
	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD COMPETENCIA					

## Tema 5

# Desarrollo de juegos en Android

## Índice

1.	Gestión de gráficos y lienzos a bajo nivel.....	1
2.	La clase View: texto, figuras y Bitmaps. ....	1
2.1.	Pantalla completa.....	8
2.2.	Pantalla siempre encendida.....	8
3.	La clase SurfaceView .....	9
3.1.	Eventos multitouch. ....	12
3.2.	Gestos .....	14
3.3.	Movimiento de un gráfico. Física básica.....	16
3.4.	Detección de colisiones .....	18
3.5.	Realización de scroll. ....	21
3.6.	Efecto Parallax.....	23
3.7.	Sprites animados.....	23
3.8.	Control temporal.....	24
3.9.	Audio y música .....	26
3.9.1.	Efectos sonoros .....	26
3.9.2.	Música.....	28
4.	Otros elementos que se deben conocer .....	30
5.	Fuentes.....	30
6.	Apéndices .....	31
6.1.	Cambiar fuente de texto.....	31
6.2.	Engines gráficos.....	31
6.3.	Programas de creación multimedia .....	31
6.4.	Comprobar la versión API de Android .....	31
6.5.	Apéndice IV: Animaciones y transiciones. ....	31
6.6.	Ejemplos de código.....	32
6.6.1.	Imagen con bodes redondeados .....	32
6.6.2.	Simétrico de una imagen .....	32
6.6.3.	Convertir una imagen a escala de grises.....	33
6.6.4.	Cambiar el brillo y el contraste de una imagen.....	33
6.6.5.	Añadir un borde a una imagen .....	34

	RAMA:	Informática	CICLO:	Desenvolvimento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

## 1. Gestión de gráficos y lienzo a bajo nivel

En este apartado veremos cómo gestionar nuestro propio lienzo de dibujo sin depender de los *layouts* predefinidos.

## 2. La clase `View`<sup>1</sup>: texto, figuras y Bitmaps.

Esta clase es la base para el manejo de gráficos en *Android*. Dispone de la posibilidad de responder a eventos y permite sobrescribir la función `onDraw()` en la cual se colocarán los elementos que se desean dibujar.

Esta clase es útil para juegos en los que la carga gráfica y la interacción con el usuario es baja. Por ejemplo puede ser usada para un juego de mesa, un puzzle sencillo, o para diseñar un menú en una pantalla de inicio (esto último, por supuesto, también se puede hacer de forma clásica con componentes ya conocidos).

En el momento que se desea hacer un juego más complejo se debe usar la clase `SurfaceView`<sup>2</sup> que hereda de `View` y de la que hablaremos más adelante.

Para comenzar veremos un ejemplo sencillo de uso de una clase `View` puesto que la mayoría de los elementos usados en este punto se podrán utilizar también con `SurfaceView`.

Crearemos el proyecto `EjemploView` en el cual vamos a dibujar una serie de elementos si utilizar los componentes predefinidos de *Android*.

Para comenzar tendremos que crear un lienzo sobre el que dibujar, para ello crearemos una clase *java* (no una *Activity*) que herede de `View` (usaremos esta clase para dibujar en vez de colocar componentes en un *layout*) lo que nos permite sobrescribir, entre otros, los siguientes métodos:


- Un constructor para inicializar los elementos.
- `onDraw`: Es el método que se invoca cuando se quiere escribir en el lienzo de dibujo. En el colocaremos la lógica de dibujo que queramos. Es en este método donde dibujaremos los elementos.
- `onTouchEvent`: Se invoca cuando se detecta una pulsación sobre el lienzo y el cual hay que gestionar los distintos eventos asociados con las pulsaciones (de estos eventos hablaremos más adelante en profundidad).
- `onSizeChanged`: Se invoca cuando el tamaño del lienzo de dibujo varía y en la cual podemos obtener su tamaño.

Por lo que según lo anterior la estructura básica de la clase quedaría de la siguiente manera (no es necesario implementar todos los métodos):

```
public class PantallaInicioView extends View {
    public PantallaInicioView(Context context) {
        super(context);
    }
    @Override
    protected void onDraw(Canvas canvas) { // Lienzo sobre el que dibujar
        super.onDraw(canvas);
    }
    @Override
    public boolean onTouchEvent(MotionEvent event) { // Representa un evento asociado a un movimiento
        return super.onTouchEvent(event);
    }
    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh) { // Nuevo (x,y) y viejo (oldw,oldh) tamaño del lienzo
        super.onSizeChanged(w, h, oldw, oldh);
    }
}
```

<sup>1</sup> <https://developer.android.com/reference/android/view/View.html>

<sup>2</sup> <https://developer.android.com/reference/android/view/SurfaceView.html>

	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

El `onCreate` de `MainActivity` se establece la vista del `activity` a la clase creada en vez de un `layout`:

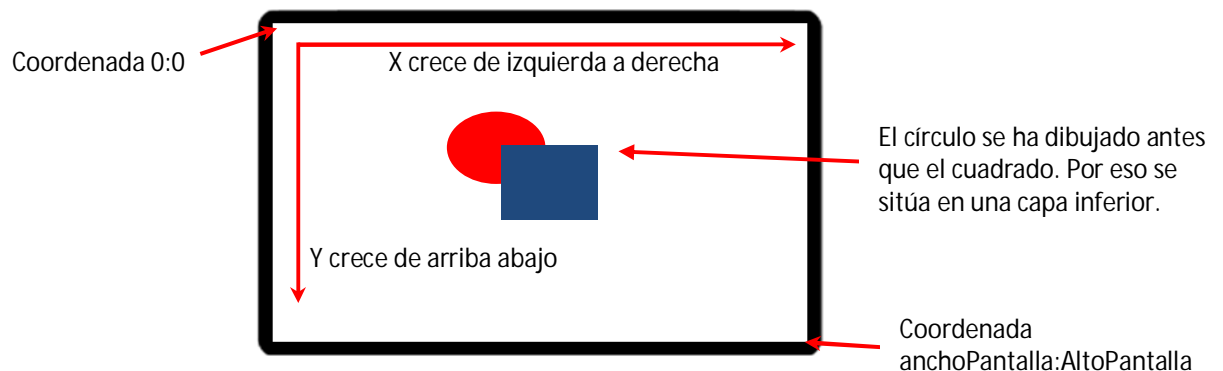
```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    PantallalnicioView pantallalnicio=new PantallalnicioView(this);
    setContentView(pantallalnicio); // establecemos la vista del activity a la clase creada
}
```

Cuando se dibuja texto, figuras o se representan imágenes, *Bitmaps*, ... se tiene, si se desea modificar sus propiedades, utilizar un objeto de tipo *Paint*<sup>3</sup>.

*Paint* permite establecer, entre otras propiedades, el color, tamaño, grosor, transparencia, ... para luego aplicárselo a los distintos objetos a dibujar.


En *Android* la coordenada 0:0 de la pantalla está situada en la esquina superior izquierda. Las X crecen hacia la derecha y las Y hacia abajo. Además un objeto dibujado se situará sobre todos los objetos dibujados previamente.



En el ejemplo siguiente se muestra como dibujar elementos en el lienzo.

```
public class PantallalnicioView extends View {
    int anchoPantalla, altoPantalla, cont=1;
    String texto;
    Paint paint;
    TextPaint tpaint;
    Rect cuadrado, cuadradoBorde, cuadradoConTexto;
    RectF cuadrado2;
    StaticLayout textLayout;
    public PantallalnicioView(Context context) {
        super(context);
        paint=new Paint(); // Se crea un objeto de tipo paint para establecer propiedades a objetos
        paint.setAlpha(240); // Se establece un valor de transparencia. Valores entre 0..255
        paint.setTextSize(110); // Tamaño del texto en pixels (no se le especifica una unidad)
        paint.setAntiAlias(true); // Habilitamos el antialiasing para evitar los dientes de sierra en el texto
        tpaint=new TextPaint();
        tpaint.setTextSize(80); // tamaño del texto en pixels
        tpaint.setTextAlign(Paint.Align.CENTER); // Alineación del texto
        tpaint.setColor(Color.WHITE); // Color del texto
        tpaint.setShadowLayer(4f, 4f, 4f, Color.RED); // Sombra del texto
        cuadrado=new Rect(20, 210, 350, 410); // Rectángulo definido por cuatro coordenadas int: left, top, right, bottom
        cuadrado2=new RectF(320, 240, 640, 440); // Rectángulo definido por cuatro coordenadas float: left, top, right, bottom
        cuadradoBorde=new Rect(620, 210, 920, 410);
        cuadradoConTexto=new Rect();
    }
}
```

<sup>3</sup> <https://developer.android.com/reference/android/graphics/Paint.html>

	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

```

@Override
protected void onDraw(Canvas canvas) { // lienzo sobre el que se va a dibujar

    canvas.drawColor(Color.BLUE); // Establecemos un color de fondo. En este caso azul

    paint.setColor(Color.WHITE); // Establecemos el color de Paint a blanco

    // Dibujamos un texto en la posición x: 20 e y: 100. Usando el estilo definido en el objeto paint
    canvas.drawText(cont + " Prueba de texto", 20, 100, paint);

    paint.setColor(Color.GREEN); // Establecemos el color de Paint a verde
    canvas.drawRect(cuadrado, paint); // Dibujamos un cuadrado con la propiedades definidas en el objeto paint

    paint.setAlpha(170); // Establecemos el valor de la transparencia
    paint.setColor(Color.RED);
    canvas.drawRoundRect(cuadrado2, 40, 30, paint); // Dibujamos un cuadrado semitransparente con bordes redondos

    paint.setAlpha(240);
    paint.setColor(Color.GREEN);
    paint.setStyle(Paint.Style.STROKE); // Indica que solo se dibujaran los bordes
    paint.setStrokeWidth(10); // Ancho de los bordes
    canvas.drawRect(cuadradoBorde, paint); // Dibujamos los bordes de un cuadrado

    paint.setStyle(Paint.Style.FILL); // Indica que se dibujara el relleno de un objeto
    paint.setColor(Color.CYAN);
    canvas.drawCircle(anchoPantalla/2, altoPantalla/2, 200, paint); // Dibujamos un círculo en el centro de la pantalla
    paint.setColor(Color.WHITE); // con radio 200px
    canvas.drawCircle(anchoPantalla/2, altoPantalla/2, 100, paint);

    // Creamos un texto multilinea que se adapta a un ancho definido
    textLayout = new StaticLayout(texto, tpaint, anchoPantalla/2, Layout.Alignment.ALIGN_CENTER, 1.0f, 0.0f, false);
    int textHeight = textLayout.getHeight(); // alto de texto dibujado
    canvas.save(); // guardamos el lienzo
    canvas.translate(anchoPantalla/2, 500); // Movemos el lienzo
    textLayout.draw(canvas); // Dibujamos el texto
    canvas.restore(); // Recuperamos el lienzo a su posición original
}

@Override
public boolean onTouchEvent(MotionEvent event) { // Representa un evento asociado a un movimiento
    cont++; // En este método solamente incrementamos un contador de la veces que se invoca el método
    int accion = event.getAction(); // Acción realizada sobre la pantalla
    float x = event.getX(), y = event.getY(); // Obtenemos las coordenadas x e y de la pulsación
    switch (accion) { // En MotionEvent tenemos todas las posibilidades de acciones
        case MotionEvent.ACTION_DOWN: // Levantamos un dedo que pulsa en la pantalla
            break;
        case MotionEvent.ACTION_UP: // Pulsamos la pantalla
            break;
    }
    invalidate(); // Forzamos el redibujado del lienzo
    return true;
}

@Override
protected void onSizeChanged(int w, int h, int oldw, int oldh) { // nuevo (x,y) y viejo (oldw,oldh) tamaño del lienzo
    super.onSizeChanged(w, h, oldw, oldh);
    this.anchoPantalla=w;
    this.altoPantalla=h;
    this.texto="El ancho de la pantalla es "+w+" y el alto "+h;
}
}

```

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

Si ejecutamos la aplicación sobre distintos dispositivos con distintas tamaños de pantalla y densidades se puede observar los siguientes aspectos (ya se verá como realizar gráficos que no dependan del dispositivo):



#### Tarea

Visualiza un círculo en la posición en la que se toca la pantalla. Este círculo deberá desaparecer haciendo un fade out en un tiempo establecido (por ejemplo dos segundos).

