PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Desarrollo de videojuegos
2D
3D

Desarrollo de un juego 2D. Interfaz y motor

El primer paso que vamos a llevar a cabo para el desarrollo de un juego en 2D es el diseño de una **interfaz que nos permita jugar** y de un **motor muy básico** que permita a nuestro videojuego funcionar.

Para crear la pantalla, crearemos una clase que herede de **SurfaceView**, que es una clase que nos permitirá **pintar elementos en su interior**, como si se tratase de un lienzo. Al crear esta clase, podremos crear un **objeto** de ella en la **interfaz gráfica**. Esta clase deberá sobrescribir el método onDraw, el cual se encargará de dibujar todos los **elementos en la pantalla**, y al que le llegará por parámetro un objeto de la clase **Canvas**, que será nuestro "lienzo".

La clase PantallaVideojuego la hemos creado en un paquete llamado vista, por lo que para crear un elemento gráfico de este tipo, deberemos escribir el código mostrado en la siguiente figura.

Creando la pantalla en la interfaz gráfica

```
<vista.PantallaVideojuego
   android:id="@+id/pantallajuego"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:layout_marginLeft="10dip"
   android:layout_marginTop="10dip"
   android:layout_marginRight="10dip"
   android:layout_marginBottom="10dip">
</vista.PantallaVideojuego>
```

Una vez hecho esto, podremos crear una clase que representará el motor de nuestro videojuego.

Para ello crearemos una **clase** que **heredará de Thread** y que tendrá los siguientes **atributos**:

- FPS: Será la velocidad, es decir, los **Frames Por Segundo**, a la que se moverá nuestro videojuego.
- Un **objeto** de la clase PantallaVideojuego.
- Running: Esta variable nos dirá si el videojuego está corriendo o no.
- Lienzo: Esta variable del **tipo Canvas** será la encargada de **dibujar los elementos** en pantalla.
- Variables para controlar las pulsaciones en los botones y en pantalla. Controlarán que haya un intervalo de tiempo entre pulsación y pulsación.

Todos los elementos necesarios para la interfaz los podemos encontrar en la parte de recursos del tema, concretamente para este caso, en la versión 1 de las clases:

- PantallaVideojuego.java
- Motor.java

```
public class Motor extends Thread
{
   // Atributos de la clase
   static final long FPS = 25;                 // Frames Por Segundo para pintar en el juego
   private PantallaVideojuego pantalla;        // Pantalla donde se va a jugar
   private boolean running = Boolean.FALSE;  // Indica si el motor está en marcha o no
   private Canvas lienzo;                      // Lienzo para dibujar
   private long ticksPS, startTime, sleepTime; // Variables para actualizaciones
```

[...]

```
public class PantallaVideojuego extends SurfaceView
{
   private Bitmap imagenspritebueno, imagenspritemalo, fondo;
   private SurfaceHolder holder;
   private Motor motor;
   private Sprite spritebueno, spritemalo;
   private BitmapDrawable fondorepetido;
```

[...]

Fases del desarrollo de videojuegos

En el proceso de desarrollo de videojuegos hay que seguir una serie de fases muy similares a la del desarrollo de cualquier proyecto software, concretamente fase:

- de concreción,
- de diseño,
- de planificación,
- de producción,
- de pruebas,
- de distribución
- y de mantenimiento.

Cada una de ellas es fundamental para un buen resultado de cualquier videojuego.

En esta unidad, iremos un poco **al grano** y no vamos a poder realizar todas y cada una de estas fases, sino que implementaremos un juego básico en 2D paso a paso para que la **parte de producción** que sería la más técnica, quede lo suficientemente clara.

No obstante, no hay que olvidar que el resto de fases poseen igual, o casi mayor importancia…

Agregando los sprites

De unidades anteriores, ya sabemos que los sprites son las imágenes que contienen los movimientos de nuestros personajes. Nosotros vamos a utilizar un sprite bastante simple en el que tendremos movimientos hacia la derecha, izquierda, arriba y abajo, únicamente. Todavía, nada de ataques, muerte, etc.

