

Gráficos avanzados

Índice

1 Elementos drawables.....	2
1.1 Imágenes.....	3
1.2 Imágenes nine-patch.....	4
1.3 Lista de estados.....	6
1.4 Animación por fotogramas.....	6
1.5 Definición programática.....	8
2 Componentes propios.....	8
2.1 Lienzo y pincel.....	9
2.2 Primitivas geométricas.....	11
2.3 Cadenas de texto.....	12
2.4 Imágenes.....	14
2.5 Drawables.....	14
2.6 Medición del componente.....	14
2.7 Atributos propios.....	15
2.8 Actualización del contenido.....	16
3 Gráficos 3D.....	17

Hasta este momento hemos visto cómo crear la interfaz de usuario de nuestra aplicación utilizando una serie de componentes predefinidos. Ahora vamos a ver cómo personalizar los componentes existentes o crear nuestros propios componentes.

Comenzaremos viendo cómo establecer el aspecto de los diferentes componentes de la interfaz de los que disponemos en Android, para así poder dar a nuestras aplicaciones un estilo gráfico propio. A continuación, veremos cómo crear componentes propios, en los que podamos tener un control absoluto sobre lo que se dibuja en pantalla. Por último, trataremos la forma de crear gráficos 3D y aplicaciones que necesiten una elevada tasa de refresco, como son los videojuegos.

1. Elementos drawables

Un *drawable* es un tipo de recurso que puede ser dibujado en pantalla. Podremos utilizarlos para especificar el aspecto que van a tener los diferentes componentes de la interfaz, o partes de éstos. Estos *drawables* podrán ser definidos en XML o de forma programática. Entre los diferentes tipos de *drawables* existentes encontramos:

- **Color:** Rellena el lienzo de un determinado color.
- **Gradiente:** Rellena el lienzo con un gradiente.
- **Forma** (*shape*): Se pueden definir una serie de primitivas geométricas básicas como *drawables*.
- **Imagen** (*bitmap*): Una imagen se comporta como *drawable*, ya que podrá ser dibujada y referenciada de la misma forma que el resto.
- **Nine-patch:** Tipo especial de imagen PNG que al ser escalada sólo se escala su parte central, pero no su marco.
- **Animación:** Define una animación por fotogramas, como veremos más adelante.
- **Capa** (*layer list*): Es un *drawable* que contiene otros *drawables*. Cada uno especificará la posición en la que se ubica dentro de la capa.
- **Estados** (*state list*): 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.
- **Niveles** (*level list*): Similar al anterior, pero en este caso cada *item* tiene asignado un valor numérico (nivel). Al establecer el nivel del *drawable* se mostrará el *item* cuyo nivel sea mayor o igual que el indicado.
- **Transición** (*transition*): Nos permite mostrar una transición de un *drawable* a otro mediante un fundido.
- **Inserción** (*inset*): Ubica un *drawable* dentro de otro, en la posición especificada.
- **Recorte** (*clip*): Realiza un recorte de un *drawable*.
- **Escala** (*scale*): Cambia el tamaño de un *drawable*.

Todos los *drawables* derivan de la clase `Drawable`. Esta nos permite que todos ellos puedan ser utilizados de la misma forma, independientemente del tipo del que se trate. Se

puede consultar la lista completa de *drawables* y su especificación en la siguiente dirección:

<http://developer.android.com/guide/topics/resources/drawable-resource.html>

Por ejemplo, vamos a definir un *drawable* que muestre un rectángulo rojo con borde azul, creando un fichero XML de nombre `rectangulo.xml` en el directorio `/res/drawable/`. El fichero puede tener el siguiente contenido:

```
<?xml version="1.0" encoding="utf-8"?>
<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>
```

Podremos hacer referencia a este rectángulo desde el código mediante `R.drawable.rectangulo` y mostrarlo en la interfaz asignándolo a un componente de alto nivel como por ejemplo `ImageView`, o bien hacer referencia a él desde un atributo del XML mediante `@drawable/rectangulo`. Por ejemplo, podríamos especificar este *drawable* en el atributo `android:background` de la etiqueta `Button` dentro de nuestro *layout*, para que así el botón pase a tener como aspecto una forma rectangular de color rojo y con borde azul. De la misma forma podríamos darle al botón el aspecto de cualquier otro tipo de *drawable* de los vistos anteriormente. A continuación vamos a ver con más detalle los tipos de *drawables* más interesantes.

Nota

Podemos definir diferentes variantes del directorio de *drawables*, para así definir diferentes versiones de los *drawables* para los distintos tipos de dispositivos, por ejemplo según su densidad, tamaño, o forma de pantalla. Esto se aplica a todos los directorios de recursos. Por ejemplo se pueden diferenciar los recursos también según el idioma del dispositivo, o la orientación de la pantalla, lo cual se puede aprovechar para definir *layouts* diferentes. Para más información sobre los tipos de variantes disponibles consultar <http://developer.android.com/guide/topics/resources/providing-resources.html#AlternativeResources>

1.1. Imágenes

Las imágenes que introduzcamos en los directorios de recursos de tipo *drawable* (`/res/drawable/`) podrán ser tratadas igual que cualquier otro tipo de *drawable*. Por ejemplo, si introducimos en dicho directorio una imagen `titulo.png`, podremos hacer referencia a ella en los atributos de los XML mediante `@drawable/titulo` (no se pone la extensión), o bien desde el código mediante `R.drawable.titulo`.

Las imágenes se encapsulan en la clase `Bitmap`. Los *bitmaps* pueden ser mutables o inmutables, según si se nos permite modificar el valor de sus píxeles o no respectivamente.

Si el *bitmap* se crea a partir de un *array* de píxeles, de un recurso con la imagen, o de otro

bitmap, tendremos un *bitmap* inmutable.