Como se puede ver en las imágenes, partiendo del mismo código se obtienen resultados totalmente diferentes. Esto es debido es que todos los tamaños se dan en *pixels* y esto provoca que sean dependientes del dispositivo ya que en un móvil con una densidad *xxhdpi* se verá mucho más pequeño que en uno con *mdpi*. Lo ideal sería utilizar unidades que no dependan de la resolución, como son *dp* y *sp*.

- *dp* (*density independent pixels*): También llamados *dip*. Un *dp* es un *pixel* virtual equivalente a un *pixel* real en un dispositivo *mdpi* (160dpi). Un dispositivo con mayor densidad estará formado por más *pixels*.
- *sp* (*Scale independent pixels*): medida similar a *dp*, es independiente de la densidad pero está ponderada por las preferencias de tamaño de fuente de texto. Si tiene un tamaño normal, el *sp* es lo mismo que el *dp*. Se usa solo en fuentes.


Para más información se pueden consultar los siguientes enlaces:

- <https://developer.android.com/guide/topics/resources/more-resources.html#Dimension>
- <https://developer.android.com/training/multiscreen/screendensities>
- [https://developer.android.com/guide/practices/screens\\_support.html#density-independence](https://developer.android.com/guide/practices/screens_support.html#density-independence)

Para solucionar el problema anterior tenemos que:

- Usando proporciones de la pantalla. Por ejemplo que un rectángulo tenga un tercio del ancho de la pantalla y un quinto del alto.
- Especificar los tamaños de los elementos en *dps* y no es *pixels*. Podemos, entre otras, realizarlo de las siguientes formas:
  - o Usando el siguiente método para realizar la conversión:

```
int getPixels(float dp) {
    DisplayMetrics metrics = new DisplayMetrics();
    ((WindowManager) getContext().getSystemService(Context.WINDOW_SERVICE)).getDefaultDisplay().getMetrics(metrics);
    return (int)(dp*metrics.density);
}
```

	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

Con lo que el código para dibujar el rectángulo verde quedaría:

```
cuadrado=new Rect(getPixels(6.66f),getPixels(70),getPixels(115),getPixels(136.6f));
```

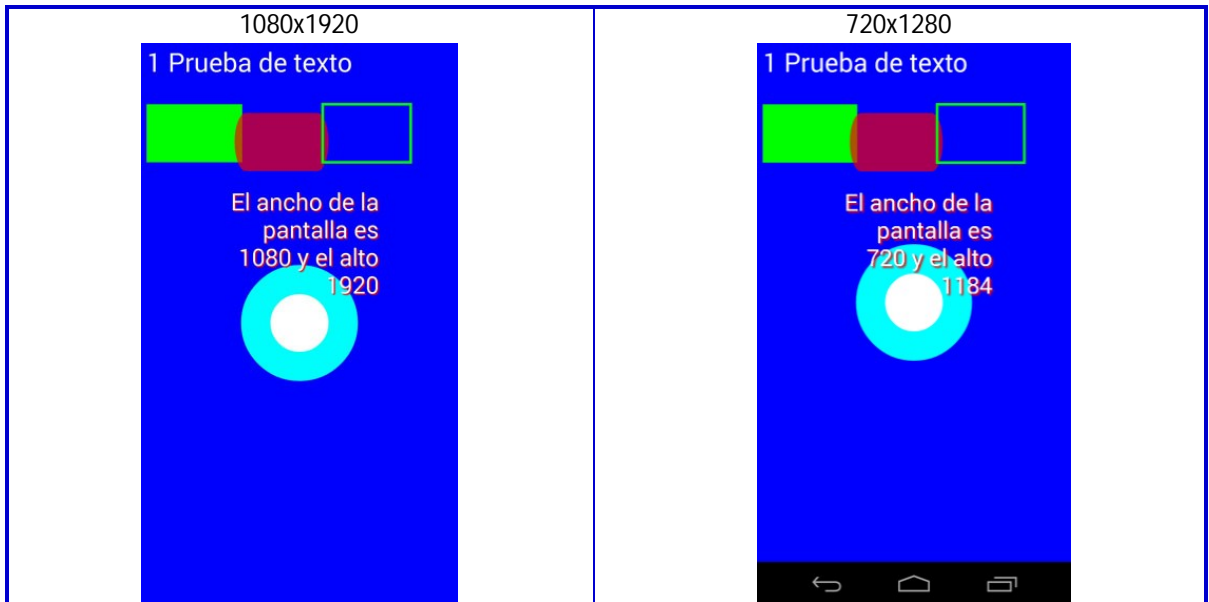
- o Usando el fichero de valores *dimens.xml*. Creo un tamaño para una fuente:

```
<dimen name="font_size">30dp</dimen>
```

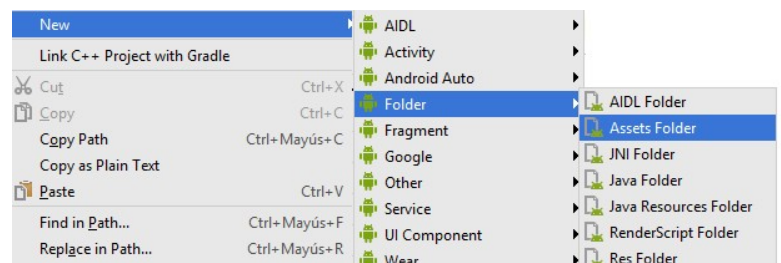
Para acceder a una dimensión se puede utilizar el siguiente código:

```
paint.setTextSize(getResources().getDimensionPixelSize(R.dimen.font_size));
```

Cambiando todos los valores del código anterior podemos obtener los siguientes resultados:



Hasta ahora no hemos dibujado ninguna imagen. Estas se representan en *Android* como *bitmaps* y se pueden colocar tanto en la carpeta *res/drawable* como en la carpeta *Assets*. Esta carpeta se debe referenciar por la ruta del archivo (no es un recurso por lo que no aparece en el archivo *R*) pero en cambio se pueden crear directorios para una mejor organización.



El siguiente es un ejemplo del uso de Imágenes en la aplicación:

Bitmap *imagen*;

- Como recurso:

```
imagen = BitmapFactory.decodeResource(getResources(), R.drawable.imagen);
```

- En el directory *assets*:


```
public Bitmap getBitmapFromAssets(String fichero) {
    try {
        InputStream is=context.getAssets().open(fichero);
        return BitmapFactory.decodeStream(is);
    } catch (IOException e) {
        return null;
    }
}

imagen = getBitmapFromAssets("imagen.png")
```

Para dibujar la imagen se usa el método *drawBitmap*:

```
canvas.drawBitmap(imagen, 0, 0, null);
```



	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

Donde el último parámetro es un objeto de tipo *Paint*, el cual nos permite establecer, por ejemplo, la transparencia de la imagen.

Vamos a ver un ejemplo de su utilización:

- Colocamos las imágenes de *run\_00.png* a *run\_09.png* en el directorio *assets* y los botones avanza y retrocede en la carpeta *drawables*.
- Realizamos las siguientes modificaciones en el código:

- o Creamos los siguientes atributos de clase:

```
Bitmap frame, btnAvz, btnRetrocede;
int posX=0;
boolean avanza=true;
```

- o Inicializamos la imagen en el constructor:

```
this.frame=getBitmapFromAssets("run_00.png");
this.btnAvz=BitmapFactory.decodeResource(getResources(), R.drawable.avanza);
this.btnRetrocede=BitmapFactory.decodeResource(getResources(), R.drawable.retrocede);
```

- o Dibujamos la imagen actual (*posX* se modifica en *onTouch*) en el método *OnDraw*.

```
canvas.drawBitmap(frame, posX, 0, null);
```

- o Dibujamos el botón correspondiente centrado horizontalmente en la parte inferior de la pantalla.


```
if (avanza) canvas.drawBitmap(btnAvz, anchoPantalla/2-btnAvz.getWidth()/2, altoPantalla-btnAvz.getHeight(), null);
else canvas.drawBitmap(btnRetrocede, anchoPantalla/2-btnRetrocede.getWidth()/2, altoPantalla-btnRetrocede.getHeight(), null);
```

- o Modificamos *onTouchEvent* con el siguiente código.

```
public boolean onTouchEvent(MotionEvent event) { // Representa un evento asociado a un movimiento
    int accion = event.getAction();
    float x = event.getX(), y = event.getY();
    switch (accion) { // En MotionEvent tenemos todas las posibilidades
        case MotionEvent.ACTION_DOWN:
            // detectamos la pulsación de un botón si el punto pulsado coincide con las coordenadas del botón
            if (x>(anchoPantalla/2-btnRetrocede.getWidth()/2) && x<(anchoPantalla/2+btnRetrocede.getWidth()/2)
                && y>altoPantalla-btnRetrocede.getHeight()) {
                avanza=!avanza;
            } else { // si no hemos pulsado un botón, cambio de frame
                cont++;
                this.frame=getBitmapFromAssets("run_0"+(cont%10)+".png");
                if (avanza) posX+=getPixels(3f);
                else posX-=getPixels(3f);
            }
            break;
        case MotionEvent.ACTION_UP:
            break;
    }
    invalidate(); // Forzamos el redibujado del lienzo
    return true;
}
```

Android recomienda crear versiones de cada imagen con distintos tamaños para cargar la que más se adapte a la densidad de nuestro dispositivo. Cada versión se tiene que colocar en el directorio asociado a su densidad y Android, dependiendo de la densidad del dispositivo, cargará una imagen u otra..

- *res/drawable-mdpi/* → Directorio para densidad media
- *res/drawable-hdpi/* → Directorio para densidad alta
- *res/drawable-xhdpi/* → Directorio para densidad extra alta

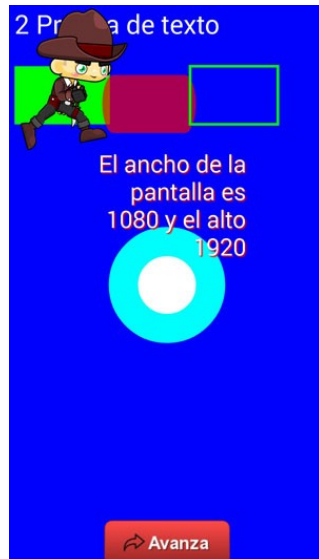
	RAMA:	Informática	CICLO:	Desenvolvimento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

- res/drawable-xxhdpi/ → Directorio para densidad extra extra alta
- res/drawable-xxxhdpi/ → Directorio para densidad extra extra extra alta

Si no se hace así, será el sistema el que se encargue de reescalar la imagen de forma se puede perder calidad.

Vamos a fijarnos el resultado de las modificaciones anteriores:

Android 5.1 Lollipop 1080x1920



Android 4.4 KitKat 720x1280



De nuevo podemos observar que las imágenes del personaje se ven con tamaños diferentes. Esto es porque Android no realiza un reescalado automático de la carpeta *assets*, por lo que la imagen tiene el mismo tamaño en *pixels*. Incluso el reescalado depende de la carpeta en donde está situada la imagen (por eso de la recomendación de crear imágenes para todas las densidades) pudiendo generar tamaños no deseados.

#### Tarea

- Insertar la imagen *caida.png* en el directorio *res/drawables* y visualiza la aplicación.
- Borra la imagen e insértala ahora en el directorio *res/drawables-xhdpi*. Visualiza la aplicación.
- Borra la imagen e insértala ahora en el directorio *res/drawables-xxxhdpi*. Visualiza la aplicación.

Puede ser una buena idea, a la hora de visualizar una imagen, reescalarla al tamaño deseado para que no surjan los problemas anteriores.