Para crear sprites personalizados más completos que el que vamos a utilizar, podemos utilizar la siguiente web:

http://gaurav.munjal.us/Universal-LPC-Spritesheet-Character-Generator/

Para la **creación de los sprites**, vamos a **necesitar dos elementos**:

- Un enumerado llamado **Direccion** que tendrá las posibles direcciones en las que se moverá nuestro personaje.
- Una clase Sprite que será la encargada de la gestión de nuestros sprites.

Con los siguientes métodos:

- Un método para generar una **posición aleatoria** dentro de nuestra pantalla.
- Un método para **cambiar las direcciones** de movimientos de nuestro personaje. Uno por posible dirección.
- Un método para **dibujar nuestro personaje**.
- Un método para **cargar el sprite** del personaje.

Lo primero será agregar en la clase PantallaVideojuego los atributos necesarios para la **gestión de los sprites y el motor**.

A continuación agregamos la **funcionalidad** al motor para que pinte nuestro sprite.

Todos los elementos necesarios para el agregado de sprites los podemos encontrar en la parte de recursos del tema, concretamente para este caso, en las clases:

- PantallaVideojuego.java, versión 2

```
public class PantallaVideojuego extends SurfaceView
{
   private Bitmap imagenspritebueno, imagenspritemalo, fondo;
   private SurfaceHolder holder;
   private Motor motor;
   private Sprite spritebueno, spritemalo;
   private BitmapDrawable fondorepetido;
```

- Dirección.java

```
public enum Direccion
{
   ARRIBA("ARRIBA", 3),
   ABAJO("ABAJO", 0),
   IZQUIERDA("IZQUIERDA", 1),
   DERECHA("DERECHA", 2);
   private String stringValue;
   private int intValue;
```

- Sprite.javavideojuego.

```
public class Sprite
{
   /*
    *    Configuración del sprite
    *
    *    He supuesto las direcciones, casillas del vector
    *
    *  dirección = 0 up, 1 left, 2 down, 3 right
    *  animación = 3 up, 1 left, 0 down, 2 right
    *
    *  En la posición 0 del vector -> imagénes para moverse hacia abajo
    *  En la posición 1 del vector -> imagénes para moverse hacia la izquierda
    *  En la posición 2 del vector -> imagénes para moverse hacia la derecha
    *  En la posición 3 del vector -> imagénes para moverse hacia arriba
    */
   int[] DIRECCION_ANIMACION = { 3, 1, 0, 2 };

   private static final int BMP_ROWS = 4;
   private static final int BMP_COLUMNS = 3;
   private Direccion direccion;
   private int x = 0;
   private int y = 0;
   private static int INCREMENTO_VELOCIDAD = 5;
   private int velocidadEjeX = INCREMENTO_VELOCIDAD;
   private int velocidadEjeY = INCREMENTO_VELOCIDAD;
   private PantallaVideojuego pantalla;
   private Bitmap imagensprite;
   private int frameactual = 0;
   private int ancho;
   private int alto;
   private Random aleatorio;
```

Estructura de los sprites

Una de las cosas que más sorprenden la primera vez que vemos cómo está diseñado un videojuego en 2D es precisamente la configuración de los personajes en los sprites. Lo más lógico es pensar que cada posible movimiento del personaje se guardase en una imagen diferente y que según el movimiento que queramos hacer, cargar una imagen u otra. Esto que parece de sentido común, no es nada eficiente, ya que tendríamos que cargar en memoria una gran cantidad de imágenes simultáneamente en cuanto se iniciara el juego, cosa que, teniendo una única imagen con todos los movimientos, no se tendrá que llevar a cabo de esa forma.

El tener un único sprite tiene también sus inconvenientes, ya que tendremos que saber de antemano dónde están cada uno de los movimientos del personaje dentro del sprite, por ejemplo, el movimiento hacia adelante son los elementos que están en las posiciones (1, 1), (1, 2) y (1, 3), y tendremos que poder obtener esos "fragmentos" de alguna forma.

Empleando siempre la **misma configuración del sprite** en cuanto a **movimientos**, y una vez que podamos "**recortar" esos fragmentos en memoria**, el juego será mucho más **rápido y fluido**.