Si creamos el *bitmap* vacío, simplemente especificando su altura y su anchura, entonces será mutable (en este caso no tendría sentido que fuese inmutable ya que sería imposible darle contenido). También podemos conseguir un *bitmap* mutable haciendo una copia de un *bitmap* existente mediante el método `copy`, indicando que queremos que el *bitmap* resultante sea mutable.

Para crear un *bitmap* vacío, a partir de un *array* de pixels, o a partir de otro *bitmap*, tenemos una serie de métodos estáticos `createBitmap` dentro de la clase `Bitmap`.

Para crear un *bitmap* a partir de un fichero de imagen (GIF, JPEG, o PNG, siendo este último el formato recomendado) 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. Por ejemplo, si tenemos una imagen (`titulo.png`) en el directorio de *drawables* podemos leerla como `Bitmap` de la siguiente forma:

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

Al crear un *bitmap* a partir de otro, podremos realizar diferentes transformaciones (escalado, rotación, etc).

Una vez no se vaya a utilizar más el *bitmap*, es recomendable liberar la memoria que ocupa. Podemos hacer esto llamando a su método `recycle`.

1.2. Imágenes nine-patch

Como hemos comentado anteriormente, utilizaremos los *drawables* para especificar el aspecto que queremos que tengan los componentes de la interfaz. La forma más flexible de definir este aspecto es especificar una imagen propia. Sin embargo, encontramos el problema de que los componentes normalmente no tendrán siempre el mismo tamaño, sino que Android los "estirará" según su contenido y según los parámetros de *layout* especificados (es decir, si deben ajustarse a su contenido o llenar todo el espacio disponible). Esto es un problema, ya que si siempre especificamos la misma imagen como aspecto para estos componentes, al estirla veremos que ésta se deforma, dando un aspecto terrible a nuestra aplicación, como podemos ver a continuación:

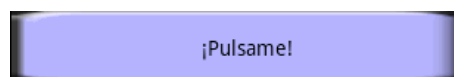
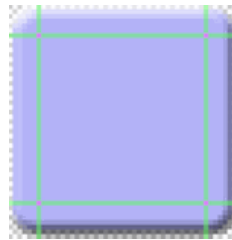


Imagen estirada

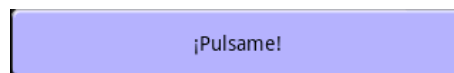
Sin embargo, tenemos un tipo especial de imágenes PNG llamadas *nine-patch* (llevan extensión `.9.png`), que nos permitirán evitar este problema. Normalmente la parte central de nuestros componentes es homogénea, por lo que no pasa nada si se estira. Sin

embargo, los bordes si que contienen un mayor número de detalles, que no deberían ser deformados, especialmente las esquinas. Las imágenes *nine-patch* se dividen en 9 regiones: la parte central, que puede ser escalada en cualquier dirección, las esquinas, que nunca pueden escaladas, y los bordes, que sólo pueden ser escalados en su misma dirección (horizontal o vertical). A continuación vemos un ejemplo de dicha división:



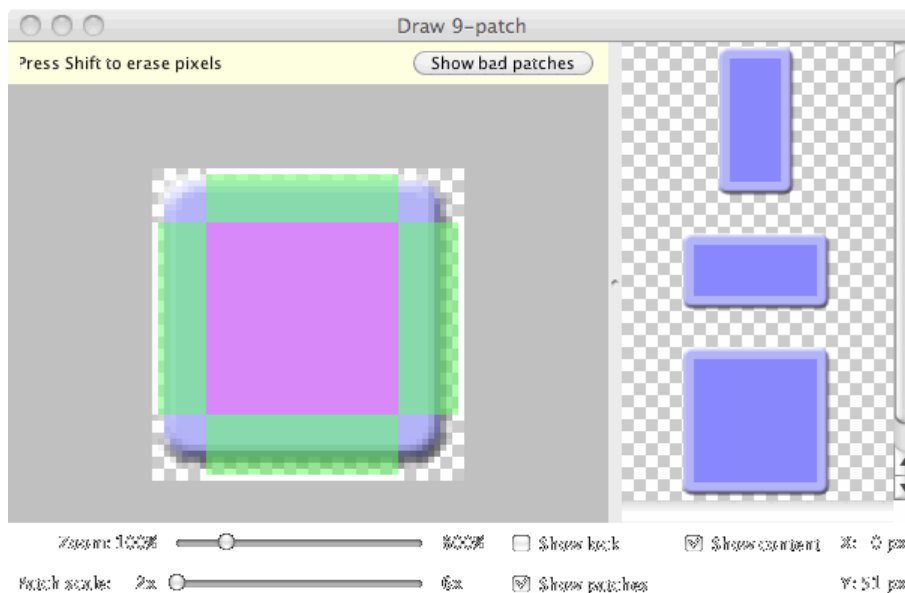
Parches de la imagen

Si ponemos una imagen de este tipo como *drawable* de fondo para un botón, veremos que siempre se mostrará con el aspecto correcto, independientemente de su contenido:



Aplicación de nine-patch a un botón

Podemos crear este tipo de imágenes con la herramienta `draw9patch` que podemos encontrar en el subdirectorio `tools` del SDK de Android. Lo único que necesitaremos es arrastrar el PNG que queramos tratar como *nine-patch*, y añadir una serie de píxeles en el marco de la imagen para marcar las regiones:



Herramienta draw9patch

La fila de píxeles superior y la columna izquierda indican las zonas de la imagen que son flexibles y que se pueden ampliar si es necesario repitiendo su contenido. En el caso de la fila superior, indica que se pueden estirar en la horizontal, mientras que los del lateral izquierdo corresponden a la vertical.

Opcionalmente podemos especificar en la fila inferior y en la columna derecha la zona que utilizaremos como contenido. Por ejemplo, si utilizamos la imagen como marco de un botón, esta será la zona donde se ubicará el texto que pongamos en el botón. Marcando la casilla *Show content* veremos en el lateral derecho de la herramienta una previsualización de la zona de contenido.

