Gráficos 2D

Contenido

Vistas personalizadas

El SDK proporciona muchos tipos de vistas (controles) para utilizar. En ocasiones, si queremos dar un toque especial y original a nuestra aplicación, o simplemente si necesitamos cierta funcionalidad no presente en los componentes estándar, nos vemos en la necesidad de crear nuestros propios controles personalizados, diseñados a la medida de nuestros requisitos.
Existen tres maneras de hacerlo:

- Extendiendo la funcionalidad de un control ya existente: Ejemplo de personalización de EditText (buena explicación del uso de los métodos **getResources().getDisplayMetrics().density** para obtener la escala necesaria para visualizar en pantallas con distintas densidades de píxeles).
- Combinando varios controles para formar otro más complejo: Ejemplo de personalización de LinearLayout
- Diseñando desde cero un nuevo control: vamos a heredar nuestro control directamente de la clase View (clase padre de la gran mayoría de elementos visuales). Esto implica, entre otras cosas, que por defecto nuestro control no va a tener ningún tipo de interfaz gráfica, por lo que todo el trabajo de "dibujar" la interfaz lo vamos a tener que hacer nosotros utilizando las API para gráficos.

API's gráficas

Android nos proporciona a través de sus API's gráficas una potente y variada colección de funciones que pueden cubrir prácticamente cualquier necesidad gráfica de una aplicación.
Existen esencialmente dos librerías para tratar con gráficos:

- una orientada principalmente a gráficos 2D llamada Graphics "Clásica" o también llamada Canvas.
- otra orientada tanto a 2D como a 3D con OpenGL ES:
  - OpenGL ES 1.0 and 1.1 - This API specification is supported by Android 1.0 and higher.
  - OpenGL ES 2.0 - This API specification is supported by Android 2.2 (API level 8) and higher.
  - OpenGL ES 3.0 - This API specification is supported by Android 4.3 (API level 18) and higher.
  - OpenGL ES 3.1 - This API specification is supported by Android 5.0 (API level 21) and higher.

Siempre hay que tener en cuenta que la memoria en los dispositivos móviles es limitada y las imágenes en general ocupan bastante espacio. Por ello, hay que tener en cuenta dos cosas:

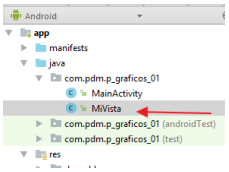
- Mantener cargadas sólo aquellas imágenes que estamos usando
- Utilizar el tamaño óptimo: es decir, si tenemos una imagen que en la pantalla tiene que mostrarse más pequeña de lo que en realidad es, hay que hacerla pequeña al cargarla para que no ocupe tanta memoria.

Diseñando vistas

El trabajo consiste en no utilizar un layout diseñado desde XML sino en diseñar programáticamente una nueva vista propia (MiVista).
Por eso, borraremos los layouts activity_main y content_main suministrados por el asistente.
La clase MiVista puede ser una clase interna de nuestra actividad o una clase propia de nuestro paquete en cuyo caso podrá ser reutilizada:

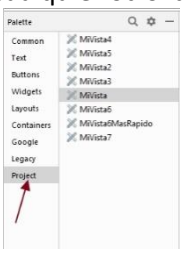
- de forma directa en diversas actividades, enviando a su constructor al menos el contexto de la aplicación para que la clase pueda acceder a los recursos:

```
public class MainActivity extends AppCompatActivity {

@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(new MiVista(this));
```

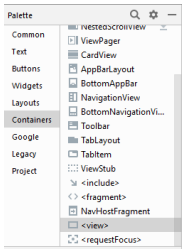



```
}
}
```

- en cualquier otro layout



o bien



Esa nueva clase puede:

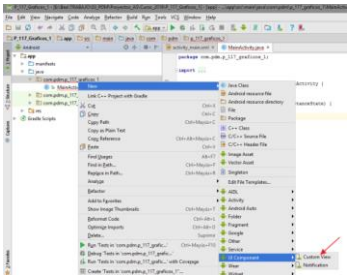
- extender de la clase **View**: trabaja en el hilo principal. Es la mejor opción si es una vista estática o si hay pocos movimientos o son lentos (si hay que redibujar habrá que llamar al método invalidate())
- extender de la clase **SurfaceView**: trabaja en hilos secundarios. Esta es la mejor opción para simular movimientos (vista dinámica, si hay que redibujar habrá que llamar al método postInvalidate()).

En cualquiera de las dos opciones se trabaja con objetos de tipo Canvas, de tipo Paint, de tipo Path, de tipo Drawable, etc.

- **Canvas**: "lienzo" que representa una superficie de nuestra pantalla donde podemos crear una imagen (Bitmap) a base de funciones básicas de dibujo, como pueden ser dibujar un rectángulo, escribir un texto o añadir otro Bitmap.
  Este mapa de bits siempre es generado de forma automática cuando sobrecargamos el evento onDraw()/lockCanvas() propio de View/SurfaceView.
  Un objeto de este tipo puede crearse:
  - A partir de un Bitmap
    ```
    Bitmap b = Bitmap.createBitmap (100, 100, Bitmap.Config.ARGB_8888);
    C = new Canvas (b);
    ```
  - Pero es **RECOMENDABLE** utilizar un Canvas ya creado para una View o SurfaceView. Puede obtenerse con los métodos:
    ```
    Void View.onDraw(Canvas canvas)
    Canvas SurfaceHolder.lockCanvas()
    ```
- **Paint**: "pincel" que permite definir el color, estilo o grosor
- **Path**: "camino" creado a partir de segmentos de rectas o curvas
- **Drawable**: abstracción que representa "algo que puede ser dibujado"

Creación de nuevas vistas de tipo View

AS nos ofrece un asistente pero es lioso y tiene errores que hay que corregir, así que lo obviaremos de momento.