Moviendo al personaje

El siguiente paso va a ser mover a nuestro personaje. En primer lugar, deberemos dar funcionalidad a nuestro motor para poder hacer que se pinte en pantalla el personaje. Esto lo haremos en la función run. Básicamente, lo que tendremos que hacer será **bloquear el Canvas** de la pantalla para que solo pueda ser utilizado por el motor, y así indicar a la pantalla que **pinte nuestro personaje**. Así, en cuanto creemos un objeto de nuestra pantalla en la clase principal y lo liguemos con su elemento de interfaz gráfica, tendremos un personaje moviéndose en una única dirección.

En este momento del desarrollo, existirá una estela dejada por el personaje al moverse. Se solventará más adelante.

A continuación, el objetivo es hacer que el personaje se mueva a los lados, y para ello, tendremos que dar funcionalidad a los botones y hacer que llamen a los métodos de cambiar dirección de la pantalla que creamos anteriormente.

El botón de 'pausar' llamará al método pausar de la **pantalla**, que a su vez llamará al método de pausar del **motor** del videojuego.

El botón de 'reseteo' servirá para cambiar el sprite de nuestro personaje en caso de tener más de uno.

Código para el movimiento del personaje en MainActivity.java

```
@Override
public void onClick(View view)
{
   switch (view.getId())
   {
      case R.id.ibAbajo:
         pantallajuego.cambiarDireccionAbajo();
         break;
      case R.id.ibArriba:
         pantallajuego.cambiarDireccionArriba();
         break;
      case R.id.ibDerecha:
         pantallajuego.cambiarDireccionDerecha();
         break;
      case R.id.ibIzquierda:
         pantallajuego.cambiarDireccionIzquierda();
         break;
      case R.id.ibPausa:
         pantallajuego.pausar();
         break;
   }
}
```

Puedes encontrar la implementación del movimiento al completo en el apartado de recursos del tema, concretamente en el archivo Sprite.java versión 2:

```java
/**
 * Cambia la dirección de movimiento del sprite hacia la derecha
 */
public void cambiarDireccionDerecha()
{
   if( !movimientoXDerecha() ) // Si el sprite no se está moviendo hacia la derecha
      cambiarSentidoEjeX();  // Cambio el sentido hacia la derecha

   if( direccion == Direccion.ARRIBA || direccion == Direccion.ABAJO )
      // Si el sprite se está movimiento hacia arriba o hacia abajo
   {
      velocidadEjeX = INCREMENTO_VELOCIDAD;
      // Hago que se mueva en el eje x, hacia la derecha o izquiereda
      velocidadEjeY = 0;
      // Deja de moverse en el eje y, hacia arriba o abajo
   }

   direccion = Direccion.DERECHA;
   // Hago que el sprite se mueva hacia la derecha
}
```

[...]

Agregado de fondo

El fondo que se utiliza en un videojuego no es más que una imagen donde están dibujados todos los personajes del videojuego.

Podemos tener dos tipos de fondos:

- Fondos de color: El fondo del videojuego será un color sólido.
- Fondos de imagen: El fondo del videojuego será una imagen que nosotros preparemos previamente.

Para dibujar el fondo del videojuego, basta con hacerlo en la **clase PantallaVideojuego** en el **método onDraw** justo antes de dibujar los personajes, ya que si primero dibujamos los personajes y luego el fondo, este se dibujará encima de ellos y no se verán.

```
public void onDraw(Canvas canvas)
{
   if(canvas != null)
   {
      // Dibujo el fondo del juego
      if(fondorepetido != null)
         fondorepetido.draw(canvas);
      // Dibujo el personaje
      if(spritebueno != null)
         spritebueno.onDraw(canvas);
```

Nosotros vamos a utilizar la imagen que se muestra en la siguiente figura como fondo.

Esta imagen **no va a cubrir la pantalla completa de nuestro dispositivo**, así que lo que tendremos que hacer es **dibujarla tantas veces como sea necesario** para que cubra la totalidad de la pantalla, pero ¿cómo podemos hacer esto?

Utilizaremos para ello las clases Bitmap y BitmapDrawable, y sus **métodos setBounds, setTileModeX y setTileModeY** para hacer que **repita la imagen** hasta rellenar.

Puedes encontrar la implementación del fondo al completo en el apartado de recursos del tema, concretamente en el archivo Pantallavideojuego.java versión 3.

