositorio do framework podedes ver unha forma de facelo:
https://github.com/libgdx/libgdx/blob/master/tests/gdx-tests/src/com/badlogic/gdx/tests/ParallaxTest.java

Nos imos facelo dunha forma un pouco máis sinxela.
http://code.google.com/p/libgdx-users/wiki/ScrollingTexture

No cartafol /texturas xa temos o gráfico 'estrelas.jpg' que vainos servir de fondo.

O seguinte será cargar dito fondo na clase AssetsXogo, PERO imos indicarlle que a textura debe repetirse dentro do seu tamaño u-v (lembrar que nas texturas o x-y se denomina u-v e se move entre valores 0-1).
Isto o facemos así:

```
textura_fondoestrelas.setWrap(Texture.TextureWrap.Repeat, Texture.TextureWrap.Repeat);
```

Arquivo AssetsXogo.java

```
public static Texture fondoestrelas;

public static void cargarGraficos(){
        ..............
        // Fondo de estrelas.
        imageFileHandle = Gdx.files.internal("texturas/estrelas.jpg");
        textura_fondoestrelas = new Texture(imageFileHandle);
        textura_fondoestrelas.setWrap(Texture.TextureWrap.Repeat, Texture.TextureWrap.Repeat);
}
public static void liberarGraficos(){
        ............
        textura_fondoestrelas.dispose();
}
```

Agora debemos de 'mover' a textura de forma indefinida entre as coordenadas u-v.
Isto o temos que facer no método render da clase SistemaSolarRenderer usando un obxecto da clase Sprite a cal garda a textura e tamén a posición.

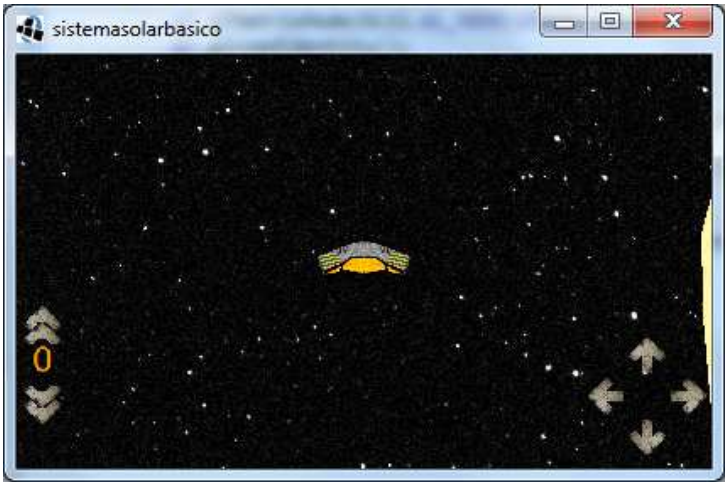
É importante ver a orde de renderizado. O fondo de estrelas ten que debuxarse en primeiro lugar xa que se non borraría todo.