Cuando se quiere crear una nueva vista, hay que extender la clase View, escribir el constructor y como mínimo sobrescribir el método onDraw().

```
public class MiVista extends View {
    public MiVista(Context context) {
        super(context);
    }

    @Override
        protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
    }
}
```

El constructor es el lugar adecuado para crear todos los componentes de la vista, pero cuidado, en este punto todavía no se conoce las dimensiones que tendrá.

El método llamado por el sistema cuando necesita dibujar la vista. Utiliza el objeto "lienzo" canvas.

Todo objeto que herede de View lleva consigo un ViewHolder, poseedor de la superficie de la vista, encargado de facilitarnos el acceso al dibujo del Canvas mediante su bloqueo para ser dibujado con el método onDraw y luego desbloquearlo para mostrarlo. Este proceso que realiza el holder se realiza en objetos View de forma automática (no así con SurfaceView que tendremos que programar).

Cuando sobrecargamos el evento onDraw() se generará de forma automática como parámetro un objeto Canvas, representando al propio Canvas de la vista personalizada y de forma implícita su mapa de bits para que podamos empezar a dibujar píxeles. Tendrá por defecto un fondo blanco con el ancho y alto de su vista contenedora.

Clase Paint

Representa un pincel. Permite definir el estilo y el color cuando se dibuja.

```
Paint miPincel = new Paint();
```

Algunos de sus métodos:

- setColor(int color): Permite indicar el color del pincel.
  - Con constantes del sistema:
    ```
    miPincel.setColor(Color.BLUE);
    ```
  - Con colores definidos en el fichero /res/values/color.xml
    ```
    miPincel.setColor(ContextCompat.getColor(context, R.color.colorAccent));
    ```
  - Con enteros de 32 bits, divididos en 4 campos de bits: alfa, rojo, verde y azul (ARGB) o en hexadecimal.
    ```
    miPincel.setColor(Color.argb(127, 0, 255, 0));
    ```
- setAlpha(int alpha): permite fijar la transparencia con valores [0,255]
  ```
  miPincel.setAlpha(125);
  ```
- setStrokeWidth (float width): grosor del pincel
  ```
  miPincel.setStrokeWidth(8);
  ```
- setStyle(Paint.Style estilo): fija el estilo solo contorno (STROKE), relleno (FILL), ambos (FILL_AND_STROKE);
  ```
  miPincel.setStyle(Paint.Style.STROKE);
  ```
- setTextAlign(Paint.Align justificación): fija el tipo de justificación con los valores posibles Align.CENTER, Align.LEFT o Align.RIGTH
- setText (float tamaño): tamaño de la fuente
- setTypeface(Typeface fuente): tipo de fuente (Typeface.MONOSPACE, Typeface.SANS_SERIF, Typeface.SERIF, Typeface.DEFAULT, Typeface.DEFAULT_BOLD))
- setUnderline(boolean subrayado): habilita/deshabilita subrayado

Clase Path

Permite definir un trazado a partir de segmentos de rectas o curvas. Dicho trazado puede dibujarse en el canvas o ser usado como guía para escribir un texto.
Algunos de sus métodos:

- addCircle(float x, float y, float radio,Direction direccion)
- moveTo(float x, float y)

Clase Canvas

Tiene muchos métodos. Veamos a continuación algunos métodos de ellos.

- Para dibujar figuras geométricas:

```
drawCircle(float cx, float cy, float radio, Paint pincel)
drawRect(float cx, float cy, float ancho, float alto, Paint pincel)
drawRect(RectF rect, Paint pincel)
drawOval(RectF ovalo, Paint pincel)
drawRect(RectF rect, Paint pincel)
drawPoint(float x, float y, Paint pincel)
drawPoints(float[] pts, Paint pincel)
```

- Para dibujar líneas y arcos:

```
drawLine(float iniX, float iniY, float finX, float finY, Paint pincel)
drawLines(float[] puntos, Paint pincel)
drawArc(RectF ovalo, float iniAnglulo, float anglulo, boolean usarCentro,Paint pincel)
drawPath(Path trazo, Paint pincel)
```

- Para dibujar texto:

```
drawText(String texto, float x, float y, Paint pincel)
drawTextOnPath(String texto, Path trazo, float desplazamHor, float desplazamVert, Paint
pincel)
drawPosText(String texto, float[] posicion, Paint pincel)
```

- Para rellenar todo el Canvas (a no ser que se haya definido un *Clip*)

```
drawColor(int color)
drawARGB(int alfa, int rojo, int verde, int azul)
drawPaint(Paint pincel)
```

- Para dibujar imágenes:

```
drawBitmap(Bitmap bitmap, Matrix matriz, Paint pincel)
```

- Si definimos un *Clip*, solo se dibujará en el área indicada:

```
boolean clipRect(RectF rectangulo)
boolean clipRegion(Region region)
boolean clipPath(Path trazo)
```

