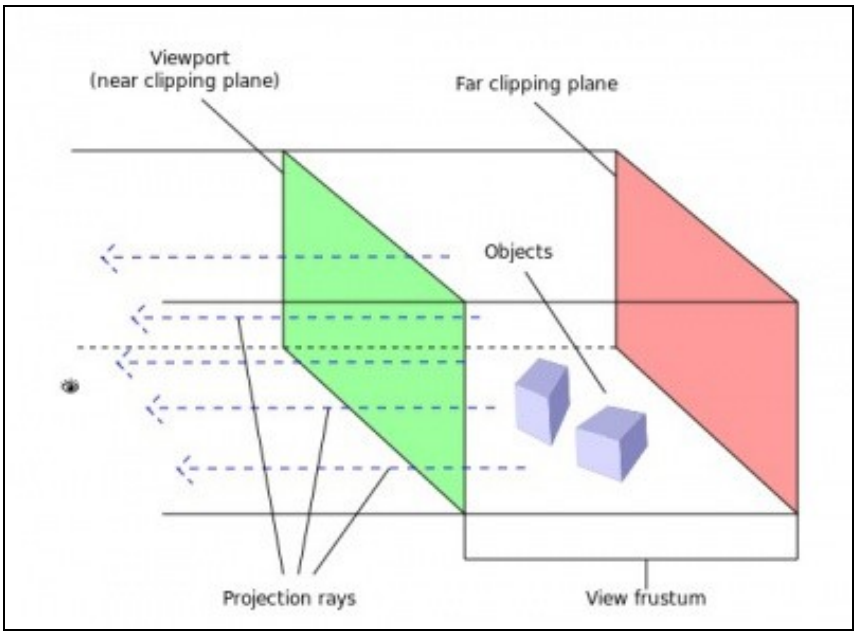
1 LIBGDX A cámara 2D

UNIDADE 2: A cámara 2D. Cámara ortográfica.

1.1 Sumario

1.2 A cámara 2D



1.2.1 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 **unproject** e **project**.

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

1.2.2 Cámara ortográfica

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

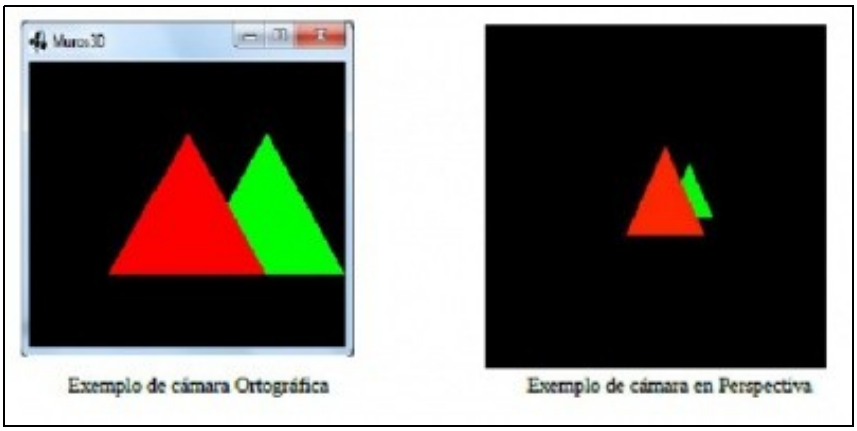
- Deriva da clase Camera

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.

Nota: A partires de agora podemos utilizar tipos de datos Vector2. Tedes unha descrición neste enlace.

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

Por exemplo:



- 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:

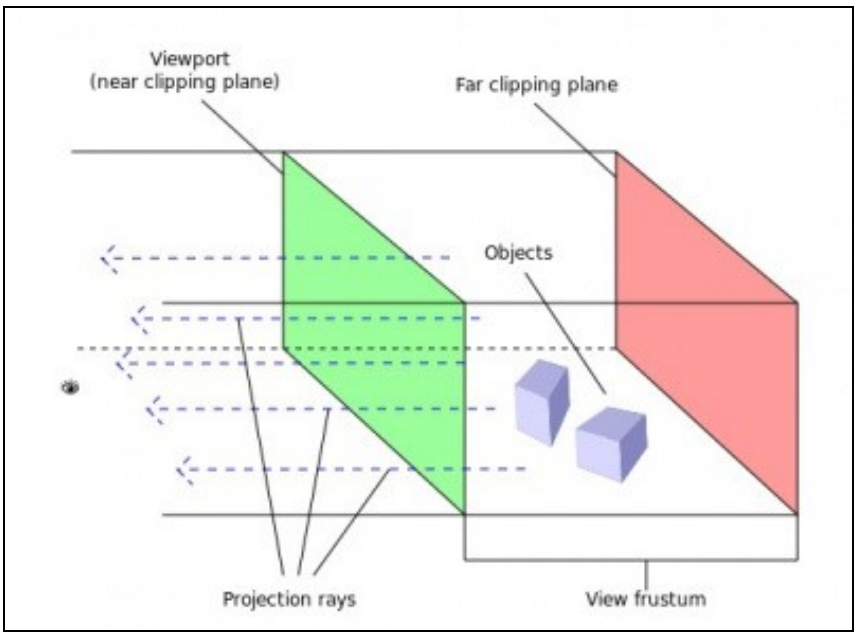
- **public void setToOrtho(boolean yDown,float viewportWidth, float viewportHeight)**
  Define o tamaño do viewport da cámara.
  - ◊ ydown: indica se o punto (0,0) está situado na parte superior esquerda (valor true) ou na parte inferior esquerda (valor false)
  - ◊ viewportWidth: ancho do viewport.
  - ◊ viewportHeight: alto do viewport.
- **public void update()**
  Actualiza a matriz de proxección e de modelado.
- **public Vector3 project(Vector3 worldCoords)**
  Pasa as coordenadas do mundo a coordenadas da pantalla.
  - ◊ worldCoords: coordenadas do mundo.
- **public Vector3 unproject(Vector3 ScreenCoords)**
  Pasa as coordenadas de pantalla a coordenadas do noso mundo.
  - ◊ screenCoords: coordenadas de pantalla.

Definiremos por tanto a cámara no noso xogo.

Código da clase RendererXogo:

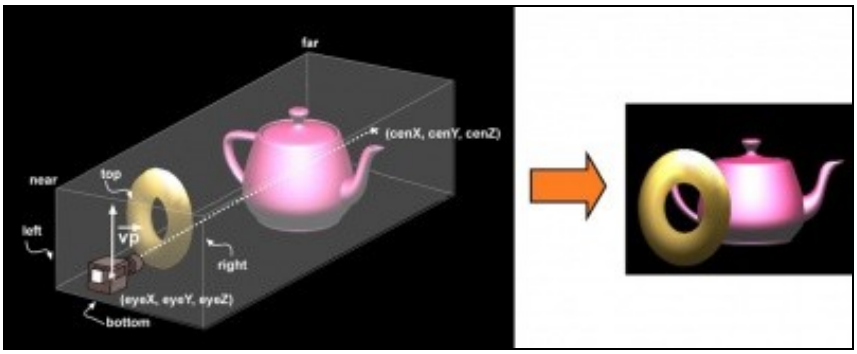
```
public class RendererXogo {

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.



Imaxe obtida dos apuntes de Cristina Cañero Morales

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.

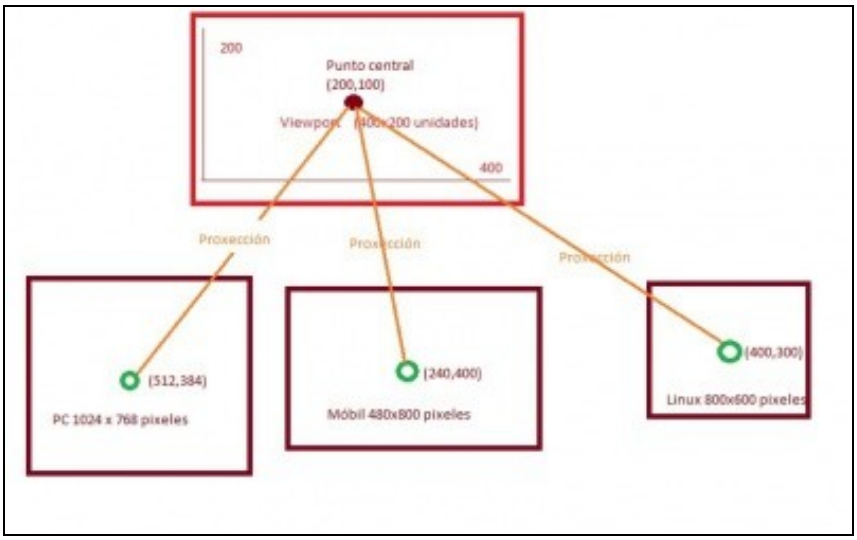
1.2.3 O tamaño da pantalla. A relación de aspecto

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)

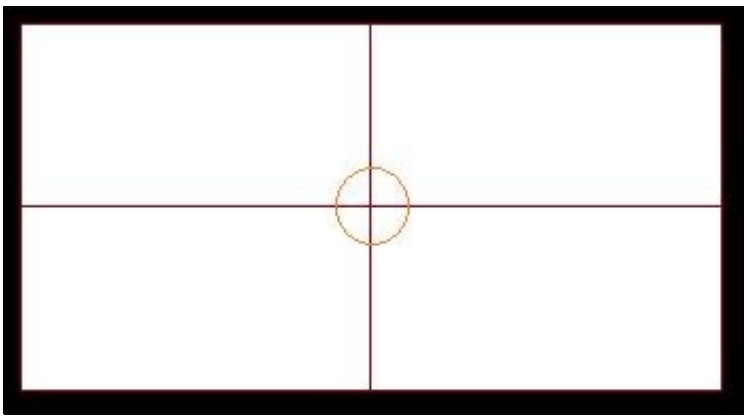
  Se trasladamos estes puntos a unha resolución de 600x600 pixeles:

  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...

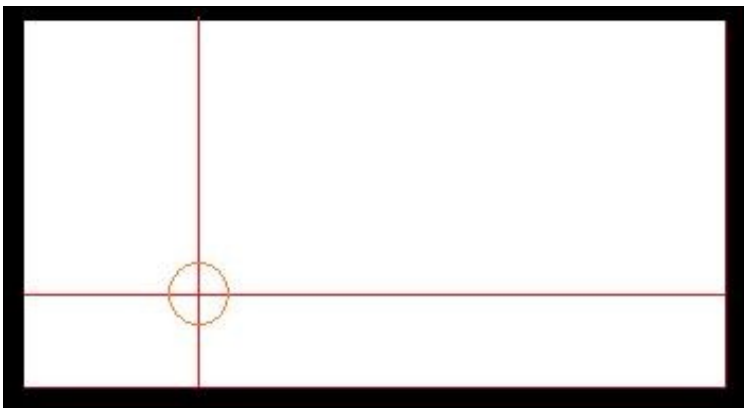
Outro problema que xurde se cambiamos o tamaño do viewport en función do tamaño da resolución é o seguinte:

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:

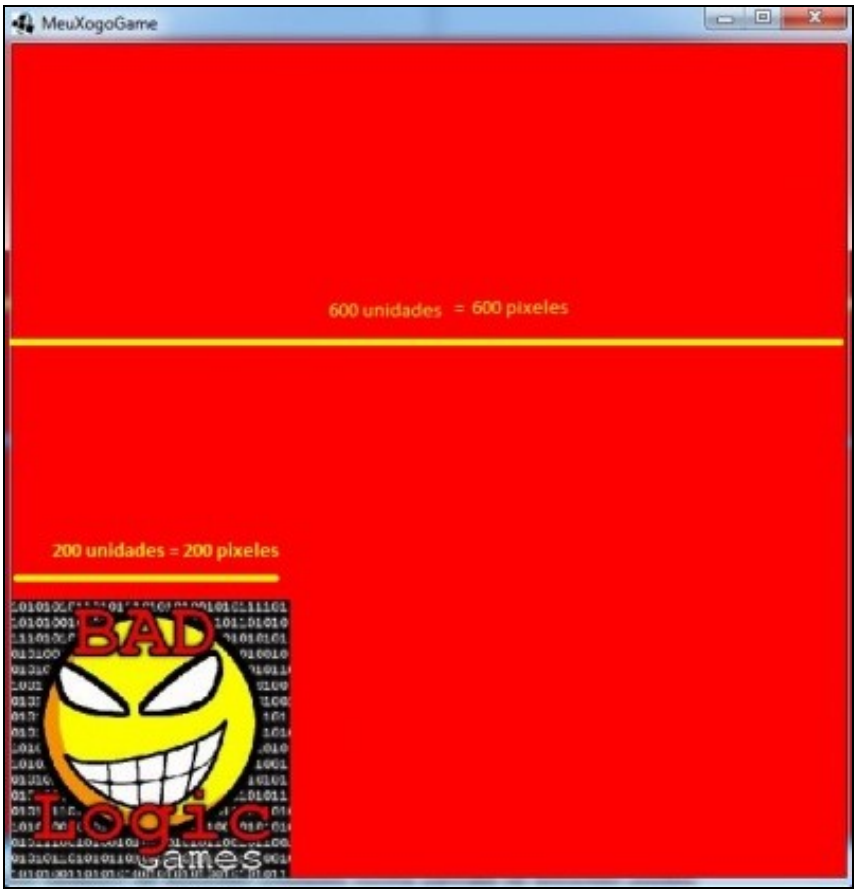


Gráfico de tamaño 200x200 unidades nun viewport de 600x600 unidades nunha pantalla de 600x600 pixeles
Fixarse como o ancho do gráfico ocupa unha terceira parte do ancho total....

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.

- **IMPORTANTE:** 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 desenvolver un xogo para unha resolución de 600x1000 pixeles. Isto da unha relación de aspecto de 600/1000 = 0,6. Como non quero crear un mundo tan grande, utilizo un ancho e alto máis pequeno pero coa mesma relación de aspecto, por exemplo 300x500 unidades (300/500=0,6).

Imos poñer un tamaño de 300x500 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=300;
    public static final int TAMANO_MUNDO_ALTO=500;
}
```

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();


    }
```

TAREFA 2.2 A FACER: Esta parte está asociada á realización dunha tarefa.

Fixarse que para debuxar algo que ocupe todo a pantalla necesitaremos 300x500 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 600 pixeles e un alto de 1000 pixeles (vos facédeo segundo a tarefa).

1.2.4 Movendo a cámara

Preparación: Agora ides facer unha copia da clase RendererXogo, xa que imos modificala para 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. Isto se fai modificando as liñas indicadas:

Código da clase PantallaXogo:
Obxectivo:Facer que se visualice o render da clase UD2_1_RendererXogo.

```
public class PantallaXogo implements Screen {

private MeuXogoGame meuxogogame;
private UD2_1_RendererXogo rendererxogo;

public PantallaXogo(MeuXogoGame meuxogogame) {

this.meuxogogame = meuxogogame;
rendererxogo = new UD2_1_RendererXogo();
}
```

Nota: Lembrar de premer a combinación de teclas Control+Shift+O para importar a 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:
Obxectivo: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 UD2_1_RendererXogo {

private Texture grafico;
private SpriteBatch spritebatch;

private OrthographicCamera camara2d;
```

```
public UD2_1_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,camara2d.viewportWidth/2,camara2d.viewportHeight/2,50,50);
spritebatch.end();

    }

    public void resize(int width, int height) {

        camara2d.setToOrtho(false,Mundo.TAMANO_MUNDO_ANCHO,Mundo.TAMANO_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 liña que debuxa a gráfico (liñá 35) chama o método draw do spritebach e recibe catro parámetros: posición x, posición y, tamaño y e tamaño y.

A cámara está situada por defecto no punto medio do viewport. No exemplo estaría no punto (150x250) que é onde estamos a colocar o gráfico (Texture).

Se executamos a versión desktop aparecerá o seguinte:

Viewport de 300x500 unidades para manter a relación de aspecto dun dispositivo móbil de 480x800 pixeles.

Para movela dispoñemos dos seguinte métodos:

- Método translate:
- **public void translate(Vector2 vec)**
- **public void translate(float x,float y)**

Traslada a cámara á posición indicada por x,y. A traslada sumando ó valor da súa posición o valor indicado.

- Método position:
- Devolve un vector3 co que podemos indicar a nova posición da cámara. Debemos de chamar ó método **set** da clase Vector3 para asinarlle un novo valor. A posición z será 0 normalmente.

- Método rotate:
- Rota a cámara o ángulo indicado no vector de dirección. Varias aclaracións:
    - ◊ O ángulo se lle suma ó que xa ten. Quere isto dicir que se o ángulo de cámara é de 90 grados e chamamos a dito método cun valor de 5. Cada vez que o chamemos sumará 5 ó valor anterior. No exemplo sería 95 grados.
    - ◊ A rotación a fai sobre o eixe que ten o vector dirección da cámara. O vector dirección é un vector que indica cara a onde está apuntando a cámara. Por defecto **coincide co eixe z** do sistema de coordenadas (x,y,z). É o mesmo eixe que se corresponde cos teus ollos mirando a pantalla. Se imaxinas unha liña que atravesa a túa cabeza e vai cara ó computador, se xiras a cabeza no eixe indicado terás o seguinte efecto:

Código da clase UD2_1_RendererXogo:
Exemplo de código para rotar a cámara.

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

camara2d.rotate(1);
camara2d.update();
spritebatch.setProjectionMatrix(camara2d.combined);

spritebatch.begin();
```

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

    }
```

Veremos os eixes de dirección máis adiante en profundidade.

Nota: Lembrade que sempre hai que chamar ó método update cando se faga unha modificación.

O código dos métodos anteriores pódese poñer:

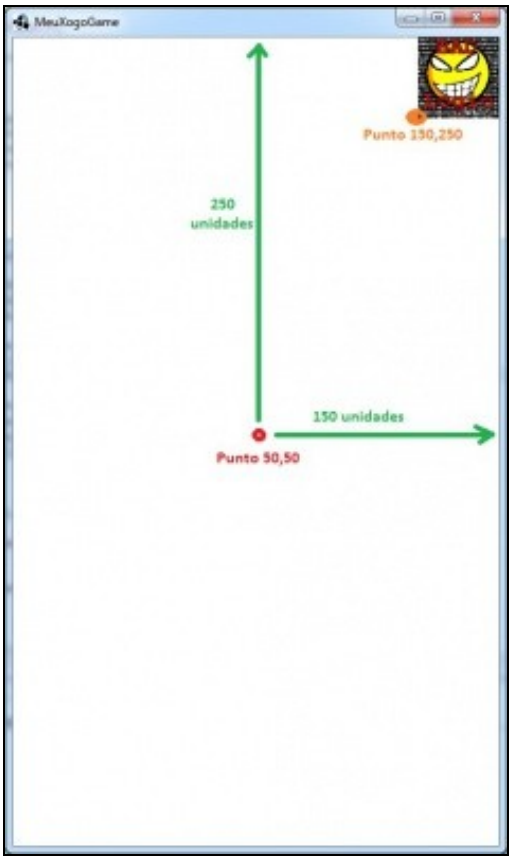
- No **método resize** se a cámara non se move durante o xogo e queremos darlle unha posición inicial diferente á predeterminada.
- No **método render** despois de borrar a pantalla e antes de debuxar os gráficos. Se o facemos neste punto, temos que informarlle ó obxecto que debuxa (no exemplo ten de 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 setProjectionMatrix da clase SpriteBatch.

Imos facelo no método render:

Código da clase UD2_1_RendererXogo:
Exemplo de código para posicionar a cámara.

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

camara2d.position.set(50,50,0);
camara2d.update();
                spritebatch.setProjectionMatrix(camara2d.combined);

spritebatch.begin();
spritebatch.draw(grafico,Mundo.TAMANO_MUNDO_ANCHO/2,Mundo.TAMANO_MUNDO_ALTO/2,50,50);
spritebatch.end();

    }
```

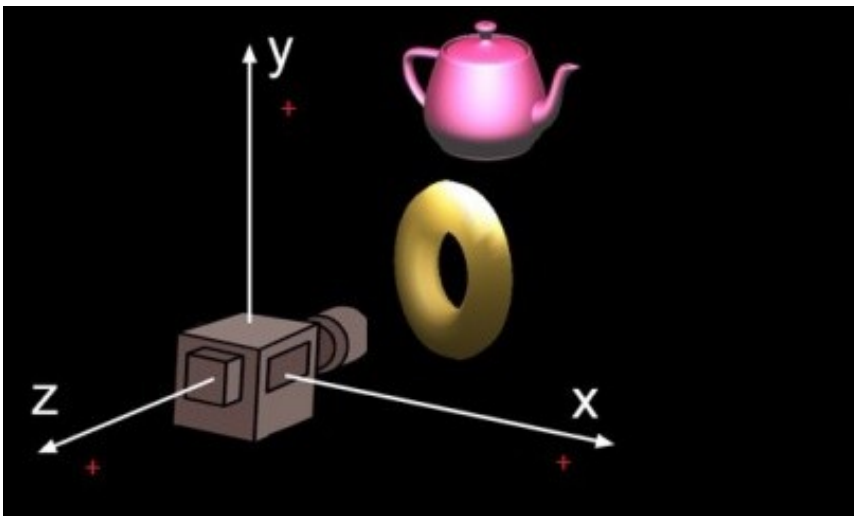
O efecto é o seguinte:



Cámara movida á posición 50,50 nun dispositivo de 400x800 pixeles nun mundo de 300x500 unidades. Fixarse que movemos a cámara pero o gráfico está na mesma posición que antes.

1.2.5 Facendo zoom

Como comentamos antes, a cámara ten un vector dirección que coincide no eixe Z do espazo. En dito eixe, os valores positivos son os que van cara a cámara e os valores negativos son os que van cara a onde apunta á cámara.



Amosa cales son os valores positivos dos eixes x,y,z

Imaxe obtida dos apuntes de Cristina Cañero Morales

Podemos facer zoom na cámara. Se poñemos un valor positivo nos afastaríamos do obxecto e cun valor negativo nos achegaríamos ó obxecto.

Indicar que a cámara, a parte de ter un vector dirección (cara onde se dirixe a cámara) ten un método (**lookat**) que indica cara onde está mirando a cámara. Por iso, no seguinte exemplo, se facedes que o zoom aumente, cando pase da imaxe dará a volta a cámara....

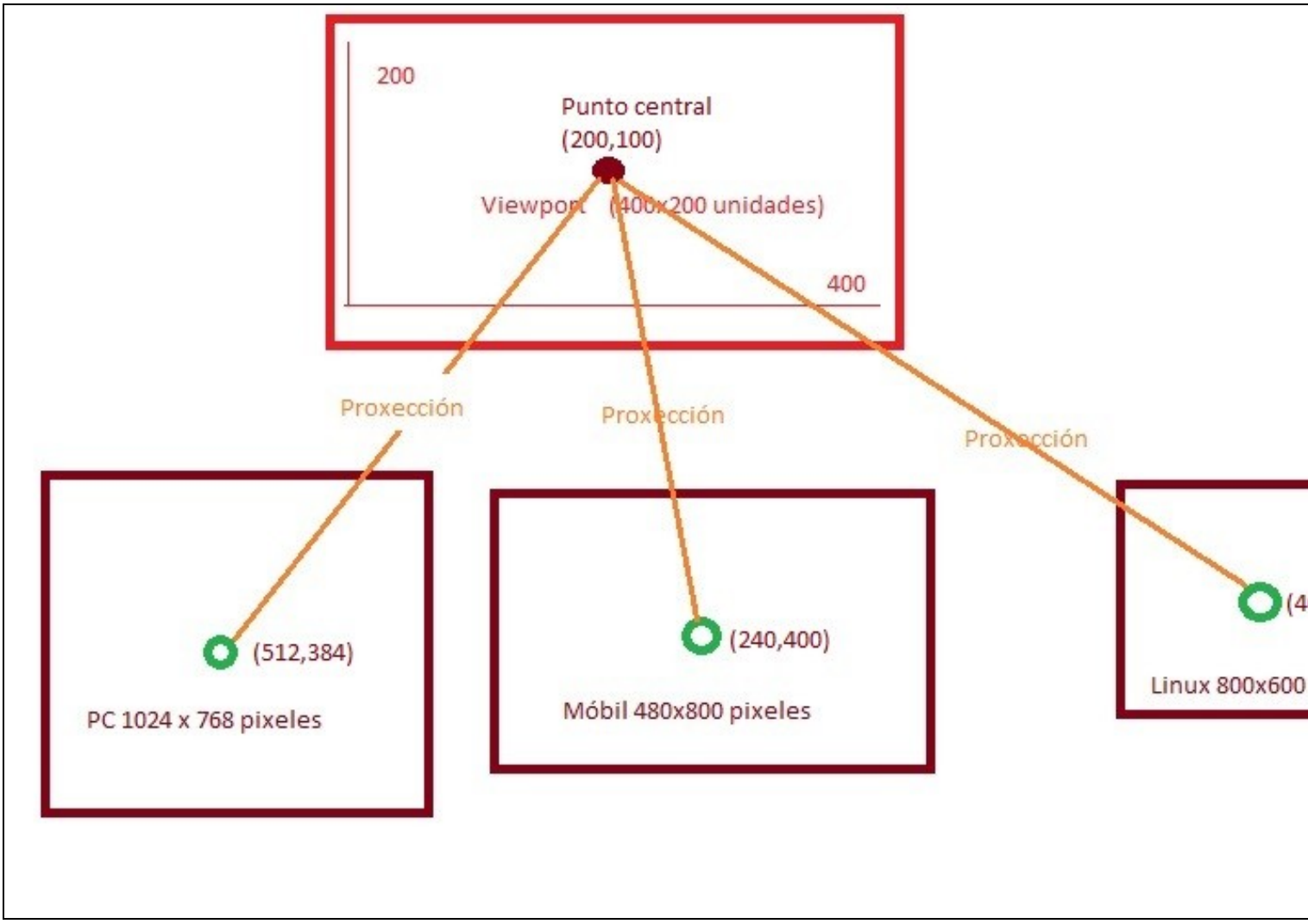
Código da clase UD2_1_RendererXogo:
Exemplo de código para facer zoom coa cámara.

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

camara2d.zoom+=0.01f;   // Neste caso se afasta. Se restamos se achegaría.
camara2d.update();
                spritebatch.setProjectionMatrix(camara2d.combined);

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

    }
```

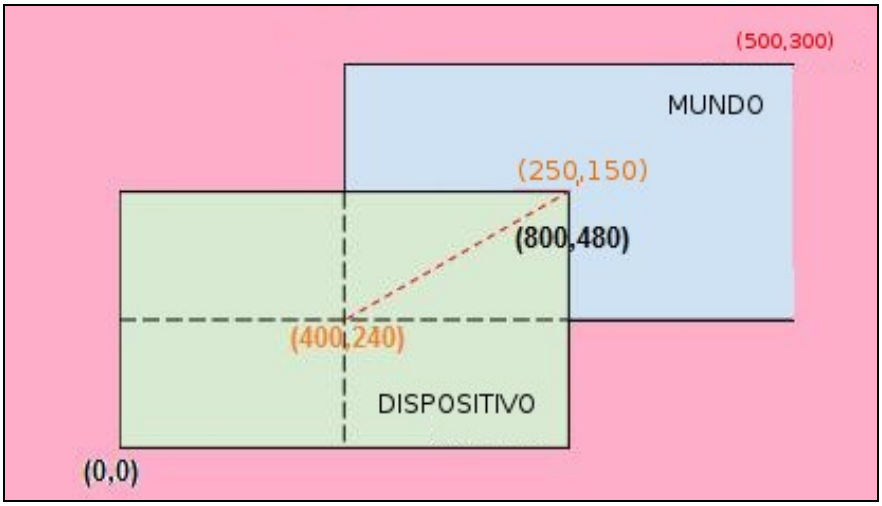
Nota: Unha vez visto os principais métodos da cámara imos volver a usar a clase RendererXogo, cambiando a clase PantallaXogo para que chame a RendererXogo como tiñamos antes.

1.2.6 Proxectando os puntos

Como comentamos anteriormente unha das cousas que podemos facer coa cámara é pasar coordenadas de puntos dende o dispositivo real a coordenadas da cámara (o noso mundo) e viceversa.



Nesta pantalla o noso mundo mide 400x200 unidades. Se prememos sobre a pantalla do dispositivo daranos as coordenadas do mesmo. Teremos que pasar dita coordenadas ó sistema de coordenadas da cámara. Tamén podemos facer o proceso contrario
Outro exemplo:



Outro exemplo de proxección. O dispositivo ten unha resolución de 800x480 pixeles e o mundo 500x300 unidades

Para facelo debemos usar os métodos:

- **unproject**:

  Transforma un Vector dato dende o sistema de coordenadas da pantalla do dispositivo ó sistema de coordenadas da cámara. Normalmente será este método o que utilicemos para interaccionar coas nosas personaxes.

- **project**.

  Fai o proceso contrario. Pasa do sistema de coordenadas da cámara ó sistema de coordenadas da pantalla do dispositivo.

Preparación: Agora ides facer unha copia da clase UD2_1_RendererXogo, xa que imos ver como facer o project e unproject. 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_2_RendererXogo**. Modificade a pantalla PantallaXogo para que chame a esta clase.

O problema que temos é que para facer esta práctica e podamos pasar dun sistema de coordenadas a outro temos que controlar cando prememos sobre a pantalla do dispositivo.

Isto o facemos cunha interface. A **interface InputProcessor**.

O veremos máis adiante neste curso. Agora copiade o seguinte código e quedádevos con a idea de que existe unha interface que incorpora uns métodos. Un destes métodos é facer 'click' sobre a pantalla (método touchDown).

O proceso para incorporar unha interface é moi sinxelo.

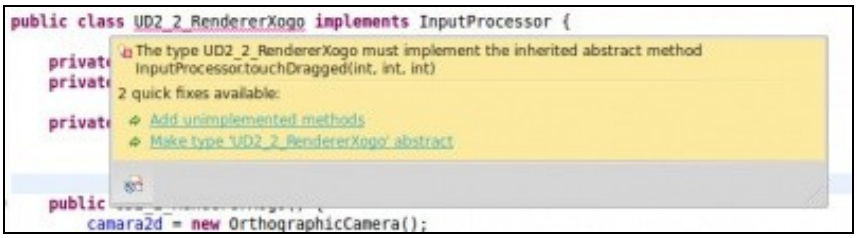
Primeiro escribimos este código:

Código da clase UD2_2_RendererXogo:

```
public class UD2_2_RendererXogo implements InputProcessor {
```

Teredes que importar a nova clase (teclas Control + Shift + O).

Unha vez importada vos dará un erro. Situar o cursor enriba da clase que da erro como amosa a seguinte imaxe:



Vos aparecerá unha ventá emerxente no que vos dará a opción de **Add unimplemented methods**. Ó premer sobre dita opción se engadirán todos os métodos de dita interface entre os que estará o método touchDown:

Código da clase UD2_2_RendererXogo:

Obxectivo: Incorpora a interface InputProcessor.