Se puede reescalar una imagen de diversas maneras, tres posibles alternativas pueden ser:

- Reescalar a un nuevo ancho y alto. Este método puede provocar una distorsión de la imagen original.

```
public Bitmap escala(int res, int nuevoAncho, int nuevoAlto){
    Bitmap bitmapAux=BitmapFactory.decodeResource(context.getResources(), res);
    return bitmapAux.createScaledBitmap(bitmapAux,nuevoAncho, nuevoAlto, true);
}
```


- Reescalar de forma proporcional a un nuevo ancho.

```
public Bitmap escalaAnchura(int res, int nuevoAncho) {
    Bitmap bitmapAux=BitmapFactory.decodeResource(context.getResources(), res);
    if (nuevoAncho==bitmapAux.getWidth()) return bitmapAux;
    return bitmapAux.createScaledBitmap(bitmapAux, nuevoAncho, (bitmapAux.getHeight() * nuevoAncho) /
                                         bitmapAux.getWidth(), true);
}
```

- Reescalar de forma proporcional a un nuevo alto.

```
public Bitmap escalaAltura(int res, int nuevoAlto) {
    Bitmap bitmapAux=BitmapFactory.decodeResource(context.getResources(), res);
    return bitmapAux.createScaledBitmap(bitmapAux, (bitmapAux.getWidth() * nuevoAlto) /
                                         bitmapAux.getHeight(), nuevoAlto, true);
}
```



	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

```

        if (nuevoAlto==bitmapAux.getHeight()) return bitmapAux;
        return bitmapAux.createScaledBitmap(bitmapAux, (bitmapAux.getWidth() * nuevoAlto) /
                                            bitmapAux.getHeight(), nuevoAlto, true);
    }

```

El último parámetro del método *createScaledBitmap* no indica si se aplica un filtro al escalado, donde:

- false: dará como resultado una imagen más *pixelada*.
- true: dará como resultado una imagen con los bordes más suaves.

No nos pararemos más con esta clase pues la que realmente nos interesa para un uso más exhaustivo es *SurfaceView* que vemos a continuación.

### Tarea

Modifica el código para que el personaje se mueva de forma continua, a una velocidad establecida, en la dirección establecida por el botón sin necesidad de estar pulsado la pantalla.

## 2.1. Pantalla completa

Por norma general los juegos se ven a pantalla completa, por lo tanto habrá que ocultar todas las barras auxiliares. En *onResume* de *MainActivity* se puede colocar:

```

if (Build.VERSION.SDK_INT < 16) { // versiones anteriores a Jelly Bean
    getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN, WindowManager.LayoutParams.FLAG_FULLSCREEN);
} else { // versiones iguales o superiores a Jelly Bean
    final int flags= View.SYSTEM_UI_FLAG_FULLSCREEN
        | View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN
        | View.SYSTEM_UI_FLAG_LAYOUT_STABLE
        | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION // Oculta la barra de navegación
        | View.SYSTEM_UI_FLAG_HIDE_NAVIGATION // Oculta la barra de navegación
        | View.SYSTEM_UI_FLAG_IMMERSIVE
        | View.SYSTEM_UI_FLAG_IMMERSIVE_STICKY;

    final View decorView = getWindow().getDecorView();
    decorView.setSystemUiVisibility(flags);

    decorView.setOnSystemUiVisibilityChangeListener(visibility -> {
        if ((visibility & View.SYSTEM_UI_FLAG_FULLSCREEN) == 0) {
            decorView.setSystemUiVisibility(flags);
        }
    });
}
getSupportActionBar().hide(); // se oculta la barra de ActionBar
}

```

Se puede consultar información es el siguiente enlace: <https://developer.android.com/training/system-ui/status#java><sup>4</sup>

## 2.2. Pantalla siempre encendida


Lo habitual cuando se juega a un juego es que la pantalla permanezca siempre encendida teniendo en cuenta que esto produce un mayor consumo de batería. Con el siguiente código en *onCreate* se consigue este efecto.

```

pantallaInicio.setKeepScreenOn(true);

```

<sup>4</sup> <https://developer.android.com/training/system-ui/status#java>

	RAMA:	Informática	CICLO:	Desenvolvimento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

### 3. La clase SurfaceView

Esta clase dispone, frente a *View*, de la ventaja de poder usar hilos diferentes para el control de eventos y para el dibujo. Esto lo hace más versátil pero más compleja de usar.

*SurfaceView*<sup>5 6</sup> hereda de *View* con el objetivo de ofrecer una superficie de dibujo en un hilo secundario de forma que la aplicación principal no necesita esperar a que se realice el dibujo.

Cuando creamos una nueva *SurfaceView*, mediante herencia, es necesario implementar también la interface *SurfaceHolder.Callback*<sup>7 8</sup> que gestiona los métodos de creación, destrucción y cambio de la superficie de dibujo.

La estructura básica de esta clase será la siguiente:

```
public class PruebaSurfaceView extends SurfaceView implements SurfaceHolder.Callback{
    public PruebaSurfaceView(Context context) {
        super(context);
    }

    // Se ejecuta inmediatamente después de la creación de la superficie de dibujo
    @Override
    public void surfaceCreated(SurfaceHolder holder) {
    }

    // Se ejecuta inmediatamente después de que la superficie de dibujo tenga cambios o bien de tamaño o bien de forma
    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {
    }

    // Se ejecuta inmediatamente antes de la destrucción de la superficie de dibujo
    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
    }
}
```

El nombre de este interface, *SurfaceHolder*, viene de que nosotros no usaremos directamente la superficie si no que lo haremos a través de un objeto que la gestiona denominado *SurfaceHolder*. Para conseguir dicho objeto y por tanto poder usar la superficie usaremos el método *getHolder()*.

Veamos un ejemplo de uso en el que una nave se mueve sobre un fondo espacial. Crearemos un nuevo proyecto que solo permita una orientación vertical y completaremos con el código que viene a continuación:

El método *onCreate* será similar al caso del *View*:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Aquí se introduciría el código del punto 2.1 para poner la aplicación en pantalla completa


    PruebaSurfaceView pantalla=new PruebaSurfaceView(this);
    pantalla.setKeepScreenOn(true);
    setContentView(pantalla);
}
```

<sup>5</sup> <https://source.android.com/devices/graphics/arch-sh>

<sup>6</sup> <https://developer.android.com/reference/android/view/SurfaceView>

<sup>7</sup> <https://developer.android.com/reference/android/view/SurfaceHolder>

<sup>8</sup> <https://developer.android.com/reference/android/view/SurfaceHolder.Callback.html>

	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

Y para la clase *PruebaSurfaceView*, que hereda de *SurfaceView*, usaremos el siguiente:

```
public class PruebaSurfaceView extends SurfaceView implements SurfaceHolder.Callback {
    private SurfaceHolder surfaceHolder; // Interfaz abstracta para manejar la superficie de dibujado
    private Context context; // Contexto de la aplicación
    private Bitmap bitmapFondo; // Imagen de fondo
    private int anchoPantalla=1; // Ancho de la pantalla, su valor se actualiza en el método surfaceChanged
    private int altoPantalla=1; // Alto de la pantalla, su valor se actualiza en el método surfaceChanged
    private Hilo hilo; // Hilo encargado de dibujar y actualizar la física
    private boolean funcionando = false; // Control del hilo

    public PruebaSurfaceView(Context context) { // Constructor
        super(context);

        this.surfaceHolder = getHolder(); // Se obtiene el holder
        this.surfaceHolder.addCallback(this); // y se indica donde van las funciones callback
        this.context = context; // Obtenemos el contexto

        hilo = new Hilo(); // Inicializamos el hilo
        setFocusable(true); // Aseguramos que reciba eventos de toque
        bitmapFondo = BitmapFactory.decodeResource(context.getResources(), R.drawable.fondovivas);
    }


    public void actualizarFisica() { // Actualizamos la física de los elementos en pantalla
    }

    public void dibujar(Canvas c) { // Rutina de dibujo en el lienzo. Se le llamará desde el hilo
        try {
            c.drawBitmap(bitmapFondo, 0, 0, null);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // Callbacks del SurfaceHolder //////////////////////////////////////
    @Override // En cuanto se crea el SurfaceView se lance el hilo
    public void surfaceCreated(SurfaceHolder holder) {
        hilo.setFuncionando(true);
        if (hilo.getState() == Thread.State.NEW) hilo.start();
        if (hilo.getState() == Thread.State.TERMINATED) {
            hilo=new Hilo();
            hilo.start();
        }
    }

    // Si hay algún cambio en la superficie de dibujo (normalmente su tamaño) obtenemos el nuevo tamaño
    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {
        anchoPantalla = width;
        altoPantalla = height;
        hilo.setSurfaceSize(width,height);
    }

    // Al finalizar el surface, se para el hilo
    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        hilo.setFuncionando(false);
        try {
            hilo.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

```

// Clase Hilo en la cual se ejecuta el método de dibujo y de física para que se haga en paralelo con la
// gestión de la interfaz de usuario
class Hilo extends Thread {
    public Hilo() {
    }

    @Override
    public void run() {
        while (funcionando) {
            Canvas c = null; //Siempre es necesario repintar todo el lienzo
            try {
                if (!surfaceHolder.getSurface().isValid()) continue; // si la superficie no está preparada repetimos

                //c = surfaceHolder.lockCanvas(); // Obtenemos el lienzo con aceleración software
                c = surfaceHolder.lockHardwareCanvas(); // Obtenemos el lienzo con Aceleración Hw. Desde la API 26
                synchronized (surfaceHolder) { // La sincronización es necesaria por ser recurso común
                    actualizarFisica(); // Movimiento de los elementos
                    dibujar(c); // Dibujamos los elementos
                }
            } finally { // Haya o no excepción, hay que liberar el lienzo
                if (c != null) {
                    surfaceHolder.unlockCanvasAndPost(c);
                }
            }
        }
    }

    // Activa o desactiva el funcionamiento del hilo
    void setFuncionando(boolean flag) {
        funcionando = flag;
    }

    // Función llamada si cambia el tamaño del view
    public void setSurfaceSize(int width, int height) {
        synchronized (surfaceHolder) { // Se recomienda realizarlo de forma atómica
            if (bitmapFondo != null) { // Cambiamos el tamaño de la imagen de fondo al tamaño de la pantalla
                bitmapFondo = Bitmap.createScaledBitmap(bitmapFondo, width, height, true);
            }
        }
    }
}

```

Una última cosa que se debe tener en cuenta es la posibilidad de salir de la aplicación temporalmente. En dicho caso al ejecutar de nuevo el *Thread* provocara una excepción. Para ello por un lado puedes comprobar el estado del *Thread* de la siguiente forma:

```

if (hilo.getState() == Thread.State.NEW) hilo.start();
if (hilo.getState() == Thread.State.TERMINATED) {
    hilo=new Hilo();
    hilo.start();
}


```

y además manejar el ciclo de vida del *SurfaceView* y quizá del *activity* para que todo quede en su lugar correspondiente.

El esquema anterior lo único que hace realmente es establecer la estructura del *SurfaceView* y presentar un fondo de pantalla. A partir de aquí lo que queda es:

- Gestionar distintos eventos.
- Crear la física del juego.
- Dibujar los distintos gráficos.

Pero esta estructura básica seguirá siendo siempre similar.

	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

### 3.1. Eventos multitouch.

Al igual que en el apartado correspondiente a *View SurfaceView* también es *listener* del eventos de toque de pantalla, de forma que podemos establecer el método *onTouchEvent* en la clase heredada de *SurfaceView* que estamos construyendo.

Revisemos dicho método:

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    int accion = event.getAction(); // Solo gestiona la pulsación de un dedo.
    int accion = event.getActionMasked(); // Gestiona el toque con múltiples dedos
    float x = event.getX();
    float y = event.getY();
    switch (accion) {
        case MotionEvent.ACTION_DOWN:
            break;
        case MotionEvent.ACTION_UP:
            break;
    }
    return true;
}
```

En este caso en el parámetro *event* viene información sobre el evento, de hecho *MotionEvent* sirve para obtener información de eventos de distintos dispositivos de entrada. Se obtiene qué evento sucede, de qué dispositivo, la posición, el movimiento, la presión, etc.

En el ejemplo anterior solo hemos tomado las posiciones X e Y y la acción a realizar.

A continuación a partir de la acción viene el *switch* para realizar distintas tareas pero ¿Qué acciones tenemos?. Las más habituales son las siguientes:

- ACTION\_DOWN: El primer dedo toca la pantalla.
- ACTION\_POINTER\_DOWN: Con al menos un dedo tocando, otro dedo toca.
- ACTION\_UP: El único dedo, o el último se había varios, que toca la pantalla se levanta.
- ACTION\_POINTER\_UP: Con al menos dos dedos tocando, uno se levanta (al menos uno la sigue tocando).
- ACTION\_MOVE: Alguno de los dedos se mueve.

Como ejemplo simple, veamos cómo hacer para que a tocar en la pantalla anterior, se pase a modo juego estando en la misma *Activity*. Esto también se puede hacer con varias *activities*, teniendo cada parte del juego en una *activity* y haciendo sólo el *SurfaceView* en la *activity* de juego propiamente dicha


Veremos un ejemplo de un caso simple de contar una única *activity*.

Lo primero es definir la booleana *esTitulo* para indicar si estamos en la pantalla de título o en la de juego.