- Definir una matriz de transformación (Matrix) nos permitirá transformar coordenadas aplicando una translación, escala o rotación.

```
setMatrix(Matrix matriz)
Matrix getMatrix()
concat(Matrix matriz)
translate(float despazX, float despazY)
scale(float escalaX, float escalaY)
rotate(float grados, float centroX, float centroY)
skew(float despazX, float despazY)
```

- **Para averiguar el tamaño del Canvas**:

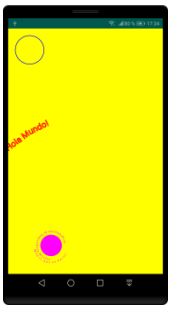
```
int getHeight()
int getWidth()
```

Pruebas básicas de dibujo

Añadimos algunas pruebas a MiVista

```
public class MiVista1 extends View {
    Context context;
    Paint miPincel;
    Path miCamino;

    public MiVista(Context context) {
        super(context);
        this.context = context;
        miPincel=new Paint();
        miCamino=new Path();
    }

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        //Fijamos color fondo pantalla
        canvas.drawColor(Color.YELLOW);
        //Fijamos las propiedades del pincel
        miPincel.setColor(Color.BLUE);
        miPincel.setStrokeWidth(4);
        miPincel.setStyle(Paint.Style.STROKE);
        miPincel.setTextSize(60);
        //Dibujamos un circulo azul
        canvas.drawCircle(150, 150, 100, miPincel);
        //Obtenemos ancho y alto de la vista
        int ancho=canvas.getWidth();
        int alto = canvas.getHeight();
        //Situamos el inicio del camino en mitad de la pantalla
        miCamino.moveTo(0, alto / 2);
        //Luego indicamos todos los otros puntos en forma
        //consecutiva llamando al método lineTo
        miCamino.lineTo(40, alto / 2 - 30);
        miCamino.lineTo(80, alto / 2 - 60);
        miCamino.lineTo(120, alto / 2 - 90);
        miCamino.lineTo(160, alto / 2 - 120);
        miCamino.lineTo(220, alto / 2 - 150);
        miCamino.lineTo(280, alto / 2 - 180);
        miCamino.lineTo(340, alto / 2 - 210);
        //cambiamos el color del pincel a rojo
        miPincel.setARGB(255, 255, 0, 0);
        //Dibujamos el texto siguiendo el camino sin desplazamientos
        canvas.drawTextOnPath("Hola Mundo!", miCamino, 0, 0, miPincel);
        //Reseteamos el camino
        miCamino.reset();
        //Fijamos como camino un circulo
        //dibujado en el sentido contrario a las agujas del reloj
        miCamino.addCircle(300, alto-200, 75, Path.Direction.CCW);
        //cambiamos propiedades pincel
        miPincel.setColor(Color.MAGENTA);
        miPincel.setStrokeWidth(1);
        miPincel.setStyle(Paint.Style.FILL);
        miPincel.setTextSize(20);
        miPincel.setTypeface(Typeface.SANS_SERIF);
        //dibujamos el camino
        canvas.drawPath(miCamino, miPincel);
        //dibujamos el texto sobre el camino
        // 10: distancia desde el comienzo del trazo al comienzo del texto
        // 40: separación del texto de la ruta (puede ser negativa)
        canvas.drawTextOnPath("Programación de Dispositivos Móviles", miCamino, 10, 30,
miPincel);
        canvas.drawTextOnPath("Mira lo que sé hacer!", miCamino, 250, 50, miPincel);
    }
}
```

Ejecuta la aplicación



Clase Drawable

Es una abstracción que representa "algo que puede ser dibujado en pantalla". Se extiende para definir gran variedad de objetos gráficos más específicos (muchos de ellos pueden ser definidos como recursos usando ficheros XML).

Known Direct Subclasses

AnimatedVectorDrawable, AnimatedVectorDrawableCompat, BitmapDrawable, ColorDrawable, DrawableContainer, DrawableWrapper, DrawerArrowDrawable, GradientDrawable, LayerDrawable, NinePatchDrawable, PictureDrawable, RoundedBitmapDrawable, ShapeDrawable, VectorDrawable, VectorDrawableCompat

Known Indirect Subclasses

AnimatedStateListDrawable, AnimationDrawable, ClipDrawable, InsetDrawable, LevelListDrawable, PaintDrawable, RippleDrawable, RotateDrawable, ScaleDrawable, StateListDrawable, TransitionDrawable

Entre ellos tenemos los siguientes:

- BitmapDrawable: Imagen basada en un fichero gráfico (PNG o JPG). Etiqueta XML <bitmap>.
- ShapeDrawable: Permite realizar un gráfico a partir de primitivas vectoriales, como formas básicas (círculos, cuadrados,…) o trazados (Path). No puede ser definido mediante un fichero XML.
- StateListDrawable: Este drawable puede mostrar diferentes contenidos (que a su vez son drawables) según el estado en el que se encuentre. Por ejemplo sirve para definir un botón, que se mostrará de forma distinta según si está normal, presionado, o inhabilitado. Ya lo hemos visto!. Etiqueta XML <selector>.
- GradientDrawable: Degradado de color que puede ser usado en botones o fondos.
- TransitionDrawable: Una extensión de LayerDrawables que permite un efecto de fundido entre la primera y la segunda capa. Para iniciar la transición hay que llamar a startTransition(inttiempo). Para visualizar la primera capa hay que llamar a resetTransition(). Etiqueta XML<transition>.
- AnimationDrawable: Permite crear animaciones frame a frame a partir de una serie de objetosDrawable. Etiqueta XML <animation-list>
- También puede ser interesante usar la clase Drawable o uno de sus descendientes como base para crear clases gráficas propias.

