LIBGDX Desenrolando o xogo 2D

UNDIDADE 2: Desenrolando o xogo 2D

Sumario

- 1 Preparando o 'esqueleto' do noso xogo
- 2 A cámara 2D
    - 2.1 Introducción
    - 2.2 Cámara ortográfica
    - 2.3 O tamaño da pantalla. A relación de aspecto
    - 2.4 Movendo a cámara
- 3 Os gráficos
- 4 As colisións

Preparando o 'esqueleto' do noso xogo

Como comentamos anteriormente, a clase que usan os diferentes proxectos debe ser unha subclase da clase *ApplicationAdapter*, pero tamén comentamos que podía ser unha subclase da clase *Game*.

Que diferenza hai entre unha opción e outra ?

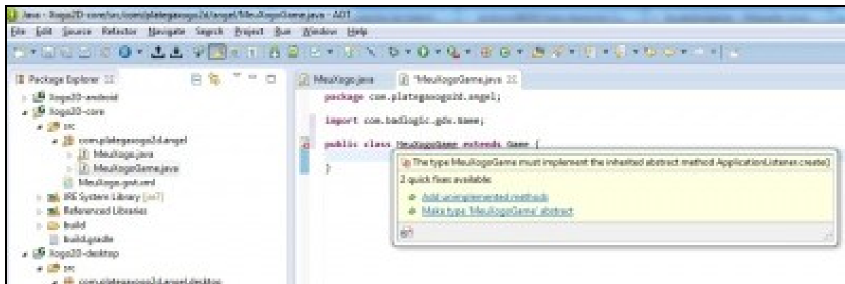
A diferenza se atopa en que se usamos a clase Game imos poder ter (a nivel de programación) unha clase por cada pantalla para o noso xogo e podemos xestionalas-programalas independentemente. Moito máis práctico, claro e doado de manter. En caso de usar a ApplicationAdapter, todo o código de noso xogo debería ir dentro do render de dita clase (que é o método que se chama de forma continuada) e se o xogo tivera varias pantallas teríamos que engadir algunha lóxica de programación para que amosara unha ou outra segundo o caso.

Polo tanto imos preparar o noso proxecto para utilizar a clase Game.

- Primeiro imos crear unha nova clase no proxecto Xogo2D-Core, e dentro deste no paquete com.plategaxogo2d.o_voso_nome, que é o paquete onde se atopa a clase MeuXogo. Dámoslle de nome MeuXogoGame e facemos que derive da clase Game.

<u>Nota:</u> Ó facelo deberemos de importar dita clase coa combinación de teclas Control+Shift+O ou ben situarnos enriba da clase e escoller a opción *Import .....*

Despois de facelo veremos que aparece un erro no nome da clase. Se nos situamos enriba dela aparecerá unha ventá para engadir os métodos que deben estar definidos na clase.



Aparecerá o método create...

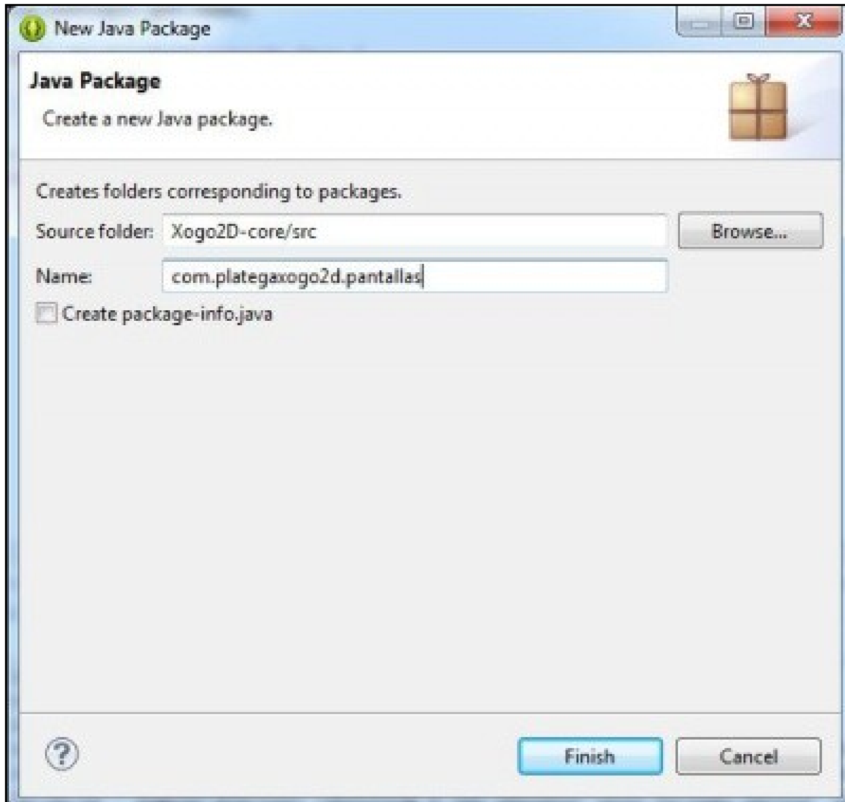
Agora debemos de cambiar a clase que usan as diferentes plataformas pola nova clase creada.

```
package com.plategaxogo2d.angel.desktop;
```

```
import com.badlogic.gdx.backends.lwjgl.LwjglApplication;
import com.badlogic.gdx.backends.lwjgl.LwjglApplicationConfiguration;
import com.plategaxogo2d.angel.MeuXogoGame;

public class DesktopLauncher {
public static void main (String[] arg) {
LwjglApplicationConfiguration config = new LwjglApplicationConfiguration();
new LwjglApplication(new MeuXogoGame(), config);
}
}
```