1.3. Lista de estados

Siguiendo con el ejemplo del botón, encontramos ahora un nuevo problema. Los botones no deben tener siempre el mismo aspecto de fondo, normalmente cambiarán de aspecto cuando están pulsados o seleccionados, sin embargo sólo tenemos la posibilidad de especificar un único *drawable* como fondo. Para poder personalizar el aspecto de todos los estados en los que se encuentra el botón tenemos un tipo de *drawable* llamado *state list drawable*. Se define en XML, y nos permitirá especificar un *drawable* diferente para cada estado en el que se puedan encontrar los componentes de la interfaz, de forma que en cada momento el componente mostrará el aspecto correspondiente a su estado actual.

Por ejemplo, podemos especificar los estados de un botón (no seleccionado, seleccionado, y pulsado) de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:state_pressed="true"
        android:drawable="@drawable/boton_pressed" /> <!-- presionado
-->
  <item android:state_focused="true"
        android:drawable="@drawable/boton_selected" /> <!-- seleccionado
-->
  <item android:drawable="@drawable/boton_normal" /> <!-- no
seleccionado -->
</selector>
```

Los *drawables* especificados para cada estado pueden ser de cualquier tipo (por ejemplo imágenes, *nine-patch*, o formas definidas en XML).

Un *drawable* similar es el de tipo *level list*, pero en este caso los diferentes posibles *drawables* a mostrar se especifican para un rango de valores numéricos. ¿Para qué tipos de componentes de la interfaz podría resultar esto de utilidad?

1.4. Animación por fotogramas

Este tipo de *drawable* nos permite definir una animación a partir de diferentes fotogramas, que deberemos especificar también como *drawables*, además del tiempo en milisegundos que durará el fotograma. Se definen en XML de la siguiente forma:

```
<animation-list
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="false">
    <item android:drawable="@drawable/spr0" android:duration="50" />
    <item android:drawable="@drawable/spr1" android:duration="50" />
    <item android:drawable="@drawable/spr2" android:duration="50" />
</animation-list>
```

Además, la propiedad *one shot* nos indica si la animación se va a reproducir sólo una vez o en bucle infinito. Al ponerla como *false* especificamos que se reproduzca de forma continuada.

Desde el código, podremos obtener la animación de la siguiente forma, considerando que la hemos guardado en un fichero `animacion.xml`:

```
AnimationDrawable animFotogramas =
    getResources().getDrawable(R.drawable.animacion);
```

De forma alternativa, podríamos haberla definido de forma programática de la siguiente forma:

```
BitmapDrawable f1 = (BitmapDrawable)getResources()
    .getDrawable(R.drawable.sprite0);
BitmapDrawable f2 = (BitmapDrawable)getResources()
    .getDrawable(R.drawable.sprite1);
BitmapDrawable f3 = (BitmapDrawable)getResources()
    .getDrawable(R.drawable.sprite2);
AnimationDrawable animFotogramas = new AnimationDrawable();

animFotogramas.addFrame(f1, 50);
animFotogramas.addFrame(f2, 50);
animFotogramas.addFrame(f3, 50);

animFotogramas.setOneShot(false);
```

Nota

La diferencia entre `Bitmap` y `BitmapDrawable` reside en que en el primer caso simplemente tenemos una imagen, mientras que en el segundo lo que tenemos es un *drawable* que encapsula una imagen, es decir, se le podrá proporcionar a cualquier componente que acepte *drawables* en general como entrada, y concretamente lo que dibujará será la imagen (`Bitmap`) que contiene.

Para que comience la reproducción deberemos llamar al método `start` de la animación:

```
animFotogramas.start();
```

De la misma forma, podemos detenerla con el método `stop`:

```
animFotogramas.stop();
```

Importante

El método `start` no puede ser llamado desde el método `onCreate` de nuestra actividad, ya que en ese momento el *drawable* todavía no está vinculado a la vista. Si lo que queremos es que se ponga en marcha nada más cargarse la actividad, el lugar idóneo para invocarlo es el evento `onWindowFocusChanged`. Lo recomendable será llamar a `start` cuando obtengamos el foco, y a `stop` cuando lo perdamos.

1.5. Definición programática

Vamos a suponer que tenemos un `ImageView` con identificador `visor` y un `drawable` de nombre `rectangulo`. Normalmente especificaremos directamente en el XML el *drawable* que queremos mostrar en el `ImageView`. Para ello deberemos añadir el atributo `android:src = "@drawable/rectangulo"` en la definición del `ImageView`.

Podremos también obtener una referencia a dicha vista y mostrar en ella nuestro rectángulo especificando el identificador del *drawable* de la siguiente forma:

```
ImageView visor = (ImageView)findViewById(R.id.visor);
visor.setImageResource(R.drawable.rectangulo);
```

Otra alternativa para mostrarlo es obtener primero el objeto `Drawable` y posteriormente incluirlo en el `ImageView`:

```
Drawable rectangulo = this.getResources()
    .getDrawable(R.drawable.rectangulo);
visor.setImageDrawable(rectangulo);
```

Estas primitivas básicas también se pueden crear directamente de forma programática. En el paquete `android.graphics.drawable.shape` podemos encontrar clases que encapsulan diferentes formas geométricas. Podríamos crear el rectángulo de la siguiente forma:

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

2. Componentes propios

Si no hay ningún componente predefinido que se adapte a nuestras necesidades, podemos crear un nuevo tipo de vista (`View`) en la que especificaremos exactamente qué es lo que queremos dibujar en la pantalla. El primer paso consistirá en crear una subclase de `View` en la que sobrescribiremos el método `onDraw`, que es el que define la forma en la que se dibuja el componente.

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

    @Override
    protected void onDraw(Canvas canvas) {
        // TODO Definir como dibujar el componente
    }
}
```


2.1. Lienzo y pincel

El método `onDraw` recibe como parámetro el lienzo (`Canvas`) en el que deberemos dibujar. En este lienzo podremos dibujar diferentes tipos de elementos, como primitivas geométricas, texto e imágenes.

Importante

No confundir el `Canvas` de Android con el `Canvas` que existe en Java ME/SE. En Java ME/SE el `Canvas` es un componente de la interfaz, que equivaldría a `View` en Android, mientras que el `Canvas` de Android es más parecido al objeto `Graphics` de Java ME/SE, que encapsula el contexto gráfico (o lienzo) del área en la que vamos a dibujar.

Además, para dibujar determinados tipos de elementos deberemos especificar también el tipo de pincel a utilizar (`Paint`), en el que especificaremos una serie de atributos como su color, grosor, etc.

Por ejemplo, para especificar un pincel que pinte en color rojo escribiremos lo siguiente:

```
Paint p = new Paint();
p.setColor(Color.RED);
```

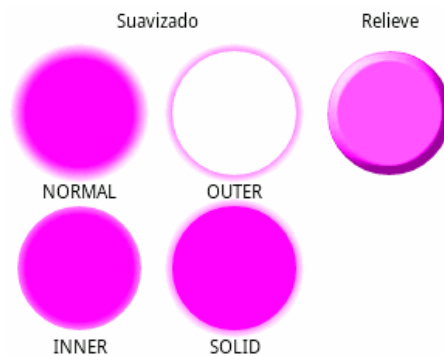
Las propiedades que podemos establecer en el pincel son:

- **Color plano:** Con `setARGB` o `setColor` se puede especificar el código ARGB del color o bien utilizar constantes con colores predefinidos de la clase `Color`.
- **Gradientes y shaders:** Se pueden rellenar las figuras utilizando *shaders* de gradiente o de bitmap. Para utilizar un *shader* tenemos el método `setShader`, y tenemos varios *shaders* disponibles, como distintos *shaders* de gradiente (`LinearShader`, `RadialShader`, `SweepShader`), `BitmapShader` para rellenar utilizando un mapa de bits como patrón, y `ComposeShader` para combinar dos *shaders* distintos.



Tipos de gradiente

- **Máscaras:** Nos sirven para aplicar un suavizado a los gráficos (`BlurMaskFilter`) o dar efecto de relieve (`EmbossMaskFilter`). Se aplican con `setMaskFilter`.



Mascaras de suavizado y relieve

- **Sombras:** Podemos crear efectos de sombra con `setShadowLayer`.
- **Filtros de color:** Aplica un filtro de color a los gráficos dibujados, alterando así su color original. Se aplica con `setColorFilter`.
- **Estilo de la figura:** Se puede especificar con `setStyle` que se dibuje sólo el trazo, sólo el relleno, o ambos.



Estilos de pincel

- **Estilo del trazo:** Podemos especificar el grosor del trazo (`setStrokeWidth`), el tipo de línea (`setPathEffect`), la forma de las uniones en las polilíneas (redondeada/ROUND, a inglete/MITER, o biselada/BEVEL, con `setStrokeJoin`), o la forma de las terminaciones (cuadrada/SQUARE, redonda/ROUND o recortada/BUTT, con `setStrokeCap`).



Tipos de trazo y límites

- **Antialiasing:** Podemos aplicar *antialiasing* con `setAntiAlias` a los gráficos para evitar el efecto *sierra*.
- **Dithering:** Si el dispositivo no puede mostrar los 16 millones de colores, en caso de haber un gradiente, para que el cambio de color no sea brusco, con esta opción (`setDither`) se mezclan pixels de diferentes colores para dar la sensación de que la transición entre colores es más suave.



Efecto dithering

- **Modo de transferencia:** Con `setXferMode` podemos cambiar el modo de transferencia con el que se dibuja. Por ejemplo, podemos hacer que sólo se dibuje encima de pixels que tengan un determinado color.
- **Estilo del texto:** Podemos también especificar el tipo de fuente a utilizar y sus atributos. Lo veremos con más detalle más adelante.

Una vez establecido el tipo de pincel, podremos utilizarlo para dibujar diferentes elementos en el lienzo, utilizando métodos de la clase `Canvas`.

En el lienzo podremos también establecer algunas propiedades, como el área de recorte (`clipRect`), que en este caso no tiene porque ser rectangular (`clipPath`), o transformaciones geométricas (`translate`, `scale`, `rotate`, `skew`, o `setMatrix`). Si queremos cambiar temporalmente estas propiedades, y luego volver a dejar el lienzo como estaba originalmente, podemos utilizar los métodos `save` y `restore`.

Vamos a ver a continuación como utilizar los métodos del lienzo para dibujar distintos tipos de primitivas geométricas.

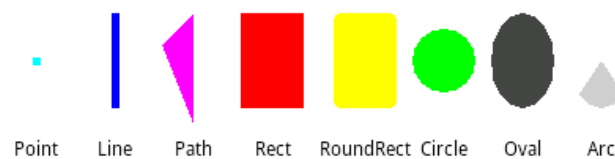
2.2. Primitivas geométricas

En la clase `Canvas` encontramos métodos para dibujar diferentes tipos de primitivas geométricas. Estos tipos son:

- **Puntos:** Con `drawPoint` podemos dibujar un punto en las coordenadas X, Y especificadas.
- **Líneas:** Con `drawLine` dibujamos una línea recta desde un punto de origen hasta un punto destino.
- **Polilíneas:** Podemos dibujar una polilínea mediante `drawPath`. La polilínea se especificará mediante un objeto de clase `Path`, en el que iremos añadiendo los segmentos de los que se compone. Este objeto `Path` representa un contorno, que podemos crear no sólo a partir de segmentos rectos, sino también de curvas cuadráticas y cúbicas.
- **Rectángulos:** Con `drawRect` podemos dibujar un rectángulo con los límites superior, inferior, izquierdo y derecho especificados.
- **Rectángulos con bordes redondeados:** Es también posible dibujar un rectángulo con esquinas redondeadas con `drawRoundRect`. En este caso deberemos especificar también el radio de las esquinas.
- **Círculos:** Con `drawCircle` podemos dibujar un círculo dando su centro y su radio.
- **Óvalos:** Los óvalos son un caso más general que el del círculo, y los crearemos con

`drawOval` proporcionando el rectángulo que lo engloba.

- **Arcos:** También podemos dibujar arcos, que consisten en un segmento del contorno de un óvalo. Se crean con `drawArc`, proporcionando, además de los mismos datos que en el caso del óvalo, los ángulos que limitan el arco.
- **Todo el lienzo:** Podemos también especificar que todo el lienzo se rellene de un color determinado con `drawColor` o `drawARGB`. Esto resulta útil para limpiar el fondo antes de empezar a dibujar.



Tipos de primitivas geométricas

A continuación mostramos un ejemplo de cómo podríamos dibujar una polilínea y un rectángulo:

```
Paint paint = new Paint();
paint.setStyle(Style.FILL);
paint.setStrokeWidth(5);
paint.setColor(Color.BLUE);

Path path = new Path();
path.moveTo(50, 130);
path.lineTo(50, 60);
path.lineTo(30, 80);

canvas.drawPath(path, paint);

canvas.drawRect(new RectF(180, 20, 220, 80), paint);
```

2.3. Cadenas de texto

Para dibujar texto podemos utilizar el método `drawText`. De forma alternativa, se puede utilizar `drawPosText` para mostrar texto especificando una por una la posición de cada carácter, y `drawTextOnPath` para dibujar el texto a lo largo de un contorno (`Path`).

Para especificar el tipo de fuente y sus atributos, utilizaremos las propiedades del objeto `Paint`. Las propiedades que podemos especificar del texto son:

- **Fuente:** Con `setTypeface` podemos especificar la fuente, que puede ser alguna de las fuentes predefinidas (*Sans Serif*, *Serif*, *Monoespaciada*), o bien una fuente propia a partir de un fichero de fuente. También podemos especificar si el estilo de la fuente será normal, cursiva, negrita, o negrita cursiva.
- **Tamaño:** Podemos establecer el tamaño del texto con `setTextSize`.
- **Anchura:** Con `setTextScaleX` podemos modificar la anchura del texto sin alterar la altura.
- **Inclinación:** Con `setTextSkewX` podemos aplicar un efecto de desencajado al texto, pudiendo establecer la inclinación que tendrán los caracteres.

- **Subrayado:** Con `setUnderlineText` podemos activar o desactivar el subrayado.
- **Tachado:** Con `setStrikeThruText` podemos activar o desactivar el efecto de tachado.
- **Negrita falsa:** Con `setFakeBoldText` podemos darle al texto un efecto de *negrita*, aunque la fuente no sea de este tipo.
- **Alineación:** Con `setTextAlign` podemos especificar si el texto se alinea al centro, a la derecha, o a la izquierda.
- **Subpixel:** Se renderiza a nivel de subpixel. El texto se genera a una resolución mayor que la de la pantalla donde lo vamos a mostrar, y para cada pixel real se habrán generado varios pixels. Si aplicamos *antialiasing*, a la hora de mostrar el pixel real, se determinará un nivel de gris dependiendo de cuantos pixels ficticios estén activados. Se consigue un aspecto de texto más suavizado.
- **Texto lineal:** Muestra el texto con sus dimensiones reales de forma lineal, sin ajustar los tamaños de los caracteres a la cuadrícula de pixels de la pantalla.
- **Contorno del texto:** Aunque esto no es una propiedad del texto, el objeto `Paint` también nos permite obtener el contorno (`Path`) de un texto dado, para así poder aplicar al texto los mismos efectos que a cualquier otro contorno que dibujemos.

Normal	<u>Subrayado</u>
Normal lineal	<i>Inclinado</i>
Negrita falsa	Antialiasing
Tachado	Antialiasing subpixel

Efectos del texto

Con esto hemos visto como dibujar texto en pantalla, pero para poderlo ubicar de forma correcta es importante saber el tamaño en pixels del texto a mostrar. Vamos a ver ahora cómo obtener estas métricas.

Las métricas se obtendrán a partir del objeto `Paint` en el que hemos definido las propiedades de la fuente a utilizar. Mediante `getFontMetrics` podemos obtener una serie de métricas de la fuente actual, que nos dan las distancias recomendadas que debemos dejar entre diferentes líneas de texto:

- `ascent`: Distancia que asciende la fuente desde la línea de base. Para texto con espaciado sencillo es la distancia que se recomienda dejar por encima del texto. Se trata de un valor negativo.
- `descent`: Distancia que baja la fuente desde la línea de base. Para texto con espaciado sencillo es la distancia que se recomienda dejar por debajo del texto. Se trata de un valor positivo.
- `leading`: Distancia que se recomienda dejar entre dos líneas consecutivas de texto.
- `bottom`: Es la máxima distancia que puede bajar un símbolo desde la línea de base. Es un valor positivo.
- `top`: Es la máxima distancia que puede subir un símbolo desde la línea de base. Es un valor negativo.

Los anteriores valores son métricas generales de la fuente, pero muchas veces

necesitaremos saber la anchura de una determinada cadena de texto, que ya no sólo depende de la fuente sino también del texto. Tenemos una serie de métodos con los que obtener este tipo de información:

- `measureText`: Nos da la anchura en pixels de una cadena de texto con la fuente actual.
- `breakText`: Método útil para cortar el texto de forma que no se salga de los márgenes de la pantalla. Se le proporciona la anchura máxima que puede tener la línea, y el método nos dice cuántos caracteres de la cadena proporcionada caben en dicha línea.
- `getTextWidths`: Nos da la anchura individual de cada carácter del texto proporcionado.
- `getTextBounds`: Nos devuelve un rectángulo con las dimensiones del texto, tanto anchura como altura.

2.4. Imágenes

Podemos también dibujar en nuestro lienzo imágenes que hayamos cargado como `Bitmap`. Esto se hará utilizando el método `drawBitmap`.

También podremos realizar transformaciones geométricas en la imagen al mostrarla en el lienzo con `drawBitmap`, e incluso podemos dibujar el *bitmap* sobre una malla poligonal con `drawBitmapMesh`

2.5. Drawables

También podemos dibujar objetos de tipo *drawable* en nuestro lienzo, esta vez mediante el método `draw` definido en la clase `Drawable`. Esto nos permitirá mostrar en nuestro componente cualquiera de los tipos disponibles de *drawables*, tanto definidos en XML como de forma programática.

2.6. Medición del componente

Al crear un nuevo componente, además de sobrescribir el método `onDraw`, es buena idea sobrescribir también el método `onMeasure`. Este método será invocado por el sistema cuando vaya a ubicarlo en el *layout*, para asignarle un tamaño. Para cada dimensión (altura y anchura), nos pasa dos parámetros:

- **Tamaño**: Tamaño en píxeles solicitado para la dimensión (altura o anchura).
- **Modo**: Puede ser `EXACTLY`, `AT_MOST`, o `UNSPECIFIED`. En el primer caso indica que el componente debe tener exactamente el tamaño solicitado, el segundo indica que como mucho puede tener ese tamaño, y el tercero nos da libertad para decidir el tamaño.

Antes de finalizar `onMeasure`, deberemos llamar obligatoriamente a `setMeasuredDimension(width, height)` proporcionando el tamaño que queramos que tenga nuestro componente. Una posible implementación sería la siguiente:

```

@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    int widthMode = MeasureSpec.getMode(widthMeasureSpec);
    int widthSize = MeasureSpec.getSize(widthMeasureSpec);
    int heightMode = MeasureSpec.getMode(heightMeasureSpec);
    int heightSize = MeasureSpec.getSize(heightMeasureSpec);

    int width = DEFAULT_SIZE;
    int height = DEFAULT_SIZE;

    switch(widthMode) {
        case MeasureSpec.EXACTLY:
            width = widthSize;
            break;
        case MeasureSpec.AT_MOST:
            if(width > widthSize) {
                width = widthSize;
            }
            break;
    }

    switch(heightMode) {
        case MeasureSpec.EXACTLY:
            height = heightSize;
            break;
        case MeasureSpec.AT_MOST:
            if(height > heightSize) {
                height = heightSize;
            }
            break;
    }

    this.setMeasuredDimension(width, height);
}

```

Podemos ver que tenemos unas dimensiones preferidas por defecto para nuestro componente. Si nos piden unas dimensiones exactas, ponemos esas dimensiones, pero si nos piden unas dimensiones como máximo, nos quedamos con el mínimo entre nuestra dimensión preferida y la que se ha especificado como límite máximo que puede tener.

2.7. Atributos propios

Si creamos un nuevo tipo de vista, es muy probable que necesitemos parametrizarla de alguna forma. Por ejemplo, si queremos dibujar una gráfica que nos muestre un porcentaje, necesitaremos proporcionar un valor numérico como porcentaje a mostrar. Si vamos a crear la vista siempre de forma programática esto no es ningún problema, ya que basta con incluir en nuestra clase un método que establezca dicha propiedad.

Sin embargo, si queremos que nuestro componente se pueda añadir desde el XML, será necesario poder pasarle dicho valor como atributo. Para ello en primer lugar debemos declarar los atributos propios en un fichero `/res/values/attrs.xml`:

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <declare-styleable name="Grafica">
        <attr name="percentage" format="integer"/>
    </declare-styleable>
</resources>

```

En el XML donde definimos el *layout*, podemos especificar nuestro componente utilizando como nombre de la etiqueta el nombre completo (incluyendo el paquete) de la clase donde hemos definido la vista:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app=
"http://schemas.android.com/apk/res/es.ua.jtech.grafica"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<es.ua.jtech.grafica.GraficaView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:percentage="60"
    />
</LinearLayout>
```

Podemos fijarnos en que para declarar el atributo propio hemos tenido que especificar el espacio de nombres en el que se encuentra. En dicho espacio de nombres deberemos especificar el paquete que hemos declarado en *AndroidManifest.xml* para la aplicación (en nuestro caso *es.ua.jtech.grafica*).

```
public GraficaView(Context context) {
    super(context);
}

public GraficaView(Context context, AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);
    this.init(attrs);
}

public GraficaView(Context context, AttributeSet attrs) {
    super(context, attrs);
    this.init(attrs);
}

private void init(AttributeSet attrs) {
    TypedArray ta = this.getContext().obtainStyledAttributes(attrs,
                                                                R.styleable.Grafica);
    this.percentage = ta.getInt(R.styleable.Grafica_percentage, 0);
}
```

En primer lugar podemos ver que debemos definir todos los posibles constructores de las vistas, ya que cuando se cree desde el XML se invocará uno de los que reciben la lista de atributos especificados. Una vez recibamos dicha lista de atributos, deberemos obtener el conjunto de atributos propios mediante *obtainStyledAttributes*, y posteriormente obtener los valores de cada atributo concreto dentro de dicho conjunto.

2.8. 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. Podemos forzar que se vuelva a dibujar llamando al método *invalidate* de nuestra vista (*View*).

Esto podemos utilizarlo también para crear animaciones. De hecho, para crear una animación simplemente deberemos cambiar el contenido del lienzo conforme pasa el tiempo. Una forma de hacer esto es simplemente cambiar mediante un hilo o temporizadores propiedades de los objetos de la escena (como sus posiciones), y forzar a que se vuelva a redibujar el contenido del lienzo cada cierto tiempo.

Sin embargo, si necesitamos contar con una elevada tasa de refresco, como por ejemplo en el caso de un videojuego, será recomendable utilizar una vista de tipo `SurfaceView` como veremos a continuación.

3. Gráficos 3D

Para mostrar gráficos 3D en Android contamos con OpenGL ES, un subconjunto de la librería gráfica OpenGL destinado a dispositivos móviles.

Hasta ahora hemos visto que para mostrar gráficos propios podíamos usar un componente que heredase de `View`. Estos componentes funcionan bien si no necesitamos realizar repintados continuos o mostrar gráficos 3D.

Sin embargo, en el caso de tener una aplicación con una gran carga gráfica, como puede ser un videojuego o una aplicación que muestre gráficos 3D, en lugar de `View` deberemos utilizar `SurfaceView`. Esta última clase nos proporciona una superficie en la que podemos dibujar desde un hilo en segundo plano, lo cual libera al hilo principal de la aplicación de la carga gráfica.

Vamos a ver en primer lugar cómo utilizar `SurfaceView`, y las diferencias existentes con `View`.

Para crear una vista con `SurfaceView` tendremos que crear una nueva subclase de dicha clase (en lugar de `View`). Pero en este caso no bastará con definir el método `onDraw`, ahora deberemos crearnos un hilo independiente y proporcionarle la superficie en la que dibujar (`SurfaceHolder`). Además, en nuestra subclase de `SurfaceView` también implementaremos la interfaz `SurfaceHolder.Callback` que nos permitirá estar al tanto de cuando la superficie se crea, cambia, o se destruye.

Cuando la superficie sea creada pondremos en marcha nuestro hilo de dibujado, y lo pararemos cuando la superficie sea destruida. A continuación mostramos un ejemplo de dicha clase:

```
public class VistaSurface extends SurfaceView
    implements SurfaceHolder.Callback {
    HiloDibujo hilo = null;

    public VistaSurface(Context context) {
        super(context);

        SurfaceHolder holder = this.getHolder();
        holder.addCallback(this);
    }
}
```

```

    public void surfaceChanged(SurfaceHolder holder, int format,
                               int width, int height) {
        // La superficie ha cambiado (formato o dimensiones)
    }

    public void surfaceCreated(SurfaceHolder holder) {
        hilo = new HiloDibujo(holder, this);
        hilo.start();
    }

    public void surfaceDestroyed(SurfaceHolder holder) {
        hilo.detener();
        try {
            hilo.join();
        } catch (InterruptedException e) { }
    }
}

```

Como vemos, la clase `SurfaceView` simplemente se encarga de obtener la superficie y poner en marcha o parar el hilo de dibujado. En este caso la acción estará realmente en el hilo, que es donde especificaremos la forma en la que se debe dibujar el componente. Vamos a ver a continuación cómo podríamos implementar dicho hilo:

```

class HiloDibujo extends Thread {
    SurfaceHolder holder;
    VistaSurface vista;
    boolean continuar = true;

    public HiloDibujo(SurfaceHolder holder, VistaSurface vista) {
        this.holder = holder;
        this.vista = vista;
        continuar = true;
    }

    public void detener() {
        continuar = false;
    }

    @Override
    public void run() {
        while (continuar) {
            Canvas c = null;
            try {
                c = holder.lockCanvas(null);
                synchronized (holder) {
                    // Dibujar aqui los graficos
                    c.drawColor(Color.BLUE);
                }
            } finally {
                if (c != null) {
                    holder.unlockCanvasAndPost(c);
                }
            }
        }
    }
}

```

Podemos ver que en el bucle principal de nuestro hilo obtendremos 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.

Para aplicaciones como videojuegos 2D sencillo un código como el anterior puede ser suficiente (la clase `View` sería demasiado lenta para un videojuego). Sin embargo, lo realmente interesante es utilizar `SurfaceView` junto a OpenGL, para así poder mostrar gráficos 3D, o escalados, rotaciones y otras transformaciones sobre superficies 2D de forma eficiente.

El estudio de la librería OpenGL queda fuera del ámbito de este curso. A continuación veremos un ejemplo de cómo utilizar OpenGL (concretamente OpenGL ES) vinculado a nuestra `SurfaceView`.

Realmente la implementación de nuestra clase que hereda de `SurfaceView` no cambiará, simplemente modificaremos nuestro hilo, que es quien realmente realiza el dibujado. Toda la inicialización de OpenGL deberá realizarse dentro de nuestro hilo (en el método `run`), ya que sólo se puede acceder a las operaciones de dicha librería desde el mismo hilo en el que se inicializó. En caso de que intentásemos acceder desde otro hilo obtendríamos un error indicando que no existe ningún contexto activo de OpenGL.

En este caso nuestro hilo podría contener el siguiente código:

```
public void run() {
    initEGL();
    initGL();

    Triangulo3D triangulo = new Triangulo3D();
    float angulo = 0.0f;

    while(continuar) {
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT |
                  GL10.GL_DEPTH_BUFFER_BIT);

        // Dibujar gráficos aquí
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();
        gl.glTranslatef(0, 0, -5.0f);
        gl.glRotatef(angulo, 0, 1, 0);

        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        triangulo.dibujar(gl);

        egl.eglSwapBuffers(display, surface);
        angulo += 1.0f;
    }
}
```

En primer lugar debemos inicializar la interfaz EGL, que hace de vínculo entre la plataforma nativa y la librería OpenGL:

```
EGL10 egl;
GL10 gl;
EGLDisplay display;
EGLSurface surface;
EGLContext contexto;
```

```

EGLConfig config;

private void initEGL() {
    egl = (EGL10)EGLContext.getEGL();
    display = egl.eglGetDisplay(EGL10.EGL_DEFAULT_DISPLAY);

    int [] version = new int[2];
    egl.eglInitialize(display, version);

    int [] atributos = new int[] {
        EGL10.EGL_RED_SIZE, 5,
        EGL10.EGL_GREEN_SIZE, 6,
        EGL10.EGL_BLUE_SIZE, 5,
        EGL10.EGL_DEPTH_SIZE, 16,
        EGL10.EGL_NONE
    };

    EGLConfig [] configs = new EGLConfig[1];
    int [] numConfigs = new int[1];
    egl.eglChooseConfig(display, atributos, configs,
        1, numConfigs);

    config = configs[0];
    surface = egl.eglCreateWindowSurface(display,
        config, holder, null);
    contexto = egl.eglCreateContext(display, config,
        EGL10.EGL_NO_CONTEXT, null);
    egl.eglMakeCurrent(display, surface, surface, contexto);

    gl = (GL10)contexto.getGL();
}

```

A continuación debemos proceder a la inicialización de la interfaz de la librería OpenGL:

```

private void initGL() {
    int width = vista.getWidth();
    int height = vista.getHeight();
    gl.glViewport(0, 0, width, height);
    gl.glMatrixMode(GL10.GL_PROJECTION);
    gl.glLoadIdentity();

    float aspecto = (float)width/height;
    GLU.gluPerspective(gl, 45.0f, aspecto, 1.0f, 30.0f);
    gl.glClearColor(0.5f, 0.5f, 0.5f, 1);
}

```

Una vez hecho esto, ya sólo nos queda ver cómo dibujar una malla 3D. Vamos a ver como ejemplo el dibujo de un triángulo:

```

public class Triangulo3D {

    FloatBuffer buffer;

    float[] vertices = {
        -1f, -1f, 0f,
        1f, -1f, 0f,
        0f, 1f, 0f };

    public Triangulo3D() {
        ByteBuffer bufferTemporal = ByteBuffer
            .allocateDirect(vertices.length*4);
        bufferTemporal.order(ByteOrder.nativeOrder());
        buffer = bufferTemporal.asFloatBuffer();
    }
}

```

```

        buffer.put(vertices);
        buffer.position(0);
    }

    public void dibujar(GL10 gl) {
        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, buffer);
        gl.glDrawArrays(GL10.GL_TRIANGLES, 0, 3);
    }
}

```

Para finalizar, es importante que cuando la superficie se destruya se haga una limpieza de los recursos utilizados por OpenGL:

```

private void cleanupGL() {
    egl.eglMakeCurrent(display, EGL10.EGL_NO_SURFACE,
        EGL10.EGL_NO_SURFACE, EGL10.EGL_NO_CONTEXT);
    egl.eglDestroySurface(display, surface);
    egl.eglDestroyContext(display, contexto);
    egl.eglTerminate(display);
}

```

Podemos llamar a este método cuando el hilo se detenga (debemos asegurarnos que se haya detenido llamando a `join` previamente).

A partir de Android 1.5 se incluye la clase `GLSurfaceView`, que ya incluye la inicialización del contexto GL y nos evita tener que hacer esto manualmente. Esto simplificará bastante el uso de la librería. Vamos a ver a continuación un ejemplo de como trabajar con dicha clase.

En este caso ya no será necesario crear una subclase de `GLSurfaceView`, ya que la inicialización y gestión del hilo de OpenGL siempre es igual. Lo único que nos interesará cambiar es lo que se muestra en la escena. Para ello deberemos crear una subclase de `GLSurfaceViewRenderer` que nos obliga a definir los siguientes métodos:

```

public class MiRenderer implements GLSurfaceView.Renderer {

    Triangulo3D triangulo;
    float angulo;

    public MiRenderer() {
        triangulo = new Triangulo3D();
        angulo = 0;
    }

    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    }

    public void onSurfaceChanged(GL10 gl, int w, int h) {
        // Al cambiar el tamaño cambia la proyección
        float aspecto = (float)w/h;
        gl.glViewport(0, 0, w, h);

        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        GLU.gluPerspective(gl, 45.0f, aspecto, 1.0f, 30.0f);
    }

    public void onDrawFrame(GL10 gl) {
        gl.glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    }
}

```

```

        gl.glClear(GL10.GL_COLOR_BUFFER_BIT |
                   GL10.GL_DEPTH_BUFFER_BIT);

        // Dibujar gráficos aquí
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();
        gl.glTranslatef(0, 0, -5.0f);
        gl.glRotatef(angulo, 0, 1, 0);

        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        triangulo.dibujar(gl);

        angulo += 1.0f;
    }
}

```

Podemos observar que será el método `onDrawFrame` en el que deberemos escribir el código para mostrar los gráficos. Con hacer esto será suficiente, y no tendremos que encargarnos de crear el hilo ni de inicializar ni destruir el contexto.

Para mostrar estos gráficos en la vista deberemos proporcionar nuestro *renderer* al objeto `GLSurfaceView`:

```

vista = new GLSurfaceView(this);
vista.setRenderer(new MiRenderer());
setContentView(vista);

```

Por último, será importante transmitir los eventos `onPause` y `onResume` de nuestra actividad a la vista de OpenGL, para así liberar a la aplicación de la carga gráfica cuando permanezca en segundo plano. El código completo de la actividad quedaría como se muestra a continuación:

```

public class MiActividad extends Activity {
    GLSurfaceView vista;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        vista = new GLSurfaceView(this);
        vista.setRenderer(new MiRenderer());
        setContentView(vista);
    }

    @Override
    protected void onPause() {
        super.onPause();
        vista.onPause();
    }

    @Override
    protected void onResume() {
        super.onResume();
        vista.onResume();
    }
}

```

Para utilizar esta vista deberemos tener al menos unos conocimientos básicos de OpenGL. Si lo que queremos es iniciarnos en el desarrollo de videojuegos para Android, contamos con diferentes librerías y *frameworks* que nos van a facilitar bastante el trabajo.

Destacamos las siguientes:

- *libgdx*: Es la más rápida y flexible, aunque requiere tener algunos conocimientos de OpenGL. <http://www.badlogicgames.com>
- *AndEngine*: Es muy sencilla de utilizar. Proporciona gran cantidad de facilidades para crear videojuegos 2D. <http://www.andengine.org/>