```
private boolean esTitulo=true;
```

Y la gestión del evento es muy simple ya que simplemente hay que cambiar el fondo e indicar que ya no estamos en la pantalla de título cambiando la variable booleana:

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    synchronized (surfaceHolder) {
        int accion = event.getActionMasked(); // Gestiona el toque con múltiples dedos
        switch (accion) {
            case MotionEvent.ACTION_UP:
                if (esTitulo) {
                    bitmapFondo = BitmapFactory.decodeResource(getResources(), R.drawable.tierraluna);
                    bitmapFondo = Bitmap.createScaledBitmap(bitmapFondo, anchoPantalla, altoPantalla, true);
                    esTitulo = false;
                }
        }
    }
}
```

	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

```

        break;
    }
}
return true; //Devuelve true si queremos indicar que hemos gestionado la pulsación, false en caso contrario
}

```

Veamos ahora un ejemplo más complejo. Haremos que debajo de cada dedo que pongamos en la pantalla aparezca una nave y se mueva con dicho dedo de forma independiente.

Para ello lo primero es cargar el *bitmap* de la nave. Lo podemos hacer en el constructor donde ya cargamos el *bitmap* de fondo, quedaría así:

```
bitmapNave= BitmapFactory.decodeResource(context.getResources(), R.drawable.nave1);
```

A continuación en *onTouchEvent* usamos *getActionIndex* para obtener el índice de la acción, normalmente es 0, pero si hay varias acciones simultáneas (por ejemplo tocamos con más de un dedo a un mismo tiempo) el valor puede cambiar. Es necesario *getPointerId* para tener un *ID* único del dedo que está asociado a la acción.

También se usará un *HashMap* para guardar cada una de las posiciones, ya que cada vez que se marca un dedo se genera un *ID* único que se usará como clave de la tabla y la posición como dato (en un objeto *PointF* que guarda coordenadas tipo *float*, *Point* en cambio las guarda como *int*).

Este *HashMap* se definirá fuera del método ya que debe ser persistente entre distintas ejecuciones y se usará en el método de dibujo.

```


private static final int MIN_DXDY = 2;
final private static HashMap<Integer, PointF> posiciones = new HashMap<>();

// Gestionamos los eventos de toque sobre la vista
public boolean onTouchEvent(MotionEvent event) {
    synchronized (surfaceHolder) {
        int pointerIndex = event.getActionIndex(); // Obtenemos el índice de la acción
        int pointerID = event.getPointerId(pointerIndex); // Obtenemos el Id del pointer asociado a la acción
        int accion = event.getActionMasked();
        switch (accion) {
            case MotionEvent.ACTION_DOWN: // Primer dedo toca
            case MotionEvent.ACTION_POINTER_DOWN: // Segundo y siguientes tocan
                if (!esTitulo) { // Si no estamos en la pantalla de título
                    // Creamos una posición y la guardamos en la tabla hash
                    PointF posicion = new PointF(event.getX(pointerIndex), event.getY(pointerIndex));
                    posiciones.put(pointerID, posicion);
                    // Líneas de depuración para comprender mejor el getActionIndex y el getPointerID
                    Log.i("ACTION ", "DOWN -> ActionIndex="+pointerIndex+" "+"PointerID="+pointerID);
                }
                break;

            case MotionEvent.ACTION_UP: // Al levantar el último dedo
            case MotionEvent.ACTION_POINTER_UP: // Al levantar un dedo que no es el último
                if (esTitulo) {
                    bitmapFondo = BitmapFactory.decodeResource(getResources(), R.drawable.tierraluna);
                    bitmapFondo = Bitmap.createScaledBitmap(bitmapFondo, anchoPantalla, altoPantalla, true);
                    esTitulo = false; // Cambiamos de pantalla
                } else {
                    posiciones.remove(pointerID); // Eliminamos el dedo que se ha levantado
                    Log.i("ACTION ", "UP -> ActionIndex="+pointerIndex+" "+"PointerID="+pointerID);
                }
                break;
        }
    }
}

```



	RAMA:	Informática	CICLO:	Desenvolvimento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

```

case MotionEvent.ACTION_MOVE: // Se mueve alguno de los dedos
    if (!esTitulo) {
        // Recorremos el HashMap con los dedos que tocan la pantalla ya que índice de acción solo
        // actúa sobre el primero (sin bucle solo se movería el primero)
        for (int i = 0; i < event.getPointerCount(); i++) {
            int ID = event.getPointerId(i);
            PointF posicion = posiciones.get(ID);
            if (posicion != null) {
                // Se actualiza su posición sólo si el dedo se mueve una distancia mínima
                if (Math.abs(posicion.x - event.getX(i)) > MIN_DXDY ||
                    Math.abs(posicion.y - event.getY(i)) > MIN_DXDY) {
                    posicion.set(event.getX(i), event.getY(i));
                }
            }
        }
        Log.i("ACTION ", "MOVE -> ActionIndex="+pointerIndex+ " "+"PointerID="+pointerID);
    }
    break;
default: Log.i("Otra acción", "Acción no definida: "+accion);
}
return true;
}

```

Y luego dibujamos. Para ello recorremos la colección de posiciones creada en la gestión de eventos y en cada posición dibujamos de forma centrada una nave:

```

public void dibujar(Canvas c) {
    try {
        c.drawBitmap(bitmapFondo, 0, 0, null); // Dibujamos el fondo
        if (!esTitulo) { // Si no estamos en la ventana de título
            for (PointF posicion : posiciones.values()) { // Se dibuja una nave en cada uno de los dedos que tocan la pantalla
                float x = posicion.x - bitmapNave.getWidth() / 2; // Centramos la nave horizontalmente en su punto de toque
                float y = posicion.y - bitmapNave.getHeight() / 2; // Centramos la nave verticalmente en su punto de toque
                c.drawBitmap(bitmapNave, x, y, null);
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

Ten en cuenta que *View*, y por herencia *SurfaceView*, ya son *listeners* para eventos *multitouch*, sin embargo si necesitas usarlo en otro componente que no sean estos, se necesita implementar la interface *OnTouchListener* y el evento *OnTouch*.

## 3.2. Gestos

Para trabajar con gestos se puede realizar diversas formas:

- Una es establecer secuencias de acciones como las que vimos antes. Por ejemplo un doble toque consiste en *ACTION\_DOWN*, *ACTION\_UP* dos veces seguidas en un intervalo corto. O *pinch out* (efecto aumento de zoom, por ejemplo) son dos *ACTION\_DOWN* de distintos *IDs* y un *ACTION\_MOVE* donde se comprueba la separación entre ambos dedos.
- La otra forma es utilizar la clase *GestureDetector* que viene por un lado con el listener *SimpleOnGestureListener* que detecta gestos sencillos como tap, double tap, fling, ... O se puede usar *GestureBuilderClass* para gestos más complejos.


Veamos un ejemplo.

Declaramos un objeto tipo *GestureDetectorCompat* como propiedad:

```

public GestureDetectorCompat detectorDeGestos; // clase encargada de detectar gestos simples

```

	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

En el constructor podemos definir directamente el *listener* para los gestos a los que se desea responder:

```
detectorDeGestos=new GestureDetectorCompat(context, new GestureDetector.SimpleOnGestureListener() {
    // aquí se sobreescribirían los métodos para cada gesto
});
```

Si en lugar de estar en una *View* a medida estuviéramos en una *Activity*, recuerda que el contexto sería la propia *Activity*, es decir, *this*.

O puede ser más cómodo, si los métodos sobreescritos son muy largos, implementarlo en una clase aparte.

```
class DetectorDeGestos extends GestureDetector.SimpleOnGestureListener {
    private static final String DEBUG_TAG = "GESTO";

    @Override
    public boolean onDown(MotionEvent event) {
        Log.i(DEBUG_TAG, "OnDown: " + event.toString());
        return true;
    }

    @Override
    public void onLongPress(MotionEvent event) {
        Log.i(DEBUG_TAG, "Long press: " + event.toString());
    }

    @Override
    public boolean onFling(MotionEvent event1, MotionEvent event2, float velocityX, float velocityY) {
        if (velocityX < -10f) { //Si desplazo cierta velocidad a la derecha
            Log.i(DEBUG_TAG, "Fling <-");
        } else {
            if (velocityX > 10f) {
                Log.i(DEBUG_TAG, "Fling ->");
            }
        }
        return true;
    }

    @Override
    public boolean onDoubleTap(MotionEvent event){
        Log.i(DEBUG_TAG, "Double tap");
        return true;
    }

    @Override
    public boolean onSingleTapUp(MotionEvent event){
        Log.i(DEBUG_TAG, "Single tap");
        return true;
    }

    @Override
    public boolean onScroll(MotionEvent e1, MotionEvent e2, float distanceX, float distanceY) {
        Log.i(DEBUG_TAG, "Scroll");
        return true;
    }
}
```


En el constructor instanciamos es detector de gestos:

```
detectorDeGestos=new GestureDetectorCompat(context, new DetectorDeGestos()); // clase para detectar gestos
```

Finalmente en el *onTouchEvent* del *SurfaceView* invocamos al *onTouchEvent* del detector de gestos:

```
public boolean onTouchEvent(MotionEvent event) {
    synchronized (surfaceHolder) {
        detectorDeGestos.onTouchEvent(event);
        .....
    }
}
```

El resto lo podemos dejar igual.

	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

### 3.3. Movimiento de un gráfico. Física básica.

Con lo que se ha visto hasta el momento, ya se debería tener una idea de cómo mover un *sprite* (gráfico 2D) en la pantalla. Veremos de todas formas con un ejemplo como establecer ese movimiento. Añadiremos un enemigo que va de arriba a abajo en una primera versión y luego en zigzag para añadir algo de física básica al movimiento.

Primero creamos una clase padre que va a representar al enemigo. Luego los enemigos concretos (marciano, asteroide, ...) si necesitan cambiar el comportamiento por defecto podrán heredar de ella.

La clase enemigo dispondrá de una imagen, unas coordenadas de pantalla y un método que se encarga de la física del objeto, es decir de su movimiento.

```
public class Enemigo {
    public PointF posicion;
    public Bitmap imagen;
    private Random g;

    public Enemigo(Bitmap imagen, float x, float y) {
        this.imagen = imagen;
        this.posicion = new PointF(x, y);
        g = new Random();
    }

    //Establece el movimiento de un enemigo en una pantalla definida por alto y ancho y cierta velocidad
    public void moverEnemigo(int alto, int ancho, int velocidad) {
        posicion.y += velocidad;
        if (posicion.y > alto) {
            posicion.y = 0;
            posicion.x = g.nextFloat() * (ancho - imagen.getWidth());
        }
    }
}
```

Definimos como atributos en el *SurfaceView* un objeto enemigo y un *bitmap* que será la representación gráfica de un tipo de enemigos.


```
Enemigo marciano;
Bitmap bitmapMarciano;
```

E inicializamos el enemigo en el método *setSurfaceSize* de hilo:

```
public void setSurfaceSize(int width, int height) { // Función llamada si cambia el tamaño del view
    synchronized (surfaceHolder) { // Se recomienda realizarlo de forma atómica
        if (bitmapFondo != null) { // Cambiamos el tamaño de la imagen de fondo al tamaño de la pantalla
            bitmapFondo = Bitmap.createScaledBitmap(bitmapFondo, width, height, true);
        }
        bitmapMarciano = BitmapFactory.decodeResource(context.getResources(), R.drawable.marciano);
        marciano = new Enemigo(bitmapMarciano, new Random().nextFloat()*(anchoPantalla-bitmapMarciano.getWidth()), 0);
    }
}
```

Luego establecemos el método *actualizarFisica()* que es donde gestionaremos todo el movimiento automático del juego. Por supuesto por ahora solo llamamos al movimiento del marciano, pues es nuestro único enemigo. Si deseas cambiar la velocidad cambia el tercer parámetro:

```
public void actualizarFisica(){
    marciano.moverEnemigo(altoPantalla, anchoPantalla, 10);
}
```

	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

Este método será ejecutado en el hilo cada vez antes de dibujar. De esta forma el método run() queda:

```
public void run() {
    while (funcionando) {
        Canvas c = null; //Necesario repintar _todo el lienzo
        try {
            // Obtenemos el lienzo. La sincronización es necesaria por ser recurso común
            c = surfaceHolder.lockCanvas();
            synchronized (surfaceHolder) {
                if (!esTitulo) actualizarFisica(); // Movimiento de los elementos
                dibujar(c);
            }
        } finally { // Haya o no excepción, hay que liberar el lienzo
            if (c != null) {
                surfaceHolder.unlockCanvasAndPost(c);
            }
        }
    }
}
```

Y finalmente dibujamos el Sprite en el lugar adecuado.

