Tema 6: Desarrollo de juegos Android

¿Qué aprenderás?

- Identificar las distintas partes que componen la arquitectura de un juego Android.
- Conocer las fases de desarrollo de un juego Android.
- Comprender cómo se puede aplicar el patrón de diseño MVC en el desarrollo de videojuegos.
- Programar la lógica de un juego Android que responda a las entradas de un jugador.

¿Sabías que…?

- Un juego consta de un bucle principal que se ejecuta de manera infinita hasta que se dan las circunstancias para su finalización.
- La lógica del juego es la parte de código responsable de actualizar el estado de los elementos del juego, desde la posición de los sprites en pantalla hasta las puntuaciones

6.1. Introducción a los vídeo juegos

La estructura de un juego en Android sigue los mismos parámetros que en cualquier otro entorno de programación. Hay que estructurar el proyecto en 3 partes:

- Mecánica de Juego (GameEngine)
- Renderizado (Rendering)
- Interactividad (Events)
- Físicas (Phisics)

La mecánica del juego consiste en un puñado de clases que permiten controlar todo el flujo del juego, así como los elementos que lo componen y todas sus características: Cuando se gana, cuando se pierde, elementos de juego, posición de cada uno, estado, ...

El **renderizado** consiste en dibujar en la pantalla el estado de la partida en cada instante. Toda la información necesaria para ello la obtenemos del bloque de mecánica de juego. **El bloque de interactividad** consiste en captar las acciones del usuario y enviarlas al bloque de mecánica de juego para aplicar sus resultados.

El bloque de físicas sirve para detectar las colisiones e interacciones entre los diferentes elementos de nuestro juego.Los juegos los podemos dividir en categorías a partir de diversas características:

Según el número de dimensiones:

- **2D:** El juego no tiene profundidad se desarrolla en un entorno plano de dos dimensiones (Ej: Tetris)
- **3D:** El juego se desarrolla en un entorno tridimensional (Ej: Quake)

Según el tipo de movimiento:

- **Estáticos o asíncronos:** Los elementos sólo cambian como reacción a una acción del usuario. No hay ningún elemento con un movimiento autónomo (Ej: Buscaminas).
- **Dinámicos o Síncronos:** El juego está formado por fotogramas (frames) que cambian a determinada velocidad (frame rate). Hay una función GameLoop que se ejecuta en cada frame y que lleva a cabo todas las tareas necesarias. De esta manera se pueden crear animaciones y movimientos autónomos de ciertos elementos. (Ej: Galaxians).

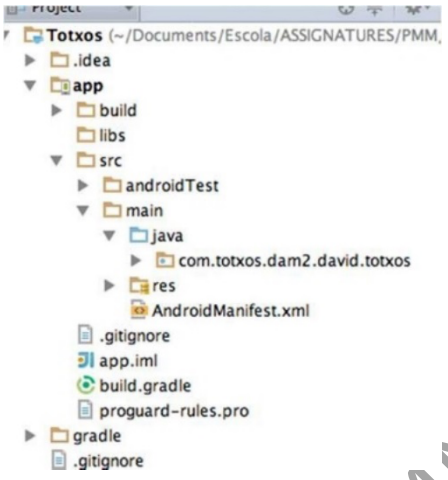
Se pueden combinar los dos tipos de clasificación como podemos observar en la siguiente tabla (aunque los juegos estáticos 3D no son demasiado frecuentes).

	Estáticos	Dinámicos
2D	Buscaminas Apalabrados Solitarios de cartas	Tetris Arkanoid Galaxians
3D	Ajedrez 3D juegos tipo puzle tridimensionales	Quake World of Warcraft

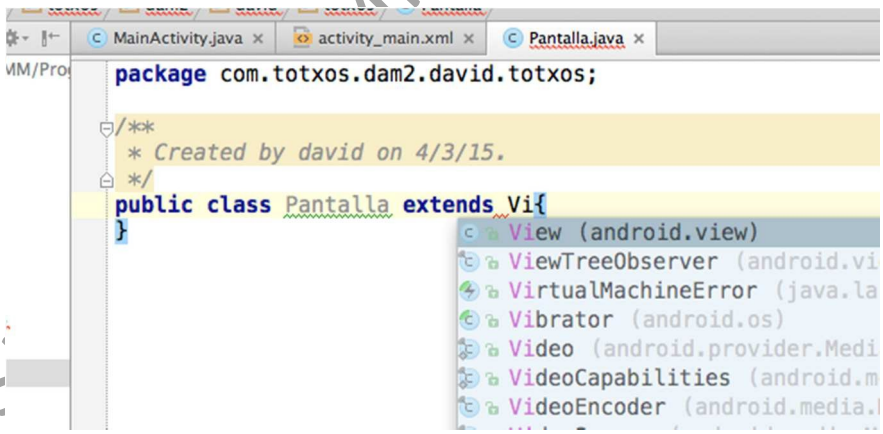
6.2. ¿Cómo diseñar un vídeo juego?

Para aprender los conceptos clave en el diseño de un videojuego lo haremos a través de un ejemplo sencillo en el que dibujaremos una bola y permitir que el usuario la impulse con el dedo. Este ejemplo, aunque muy sencillo, nos permitirá practicar la mayoría de conceptos en un entorno nada complejo.

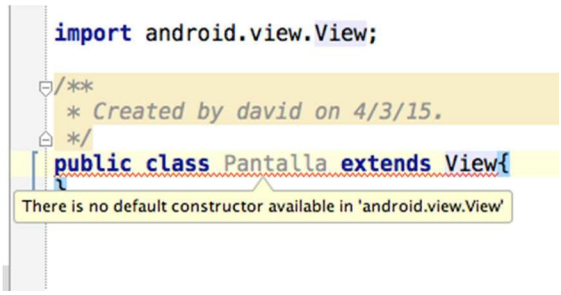
El primer paso es crear un nuevo proyecto con el asistente. Una vez creado desplegamos el árbol de proyecto hasta localizar la sección donde está el código Java:



Entonces añadimos una nueva clase Java llamada Pantalla. Una vez creada añadiremos la cláusula **"extends View"** detrás de la palabra Pantalla asegurándonos de elegir el elemento View **(android.view)** del desplegable emergente.



Esto lo que hace es que esta clase se convierta en un elemento dibujable en la pantalla del móvil. Aquí es donde desarrollaremos todo en **render** de nuestro juego. Observamos que el compilador subraya el nombre de la clase en color rojo indicando un error. Si acercamos el cursor aparece un mensaje emergente indicando cuál es el motivo del error.

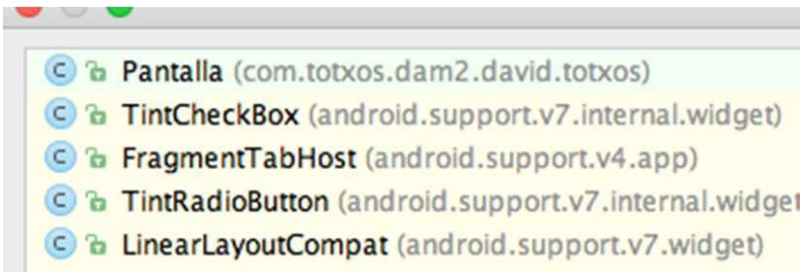


En este caso que nuestra clase no tiene ningún constructor definido. Podemos generar un automáticamente con la opción Generate y después Constructor.