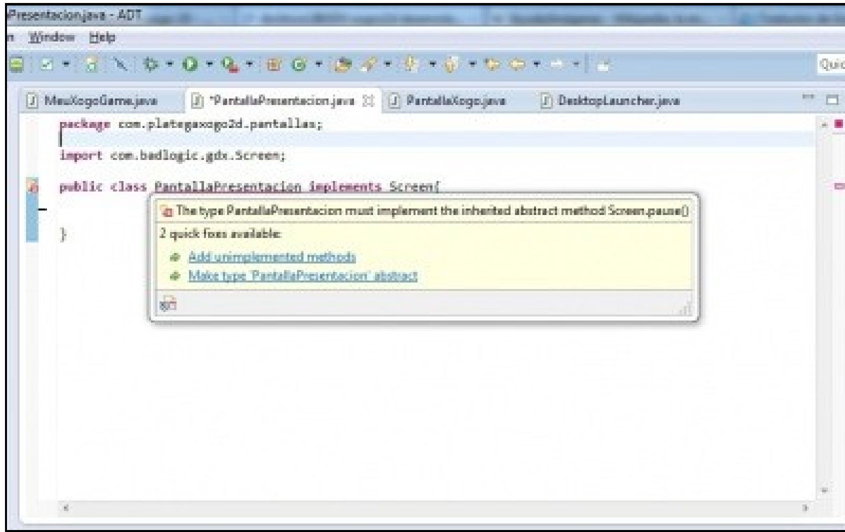
Nota: Exemplo feito sobre a versión Desktop do xogo. Deberemos facelo en todos os proxectos (html-android).

Agora imos crear un paquete onde van estar as pantallas do noso xogo. Dito paquete terá de nome *com.plategaxogo2d.pantallas*. Para crear o paquete prememos o botón dereito sobre o cartafol 'src' do proxecto Xogo2D-Core e escollemos a opción *New => Package*.



Dentro de dito paquete creamos unha clase de nome *PantallaPresentacion* que implemente a interface *Screen* (despois da definición da clase deberemos poñer *implements Screen*).

Ó facelo e despois de facer o import da clase (control+shift+O) aparecerá un erro enriba da clase. Igual que fixemos anteriormente, nos situamos enriba do nome da clase e escollemos a opción de Add Unimplemented Methods.



Faremos as seguintes pantallas (repetiremos o proceso anterior):

- A do xogo. Chamarémoslle *PantallaXogo*.
- A da axuda. Chamarémoslle *PantallaAxuda*.
- A da High Score. Chamarémoslle *PantallaScore*.

Como facemos agora para pasar o control a cada unha das pantallas ? Faise a través do método setScreen da clase Game.

Imos facer no noso xogo que o control pase á pantalla de xogo:

```
package com.plategaxogo2d.angel;

import com.badlogic.gdx.Game;
import com.plategaxogo2d.pantallas.PantallaXogo;

public class MeuXogoGame extends Game {

private PantallaXogo pantallaxogo;

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

pantallaxogo = new PantallaXogo();
setScreen(pantallaxogo);
}

}
```

A partir deste momento, o framework páselle o control o método render da clase PantallaXogo e se queda nese método ata que decidamos cambiar de pantalla...

Analizamos agora os métodos implementados na clase *PantallaXogo*.

```
package com.plategaxogo2d.pantallas;

import com.badlogic.gdx.Screen;
import com.plategaxogo2d.angel.Utiles;
```

```
public class PantallaXogo implements Screen {

@Override
public void render(float delta) {
// TODO Auto-generated method stub

}

@Override
public void resize(int width, int height) {
// TODO Auto-generated method stub
Utiles.imprimirLog("Resize", "RESIZE", "RESIZE");

}

@Override
public void show() {
// TODO Auto-generated method stub
Utiles.imprimirLog("PantallaXogo", "SHOW", "SHOW");
}

@Override
public void hide() {
// TODO Auto-generated method stub
Utiles.imprimirLog("PantallaXogo", "HIDE", "HIDE");

}

@Override
public void pause() {
// TODO Auto-generated method stub
Utiles.imprimirLog("PantallaXogo", "PAUSE", "PAUSE");

}

@Override
public void resume() {
// TODO Auto-generated method stub
Utiles.imprimirLog("PantallaXogo", "RESUME", "RESUME");

}

@Override
public void dispose() {
// TODO Auto-generated method stub
Utiles.imprimirLog("PantallaXogo", "DISPOSE", "DISPOSE");

}

}
```

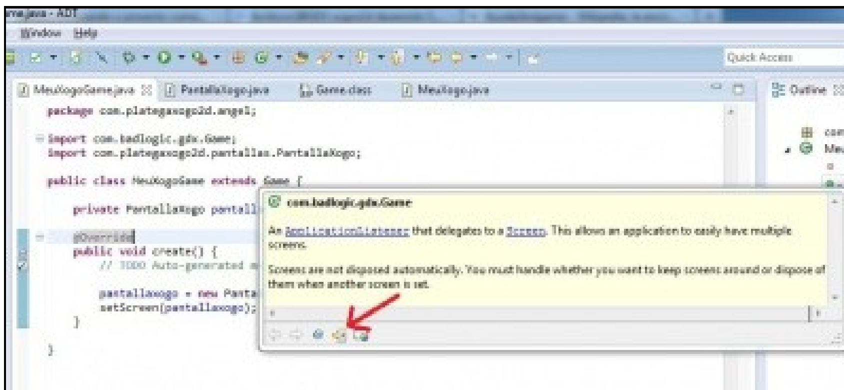
Como vemos son moi parecidos os vistos anteriormente na clase *ApplicationAdapter* pero con algunha pequena diferenza...

Se executados o proxecto Desktop, por exemplo, aparecerá unha ventá negra (é normal). Se minimizades a ventá e despois pechades o xogo veredes a secuencia de eventos.

```
Problems  @ Javadoc  Declaration  Console
<terminated> DesktopLauncher [Java Application] C:\Program File
XOGO2D: PantallaXogo:SHOW:SHOW
XOGO2D: Resize:RESIZE:RESIZE
XOGO2D: Resize:RESIZE:RESIZE
XOGO2D: PantallaXogo:PAUSE:PAUSE
XOGO2D: PantallaXogo:RESUME:RESUME
XOGO2D: Resize:RESIZE:RESIZE
XOGO2D: PantallaXogo:PAUSE:PAUSE
XOGO2D: PantallaXogo:HIDE:HIDE
```

Como vemos, cando pechamos o xogo non executa o método dispose, se non o método hide. Por qué é así ? Podemos ver que é o que fai a clase Game cando chama o método dispose.