Motores comerciales y open source

Como ya vimos en la unidad anterior, los motores de los videojuegos son entornos de desarrollo especialmente enfocados a facilitar la programación de este tipo de software. Como casi todo software, los motores pueden ser privativos u open source.

Ejemplos de motores de videojuegos privativos pueden ser

- Unity 3D,
- Unreal Engine.

Ejemplos de motores de videojuegos open source pueden ser

- Xenko,
- Godot Engine.

Algunas de las propiedades con las que trabajan los motores son **luz, texturas, reflejos, sombras**... que permitirán hacer al videojuego lo más **realístico** posible. Con respecto a las sombras, se utilizan lo que se conoce como Shaders, que son los elementos encargados de gestionar todo lo relacionado con las sombras de los objetos y personajes del juego, contribuyendo así, a otorgar el **mayor realismo** posible.

Agregando los enemigos

Los enemigos serán otro personaje cargado con un **sprite** diferente, que se estará moviendo de forma aleatoria en nuestra pantalla. El sprite de nuestro enemigo será el mostrado en la siguiente figura.

Como podemos observar, es muy parecido al sprite de nuestro personaje, de hecho, podríamos intercambiarlos y no provocaríamos ningún error en la implementación a nivel de código. Para el agregado del enemigo, deberemos crear otro **objeto de tipo sprite** en nuestra clase **PantallaVideojuego** y después seguir los siguientes pasos:

```
private Sprite spritemalo;
```

- En el método onDraw, deberemos pintar al enemigo después de pintar el fondo de pantalla.

```
if(spritemalo != null)
   spritemalo.onDraw(canvas);
```

- En el método funcionalidad, deberemos cargar y crear el sprite de nuestro enemigo.

```
spritemalo = new Sprite(this, imagenspritemalo);
```

• Una vez hecho esto, podremos compilar nuestro videojuego y tendremos el resultado que se muestra en la siguiente figura.

Una posible mejora podría ser la incorporación de sonido al juego, por ejemplo, al aplastar a un enemigo. Para ello, podemos emplear las clases MediaPlayer y SoundPool.

Puedes encontrar la implementación de la inclusión del enemigo al completo en el apartado de recursos del tema, concretamente en el archivo Pantallavideojuego.java versión 4.

Detectando las colisiones

Exploraremos cómo detectar colisiones entre dos sprites en un videojuego 2D. Recordemos que una colisión ocurre cuando dos sprites se cruzan o, en otras palabras, cuando ocupan una cierta parte de la pantalla simultáneamente. Para lograr esto, hemos creado una función llamada "hayColision" en nuestra clase "Sprite".

```
public boolean hayColision(float x2, float y2)
{
   return (x2 > x) && (x2 < (x + ancho)) && (y2 > y) && (y2 < (y + alto));
}
```

Esta función toma como parámetros las **coordenadas del sprite** con el que queremos verificar la colisión y devuelve verdadero o falso, indicando si se ha producido la colisión o no.

La función realiza el cálculo dentro de este código para determinar si las coordenadas (x2, y2) que le proporcionamos están dentro del área cubierta por las coordenadas del sprite con el que realizamos la llamada.

Además, hemos creado dos métodos adicionales, "getX" y "getY", que devuelven las posiciones X e Y del sprite, es decir, su ubicación en el momento de la llamada.

```
/**
 * Devuelve la coordenada X del sprite
 * @return Posicion X del sprite
 */
public float getX()
{
   return x;
}

/**
 * Devuelve la coordenada x del sprite
 * @return Posición y del sprite
 */
public float getY()
{
   return y;
}
```

Nos dirigimos a la clase "PantallaVideojuego". En el método "onDraw", donde dibujamos elementos en pantalla, incluiremos el fondo y los sprites de nuestro personaje y del enemigo. Luego, verificaremos si hay una colisión entre el sprite de nuestro personaje y las coordenadas X e Y del enemigo. Si se detecta una colisión, se imprimirá un mensaje en la consola indicando "Colisión detectada".

```
if( spritebueno.hayColision(spritemalo.getX(), spritemalo.getY()) )
{
   System.out.println("COLISION DETECTADA");
}
```

Finalmente, ejecutamos nuestra aplicación para observar el resultado. En la consola, al hacer que nuestro personaje colisione con el enemigo, verificamos que se ha detectado una colisión entre los dos personajes.