```
public void dibujar(Canvas c) {
    try {
        c.drawBitmap(bitmapFondo, 0, 0, null); // Dibujamos el fondo

        if (!esTitulo) {
            c.drawBitmap(marciano.imagen, marciano.posicion.x, marciano.posicion.y, null);
        }
    } catch (Exception e) {
    }
}
```

Si se desea cambiar la física del marciano simplemente se trata de complicar la función de movimiento. Por ejemplo, si añadimos la siguiente propiedad y la siguiente función a la clase Enemigo:


```
private int direccion=1;
public void zigzag(int alto, int ancho, int velocidad){
    posicion.y += velocidad;
    posicion.x += velocidad*direccion;

    if (posicion.y > alto) {
        posicion.y = 0;
        posicion.x = g.nextFloat() * (ancho - imagen.getWidth());
    }

    if (posicion.x+imagen.getWidth()>ancho){
        direccion=-1; //cambiamos de dirección
        posicion.x =ancho - imagen.getWidth(); //ponemos en la posición límite
    }

    if (posicion.x<0){
        direccion=1;
        posicion.x=0;
    }
}
```

Y en actualizarFisica() sustituye la llamada a moverEnemigo por zigzag. Ejecuta y observa el cambio. Analiza el código para entenderlo.

	RAMA:	Informática	CICLO:	Desenvolvimento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

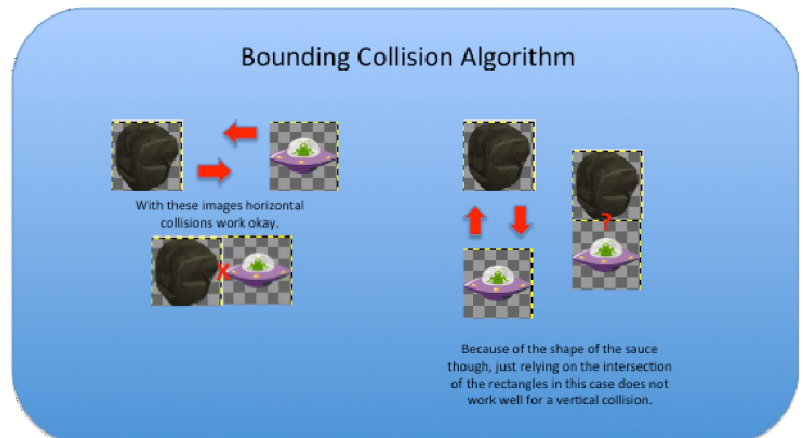
### 3.4. Detección de colisiones

La detección de colisiones entre *Sprites* o de un *Sprite* con una región de la pantalla se basa simplemente en comprobar que no se ha superado cierto límite o que se ha producido una intersección entre dos objetos.

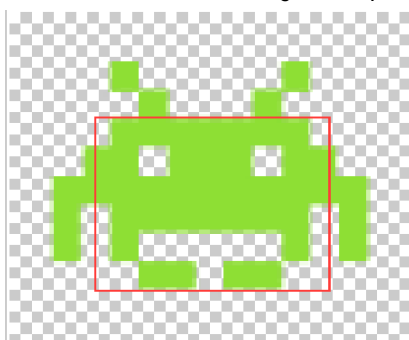
Tenemos un ejemplo muy claro en el método zigzag anterior donde se comprueba en todo momento que en la coordenada x toque una de las “paredes” de la pantalla. Si se detecta ese límite, algo cambia. En este caso es la dirección pero puede ser también la desaparición de un elemento, la destrucción del mismo, etc...

El problema de estas colisiones estriba en qué frontera usar ya que por ejemplo si se desea comprobar dos personajes habría que ver si se solapan alguno de sus *píxeles*, pero esto puede suponer mucho coste computacional por lo que se suelen realizar comprobación de regiones. Así por ejemplo se toma un rectángulo que englobe a los personajes y si dichos rectángulos se solapan, se indica que ha habido colisión.

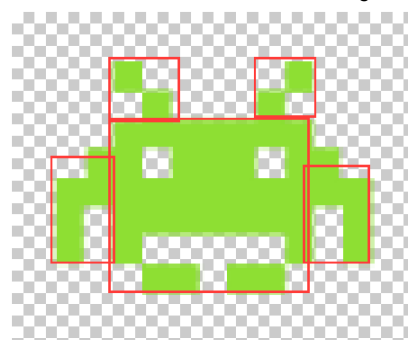
Pero ¿qué pasa si el recuadro es demasiado grande? ¿O si lo que tocan son las esquinas del rectángulo pero no los gráficos? Esto puede llevar a realizar regiones rectangulares más pequeñas o incluso disponer de varios rectángulos de detección para cada gráfico, esto haría el programa más complejo y mayor cantidad de comprobaciones (coste computacional).



Un diseño de rectángulo simple



Un diseño con varios rectángulos



De esta forma, después de cada movimiento, se comprueba si el rectángulo o los rectángulos de los distintos Sprites se solapan, y si es así se genera la colisión.

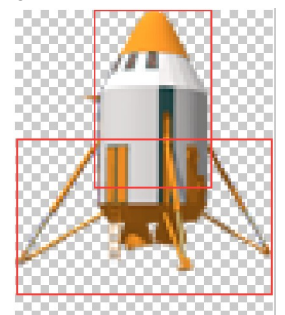
En cualquier caso la complejidad puede llegar a ser alta como se indica en este artículo:

<http://www.hobbygamedev.com/adv/2d-platformer-advanced-collision-detection/>

La demo del juego que comenta se encuentra en la siguiente dirección:


<http://www.hobbygamedev.com/flashplatformer/FlashPlatformerDemo.swf>

Vamos a ampliar nuestro ejemplo añadiéndole colisiones. Lo primero es mostrar nuestra nave y permitir que se mueva hacia los lados al pulsar en cada lado de la pantalla.



Creamos la clase Nave con dos rectángulos para tener las regiones aproximadas tal y como se ven en la imagen. Además en la clase nave tendremos la imagen y las funciones de movimiento quedando de la siguiente forma:

```
public class Nave {
    public PointF posicion;
    public Bitmap imagen;
    public Rect[] rectangulos = new Rect[2];
}
```

	RAMA:	Informática	CICLO:	Desenvolvimento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

```

public Nave(Bitmap imagen, float x, float y) { //Constructor
    this.imagen = imagen;
    this.posicion = new PointF(x, y);
    this.setRectangulos();
}

public void setRectangulos() { // Actualiza rectángulos para control de colisiones
    int anchoNave = imagen.getWidth();
    int altoNave = imagen.getHeight();
    float x = posicion.x;
    float y = posicion.y;
    rectangulos[0] = new Rect(
        (int) (x + anchoNave / 3.0),
        (int) y,
        (int) (x + anchoNave / 3 * 2.25),
        (int) (y + altoNave / 2.0));

    rectangulos[1] = new Rect(
        (int) x,
        (int) (y + altoNave / 2.0),
        (int) (x + anchoNave),
        (int) (y + altoNave - altoNave / 5));
}

public void moverDerecha(int alto, int ancho, int velocidad) { // Mueve la nave a la derecha
    if (posicion.x + imagen.getWidth() < ancho) {
        posicion.x += velocidad;
        this.setRectangulos();
    }
}

public void moverIzquierda(int alto, int ancho, int velocidad) { // Mueve la nave a la izquierda
    if (posicion.x > 0) {
        posicion.x -= velocidad;
        this.setRectangulos();
    }
}
}

```

La función setRectangulos() es la encargada de definir la región dentro de la pantalla y por tanto hay que llamarla cada vez que se mueve la nave, ya que se mueven también los rectángulos que definen su zona de colisión.

La aproximación más correcta sería realizar un set sobre las coordenadas (setX y setY) de forma que al darle un nuevo valor se ejecutara el método setRectangulos.

A continuación modificamos la clase Enemigo para que también contemple un rectángulo algo más pequeño que la propia imagen (similar al dibujo de más arriba). Le añadimos el método setRectangulo:

```

public void setRectangulo(){
    float x=posicion.x;
    float y=posicion.y;
    rectangulo=new Rect( (int)(x+0.2*imagen.getWidth()), (int)(y+0.2*imagen.getHeight()),
        (int)(x+0.8*imagen.getWidth()), (int)(y+0.8*imagen.getHeight()));
}


```

Y llamamos a dicha función al final del constructor y al final de las funciones de mover (moverEnemigo y zigzag).

Definimos ahora un objeto tipo nave en la clase principal:

```
Nave nave;
```



	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

Y lo inicializamos en el método `setSurfaceSize()` añadiendo las siguientes líneas de código dentro del `synchronize`:

```
bitmapNave = BitmapFactory.decodeResource(context.getResources(), R.drawable.nave);
nave = new Nave(bitmapNave, (anchoPantalla - bitmapNave.getWidth()) / 2, (int) (0.8 * altoPantalla));
```

Hacemos que se muestre la nave además del enemigo. Añadimos también unas líneas de depuración para que se vean los rectángulos de control de colisión:

```
public void dibujar(Canvas c) {
    try {
        c.drawBitmap(bitmapFondo, 0, 0, null); // Dibujamos el fondo
        if (!esTitulo) {
            c.drawBitmap(nave.imagen, nave.posicion.x, nave.posicion.y, null);
            c.drawBitmap(marciano.imagen, marciano.posicion.x, marciano.posicion.y, null);

            Paint p = new Paint();
            p.setColor(Color.RED);
            p.setStyle(Paint.Style.STROKE);
            p.setStrokeWidth(5);
            c.drawRect(marciano.rectangulo, p); // Dibujamos los rectángulos de colisión
            c.drawRect(nave.rectangulos[0], p);
            c.drawRect(nave.rectangulos[1], p);
        }
    } catch (Exception e) {
    }
}
```

Añadimos movimiento a la nave con toques. Lo haremos muy simple, si se pulsa en el lado derecho de la pantalla mueve la nave a la derecha y si se pulsa en el izquierdo la mueve a la izquierda. Por tanto dentro del `switch` del `onTouchEvent` ponemos:

```
case MotionEvent.ACTION_DOWN: // Primer dedo toca
    if (!esTitulo) { // Si no estamos en la pantalla de título
        if (event.getX() < anchoPantalla / 2) {
            nave.moverIzquierda(altoPantalla, anchoPantalla, PASO_NAVE);
        } else {
            nave.moverDerecha(altoPantalla, anchoPantalla, PASO_NAVE);
        }
    }
    break;
```

La constante `PASO_NAVE` la definimos como de clase:


```
private final int PASO_NAVE = 15;
```

Si queremos que la nave se mueva a una velocidad distinta tendremos que cambiar dicho valor.

Todo lo anterior lleva a la función `actualizarFisica()` donde se comprueban las colisiones, objetivo de este apartado, para lo cual se tendría que añadir el código siguiente:

```
for (Rect r : nave.rectangulos) { // Para cada rectángulo de la nave
    if (r.intersect(marciano.rectangulo)) { // Comprobamos si se solapa con el del marciano
        funcionando = false; // Si hay colisión termina el hilo y cambiamos la imagen de la nave
        nave.imagen = BitmapFactory.decodeResource(getResources(), R.drawable.nave_destruida);
    }
}
```

Recuerda que en el hilo estamos continuamente actualizando la física que gestiona el movimiento automático de lo que sucede en la `surfaceview` e inmediatamente después, una vez actualizadas las posiciones, se renderizan todos los gráficos.

	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

De forma paralela, en el hilo principal del surfaceview, se comprueba mediante eventos el movimiento del usuario.

*Nota: Al usar el Rect.intersects hay que tener en cuenta que para las comprobaciones usa el punto inicial y el ancho y el alto, esto lleva a que si definimos un rectángulo con sus puntos arriba a la derecha y abajo a la izquierda, al hallar una intersección lo va a hacer mal pues tomará un rectángulo que empiece en el mismo punto, pero lo tomará como arriba a la izquierda y luego usará el ancho y alto hacia la derecha y abajo.*

### 3.5. Realización de scroll.

Una parte importante de muchos juegos es la de dar sensación de movimiento continuo a través de un escenario que excede los límites de la pantalla del dispositivo. Como si dicha pantalla fuese una ventana a través de la que mira el jugador. En esta situación al ir moviéndose el protagonista del juego, en este caso una nave, realmente lo que se desplaza es el escenario dando efecto de movimiento más allá de los bordes del dispositivo.

Hay muchas formas de establecer scroll y lo hay en muchas direcciones, puede ser horizontal, vertical, en todas direcciones, etc. Además este puede ser automático, es decir, continuo durante el desarrollo del juego o dependiente del jugador, es decir, la pantalla se moverá según se mueve el protagonista y en dirección contraria al mismo.

En nuestro ejemplo vamos a establecer como haríamos un scroll vertical continuo, es decir, la nave permanecerá fija en la parte inferior de la pantalla pero veremos cómo se mueven las estrellas y planetas de fondo.

Cuando se establece un scroll este puede ser simplemente el movimiento de una imagen muy grande finita (por ejemplo un mapa de juego) de forma que solo se vea un fragmento a través de la pantalla. En ese caso cuando se llega a los límites de la imagen "mapa", el scroll para (típico en los juegos Beat 'em up como puede ser Final Fight).

Sin embargo en nuestro juego tenemos una nave que se desplaza y no sabemos cuánto tiempo va a estar desplazándose por lo que el scroll es continuo. Esto hay varias formas de simularlo. Una de ellas sería mediante un campo de estrellas aleatorias que vamos generando a medida que avanzamos. Otra, que es la que aplicaremos, es tener una imagen que se une perfectamente el principio y el fin y la vamos repitiendo continuamente (otra forma de realizar esto es a partir de una imagen obtenemos su simétrica, con lo que nos garantizamos que se unen tanto al principio como al final).

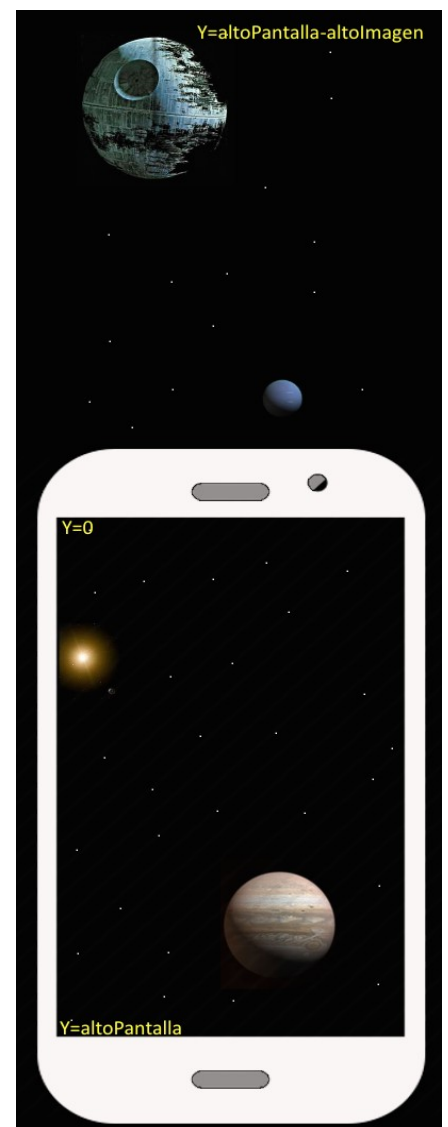
Con este planteamiento vamos a establecer el scroll en nuestro juego. Para empezar debemos establecer una clase que la controle. Esta clase dispone de unas coordenadas, una imagen y una función de movimiento.


```
public class Fondo {
    public PointF posicion;
    public Bitmap imagen;

    public Fondo(Bitmap imagen, float x, float y) { // Constructores
        this.imagen = imagen;
        this.posicion = new PointF(x, y);
    }

    public Fondo(Bitmap imagen, int altoPantalla) {
        this(imagen, 0, altoPantalla - imagen.getHeight());
    }

    public void mover(int velocidad) { // Desplazamiento
        posicion.y += velocidad;
    }
}
```



	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

Como se puede comprobar, creamos un segundo constructor que coloca la imagen en X=0 y en la Y adecuada para que lo que primero veamos sea la parte inferior de la imagen tal y como se ve en la siguiente imagen:

El movimiento de la pantalla es muy simple, sólo se desplaza a cierta velocidad en el eje Y.

Ahora que tenemos esta clase, la iniciamos cambiando la imagen antigua tierraluna.png por espacio.png. Además reescalaremos sólo el ancho, no el alto. Pero no vamos a usar una única imagen de fondo ya que al ir desplazándola esta se acaba. Por ello es necesario jugar con dos imágenes e ir colocando una encima de la otra a medida que el scroll avanza. De ahí que la declaración en la clase es un array:

```
private Fondo[] fondo;
```

Y el onTouchEvent en el case ACTION\_UP queda así:

```
case MotionEvent.ACTION_UP: // Al levantar el último dedo
    move=false;
    if (esTitulo) {
        esTitulo = false;
        bitmapFondo = BitmapFactory.decodeResource(getResources(), R.drawable.espacio);
        bitmapFondo = Bitmap.createScaledBitmap(bitmapFondo, anchoPantalla, bitmapFondo.getHeight(), true);
        fondo = new Fondo[2];
        fondo[0] = new Fondo(bitmapFondo, altoPantalla);
        fondo[1] = new Fondo(bitmapFondo, 0, fondo[0].posicion.y - bitmapFondo.getHeight());
    }
    break;
```

Cambiamos ahora la función dibujar() al principio del if dibujando los dos fondos. En la función dibujar es importante el orden de dibujo, ya que si dibujamos los fondos al final, tapará los otros sprites.

```
if (esTitulo) c.drawBitmap(bitmapFondo, 0, 0, null); // Dibujamos el fondo
else {
    c.drawBitmap(fondo[0].imagen, fondo[0].posicion.x, fondo[0].posicion.y, null);
    c.drawBitmap(fondo[1].imagen, fondo[1].posicion.x, fondo[1].posicion.y, null);
    .....
}
```

Y ahora la de actualizarFisica() donde movemos ambos fondos a la misma velocidad y comprobamos que se haya salido por la parte inferior, momento en el cual volvemos a colocarla arriba del todo, encima de la que se está dibujando actualmente. Añadimos al final de dicha función las siguientes líneas:


```
// Movemos
fondo[0].mover(5);
fondo[1].mover(5);

// Comprobamos que se sobrepase la pantalla y reiniciamos
if (fondo[0].posicion.y > altoPantalla) {
    fondo[0].posicion.y = fondo[1].posicion.y - fondo[0].imagen.getHeight();
}

if (fondo[1].posicion.y > altoPantalla) {
    fondo[1].posicion.y = fondo[0].posicion.y - fondo[1].imagen.getHeight();
}
```

Podemos usar la imagen espacio\_prueba.png (la cual tiene una línea verde en un extremo y roja en el otro) para comprobar que el scroll y el posicionamiento de las dos imágenes es el correcto (podemos observar si se solapan o si hay separación entre las imágenes).

Esto que hacemos con una imagen, podría hacerse con bloques de forma que los fondos fueran menos repetitivos y más aleatorios. Por ejemplo, se tienen 6 o 7 fondos del tamaño de una o dos pantallas y se van alternando de forma más o menos aleatoria lo que da más variabilidad al fondo, incluso se podría comprobar para dibujarlo que se "ven", es decir sus coordenadas cuadran dentro de la parte visible de la pantalla.

	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

### 3.6. Efecto Parallax.

Cuando se hace un scroll en un juego en 2D se puede mejorar el realismo del juego dándole sensación de profundidad. Una especie de 3D que lo haga más inmersivo aunque el movimiento general del juego sigue siendo en 2D.

Esto se hace usando varias capas de imágenes de fondo y moviendo cada una a una velocidad distinta. Dicha técnica es conocida como parallax y fue muy usada en cine y dibujos animados a lo largo del sXX y usada por primera vez en videojuegos en el Moon Patrol: <https://www.youtube.com/watch?v=39EsNumG3Fc>

Por tanto, si se entiende bien la técnica de scroll simple vista, aplicar el parallax es realizar lo mismo varias veces con varios fondos que, por supuesto, dispongan de transparencia para dejar ver los fondos que se encuentran más abajo.

Puedes ver distintas formas de parallax en el siguiente artículo:

<http://gamedevelopment.tutsplus.com/tutorials/parallax-scrolling-a-simple-effective-way-to-add-depth-to-a-2d-game--cms-21510>

### 3.7. Sprites animados.

Hasta ahora nuestros sprites son estáticos, es decir, se desplazan pero el gráfico no cambia, lo que hace que sea algo tosco. Vamos a hacer que cambien los gráficos a medida que el gráfico se desplaza.

Para esto hay varios métodos. El más habitual en el mundo de los videojuegos es quizá el de disponer una gran imagen con todos los frames de movimiento como la del dibujo y a medida que se mueve el sprite, cambiamos de imagen.

En Android disponemos también de otra posibilidad que es usar la clase AnimationDrawable. Es menos eficiente y consume más recursos pero en el caso de que no ser un videojuego o app que cargue demasiado el sistema se puede usar perfectamente. Además en un SurfaceView es más engorroso usarlo.

Veamos por tanto un ejemplo del primer caso. Vamos a sustituir el marciano por el pájaro de la imagen. Usaremos solo las dos primeras filas para indicar movimiento con aleteo a la derecha o a la izquierda.


Para ello lo primero es tomar la clase Enemigo y realizar algunos cambios. Añadimos, para empezar, las siguientes propiedades:

```
private Bitmap imagenes; // Bitmap con todas las imágenes
private int ancholimagenes; //Ancho del bitmap
private int altolimagenes; //alto del bitmap
private int fila = 1, col = 0; //Fila y columna de la imagen a representar
```

En el constructor lo que nos pasan es imágenes en lugar de imagen, es decir, el bitmap con todos los frames. Y dentro del constructor nos quedamos con la imagen inicial que nos interese. Como en este caso hemos establecido que la dirección original es a la derecha (direccion=1), cogemos el pájaro de la primera columna y segunda fila.

```
public Enemigo(Bitmap imagenes, float x, float y) {
    this.imagenes = imagenes;
    this.posicion = new PointF(x, y);
    ancholimagenes = imagenes.getWidth();
    altolimagenes = imagenes.getHeight();
    this.imagen = Bitmap.createBitmap(imagenes, 0, altolimagenes / 4, ancholimagenes / 4, altolimagenes / 4);
    g = new Random();
    this.setRectangulo();
}
```



	RAMA:	Informática	CICLO:	Desenvolvemto de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

Añadimos una nueva función, a la clase enemigo, la cual a partir de los valores de fila y col selecciona la imagen del bitmap para guardar en la propiedad imagen.

```
private void actualizalmagen() {
    int posIniX = (col % 4) * altoImágenes / 4;
    int posIniY = fila * anchoImágenes / 4;
    this.imagen = Bitmap.createBitmap(imágenes, posIniX, posIniY, anchoImágenes / 4, altoImágenes / 4);
}
```

Se usa col%4 porque col solo lo vamos a incrementar, sin reiniciarlo.

En esta clase por último vamos a actualizar los valores de fila y columna dependiendo del movimiento. Esto lo hacemos al final de la método zigzag.

```
fila = (direccion + 1) / 2;
col++;
actualizalmagen();
```

Haremos luego una ligera mejora sobre esto.

Ahora en la clase PruebaSurfaceView en el método setSurfaceSize() cambiamos la línea donde cargábamos el marciano por la nueva imagen:

```
bitmapMarciano = BitmapFactory.decodeResource(context.getResources(), R.drawable.pajaros);
```

Si ahora ejecutas el programa verás el movimiento del sprite. Probablemente parezca un poco brusco, para ello habría que controlar algo mejor la velocidad. Una primera posibilidad es añadir un contador de clase en Enemigo:

```
private int nuevoFrame = 0;
```

Aumentarlo al dibujar.