Though usually not visible to the application, Drawables may take a variety of forms:

- **Bitmap**: the simplest Drawable, a PNG or JPEG image.
- **Nine Patch**: an extension to the PNG format allows it to specify information about how to stretch it and place things inside of it.
- **Vector**: a drawable defined in an XML file as a set of points, lines, and curves along with its associated color information. This type of drawable can be scaled without loss of display quality.
- **Shape**: contains simple drawing commands instead of a raw bitmap, allowing it to resize better in some cases.
- **Layers**: a compound drawable, which draws multiple underlying drawables on top of each other.
- **States**: a compound drawable that selects one of a set of drawables based on its state.
- **Levels**: a compound drawable that selects one of a set of drawables based on its level.
- **Scale**: a compound drawable with a single child drawable, whose overall size is modified based on the current level.

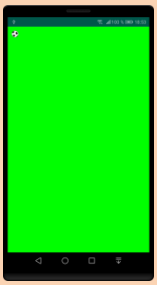
Métodos de la clase Drawable

La clase dispone de métodos. Algunos de ellos

- El método (de llamada **OBLIGATORIA**) **setBounds(x1,y1,x2,y2)** permite indicar el rectángulo donde ha de ser dibujado. Todo Drawable debe respetar el tamaño solicitado por el cliente, es decir, ha de permitir el escalado. Podemos consultar el tamaño preferido de un Drawable mediante los métodos **getIntrinsicHeight()** y **getIntrinsicWidth()**.
- Podremos dibujar objetos de tipo *Drawable* en nuestro lienzo, mediante el método **draw(canvas)** de la clase Drawable.

La forma más sencilla de añadir gráficos a la aplicación es incluirlos en la carpeta res/drawable del proyecto. El SDK de Android soporta los formatos PNG, JPG y GIF. El formato aconsejado es PNG, aunque si el tipo de gráfico así lo recomienda también puedes utilizar JPG. El formato GIF está desaconsejado.

```
public class MiVista2 extends View {
    private Drawable miImagen;

    public MiVista2(Context context) {
        super(context);
        miImagen= ContextCompat.getDrawable(context, R.drawable.ball);
        int alto=miImagen.getIntrinsicWidth();
        int ancho=miImagen.getIntrinsicHeight();
        miImagen.setBounds(30,30,30+ancho,30+alto);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        canvas.drawColor(Color.GREEN);
        miImagen.draw(canvas);
    }

}
```



BitMapDrawable

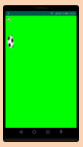
En vez de trabajar con la clase Drawable, lo podemos hacer también con su subclase BitMapDrawable y los objetos de tipo BitMap.

```
public class MiVista3 extends View {
    private Bitmap bitmap;
    private Bitmap bitmapEscalado;

    public MiVista3(Context context) {
        super(context);
        bitmap = BitmapFactory.decodeResource(context.getResources(), R.drawable.ball);
        bitmapEscalado = Bitmap.createScaledBitmap(bitmap, 100, 200, false);
    }

    public void onDraw(Canvas canvas) {
        canvas.drawColor(Color.GREEN);
        canvas.drawBitmap(bitmap, 30, 30, null);
        canvas.drawBitmap(bitmapEscalado, 30, 300, null);
    }
}
```

Las imágenes se encapsulan en la clase Bitmap

Para crear un bitmap a partir de un fichero de imagen utilizaremos la clase BitmapFactory. Dentro de ella tenemos varios métodos con prefijo decode que nos permiten leer las imágenes de diferentes formas: de un array de bytes en memoria, de un flujo de entrada, de un fichero, de una URL o de un recurso de la aplicación (nuestro caso).

Podemos "clonar" el bitmap con nuevas dimensiones con el método createScaledBitmap()



Se dibujan en el canvas con el método drawBitmap()

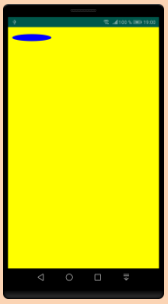
Podemos obtener un objeto de la clase Drawable a partir de un objeto de la clase Bitmap:

```
Drawable drawable = new BitmapDrawable(getResources(),bitmap);
```

ShapeDrawable

Esta clase permite dibujar gráficos a partir de formas básicas. Un ShapeDrawable es una extensión de Drawable, por lo tanto puede utilizar todo lo que permite Drawable.

```
public class MiVista4 extends View {
    private ShapeDrawable shapeDrawable;

    public MiVista4(Context context) {
        super(context);
        shapeDrawable = new ShapeDrawable(new OvalShape());
        shapeDrawable.getPaint().setColor(Color.BLUE);
        shapeDrawable.setBounds(30, 50, 310, 100);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        canvas.drawColor(Color.YELLOW);
        shapeDrawable.draw(canvas);
    }

}
```



Otros métodos de la clase View

Nuestra vista, puede tener sobrescritos entre otros los siguientes métodos:

Category	Methods	Description
Creation	Constructors	There is a form of the constructor that are called when the view is created from code and a form that is called when the view is inflated from a layout file. The second form should parse and apply any attributes defined in the layout file.
	`onFinishInflate()`	Called after a view and all of its children has been inflated from XML.
Layout	`onMeasure(int, int)`	Called to determine the size requirements for this view and all of its children.
	`onLayout(boolean, int, int, int, int)`	Called when this view should assign a size and position to all of its children.
	`onSizeChanged(int, int, int, int)`	Called when the size of this view has changed.
Drawing	`onDraw(Canvas)`	Called when the view should render its content.
Event processing	`onKeyDown(int, KeyEvent)`	Called when a new key event occurs.
	`onKeyUp(int, KeyEvent)`	Called when a key up event occurs.
	`onTrackballEvent(MotionEvent)`	Called when a trackball motion event occurs.
	`onTouchEvent(MotionEvent)`	Called when a touch screen motion event occurs.
Focus	`onFocusChanged(boolean, int, Rect)`	Called when the view gains or loses focus.
	`onWindowFocusChanged(boolean)`	Called when the window containing the view gains or loses focus.
Attaching	`onAttachedToWindow()`	Called when the view is attached to a window.
	`onDetachedFromWindow()`	Called when the view is detached from its window.
	`onWindowVisibilityChanged(int)`	Called when the visibility of the window containing the view has changed.

Utilizando constructores convenientemente

Hasta ahora nuestras vistas personalizadas solo las hemos utilizado programáticamente:

```
setContentView(new MiVista4(this));
```

pero a veces nos interesa utilizarla en recursos de tipo layout a cuyos atributos desearemos acceder.

Necesitamos cambiar el constructor:

```
public class MiVista5 extends View {
    private ShapeDrawable miImagen;

    public MiVista5(Context context, AttributeSet attrs) {
        super(context, attrs);
        //Inicializa la vista
        //Ojo: Aún no se conocen sus dimensiones si está dentro de un layout
        miImagen = new ShapeDrawable(new OvalShape());
        miImagen.getPaint().setColor(Color.BLUE);
    }

    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh) {
        super.onSizeChanged(w, h, oldw, oldh);
        //Informan del ancho y el alto
        miImagen.setBounds(0, 0, w, h);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        miImagen.draw(canvas);
    }
}
```

Observa como el constructor utilizado tiene dos parámetros: El primero de tipo Context permitirá acceder al contexto de aplicación, por ejemplo para utilizar recursos de esta aplicación. El segundo, de tipo AttributeSet, permitirá acceder a los atributos de esta vista, cuando sea creada desde XML. El constructor es el lugar adecuado para crear todos los componentes de tu vista, pero cuidado, en este punto todavía no se conoce las dimensiones que tendrá.

Android realiza un proceso de varias pasadas para determinar el ancho y alto de cada vista dentro de un *Layout*. Cuando finalmente ha establecido las dimensiones de una vista llamará a su método onSizeChanged(). Nos indica como parámetros el ancho (w) y alto asignado (h). En caso de tratarse de un reajuste de tamaños, por ejemplo una de las vistas del Layout desaparece y el resto tienen que ocupar su espacio, se nos pasará el ancho y alto anterior. Si es la primera vez que se llama al método estos parámetros valdrán 0.

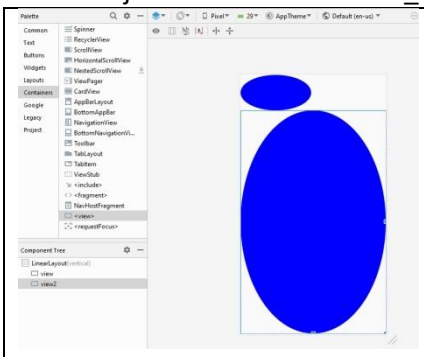
En nuestra activity podremos seguir utilizando

```
setContentView(new MiVista5(this,null));
```

O utilizar un layout creado con nombre mi_layout.xml y que contenga la vista (en el ejemplo 2 veces)

```
setContentView(R.layout.mi_layout);
```

Hemos fijado el contenido de mi_layout.xml:



```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
android:layout_width="match_parent"
    android:layout_height="match_parent">

    <view
        android:layout_width="200dp"
        android:layout_height="100dp"
        class="com.pdm.p_102_graficos.MiVista5"
        android:id="@+id/view"/>

    <view
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        class="com.pdm.p_102_graficos.MiVista5"
        android:id="@+id/view2" />

</LinearLayout>
```

Este constructor también puede utilizarse para crear atributos propios en la vista

Más usos de ShapeDrawable

Drawables en XML

- Definimos `drawable/rectangulo.xml`

```
<shape xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:shape="rectangle">
  <solid android:color="#f00"/>
  <stroke android:width="2dp" android:color="#00f"
    android:dashWidth="10dp" android:dashGap="5dp"/>
</shape>
```

- Lo utilizamos en un componente `Button`

```
<Button android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:background="@drawable/rectangulo" />
```

Drawables en código Java

- Definir el objeto `Drawable`

```
RectShape r = new RectShape();
ShapeDrawable sd = new ShapeDrawable(r);
sd.getPaint().setColor(Color.RED);
sd.setIntrinsicWidth(100);
sd.setIntrinsicHeight(50);
```

- Mostrar en un componente

```
ImageView visor = (ImageView)findViewById(R.id.visor);
visor.setImageDrawable(sd);
```

Usando SurfaceView

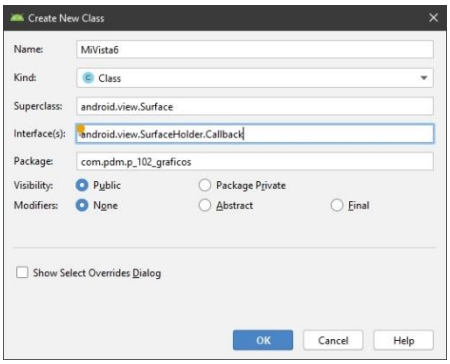
Si necesitamos contar con una elevada tasa de refresco, como por ejemplo en el caso de un videojuego con mucho movimiento, será recomendable utilizar una vista de tipo **SurfaceView** que está diseñada para ser más eficiente con el renderizado (generación) de formas e imágenes ya que es una subclase especial de View que nos permite tomar control total del dibujo de la pantalla en un hilo secundario, lo que es imprescindible para juegos o animaciones fluidas.