Agregando texto al juego

Aprenderemos cómo dibujar texto en la pantalla de nuestro videojuego 2D. Para lograrlo, nos dirigimos a la clase "PantallaVideojuego" y creamos una función llamada "dibujarTexto", la cual se encargará de dibujar el texto en el lienzo. Esta función aceptará los siguientes parámetros:

- "texto" (el texto a dibujar),
- "x" e "y" (las coordenadas donde se ubicará el texto dibujado),
- "tamañoTexto" (el tamaño del texto),
- "rojo", "verde" y "azul" (componentes RGB del color del texto),
- y "canvas" (el lienzo donde se dibujará el texto).

Utilizaremos un objeto del tipo "Paint" para dibujar el texto en la pantalla.

Crearemos una fuente del tipo "Typeface", en este caso, será de tipo "Serif", cursiva y en negrita. Asignamos esta fuente a nuestro pincel para que dibuje con ella.

Utilizaremos el método "drawText" de "canvas" para pintar el texto deseado. El color se especificará en formato ARGB, por lo que se deberá proporcionar un grado de transparencia. En este caso, se establece en 255, lo que significa que no será transparente; si fuese 0, sería totalmente transparente.

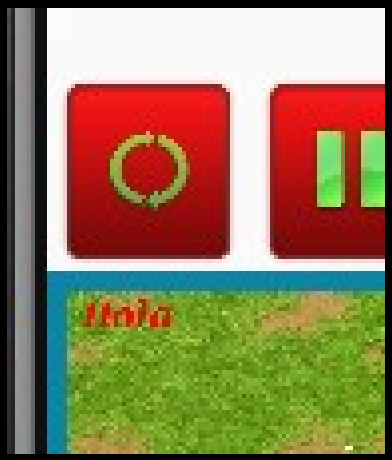
```
private void dibujarTexto(String texto, int x, int y, int tamatexto, int rojo, int verde, int azul,
Canvas canvas)
{
   // no será transparebte:
   int transparencia = 255;
   Paint pincel1 = new Paint();
   /*
    * Configuramos el color del texto, que ha de estar en formato ARGB
    */
   pincel1.setARGB(transparencia, rojo, verde, azul);
   pincel1.setTextSize(tamatexto);
   pincel1.setTypeface(Typeface.SERIF);
   @SuppressLint("WrongConstant") Typeface tf = Typeface.create(Typeface.SERIF,
Typeface.ITALIC | Typeface.BOLD);
   pincel1.setTypeface(tf);
   canvas.drawText(texto, x, y, pincel1);
}
```

Después de dibujar todos los elementos del juego y verificar colisiones en el método "onDraw", añadiremos la instrucción para dibujar el texto "hola" en la posición 20-50, con un tamaño de texto de 50 y utilizando los componentes de color proporcionados, lo cual resultará en un color rojo.

```
// Dibujo el texto
dibujarTexto("Hola", 20, 50, 50, 255, 0, 0, canvas);
```

Al ejecutar el juego, observaremos el resultado:

el texto "hola" en rojo, en cursiva y en negrita en la pantalla.



Concepto de animación en el desarrollo de videojuegos 3D

El desarrollo de videojuegos en 3D es algo más complejo de realizar que el de 2D, debido a que interviene una dimensión más: la profundidad, lo que complicará los movimientos de los personajes de forma exponencial. Este tipo de videojuego **no se presta tanto a la realización desde cero**, **como** hemos hecho en los puntos anteriores con el **juego en 2D**.

En un videojuego 3D, ¿cómo se representan los personajes? Mediante modelos.

A continuación, vamos a conocer una serie de **conceptos básicos** para la creación de un videojuego en 3D:

• Mundo: El concepto de mundo en los videojuegos 3D se refiere al **espacio** donde se desarrollará el videojuego, es decir, por donde podrán **moverse los personajes** del mismo.

• Sistema de coordenadas: Los videojuegos en 2D tenían únicamente dos coordenadas, el alto y el ancho, sin embargo, cuando tratamos con videojuegos en 3D, tendremos las tres dimensiones del espacio, que son alto, ancho y profundidad. Este sistema de coordenadas tendrá que ser representado de una forma totalmente diferente al de dos dimensiones, utilizando ejes cartesianos con **tres coordenadas: X, Y, y Z**.