Una vez hecho esto hay que definir una función donde programaremos que se debe dibujar en nuestro control Pantalla. Para ello sobrecargamos la función onDraw. Y de momento todo lo que haremos en dibujar un círculo rojo de radio 100 en la coordenada (0,0).

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    Paint p=new Paint();
    p.setColor(Color.RED);
    canvas.drawOval(new RectF(0,0,100,100),p);
}
```

Ahora hay que poner un elemento de tipo Pantalla en nuestro móvil. Para ello vamos a la zona de recursos **Layout** y abrimos lo que se dice **activity_main.xml**. Ahora veremos el aspecto del móvil en marcha nuestra App. Lo que tenemos que hacer es eliminar el elemento de texto que hay de forma predeterminada ya continuación el menú de componentes y tirar el último de todos **CustomView**. Ahora se muestra una ventana o (entre muchos otros) está nuestro elemento Pantalla.



El tiramos y lo colocamos en nuestro **layout**. Observamos que no tiene muy buena pinta. Para arreglarlo vamos a la vista de código (solapa Texto de la parte inferior) y modificamos las propiedades **layout_heigh** y **layout_width** para que tengan el valor **match_parent** a la propiedad id le ponemos Pantalla y eliminamos cualquier propiedad **margin** que haya. El código debe quedar así:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    tools:context=".MainActivity">
    <view
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        class="com.totxos.dam2.david.totxos.Pantalla"
        android:id="@+id/Pantalla"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true" />
</RelativeLayout>
```

Ahora si volvemos a la vista gráfica el control ya ocupa toda la pantalla del móvil, pero todavía no vemos el círculo. Si nos fijamos en la parte inferior nos recomienda que compilemos el proyecto. Para ello vamos al menú **Build** y elegimos **Rebuild Project**, ahora ya vemos nuestro círculo.

Podemos ejecutar nuestra App en un emulador o teléfono para comprobar cómo funciona. Uno de los hechos más inconvenientes de las aplicaciones para móvil es la diferencia entre las dimensiones de la pantalla de una móvil a otro.

Esto hace que el aspecto de nuestra aplicación cambie mucho en función del teléfono donde se ejecuta. Para solucionar este hecho utilizaremos un sistema de coordenadas doble:

- **Coordenadas de Mundo (World coordinates):** Estas son nuestras coordenadas reales (en píxeles). Nuestro mundo puede tener el tamaño que nosotros queramos y nunca dependerá del móvil cliente.
- **Coordenadas de pantalla (Screen coordinates):** Estas son las coordenadas (en píxeles) de la pantalla. Estas sí que dependen del móvil cliente Para pasar de coordenadas de pantalla en el mundo hay que aplicar un factor de escala que hay que determinar en iniciar la aplicación a partir de las dimensiones reales de la pantalla del teléfono.

Por lo tanto, lo primero que hay que hacer es decidir cuál será el tamaño de nuestro mundo. Haremos un mundo de 1000x1000 píxeles. Crearemos dos variables en nuestra clase pantalla para guardar estos valores. A continuación, crearemos dos variables más para guardar la información sobre el ancho y alto reales de la pantalla del móvil. Entonces utilizaremos las funciones **getWidth** y **getHeight** en el constructor para consultar los valores.

```
public class Pantalla extends View{
    public int anchoMundo,altoMundo,ancho,alto;
    public Pantalla(Context context, AttributeSet attrs) {
        super(context, attrs);
        anchoMundo =1000;
        altoMundo =1000;
        ancho=this.getWidth();
        alto=this.getHeight();
}
```

El problema de utilizar las funciones getHeight y getWidth dentro del constructor es que no funcionan. Lo podemos comprobar si dibujamos un rectángulo de color rojo con las dimensiones ancho y alto.

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    Paint p=new Paint();
        p.setColor(Color.RED);
        p.setStyle(Paint.Style.FILL_AND_STROKE);
        canvas.drawRect(new Rect(0,0,ancho,alto),p);
}
```

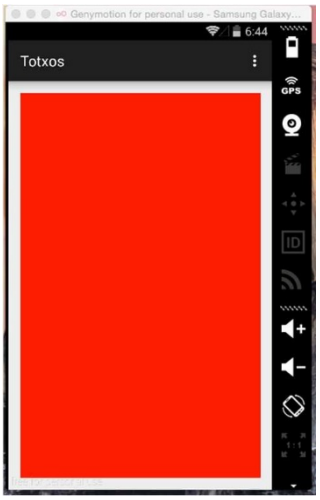
Cuando ejecutamos nuestra aplicación la pantalla es toda en blanco y no aparece ningún rectángulo. La razón es que las funciones **getHeight** y **getWidth** devuelven un 0 en lugar de los valores reales. Esto ocurre porque el móvil no conoce las dimensiones reales de las vistas hasta que las dibuja en la pantalla, por lo tanto, hay que esperar que el móvil dibuje todo el **layout** para consultar las medidas de los controles. Esto lo tenemos que hacer en el constructor de la actividad principal. Por lo tanto, vamos al fichero de **MainActivity.java** y hacemos las siguientes modificaciones:

```
public class MainActivity extends ActionBarActivity {
    Pantalla p;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        p=(Pantalla)findViewById(R.id.Pantalla);
        ViewTreeObserver obs=p.getViewTreeObserver();
        obs.addOnGlobalLayoutListener(new ViewTreeObserver.OnGlobalLayoutListener() {
            @Override
            public void onGlobalLayout() {
            }
        });
    }
```

Hemos creado una referencia a nuestro control pantalla con el nombre p. Y entonces hemos creado un objeto del tipo **ViewTreeObserver** que dispone de un **listener** que nos informará del momento en que el layout está completamente dibujado. Aquí es donde los métodos **getWidth** y **getHeight** nos proporcionan los valores "reales" del tamaño de nuestra pantalla. Por lo tanto, lo que hacemos es añadir las líneas siguientes a la función **onGlobalLayout**:

```
public class MainActivity extends ActionBarActivity {
    Pantalla p;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        p=(Pantalla)findViewById(R.id.Pantalla);
        ViewTreeObserver obs=p.getViewTreeObserver();
        obs.addOnGlobalLayoutListener(new ViewTreeObserver.OnGlobalLayoutListener() {
            @Override
            public void onGlobalLayout() {
                        p.ancho=p.getWidth();
                p.alto=p.getHeight();
            }
        });
    }
```

Ahora si ejecutamos nuestra App ya veremos un rectángulo que ocupa toda la pantalla.



Ahora hay que establecer dos funciones en la clase pantalla que nos transformen las coordenadas de mundo en pantalla y al revés. Para ello hay que establecer el factor de escala de anchura y de altura y quedarnos con el más pequeño de ambos (ya que es la dimensión que nos limita).

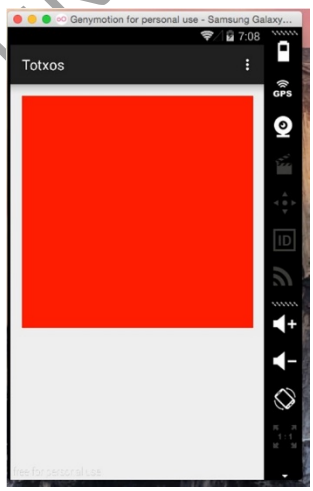
```
public void setEscala()
{
    float escalaX,escalaY;
    escalaX=(float)ancho/(float)anchoMundo;
    escalaY=(float)alto/(float)altoMundo;
    escala=Math.min(escalaX,escalaY);
}
public int world2screen(int worldCoord) {
    return (int)(worldCoord*escala);
}
public int screen2world(int screenCoord) {
    return (int)(screenCoord/escala);
}
```

Ahora sólo hace falta invocar el método **setEscala** tras establecer las dimensiones de nuestra pantalla (esto es en la actividad principal en el método **onGlobalLayout**) y modificar nuestro rectángulo para que tenga las dimensiones de nuestro mundo.