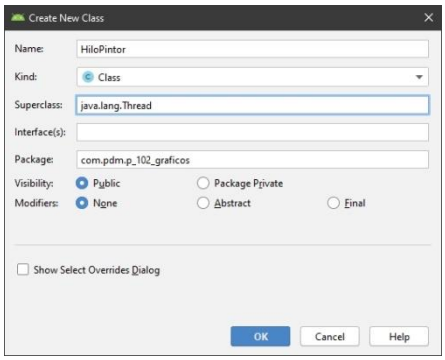
Para hacer esto tenemos que crear una clase que extienda de SurfaceView. Pero en este caso no bastará con definir el método onDraw, ahora deberemos proporcionarle la superficie en la que dibujar (SurfaceHolder) e implemente la interfaz SurfaceHolder.Callback que nos notificará cuando ocurra algún cambio en la superficie (la creación, modificación o destrucción de la misma).

Crearemos también una clase HiloPintor que extiende de Thread y que se encargará de realizar todas las operaciones de dibujo en la superficie.

(El código fuente de este hilo podría incluirse en el interior de la clase SurfaceView, pero si es una clase propia, puede utilizarse en aplicaciones que tengan más de una SurfaceView).

Volvamos a querer una actividad que muestre un balón de fútbol en la pantalla, pero ahora utilizando SurfaceView.





(Nota: si usando el asistente escoges "Show Select Overrides Dialog", puedes escoger los métodos que quieres que tenga la clase y evitas teclear algo de código).

Introduciendo la interfaz SurfaceHolder.

Si queremos dibujar el Canvas de la superficie debemos hacerlo de forma explícita, solicitando los recursos a la interfaz SurfaceHolder, poseedora de la superficie y consecuentemente poseedora del Canvas de la clase SurfaceView para su acceso.

```
public class MiVista6 extends SurfaceView implements SurfaceHolder.Callback {
    SurfaceHolder holder;
    HiloPintor hiloPintor;
    private Bitmap balon;

    public MiVista6(Context context) {
        super(context);
        holder=getHolder();
        holder.addCallback(this);
        balon = BitmapFactory.decodeResource(getResources(), R.drawable.ball);
    }

    @Override
    public void surfaceCreated(SurfaceHolder holder) {
        hiloPintor = new HiloPintor(holder, this);
        hiloPintor.start();
    }

    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {
        //Método por el que pasa cuando la superficie cambia
    }

    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        try {
            hiloPintor.debeEjecutarse =false;
            hiloPintor.join();
        } catch (InterruptedException ignored) {
        }
    }
```

En el constructor de nuestra vista declararamos nuestra interfaz y llamamos al método getHolder() para que pueda ser usada en nuestro hilo y realizar la secuencia de dibujo.

Cuando la superficie se ha creado pondremos en marcha nuestro hilo de dibujado con el método de la clase Thread start().

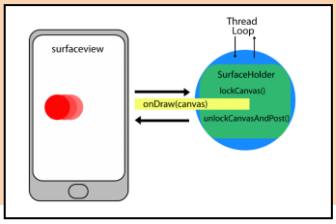
Cuando la superficie sea destruida pararemos el hilo.

```
    @Override
    public void onDraw(Canvas canvas) {
    }

    void pintarBalon(Canvas canvas) {
        canvas.drawColor(Color.GREEN);
        canvas.drawBitmap(balon, 30, 30, null);
    }
}
```

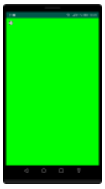
Veamos el código de nuestra clase HiloPintor:

```
public class HiloPintor extends Thread {
    private final MiVista6 miVista6;
    private final SurfaceHolder holder;
    boolean debeEjecutarse;


    HiloPintor(SurfaceHolder holder, MiVista6 miVista6) {
        this.holder=holder;
        this.miVista6=miVista6;
        debeEjecutarse =true;
    }

    @Override
    public void run() {
        super.run();
        while (debeEjecutarse) {
            Canvas canvas = null;
            try {
                canvas = holder.lockCanvas(null);
                synchronized (holder) {
                    // Dibujar los gráficos
                    miVista6.pintarBalon(canvas);
                }
            } finally {
                if (canvas != null) {
                    holder.unlockCanvasAndPost(canvas);
                }
            }
        }
    }
}
```

Podemos ver que en el bucle principal de nuestro hilo obtenemos el lienzo (`Canvas`) a partir de la superficie (`SurfaceHolder`) mediante el método `lockCanvas`. Esto deja el lienzo bloqueado para nuestro uso, por ese motivo es importante asegurarnos de que siempre se desbloquee. Para tal fin hemos puesto `unlockCanvasAndPost` dentro del bloque `finally`. Además debemos siempre dibujar de forma sincronizada con el objeto `SurfaceHolder`, para así evitar problemas de concurrencia en el acceso a su lienzo.