```
nuevoFrame++;
```

Y en el método zigzag cambiaremos las líneas donde actualizamos filas y columnas:

```
if (nuevoFrame % 10 == 0) {
    fila = (direccion + 1) / 2; // Se puede hacer con if
    col++;
    actualizalmagen();
}
```

Este método de control de frames es muy burdo, más adelante veremos otro más efectivo.

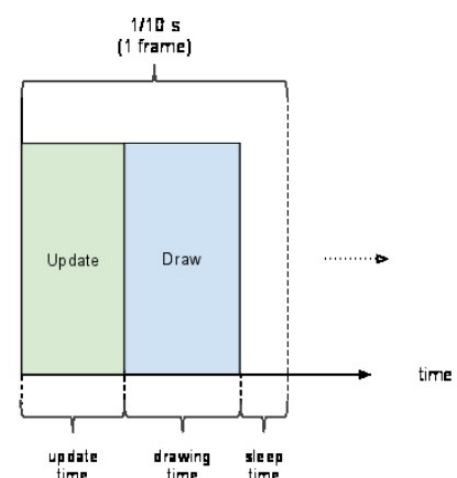
### 3.8. Control temporal


Hasta el momento, la velocidad del bucle de repetición depende de la velocidad del dispositivo, y no tenemos ningún control sobre ella. Esto no es correcto ya que en los videojuegos el hecho de que un personaje o elemento del juego vaya a más o menos velocidad puede ser determinante para la jugabilidad del mismo.

Por tanto vamos a ver cómo controlar la velocidad del juego. Primero hay que tener claro dos términos:

- Frames por segundo (FPS): que vienen a ser las veces que se dibuja en cada segundo o, según el código que hemos estado realizando, el número de veces que se ejecuta la función dibujar() en cada segundo.
- Número de veces que se actualiza la física en cada segundo, que equivale al número de veces que se ejecuta la función actualizarFisica() en cada segundo.

Lo que establece la velocidad del juego es este segundo parámetro, las actualizaciones de la física por cada segundo, ya que es lo que implica que un elemento vaya más o menos rápido.



	RAMA:	Informática	CICLO:	Desenvolvimento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

Los FPS si son iguales o mayores que las actualizaciones sólo se va a notar en que la pantalla “parpadeará” menos y siempre que sean menores que la velocidad de refresco de la pantalla, ya que si es mayor no influye.

Por tanto lo que nos interesará es fijar que número de veces vamos a mover los elementos, fijar el numero de fps y el resto del tiempo no hacer nada.

Existen mucho métodos de control de velocidad de ambos parámetros dependiendo de lo que queramos conseguir. En este artículo puedes ver 4 de ellos bastante bien explicados (ojo, es pseudocódigo, no java):

<http://www.koonsolo.com/news/dewitters-gameloop/>

En este apartado vamos a ver el primer caso que es quizá el más interesante para móviles ya que durante el tiempo que no hace nada realmente vamos a esperar mediante un `Thread.sleep()` con lo cual el consumo de batería es menor.

Por tanto en nuestro bucle calcularemos el tiempo que le lleva realizar las dos tareas (actualizar y dibujar) y dejaremos que duerma el tiempo restante.

Usaremos la función `System.nanoTime()` en lugar de `System.currentTimeMillis()` por ser más precisa. Tenéis más información y la explicación de otro de los métodos que es más efectivo en sistemas lentos (a costa de consumir más batería) en el video siguiente:

<https://www.youtube.com/watch?v=udZ4q5jUyWE>

Nuestro nuevo bucle queda de la siguiente forma:

```
@Override
public void run() {
    long tiempoDormido = 0; //Tiempo que va a dormir el hilo
    final int FPS = 50; // Nuestro objetivo
    final int TPS = 1000000000; //Ticks en un segundo para la función usada nanoTime()
    final int FRAGMENTO_TEMPORAL = TPS / FPS; // Espacio de tiempo en el que haremos todo de forma repetida

    // Tomamos un tiempo de referencia actual en nanosegundos más preciso que currenTimeMillis()
    long tiempoReferencia = System.nanoTime();


    while (funcionando) {
        Canvas c = null; //Necesario repintar _todo el lienzo
        try {
            c = surfaceHolder.lockCanvas(); // Obtenemos el lienzo. La sincronización es necesaria por ser recurso común
            synchronized (surfaceHolder) {
                if (!esTitulo) actualizarFisica(); // Movimiento de los elementos
                dibujar(c);
            }
        } finally { // haya o no excepción, hay que liberar el lienzo
            if (c != null) {
                surfaceHolder.unlockCanvasAndPost(c);
            }
        }

        // Calculamos el siguiente instante temporal donde volveremos a actualizar y pintar
        tiempoReferencia += FRAGMENTO_TEMPORAL;

        // El tiempo que duerme será el siguiente menos el actual (Ya ha terminado de pintar y actualizar)
        tiempoDormido = tiempoReferencia - System.nanoTime();

        //Si tarda mucho, dormimos.
        if (tiempoDormido > 0) {
            try {
                Thread.sleep(tiempoDormido / 1000000); //Convertimos a ms
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```



	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

Si nuestro juego carga bastante el sistema y mediante este método no se alcanza la velocidad necesaria, quizá haya que poner en práctica algunos de los otros métodos en los que los FPS y las actualizaciones de física en cada segundo no son iguales. Por ejemplo pueden lanzarse 25 FPS pero hacer 40 actualizaciones de física. O realizar muchos más FPS para un movimiento más “suave” aunque algunos frames se repitan.

### 3.9. Audio y música

Es evidente que en un juego la música y los efectos sonoros son de gran importancia. Veremos en este apartado como usar ambos en nuestro juego de una forma sincronizada con los sucesos. Antes un resumen breve de algunas clases de uso de audio (no usaremos todas):

- AudioManager: se coge con getSystemService (Context.AUDIO\_SERVICE). Gestiona el volumen y los efectos sonoros del sistema así como periféricos relacionados.
- SoundPool: colección de samples o streams de audio. Permite mezclarlos y reproducirlos al mismo tiempo.
- Ringtone y RingtoneManager: acceso a audioclips predefinidos.
- MediaPlayer: para tocar musica en playback puede resultar útil.
- MediaRecorder: para grabar audio y video.

#### 3.9.1. Efectos sonoros

Empezaremos añadiendo un efecto predefinido mediante el AudioManager. Para eso declaramos una nueva propiedad de este tipo:

```
private AudioManager audioManager;
```

A continuación la iniciamos en el constructor:

```
audioManager=(AudioManager)context.getSystemService(Context.AUDIO_SERVICE);
```

Finalmente queremos que suene un pequeño ruido en el momento que el usuario toca la pantalla principal, por lo que nos vamos al método onTouchEvent y añadimos el siguiente código:

```
case MotionEvent.ACTION_POINTER_UP: // Al levantar un dedo que no es el último
    if (esTitulo) {
        audioManager.playSoundEffect(AudioManager.FX_KEYPRESS_SPACEBAR);
        .....
    }
```

El método anterior de generar efectos es muy común sobre todo en aplicaciones y, dentro de juegos, al pulsar algún botón o similar.

A continuación añadamos efectos sonoros pero de sonidos propios a nuestro juego. Existen gran cantidad de webs donde conseguir efectos sonoros como <http://soundbible.com>, simplemente busca en google: *free sound effect*.


Una vez que tenemos los archivos de sonidos, los colocaremos en res/raw dentro de nuestro proyecto. Si no existe el directorio, créalo.

En este caso usaremos tres archivos de audio: explosión, pájaro y woosh. El primero lo lanzaremos al chocar, el segundo cada vez que el pájaro salga de la parte superior, el tercero al mover la nave.

Lo primero es crear nuestra colección de sonidos mediante SoundPool. Se debe tener en cuenta que desde la API21 se usa SoundPool.Builder.

Declaramos como propiedades de clase:

```
private SoundPool efectos;
private int sonidoWoosh, sonidoPajaro, sonidoExplosion;
final private int maxSonidosSimultaneos=10;
```

	RAMA:	Informática	CICLO:	Desenvolvemto de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

En el constructor iniciamos el SoundPool y asignamos los sonidos.

```
if ((android.os.Build.VERSION.SDK_INT) >= 21) {
    SoundPool.Builder spb=new SoundPool.Builder();
    spb.setAudioAttributes(new AudioAttributes.Builder().setUsage(AudioAttributes.USAGE_MEDIA)
        .setContentType(AudioAttributes.CONTENT_TYPE_SONIFICATION).build());
    spb.setMaxStreams(maxSonidosSimultaneos);
    this.efectos=spb.build();
} else {
    this.efectos=new SoundPool(maxSonidosSimultaneos, AudioManager.STREAM_MUSIC, 0);
}

sonidoWoosh=efectos.load(context, R.raw.woosh,1);
sonidoPajaro=efectos.load(context,R.raw.pajaro,1);
sonidoExplosion=efectos.load(context,R.raw.explosion,1);
```

Donde:

- El primer parámetro de SoundPool es la cantidad de sonidos que pueden sonar simultáneamente. Ponemos un valor alto porque puede darse el caso de pulsar varias veces seguidas a la nave, con lo que sonará múltiples veces el sonido woosh.
- El tercer parámetro de SoundPool (calidad) y el tercer parámetro de load (prioridad) no se usan actualmente por lo que siempre tendrán esos valores.

El efecto sonoro de mover nave se ejecutará en el onTouchEvent:

```
case MotionEvent.ACTION_POINTER_DOWN: // Segundo y siguientes tocan
    if (!esTitulo) { // Si no estamos en la pantalla de título
        int v= audioManager.getStreamVolume(AudioManager.STREAM_MUSIC);
        efectos.play(sonidoWoosh,v,v,1,0,1);
        .....
    }
```

La variable v (volumen) q tiene que ser un float entre 0 y 1. Además se ve afectado al final por el volumen general del sistema


Los parámetros de play son:

- El sonido a tocar
- Volumen altavoz izquierdo
- Volumen altavoz derecho
- Prioridad (si se supera el número de sonidos simultáneos)
- Número de repeticiones (-1 continuo, 0 suena una vez, 1 suena dos veces, ...)
- velocidad (0.5 la mitad de rápido, 1 velocidad normal, 2 el doble de rápido)

En el caso del sonido del pájaro, como queremos que se ejecute cuando el pájaro salga por la parte superior podemos plantearlo de dos formas: O le pasamos el sonido a la clase Enemigo o añadimos una propiedad en dicha clase que si está a true se ejecute el sonido en la función dibujar() o en actualizarFisica().

Lo haremos con la segunda forma. Es decir, definimos una nueva propiedad que controlamos en el método zigzag:

```
public boolean sonido=false;
public void zigzag(int alto, int ancho, int velocidad){
    posicion.y += velocidad;
    posicion.x += velocidad*direccion;
    sonido=false;
    if (posicion.y > alto) {
        sonido=true;
    }
    .....
}
```

	RAMA:	Informática	CICLO:	Desenvolvemto de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

Ahora por ejemplo en la función dibujar, comprobamos si dicha booleana está activa:

```

.....
c.drawBitmap(marciano.imagen, marciano.posicion.x, marciano.posicion.y, null);
if (marciano.sonido) {
    int v= audioManager.getStreamVolume(AudioManager.STREAM_MUSIC);
    efectos.play(sonidoPajaro,v,v,1,0,1);
}
.....

```

Finalmente en actualizarFisica al detectar la colisión que suene la explosión:

```

if (r.intersect(marciano.rectangulo)) {
    funcionando = false;
    nave.imagen = BitmapFactory.decodeResource(getResources(), R.drawable.nave_destruida);
    int v= audioManager.getStreamVolume(AudioManager.STREAM_MUSIC);
    efectos.play(sonidoExplosion,v,v,1,0,1);
}

```

### 3.9.2. Música

Para hacer sonar música de fondo, o en general archivos de audio más largos (por encima de 1 MB aproximadamente) se recomienda usar la clase MediaPlayer<sup>9</sup>. Esta dispone de las siguientes funciones:

- setDataSource: indica el stream (audio o video) para reproducir.
- prepare: prepara el stream. Es necesario preparar el stream tras realizar un stop para volver a reproducirlo.
- start: inicia la reproducción.
- pause: pausa la reproducción.
- stop: para la reproducción.
- seekTo: se sitúa en cierto punto del stream.
- release: libera recursos.
- getDuration: consigue la duración del archivo de audio.
- getTrackInfo: obtiene información de la pista de audio.
- getTimestamp: consigue la posición actual de la reproducción.
- isPlaying: devuelve verdadero si la canción está sonando, en otro caso es falso.
- reset: reinicia es stream de audio.
- setVolume: establece el volumen del MediaPlayer.
- isLooping: devuelve verdadero si la canción se reproduce continuamente, en otro caso es falso.
- setLooping(boolean looping): establece el estado de la reproducción continua.