```
public class MainActivity extends ActionBarActivity {
    Pantalla p;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        p=(Pantalla)findViewById(R.id.Pantalla);
        ViewTreeObserver obs=p.getViewTreeObserver();
        obs.addOnGlobalLayoutListener(new ViewTreeObserver.OnGlobalLayoutListener() {
            @Override
            public void onGlobalLayout() {
                        p.ancho=p.getWidth();
                p.alto=p.getHeight();
                        p.setEscala();
            }
        });
    }
```

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    Paint p=new Paint();
    p.setColor(Color.RED);
    p.setStyle(Paint.Style.FILL_AND_STROKE);
    canvas.drawRect(new Rect(0,0,world2screen(anchoMundo),world2screen(altoMundo)),p);
}
```

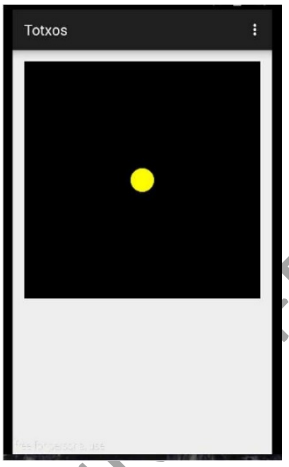
Ahora veremos que al ejecutar la aplicación nos dibuja un cuadrado perfecto (ya que nuestro mundo hace 1000x1000) y ajustamos las dimensiones para que se adapte perfectamente a la pantalla de nuestra móvil (independientemente del tamaño que tenga esta).



Ahora modificamos nuestra función de dibujo para dibujar un fondo de color negro y un círculo de color amarillo de radio 50 (en coordenadas de mundo ya que éstas son independientes del tamaño del móvil) y situado en el centro exacto de nuestro mundo.

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    Paint p1=new Paint();
    p1.setColor(Color.BLACK);
    p1.setStyle(Paint.Style.FILL_AND_STROKE);
    Paint p2=new Paint();
    p2.setColor(Color.YELLOW);
    p2.setStyle(Paint.Style.FILL_AND_STROKE);
    canvas.drawRect(new Rect(0,0,world2screen(anchoMundo),
                                    world2screen(altoMundo)),p1);
    int posX=anchoMundo/2;
    int posY=altoMundo/2;
    int radio=50;
    canvas.drawOval(new RectF(
                world2screen(posX-radio),world2screen(posY-radio),
        world2screen(posX+radio),world2screen(posY+radio)),p2);
}
```

Este es el resultado:



Ahora que hemos cumplido el bloque de Rendering vamos a añadir el bloque de interactividad. Para ello hay que añadir un escuchador de toques. Podemos hacerlo añadiendo a nuestra clase pantalla una clase interna **detectorGestos** que extienda la clase base **GestureDetector.SimpleOnGestureListener**.

```
public class detectorGestos extends GestureDetector.SimpleOnGestureListener
{


}
```

Ahora hay que definir el tipo de gestualidades que nos interesan. En nuestro caso queremos que el usuario pueda impulsar la bola deslizando el dedo sobre la pantalla del móvil. Entonces hay que desarrollar la gestualidad **onFling**. Al hacerlo veremos que aparece un bloque de código como el siguiente:

```
public class detectorGestos extends GestureDetector.SimpleOnGestureListener
{
  @Override
  public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float
  velocityY)
  {
    return super.onFling(e1, e2, velocityX, velocityY);
  }
}
```

Entonces borramos la sentencia **return** y la cambiamos por una que devuelva un valor **true**. Observamos que la función **onFling** nos proporciona como dato la velocidad del dedo en dirección X e Y. Aprovecharemos estos datos para impulsar nuestra bola. Para ello hay que establecer las variables de posX, Posy y radio (que determinan la posición y tamaño de la bola como variables globales de la clase pantalla e inicializarlas en el constructor.

Una vez hecho esto lo que haremos será poner en posX y Posy el valor 50 dentro de nuestra función **onFling** (aunque esto no es el objetivo final, servirá para poner de manifiesto algunos aspectos del movimiento de objetos).

Nuestra clase **detectorGestos** quedará así:

```
public class detectorGestos extends GestureDetector.SimpleOnGestureListener
{
  @Override
  public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float
  velocityY)
  {
    X=50;
    Y=50;
    return true;
  }
}
```

Una vez definida esta clase hay que generar un objeto del tipo **SimpleGestureDetector** y asignarle nuestra clase **detectorGestos** como función de gestión. El código del constructor quedará así:

```
public int anchoMundo,altoMundo,ancho,alto;
públic int posX,posY,radio;
public float escala;
private GestureDetector gestos;
public Pantalla(Context context, AttributeSet attrs) {
  super(context, attrs);
  anchoMundo=1000;
  altoMundo=1000;
  posX=anchoMundo/2;
  posY=altoMundo/2;
  radio=50;
  gestos=new GestureDetector(context,new detectorGestos());
}
```

Ahora sólo queda dirigir todas las gestualidades del usuario hacia nuestro objeto **gestos** para que las pueda procesar. Para ello tenemos que poner una función **OnTouchEvent** a nuestra clase pantalla y dejar el código como se muestra a continuación:

```
@Override
public boolean onTouchEvent(MotionEvent event) {
  gestos.onTouchEvent(event);
  return true;
}
```

Ahora ya podemos ejecutar nuestra App. Observaremos que por mucho que hacemos deslizar el dedo la bola no se mueve. Esto ocurre porque, aunque hemos modificado los valores de X e Y no hemos vuelto a dibujar la bola en la nueva posición. Cada vez que las condiciones de pantalla han cambiado y hay que redibujar debemos ejecutar la sentencia **invalidate()** que fuerza el repintado.

Así el código de la clase **detectorGestos** requiere invocar **invalidate** después de actualizar las variables X e Y.