Situar o cursor sobre o nome Game na clase MeuXogoGame, situarvos enriba da parte amarela na ventá que aparece, e premede sobre o botón de *Open Declaration*...



Abrirase unha nova ventá có código de dita clase. Podemos observar o que fai cando se chama ó método dispose...

```java
 * screen is set.
 * </p> */
public abstract class Game implements ApplicationListener {
    private Screen screen;

    @Override
    public void dispose () {
        if (screen != null) screen.hide();
    }

    @Override
    public void pause () {
        if (screen != null) screen.pause();
    }

    @Override
    public void resume () {
        if (screen != null) screen.resume();
    }

    @Override
    public void render () {
        if (screen != null) screen.render(Gdx.graphics.getDeltaTime());
    }
```

Polo tanto, temos que ser nos o que chamemos a dito método dende a clase MeuXogoGame cando se produza o evento de dispose.

Para facelo só temos que sobrescribir dito método.

```
package com.plategaxogo2d.angel;

import com.badlogic.gdx.Game;
import com.plategaxogo2d.pantallas.PantallaXogo;

public class MeuXogoGame extends Game {

private PantallaXogo pantallaxogo;

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

pantallaxogo = new PantallaXogo();
setScreen(pantallaxogo);
}

@Override
public void dispose(){
super.dispose();
pantallaxogo.dispose();
}
}
```

É IMPORTANTE LEMBRAR QUE SEMPRE TEREMOS QUE SER NOS O QUE DEBEMOS CHAMAR O MÉTODO DISPOSE DA CLASE QUE IMPLEMENTE A INTERFACE SCREEN.

- Vos estaredes preguntando, e por que é tan importante...? Porque escribiremos nese método as ordes necesarias para liberar da memoria os recursos que teña dita pantalla, como poden ser os gráficos...
- E por que non facelo no método hide ? Poderíamos, pero se vos fixades no método setScreen da clase Game, este chama o método hide. Polo tanto se no noxo xogo queremos manter os datos e non liberar os recursos cando cambiamos de pantalla (por exemplo na pausa do xogo...) non podemos facelo en dito método...

Como comentamos anteriormente, imos usar a clase Game para poder cambiar de pantalla. Agora mesmo, o control está na clase PantallaXogo, no método render. Ó cabo dun tempo quereremos cambiar de pantalla e acceder á pantalla principal, ou a de High Scores ou outra calquera. Como vimos, deberemos chamar ó método setScreen da clase Game.

Unha forma de facelo é pasando ó constructor da clase PantallaXogo o obxecto que deriva da clase Game da seguinte forma.

Código da clase PantallaXogo:

```
public class PantallaXogo implements Screen {

private MeuXogoGame meuxogogame;

public PantallaXogo(MeuXogoGame meuxogogame){
this.meuxogogame=meuxogogame;
}

        .............
}
```

Código da clase MeuXogoGame:

```
public class MeuXogoGame extends Game {

private PantallaXogo pantallaxogo;
```

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

pantallaxogo = new PantallaXogo(this);
setScreen(pantallaxogo);
}

@Override
public void dispose(){
super.dispose();
pantallaxogo.dispose();
}
}
```

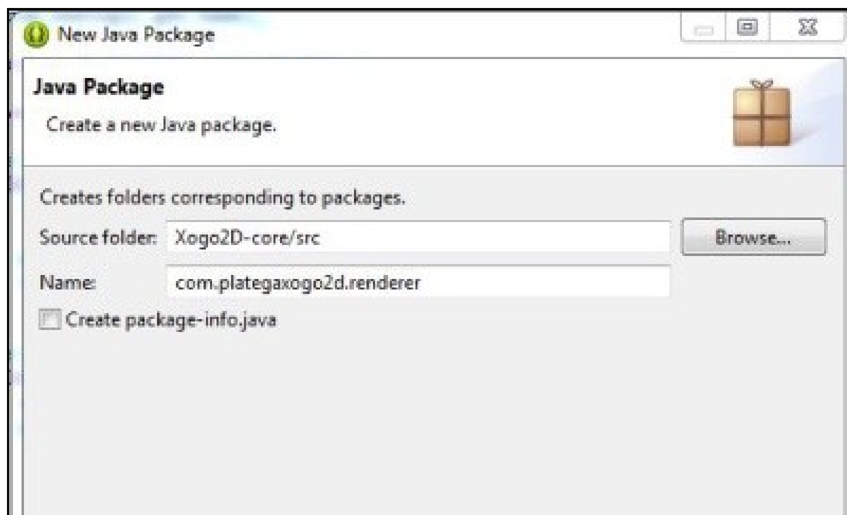
Agora podemos dende a clase PantallaXogo facer uso do obxecto *meuxogogame* e chamar ó método setScreen para cambiar de pantalla...

- **TRABALLO A REALIZAR:** facer o mesmo que na PantallaXogo e crear o contructor para pasarlle un obxecto da clase MeuXogoGame ás pantallas: PantallaMenu, PantallaAxuda.

Agora imos crear un novo paquete de nome **com.plategaxogo2d.renderer** onde van ir as clases necesarias para debuxar cada unha das pantallas. Así teremos a clase RendererXogo, RendererMenu, RendererAxuda,...

Loxicamente poderíamos aproveitar as posibilidades da programación orientada a obxectos e facer unha clase con todo o común e herdar dela, pero como estamos a facer unha primeira aproximación a como funciona o framework isto xa quedaría para máis adiante unha vez que teñades soltura no manexo do framework.

Igual que fixemos antes, crearemos un paquete que terá de nome com.plategaxogo2d.renderer. Para crear o paquete prememos o botón dereito sobre o cartafol 'src' do proxecto Xogo2D-Core e escollemos a opción New => Package.



- Dentro de dito paquete creamos unha clase de nome **RendererXogo**.

Esta clase é a que vai a debuxar todos os elementos gráficos do xogo. Poderíamos facelo na mesma clase PantallaXogo ? Si. Por que non o facemos ? Por facilidade á hora de manter o xogo. Desta forma imos separar a parte de 'Control' da parte de 'Visualización' ou 'Render'.

Lembrar que ata o visto ata aquí, agora mesmo o control do programa se atopa no método render da clase PantallaXogo. É esta clase a que vai recibir o control do programa, a que ten o método resize que se chama de forma automática cando se cambia o tamaño,o método dispose que o temos que chamar nos dende a clase MeuXogoGame... o que imos facer será chamar ós métodos da clase RendererXogo que imos usar: dispose, render e resize.

- Agora temos que chamar a un método da clase RendererXogo de forma continua. Creamos por tanto un método en dita clase. Imos chamarlle render e vai levar un parámetro de tipo float de nome delta (xa falaremos para que serve).

- Imos definir un método de nome dispose no que poñeremos o código necesario para liberar a memoria da clase RendererXogo.
- Imos definir un método resize que levará dous parámetros (width e height de tipo int) no que modificaremos o tamaño da cámara se é necesario(o veremos posteriormente).

Código da clase RendererXogo:

```
package com.plategaxogo2d.renderer;

public class RendererXogo {

/**
         * Debuxa todos os elementos gráficos da pantalla
         * @param delta: tempo que pasa entre un frame e o seguinte.
         */
public void render(float delta){

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


}
public void dispose(){

}
}
```

Agora dende a PantallaXogo crearemos un obxecto de dita clase e chamaremos ós métodos *render*, *dispose* e *resize*.

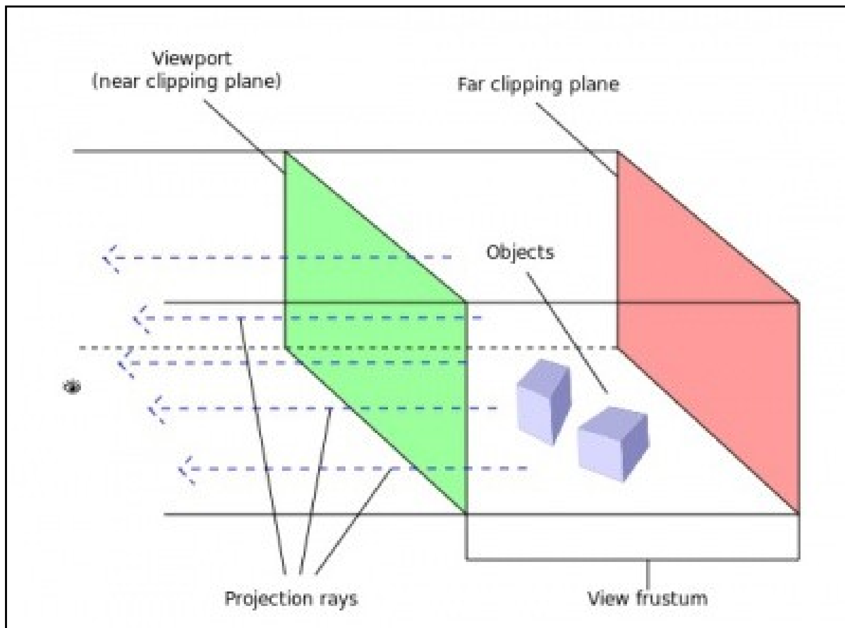
Código da clase PantallaXogo:

```
package com.plategaxogo2d.pantallas;

import com.badlogic.gdx.Screen;
import com.plategaxogo2d.angel.MeuXogoGame;
import com.plategaxogo2d.renderer.RendererXogo;

public class PantallaXogo implements Screen {

private MeuXogoGame meuxogogame;
private RendererXogo rendererxogo;

public PantallaXogo(MeuXogoGame meuxogogame){
this.meuxogogame=meuxogogame;
rendererxogo=new RendererXogo();
}

@Override
public void render(float delta) {
// TODO Auto-generated method stub

rendererxogo.render(delta);
}

@Override
```

```
public void resize(int width, int height) {
// TODO Auto-generated method stub
rendererxogo.resize(width, height);
}

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

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

}

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

}

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

}

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

rendererxogo.dispose();
}

}
```

Neste punto podedes facer unha copia de todos os proxectos e así tedes unha copia para restaurar ademais de ter unha base para empezar outro xogo.

A cámara 2D



Introducción

Temos que entender que todo o que se visualiza nun xogo son puntos nun espazo. No caso dos xogos 2D estamos falando de coordenadas X,Y (Z que sería a profundidade ten un valor de 0)

Cando nos indicamos que queremos ver algo na coordenada (x=10,Y=15) vai existir unha cámara que vai 'transformar' esas coordenadas a coordenadas do noso dispositivo móbil ou pantalla de PC e fará que se visualice no lugar correcto. A cámara vai ter unha posición e un tamaño (área que vai visualizar).

Todos estes datos son aplicados a cada un dos puntos que queremos debuxar en forma dunha serie de operacións matemáticas usando matrices. En OPEN GL existen dúas matrices que veremos en profundidade na parte 3D. Unha matriz vai a establecer o tamaño do que se visualiza (matriz de proxección) e outra vai establecer a posición da cámara e cara a onde mira, é dicir, a súa dirección (matriz de modelado). Se xuntamos as dúas matrices obtemos unha matriz combinada que vai ser a que se aplique a cada un dos puntos do noso xogo.

Cunha cámara poderemos:

- Mover ou rotar a cámara: propiedade **location** / método **translate** e método **rotate**
- Facer zoom ou afastarse: método **zoom**
- Cambiar o tamaño do que visualiza a cámara: propiedades **viewportwidth** e **viewportheight**
- Pasar coordenadas de puntos dende o dispositivo real a coordenadas da cámara e viceversa: método **unproyect** e **proyect**.

Existen outros métodos e propiedades que iremos vendo cando o necesitemos.

Cámara ortográfica

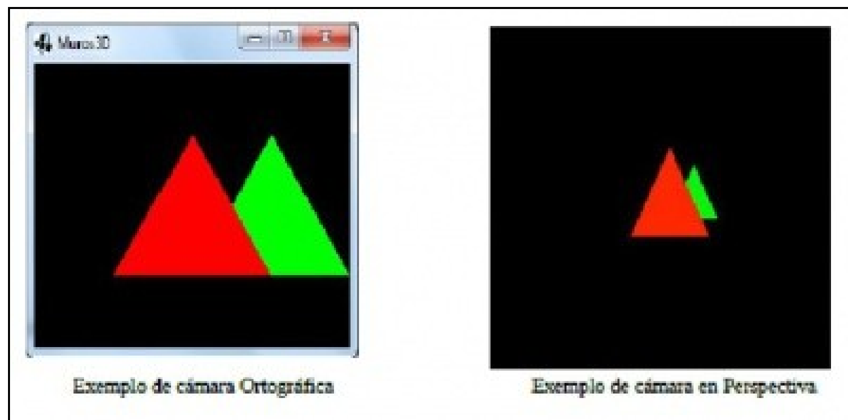
- Clase **OrthographicCamera**: http://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/OrthographicCamera.html

- Deriva da clase Camera

A cámara que se usa nos xogos 2D denomínase cámara ortográfica (orthographic camera). A diferenza da cámara 3D (que se denomina cámara en perspectiva ou perspective camera) esta non ten perspectiva.

En 3D os obxectos mais afastados vense máis pequenos que os próximos.

Por exemplo:



Exemplo de cámara Ortográfica   Exemplo de cámara en Perspectiva

- Neste exemplo estamos a visualizar os mesmos obxectos (dous triángulos) situados na mesma posición nos dous casos. A diferenza é que a cámara ortográfica non ten en conta a distancia/perspectiva.

Métodos máis importantes:

- public void setToOrtho(boolean yDown,float viewportWidth, float viewportHeight)

Define o tamaño do viewport da cámara.

  ◊ ydown: indica se o punto (0,0) está situado na parte superior esquerda (valor true) ou na parte inferior esquerda (valor false)
  ◊ viewportWidth: ancho do viewport.
  ◊ viewportHeight: alto do viewport.

- public void translate(float x,float y)

Traslada a cámara á posición indicada por x,y.

- public void update()

Actualiza a matriz de proxeción e de modelado.

- public Vector3 project(Vector3 worldCoords)

Pasa as coordenadas do mundo a coordenadas da pantalla.

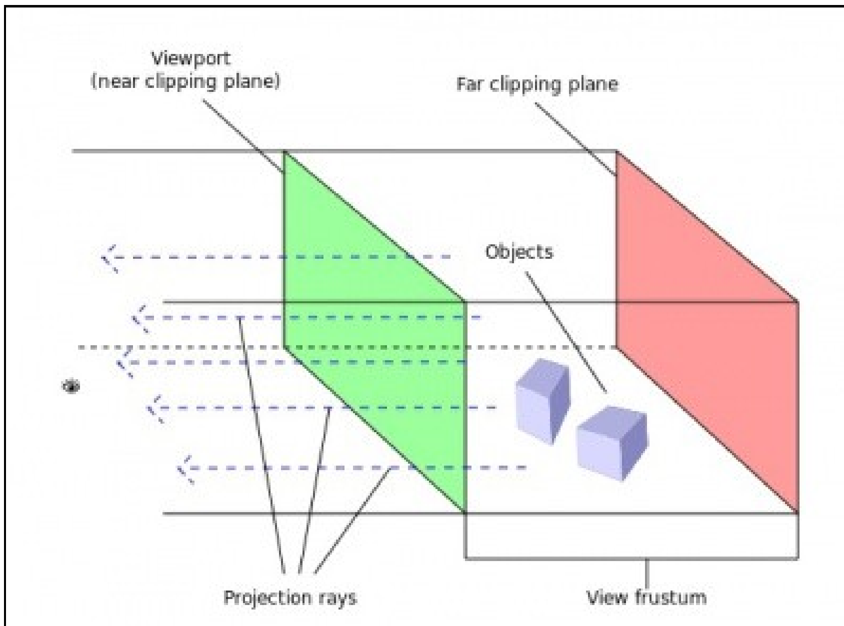
  ◊ worldCoords: coordenadas do mundo.

Definiremos por tanto a cámara no noso xogo.

Código da clase RendererXogo:

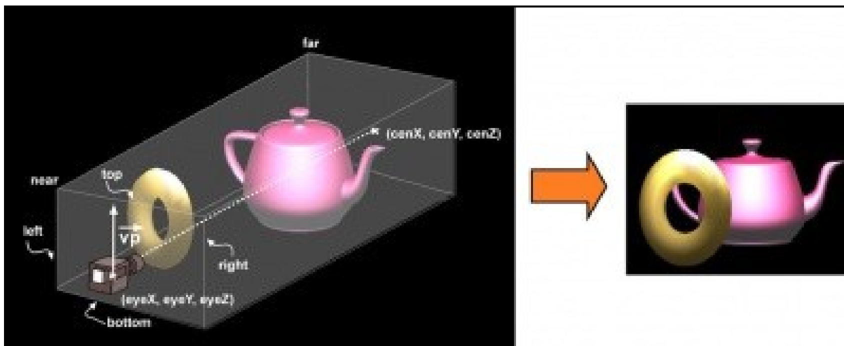
```
public class RendererXogo {

private OrthographicCamera camara2d;