Ejecuta la aplicación



Cambia la orientación del dispositivo varias veces. Puede que se rompa! El objeto canvas es null:

Un truco muy utilizado en muchos videojuegos es permitir solo una determinada orientación ("landscape" normalmente). Puede fijarse en la actividad de dos maneras diferentes:

- En Manifest: **RECOMENDADA POR SER LA MÁS "CORTA"**

```
<activity android:name=".MainActivity"
    android:screenOrientation="landscape"
```

- Por código java: En el método onCreate() y antes de poner el contenido de la vista:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
    getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
WindowManager.LayoutParams.FLAG_FULLSCREEN);
    setContentView(new MiVista6(this));
}
```

Observa que ya de paso, le hemos indicado que ocupe toda la pantalla. Pero esta solución nos obliga a declarar en el Manifest que están permitidos los cambios:

```
<activity android:name=".MainActivity"
    android:configChanges="screenSize|orientation"
```

Vuelve a ejecutar, gira y no se rompe.

Pero intenta abandonar la aplicación (◁ o ○ o □).
En un emulador ni nos deja siquiera muchas veces (el hilo está a su trabajo "dale que dale") y cuando nos deja, se rompe la aplicación y en un dispositivo real siempre se rompe.
Solución: Debemos jugar con los estados del hilo de la siguiente manera:

a) En la actividad, cambiaremos el estado en el método onPause()

```
public class MainActivity extends AppCompatActivity {

    private MiVista6 miVista6;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
WindowManager.LayoutParams.FLAG_FULLSCREEN);
        miVista6=new MiVista6(this);
        setContentView(miVista6);
    }

    @Override
    protected void onPause() {
        super.onPause();
        miVista6.hiloPintor.debeEjecutarse=false;
    }

}
```

Obviamente al reanudar la aplicación, el hilo volverá a crearse. Si el dibujo hubiese estado en movimiento y nos hubiese interesado guardar su posición, deberíamos haber guardado esos datos en savedinstanceState o en preferencias y utilizar también el método onResume() para recogerlos.

Habrás observado que en el emulador (ya de por sí lento) todo es mucho más lento. El hilo tal y como está diseñado, está continuamente pintando el balón aunque nuestro ojo es incapaz de notarlo. Una manera eficiente es fijar una pausa en el método run() del hilo.
Dependiendo de la complejidad de los movimientos/de los fotogramas por segundo deseados el tiempo que debe dormir el hilo varía.
En nuestro caso deseamos unos 70 fotogramas por segundo por lo que nuestra clase HiloPintor2 queda:

```
public class HiloPintor2 extends Thread {
    private final MiVista7 miVista7;
    private final SurfaceHolder holder;
    boolean debeEjecutarse;

    HiloPintor2(SurfaceHolder holder, MiVista7 miVista7) {
        this.holder=holder;
        this.miVista7=miVista7;
    }

    @Override
    public void run() {
        super.run();
        while (debeEjecutarse) {
            long antesDePintar=System.nanoTime();
            Canvas canvas = null;
            try {
                canvas = holder.lockCanvas(null);
                synchronized (holder) {
                    // Dibujar los gráficos
                    miVista7.pintarBalon(canvas);
                }
            } finally {
                if (canvas != null) {
                    holder.unlockCanvasAndPost(canvas);
                }
            }
```

```
//DORMIR HILO para mantener el "juego" coherente
            //Comienza con el tiempo de retraso especificado (en milisegundos) y luego
resta de
            //tiempo real que tardó en actualizar y renderizar el juego. Esto permite que
nuestro juego se procese suavemente.
            //para renderización eficiente tiempo de proceso entre dibujos consecutivos (in
milliseconds)
            long retraso = 70;
            long despuesDePintar=System.nanoTime();
            long tiempoSleep = retraso - ((despuesDePintar - antesDePintar) / 1000000L);
            try {
                if (tiempoSleep > 0) {
                    sleep(tiempoSleep);
                }
            } catch (InterruptedException ignored) {
            }
        }
    }
}
```

Gráficos dinámicos

Queremos que nuestro balón se mueva por la pantalla, no esté siempre en las coordenadas 30,30 del balón estático.

Debemos por tanto fijar una posicionX y una posicionY, sin olvidarnos que el balón no puede "salir" de la pantalla, por tanto debemos conocer el ancho y el alto de la pantalla y de la vista. El desplazamiento tanto en el eje "x" como en el eje "y" será un número aleatorio.

La clase Vista8 queda:

```
public class MiVista8 extends SurfaceView implements SurfaceHolder.Callback {
    SurfaceHolder holder;
    HiloPintor3 hiloPintor3;
    private Bitmap balon;
    private  int anchoBalon;
    private  int altoBalon;
    private  int anchoVista;
    private  int altoVista;
    private int pos_x;
    private int pos_y;
    private int desp_x;
    private int desp_y;

    public MiVista8(Context context) {
        super(context);
        holder=getHolder();
        holder.addCallback(this);
        balon = BitmapFactory.decodeResource(getResources(), R.drawable.ball);
        anchoBalon = balon.getWidth();
        altoBalon = balon.getHeight();
        anchoVista = getWidth();
        altoVista = getHeight();
        pos_x = anchoBalon;
        pos_y = altoBalon;
        Random random = new Random();
        desp_x = (int) (random.nextDouble() * anchoBalon);
        desp_y = (int) (random.nextDouble() * altoBalon);
    }