```
public class detectorGestos extends GestureDetector.SimpleOnGestureListener
{
  @Override
  public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float
  velocityY)
  {
    X=50;
    Y=50;
    invalidate();
    return true;
  }
}
```

¡Ahora ya funciona!

Al deslizar el dedo la bola se desplaza a la esquina superior izquierda de la pantalla.

Para poder mover la bola de forma continua después de que el usuario haya deslizado el dedo por la pantalla hay que generar una animación y esto requiere la creación de un bucle de juego (**GameLoop**).

Para hacer esto hay que ir a la actividad principal y hacer las modificaciones siguientes:

- Crear un objeto de tipo **Runnable** (llamado **GameLoop**).
- Crear un objeto **Handler** (llamado h).
- Utilizar la función **postDelayed** del **Handler** para programar la ejecución de un **GameLoop**.
- Crear una variable **frame_rate** para controlar la velocidad de nuestro juego
- Dentro del **GameLoop** volver a reprogramarlo con un **postDelayed** de **frame_rate**
- milisegundos.

El código quedará así:

```
public class MainActivity extends ActionBarActivity {
    Handler h;
    Pantalla p;
    int frame_rate;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        p=(Pantalla)findViewById(R.id.Pantalla);
        ViewTreeObserver obs=p.getViewTreeObserver();
        obs.addOnGlobalLayoutListener(new ViewTreeObserver.OnGlobalLayoutListener() {
            @Override
            public void onGlobalLayout() {
                p.ancho=p.getWidth();
                p.alto=p.getHeight();
                p.setEscala();
            }
        });
        frame_rate=100;
        h=new Handler();
        h.postDelayed(GameLoop,1000);
    }
    public Runnable GameLoop = new Runnable() {
        @Override
        public void run() {
            h.postDelayed(GameLoop,frame_rate);
        }
    };
```

Ahora tenemos que crear la animación para ello volvemos en nuestra clase Pantalla y hacemos los cambios siguientes:

- Crear dos variables **int** llamadas velX y velY. Darles el valor inicial 0.
- Modificar el código del **onFling** para asignar el valor 50 a las variables velX y velY. Eliminar el resto de líneas salvo **return**.
- Crear un método llamado **nuevo_frame**.
- Escribir en el método **nuevo_frame** el código necesario para sumar velX y velY a posX y PosY respectivamente y luego hacer un **invalidate**.

Ahora volvemos a la actividad principal y hacemos los cambios siguientes:

- Invocar la función **nueva_frame** dentro del **GameLoop**.

Ahora cuando ejecutamos., Al hacer deslizar el dedo la bola empieza a moverse y no se detiene. ¡Ya tenemos nuestra primera animación!

El próximo objetivo es conseguir que la bola rebote en los bordes de la pantalla en lugar de pasar de largo y desaparecer. Esto lo tenemos que hacer a la función **nuevo_frame**. Hay que comprobar si hemos salido de los límites de la pantalla y en caso afirmativo cambiar el signo de la velocidad correspondiente.

Aquí es donde vemos la importancia de usar un sistema de coordenadas absolutamente independiente del tamaño de pantalla del móvil. Ya que sus límites son perfectamente conocidos.

```
public void nuevo_frame() {
    posX+=velX;
    posY+=velY;
    if (posX<50) {
       posX=50;
       velX=-velX;
    }
    if (posY<50) {
       posY=50;
       velY=-velY;
    }
    if (posX>950) {
       posX=950;
       velX=-velX;
    }
    if (posY>950) {
       posY=950;
       velY=-velY;
    }
          invalidate();
}
```

Entonces sólo hace falta que las variables velX y velY iniciales dependan de las que nos proporciona el evento **onFling** así la sensación de que tendrá el usuario se puede controlar la fuerza del impacto.

Es importante dividirlas por la escala ya que de esta manera la sensación no dependerá del tamaño de la pantalla del móvil. Sino en los móviles pequeños la sensación sería que la bola se impulsa con más fuerza porque el espacio de pantalla es más pequeño.

Utilizaremos también un factor de sensibilidad para poder controlar la sensación de fuerza de impulso al deslizar el dedo. Este debe ajustarse de una forma subjetiva por prueba y error hasta obtener la sensación que queremos.

```
public class detectorGestos extends GestureDetector.SimpleOnGestureListener
{
  @Override
  public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float
  velocityY) {
    float sesibl=0.027f;
    velX=(int)((velocityX/escala)*sensibl);
    velY=(int)((velocityY/escala)*sensibl);
    return true;
  }
}
```

Finalmente iremos reduciendo las velocidades velX y velY cada **frame** para que la bola se vaya frenando paulatinamente hasta detenerse. Para hacer esto lo mejor es cambiar el tipo de variable de velX y velY a **float** y multiplicarlas por un valor inferior a 1 y de esta manera ir las disminuyendo poco a poco.

Este valor lo guardaremos en un parámetro llamado frenado que también ajustaremos hasta obtener la sensación de **frenado** que nos interesa. Para evitar el hecho que la bola no termina de detenerse nunca (ya que multiplicando por un número distinto de 0 no se puede lograr un 0) cuando la velocidad en ambas direcciones sea inferior a

1 píxel por **frame** pondremos las velocidades a 0 y consideraremos la bola parada.

El código quedará así:

```
public void nuevo_frame()
{
  posX+=velX;
  posY+=velY;
  if (posX<50) {
    posX=50;
    velX=-velX;
  }
  if (posY<50) {
    posY=50;
    velY=-velY;
  }
  if (posX>950) {
    posX=950;
    velX=-velX;
  }
  if (posY>950) {
    posY=950;
    velY=-velY;
  }
```


Con este proyecto hemos visto de una forma sencilla casi todos los conceptos involucrados en el desarrollo de un juego en Android. Nos quedaría pendiente la detección de colisiones entre objetos de nuestro juego. Este concepto se denomina **físicas** en el mundo del **GameDesign**.

Para desarrollar este concepto la idea es poner una segunda bola quieta y resolver las colisiones con la bola que nosotros impulsamos con el dedo como si se tratara de un juego de billar. Como ahora tenemos dos bolas que hay que controlar, mover, frenar, dibujar, etc ... una buena parte del código que tenemos quedará duplicado.

Para solucionar este problema lo que haremos es crear una nueva clase llamada **Bola** y poner todo el código necesario para gestionar una bola dentro de esta clase.

De esta forma podremos generar tantas instancias de la clase **Bola** como queramos para nuestro juego. De hecho, cuando diseñamos un juego es muy recomendable crear una clase para cada tipo de objeto (**Game Object**) que hay en nuestro juego (independientemente de la cantidad que haya) ya que esto facilita mucho la estructuración del proyecto. Nuestra clase **Bola** será así:

```
public class Bola {
   private float velX,velY;
   private int posX,posY,radio;
   private int color;
   public Bola(int posX,int posY,int radio, int color) {
      this.posX=posX;
      this.posY=posY;
      this.radio=radio;
      this.color=color;
      velX=0;
      velY=0;
   }
   public Rect zona() {
      return new Rect(posX-radio,posY-radio,posX+radio,posY+radio);
   }
   public void impulsar(float velX,float velY) {
      this.velX=velX;
      this.velY=velY;
   }
   public void mover(){
      posX+=velX;
      posY+=velY;
      if (posX<50) {
         posX=50;
         velX=-velX;
      }
      if (posY<50) {
         posY=50;
         velY=-velY;
      }
      if (posX>950) {
         posX=950;
         velX=-velX;
      }
      if (posY>950) {
         posY=950;
         velY=-velY;
      }
   }

   public void frenar(float frenada){
      velX*=frenada;
      velY*=frenada;
      if ((Math.abs(velX)<=1)||(Math.abs(velY)<=1))
      {
         velX=0.0f;
         velY=0.0f;
      }
   }

   public void dibujar(Canvas canvas,float escala){
      Paint p=new Paint();
      p.setColor(color);
      p.setStyle(Paint.Style.FILL_AND_STROKE);
      canvas.drawOval(new RectF((posX-radio)*escala,(posY-radio)*escala,
            (posX+radio)*escala,(posY+radio)*escala),p);
   }
}
```

Ahora a partir de esta clase hacemos las siguientes modificaciones en nuestra clase pantalla para crear dos instancias de **Bola**. Una amarilla (la que nosotros impulsamos con el dedo) y otra de verde que será "pasiva".