public RendererXogo(){
camara2d = new OrthographicCamera();
}


        ......
```

Agora temos que darlle un tamaño. O tamaño (width e height) da cámara é o que se coñece como **VIEWPORT**. No seguinte debuxo se corresponde co **plano near**.



Os raios indican como nos vemos os obxectos con dita cámara. Temos que imaxinar un encerado e pensar que todo o que vemos vaise 'esmagar' contra dito encerado. É como se leváramos fisicamente os obxectos ata o encerado e os esmagamos. Por iso os obxectos non teñen perspectiva con dita cámara.

Cando definimos unha cámara definimos o tamaño do plano near (viewport width e viewport height) que é igual ó tamaño do plano far . Ó ser unha cámara ortográfica o tamaño dos dous planos é o mesmo sempre. A distancia entre os dous planos é o que se coñece como **VIEW FRUSTRUM** e ven ser o volume de visualización. Todo o que está dentro deste volume é o que se verá. Este volume está definido polas propiedades far e near da cámara e veñen a representar a distancia á cámara.



No caso da cámara 2D estes xa teñen uns valores por defecto. Asi o plano far é 100, o plano near é 0 e o tamaño o temos que asinar nos.

Todos os gráficos os imos debuxar no plano near. Tanto daría o lonxe que os debuxaramos xa que mentres estiveran dentro do plano far se verían igual.

O tamaño da pantalla. A relación de aspecto

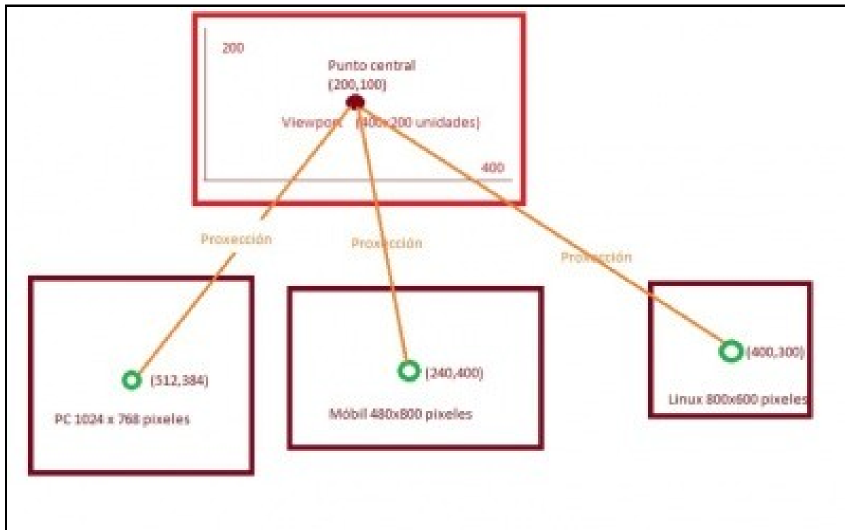
Un dos primeiros problemas a que nos temos que enfrontar cando deseñamos un xogo é determinar a que resolución imos dirixir o noso xogo e cal vai ser a relación de aspecto da pantalla.

A relación de aspecto é a relación entre o seu ancho e o seu alto. Normalmente se dividen e o resultado representa cuanto máis ancho é a pantalla con respecto o seu alto.

- Agora que queda claro o que é e que representa a relación de aspecto temos que aprender outro concepto.

Cando definimos unha cámara e o seu tamaño (viewport) estamos definindo un tamaño 'ficticio' que non ten por que corresponderse coa resolución da

pantalla do noso dispositivo. Así eu podo definir un tamaño para o meu xogo de 400x200 unidades (fixarvos que digo unidades, non pixeles). O que vai facer a cámara é **proxectar** o punto á resolución correspondente.



Nesta pantalla cunha resolución de 400x200 unidades o punto central estaría na coordenada 200x100. O que vai facer a cámara é proxectar este punto ó sistema de coordenadas do noso dispositivo

- Que vantaxes temos se non cambiamos o tamaño do noso Viewport en función da resolución do dispositivo ?

  Pois que se colocamos algo no punto central este estará no centro en todas as resolucións de todos os dispositivos...

- Desvantaxes: Perdemos a relación de aspecto.

  Por exemplo, temos como no exemplo anterior definido o noso xogo cun tamaño de 400x200 unidades. O noso gráfico ocupa 200 unidades de ancho (a metade do ancho total) e 100 unidades de alto (a metade do alto total).
  Proxectemos estes datos a un dispositivo cunha resolución de 600x600 pixeles...

  Relación de aspecto do gráfico na cámara (no viewport) 200/100=2 (o seu ancho é o dobre do seu alto)

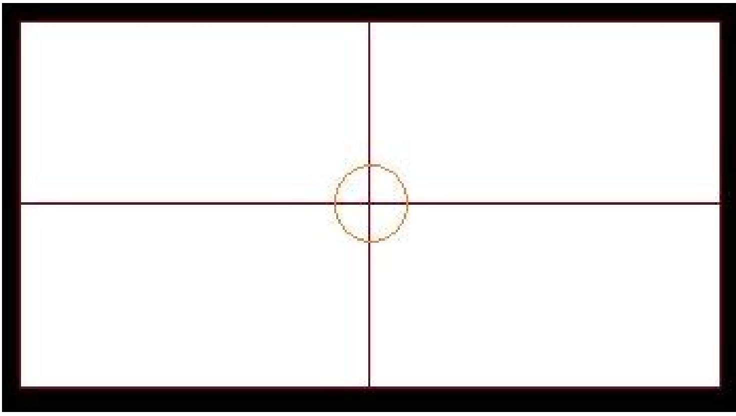
  Se trasladamos estes puntos a unha resolución de 600x600 pixeles:

  Ancho debe ocupar a metade: 600/2 = 300 pixeles de ancho.
  Alto debe ocupar a metade: 600/2 = 300 pixeles de alto.
  Relación de aspecto no dispositivo: 300/300=1. Ten o mesmo alto que ancho e polo tanto sairía deformado.

O veremos máis adiante cun exemplo gráfico e a súa posible solución...

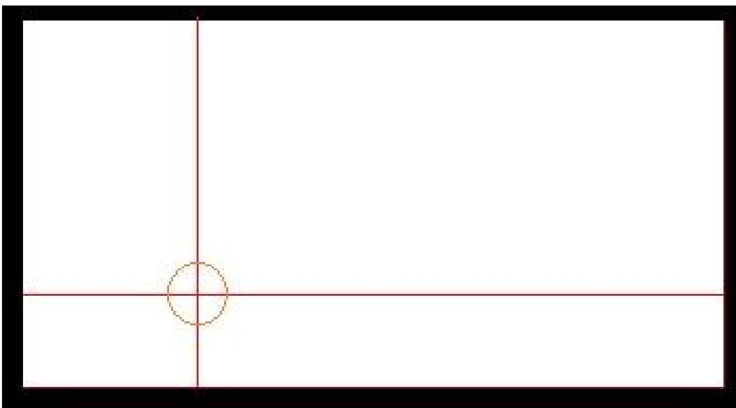
Outro problema que xurde se cambiamos o tamaño do viewport en función do tamaño da resolución é o seguinte:

O tamaño da pantalla pode variar xa que varía a resolución da mesma. Se aplicamos como tamaño do noso xogo dita resolución non poderemos posicionar de forma absoluta os nosos protagonistas, xa que se por exemplo:



Nesta pantalla cunha resolución de 400x200 o punto central estaría na coordenada 200x100.

Cambiemos de resolución:



Resolución da pantalla cambia a 800x400 e punto 200x100
Como vemos a posición non é a mesma.

Para evitalo temos varias aproximacións.

- Se modificamos o tamaño do viewport e a axustamos á resolución do dispositivo, podemos establecer unha unidade de medida definida por nos, no exemplo anterior definiríamos o noso mundo cunha resolución de 400x200 unidades (viewport). Se cambiamos de resolución determinamos cantos pixeles por unidade temos no eixe x (ppux) e cantos no eixe y (ppuy).