Arquivo SistemaSolarRenderer.java

```
/**
 * Usado para mover as estrelas de fondo
 */
private Vector2 scrollTimer=new Vector2(0,0);
/**
 * Garda o campo de estrelas
 */
private Sprite campoestrelas;

public SistemaSolarRenderer(Mundo3D mundo3d) {
        ..................
        sprite = new SpriteBatch();
        campoestrelas = new Sprite(AssetsXogo.textura_fondoestrelas);

}
/**
 * Dibuxa o fondo de estrelas
 */
private void dibuxarFondo(){
        sprite.disableBlending();       // Para aforrar recursos. Non ten transparencias.

        scrollTimer.add(0.00001f,0.00001f);

        if(scrollTimer.x>1.0f)
                scrollTimer.x = 0.0f;
        if(scrollTimer.y>1.0f)
                scrollTimer.y = 0.0f;

        campoestrelas.setU(scrollTimer.x);
        campoestrelas.setU2(scrollTimer.x+1);
        campoestrelas.setV(scrollTimer.y);
        campoestrelas.setV2(scrollTimer.y+1);

        campoestrelas.draw(sprite);

        sprite.enableBlending();

}

public void render(){
        gl.glClear(GL11.GL_COLOR_BUFFER_BIT | GL11.GL_DEPTH_BUFFER_BIT);

          // DIBUXA FONDO DE ESTRELAS
          sprite.begin();
          dibuxarFondo();
          sprite.end();


        gl.glEnable(GL11.GL_DEPTH_TEST);
        gl.glEnable(GL11.GL_TEXTURE_2D);

        gl.glMatrixMode(GL11.GL_MODELVIEW);
        gl.glLoadIdentity();

        vector3temp.set(nave.getPosicion());
                camara3d.position.set(vector3temp.sub(0,-0.2f,0.5f));
                camara3d.position.set(nave.getPosicion().cpy().sub(0,-0.2f,0.5f));
                camara3d.lookAt(nave.getPosicion());

        camara3d.update();
        camara3d.apply(gl);

                amosarPlanetaseSol();
                amosarNave();

        gl.glDisable(GL11.GL_DEPTH_TEST);
        gl.glDisable(GL11.GL_TEXTURE_2D);
```

```
        // PARTE 2D
        sprite.begin();
        dibuxarControis();
        sprite.end();

    }
```

Se probades podedes ver como se moven as estrelas.



Pero para que quede realista teremos que facer que as estrelas se movan máis rápido e no sentido contrario ó da nave cando esta se mova.
Polo tanto necesitamos que dende a clase SistemaSolarRenderer ter acceso ó estado da nave para saber se se está movendo ou non.

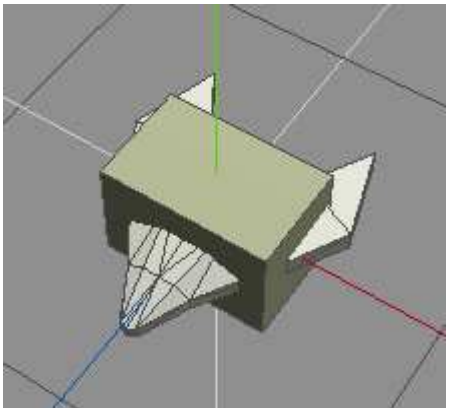
Arquivo SistemaSolarRenderer.java

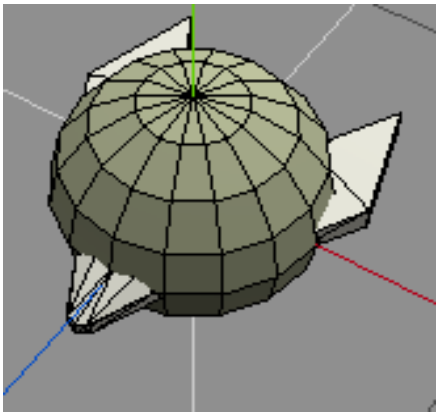
```
    /**
     * Dibuxa o fondo de estrelas
     */
    private void dibuxarFondo(){
        sprite.disableBlending();       // Para aforrar recursos. Non ten transparencias.

        if (nave.getEstado()==Nave.Keys.XIRARANDO_DEREITA)      // XIRO DEREITA
            scrollTimer.x += 0.05f*Gdx.graphics.getDeltaTime();
        if (nave.getEstado()==Nave.Keys.XIRARANDO_ESQUERDA)     // XIRO ESQUERDA
                scrollTimer.x -= 0.05f*Gdx.graphics.getDeltaTime();
        if (nave.getEstado()==Nave.Keys.SUBINDO)//  ARRIBA
            scrollTimer.y -= 0.05f*Gdx.graphics.getDeltaTime();
        if (nave.getEstado()==Nave.Keys.BAIXANDO)       //  ABAIXO
            scrollTimer.y += 0.05f*Gdx.graphics.getDeltaTime();

        scrollTimer.add(0.00001f,0.00001f);

        if(scrollTimer.x>1.0f)
            scrollTimer.x = 0.0f;
        if(scrollTimer.y>1.0f)
            scrollTimer.y = 0.0f;

        campoestrelas.setU(scrollTimer.x);
        campoestrelas.setU2(scrollTimer.x+1);
        campoestrelas.setV(scrollTimer.y);
        campoestrelas.setV2(scrollTimer.y+1);

        campoestrelas.draw(sprite);

        sprite.enableBlending();
    }
```

Controlando os choques.