```
public class Pantalla extends View{

   private float sensibl,frenada;
   public int anchoMundo,altoMundo,ancho,alto;
   public float escala;
   public Bola b1,b2;
   private GestureDetector gestos;

   public Pantalla(Context context, AttributeSet attrs) {
      super(context, attrs);
      anchoMundo=1000;
      altoMundo=1000;
      b1=new Bola(anchoMundo/2,altoMundo/2,50,Color.YELLOW);
      b2=new Bola(800,150,50,Color.GREEN);
      sensibl=0.04f;
      frenada=0.95f;
            gestos=new GestureDetector(context,new detectorGestos());
   }
   public void setEscala() {
      float escalaX,escalaY;
      escalaX=(float)ancho/(float) anchoMundo;
      escalaY=(float)alto/(float) altoMundo;
      escala=Math.min(escalaX,escalaY);
   }
   public int world2screen(int worldCoord) {
      return (int)(worldCoord*escala);
   }
   public int screen2world(int screenCoord) {
      return (int)(screenCoord/escala);
   }

   @Override
   public boolean onTouchEvent(MotionEvent event) {
      gestos.onTouchEvent(event);
      return true;
   }

   public void nou_frame() {
      b1.mover();
      b1.frenar(frenada);
            invalidate();
   }
   public class detectorGestos extends GestureDetector.SimpleOnGestureListener {
      @Override
      public boolean onFling(MotionEvent e1, MotionEvent e2,
                        float velocityX, float velocityY) {
         b1.impulsar((velocityX/escala)*sensibl,(velocityY/escala)*sensibl);
         return true;
      }
   }

   @Override
   protected void onDraw(Canvas canvas) {
      super.onDraw(canvas);
      Paint p1=new Paint();
      p1.setColor(Color.BLACK);
      p1.setStyle(Paint.Style.FILL_AND_STROKE);
      canvas.drawRect(new Rect(0,0,world2screen(anchoMundo),
                              world2screen(altoMundo)),p1);
      b1.dibujar(canvas,escala);
      b2.dibujar(canvas,escala);

   }
}
```

Como se puede observar ahora el código de la clase pantalla queda mucho más claro y organizado que antes.Ahora lo primero que hay que hacer es generar una función que compruebe si las dos bolas están colisionando. Esto lo haremos a nuestra clase pantalla:

```
private boolean colision()
{
  if (Rect.intersects(b1.zona(),b2.zona()) {
     return true;
  }
  return false;
}
```

Una vez hecho esto en nuestra clase **Bola** crearemos un método estático que a partir de los datos de dos bolas que colisionan resuelva el choque. Para ello se necesitan algunos fundamentos de física y geometría 2D. El código que hay que añadir a nuestra clase bola es el siguiente:

```
  private double modulo(){
     return Math.sqrt((velX*velX)+(velY*velY));
  }
  private double argumento(){
     return Math.atan2(velY,velX);
  }
  public static void colisionar (Bola b1,Bola b2){
     double m1=b1.radio;
     double m2=b2.radio;
     double alfa=Math.atan2(b1.posY-b2.posY,b1.posX-b2.posX);
     double u1X=b1.modulo()*Math.cos(b1.argumento()-alfa);
     double u1Y=b1.modulo()*Math.sin(b1.argumento()-alfa);
     double u2X=b2.modulo()*Math.cos(b2.argumento()-alfa);
     double u2Y=b2.modulo()*Math.sin(b2.argumento()-alfa);
     double v1X=((u1X*(m1-m2))+(2*m2*u2X))/(m1+m2);
     double v2X=((u2X*(m1-m2))+(2*m2*u1X))/(m1+m2);
     double v1Y=u1Y;
     double v2Y=u2Y;
     b1.velX=(float)((Math.cos(alfa)*v1X)+(Math.cos(alfa+(Math.PI/2))*v1Y));
     b1.velY=(float)((Math.sin(alfa)*v1X)+(Math.sin(alfa+(Math.PI/2))*v1Y));
     b2.velX=(float)((Math.cos(alfa)*v2X)+(Math.cos(alfa+(Math.PI/2))*v2Y));
     b2.velY=(float)((Math.sin(alfa)*v2X)+(Math.sin(alfa+(Math.PI/2))*v2Y));
  }
```

Ahora ya tenemos todas las piezas necesarias para activar nuestro sistema de colisiones. Modificamos el método **nuevo_frame** de la clase pantalla así:

```
public void nuevo_frame() {
  b1.mover();
  b1.frenar(frenada);
  b2.mover();
  b2.frenar(frenada);
  if (colision()) Bola.colisionar(b1,b2);
  invalidate();
}
```

Ahora ya debería funcionar. Pero si lo probamos veremos que de vez en cuando pasan cosas raras (sobre todo si impulsamos nuestra bola muy fuerte y por lo tanto las dos bolas quedan muy superpuestas la una a la otra). Esto es porque al colisionar modificamos la velocidad de las dos bolas, pero no su posición y por lo tanto el siguiente **frame** se vuelve a activar el mecanismo de colisión ya que las bolas no se han separado lo suficiente.

Frame 1 Frame 2 Frame 3

Esto lo podemos solucionar utilizando una bandera booleana que activaremos en el momento en que detectamos una colisión y no la desactivaremos hasta que los objetos estén completamente separados. Mientras la bandera está activa ignoraremos todas las colisiones entre ellos.

```
public class Pantalla extends View{
    private boolean chocando;
    private float sensibl,frenada;
    public int anchoMundo,altoMundo,ancho,alto;
    public float escala;
    public Bola b1,b2;
    private GestureDetector gestos;
    public Pantalla(Context context, AttributeSet attrs) {
        super(context, attrs);
        anchoMundo =1000;
        altoMundo =1000;
        b1=new Bola(anchoMundo /2, altoMundo /2,50,Color.YELLOW);
        b2=new Bola(800,150,50,Color.GREEN);
        sensibl=0.04f;
        frenada=0.95f;
        chocando=false;
        gestos=new GestureDetector(context,new detectorGestos());
    }
    ...