• Vectores: Un vector no es más que un **conjunto de coordenadas**, **tres** en el caso de los videojuegos 3D, que representarán el **punto exacto del mundo** donde se encuentra un **objeto**.

• Creación de componentes: Para la creación de los componentes del mundo en los videojuegos 3D, no se utilizarán sprites, sino que tendremos que **crear figuras** mediante el uso de:

- **vértices**,
- **polígonos**,
- **mallas** de polígonos.

• Grafo de escena: Los grafos de escena van a contener toda la **estructura de nuestro videojuego 3D en su conjunto**. Tienen **aspecto de árbol** y de él "colgarán" todos los elementos del mismo. Es un **análogo** a cuando **pintamos los elementos** en el motor de nuestro videojuego **2D**.

Grafos de Escena en Videojuegos: Estructura y Funciones

Exploraremos qué son los grafos de escena en un videojuego y cuál es su utilidad. Un grafo de escena es un **grafo dirigido acíclico** compuesto por **nodos** que contiene los datos que definen un **escenario virtual** y controlan su **proceso de dibujado**.

Su **función principal** es almacenar la información del escenario virtual, representando todos los **objetos** del videojuego **mediante nodos**.

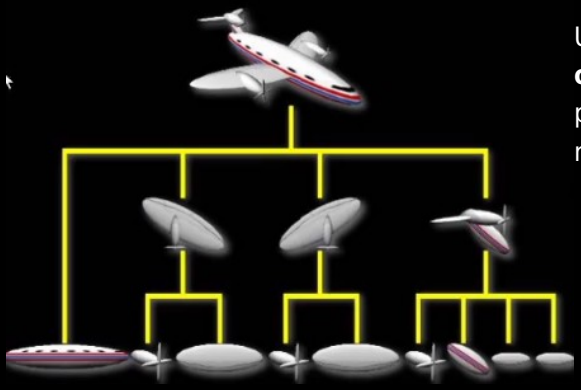
Existen diversos **tipos de nodos**, **algunos** de los cuales almacenan **información geométrica** de los objetos en la pantalla.

Las **funciones clave de un grafo de escena** en un videojuego son:

1. **Organización Lógica de la Escena**: Contribuye a establecer una estructura ordenada y conexa en la escena virtual.
2. **Dependencias Jerárquicas**: Establece dependencias jerárquicas entre los nodos de cada objeto, facilitando la organización.
3. **Selección de Nodos**: Permite la selección de varios nodos, incluso en diferentes niveles.
4. **Proceso de Culling (sacrificio)**: Posibilita la **eliminación automática** de objetos fuera del campo de visión.
5. **Acceso Cómodo a Librerías Gráficas**: Facilita el acceso a librerías gráficas de bajo nivel, como **OpenGL**.

En este ejemplo de un **grafo de escena** de un avión, observamos **tres niveles**:

1. el superior con el **avión completo**.
2. dos niveles más donde se **descompondrán las piezas** del avión, como el cuerpo del avión que se muestra.



Un grafo de escena es fundamental para establecer una **organización lógica y jerárquica** en los videojuegos, permitiendo un **control eficiente** de los objetos y mejorando la interacción visual.

Escena de un juego real

Como ya sabemos después del estudio del tema, un videojuego se va a componer de multitud de motores y elementos.

Un ejemplo claro del tipo de componentes que podemos encontrar en un juego, lo tenemos en la imagen mostrada , que representa una escena real del videojuego *Dark Souls Remasteres*, en la que podemos apreciar varios elementos como:

- nuestro **personaje** de espaldas,
- un **dragón** al fondo,
- y el escenario.

Dentro del escenario podemos observar un **juego de luz** que incide desde la parte trasera, haciendo que resulten sombras, como por ejemplo la de nuestro personaje.

También se puede apreciar la **interfaz del juego**, que se compone de

- la **vida y estamina**, en la parte superior,
- de nuestra **arma, escudo y número de almas**, en la parte inferior.

A groso modo, podemos deducir que como mínimo se están utilizado **tres grafos**:

- uno del personaje,
- otro del dragón
- y otro del propio escenario.

Te invitamos a que sigas analizando la imagen para que descubras más grafos y elementos...