libgdx / DAM2
IES Chan do Monte

Contenidos

Introducción

LibGDX es una plataforma de programación de juegos con las siguientes características:

- Open Source
- Multiplataforma
  - Desktop – Windows, Linux, OS X (Mac)
  - HTML5
  - Android
  - iOS

Se trata de una plataforma (framework) de desarrollo y no un motor de juegos (game engine) debido a que éstos suelen traer multitud de herramientas y especifican totalmente el flujo de trabajo. En cambio LibGDX permite libertad en el diseño además de permitir el acceso a llamadas de bajo nivel, en este caso a OpenGL – aunque no suele ser necesario.

Al ser multiplataforma, se apoya en librerías de terceros para generar el código específico:
- LWJGL (Lightweight Java Game Library)
- JOGL (Java OpenGL) – biblioteca que permite acceder a OpenGL desde Java
- GWT: (Google Web Toolkit – permite escribir Java y generar webs con JS + AJAX)
- iOS/RoboVM e iOS/MOE

APIs gráficos 2D y 3D

Para utilizar de una manera sencilla la diversidad de tarjetas gráficas existentes en el mercado, existen especificaciones (APIs genéricas) que abstraen las características concretas de cada hardware y permiten a los programadores utilizarlas. La implementación en bibliotecas de funciones de cada dispositivo queda en manos de su fabricante. Dicha implementación la realizará utilizando las capacidades hardware de su producto, o bien utilizando emulación software.

- OpenGL. Open Graphics Library es una especificación estándar multilenguaje y multiplataforma, escrita por Silicon Graphics Inc. en 1992. Las implementaciones de los fabricantes deben superar unos controles para poder lucir el logotipo oficial de OpenGL.
- OpenGL ES. OpenGL for Embedded Systems es una versión sencilla de OpenGL diseñada para móviles, tabletas, consolas y sistemas integrados en general.
- Vulkan. Sucesor de OpenGL, es un API multiplataforma para desarrollo de aplicaciones con gráficos 3D.
- WebGL. API JavaScript para renderizar gráficos 2D y 3D en un navegador. Basado en OpenGL ES 2.0.
- Direct3D. Creado por Microsoft en 1995 y principal competidor de OpenGL, es un subconjunto de DirectX – API de gestión de multimedia de productos Microsoft – Windows, Xbox, etc.

Muchos de los API's comentados anteriormente, entre otros muchos, son diseñados y mantenidos por la organización sin ánimo de lucro, Khronos Group https://www.khronos.org/

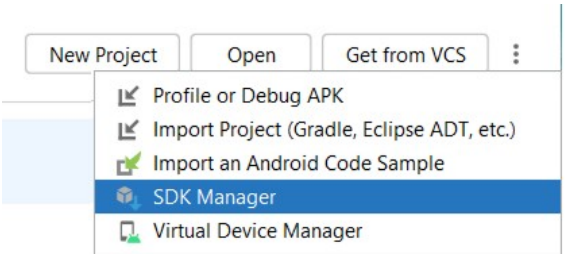
Generación y estructura de un proyecto

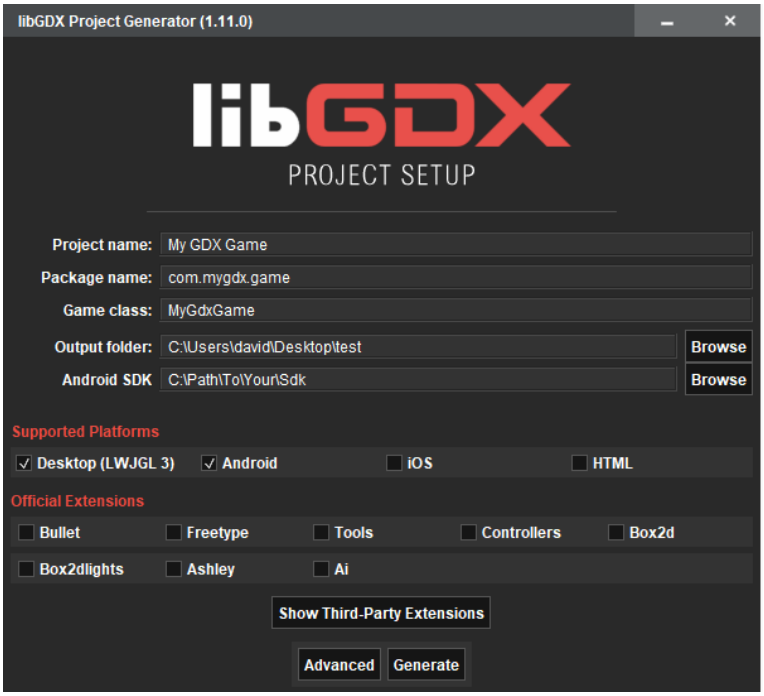
Para generar un proyecto libGDX, y siempre con el Android Studio instalado, bajaremos la herramienta **gdx-setup.jar** de https://libgdx.com/wiki/start/project-generation. No confundir con el generador de archivos antiguo denominado gdx-setup-ui.jar, y que no utilizaba Gradle.

Será necesario especificar los siguientes parámetros:

- nombre del proyecto
- paquete del proyecto
- nombre de la clase que modeliza el juego
- carpeta donde se almacenarán el proyecto y sus archivos
- carpeta donde se encuentra alojado el SDK de Android. Si no conocemos su ubicación exacta, se puede obtener facilmente en la opción SDK Manager de la opción Configure de la ventana inicial del Android Studio (ver imagen)

Se generará un único proyecto con diversos subproyectos (tantos como plataformas seleccionadas: (Desktop, Android, iOS, Html), junto con un proyecto denominado **core,** donde va todo el código del juego.





Es recomendable generar el proyecto Desktop por razones prácticas de prueba del juego, y es obligatorio generar el proyecto android porque la carpeta assets, que almacena los recursos del juego, tienen que estar situados en este proyecto. Esto es debido a que la compilación y generación del archivo R.java de android necesita acceso directo. La carpeta *resources* del proyecto desktop es simplemente un acceso directo.

Se permite instalar extensiones que añaden funcionalidad, por ejemplo:

- Bullet > Motor de colisiones y cuerpos rígidos para 3D
- Box2D > Motor de físicas 2D
- Box2DLights > Extensión para trabajar con iluminación (luces de distintos tipos)
- Freetype > Para dibujar texto generando BitmapFonts de un .ttf en tiempo de ejecución

Desde Android Studio lo abriremos desde "Import project (Gradle, Eclipse, ADT, etc.)".

Finalmente crearemos una configuración para ejecutar directamente la versión java (Desktop) que, por razones obvias, es mucho más ágil para el desarrollo/pruebas del juego. Para ello, iremos a Run>Edit configurations… (o pinchando en el desplegable). Ahí en el + (Add new Configuration) seleccionamos Application y establecemos Name (p.e. Desktop), en Main class: DesktopLauncher, como Working directory: android\assets y Use classpath of module: desktop. Guardamos y ya podemos ejecutarlo con Run (Shift+F10) o directamente en el botón (play) asociado.

Este último párrafo se podría obviar abriendo directamente el archivo DesktopLauncher.java (del proyecto Desktop), y con el menú contextual seleccionar directamente Run DesktopLauncher.

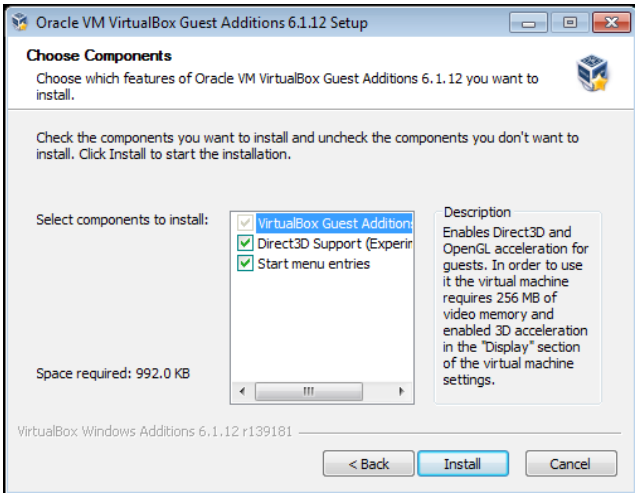
Es aconsejable analizar los main's (los *launcher*) de cada proyecto (desktop, android, etc.), fijándose en que todos ellos crean un objeto de la misma clase común del juego (core) y un objeto config con la configuración específica de cada plataforma. Por ejemplo, en el subproyecto Desktop, en DesktopLauncher.java, el objeto config dispone de los atributos width y height, donde se puede especificar las dimensiones iniciales de la ventana del juego.

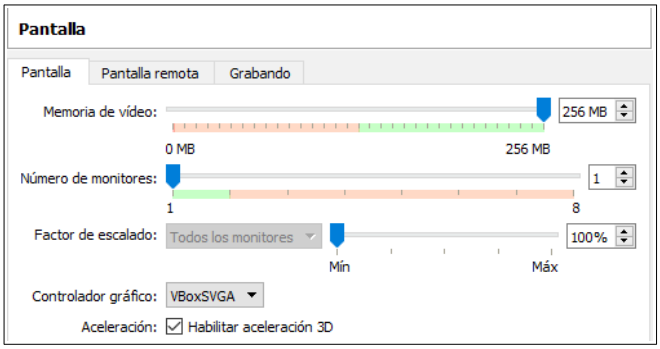

```
LwjglApplicationConfiguration config = new LwjglApplicationConfiguration();
config.width=480;
config.height=320;
```

Consideraciones en el uso de Oracle Virtual Box:

En el caso de desarrollar y ejecutar el proyecto en una máquina virtual de Oracle Virtual Box:

- Activar Direct3D Support en la instalación de las VirtualBox Guest Additions
- En la sección Pantalla establecer 256Mb de memoria de video y activar aceleración 3D



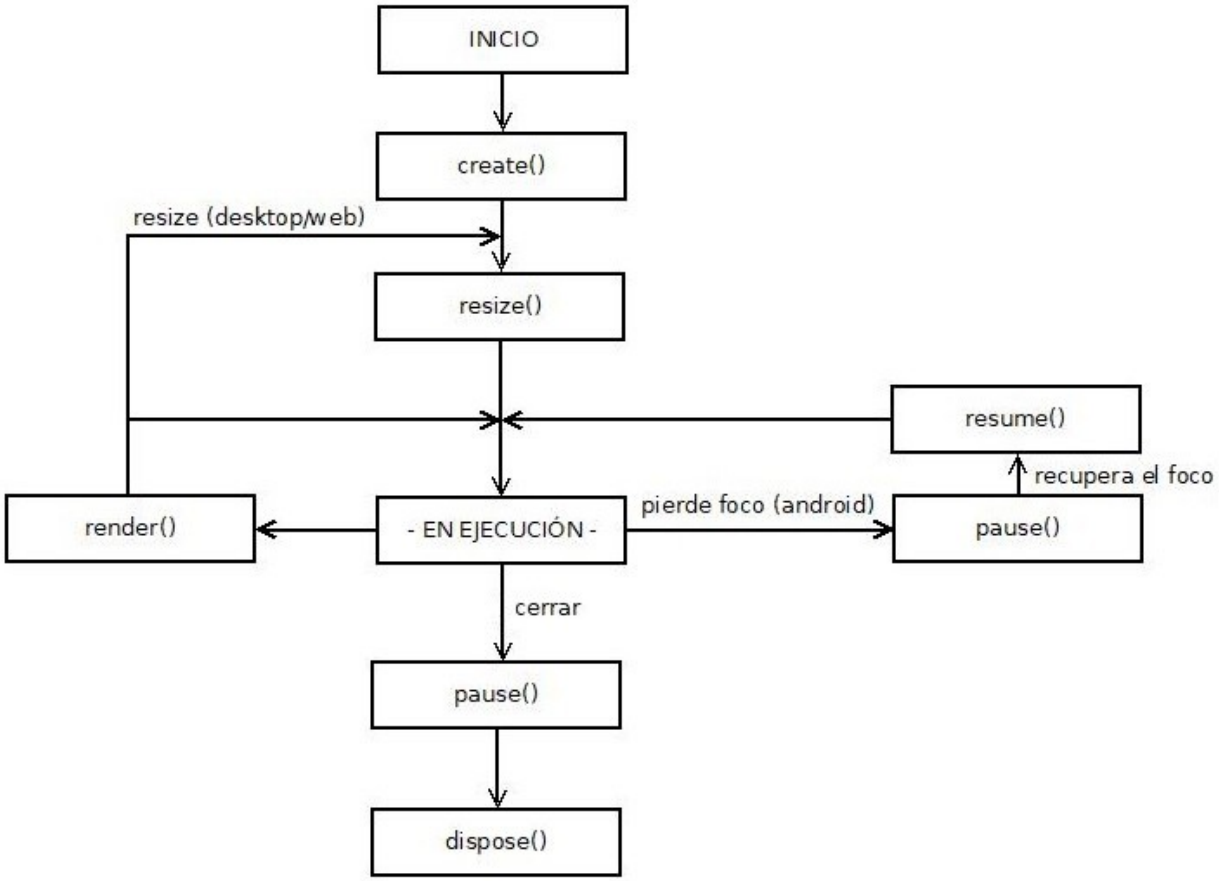


Ciclo de vida (bucle del juego)

Todos los proyectos específicos (de cada plataforma) necesitan un objeto que representa el juego, el cual tiene que implementar el *interface ApplicationListener*:

```
public interface ApplicationListener {
  public void create ();
  public void resize (int width, int height);
  public void render ();
  public void pause ();
  public void resume ();
  public void dispose ();
}
```

Otra posibilidad es heredar de la clase abstracta *ApplicationAdapter,* que a su vez implementa el *interface ApplicationListener*. Es la típica clase que implementa (sin hacer nada) un *interface* para poder heredar directamente y sobreescribir únicamente lo que nos interesa.



El método *create()* es el adecuado para inicializar la aplicación y cargar los *assets* en memoria
El método *resize()* es el adecuado para ajustarse al tamaño disponible (en los parámetros).
El método *render()* es el método que se llama continuamente (30~60 veces), en donde debemos realizar dos procesos fundamentales:

- actualizar el modelo (el mundo del juego)
- dibujar la nueva escena

El método *resize()* puede ser llamado más veces, p.e. en plataformas Desktop o WebGL

También puede saltar un evento *exit*, bien debido a un Gdx.app.exit() o bien porque lo envía el propio sistema. En este caso se llamará al método *pause()* y posteriormente al método *dispose()*.

Por esta causa *pause()* es el lugar adecuado para guardar el estado de la aplicación antes de que se cierre, y *dispose()* el lugar adecuado para liberar todos los recursos y recuperar memoria de la tarjeta. Los objetos de las clases de la librería libgdx que implementan el *interface Disposable* no son liberados automáticamente por la garbage collection, puesto que están en memoria de la tarjeta gráfica y no memoria principal. Por ello es importante liberarlos "a mano". Las clases que implementan este interface son (entre otras) BitmapFont,SpriteBatch,ShapeRenderer,Texture, etc.

El caso de las aplicaciones Android tiene una peculiaridad, puesto que por su naturaleza, una aplicación puede ser interrumpida y puesta en segundo plano (se pulsa el botón Home, salta otra aplicación por encima – una llamada, un alarma, etc.). En este caso, el ciclo de vida se amplía de manera que saltará igualmente el evento *pause()* pero seguido (en su momento), por un evento *resume()* en donde habría que recuperar el estado almacenado anteriormente.

Código generado por el asistente

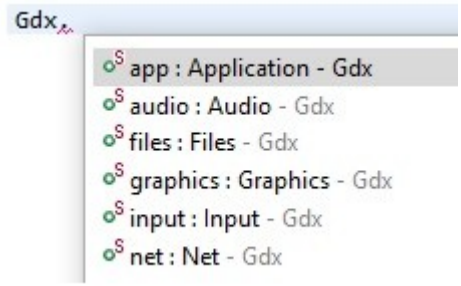
La clase principal del juego que genera el asistente hereda de la clase abstracta *ApplicationAdapter,* comentada en el epígrafe anterior. Entre en las clases de libGDX (Ctrl+clic en LwjglApplication en DesktopLauncher.java, p.e.) para ver el código de la librería, y cómo se implementan sus características (el ciclo de vida, etc.).

```
public class JuegoMinimo extends ApplicationAdapter {
	SpriteBatch batch;
	Texture img;

	@Override public void create () {
		batch = new SpriteBatch();
		img = new Texture("badlogic.jpg");
	}
	@Override public void render () {
		ScreenUtils.clear(1, 1, 1, 1);
		batch.begin();
		batch.draw(img, 0, 0);
		batch.end();
	}
	@Override public void dispose() {
		batch.dispose();
		img.dispose();
	}
}
```

Módulos de la librería

La aplicación controla el bucle del juego y su ciclo de vida. A través de distintos campos estáticos de la clase Gdx tendremos acceso a sus módulos principales.

- Aplicación (app)
- Gráficos (graphics)
- Audio (audio)
- Entrada (input)
- E/S archivos (files)
- Comunicaciones (net)



Módulo de aplicación

Accesible a través de *Gdx.app* nos permite acceder a distintas características, entre las que se encuentran las siguientes:

Log

Para generar líneas de log, en nuestro código deberemos primero establecer un nivel de log con el método *Gdx.app.setLogLevel(Application.LOG_);* con cuatro posibles niveles:

- LOG_NONE log deshabilitado
- LOG_ERROR genera logs de error, utilizando *Gdx.app.error(tag, message);*
- LOG_INFO genera además logs de información, utilizando *Gdx.app.log(tag, message);*
- LOG_DEBUG genera además logs de depuración, utilizando *Gdx.app.debug(tag,message);*

  Los mensajes salen en Console en las aplicaciones de escritorio y en LogCat en Android

Salir del juego

Para salir del juego de manera estructurada, es decir, asegurando que se ejecutan los métodos que responden a los distintos eventos del ciclo de vida, ejecutaremos *Gdx.app.exit();*

Por supuesto, es nuestra responsabilidad escribir el código para realizar las acciones de asignación y liberación de recursos, así como llamarlos en los métodos adecuados.

Almacenamiento de datos

Para almacenar pares clave-valor en archivos, utilizaremos los archivos de preferencias.

```
import com.badlogic.gdx.Preferences;
        …
Preferences preferences = Gdx.app.getPreferences("com.empresa.juego.record.prefs");
// Escribir
preferences.putString("usuario", usuario);
preferences.putInteger("puntuacion", puntuacion);
preferences.flush();
//Leer
int puntuacionRecord = preferences.getInteger("puntuacion",-1);
```

Después de obtener la referencia al archivo, podremos:

- escribir con los métodos putBoolean/Float/Integer/Long/String pasándole clave y valor
- escribir con el método put pasándole un mapa de claves/valores
- asegurarse que los cambios persistan con flush()
- leer con los métodos getBoolean/Float/Integer/Long/String, pasándole la clave o la clave y el valor que devuelven por defecto en caso de no existir la clave
- leer con el método get un mapa de sólo lectura con todos los pares clave/valor

Dependiendo de la plataforma, dicho archivo se almacenara:

- en Android, como SharedPreferences
- en iOS, como un NSMutableDictionary
- en GWT (HTML), como LocalStorage
- en Desktop (Java), en una carpeta prefs compartida.
  En Desktop es conveniente llamar al archivo con su nombre totalmente cualificado (p.e. com.mijuego.juego.ajustes en vez de ajustes) para evitar colisiones en dicha carpeta.

Tipo y versión de la plataforma

Para obtener el tipo de plataforma en que se está ejecutando el juego (y poder insertar código específico), podemos utilizar el método *Gdx.app.getType()*

```
switch(Gdx.app.getType()) { case Desktop: case Android: case WebGL: case iOS: }
```

Además, si estamos en Android o iOS, podremos interrogar al sistema por el Level API (Android) o por el número principal de la versión (iOS) con el método *Gdx.app.getVersion()*

Módulo de gráficos

Accesible a través del método *Gdx.getGraphics()* o *Gdx.graphics* nos permite acceder a distintos métodos y propiedades imprescindibles para el correcto desarrollo del juego:

- setTitle(String titulo) establece el título de la ventana (ignorado en Android)
- getWidth() anchura en pixels del dispositivo (del área de visualización)
- getHeight() altura en pixels del dispositivo (del área de visualización)
- getFramesPerSecond() devuelve la media del fps (frames/fotogramas por segundo). Este valor nos indica de alguna manera la fluidez percibida en los movimientos del juego. Cuántos mayor sea el valor de fps más fluido se percibirá, y viceversa. Un valor a partir del que probablemente se vea fluido (siempre dependiendo de la velocidad de los personajes en la pantalla) es entre 30 y 60fps. Conseguir un valor adecuado está influenciado por diversos aspectos entre los que se encuentran:
  - la velocidad del hardware
  - la programación (más o menos óptima) del juego
- getDeltaTime() nos devuelve el valor delta, que explicamos a continuación:

El valor delta es el tiempo (en segundos) transcurrido entre el frame actual y anterior, y posiblemente el dato más importante en un juego, pues nos permite llevar un control del tiempo.

Es habitual crear un campo *float stateTime* en el juego, e incluir en render una línea *stateTime += delta* para tener siempre a mano el tiempo de juego. Este valor tendrá multitud de utilidades, p.e.:

- poner un límite temporal a una misión
- incrementar la dificultad con el tiempo (velocidades, aceleraciones, nº enemigos)
- etc.

Es un valor inversamente proporcional a los fps, y que obtendremos en cada render a través del método comentado, aunque existen clases/interfaces de más alto nivel como Game/Screen que sobrescriben el método render proporcionando el valor delta directamente como parámetro, evitándonos una línea de código.

Módulo de archivos

Accesible a través del método *Gdx.getFiles()* o *Gdx.files* nos permite acceder a los archivos de la aplicación. Estos archivos pueden ser de dos tipos:

- Internos. Accesibles con *Gdx.files.internal*(), son los archivos localizados en *assets* en las plataformas Android y WebGL, y en la carpeta raíz de la aplicación en *Desktop.*
- Externos. Accesibles con *Gdx.files.external*(), son los archivos localizados en la tarjeta SD en la plataforma Android y en la carpeta raíz del usuario en *Desktop*, no disponible en WebGL.

Módulo de audio

Accesible a través del método *Gdx.getAudio()* o *Gdx.audio* nos permite utilizar:

- Sonidos pequeños efectos sonoros (un disparo, una nueva vida, un paso) de < 1MB
- Música música (streaming) que suena de fondo en el juego

Con Gdx.audio.newSound/newMusic(FileHandler archivo) creamos un objeto que nos permitirá controlar el sonido/música con métodos play(), pause(), stop(), etc. Los parámetros habituales son:

- float volume: un valor entre 0 (silencio) y 1 (volumen máximo)
- float pitch: velocidad del sonido, entre [0.5,2.0]. Menor que 1 es lento y mayor es rápido.
- float pan: entre [-1,1] indica por qué altavoz sale el sonido (-1 solo izq, 1 der, 0 por ambos)

```
Music musica=Gdx.audio.newMusic(Gdx.files.internal( path: "musica.mp3"));
Sound sonido=Gdx.audio.newSound(Gdx.files.internal( path: "sonido.mp3"));
sonido.
```

```
play()                                  long
play(float volume)                      long
play(float volume, float pitch, float p… long
dispose()                               void
loop()                                  long
loop(float volume)                      long
loop(float volume, float pitch, float p… long
pause()                                 void
pause(long soundId)                     void
resume()                                void
resume(long soundId)                    void
```

Módulo de entrada

Accesible a través de *Gdx.getInput()* o *Gdx.input*, nos permitirá interrogar el estado de los dispositivos de entrada (teclado, ratón, toques en interfaz táctil, gestos, acelerómetro, etc.), así como responder a los avisos generados ante cualquier cambio.

Dada la importancia que tiene el módulo de entrada en el juego, y teniendo en cuenta que su programación debe se la adecuada para obtener una respuesta óptima, se deja su explicación en profundidad para un epígrafe posterior.

Módulo de comunicaciones

Accesible a través del método *Gdx.getNet()* o *Gdx.net*, podremos:

- Abrir un URI en la aplicación por defecto (navegador) con Gdx.net.openURI(URI)
- Gestionar sockets con Gdx.net.newClientSocket() y Gdx.net.newServerSocket()
- Gestionar peticiones HTTP con Gdx.net.sendHttpRequest() y Gdx.net.cancelHttpRequest()

En el siguiente ejemplo se puede ver el código necesario para obtener de un servicio web REST la información del actual record de un juego:

```
String urlRecordsREST = "http://localhost/record/pescador.php";

Net.HttpRequest request=new Net.HttpRequest("GET");
request.setUrl(urlRecordsREST);
request.setHeader("Accept","application/json");
Gdx.net.sendHttpRequest(request, new Net.HttpResponseListener() {

    @Override
    public void handleHttpResponse(Net.HttpResponse httpResponse) {
        if(httpResponse.getStatus().getStatusCode()== HttpStatus.SC_OK) {
            JsonReader jsonReader=new JsonReader();
             JsonValue datos=jsonReader.parse(httpResponse.getResultAsString());
             nombreRecord=datos.getString("jugador");
             tiempoRecord=datos.getInt("tiempo");
        }
        else
             SinRecord();
    }

    @Override public void failed(Throwable t) { SinRecord(); }

    @Override public void cancelled() { SinRecord(); }

    private void SinRecord() { tiempoRecord=-1; nombreRecord=""; }

});
```

Dibujando en la pantalla

Todo el código de dibujo de la pantalla (fondos, personajes, etc.) se concentrará en el método render, pero antes de dibujar deberá llamarse a las funciones que borran el buffer de la pantalla (de la tarjeta gráfica) con el color de fondo indicado (Red,Green,Blue,Alpha) especificados como argumentos float con rango [0f-1f] del método glClearColor.

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

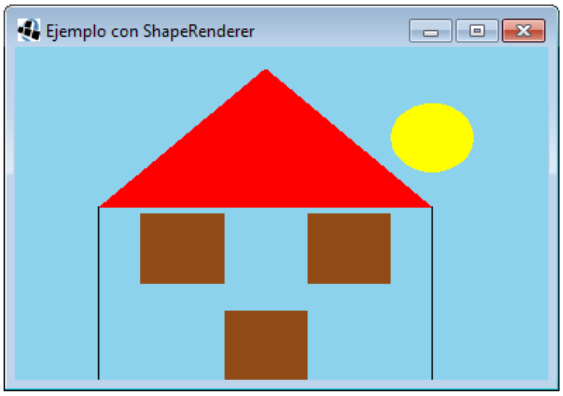
El alfa indica la opacidad de cada píxel y permite que una imagen se combine con otras utilizando la composición alfa, con áreas transparentes .

ShapeRenderer y SpriteBatch

Debido a que es muy ineficiente enviar a la GPU de manera independiente distintas imágenes, exiten dos clases que nos permiten añadir los componentes que queramos dibujar entre los métodos begin() y end() y así enviarlos de una vez, por lo que ganaremos en rendimiento.

ShapeRenderer – sirve para dibujar gráficos vectoriales, como puntos, líneas y figuras en una única llamada.

Fijarse que en el ejemplo se han utilizado dos pares begin/end, pues se ha pasado como parámetro del begin el tipo de figura (ShapeType.Line/Filled/Point). Se podría haber hecho sin problema un begin() sin parámetros y asignar las veces que fuese necesario el tipo de figura en su interior a través del método set(ShapeType), aunque habría que inicializar sr.setAutoShapeType(true); (es más ineficiente...?)




```
public class MyGdxGame extends ApplicationAdapter {
    ShapeRenderer sr;
    private static final float X_CASA=100;
    private static final float ANCHO_CASA=400;
    private static final float ALTO_CASA=250;
    static final float ANCHO_ELEMENTOS =ANCHO_CASA*.25f;
    static final float ALTO_ELEMENTOS =ALTO_CASA*.40f;

    @Override
    public void create () {
        sr=new ShapeRenderer();
    }
```

```
    @Override
    public void render () {
     Gdx.gl.glClearColor(.53f,.81f, .92f, 1);
     Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

     sr.begin(ShapeRenderer.ShapeType.Line);
     sr.setColor(Color.BLACK);
     sr.rect(X_CASA,0,ANCHO_CASA,ALTO_CASA);
     sr.end();

     sr.begin(ShapeRenderer.ShapeType.Filled);
     sr.setColor(Color.BROWN);
     sr.rect(X_CASA+(ANCHO_CASA/2- ANCHO_ELEMENTOS /2),0,
             ANCHO_ELEMENTOS,ALTO_ELEMENTOS);
     sr.rect(X_CASA+ANCHO_ELEMENTOS/2,ALTO_CASA-ALTO_ELEMENTOS-10,
             ANCHO_ELEMENTOS,ALTO_ELEMENTOS);
     sr.rect(X_CASA+ANCHO_CASA-ANCHO_ELEMENTOS*1.5f,ALTO_CASA-ALTO_ELEMENTOS-10,
             ANCHO_ELEMENTOS,ALTO_ELEMENTOS);
     sr.setColor(Color.RED);
     sr.arc(X_CASA+ANCHO_CASA/2,ALTO_CASA,ANCHO_CASA/2,0,180,2);
     sr.setColor(Color.YELLOW);
     sr.circle(X_CASA+ANCHO_CASA,ALTO_CASA+100,50);
     sr.end();
    }

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

SpriteBatch – para dibujar varias texturas (imágenes) en una única llamada.

```
public class MyGdxGame extends ApplicationAdapter {
    SpriteBatch sb;

    @Override
    public void create () {
        sb = new SpriteBatch();
    }
    @Override
    public void render () {
        Gdx.gl.glClearColor(1, 0, 0, 1);
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
        sb.begin();
          // aquí añadimos al SpriteBatch las texturas que queremos dibujar
        sb.end();
    }

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

Texture y TextureRegion

Una textura es una imagen subida a la memoria gráfica después de ser decodificada de su formato origen (PNG, JPG, …).

Las imágenes las almacenamos en la carpeta assets del proyecto de Android (o una subcarpeta). Las carpetas *resources* del resto de proyectos son un simple acceso directo.

```
Texture imagen = new Texture(Gdx.files.internal("imagenes/dibujo.jpg"));
~
Texture imagen = new Texture("imagenes/dibujo.jpg"");

batch.begin();
batch.draw(imagen,0,0);
batch.end();
```

Las imágenes que se pintan en pantalla utilizarán un sistema de coordenadas donde, salvo que indiquemos lo contrario (ver cámara más adelante), la esquina inferior izquierda el el origen (0,0), el eje de las x apunta hacia la derecha y el eje y apunta hacia arriba.

Sus dimensiones (altura y anchura) deberían ser potencias de 2 (POT, Power of Two – 16x16, 64x256, etc) por razones de compatibilidad y rendimiento. Para trabajar con tamaños NPOT (Non POT), podríamos extraer de la textura una región con la imagen deseada en forma de TextureRegion. Para ello crearemos un objeto TextureRegion, pasándole a su constructor la Texture y la zona a recortar. Tener en cuenta que en este caso las coordenadas 0,0 sí se corresponden con la posición superior izquierda.

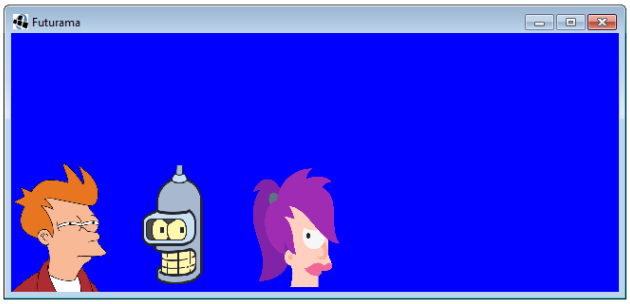
Es muy habitual utilizar un Texture potencia de dos en las dimensiones para extraer varias imágenes creando así varios TextureRegion

Los TextureRegion no necesitan dispose, sí sus texturas asociadas.

Todas las texturas (Textures y TextureRegion) las dibujaremos a través del método draw del SpriteBatch, método muy sobrecargado:

```
void draw(Texture texture, float x, float y)
void draw(Texture texture, float[] spriteVertices, int offset, int count)
void draw(Texture texture, float x, float y, float width, float height)
void draw(Texture texture, float x, float y, float width, float height,
                float u, float v, float u2, float v2)
void draw(Texture texture, float x, float y, float originX, float originY,
                float width, float height, float scaleX, float scaleY, float rotation,
                int srcX, int srcY, int srcWidth, int srcHeight, boolean flipX, boolean flipY)
void draw(Texture texture, float x, float y, float width, float height, int srcX, int srcY,
                int srcWidth, int srcHeight, boolean flipX, boolean flipY)
void draw(Texture texture, float x, float y, int srcX, int srcY, int srcWidth, int srcHeight)
void draw(TextureRegion region, float x, float y)
void draw(TextureRegion region, float width, float height, Affine2 transform)
void draw(TextureRegion region, float x, float y, float width, float height)
void draw(TextureRegion region, float x, float y, float originX, float originY,
                float width, float height, float scaleX, float scaleY, float rotation)
void draw(TextureRegion region, float x, float y, float originX, float originY,
                float width, float height, float scaleX, float scaleY, float rotation, boolean
clockwise)
```

```java
public class MyGdxGame extends ApplicationAdapter {
    SpriteBatch sb;
    Texture bender,leela,fry;
    @Override
    public void create () {
        Gdx.graphics.setTitle("Futurama");
        sb = new SpriteBatch();
        bender = new Texture("personajes/bender.png");
        leela = new Texture("personajes/leela.png");
        fry = new Texture("personajes/fry.png");
    }
    @Override
    public void render () {
        Gdx.gl.glClearColor(0,0, 1, 1);
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
        sb.begin();
        sb.draw(fry, 0, 0, 100,100);
        sb.draw(bender, 125, 0, 100,100);
        sb.draw(leela, 250, 0, 100,100);
        sb.end();
    }

    @Override
    public void dispose () {
        sb.dispose();
        bender.dispose();
        leela.dispose();
        fry.dispose();
    }
}
```



En el caso de querer pintar una textura al revés en horizontal (o vertical), podemos indicar el ancho (o el alto) en negativo. Podemos observarlo en el ejemplo anterior, si realizamos los siguientes cambios:

```java
sb.draw(bender, 125, 0, -100,100);
```
Gira, pero se pinta hacia atrás.

```java
sb.draw(bender, 125+100, 0, -100,100);
```
Empezamos a pintar sumándole su ancho a la posición

En el ejemplo anterior, le sumamos 100 porque es el ancho al que estamos pintando la textura en el método draw. En el caso de pintarla a su tamaño original, podríamos obtener el ancho y alto original de la textura con los métodos **textura**.getWidth() y **textura**.getHeight()

Sprite

Cuando utilizamos texturas sobre todo al dotarlas de movimiento, es muy habitual tener que almacenar una serie de datos para cada una, como puede ser la posición o el color, lo cual suele encapsularse en clases como Personaje o Enemigo. Para simplificar la gestión de la textura, sus datos básicos y sus acciones más comunes, existe la clase Sprite ~ Texture + geometría + color

En el siguiente ejemplo los personajes son sprites, no simples texturas, y se giran (al tocar con el dedo/ratón) y se dan la vuelta (al tocar cq. tecla) aleatoriamente utilizando métodos específicos de su clase (rotate90 y flip). Fijarse el uso de justTouched/isKeyJustPressed vs isTouched/isKeyPressed (y, como veremos, mejorable utilizando un procesador de entrada implementando InputProcessor)

```
public class MyGdxGame extends ApplicationAdapter {
    SpriteBatch sb;
    Random random=new Random();
    Sprite[] personajes;

    @Override public void create () {
        Gdx.graphics.setTitle("Futurama");
        sb = new SpriteBatch();
        crearPersonajes();
    }
    @Override public void render () {
        Gdx.gl.glClearColor(0,0, 1, 1);
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
        if (Gdx.input.justTouched()) {
            boolean sentidoDelReloj=random.nextBoolean();
            for (Sprite personaje : personajes)
                personaje.rotate90(sentidoDelReloj);
        }
        if (Gdx.input.isKeyJustPressed(Input.Keys.ANY_KEY)) {
            boolean enLasX=random.nextBoolean();
            for(Sprite personaje:personajes)
                personaje.flip(enLasX,!enLasX);
        }
        sb.begin();
        for(Sprite personaje:personajes)
            personaje.draw(sb);
        sb.end();
    }
    @Override public void dispose () {
        sb.dispose();
        liberarPersonajes();
    }
    private void crearPersonajes() {
        String[] strPersonajes={"fry","bender","leela"};
        personajes=new Sprite[strPersonajes.length];
        int i=0;
        for(String nombrePersonaje:strPersonajes) {
            Texture texture=new Texture("personajes/" + nombrePersonaje + ".png");
            Sprite personaje=new Sprite(texture);
            personaje.setPosition(i*125,0);
            personaje.setSize(100,100);
            personajes[i++] = personaje;
        }
    }
    private void liberarPersonajes() {
        for(Sprite personaje:personajes)
            personaje.getTexture().dispose();
    }
}
```

Atlas

Cuando tenemos muchas imágenes en un juego (lo habitual), es más eficiente crear una imagen grande con todas las que necesitamos y un archivo de texto asociado con las regiones creadas y sus datos asociados. Pueden crearse con la utilidad de línea de comandos TexturePacker (existe un Texture Packer GUI), o alternativas (Free Texture Packer, Free Sprite Sheet Packer, etc.)

Indicando como parámetro el nombre de la región existente en el archivo de configuración del atlas, podemos extraer las imágenes del objeto TextureAtlas de dos maneras:

- con findRegion obteniendo objetos de tipo TextureRegion
- Con createSprite, creando un Sprite con la textura extraída

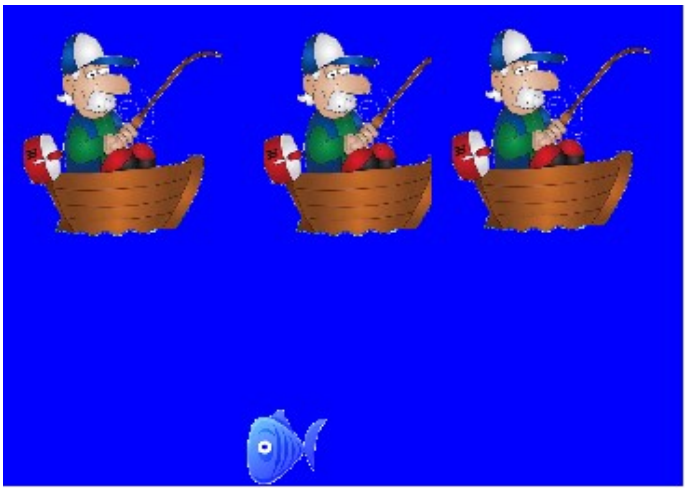
```
personajes.png
format: RGBA8888
filter:
Nearest,Nearest
repeat: none
pescador
  rotate: false
  xy: 1, 99
  size: 256, 256
  orig: 256, 256
  offset: 0, 0
  index: -1
pez_amarillo
  rotate: false
  xy: 259, 163
  size: 288, 192
  orig: 288, 192
  offset: 0, 0
  index: -1
```

```
pez_lila
  rotate: false
  xy: 549, 259
  size: 384, 96
  orig: 384, 96
  offset: 0, 0
  index: -1
pez
  rotate: false
  xy: 1, 1
  size: 96, 96
  orig: 96, 96
  offset: 0, 0
  index: -1
anzuelo
  rotate: false
  xy: 259, 105
  size: 35, 56
  orig: 35, 56
  offset: 0, 0
  index: -1
```

```
punto
  rotate: false
  xy: 549, 252
  size: 5, 5
  orig: 5, 5
  offset: 0, 0
  index: -1
```

En el siguiente ejemplo en cada fotograma vamos incrementando la posición del pescador:

```
public class MyGdxGame extends ApplicationAdapter {
    SpriteBatch sb;
    TextureAtlas atlas;
    TextureRegion pez;
    Sprite pescador;
    @Override
    public void create () {
        Gdx.graphics.setTitle("Atlas");
        sb = new SpriteBatch();
        atlas=new TextureAtlas("personajes.atlas");
        pez=atlas.findRegion("pez");
        pescador=atlas.createSprite("pescador");
        pescador.setPosition(0,300);
    }
    @Override
    public void render () {
        Gdx.gl.glClearColor(0,0, 1, 1);
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
        sb.begin();
        sb.draw(pez,300,0);
        pescador.draw(sb);
        pescador.setPosition(pescador.getX()+1,pescador.getY());
        sb.end();
    }

    @Override
    public void dispose () {
        sb.dispose();
        atlas.dispose();
    }
}
```



Animation

En el caso de que nuestras texturas tengan varias imágenes relacionadas, creadas para visualizar una distinta cada cierto tiempo, dando la impresión de que el dibujo está animado, entonces podemos crear objetos de la clase Animation para poder visualizarlas de esa manera.

Lo primero que hay que hacer es utilizar el método split para dividir la textura en cada una de las partes individuales de la animación. Como los fotogramas se pueden repartir por filas y columnas en el dibujo (1Fil x nCol, nFil x 1Col o nFil x nCol), split devolverá un array bidimensional TextureRegion[][].

La función split existe como método de instancia pasándole la anchura y la altura de cada imagen individual, y como método *static* de TextureRegion pasándole además la textura.

Ya podemos crear un objeto Animation, pasándole como parámetros float *frameDuration* (tiempo en segundos entre frames) y el array de frames. Pero tenemos un problema, pues como se puede observar en el constructor de Animation, los frames los recibe en un array unidimensional, por lo que hay que convertir nuestro array bidimensional en un array unidimensional (que contenga los mismos frames). La manera más obvia de realizar esta conversión sería recorrer las dos dimensiones creando simultáneamente el array de una dimensión con cada elemento. Aunque a veces resulta que nuestro array de dos dimensiones solo tiene una fila (o columna), porque la animación viene en un formato lineal (no como cuadrícula) en la textura original, por lo que nuestro array unidimensional ya estará directamente accesible como elemento en una de las dimesiones.

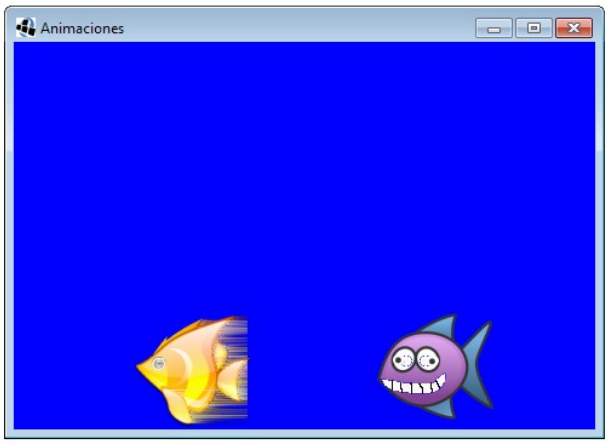
Una vez creada la animación, en el renderizado únicamente tenemos que dibujar el frame “que toca en ese momento”. Para obtenerlo tenemos que llamar a getKeyFrame pasándole dos parámetros:

- el tiempo que llevamos de juego (*stateTime*) . Lo necesita para saber qué frame devolver, pues depende del valor *frameDuration* que le hemos proporcionado al objeto Animation.
- un boolean loooping para decirle si queremos que, al acabarse los frames, vuelva a devolver el primero, y continúe así indefinidamente.

Usando el código y el atlas de un epígrafe anterior, podemos crear las animaciones del pez amarillo y el pez lila de la siguiente manera. Fijarse que:

- en los split se indican las fórmulas (no los cálculos finales) para entender cómo se calculan, básicamente con los anchos toales de las texturas (que se ven en la información del atlas) y el número de filas y columnas de frames de la animación (que se ve en la imagen del atlas – 4x1 para el pez lila y 2x3 para el pez amarillo). Deberían usarse constantes con dichos valores en las clases adecuadas cuando se codifique mejor el juego.
- al crear la animación, en el caso del pez lila, le pasamos la primera fila (índice 0) del array, y en el caso del pez amarillo, creamos un array de una dimensión. El código para “aplanar” el array de dos dimensiones debería sacarse a una función y generalizarlo…
- el frameDuration se ha establecido a .15f. Cuanto más pequeño sea el valor, más rápida irá la animación, y viceversa.

```
public class MyGdxGame extends ApplicationAdapter {
 SpriteBatch sb;
 TextureAtlas atlas;
 Animation aniPezAmarillo,aniPezLila;
 private float stateTime;

 @Override
 public void create () {
  Gdx.graphics.setTitle("Animaciones");
  sb = new SpriteBatch();
  atlas=new TextureAtlas("personajes.atlas");

  TextureRegion pezLila=atlas.findRegion("pez_lila");
  TextureRegion[][] framesPezLila=pezLila.split(384/4,96/1);
  aniPezLila=new Animation(.15f,framesPezLila[0]);

  TextureRegion pezAmarillo=atlas.findRegion("pez_amarillo");
  TextureRegion[][] framesPezAmarillo=pezAmarillo.split(288/3,192/2);
  int filasPezAmarillo=framesPezAmarillo.length;
  int columnasPezAmarillo=framesPezAmarillo[0].length;
  TextureRegion[] framesPA=new TextureRegion[filasPezAmarillo*columnasPezAmarillo];
  int numFrame=0;
  for(int fila=0;fila<filasPezAmarillo;fila++)
     for(int columna=0;columna<columnasPezAmarillo;columna++)
       framesPA[numFrame++]=framesPezAmarillo[fila][columna];
  aniPezAmarillo=new Animation(.15f,framesPA);
 }

 @Override
 public void render () {
  Gdx.gl.glClearColor(0,0, 1, 1);
  Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
  stateTime+=Gdx.graphics.getDeltaTime();
  sb.begin();
  TextureRegion frameLila=(TextureRegion)aniPezLila.getKeyFrame(stateTime,true);
  TextureRegion frameAmarillo=(TextureRegion)aniPezAmarillo.getKeyFrame(stateTime,true);
  sb.draw(frameLila,300,0);
  sb.draw(frameAmarillo,100,0);
  sb.end();
 }

 @Override
 public void dispose () {
  sb.dispose();
  atlas.dispose();
 }
}
```



Textos y fuentes

Las fuentes se renderizan utilizando la clase BitmapFont.

Lo más sencillo es crear un BitmapFont con el constructor por defecto, o también se pueden utilizar una de las muchas sobrecargas que tiene, por ejemplo, la que se le pasa un archivo de texto de formato.fnt (AngleCode BMFont), donde describe donde está cada glifo en la imagen (glyphs, representación gráfica de un elemento de la escritura), y otro conteniendo una imagen con dichos glifos.

Si añadimos las siguientes líneas a un ejemplo anterior, visualizaremos un texto:

```
public class MyGdxGame extends ApplicationAdapter {
    …
    BitmapFont fuente;
    float alto;
    @Override
    public void create () {
        …
        fuente=new BitmapFont();
        fuente.setColor(Color.YELLOW);
        fuente.getData().setScale(1.9f);
        alto=Gdx.graphics.getHeight();
    }
    @Override
    public void render () {
        …
        sb.begin();
        fuente.draw(sb,"Personajes de Futurama ",10,alto-10);
        …
        sb.end();
    }

    @Override
    public void dispose () {
        …
       fuente.dispose();
    }
}
```

Para crear los archivos .fnt y .png podemos utilizar diversas herramientas, como por ejemplo Hiero https://github.com/libgdx/libgdx/wiki/Hiero. De esta manera podemos crear un archivo con todos los caracteres que necesitemos.

Blending

Cuando el SpriteBatch tiene el blending activado (por defecto lo está) quiere decir que cuando se dibuja una textura se tienen en cuenta sus partes traslúcidas que serán combinadas con las del fondo.

Cuando se desactiva, todo lo que haya en la pantalla se reemplaza por la textura, lo cual es mucho más eficiente (por ejemplo, al pintar un fondo). Por esta razón, salvo que sea necesario se debería desactivar siempre.

```
sb.begin();
sb.disableBlending();
fondo.draw(...);
sb.enableBlending();
// seguimos dibujando ...
sb.end();
```

TiledMap

Para minimizar la RAM e incrementar el rendimiento en el renderizado, se suele utilizar una técnica a la hora de dibujar gráficos grandes (típicamente mapas o fondos del juego) que consiste en reutilizar pequeños gráficos a lo largo de la imagen (ladrillos, cesped, etc.)

Para crear un mapa de este tipo podemos utilizar una aplicación como Tiled: www.mapeditor.org , que gestionará los archivos .TMX (Tile Map Format), .TSX (Tile Set XML) y un .PNG donde estarán todas las imágenes del mapa. El archivo TSX es opcional, pues almacena el contenido del atributo <tileset> en un archivo externo.

Ver más de Tiled en http://www.gamefromscratch.com/post/2014/04/15/A-quick-look-at-Tiled-An-open-source-2D-level-editor.aspx

Velocidad y renderizado

El método *render* se llama continuamente y es donde debemos realizar con todos los personajes de nuestro juego, al menos, dos acciones:

- actualizar su estado (posiciones, tamaños, colores, etc.)
- pintar la pantalla teniendo en cuenta el nuevo estado

El nuevo estado del juego dependerá de múltipes factores, entre los que se encuentran:

- el tiempo transcurrido desde el fotograma anterior (delta): hay que actualizar la posición de un personaje que se mueve automáticamente
- el tiempo transcurrido de juego (statetime): se ha acabado el tiempo de la partida, o se está agotando el oxígeno para respirar, se ha acabado el tiempo de un escudo de protección temporal, a partir de cierto momento añadimos más enemigos o incrementamos su velocidad de ataque
- el estado de la entrada (input): se está pulsando una tecla para mover a un personaje, para disparar, para recoger algo, o para activar un escudo de protección temporal
- la propia lógica del juego: colisiones entre personajes, acciones realizadas, etc.

Con respecto a las posiciones de los personajes, si en cada render lo incrementamos en un valor constante, éste se moverá más rápido o más despacio en función de los fps de ese dispositivo. Es decir, el juego irá más rápido en máquinas potentes, y más despacio en el resto. Esto no debería ocurrir nunca, puesto que, además de hacer que en algunos dispositivos pueda ser injugable (por lento o por rápido), no sería muy justo.

El objetivo es que debemos decidir de antemano la velocidad adecuada para los personajes de nuestro juego (que obviamente podremos cambiar si lo deseamos durante la ejecución), y para calcularla utilizaremos la conocida fórmula:

espacio = velocidad * tiempo <=> velocidad = espacio / tiempo.

En un caso concreto, por ejemplo, si queremos que nuestro personaje tarde 8 segundos en recorrer la pantalla de un extremo a otro (p.e. 400 puntos), pues la velocidad que tendremos que asignarle será 400/8 = 50. De esta manera, en cada render incrementaremos (o decrementaremos) la posición del personaje en velocidad * delta => 50 * delta.

En un dispositivo muy rápido (fps alto) se llamará muchas veces al render y se actualizará muchas veces la posición pero en poquitos puntos, puesto que el delta será muy pequeño.

En un dispositivo muy lento (fps bajo) se llamará pocas veces al render y se actualizará pocas veces la posición, pero en bastantes puntos, puesto que el delta será alto. En este dispositivo podremos llegar a notar que el personaje va a saltos (poca fluidez), pero tardará lo mismo que el anterior en recorrer la misma distancia.

La cámara

Por lo general nos interesa trabajar con coordenadas referidas a nuestro "Mundo" en el juego, en el que establecemos el tamañ´beo que nos interese. Para calcular las coordenadas reales que se van a utilizar en el dispositivo final, necesitaremos una cámara.

La cámara que utilizaremos en nuestros juegos 2D será de tipo ortográfica (ortogonal) – que básicamente es una cámara que mira en ángulo recto a la escena. Trabajaremos con coordenadas x,y,z encapsuladas en las clases Vector2/Vector3 (en 2D usaremos z--la profundidad-- con valor 0)

Para realizar esta conversión – entre coordenadas del juego (mundo) y coordenadas del dispositivo (pantalla) – se realizan una serie de operaciones matemáticas con matrices. Una cámara de este tipo tiene una posición (su dirección, hacia dónde está mirando), lo cual se define en una matriz de modelado, y un tamaño del contenido que se va a visualizar (matriz de proyección). El cálculo de nuestra proyección se calculará utilizando una matriz combinada de ambas.

Una cámara será un objeto de la clase *OrthographicCamera* que nos permitirá:

- mover o rotar la cámara .- a través de los métodos *translate* y *rotate*
- hacer zoom .- con el método *zoom*
- cambiar el tamaño de lo visualizado .- propiedades *viewportwidth* y *viewportheight*
- calcular coordenadas del mundo a pantalla y viceversa .- *project* y *unproject*

Los métodos más utilizados serán:

- *setToOrtho(boolean yDown, ...)* .- define el viewport de la cámara. A continuación hay que hacer un *update*. El primer parámetro indica si el eje y apunta hacia arriba o hacia abajo, o sea, si la coordenada (0,0) está en la esquina inferior izquierda, como en matemáticas, o en la superior.
- *update* .- actualiza las nuevas matrices de la cámara. Hay que llamarlo después de cualquier cambio de posición o tamaño de la cámara.

Se denomina *viewport* a la vista –definida con un ancho y un alto– que en un momento determinado tiene el juego de una escena. Cuando ocurre un evento *resize* (al menos una vez al comenzar la aplicación), se recalcularán los parámetros del *viewport* y se actualizará la cámara.

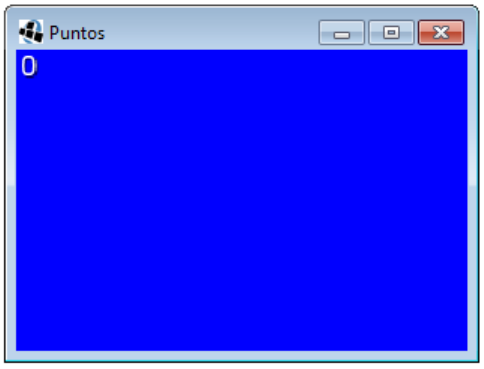
```
OrthographicCamera camara = new OrthographicCamera();

@Override public void resize(int width, int height) {
       camara.setToOrtho(false, Mundo.ANCHO,Mundo.ALTO);
    camara.update();
    sb.setProjectionMatrix(camara.combined); // SpriteBatch
    sr.setProjectionMatrix(camara.combined); // ShapeRenderer
}

@Override public boolean touchDown(int screenX, int screenY, int pointer, int button {
       Vector3 v3CoordenadasDePantalla=new Vector3(screenX,screenY,0);
    Vector3 v3CoordenadasDeMundo = camara.unproject(v3CoordenadasDePantalla);
}
```

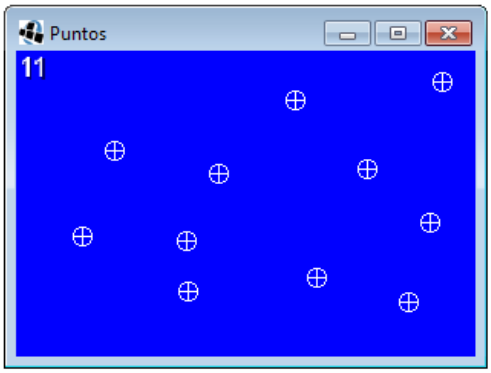
El uso de la cámara hace facilita la programación del juego centrándonos en las coordenadas de nuestro Mundo, olvidándonos en parte del dispositivo de visualización,

Veamos el código necesario para realizar una aplicación que nos vaya pintando los toques en la pantalla (ratón o dedo en dispositivo táctil), en un juego con unas dimensiones de Mundo.ANCHO=480 y Mundo.ALTO=320 (ratio 1,5). (Tip: cambie por touchDown por touchDragged)

```
public class Mundo {
    public static final float ANCHO = 480 ;
    public static final float ALTO = 320 ;
    public static final int ANCHO_PUNTO = 20;
}
```

```
public class MyGdxGame extends InputAdapter implements ApplicationListener {
   OrthographicCamera camara;
   ShapeRenderer sr;
   SpriteBatch sb;
   BitmapFont fuente;
   ArrayList<Vector3> puntos;

   @Override public void create () {
      camara=new OrthographicCamera();
      Gdx.graphics.setTitle("Puntos");
      puntos = new ArrayList();
      fuente = new BitmapFont();
      fuente.getData().setScale(2f);
      sr = new ShapeRenderer();
      sb = new SpriteBatch();
      Gdx.input.setInputProcessor(this);
   }

   @Override public boolean touchDown(int screenX, int screenY, int pointer, int button) {
      puntos.add(camara.unproject(new Vector3(screenX,screenY,0)));
      return true;
   }

   private void dibujarPuntos(ShapeRenderer sr) {
      int r=Mundo.ANCHO_PUNTO/2;
      sr.begin(ShapeRenderer.ShapeType.Line);
      for(Vector3 punto: puntos) {
         sr.circle(punto.x, punto.y, r);
         sr.line(punto.x - r, punto.y, punto.x + r, punto.y);
         sr.line(punto.x, punto.y - r, punto.x, punto.y + r);
      }
      sr.end();
   }

   @Override public void resize(int width, int height) {
      camara.setToOrtho(false,Mundo.ANCHO,Mundo.ALTO);
      camara.update();
      sr.setProjectionMatrix(camara.combined);
      sb.setProjectionMatrix(camara.combined);
   }

   @Override public void render () {
      Gdx.gl.glClearColor(0,0, 1, 1);
      Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
      sb.begin();
      fuente.draw(sb,""+puntos.size(), 5, Mundo.ALTO-5);
      sb.end();
      dibujarPuntos(sr);
   }

   @Override public void pause() {}
   @Override public void resume() {}
   @Override
   public void dispose () {
      sr.dispose();
   }
}
```





Gestión de la entrada

Accesible a través de *Gdx.getInput()* o *Gdx.input*. Podemos usar los siguientes métodos:

Existen dos mecanismos básicos para gestionar la entrada.

Interrogación directa y constante (polling)

Se basa en interrogar directamente y de manera contínua (en cada render) sobre el estado de los dispositivos de entrada. Generalmente no es una buena práctica como mecanismo de entrada general del juego, sino como apoyo para conocer el estado de la entrada en un momento puntual.

Con respecto a tocar la pantalla (== clic del ratón en Desktop) disponemos de

- isTouched*()*
- justTouched*()*
  que devuelven true si en ese momento se está tocando la pantalla (*)

Con respecto al teclado (si existe), disponemos de:

- isKeyPressed*(Input.Keys)*
- *isKeyJustPressed(Input.Keys)*
  que nos informa de si en ese momento se está presionando la tecla indicada (*)

Con respecto al ratón disponemos de:

- isButtonPressed(Input.Buttons)
- isButtonJustPressed(Input.Buttons)
  que nos informan de si en ese momento se está presionando el botón indicado (*)
- getX() y getY()
  que nos devuelven las coordenadas x e y del puntero (al tocar movemos el puntero)

Los parámetro son valores de las enumeraciones incluyen:

- Input.Keys incluye:
  - todas las teclas de cualquier teclado, incluyendo F's, NUM_7, NUMPAD_7, PAGE_UP, PAGE_DOWN, SEARCH, etc.
  - botones especiales de control multimedia: MEDIA_PLAY_PAUSE, MEDIA_STOP, MEDIA_PREVIOUS, MEDIA_NEXT, MEDIA_REWIND, etc.
  - teclas de móvil HOME, BACK, VOLUME_UP, VOLUME_DOWN, CAMERA, etc.
- Input.Buttons incluye botones del ratón Buttons.LEFT, Buttons.MIDDLE, Buttons.RIGHT, etc.

(*) los métodos "Just" nos informa de lo mismo que su análogo, pero no vuelve a devolver true hasta que se dejado de realizar la acción (tocado la pantalla, pulsada la tecla, pinchado con el ratón) y se ha vuelto a realizar de nuevo, por lo que es más adecuado, para no confundir una acción con otra distinta (sin "just" en el render podrían devolverían true decenas de veces en un segundo para una única acción).

Registrar un escuchador

A través del método Gdx.input.setInputProcessor informamos a libGDX del objeto que recibirá los avisos (en el momento en que se produzcan) de los cambios de estado en los dispositivos de entrada (se pulsa una tecla, se toca la pantalla, se realiza un determinado gesto, etc.)

Existen dos interfaces principales que se pueden implementar para registrar el escuchador: InputProcessor y GestureListener.

```
public interface InputProcessor {
   /* Llamados al pulsar una tecla */
   public boolean keyDown (int keycode);
   public boolean keyUp (int keycode);
   public boolean keyTyped (char character);
  /* Llamados cuando se toca la pantalla o se presiona un botón del ratón  */
   public boolean touchDown (int screenX, int screenY, int pointer, int button);
   public boolean touchUp (int screenX, int screenY, int pointer, int button);
   public boolean touchDragged (int screenX, int screenY, int pointer);
   public boolean mouseMoved (int screenX, int screenY);
   public boolean scrolled (float amountX, float amountY);
}
```

donde se pueden consultar los parámetros de entrada como son las coordenadas de pantalla donde se ha tocado (screenX/screenY), la tecla que se ha pulsado o soltado (keycode), el dedo con el que se ha tocado (pointer: desde 0 hasta el número de puntos independientes que detecta la pantalla -1), button (para saber qué botón del ratón estaba pulsado 0-izq,1-der,2-medio), etc.

Cada método devuelve un boolean que será true en el caso de que se haya procesado (se haya tenido en cuenta) la entrada, falso en caso contrario.

Existe una clase denominada InputAdapter que implementa el interfaz InputProcessor devolviendo false en todos los métodos de modo que, en vez de tener que implementarlos todos, puede resultar cómodo extender de ella sobrescribiendo justo los métodos que nos interesan.

```
public static interface GestureListener {
 public boolean touchDown (float x, float y, int pointer, int button);
 public boolean tap (float x, float y, int count, int button);
 public boolean longPress (float x, float y);
 public boolean fling (float velocityX, float velocityY, int button);
 public boolean pan (float x, float y, float deltaX, float deltaY);
 public boolean panStop (float x, float y, int pointer, int button);
 public boolean zoom (float initialDistance, float distance);
 public boolean pinch (Vector2 initialPointer1,Vector2 initialPointer2,Vector2 pointer1,Vector2 pointer2);
 public void pinchStop ();
}
```

Para registrar un *GestureListener*, le pasaremos a *setInputProcessor* un objeto *GestureDetector*, que a su vez recibe el objeto que implementa los métodos de dicho interface.

Igual que existía la clase InputAdapter para el interfaz InputProcessor, existe la clase *GestureAdapter* que implementa *GestureListener*.

Otros mecanismos de E/S

Podemos consultar la existencia y los valores del acelerómetro:

```
Gdx.input.isPeripheralAvailable(Input.Peripheral.Accelerometer)
Gdx.input.getAccelerometerX()   getAccelerometerY()   getAccelerometerZ()
```

Podemos consultar la existencia y utilizar el vibrador:

```
Gdx.input.isPeripheralAvailable(Input.Peripheral.Vibrator)
Gdx.input.vibrate(milisegundos);  Gdx.input.cancelVibrate();
```

Uso de un multiplexor

Un InputMultiplexer es una clase que nos permite crear un multiplexador de procesadores de entrada, es decir, gestionar varios procesadores de entrada simultáneamente.

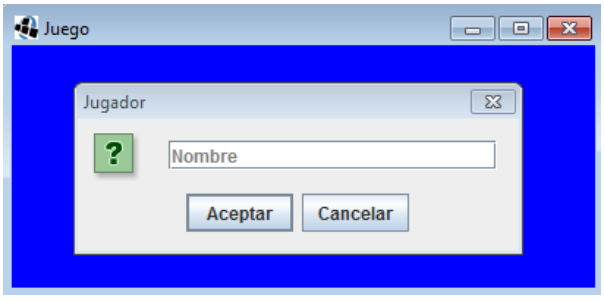
InputMultiplexer implementa InputProcessor, por lo que será el procesador de entrada principal del juego (se le pasará de parámetro a Gdx.input.setInputProcessor), y se le añadirán, por orden de prioridad, los procesadores de entrada (objetos que implementen InputProcessor) que deseemos. Dichos procesadores serán consultados de manera ordenada hasta que alguno informe que ha procesado la consulta (devolviendo true como resultado del método implementado).

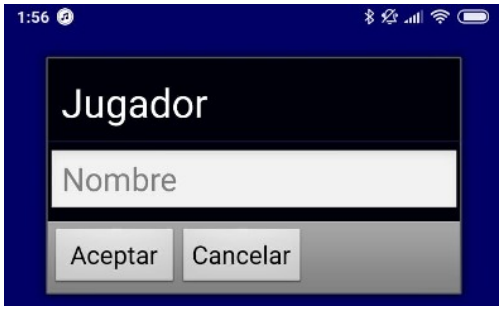
Pidiendo datos a través de una caja de diálogo

Se crea una caja de diálogo con el título y el mensaje proporcionado. Al cerrar la caja de diálogo (aceptando o cancelando) se llama a las funciones implementadas en el listener (ojo: el ciclo de vida de la aplicación no se detiene con la caja de diálogo abierta).

```
public interface TextInputListener {
   public void input (String text);
   public void canceled ();
}
public void getTextInput (TextInputListener listener, String title, String text, String hint);
```

También se puede pasar un último parámetro Input.OnscreenKeyboardType indicando el tipo de teclado que queremos utilizar.





El interface Screen y la clase Game

El interface Screen representa una posible pantalla del juego (principal, ajustes, marcadores, el propio juego, …). https://github.com/libgdx/libgdx/wiki/Extending-the-simple-game

```
public interface Screen {
      @Override public void show() {}
      @Override public void render(float delta)  {}
      @Override public void resize(int width, int height)  {}
      @Override public void pause() {}
      @Override public void resume() {}
      @Override public void hide() {}
      @Override public void dispose() {}
}
```

Los objetos que implementan el interfaz Screen no se liberan de memoria automáticamente puesto que al programador puede interesarle tenerlas en memoria por motivos de rendimiento. Hay que hacerlo manualmente llamando a *.dispose*().

La clase abstracta *Game* nos proporciona una implementación de *ApplicationListener,* añadiendo la gestión de la pantalla actual:

```
void setScreen(Screen screen);
Screen getScreen();
```

Básicamente, usar la clase Game y objetos que implementen Screen nos facilitan estructurar de manera sencilla nuestro juego. El código de la clase Game que nos proporciona libGDX, y que nos ayuda a entender su funcionalidad, es el siguiente:

```
public abstract class Game implements ApplicationListener {
 protected 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());
 }
 @Override public void resize (int width, int height) {
      if (screen!=null) screen.resize(width, height);
 }
 public void setScreen (Screen screen) {
      if (this.screen != null) this.screen.hide();
      this.screen = screen;
      if (this.screen != null) {
         this.screen.show();
         this.screen.resize(Gdx.graphics.getWidth(), Gdx.graphics.getHeight());
      }
 }
 public Screen getScreen () { return screen; }
}
```

Consideraciones sobre optimización

En general, en cualquier aplicación es importante optimizar el código para obtener algunos de los siguientes objetivos:

- mayor rapidez en la ejecución
- menor consumo de recursos (memoria, CPU, …)
- claridad del código para facilitar futuras modificaciones

En el caso de las aplicaciones de juegos (y más si se van a ejecutar en un dispositivo móvil) estas consideraciones son todavía más importantes. Por ello tendremos en cuenta los siguientes consejos:

- Pensar detenidamente cuándo realizar la carga y liberación de recursos. Por ejemplo, podría no interesar cargar las texturas y demás información de los personajes de la segunda pantalla hasta que no se pasa la primera. Si de la pantalla dos no se puede volver a la primera, pues podríamos liberar la memoria utilizada por los recursos de la primera. Aunque dependiendo del tiempo de carga, puede interesar cargar todo al principio para que la transición entre pantalllas no se ralentice. La solución óptima depende sobre todo de las particularidades de cada juego.
- Estudiar bien el código de renderizado del juego. Debido a sus características especiales, al ser un código que se ejecuta continuamente, cualquier optimización, por mínima que sea, redunda en que consigamos más FPSs.
- Reutilizar objetos cuando sea posible. Es decir, si necesitamos un objeto puntualmente en un método, y este método se llama en una estructura repetitiva, será mejor reutilizar un objeto asignando su nuevo estado que crear un objeto nuevo en cada iteración. De esta manera agilizamos el propio programa y el sistema operativo, pues minimizamos el uso de la memoria y evitamos trabajo futuro a la garbage collection.

Un buen ejemplo de esta reutilización sería el uso de zonas de colisión en los personajes de un videojuego. La hitbox es una (o varias) figuras (rectángulos/círculos/etc.) que acompaña al personaje, pero es importante:

- actualizarla solo cuando cambia algo que le afecta (posición, tamaño…). Por ejemplo, solo actualizamos la posición si el personaje se está moviendo.
- actualizar solo lo que cambia (p.e. solo se cambia la coordenada y si se está moviendo verticalmente, no todo su estado)
- no crear un objeto nuevo continuamente (en cada render) perdiendo la referencia antigua

Object pooling

Así es como se denomina el principo de reutilización de objetos, en contraposición a crearlos de nuevo cuando son necesarios.

Generalmente realizaremos una estructura (array) de objetos “inactivos”, al cual añadiremos dicho objetos cuando ya no se vayan a utilizar (p.e., personajes que se hayan salido de la pantalla). Cuando necesitamos crear uno nuevo, si el array no está vacío extraeremos uno y actualizaremos su estado (con un código ~similar a su constructor).

Esta estructura, que puede ser perfectamente programada, ya nos la facilita libgdx a través de las siguientes características: clase Pool e interface Poolable.

Un Pool<> gestiona (a través de su parámetro <>) un tipo de objetos. Por ejemplo, podemos tener un Pool<Bala>, o Pool<Pez>.

Los objetos se obtendrán a través del método *obtain* y se devolverán al pool a través del método *free* (que llamará automáticamente a su método void reset() si implementa el interface Poolable)

Como la clase Pool es abstracta (puesto que su método newObject() es abstracto), deberemos crear una subclase implementando dicho método.

Existe también el método fill(int size) para añadir un cierto número de objetos nuevos.

A continuación podemos ver una implementación del código necesario para un pool de balas.

Dicho código está en la documentación de libgdx: https://libgdx.com/wiki/articles/memory-management

```
public class World {
   private final Array<Bullet> activeBullets = new Array<Bullet>(); // array containing the active bullets.

   private final Pool<Bullet> bulletPool = new Pool<Bullet>() {     // bullet pool.
      @Override
      protected Bullet newObject() {
         return new Bullet();
      }
   };

   public void update(float delta) {
      // if you want to spawn a new bullet:
      Bullet item = bulletPool.obtain();
      item.init(2, 2);
      activeBullets.add(item);

      // if you want to free dead bullets, returning them to the pool:
      int len = activeBullets.size;
      for (int i = len; --i >= 0;) {
         item = activeBullets.get(i);
         if (item.alive == false) {
            activeBullets.removeIndex(i);
            bulletPool.free(item);
         }
      }
   }
}
```

```
public class Bullet implements Pool.Poolable {
   public Vector2 position;
   public boolean alive;

   /* Bullet constructor. Just initialize variables. */
   public Bullet() {
      this.position = new Vector2();
      this.alive = false;
   }

   /* Initialize the bullet. Call this method after getting a bullet from the pool. */
   public void init(float posX, float posY) {
      position.set(posX,  posY);
      alive = true;
   }

   /* Callback method when the object is freed. It is automatically called by Pool.free()
    Must reset every meaningful field of this bullet. */
   @Override
   public void reset() {
      position.set(0,0);
      alive = false;
   }

   /* Method called each frame, which updates the bullet.*/
   public void update (float delta) {
      // update bullet position
      position.add(1*delta*60, 1*delta*60);

      // if bullet is out of screen, set it to dead
      if (isOutOfScreen()) alive = false;
   }
}
```