    private boolean colision()
    {
        if (Rect.intersects(b1.zona(),b2.zona()) && !chocando)
        {
            chocando=true;
            return true;
        }
        chocando=false;
        return false;
    }
```

¡¡Ahora ya tenemos nuestro juego terminado!

6.3. Desarrollo de juegos Android

Los juegos son un tipo muy concreto de aplicación dirigido al ocio digital que utiliza al máximo las capacidades interactivas del dispositivo Android en el que se está ejecutando.

Un juego es básicamente una aplicación que recibe una serie de entradas por parte del usuario (que se convierte en el jugador) y realizar un proceso lógico en función del argumento del juego. Esta arquitectura del juego utiliza elementos gráficos y sonidos para producir una salida que es el estado actual del juego.

6.3.1. Arquitectura de un juego

La arquitectura de un juego es el conjunto de partes que componen un juego y las interacciones que se producen entre cada una de estas partes.En el centro de la arquitectura de un juego Android se encuentra el bucle principal o Game Loop, que toma los valores de entrada, actualiza el estado interno del juego según la lógica del juego y, finalmente, actualiza los elementos en la pantalla para representar el nuevo estado del juego.

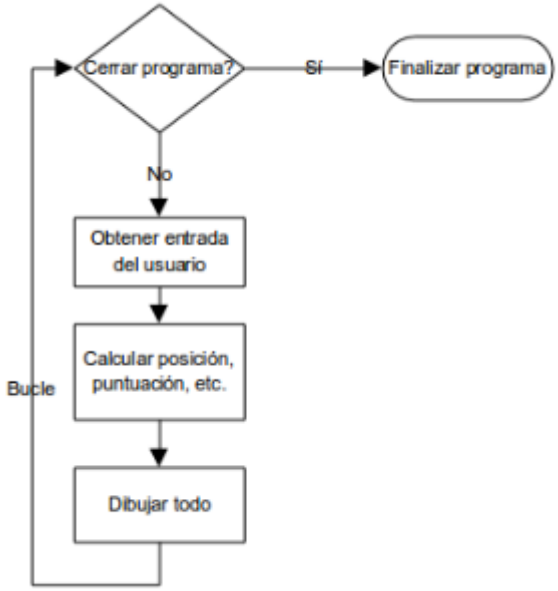


Figura 1: Arquitectura de un juego

El bucle principal se ejecuta de manera infinita: el bucle sólo se rompe bajo ciertas condiciones ocasionadas por el usuario o generadas por el propio desarrollo del juego.

Por ejemplo, en un juego sencillo 2D el juego finalizará cuando se alcance una determinada puntuación, se alcance el final de un nivel, se pierdan las vidas, … Hasta ese momento, se producen tres estados de manera repetitiva:

- Entrada: Se recogen los eventos del juego.
- Actualización: Se recalcula la lógica del juego.
- Representación: Se actualiza la pantalla para mostrar el nuevo estado del juego.

La lógica del juego es la parte de código que responde a las entradas (los eventos generados por el jugador) y actualiza el estado del mismo. También deberá llamar al código encargado de la representación (la visualización en pantalla).

6.3.2. Inicialización del juego

Como paso inicial en la arquitectura del juego se realizar un proceso de inicialización de variables y del estado del juego. Durante esta inicialización se establece la vista o escena inicial del juego, la puntuación y la posición inicial de cada uno de los elementos gráficos del juego en la escena inicial.

Sabemos que Android ofrece subclases View predefinidas para construir la interfaz gráfica: los widgets (Button, TextView, EditText, ListView, …) y los layouts. Estos componentes gráficos los hemos utilizado para desarrollar aplicaciones de propósito general. Sin embargo, para crear un juego se utiliza una vista propia en la que se dibuja el fondo del juego, los sprites y cualquier otro elemento gráfico como líneas, círculos o texto. Utilizando este enfoque, la vista principal del juego se define mediante una clase que hereda de View y sobreescribe sus métodos.

Un ejemplo de una clase propia del tipo View sería el siguiente:

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        MiJuego juego = new MiJuego(this);
        setContentView(juego);
    }
}

class MiJuego extends View {
    public MiJuego(Context context) {
        super(context);
    }
    // sobreescribiremos método onDraw()
    @Override
    protected void onDraw(Canvas canvas) {
    }
}
```

El método onDraw() de una clase de tipo View recibe como parámetro un objeto de tipo Canvas. Como veremos más adelante, la clase Canvas representa una superficie donde podemos dibujar el fondo del juego y los sprites (mapas de bits).

6.3.3. Entradas del usuario

En los dispositivos Android la entrada suele ser por la pantalla táctil o bien mediante el movimiento que realiza el jugador capturado por el osciloscopio del dispositivo. También es posible utilizar otros componentes para capturar entradas como por ejemplo el micrófono, el acelerómetro o el GPS. En cualquier caso, toda entrada produce un evento que es capturado y tratado.

Android implementa un modelo de eventos de entrada donde las acciones del usuario se convierten en eventos que activan llamadas determinadas, que pueden personalizar para responder a las entradas del usuario.

Una forma de gestionar los eventos que produce el jugador en la pantalla táctil del dispositivo es definiendo áreas de control, de tal manera que en función de la posición en donde el jugador toca en la pantalla se realizará una determinada acción. Tras cada evento se ejecutará un determinado código y a continuación se llamará al método onDraw() que volverá a dibujar la vista del juego actualizando la posición y las características de todos sus elementos.

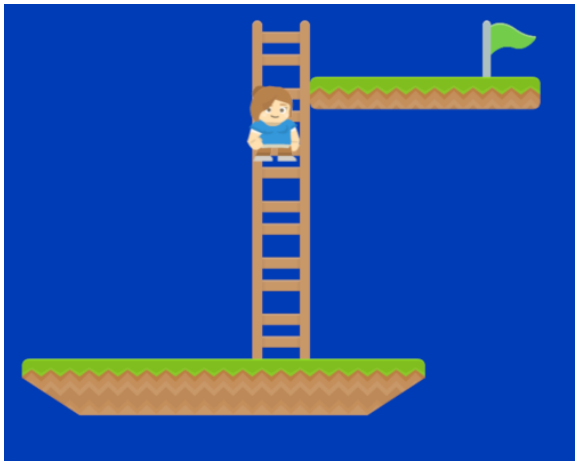


Figura: Entradas de usuario

Un ejemplo de código para capturar pulsaciones del jugador en la pantalla y actuar en consecuencia se puede implementar sobrescribiendo el método onTouchEvent().

Este es el código Java para sobrescribir el método, capturar las pulsaciones del jugador en pantalla y mover el sprite un escalón hacia arriba o hacia abajo:

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    if (event.getAction() == MotionEvent.ACTION_DOWN){
        float posicionY= event.getY();
        if (posicionY> escalon) {
            escalon = escalon + 1;}
        else {
            escalon = escalon - 1;}
    }
    invalidate(); // llamada a onDraw() para redibujar vista
    return true;
}
```

6.3.4. La lógica del juego

La lógica del juego está contenida en el bucle infinito del juego, que se repite muchas veces por segundo, y es la parte de código responsable de actualizar el estado de los elementos del juego, desde la posición de los sprites hasta las puntuaciones. Dependiendo de cada juego se realizarán acciones diferentes:

1. Comprobar los eventos del usuario: si el jugador ha pulsado sobre la pantalla, si ha oscilado el dispositivo, …
2. Actualizar las posiciones de los elementos del juego: el personaje principal, los enemigos u otros integrantes, los ítems que hay que recoger, ….
3. Detectar colisiones entre elementos; por ejemplo, el personaje contra una pared que no puede atravesar o bien recoger un objeto.
4. Dibujar de nuevo la pantalla en función de todo lo anterior (movimientos, saltos, heridas, explosiones, …)
5. Añadir un pequeño retardo o pausa para acomodar la velocidad de la acción.

El bucle principal que contiene la lógica del juego se puede programar en Java como un subproceso (hilo o thread). Este hilo de ejecución se ejecuta de manera concurrente a la aplicación; en Java disponemos de la clase Thread que implementa la interface Runnable.

Por otro lado, se puede utilizar también la clase Timer para planificar la ejecución de este hilo de manera que se ejecute cada cierto tiempo: la representación de los gráficos se mide en cuadros por segundo o FPS (*Frames Per Second*). De esta forma el juego ofrece el aspecto de estar ejecutándose en tiempo real.

El código Java que implementa el bucle principal del juego ejecuta el método invalidate() cada 50 milisegundos se muestran a continuación. Este método fuerza a inicializar la vista o lo que es lo mismo redibujar todos los elementos del juego, consiguiendo dotar de movimiento a los sprites y ofreciendo un aspecto realista del juego.Utilizamos la clase Timer para programar la ejecución del hilo cada 50 milisegundos; el hilo queda definido implementando la interfaz Runnable tal y como se muestra a continuación:

```
@Override
  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(juego);
      // bucle principal del juego
      Timer timer = new Timer();
      timer.schedule(new TimerTask() {
          @Override
          public void run() {
              handler.post(new Runnable() {cada
                  @Override
                  public void run() {
                      juego.invalidate(); // evento que fuerza a
  "redibujar"
                  }
              });
          }
      }, 0, 50); // la tarea se ejecuta cada 50 milisegundos
  }
```

En este punto también toman importancia las diferentes técnicas de detección de colisiones que sirven para controlar el movimiento de los sprites y la manera en que éstos interactúan con el mundo virtual. Vamos a analizar dos maneras de detectar las colisiones de sprites para que la lógica del juego pueda ejecutar las acciones definidas en cada caso, ya sea una colisión o la aproximación a un objeto del juego.

6.3.4.1. Detección geométrica de colisiones

Una técnica sencilla para detectar colisiones en un juego consiste en dibujar un rectángulo alrededor de cada sprite y utilizar el método intersects() para decidir si hay colisión o no.

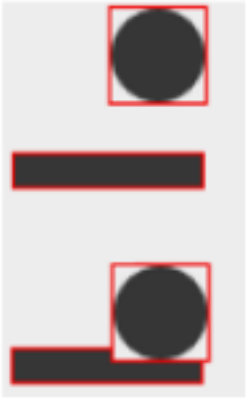


Figura 3: Detección geométrica de colisiones

Siguiendo esta técnica, se detecta la colisión entre una nave y un misil conociendo sus coordenadas y dibujando un rectángulo alrededor de cada objeto.

El código Java para detectar colisiones entre una nave espacial y un misil podría ser el siguiente:

```
void detectaColision(Canvas canvas ) {
    Rect naveRect = new Rect(naveX, naveY, naveX+150, naveY+50);
    Rect misilRect = new Rect(misilX, misilY, misilX + 250,misilY+100);
    if (Rect.intersects(naveRect,misilRect)) {
        // colisión
        sonidoImpacto();
        finJuego();
    }
}
```

En este ejemplo, el recuadro alrededor de cada objeto se calcula mediante el objeto Rect(cuyas coordenadas se indican de la siguiente manera: izquierda, arriba, derecha, abajo); el método intersects() nos dice si hay o no colisión.

6.3.4.2. Sensores de sprites

Otra técnica sencilla para detectar cuando un objeto entra en una determinada zona (por ejemplo, un personaje del juego se acerca a una puerta para que se abra) consiste en utilizar sensores de proximidad. Esta misma idea se puede utilizar también para detectar una colisión de sprites y se puede implementar calculando la distancia existente entre los dos objetos. Calculando la diferencia absoluta entre las coordenadas de los dos objetos podemos averiguar si estos objetos están lo suficientemente cerca y determinar que han colisionado.

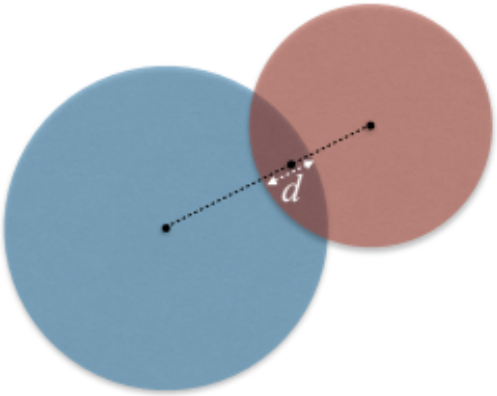


Figura: Sensores de sprites

El código Java que implementaría esta técnica de detección de colisiones podría ser el siguiente:

```
void detectaColision(Canvas canvas) {
    int  distanciaX  = abs(naveX - misilX);
    int distanciaY = abs(naveY - misilY);
    if (distanciaX<nave.getWidth()/2 && (distanciaY<nave.getHeight()/2)
    {
        // colisión
        sonidoImpacto();
        finJuego();
    }
}
```

Esta técnica puede complicarse dependiendo de la forma de los sprites: en el caso de sprites con forma de polígonos irregulares, resultará más complicado calcular la distancia entre dos objetos.

6.3.5. Comportamiento inteligente de los elementos del juego

Utilizar técnicas de inteligencia artificial en los videojuegos produce una ilusión de inteligencia en el comportamiento de los distintos actores del juego, desde el personaje principal hasta los oponentes y objetos del juego.

Las funcionalidades de inteligencia artificial de los juegos se agrupan en tres grupos diferentes. Por un lado, los sensores de percepción que permiten detectar objetos o situaciones. De esta forma el personaje principal del juego es capaz de interactuar con el mundo del videojuego: luchar con oponentes, recoger objetos, abrir puertas, construir estructuras, etcétera.

Los elementos de toma de decisiones marcan qué acción realizar en cada momento del juego; esta característica se aplica no solo al personaje principal del juego, sino también a los oponentes. Por último, el comportamiento inteligente incluye también determinar el movimiento adecuado para acercarse a objetivos superando y evitando obstáculos u oponentes. Este comportamiento puede ser determinista, más fácil de implementar ya que simplemente genera un comportamiento predecible; o bien no determinista, consiguiendo un comportamiento impredecible gracias al empleo de redes neuronales o algoritmos genéticos avanzados.

6.3.6. Otros aspectos del juego

Conseguir que un juego resulte divertido y adictivo es una tarea difícil que requiere cuidar todos los detalles del juego. En este apartado veremos algunos aspectos del juego que cruciales para que el juego tenga un aspecto profesional.

6.3.6.1. El patrón de diseño MVC aplicado al desarrollo de juegos

Los patrones de diseño de software imponen una serie de reglas simples que definen soluciones ampliamente probadas y que ayudan a resolver problemas complejos separando y organizando las distintas partes de un proyecto.

El patrón de diseño MVC (Modelo Vista Controlador) es uno de los más populares. Ampliamente utilizado en el mundo del desarrollo web, también puede aplicarse para simplificar la complejidad que acarrea el desarrollo de un juego.

En el caso del desarrollo de videojuegos, el patrón de diseño MVC descompone el proyecto en un modelo que contiene la representación abstracta del juego (por lo general, el mundo en el que se desarrolla el juego, los personajes y el resto de objetos), varias vistas que visualizan los elementos del modelo (las diferentes pantallas del juego) y un controlador que recibirá las entradas del usuario, las procesará y actualizará la representación del juego.

6.3.6.2. Melodías y sonidos

Los motores de juegos simplifican la tarea de realizar sonidos en los videojuegos proporcionando librerías de sonidos listos para ser utilizados. Por un lado, los sonidos melódicos se utilizan como melodía de fondo en muchos videojuegos. Por otro lado, los efectos sonoros están asociados a cada uno de los elementos del juego. Además de las librerías de sonidos, algunos motores de juegos incluyen también editores de sonido y herramientas de grabación que permiten manipular, grabar y reproducir todo tipo de sonidos para ser utilizados en el videojuego, desde disparos y explosiones hasta conversaciones de los personajes. Los sonidos y en especial la banda sonora definirán la personalidad del juego. Los archivos de sonido (por ejemplo intro.mp3, disparo.mp3 y explosion.mp3) se pueden incluir en la carpeta */res/raw* del proyecto y reproducirse mediante un objeto de la clase MediaPlayer (consulta el Tema 4 “Utilización de librerías multimedia”).

6.3.6.3. El nivel de dificultad

Todos los juegos implementan un sistema de puntuación que suele incrementar progresivamente el nivel de dificultad a medida que el jugador va superando niveles y sumando puntos. La forma más sencilla de incrementar la dificultad del juego se consigue incrementando la velocidad de los enemigos u objetos del juego, o bien incrementado la cantidad de enemigos u objetos que aparecen en el juego.

Para aplicar la anterior estrategia bastaría con disponer de una serie de variables y controlar sus valores:

- Un variable que actúa como contador de puntos.
- Una variable que actúa como contador de niveles.

Los puntos se contabilizan cada vez que el jugador recoge objetos, mata enemigos, … y la lógica del juego decide en qué momento se incrementa la dificultad del juego.

6.3.6.4. Barras de herramientas

Una barra de herramientas permite realizar acciones, buscar, navegar y configurar las opciones del juego. La barra de menús también puede utilizarse para personalizar el juego, ofrecer información y proporcionar una experiencia de usuario diferente.

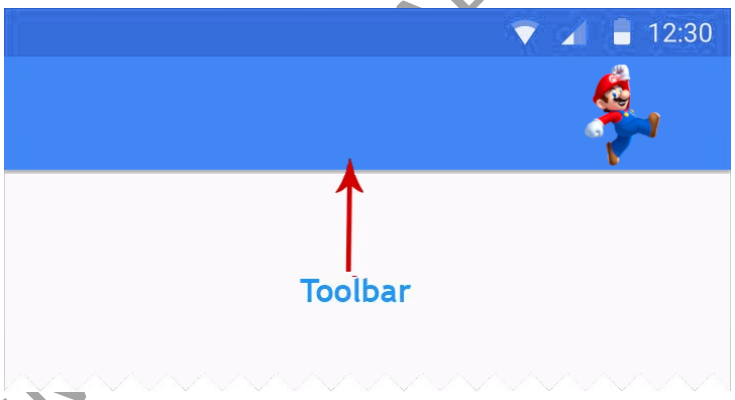


Figura 5: Toolbar de una aplicación

A continuación, encontrarás tres vídeos en los que se explica paso a paso cómo añadir una barra de herramientas a una aplicación y configurarla.

El siguiente vídeo nos muestra cómo crear una barra de herramientas.

Barras de herramientas – Toolbars 01 (Parte 1)

Barras de herramientas – Toolbars 01 (Parte 2)

En el siguiente vídeo sobre Toolbars, se muestra cómo dar forma y trabajar el aspecto de la barra de herramientas.

Barras de herramientas – Toolbars 02

En el último vídeo se añaden opciones a la barra de herramientas que añaden una nueva funcionalidad y permiten al usuario interactuar con la aplicación.

Barras de herramientas – Toolbars 03 (Parte 01)

Barras de herramientas – Toolbars 03 (Parte 02)

6.3.6.5. Librerías de juegos

En el Tema 5 "Motores de juegos" se trataron los motores, aplicaciones que integran diversas funcionalidades y que facilitan el desarrollo de un juego completo o bien el desarrollo de una parte concreta del mismo. Una librería en cambio es un módulo de software orientado a suministrar una funcionalidad concreta.

Existen diversas librerías pensadas para desarrollar juegos Android tanto en 2D como en 3D, entre las que destacan Unity 3D, libGDX, Cocos2D-X y AndEngine. Una de las librerías más populares es sin duda Open GL ya que además de tratarse de un producto de código abierto, sus orígenes se remontan a los años 90, por lo que se trata de un producto consolidado, multiplataforma y con una amplia base de programadores que han utilizado la librería en diversidad de proyectos y con diferentes lenguajes de programación.

Con OpenGL podemos desarrollar juegos para Android tanto en 2D como en 3D. Hay dos clases que nos permiten crear y manipular gráficos con OpenGL en Android: por un dado, la clase GLSurfaceView es un tipo de Vista en donde podemos dibujar objetos; por otro lado, la clase GLSurfaceView.Renderer define los métodos para dibujar los gráficos.

Recursos y enlaces

- Los Doce Principios Básicos de la animación

- Desarrollo de videojuegos

- Fases en el desarrollo de un videojuego

- OpenGameArt.org: recursos gratuitos para juegos

- Librería OpenGL para Android

Conceptos clave

- **Arquitectura de un juego**: Conjunto de partes que componen un juego y las interacciones que se producen entre cada una de estas partes.
- **Librerías de juegos:** Conjunto de código que facilita la programación de una funcionalidad específica o una parte de un videojuego.
- **Motor de juegos:** Conjunto de librerías de programación que permiten el diseño, la creación y la representación de un videojuego.
- **MVC**: Patrón de diseño que utiliza tres componentes (el modelo, la vista y el controlador) para separar la lógica de aplicación de su presentación.
- **OpenGL:** Librería de código abierto que permite desarrollar juegos 2D y 3D en diversas plataformas y utilizando distintos lenguajes de programación.

Test de autoevaluación

1. ¿Cuál es el nombre con el que se denomina el solapamiento de un sprite del videojuego con otro?
   a) Colisión.
   b) Intersección.
   c) Interpolación.
   d) Ninguna de las respuestas anteriores es correcta.

2. ¿Qué nombre reciben las técnicas inteligencia artificial que dotan de comportamiento impredecible a los personajes y objetos de un juego?
   a) No deterministas.
   b) Deterministas.
   c) Bots.
   d) Ninguna de las respuestas anteriores es correcta.

3. ¿Qué funcionalidades proporciona un motor de juegos?
   a) Generación de escenarios.
   b) Librerías de sonidos.
   c) Renderización de animaciones.
   d) Todas las respuestas anteriores son correctas.

4. ¿Qué nombre recibe un elemento gráfico de un videojuego con información como posición geométrica y velocidad?
   a) Objeto.
   b) Sprite.
   c) Mundo.
   d) Escenario.

Solucionarios

Test de autoevaluación

1. ¿Cuál es el nombre con el que se denomina el solapamiento de un sprite del videojuego con otro?
   a) **Colisión**.
   b) Intersección.
   c) Interpolación.
   d) Ninguna de las respuestas anteriores es correcta.

2. ¿Qué nombre reciben las técnicas inteligencia artificial que dotan de comportamiento impredecible a los personajes y objetos de un juego?
   a) **No deterministas.**
   b) Deterministas.
   c) Bots.
   d) Ninguna de las respuestas anteriores es correcta.

3. ¿Qué funcionalidades proporciona un motor de juegos?
   a) Generación de escenarios.
   b) Librerías de sonidos.
   c) Renderización de animaciones.
   d) **Todas las respuestas anteriores son correctas.**

4. ¿Qué nombre recibe un elemento gráfico de un videojuego con información como posición geométrica y velocidad?
   a) Objeto.
   b) **Sprite.**
   c) Mundo.
   d) Escenario.