    @Override
    public void surfaceCreated(SurfaceHolder holder) {
        hiloPintor3 = new HiloPintor3(holder, this);
        hiloPintor3.start();
    }

    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {
        anchoVista = width;
        altoVista = height;
    }

    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        try {
            hiloPintor3.ejecutandose=false;
            hiloPintor3.join();
        } catch (InterruptedException ignored) {
        }
```

```
    }

    @Override
    public void onDraw(Canvas canvas) {
    }

    public void pintarBalon(Canvas canvas) {
        canvas.drawColor(Color.GREEN);
        // comprobamos que no se salga de los limites
        if ((pos_x + desp_x + (anchoBalon / 2) > anchoVista) ||
                (pos_x + desp_x - (anchoBalon / 2) < 0)) {
            desp_x = -desp_x;
        }
        if ((pos_y + desp_y + (altoBalon / 2) > altoVista)
                || (pos_y + desp_y - (altoBalon / 2) < 0)) {
            desp_y = -desp_y;
        }
        // actualizamos la posicion x e y
        pos_x += desp_x;
        pos_y += desp_y;
        // pintamos la bola en su posicion
        canvas.drawBitmap(balon, pos_x - (anchoBalon / 2), pos_y - (altoBalon / 2), null);
    }
}
```

¿Por qué pintamos cada vez el fondo verde?

Para evitar que se vean los "rastros" que van dejando las distintas posiciones de los balones:

> **Note:** On each pass you retrieve the Canvas from the SurfaceHolder, the previous state of the Canvas will be retained. In order to properly animate your graphics, you must re-paint the entire surface. For example, you can clear the previous state of the Canvas by filling in a color with `drawColor()` or setting a background image with `drawBitmap()`. Otherwise, you will see traces of the drawings you previously performed.

Trabajando con varios objetos dibujables

Hasta ahora en nuestros ejemplos solo había un objeto dibujable (el balón), pero en la mayoría de los casos habrá más de uno.

Si el comportamiento de todos estos elementos es muy similar, con el fin de reutilizar el código y mejorar su comprensión, se debe crear una clase que represente un gráfico a desplazar por pantalla.

```
public class Grafico {
    private Bitmap miBitmap;
    private int pos_x, pos_y;
    private int desp_x, desp_y;
    private int ancho, alto;
    private MiVista view;

    public Grafico(MiVista view, Bitmap miBitmap) {
        this.view = view;
        this.miBitmap = miBitmap;
        ancho = miBitmap.getWidth();
        alto = miBitmap.getHeight();
    }

    public Bitmap getMiBitmap() {
        return miBitmap;
    }
    public void setMiBitmap(Bitmap miBitmap) {
        this.miBitmap = miBitmap;
    }
    public int getPos_x() {
        return pos_x;
    }
    public void setPos_x(int pos_x) {
        this.pos_x = pos_x;
    }
    public int getPos_y() {
        return pos_y;
    }
    public void setPos_y(int pos_y) {
        this.pos_y = pos_y;
    }
    public int getDesp_x() {
        return desp_x;
    }
    public void setDesp_x(int desp_x) {
        this.desp_x = desp_x;
    }
```

```
    public int getDesp_y() {
        return desp_y;
    }
    public void setDesp_y(int desp_y) {
        this.desp_y = desp_y;
    }
    public int getAncho() {
        return ancho;
    }
    public void setAncho(int ancho) {
        this.ancho = ancho;
    }
    public int getAlto() {
        return alto;
    }
    public void setAlto(int alto) {
        this.alto = alto;
    }
    public MiVista getView() {
        return view;
    }
    public void setView(MiVista view) {
        this.view = view;
    }
}
```

Actualización del contenido

Es posible que en un momento dado cambien los datos a mostrar y necesitemos actualizar el contenido que nuestro componente está dibujando en pantalla.
La clase View tiene dos métodos:

- cuando se llama a **invalidate()** en el contexto de un no-UIThread, se le dice al sistema que redibuje la ventana lo más pronto posible (ejecute el método onDraw()).
- La llamada a **postInvalidate()** tiene un efecto similar para un UIThread. En este caso se le indica al sistema que redibuje las ventanas asociadas a ese Thead en el próximo ciclo de refrescos.

Si en algún momento estamos dibujando objetos en el Canvas y por cualquier cosa necesitamos limpiarlo lo podemos hacer con la siguiente línea de código:

- `canvas.drawColor(0, Mode.CLEAR);`

Aceleración por hardware

Para aumentar el rendimiento de los gráficos se hace uso de la aceleración por hardware que está activada por defecto para aplicaciones construidas con API mínima 14.
Puede activarse/desactivarse a nivel de aplicación/actividad/ventana/vista. Información

Gráficos 3D

Lo realmente interesante es utilizar SurfaceView junto a **OpenGL**, para así poder mostrar gráficos 3D o escalados, rotaciones y otras transformaciones.
A partir de Android 1.5 se incluye la clase **GLSurfaceView**, que ya incluye la inicialización del contexto GL y nos evita tener que hacerlo manualmente. Esto simplificará bastante el uso de la librería pero aún así deberemos tener al menos unos conocimientos básicos de OpenGL.
O bien utilizar librerías (AndEngine, libgdx,…) que facilitan el trabajo.

Prácticas

P_103_Graficos_ArbitroBalon: El balón y el árbitro deben moverse por la pantalla independientemente: