LIBGDX Xestion Eventos GestureListener

UNIDADE 3: Xestión de Eventos: GestureListener

Sumario

Introdución

Nota: Esta explicación está relacionada coa sección de Interfaces para capturar eventos.

Información na wiki: https://github.com/libgdx/libgdx/wiki/Gesture-detection

Clases utilizadas:

- GestureListener
- Clase InputMultiplexer

O obxectivo deste punto é ver como podemos capturar outro tipo de eventos diferentes dos que nos permite a interface InputListener xa vista nun punto anterior.

Tamén veremos como nese caso necesitaremos capturar eventos de dúas interfaces diferentes e como temos que facer para que isto sexa posible.

Interface GestureListener

Ata o de agora, para controlar os eventos engadimos a interface InputProcessor, co que controlamos os eventos de pulsar sobre a pantalla.

Pero temos a posibilidade de controlar outro tipo de eventos, coma son os de doble pulsación (evento tap), o clásico movemento con dous dedos para facer un zoom da pantalla (evento zoom)....

Todos estes eventos se atopan noutra interface denominada GestureListener.

- Para usala, temos que implementar dita interface.

```
public class EventosGestureListener extends ApplicationAdapter implements GestureListener{
```

E implentar os métodos que veñen coa interface (situarse enriba da clase e escoller a opción **Add unImplemmented Methods**).

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

@Override
public boolean tap(float x, float y, int count, int button) {
// TODO Auto-generated method stub
return false;
}
```

```
@Override
public boolean longPress(float x, float y) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean fling(float velocityX, float velocityY, int button) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean pan(float x, float y, float deltaX, float deltaY) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean panStop(float x, float y, int pointer, int button) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean zoom(float initialDistance, float distance) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean pinch(Vector2 initialPointer1, Vector2 initialPointer2,
Vector2 pointer1, Vector2 pointer2) {
// TODO Auto-generated method stub
return false;
}
```

- Unha vez a temos e engadidos os métodos da interface á nosa clase, temos que dicirlle á clase que use dita interface.

Có control de eventos anteriores (InputProcessor) facíamos isto no evento show da clase Screen:

```
public void show() {
........
Gdx.input.setInputProcessor(this);
}
```

E no evento hide:

```
@Override
public void hide() {
Gdx.input.setInputProcessor(null);
}
```

Agora cambia por isto:

- Creamos un obxecto da clase GestureDetector:

```
private GestureDetector gd;
```

- No método show creamos dito obxecto, tendo que pasarlle como parámetro un obxecto dunha clase que implemente a interface

GestureListener. No noso caso é a propia pantalla, por iso poñemos this.

Despois facemos coma no caso anterior, pero pasándolle o obxecto GestureDetector.

```
@Override
public void show() {
// TODO Auto-generated method stub
gd = new GestureDetector(this);

Gdx.input.setInputProcessor(gd);
}
```

- O método hide queda igual:

```
@Override
public void hide() {
// TODO Auto-generated method stub
Gdx.input.setInputProcessor(null);
}
```

Unha vez feito isto, xa controlamos os eventos da interface nos respectivos métodos.

Vexamos algúns dos métodos novos.

Exemplo de código

Deberedes de cambiar a clase co que inician as diferentes plataformas pola seguinte:

- Deberedes copiar o gráfico seguinte ó cartafol assets do proxecto Android:

- Crear unha nova clase á que chamen as diferentes versións.

Código da clase EventosGestureListener
Obxectivo: Amosar como funciona a interface GestureListener.

```
import com.badlogic.gdx.ApplicationAdapter;
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.badlogic.gdx.input.GestureDetector.GestureListener;
import com.badlogic.gdx.math.Vector2;

public class EventosGestureListener extends ApplicationAdapter implements GestureListener{
private SpriteBatch batch;
private Texture img;
private OrthographicCamera _camera;

private float ANCHO_MUNDO_METROS = 100;
private float ALTO_MUNDO_METROS = 100;

@Override
public void create () {
batch = new SpriteBatch();
img = new Texture("LIBGDX_fondoscroll.png");

_camera = new OrthographicCamera();
_camera.setToOrtho(false, 15, 15);
_camera.update();

batch.setProjectionMatrix(_camera.combined);

}

@Override
public void render() {
Gdx.gl.glClearColor(1, 0, 0, 1);
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
batch.begin();
batch.draw(img,
               -ANCHO_MUNDO_METROS/2f,
               -ALTO_MUNDO_METROS/2f,
               ANCHO_MUNDO_METROS,
               ALTO_MUNDO_METROS);
batch.end();

}

@Override
public void dispose() {
img.dispose();
batch.dispose();
}

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

@Override
public boolean tap(float x, float y, int count, int button) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean longPress(float x, float y) {
// TODO Auto-generated method stub
return false;
}
```

```
@Override
public boolean fling(float velocityX, float velocityY, int button) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean pan(float x, float y, float deltaX, float deltaY) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean panStop(float x, float y, int pointer, int button) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean zoom(float initialDistance, float distance) {
// TODO Auto-generated method stub
return false;
}

@Override
public boolean pinch(Vector2 initialPointer1, Vector2 initialPointer2,
Vector2 pointer1, Vector2 pointer2) {
// TODO Auto-generated method stub
return false;
}


}
```

Se executamos teremos isto:

Analicemos parte do código:

```
_camera = new OrthographicCamera();
_camera.setToOrtho(false, 15, 15);
_camera.update();
```

No método setToOrtho o primeiro parámetro indica se a coordenada 'Y' (posición 0) comeza na parte de arriba (valor true) ou se empeza na parte de abaixo (valor false):

O mesmo tempo que lle damos un tamaño á cámara, a estamos posicionando na coordenada 15/2 e 15/2 = (7.5,7.5). A esta coordenada apunta o centro da cámara.

Se queremos posicionala noutro punto do noso mundo, teríamos que utilizar a propiedade 'position'

_camera.position.x = valor _camera.position.y = valor _camera.position.z = valor

O que facemos despois e renderizar a textura carga previamente. Lembrar que o tamaño do mundo é de 100x100 unidades.

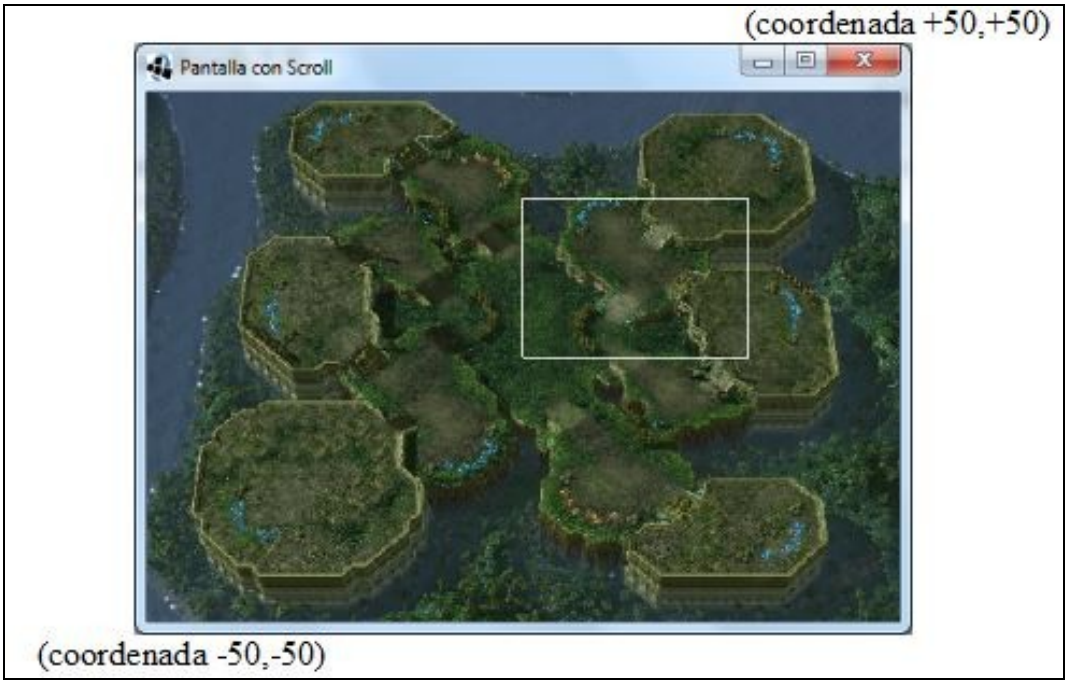
```
@Override
public void render() {
Gdx.gl.glClearColor(1, 0, 0, 1);
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
batch.begin();
batch.draw(img,
               -ANCHO_MUNDO_METROS/2f,
               -ALTO_MUNDO_METROS/2f,
               ANCHO_MUNDO_METROS,
               ALTO_MUNDO_METROS);
batch.end();

}
```

Polo tanto debuxamos TODO O FONDO (de tamaño 1024x1024) dentro dun rectángulo de tamaño 100x100, empezando nas coordenadas (-50,-50). Fixarse que as coordenadas son negativas.

O que aquí amosamos é a visión da cámara, por tanto as coordenadas son as da cámara:

So usamos o obxecto ShapeRenderer para visualizar un rectángulo que represente a cámara, facendo que o fondo ocupe toda a pantalla, teríamos este resultado:



Nota: Como podemos observar, podemos modificar o 'zoom' da cámara en función do tamaño do seu viewport, anque isto tamén o podemos cambiar chamando o método zoom da cámara.

IDEA: O ter un tamaño fixo do viewport da cámara (15x15) vai suceder que cando aumente a resolución, esta siga sendo 15x15, chegando a ter un aumento demasiado grande.

Unha forma de evitar isto é agrandando o tamaño da cámara en función da resolución. Para isto imos crear unha constante nova na interface de constantes:

```
float PIXELS_PER_METER = 32;
```

Esta constante vai determinar cantos píxeles hai dentro dunha das unidades creadas por nos (lembrade que o noso mundo é de 100x100 unidades=metros por dicir algo).

De tal forma que se a resolución aumenta, tamén aumentará o tamaño do viewport se poñemos isto:

```
@Override
public void resize(int width, int height) {
// TODO Auto-generated method stub
   _camera.setToOrtho(false,
                      width/PIXELS_PER_METER,
                      height/PIXELS_PER_METER);
   _camera.update();
}
```

O facer isto teríamos o seguinte en diferentes resolucións (o viewport aumenta o tamaño para ter un zoom menor).

Nota: tamaño de pantalla de 480x320 pixeles en versión Desktop


Nota: tamaño de pantalla de 1280x800 pixeles en versión Desktop

- **Método tap:** Para facer que cando o usuario preme dúas veces a pantalla, a cámara se mova a posición indicada.

Xa temos engadidos os métodos da interface no paso inicial.

```
@Override
public boolean tap(float x, float y, int count, int button) {
// TODO Auto-generated method stub

if (count!=2) return false;

return false;
}
```

Co return false indicamos que o evento sexa enviado pola xerarquñía de elementos gráficos por se queremos capturar dito evento. Se poñemos true xa non se envía a ningún outro elemento. Veremos despois cando xestionamos máis dunha interface para que usalo.

O parámetro count sirve para saber cantas veces se pulsa o pantalla de forma seguida. Se pode configurar o tempo que hai que pasar entre pulsacións para que 'sume' as pulsacións e o tamaño do cadrado no que se considera que está pulsando na mesma área. Isto se fai no evento show configurando o obxecto GestureDetection:

```
gd.setTapCountInterval(x)
gd.setTapSquareSize(x)
```

Seguimos co método tap.

- Cando pulsamos na pantalla, o que nos devolve é a posición real da pantalla en píxeles. Pero nos queremos cambiar estes valores polos valores do noso mundo ficticio (vai dende -50x-50 a 50x50).

Para obtelas temos que facer uso do método unproyect do obxecto cámara.

```
@Override
public boolean tap(float x, float y, int count, int button) {
// TODO Auto-generated method stub

if(count==2){
Vector3 coordreais = new Vector3(x,y,0);
_camera.unproject(coordreais);
}

return false;
}
```

- Agora dentro do método tap calculamos a distancia:

```
private Vector3 distanciaCamara = new Vector3(0f,0f,0f);  //Distancia da cámara ata o dedo
@Override
public boolean tap(float x, float y, int count, int button) {
// TODO Auto-generated method stub

if(count==2){
Vector3 coordreais = new Vector3(x,y,0);
_camera.unproject(coordreais);

Vector3 poscam = _camera.position.cpy();
distanciaCamara = poscam.sub(coordreais);

}

return false;
}
```

- Agora necesitamos mover a cámara.

Definimos a nivel global un Vector2 para asinarlle a velocidade:

```
private Vector2 movementoCamara = new Vector2(0f,0f);  // Velocidade da cámara
```

Para poder modificar a velocidade de cámara cambiando o valor dunha constante definimos a nivel global a velocidade da mesma:

```
private final Vector3 VELOCIDADE_CAMARA = new Vector3(0.1f,0.1f,0f);
```

- Volvemos agora ó método tap para asinar a velocidade:

```
@Override
public boolean tap(float x, float y, int count, int button) {
// TODO Auto-generated method stub

if(count==2){
Vector3 coordreais = new Vector3(x,y,0);
_camera.unproject(coordreais);

Vector3 poscam = _camera.position.cpy();
distanciaCamara = poscam.sub(coordreais);

if (distanciaCamara.x>0) {  // Pulsado dedo na parte esquerda da pantalla dende o centro
movementoCamara.x=-VELOCIDADE_CAMARA.x;
}
else {
movementoCamara.x=VELOCIDADE_CAMARA.x;
}
if (movementoCamara.y>0) {
movementoCamara.y=-VELOCIDADE_CAMARA.y;
}
else {
movementoCamara.y=VELOCIDADE_CAMARA.y;
}
}

return false;
}
```

- Pero claro, con isto facemos que a cámara se mova, pero non vai parar. É necesario gardar a distancia que ten que percorrer.

```
@Override
public boolean tap(float x, float y, int count, int button) {
// TODO Auto-generated method stub

if(count==2){
Vector3 coordreais = new Vector3(x,y,0);
_camera.unproject(coordreais);

Vector3 poscam = _camera.position.cpy();
distanciaCamara = poscam.sub(coordreais);

if (distanciaCamara.x>0) {  // Pulsado dedo na parte esquerda da pantalla dende o centro
movementoCamara.x=-VELOCIDADE_CAMARA.x;
}
else {
movementoCamara.x=VELOCIDADE_CAMARA.x;
}
if (distanciaCamara.y>0) {
movementoCamara.y=-VELOCIDADE_CAMARA.y;
}
else {
movementoCamara.y=VELOCIDADE_CAMARA.y;
}

distanciaCamara.x = Math.abs(distanciaCamara.x);
distanciaCamara.y = Math.abs(distanciaCamara.y);
}
return false;
}
```

- Agora no método render temos que chamar a un método para que mova a cámara e actualice:

```
private void actualizarCamara(){

}

@Override
public void render() {
Gdx.gl.glClearColor(1, 0, 0, 1);
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

actualizarCamara();

batch.begin();
batch.draw(img,
               -ANCHO_MUNDO_METROS/2f,
               -ALTO_MUNDO_METROS/2f,
               ANCHO_MUNDO_METROS,
               ALTO_MUNDO_METROS);
batch.end();

}
```

- Para mover a cámara usamos o método translate da cámara, que move a cámara o nº de unidades indicadas. Imos diminuíndo a distancia en cada iteración, restando á distancia á velocidade da cámara.

```
private void actualizarCamara(){
if ((distanciaCamara.x>0) || (distanciaCamara.y>0)){
distanciaCamara.sub(VELOCIDADE_CAMARA);
if (distanciaCamara.x <=0)// Paramos o movemento
movementoCamara.x=0;
if (distanciaCamara.y <=0) // Paramos o movemento
movementoCamara.y=0;

_camera.translate(movementoCamara);
_camera.update();
batch.setProjectionMatrix(_camera.combined);
}
}
```

- **Método zoom:** Facer que a cámara se achegue ou afaste en función de se facemos na pantalla o movemento de abrir ou pechar dous dedos sobre a pantalla.

```
@Override
public boolean zoom(float initialDistance, float distance) {
// TODO Auto-generated method stub

if (initialDistance > distance)
_camera.zoom+=0.05f;
else
_camera.zoom-=0.05f;

if (_camera.zoom>5) _camera.zoom=5;
if (_camera.zoom<1) _camera.zoom=1;

_camera.update();
batch.setProjectionMatrix(_camera.combined);

return false;
}
```

Xestionando múltiples interfaces de eventos

- Clase InputMultiplexer.

Pode suceder que necesitemos xestionar máis dunha interface de eventos. Veremos na sección de 3D avanzada que temos unha interface para xestionar a cámara en 3D, e do que levamos visto ata o de agora tamén temos a interface GestureListener e InputProcessor.

O proceso é o seguinte:

- Modificamos a clase do exemplo para incorporar a interface InputProcessor:

```
public class EventosInputMultiplexer extends ApplicationAdapter implements GestureListener, InputProcessor{
```

- Teremos dous métodos touchDown, un de cada Interface. Amosaremos unha mensaxe en cada un deles:

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

Gdx.app.log("MENSAXES","TOUCH DOWN DE GESTURELISTENER");
return false;
}

        ..........
@Override
public boolean touchDown(int screenX, int screenY, int pointer, int button) {
// TODO Auto-generated method stub
Gdx.app.log("MENSAXES","TOUCH DOWN DE InputProcessor");
return false;
}
```

- Creamos un obxecto da clase InputMultiplexor e o instanciaremos no constructor:

```
inputMultiplexer = new InputMultiplexer();
```

- Agora chamaremos ó método addProcessor para engadir **EN ORDEN** as diferentes interfaces. A orde é importante xa que primeiro irá os eventos da interface engadida en primeiro lugar:

```
gd = new GestureDetector(this);

inputMultiplexer.addProcessor(gd);
inputMultiplexer.addProcessor(this);
```

Fixarse como no exemplo engadimos primeiro a interface GestureListener e despois a InputProcessor.

- Indicamos ó framework que xestiona os eventos:

```
Gdx.input.setInputProcessor(inputMultiplexer);
```

Se agora executades o código e premedes unha vez sobre a pantalla recibiredes os dous avisos:

- Agora é cando podemos xogar con return dos eventos.

Se no primeiro touchDown (o do GestureListener) cambiamos a true, indicaremos que o evento xa non debe ir por máis controis nin por outra interface, polo que só recibiremos o aviso da interface GestureListener:

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

Gdx.app.log("MENSAXES","TOUCH DOWN DE GESTURELISTENER");
return true;
}
```

TAREFA OPTATIVA A FACER

Modificade o xogo para que se poida facer Zoom.

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