En nuestro caso vamos reproducir el tema Assignment de BoxCat Games. Existen numerosas webs con música libre de derechos o podéis realizar vuestra propia música con programas tipo GarageBand, FL Studio o similares.

Cogemos el archivo y lo añadimos a res/raw como hicimos anteriormente.

Declaramos el MediaPlayer y lo inicializamos en el constructor. Mediante la función estática create indicamos el archivo a usar.


```

public MediaPlayer mediaPlayer;

// Constructor
public PruebaSurfaceView(Context context) {
    super(context);
}

```

<sup>9</sup> <https://developer.android.com/reference/android/media/MediaPlayer>

	RAMA:	Informática	CICLO:	Desenvolvimento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

```

audioManager=(AudioManager)context.getSystemService(Context.AUDIO_SERVICE);

mediaPlayer= MediaPlayer.create(context, R.raw.musica);
int v= audioManager.getStreamVolume(AudioManager.STREAM_MUSIC);
mediaPlayer.setVolume(v/2,v/2);

```

Declaramos mediaPlayer público porque vamos a usarla en la activity principal dentro del ciclo de vida de la misma.

Iniciamos la música en el inicio del juego, es decir, en el onTouchEvent:

```

case MotionEvent.ACTION_UP: // Al levantar el último dedo
    if (esTitulo) {
        mediaPlayer.start();
    }
    .....

```

Y la paramos en el momento de la colisión:

```

if (r.intersect(marciano.rectangulo)) {
    mediaPlayer.pause();
}
    .....

```

Finalmente en el ciclo de vida del activity principal, sobreescribimos las funciones onPause y onResume para parar la música y volverla a reproducir según le suceda lo mismo a la activity. Además usamos una variable booleana para que no se ejecute el start al inicio de la app, sino solo después de pasar por un onPause.

```


// Con pausa evitamos que se ejecute con la activity principal.
private boolean pausa=false;

@Override
protected void onPause() {
    super.onPause();
    pausa=true;
    pantalla.mediaPlayer.pause();
}

@Override
protected void onResume() {
    super.onResume();
    if (pausa)
        pantalla.mediaPlayer.start();
}

```

Si pantalla no la tenemos como propiedad de la clase seguramente está como variable local en onCreate. Se debe poner como propiedad de clase.

	RAMA:	Informática	CICLO:	Desenvolvimento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				


## 4. Otros elementos que se deben conocer

A continuación se plantea una serie de elementos Android que no han podido desarrollarse a lo largo del ciclo pero que es imprescindible que el alumno conozca, por lo que debería leer documentación sobre los mismos de cara incluso a la realización del proyecto:

- Content Provider: Necesario para compartir información entre Apps. Por ejemplo para acceder a las fotos o a los contactos desde una app, se utilizan contentproviders del carrete y de la agenda respectivamente. Podemos hacer apps con contentproviders para facilitar información de nuestra app a otras.
- Broadcast Receiver: Para recibir distintos avisos tanto del dispositivo (notificaciones, mensajes o llamadas entrantes, etc.) como de servicios creados.
- Centro de notificaciones: Necesario para dar avisos al usuario de distintas maneras.
- Uso de redes sociales: Hoy en día muchas apps registran distinta información en redes como twitter, facebook, instagram, foursquare, etc. Conviene conocer las APIs de acceso a estas redes para incluirlas en las apps.
- Más eventos: Cuantos más eventos sepamos controlar mejor. Es necesario profundizar en listeners como: OnClickListener, OnKeyListener, OnFocusChangeListener, etc.
- Más componentes: Y también es cierto que hemos dejado varios componentes de interfaz de usuario por ver. Cuanto más se conozca más usable llegaremos a realizar nuestras apps.
- Nuevas versiones: En cualquier ámbito de programación es necesario estar al día y pendiente de nuevas versiones de las distintas librerías que usamos así como de las novedades en las nuevas versiones del sistema operativo.

## 5. Fuentes

- <http://developer.android.com/guide/topics/graphics/2d-graphics.html>
- <http://developer.android.com/intl/es/training/gestures/multi.html>
- <http://www.edu4java.com/en/androidgame/androidgame4.html>
- <http://www.techrepublic.com/blog/software-engineer/the-abcs-of-android-game-development-detect-collisions/>
- <http://www.linux.com/learn/tutorials/707993-how-to-draw-2d-object-in-android-with-a-canvas>
- <http://www.tutorialeshtml5.com/2012/11/android-crear-animacion-con-sprites.html>

	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO	Programación Multimedia y Dispositivos Móviles					CURSO:	2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022		
	UNIDAD		COMPETENCIA					

## 6. Apéndices

### 6.1. Cambiar fuente de texto

```
Typeface faw = Typeface.createFromAsset(context.getAssets(), "fonts/fontawesome-webfont.ttf");
Paint p=new Paint();
p.setTypeface(faw);
```

### 6.2. Engines gráficos

- Libgdx:
  - <http://www.codeproject.com/Articles/702957/Create-your-first-Android-Game-with-libgdx>
  - [https://www.youtube.com/playlist?list=PLaNw\\_AbDFccHbzuObI4xHHp6WtiN2cROv](https://www.youtube.com/playlist?list=PLaNw_AbDFccHbzuObI4xHHp6WtiN2cROv)
- Cocos-2D:
  - <http://www.raywenderlich.com/33750/cocos2d-x-tutorial-for-ios-and-android-getting-started>
- OpenGL:
  - <http://developer.android.com/intl/es/guide/topics/graphics/opengl.html>

### 6.3. Programas de creación multimedia

- Inkscape: Diseño gráfico
- Gimp: Retoque digital
- Audacity: Procesado de audio

### 6.4. Comprobar la versión API de Android

```
int currentapiVersion = android.os.Build.VERSION.SDK_INT;
if (currentapiVersion >= android.os.Build.VERSION_CODES.FROYO){
    // Instrucciones para versiones mayores que FROYO
} else
{
    // Instrucciones para versiones menores que FROYO
}
```

Más en:

- [http://developer.android.com/reference/android/os/Build.VERSION\\_CODES.html](http://developer.android.com/reference/android/os/Build.VERSION_CODES.html)
- <http://developer.android.com/guide/topics/manifest/uses-sdk-element.html>

### 6.5. Apéndice IV: Animaciones y transiciones<sup>10</sup>.

Para realizar ciertas animaciones (cambios de tamaño, posición, transparencia, orientación) de una forma simple utilizando actividades y la clase View de forma normal (no SurfaceView) se pueden usar las siguientes clases:

- [Animation](#): Permite aplicar una serie de transformaciones a una View: Tamaño, posición, rotación,...
- [TransitionDrawable](#): Define transiciones entre dos capas mediante un Fade.
- [AnimationDrawable](#): Anima Drawables. Permite una animación frame a frame.

<sup>10</sup> <https://developer.android.com/training/animation/index.html>



<b>COLEXIO</b> <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	Desenvolvimento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

## 6.6. Ejemplos de código.

Veremos en este punto algunos ejemplos de código para realizar transformaciones sobre imágenes. Tomaremos como imagen de prueba la imagen siguiente:



### 6.6.1. Imagen con bodes redondeados

```
/**
 * A partir de una imagen obtenemos la misma imagen con los bordes redondeados
 * @param bitmap Imagen original
 * @param borderRadius Tamaño en px del borde redondeado
 * @return Imagen con el borde redondeado
 */
public static Bitmap bitmapRedondeado(Bitmap bitmap, float borderRadius) {
    Bitmap output = Bitmap.createBitmap(bitmap.getWidth(), bitmap.getHeight(),
        Bitmap.Config.ARGB_8888);

    Canvas canvas = new Canvas(output);

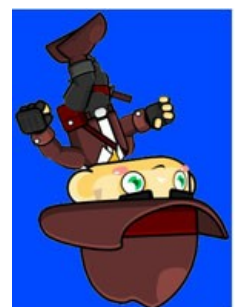
    final int color = 0xff424242;
    final Rect rect = new Rect(0, 0, bitmap.getWidth(), bitmap.getHeight());
    final RectF rectF = new RectF(rect);

    final Paint paint = new Paint();
    paint.setAntiAlias(true);
    paint.setColor(color);
    canvas.drawARGB(0, 0, 0, 0);
    canvas.drawRoundRect(rectF, borderRadius, borderRadius, paint);
    paint.setXfermode(new PorterDuffXfermode(PorterDuff.Mode.SRC_IN));
    canvas.drawBitmap(bitmap, rect, rect, paint);
    return output;
}
```



### 6.6.2. Simétrico de una imagen

```
public Bitmap espejo(Bitmap imagen, Boolean horizontal){
    Matrix matrix = new Matrix();
    if (horizontal) matrix.preScale(-1, 1);
    else matrix.preScale(1, -1);
    return Bitmap.createBitmap(imagen, 0, 0, imagen.getWidth(),
        imagen.getHeight(), matrix, false);
}
```



<b>COLEXIO VIVAS S.L.</b>	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

### 6.6.3. Convertir una imagen a escala de grises

```

/**
 * Convierte una imagen a escala de grises
 * @param imgOriginal Imagen original
 * @return Imagen convertida a escala de grises
 */
public Bitmap toGrayscale(Bitmap imgOriginal) {
    Bitmap bmpGrayscale = Bitmap.createBitmap(imgOriginal.getWidth(),
                                                imgOriginal.getHeight(), Bitmap.Config.ARGB_8888);

    Canvas c = new Canvas(bmpGrayscale);
    Paint paint = new Paint();
    ColorMatrix cm = new ColorMatrix();
    cm.setSaturation(0);
    ColorMatrixColorFilter f = new ColorMatrixColorFilter(cm);
    paint.setColorFilter(f);
    c.drawBitmap(imgOriginal, 0, 0, paint);
    return bmpGrayscale;
}

```



### 6.6.4. Cambiar el brillo y el contraste de una imagen

```

/**
 * Cambia el contraste y el brillo de una imagen
 * @param bmp imange de entrada
 * @param contraste Contraste 0..10 -> 1 es el valor por defecto
 * @param brillo Brillo -255..255 -> 0 es el valor por defecto
 * @return La nueva imagen con el contraste y el brillo cambiados
 */
public Bitmap cambiarContrasteBrillo(Bitmap bmp, float contraste, float brillo) {
    ColorMatrix cm=new ColorMatrix(new float[] {
        contraste, 0, 0, 0, brillo,
        0, contraste, 0, 0, brillo,
        0, 0, contraste, 0, brillo,
        0, 0, 0, 1, 0
    });

    Bitmap ret=Bitmap.createBitmap(bmp.getWidth(), bmp.getHeight(), bmp.getConfig());
    Canvas canvas=new Canvas(ret);
    Paint paint=new Paint();
    paint.setColorFilter(new ColorMatrixColorFilter(cm));
    canvas.drawBitmap(bmp, 0, 0, paint);
    return ret;
}

```



<b>COLEXIO</b> <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma		
	MÓDULO	Programación Multimedia y Dispositivos Móviles				CURSO: 2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2021/2022
	UNIDAD	COMPETENCIA				

### 6.6.5. Añadir un borde a una imagen

```

/**
 * Añade un borde a una imagen
 * @param imagen Imagen a añadir el borde
 * @param anchoBorde Ancho del borde
 * @param color Color de borde
 * @return Imagen con el borde
 */
public Bitmap addImageBorder(Bitmap imagen, int anchoBorde, int color) {
    Bitmap imagenConBorde = Bitmap.createBitmap(imagen.getWidth() + anchoBorde * 2,
        imagen.getHeight() + anchoBorde * 2, imagen.getConfig());

    Canvas canvas = new Canvas(imagenConBorde);
    canvas.drawColor(color);
    canvas.drawBitmap(imagen, anchoBorde, anchoBorde, null);
    return imagenConBorde;
}

```