```
/**
 *  Explica como facer o project e unproject da cámara.
 *   Move o gráfico ó punto indicado ó premer sobre a pantalla.
 */

package com.plategaxogo2d.renderer;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.InputProcessor;
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 UD2_2_RendererXogo implements InputProcessor {

private Texture grafico;
private SpriteBatch spritebatch;

private OrthographicCamera camara2d;



public UD2_2_RendererXogo() {
camara2d = new OrthographicCamera();
grafico = new Texture(Gdx.files.internal("badlogic.jpg"));
spritebatch = new SpriteBatch();
}


    /**
     * Debuxa todos os elementos gr?ficos 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,Mundo.TAMANO_MUNDO_ANCHO/2,Mundo.TAMANO_MUNDO_ALTO/2,50,50);
spritebatch.end();

    }

    public void resize(int width, int height) {

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

        spritebatch.setProjectionMatrix(camara2d.combined);

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

```
    }


@Override
public boolean keyDown(int keycode) {
// TODO Auto-generated method stub
return false;
}


@Override
public boolean keyUp(int keycode) {
// TODO Auto-generated method stub
return false;
}


@Override
public boolean keyTyped(char character) {
// TODO Auto-generated method stub
return false;
}


@Override
public boolean touchDown(int screenX, int screenY, int pointer, int button) {
// TODO Auto-generated method stub
return false;
}


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


@Override
public boolean touchDragged(int screenX, int screenY, int pointer) {
// TODO Auto-generated method stub
return false;
}


@Override
public boolean mouseMoved(int screenX, int screenY) {
// TODO Auto-generated method stub
return false;
}


@Override
public boolean scrolled(int amount) {
// TODO Auto-generated method stub
return false;
}
}
```

A maiores, para que funcione a xestión de eventos temos que chamar ó método:

setInputProcessor e indicarlle a clase que vai xestionar os eventos.

Cando saiamos da mesma deberemos de quitar dita asignación, como facemos no seguinte código:

Código da clase UD2_2_RendererXogo:
Obxectivo: Indicar que clase xestiona a InputProcessor.

```
public UD2_2_RendererXogo() {
camara2d = new OrthographicCamera();
```

```
grafico = new Texture(Gdx.files.internal("badlogic.jpg"));
spritebatch = new SpriteBatch();

Gdx.input.setInputProcessor(this);
}

public void dispose() {

Gdx.input.setInputProcessor(null);
spritebatch.dispose();
grafico.dispose();
}
```

Nota:: Lembrar que estamos a facer isto para poder explicar o project e unproject da cámara. Xa explicaremos en profundidade a xestión de eventos.

- Analicemos o método touchdown:

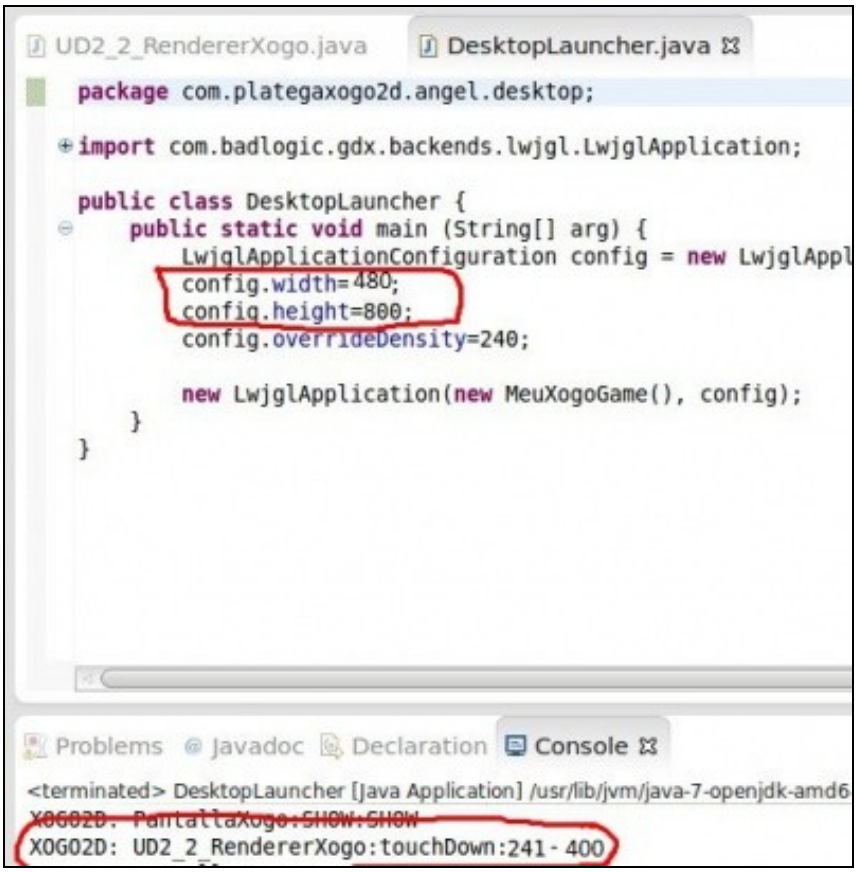
```
@Override
public boolean touchDown(int screenX, int screenY, int pointer, int button) {
// TODO Auto-generated method stub
return false;
}
```

Entre os seus parámetros veñen as coordenadas x,y do dispositivo **EN PIXELES DE PANTALLA**.

Podemos comprobalo:

Código da clase UD2_2_RendererXogo:
Obxectivo: Imprimir en pixeles o punto onde se preme na pantalla.

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

Utiles.imprimirLog("UD2_2_RendererXogo", "touchDown", String.valueOf(screenX)+"-"+String.valueOf(screenY));
return false;
}
```



1.2.6.1 Realizar un unproject

Imos ver como pasar as coordenadas en pixeles a coordenadas do noso mundo (que son as coordenadas do viewport da cámara). Lembrar que ditas coordenadas as ten definidas a cámara no seu viewport e se corresponden coas constantes de clase definidas na clase Mundo.

Código da clase UD2_2_RendererXogo:
Obxectivo: Pasar das coordenadas reais ás coordenadas da cámara.

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

Utiles.imprimirLog("UD2_2_RendererXogo", "touchDown", String.valueOf(screenX)+"-"+String.valueOf(screenY));

Vector3 temporal = new Vector3(screenX,screenY,0);
camara2d.unproject(temporal);
Utiles.imprimirLog("UD2_2_RendererXogo", "touchDown", "COORDENADAS MUNDO:" + String.valueOf(temporal));


return false;
}
```

Algunhas aclaracións:

- O vector temporal leva como valor no eixe z un 0. Isto fai que a proxección se faga sobre o plano near da cámara. Se levase o valor 1 se faría sobre o plano far.
- Ó chamar ó método unproject modifícase o vector temporal.
- O método toma como punto 0,0 a esquina inferior esquerda.
- Por defecto establece como ancho e alto do dispositivo o total da pantalla (no exemplo 800x480). Pero podemos modificar isto se ides o enlace do método unproject.

Lembra que podemos aplicar o consello de programación neste caso sobre o vector temporal.

1.2.6.2 Realizar un project

Imos ver como pasar as coordenadas do noso mundo a coordenadas en pixeles.

Supoñamos que movemos a imaxe á coordenada 100,400 do noso mundo e queremos saber a que coordenada se corresponde do noso dispositivo / pantalla en pixeles. Lembrar que no exemplo estamos traballando nunha **resolución de 480x800** pixeles.

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

temporal.set(100,400,0);
camara2d.unproject(temporal);
Utiles.imprimirLog("UD2_2_RendererXogo", "touchDown", "COORDENADAS MUNDO:" + String.valueOf(temporal));

return false;
}
```

Resultado: 160,640

Para comprobalo podemos facer unha regra de tres:

> Se 300 unidades son 480 pixeles
> 100 unidades son x
>
> x = 48000/300 = 160

E o mesmo para o y.

Algunhas aclaracións:

- Ó chamar ó método project modifícase o vector temporal.
- O método toma como punto 0,0 a esquina inferior esquerda.
- Por defecto establece como ancho e alto do dispositivo o total da pantalla (no exemplo 800x480). Pero podemos modificar isto se ides o enlace do método project.

1.2.6.3 Exemplo: Movendo o gráfico

O aprendido ata o de agora non nos deixar facer moitas cousas...

Isto vai ser unha parte a utilizar posteriormente no desenvolvemento do xogo.

Agora a modo de exemplo imos facer que o gráfico que aparece en pantalla se mova ó lugar onde prememos na pantalla.

Lembrar que a xestión de eventos non a imos facer aquí, como será explicado posteriormente.

Este punto trataba de coñecer como pasar dun sistema de coordenadas a outro.

Exercicio proposto: Aquí vos deixo a solución.

Código da clase UD2_2_RendererXogo:
Obxectivo: Mover o gráfico ó punto indicado.

```
/**
 *  Explica cales son os métodos que temos para mover a cámara.
 *   Carga un gráfico sinxelo no centro da pantalla
 */

package com.plategaxogo2d.renderer;
```

```
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.InputProcessor;
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.badlogic.gdx.math.Vector3;
import com.plategaxogo2d.angel.Utiles;
import com.plategaxogo2d.modelo.Mundo;

public class UD2_2_RendererXogo implements InputProcessor {

private Texture grafico;
private SpriteBatch spritebatch;

private OrthographicCamera camara2d;

private Vector3 temporal;

public UD2_2_RendererXogo() {
camara2d = new OrthographicCamera();
grafico = new Texture(Gdx.files.internal("badlogic.jpg"));
spritebatch = new SpriteBatch();

temporal = new Vector3();

Gdx.input.setInputProcessor(this);
}

/**
         * Debuxa todos os elementos gr?ficos 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, temporal.x,temporal.y, 50, 50);
spritebatch.end();

}

public void resize(int width, int height) {

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

spritebatch.setProjectionMatrix(camara2d.combined);

}

public void dispose() {

Gdx.input.setInputProcessor(null);
spritebatch.dispose();
grafico.dispose();
}

@Override
public boolean keyDown(int keycode) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean keyUp(int keycode) {
// TODO Auto-generated method stub
return false;
}
```

```
@Override
public boolean keyTyped(char character) {
// TODO Auto-generated method stub
return false;
}

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

temporal.set(screenX,screenY,0);
camara2d.unproject(temporal);

return false;
}

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

@Override
public boolean touchDragged(int screenX, int screenY, int pointer) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean mouseMoved(int screenX, int screenY) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean scrolled(int amount) {
// TODO Auto-generated method stub
return false;
}
}
```

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