En 3D temos dúas formas de controlar cando unha figura 'choca' con outra.
A idea é envolver a figura cun prisma-cubo ou esfera e comprobar se estas chocan.

O cubo-prisma denomínase BoundingBox e pódese obter a partires dun obxecto da clase Mesh ou ben crear nos un obxecto da clase BoundingBox e definir o seu tamaño e posición.





A esfera defínese cun centro (nunha posición no espazo 3D) e un radio.

O bounding box defínese con dous vectores (min-max) que conforman o volume.

Máis información en: http://code.google.com/p/libgdx/wiki/Geometry

Temos outras opcións para detectar colisións en outras situacións como poden ser o uso de raios (parten dun punto e teñen unha dirección).

As colisións as temos que controlar usando a clase Intersector (no caso dos bounding box) ou a clase Sphere que ten un método que indica cando se 'choca' con outra esfera (método overLaps).

No noso caso imos facer uso da clase Sphere.
Polo tanto imos definir unha esfera para cada un dos corpos que queiramos controlar o seu choque, é dicir, a nave e os planetas.

O tamaño da esfera non vai variar, pero sí a posición que terá que actualizarse no método update correspondente.
Polo tanto o tamaño o imos definir no constructor da clase. O tamaño ten que ver coa escala que teñan os obxecto.
Por exemplo, a nave ten un tamaño dunha unidade no programa wings3d e polo tanto a esfera terá un valor de 0.5 no seu radio.

Os planetas tamén parten do valor dous unidades e polo tanto teñen un radio de 1, pero o escalalos o seu tamaño se modifica.
Así, o tamaño de Mercurio é:

```
planetas.add(new Planeta(1500,0,0,10f,3f,Planeta.NOMES_PLANETAS.MERCURIO,3f));
```

Isto quere dicir que o seu tamaño é 3 veces (escala) o modelo usado (que ten tamaño 2):

3 x 2 = 6 unidades de diámetro.

De radio será 3 e polo tanto o radio se corresponde coa escala do planeta.

Imos comezar coa nave.

Arquivo Nave.java

```
/**
 * Para controlar os choques
 */
private Sphere esfera;
/**
 * Tamaño da nave.
 */
private final static float ESCALA_NAVE = 0.5f;

public Nave(){
        posicion = new Vector3(2590,0,0);
        angulo_rotacion = new Vector3(0,0,0);
        velocidade=0f;
        estado = Keys.IDLE;
        direccion = new Vector3(0,0,1);
        vectortemp = new Vector3();

        esfera = new Sphere(posicion,ESCALA_NAVE);
}

public Sphere getEsfera() {
        return esfera;
}

public void update(float delta){


        vectortemp.set(direccion);
        posicion.add(vectortemp.scl(velocidade*delta));

        esfera.center.set(posicion);
}
```

Exercicio: facer o mesmo cos planetas.

Nota: ter en conta que a posición dos planetas non está rotada (se fai no renderer rotando a matriz opengl) e polo tanto será necesario facelo nun vector temporal antes de asinarlle a posición a esfera.