  Así, no caso anterior:

  ppux = 800/400 = 2
  ppuy = 400/200 = 2

  Quere dicir que cando queiramos debuxar algo no punto 200,100 teríamos que multiplicalo por ppux e ppuy respectivamente dándonos o punto central da nova resolución (200x2,100x2) = (400,200). O mesmo principio o poderíamos aplicar o tamaño dos gráficos para que tiveran a mesma relación de aspecto.

- Outra aproximación podería ser ampliar o tamaño do viewport en función da relación de aspecto.

Isto o explicaremos co seguinte exemplo. Imaxinemos que definimos un viewport para a cámara de 600x600 unidades....Se proxectamos estas unidades a un dispositivo coa mesma resolución (600x600 pixeles) quedaría así:

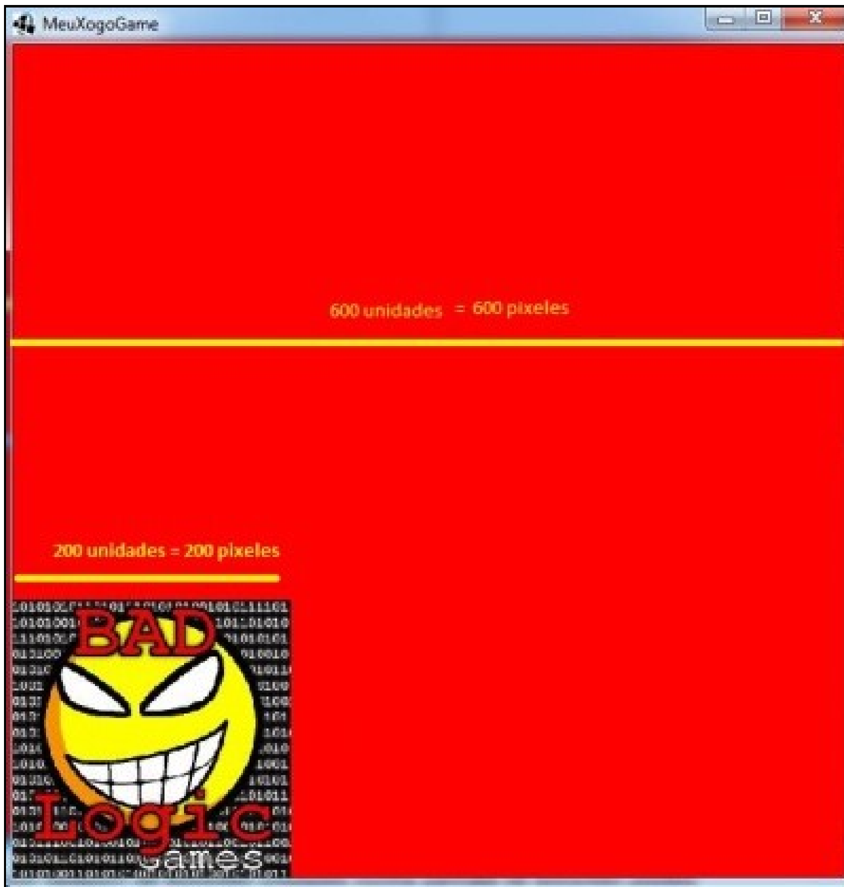


Gráfico de tamaño 200x200 unidades nun viewport de 600x600 unidades nunha pantalla de 600x600 pixeles
Fixarse como o ancho do gráfico ocupa unha terceira parte do ancho total....

Se agora aumentamos o ancho da pantalla a 800x480 pixeles podemos comprobar como o gráfico perde a relación de aspecto.

Gráfico de tamaño 200x200 unidades nun viewport de 600x600 unidades nunha pantalla de 800x480 pixeles
Agora o gráfico mide de ancho 266 pixeles fronte os 200 pixeles que tiña nun dispositivo de 600x600 pixeles de resolución...Isto o sabemos facendo unha regra de tres: se 600 unidades son 800 pixeles, 200 unidades son... Polo tanto o seu ancho vai ser maior que o seu alto. Para evitalo poderíamos calcular a relación de aspecto do dispositivo:

> Relación de aspecto: 800/480 = 1,666. Isto quere dicir que o ancho é 1,666 veces maior que o seu ancho.
>
> E definimos o viewport da cámara, en vez de 600x600 unidades con 600*relacion_aspectox600 unidades = 1000x600 unidades que serán proxectadas nun dispositivo de 800x480pixeles.

Dará como resultado o seguinte:

ViewPort modificado a 1000x600 unidades para manter a relación de aspecto nun dispositivo de 800x480 pixeles
Como vemos aumentamos o ancho de visualización do noso mundo. Isto pode non importar dependendo do tipo de xogo, como por exemplo un de tipo scroll como Replica Island, e non nos importa que un xogador vexa máis 'terreo' do noso xogo que outro dependendo da resolución.

Existen moitos artigos que dan diferentes solucións ó problema:

- http://blog.gemserk.com/2013/01/22/our-solution-to-handle-multiple-screen-sizes-in-android-part-one/
- http://www.acamara.es/blog/2012/02/keep-screen-aspect-ratio-with-different-resolutions-using-libgdx/
- http://www.badlogicgames.com/forum/viewtopic.php?f=11&t=860&p=4965

No noso caso imos utilizar un tamaño fixo (independente da resolución) para o noso mundo e imos facer que todas as resolucións se axusten a dito tamaño. Se cambia a resolución do dispositivo este tamaño 'inventado' mantense e é a cámara a que fai os cálculos para debuxar os puntos no sitio correcto.

Desta forma esquecemos o problema do posicionamento e teremos o problema de que os gráficos poden saír algo deformados.

A forma de establecer o tamaño da cámara (viewport):

- **Propiedade viewportwidth**: ancho da cámara.
- **Propiedade viewportheight**: alto da cámara.

Normalmente se fai uso do método:

- **public void setToOrtho(boolean yDown,float viewportWidth, float viewportHeight)**

  Define o tamaño do viewport da cámara.

Parámetros:

ydown: indica se o punto (0,0) está situado na parte superior esquerda (valor true) ou na parte inferior esquerda (valor false)
viewportWidth: ancho do viewport.
viewportHeight: alto do viewport.

- **<u>IMPORTANTE:</u>** Despois de facer calquera cambio na cámara (posición, tamaño,...) hai que chamar ó método **update()** para que actualice as matrices de proxección e modelado.

O método onde normalmente se establece o seu tamaño é o resize.

No noso caso:

Creamos un paquete novo de nome **com.plategaxogo2d.modelo** (axustade o nome ó voso caso) e definimos unha clase Mundo. En dita clase Mundo imos definir todo o que forma o noso xogo e definiremos o tamaño do noso mundo (o xogo).

- **Un caso práctico**

No meu caso vou desenrolar un xogo para unha resolución de 800x600 pixeles. Isto da unha relación de aspecto de 800/600 = 1,33. Como non quero crear un mundo tan grande, utilizo un ancho e alto máis pequeno pero coa mesma relación de aspecto, por exemplo 400x300 unidades (400/300=1,33).

Imos poñer un tamaño de 400x300 unidades para o noso xogo.

Definiremos dúas propiedades de clase públicas de nomes TAMAÑO_MUNDO_ANCHO, TAMAÑO_MUNDO_ALTO.

Código da clase Mundo:

```
package com.plategaxogo2d.modelo;

public class Mundo {

    public static final int TAMANO_MUNDO_ANCHO=400;
    public static final int TAMANO_MUNDO_ALTO=300;
}
```

Definimos agora o tamaño da cámara 2D no noso xogo:

Código da clase RendererXogo:

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

            camara2d.setToOrtho(false,Mundo.TAMANO_MUNDO_ANCHO,Mundo.TAMANO_MUNDO_ALTO);
            camara2d.update();


    }
```

<u>TAREFA 2.2 A FACER:</u> Esta parte está asociada á realización dunha tarefa.

Fixarse que para debuxar algo que ocupe todo a pantalla necesitaremos 400x300 unidades. A cámara xa se encargará de *proxectar* esas unidades ficticias a pixeles de resolución de pantalla, que poden ser 800x600, 1024x768....

Para ver como queda só temos que modificar o arquivo de configuración da versión Desktop e asinarlle un ancho de 800 pixeles e un alto de 600 pixeles (vos facédeo segundo a tarefa).

Movendo a cámara

Preparación: Agora ides facer unha copia da clase RendererXogo, xa que imos modificala vara amosarvos como se pode mover a cámara. Premede co rato sobre a clase, botón dereito e escollede a opción Copy. Despois repetides a operación pero escolledes a opción Paste. Vos preguntará un nome para a clase. Indicade **UD2_1_RendererXogo**. Modificade a pantalla PantallaXogo para que chame a esta clase.

O problema que temos é que para que se vexa que se move a cámara necesitamos cargar algún gráfico. Isto vai ser explicado no seguinte punto.

Agora imos indicar o código necesario para cargar o gráfico no centro da pantalla (xa explicaremos logo o que estamos a facer).

Código da clase UD2_1_RendererXogo: Explica cales son os métodos que temos para mover a cámara.

```
package com.plategaxogo2d.renderer;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.plategaxogo2d.modelo.Mundo;

public class RendererXogo {

private Texture grafico;
private SpriteBatch spritebatch;

private OrthographicCamera camara2d;



public RendererXogo() {
camara2d = new OrthographicCamera();
grafico = new Texture(Gdx.files.internal("badlogic.jpg"));
spritebatch = new SpriteBatch();
}


    /**
     * Debuxa todos os elementos gráficos da pantalla
     * @param delta: tempo que pasa entre un frame e o seguinte.
     */
    public void render(float delta){
Gdx.gl.glClearColor(1, 1, 1, 1);
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

spritebatch.begin();
spritebatch.draw(grafico,200,150,50,50);
spritebatch.end();

    }

    public void resize(int width, int height) {

            camara2d.setToOrtho(false,Mundo.TAMAÑO_MUNDO_ANCHO,Mundo.TAMAÑO_MUNDO_ALTO);
            camara2d.update();

            spritebatch.setProjectionMatrix(camara2d.combined);

    }
    public void dispose(){
          spritebatch.dispose();
          grafico.dispose();
    }
}
```

Neste punto o único que nos interesa é o visto ata do agora e saber que no método render se está chamando de forma continua e estamos borrando a pantalla (liñas 31 e 32) e debuxando un gráfico (liñas 34-36).

A cámara está situada por defecto no punto medio do viewport. No exemplo estaría no punto (200x150).

Se executamos a versión desktop aparecerá o seguinte:

Para movela faremos uso do método translate.

public void translate(float x,float y)

Traslada a cámara á posición indicada por x,y.

Lembrade que sempre hai que chamar ó método update cando se faga unha modificación. O código pode poñer:

- No método resize se a cámara non se move durante o xogo e queremos darlle unha posición inicial diferente á predeterminada.
- No método render despois de borrar a pantalla e antes de debuxar os gráficos. Se o facemos neste punto, temos que informarlle ó obxecto que debuxa (no exemplo ten de nome spritebatch) que debuxe todo de acordo ás novas matrices de proxección e modelado da cámara (ó cambiar a posición da cámara cambiamos a súa matriz de modelado). Isto se fai chamando o método setProyectionMatrix da clase SpriteBatch.

Imos facelo no método render:

Código da clase RendererXogo_Unidade2_Mover:

```
public void render(float delta){
Gdx.gl.glClearColor(1, 1, 1, 1);
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

camara2d.position.set(100,150,0);
camara2d.update();
                    spritebatch.setProjectionMatrix(camara2d.combined);

spritebatch.begin();
spritebatch.draw(grafico,200,150,50,50);
spritebatch.end();

    }
```

Os gráficos

As colisións