Arquivo Planeta.java

```
/**
 * Para controlar os choques
 */
private Sphere esfera;

private Vector3 postemporal;


public Planeta(){
        posicion = new Vector3(0,0,0);
        angulo_rotacion=0;
        angulo_traslacion=0;
        velocidaderotacion=0.3f;
        velocidadetraslacion=0f;
        escala.set(1000f,1000f,1000f);
        nome_planeta=NOMES_PLANETAS.SOL;

        postemporal = new Vector3(0,0,0);
        esfera = new Sphere(postemporal, escala.x);
}

public Sphere getEsfera() {
        return esfera;
}

public void update(float delta){

        angulo_rotacion+=velocidaderotacion*delta;
        if (angulo_rotacion>=360) angulo_rotacion=0;

        angulo_traslacion+=velocidadetraslacion*delta;
        if (angulo_traslacion>=360) angulo_traslacion=0;

        postemporal.set(posicion);
        postemporal.rotate(angulo_traslacion, 0, 1,0);
        esfera.center.set(postemporal); // Para detectar os choques
}
```

Unha vez que xa temos as esferas podemos controlar os choques.
Isto o facemos na clase ControladorSistemaSolar, chamando o método overLaps da clase esfera para cada un dos corpos que queremos controlar.

```
/**
 * Controla que a nave choque e emite un son máis rápido cada vez
 * @param delta
 */
private void controlarChoques(float delta){

        tempVector1.set(nave.getPosicion());

        // Controlamos planetas
        for (Planeta planeta : mundo3d.getPlanetas()){
                tempVector2.set(planeta.getEsfera().center);

                if (planeta.getEsfera().overlaps(nave.getEsfera())){
                        SistemaSolar.imprimirLog("SE ACABO");
                }
        }
}

public void update(float delta) {

        actualizarPlanetas(delta);
        actualizarNave(delta);
        controlarChoques(delta);
        processInput();
}
```

Incorporando efectos de son.

Imos facer que cando a nave se achegue a un obxecto co que poida chocar, empeza a soar un beep cada vez máis rápido.

Necesitamos cargar o son. Xa temos no proxecto dentro do cartafol Assets/Sonidos os arquivos necesarios.

Definimos a clase Sonidos dentro do paquete principal.

Arquivo Sonidos.java

```
public class Sonidos {

	public static Sound choque;

	public static void inicializarSonidos(){


		choque = Gdx.audio.newSound(Gdx.files.internal("sonidos/choque.mp3"));

	}
	public static void playMusica(){

	}

	public static void stopMusica(){

	}

	public static void dispose(){
		choque.dispose();
	}
}
```

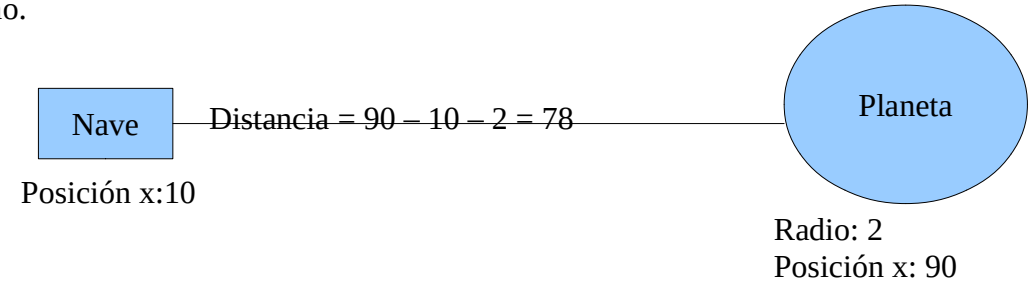
Como vemos é a mesma clase que no caso do xogo do pescador pero só imos a implementar o necesario para tocar un son.

Definimos na clase Nave a distancia a partires de que comeza a avisar.

Arquivo Nave.java

```
	/**
	 * Distancia a partires de que avisa o sonar perigo de colision
	 * O tamaño da nave é unha unidade. A velocidade máxima tardaría 2 segundos en chocar.
	 */
	public static final float DIST_COLISION=100f;
```

Agora temos que implementar o código.
A idea é de calcular a distancia dende a nave á cada planeta. A dita distancia restámoslle a escala do mesmo.



Non teremos que tocar o 'beep' ata que a distancia sexa menor que a distancia de seguridade. O código para facer isto dende a clase ControladorSistemaSolar será:

```
float dist = Math.abs(tempVector2.dst(tempVector1))-planeta.getEscala().x;
if (dist <= Nave.DIST_COLISION){


}
```

Nese intre teremos que tocar o beep, pero este dura polo menos 1/4 de segundo. A idea é que cando esteamos á máxima distancia (DIST_COLISION en nave) empece a tocar cun intervalo de dous segundos, e a medida que imos achegándonos toque máis rápido ata un máximo de 1/4 de segundo.

Polo tanto teremos que pasar a distancia (que vai dende DIST_COLISION a 0) a valores de 2 a 0.25 e teremos que facer que non volva a tocar o son ata que o anterior rematou.
Como nos atopamos nun bucle infinito dentro da clase controladora imos ter que usar un cronómetro para levar a conta do tempo.

O código completo.
Arquivo ControladorSistemaSolar.java

```
private float crono=0f;
private void controlarChoques(float delta){

	tempVector1.set(nave.getPosicion());

	// Controlamos planetas
	for (Planeta planeta : mundo3d.getPlanetas()){
		tempVector2.set(planeta.getEsfera().center);

		if (planeta.getEsfera().overlaps(nave.getEsfera())){
			SistemaSolar.imprimirLog("SE ACABO");
		}
		float dist = Math.abs(tempVector2.dst(tempVector1))-planeta.getEscala().x;
		if (dist <= Nave.DIST_COLISION){
			if (crono<=0){
			    crono = (dist*2)/Nave.DIST_COLISION;
			    if (crono < 0.25f) crono = 0.25f;
			    Sonidos.choque.play();
			}
			crono-=delta;;
		}
	}
}
```

Por motivos de tempo non imos crear novas pantallas e simplemente poñeremos unha mensaxe de fin de xogo e será necesario que o usuario preme unha tecla para comezar de novo.

Polo tanto creamos unha variable boolean na clase PantallaSistemaSolar igual que se fixo no xogo do pescador e no caso de que se acabe non chamaremos á clase controladora.

Arquivo PantallaSistemaSolar.java

```
public static boolean finxogo=false;

@Override
public void render(float delta) {
	// TODO Auto-generated method stub
	xogorender.render();
	if (!finxogo)
		controladorSistemaSolar3D.update(delta);
}
```

Arquivo ControladorSistemaSolar.java

```
private void controlarChoques(float delta){

	tempVector1.set(nave.getPosicion());

	// Controlamos planetas
	for (Planeta planeta : mundo3d.getPlanetas()){
		tempVector2.set(planeta.getEsfera().center);

		if (planeta.getEsfera().overlaps(nave.getEsfera())){
			PantallaSistemaSolar.finxogo=true;
		}
		float dist = Math.abs(tempVector2.dst(tempVector1))-planeta.getEscala().x;
		if (dist <= Nave.DIST_COLISION){
			if (crono<=0){
			    crono = (dist*2)/Nave.DIST_COLISION;
			    if (crono < 0.25f) crono = 0.25f;
			Sonidos.choque.play();
			}
			crono-=delta;;
		}
	}
}
```

Arquivo SistemaSolarRenderer.java

```
private void dibuxarMensaxes(){

	if (PantallaSistemaSolar.finxogo){
		AssetsXogo.bitmapfont.draw(sprite, "FIN DE XOGO", camara2d.viewportWidth/2-
6,camara2d.viewportHeight/2);
	}

}


public void render(){
gl.glClear(GL11.GL_COLOR_BUFFER_BIT | GL11.GL_DEPTH_BUFFER_BIT);

// DIBUXA FONDO DE ESTRELAS
sprite.begin();
dibuxarFondo();
sprite.end();
```

```
        if (!PantallaSistemaSolar.finxogo){
            gl.glEnable(GL11.GL_DEPTH_TEST);
            gl.glEnable(GL11.GL_TEXTURE_2D);

            gl.glMatrixMode(GL11.GL_MODELVIEW);
            gl.glLoadIdentity();


            vector3temp.set(nave.getPosicion());
                camara3d.position.set(vector3temp.sub(nave.getDireccion().cpy().scl(3)).add(0,0.5f,0));
            vector3temp.set(nave.getPosicion());
                camara3d.direction.set(vector3temp.sub(camara3d.position));

            camara3d.update();
            camara3d.apply(gl);

                amosarPlanetaseSol();
                amosarNave();

            gl.glDisable(GL11.GL_DEPTH_TEST);
            gl.glDisable(GL11.GL_TEXTURE_2D);

        }


        // PARTE 2D
        sprite.begin();
        dibuxarControis();
        dibuxarMensaxes();
        sprite.end();

 }
```

Ademais, cando o usuario prema sobre a pantalla cando rematou o xogo se debe inicializar todo para comezar de novo.

Por tanto creamos un método inicializar na clase Nave, outro na clase Mundo.

Arquivo Nave.java

```
        public void inicializarNave(){
                posicion.set(2590,0,0);
                direccion.set(0,0,1);
                angulo_rotacion.set(0,0,0);
                velocidade=0;

        }
```

Arquivo Mundo3d.java

```
        public void incializarXogo(){
                nave.inicializarNave();
        }
```

Arquivo PantallaSistemaSolar.java

```
        @Override
        public boolean touchDown(int screenX, int screenY, int pointer, int button) {
                ..................

                if (finxogo){
                        mundo3d.incializarXogo();
                        finxogo=false;
                }

                return false;
        }
```

OS DISPAROS.

Imos facer que a nosa nave dispare raios láser.
Usaremos o modelo obj dos planetas, pero dándolle unha escala pequena.
Xa temos a textura (laser.bmp) no noso proxecto base.

O primeiro será cargar dita textura no AssetsXogo.

Arquio AssetsXogo.java

```
        public static Texture textura_laser;

        public static void cargarGraficos(){
                ..............
                imageFileHandle = Gdx.files.internal("texturas/laser.bmp");
                textura_laser = new Texture(imageFileHandle);

                ....................
```

Despois crearemos unha clase Disparo no paquete Modelo e que gardaremos toda a información necesaria para un disparo.
O disparo partirá dunha posición inicial (igual á da nave) e terá un alcance (posición final) en base a dirección que teña a nave no momento do disparo.
Polo tanto o disparo terá unha dirección (que será a que teña a nave cando dispara) e terá unha distancia máxima.

Arquivo Disparo.java

```
public class Disparo {
        /**
         * Posición do disparo
         */
        private Vector3 posicion;

        /**
         * Dirección do disparo
         */
        private Vector3 direccion;

        /**
         * Velocidade do disparo
         */
        private float velocidade=0f;

        /**
         * Para controlar os choques
         */
        private Sphere esfera;

        /**
         * Tamaño do disparo.
         */
        public final static float ESCALA_DISPARO = 0.2f;

        /**
         * Distancia máxima do láser.
         */
        public final static float DIST_MAXIMA = 150;

        /**
         * Distancia actual que leva percorrido o láser
         */
```

```
	private float distancia_percorrida;


	/**
	 * Vector para non usar cpy()
	 */
	private Vector3 vectortemp;


	public Disparo(){
		vectortemp = new Vector3();
	}

	public Disparo(Nave nave){
		this();
		this.direccion = nave.getDireccion().cpy();
		this.posicion=nave.getPosicion().cpy();
		esfera = new Sphere(posicion,ESCALA_DISPARO);
		velocidade = nave.getVelocidade()+50;
	}


	public float getDistancia_percorrida() {
		return distancia_percorrida;
	}


	public Vector3 getPosicion() {
		return posicion;
	}

	public void setPosicion(Vector3 posicion) {
		this.posicion = posicion;
	}

	public Vector3 getDireccion() {
		return direccion;
	}

	public void setDireccion(Vector3 direccion) {
		this.direccion = direccion;
	}

	/**
	 * Actualiza a posición da nave.
	 * @param delta
	 */
	public void update(float delta){


		vectortemp.set(direccion).scl(velocidade*delta);
		distancia_percorrida+=vectortemp.dst(0,0,0);
		posicion.add(vectortemp);

		esfera.center.set(posicion);
	}

}
```

Agora temos que modificar a clase Nave para engadir un array de disparos.

Arquivo Nave.java

```
	/**
	 * Disparos da nave
	 */
	private Array<Disparo>disparos;

	public Nave(){
		.............
```

```
                disparos = new Array<Disparo>();
        }
        public Array<Disparo> getDisparos() {
                return disparos;
        }

        /**
         *  Dispara o laser
         */
        public void disparar(){
                disparos.add(new Disparo(this));

        }
```

Unha vez feito isto temos que controlar cando prememos o control de disparo. No noso caso o control vai estar no centro da pantalla e ocupará unha terceira parte da mesma:

Arquivo Controis.java

```
        public static enum CONTROIS {ARRIBA,ABAIXO,ESQUERDA,DEREITA,ACELERAR,FRENAR,PARAR,DISPARAR};

        public static void asinarTamañoControis(){
                ..................
                tamaño_controis.put(CONTROIS.DISPARAR,new
Rectangle(cam2d.viewportWidth/3,cam2d.viewportWidth/3,cam2d.viewportWidth/3,cam2d.viewportWidth/3));
```

Agora é necesario modificar a clase PantallaSistemaSolar.

Arquivo PantallaSistamaSolar.java (hai que engadir DISPARO nas keys da clase Controladora)

```
        @Override
        public boolean touchDown(int screenX, int screenY, int pointer, int button) {
        ......................
                if (Intersector.overlaps(pulsa,Controis.tamaño_controis.get(Controis.CONTROIS.DISPARAR))){
                        controladorSistemaSolar3D.presionaTecla(ControladorSistemaSolar.Keys.DISPARO);
                }
                ............
        @Override
        public boolean touchUp(int screenX, int screenY, int pointer, int button) {
                ...........
                if (Intersector.overlaps(pulsa,Controis.tamaño_controis.get(Controis.CONTROIS.DISPARAR))){
                        controladorSistemaSolar3D.liberaTecla(ControladorSistemaSolar.Keys.DISPARO);
                }
                ............
```

Imos a clase Controladora.
En dita clase, cando prememos a tecla de disparo engadimos un disparo ó array de disparos da clase nave. Poderíamos poñer algunha condición para que disparase un número de disparos á vez (propiedade size da clase Array).

O mesmo tempo controla que cando un disparo chega á distancia máxima o fai desaparecer.

Arquivo ControladorSistemaSolar.java

```
        public static enum Keys {
                DISPARO,XIRAR_ESQUERDA, XIRAR_DEREITA,XIRAR_ARRIBA,XIRAR_ABAIXO, AUMENTAR_VELOCIDADE,
DISMINUIR_VELOCIDADE,PARAR
        };


        /**
         * Actualiza os disparos da nave.
         * @param delta
```

```
         */
        private void actualizarDisparos(float delta){
                for (Disparo disparo : nave.getDisparos()){
                        disparo.update(delta);
                }

        }

        /**
         * Controla os disparos da nave.
         * @param delta
         */
        private void controlarDisparos(float delta){
                for (Disparo disparo : nave.getDisparos()){
                        if (disparo.getDistancia_percorrida()>= Disparo.DIST_MAXIMA){
                                nave.getDisparos().removeValue(disparo, true);
                        }
                }

        }

        public void update(float delta) {

                actualizarPlanetas(delta);
                actualizarNave(delta);
                actualizarDisparos(delta);

                controlarChoques(delta);
                controlarDisparos(delta);
                processInput();
        }

        private void processInput(){
                ..............
                if (keys.get(Keys.DISPARO)){
                        if (nave.getDisparos().size==0)
                                nave.disparar();
                }

                ...................
```

Agora queda debuxar os disparos na clase SistemaSolarRenderer.

Arquivo SistemaSolarRenderer.java

```
        private void amosarDisparos(){
                for (Disparo disparo : nave.getDisparos()){
                        gl.glPushMatrix();

                        gl.glTranslatef(disparo.getPosicion().x, disparo.getPosicion().y,
disparo.getPosicion().z);
                        gl.glScalef(Disparo.ESCALA_DISPARO,Disparo.ESCALA_DISPARO,Disparo.ESCALA_DISPARO);
                        AssetsXogo.textura_laser.bind();
                        AssetsXogo.mesh_planetas.render(GL11.GL_TRIANGLES);


                        gl.glPopMatrix();
                }
        }


        public void render(){
                ..............
                amosarPlanetaseSol();
                amosarNave();
                amosarDisparos();
```

Exercicio: engadir o son do disparo. O arquivo se atopa no cartafol SONIDOS e ten de nome disparo.mp3.

Solución:

Arquivo Sonido.java

```
public static Sound disparo;

public static void inicializarSonidos(){
        choque = Gdx.audio.newSound(Gdx.files.internal("sonidos/choque.mp3"));
        disparo = Gdx.audio.newSound(Gdx.files.internal("sonidos/disparo.mp3"));
}
public static void dispose(){
        choque.dispose();
        disparo.dispose();
}
```

Arquivo ControladorSistemaSolar.java

```
private void processInput(){
        .............
        if (keys.get(Keys.DISPARO)){
                if (nave.getDisparos().size==0){
                        Sonidos.disparo.play();
                        nave.disparar();
                }
        }
```

OS ASTERIODES.

Neste punto xa debedes ser capaces de crear os asteroides, controlar se chocan cun planeta ou a nave e movelos nunha dirección.

Agora debemos de crear os asteroides que van dirixirse á Terra.
A información a gardar será a mesma que no caso dos disparos.
Para facelo máis rápido non imos rotar os asteroides.
Como no caso do pescador poderíamos crear unha clase que tivese os atributos comúns e así aforrar código (non o imos facer).

Exercicio: intentade crear 10 asteroides que vaian cara a Terra e controlar o impacto dos mesmos. En caso de impacto coa Terra débese informar e rematar o xogo.

Cando o disparo da nave impacte sobre un asteroide os dous desaparecerán.
Podedes buscar un efecto de son de tipo explosión. Tedes xa un no cartafol 'Sonidos'.

Nota: A textura do asteroide chámase asteroid.png e o modelo asteroid.obj. Xa se atopa na proxecto base.

Tamén dispoñedes dun radar 2D (nome 'radar.png') no cartafol Texturas. A idea é que amose os meteriotos con respecto á posición da nave.

SOLUCIÓN: Tedes a solución no DVD, no cartafol /Proxectos Eclipse/Proxecto 3D Planetas/6.- Con Radar e Asteroides/

FUNCIÓNS NON VISTAS.

Neste apartado nomearei algunhas das funcións non vistas neste curso.

- **Nova API 3D:** http://www.badlogicgames.com/forum/viewtopic.php?f=23&t=7020
- **Open GL 2.0: Uso de Shaders:** https://github.com/mattdesl/lwjgl-basics/wiki/Shaders
- **Motor de físicas 3D:** http://code.google.com/p/bullet/
- **Animación 3D usando arquivos MD2:** http://code.google.com/p/libgdx-users/wiki/MD2_Keyframe_Animation

ENLACES E BIBLIOGRAFIA.

- **Libro 'Beginning Android Games':** http://www.apress.com/9781430246770
- **Exemplo de uso da nova API 3D:** http://blog.xoppa.com/basic-3d-using-libgdx-2/?utm_source=rss&utm_medium=rss&utm_campaign=basic-3d-using-libgdx-2
- **Explicacion matrices opengl:** http://www.learnopengles.com/understanding-opengls-matrices/
- **Explicación gráfica da cámara nas dous proxeccións (orthogonal e perspective):** http://code.google.com/p/libgdx/wiki/GraphicsFundamentalsViewport