Canvas and Drawables

In this document

See also

- OpenGL with the Framework APIs
- RenderScript

The Android framework provides a set of two-dimensional drawing APIs that allow you to render your own custom graphics onto a canvas or to modify existing views to customize their look and feel. You typically draw 2-D graphics in one of the following ways:

a. Draw your graphics or animations on a `View` object in your layout. By using this option, the system's rendering pipeline handles your graphics—it's your responsibility to define the graphics inside the view.

b. Draw your graphics in a `Canvas` object. To use this option, you pass your canvas to the appropriate class' `onDraw(Canvas)` method. You can also use the drawing methods in `Canvas`. This option also puts you in control of any animation.

Drawing to a view is a good choice when you want to draw simple graphics that don't need to change dynamically and aren't part of a performance-intensive app, such as a game. For example, you should draw your graphics into a view when you want to display a static graphic or predefined animation, within an otherwise static app. For more information, read Drawables.

Drawing to a canvas is better when your app needs to regularly redraw itself. Apps, such as video games, should draw to the canvas on their own. However, there's more than one way to do this:

a. In your app's main thread, wherein you create a custom view component in your layout, call `invalidate()` and then handle the `onDraw(Canvas)` callback.

b. In a worker thread that manages a `SurfaceView`, use drawing methods of the canvas. You don't need to call `invalidate()`.

Draw with a canvas

You can meet the requirements of an app that needs specialized drawing and/or control of the graphics animation by drawing to a canvas, which is represented by the `Canvas` class. A canvas serves as a pretense, or interface, to the actual surface upon which your graphics are drawn—you can perform your *draw* operations to the canvas. Via the canvas, your app draws to the underlying `Bitmap` object, which is placed into the window.

If you're drawing within the `onDraw(Canvas)` callback, the canvas is already provided and you only need to place your drawing calls upon it. If you're using a `SurfaceView` object, you can acquire a canvas from `lockCanvas()`. Both of these scenarios are discussed in the following sections.

If you need to create a new `Canvas` object, then you must define the underlying `Bitmap` object that is required to place the drawing into a window. The following code example shows how to set up a new canvas from a bitmap:

```
Bitmap b = Bitmap.createBitmap(100, 100, Bitmap.Config.ARGB_8888);
Canvas c = new Canvas(b);
```

It's possible to use the bitmap in a different canvas by using one of the `drawBitmap()` methods. However, we recommend that you use a canvas provided by the `onDraw(Canvas)` callback or the `lockCanvas()` method. For more information, see Drawing on a View and Drawing on a SurfaceView.

The `Canvas` class has its own set of drawing methods, including `drawBitmap()`, `drawRect()`, `drawText()`, and many more. Other classes that you might use also have `draw()` methods. For example, you probably have some `Drawable` objects that you want to put on the canvas. The `Drawable` class has its own `draw(Canvas)` method that takes your canvas as an argument.

Drawing on a view

If your app doesn't require a significant amount of processing or a high frame rate (for example a chess game, a snake game, or another slowly animated app), then you should consider creating a custom view and drawing with a canvas in the `View.onDraw(Canvas)` callback. The most convenient aspect of this is that the Android framework provides a predefined canvas that can perform your drawing operations.

To start, create a subclass of `View` and implement the `onDraw(Canvas)` callback, which the Android framework calls to draw the view. Then perform the draw operations through the `Canvas` object, which is provided by the framework.

The Android framework only calls `onDraw(Canvas)` when necessary. When it's time to redraw your app, you must invalidate the view by calling `invalidate()`. Calling `invalidate()` indicates that you'd like your view to be drawn. Android then calls your view's `onDraw(Canvas)` method, though the call isn't guaranteed to be instantaneous.

Inside your view's `onDraw(Canvas)` method, call drawing methods on the canvas or on other classes' methods that can take your canvas as an argument. Once your `onDraw()` finishes, the Android framework uses your canvas to draw a bitmap that is handled by the system.

> **Note:** To invalidate a view from a thread other than the app's main thread, you must call `postInvalidate()` instead of `invalidate()`.

For information about extending the `View` class, read Creating a view class.

Drawing on a SurfaceView

`SurfaceView` is a special subclass of `View` that offers a dedicated drawing surface within the view hierarchy. The goal is to offer this drawing surface to an app's worker thread. This way, the app isn't required to wait until the system's view hierarchy is ready to draw. Instead, a worker thread that has a reference to a `SurfaceView` object can draw to its own canvas at its own pace.

To begin, you need to create a new class that extends `SurfaceView`. This class should also implement the `SurfaceHolder.Callback` interface, which provides events that happen in the underlying `Surface` object, such as when it's created, changed, or destroyed. These events let you know when you can start drawing, whether you need to make adjustments based on new surface properties, and when to stop drawing and potentially terminate some tasks. The class that extends `SurfaceView` is also a good place to define your worker thread, which calls all the drawing procedures in your canvas.

Instead of handling the `Surface` object directly, you should handle it via a `SurfaceHolder`. After your `SurfaceView` object is initialized, you can get a `SurfaceHolder` object by calling `getHolder()`. You should register your `SurfaceView` object to receive notifications from the `SurfaceHolder` by calling `addCallback()`. Then implement each `SurfaceHolder.Callback` abstract method in your `SurfaceView` class.

You can draw to the surface canvas from a worker thread that has access to a `SurfaceHolder` object. Perform the following steps from inside the worker thread every time your app needs to redraw the surface:

1. Use `lockCanvas()` to retrieve the canvas.
2. Perform drawing operations on the canvas.
3. Unlock the canvas by calling `unlockCanvasAndPost(Canvas)` passing the `Canvas` object that you used for your drawing operations.

The surface draws the canvas considering all the drawing operations you performed on it.

> **Note:** Every time you retrieve the canvas from the `SurfaceHolder`, the previous state of the canvas is retained. In order to properly animate your graphics, you must repaint the entire surface. For example, you can clear the previous state of the canvas by filling in a color using the `drawColor()` method or setting a background image using the `drawBitmap()` method. Otherwise, your canvas could show traces of previous drawings.

Drawables

The Android framework offers a custom 2-D graphics library for drawing shapes and images. The `android.graphics.drawable` package contains common classes used for drawing in two dimensions.

This section discusses the basics of using drawable objects to draw graphics and how to use a couple of subclasses of the `Drawable` class. For information on how to use drawables for frame-by-frame animation, see Drawable Animation.

A `Drawable` is a general abstraction for *something that can be drawn*. The Android framework offers a set of direct and indirect subclasses of `Drawable` that you can use in a variety of scenarios. You can also extend these classes to define your own custom drawable objects that behave in unique ways.

There are two ways to define and instantiate a `Drawable` besides using the standard class constructors:

a. Using an resource image saved in your project.

b. Using an XML resource that defines the drawable properties.

Creating drawables from resource images

You can add graphics to your app by referencing an image file from your project resources. Supported file types are PNG (preferred), JPG (acceptable), and GIF (discouraged). App icons, logos, and other graphics, such as those used in games, are well suited for this technique.

To use an image resource, add your file to the `res/drawable/` directory of your project. Once in your project, you can reference the image resource from your code or your XML layout. Either way, it's referred to using a resource ID, which is the file name without the file type extension. For example, refer to `my_image.png` as `my_image`.

> **Note:** Image resources placed in the `res/drawable/` directory may be automatically optimized with lossless image compression by the `aapt` tool during the build process. For example, a true-color PNG that doesn't require more than 256 colors may be converted to an 8-bit PNG with a color palette. This results in an image of equal quality but which requires less memory. As a result, the image binaries placed in this directory can change at build time. If you plan on reading an image as a bitstream in order to convert it to a bitmap, put your images in the `res/raw/` folder instead, where the `aapt` tool doesn't modify them.

The following code snippet demonstrates how to build an `ImageView` that uses an image created from a drawable resource and adds it to the layout:

```
LinearLayout mLinearLayout;

protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Create a LinearLayout in which to add the ImageView
  mLinearLayout = new LinearLayout(this);

  // Instantiate an ImageView and define its properties
  ImageView i = new ImageView(this);
  i.setImageResource(R.drawable.my_image);

  // set the ImageView bounds to match the Drawable's dimensions
  i.setAdjustViewBounds(true);
  i.setLayoutParams(new Gallery.LayoutParams(LayoutParams.WRAP_CONTENT,
      LayoutParams.WRAP_CONTENT));

  // Add the ImageView to the layout and set the layout as the content view
  mLinearLayout.addView(i);
  setContentView(mLinearLayout);
}
```

In other cases, you may want to handle your image resource as a `Drawable` object, as shown in the following example:

```
Resources res = mContext.getResources();
Drawable myImage = res.getDrawable(R.drawable.my_image);
```

> **Note:** Each unique resource in your project can maintain only one state, no matter how many different objects you instantiate for it. For example, if you instantiate two `Drawable` objects from the same image resource and change a property (such as the alpha) for one object, then it also affects the other. When dealing with multiple instances of an image resource, instead of directly transforming the `Drawable` object you should perform a tween animation.

The XML snippet below shows how to add a drawable resource to an `ImageView` in the XML layout:

```
<ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/my_image" />
```

For more information about using project resources, see Resources and Assets.

Creating drawables from XML resources

If there is a `Drawable` object that you'd like to create, which isn't initially dependent on variables defined by your code or user interaction, then defining the `Drawable` in XML is a good option. Even if you expect your `Drawable` to change its properties during the user's interaction with your app, you should consider defining the object in XML, as you can modify properties after it's instantiated.

After you've defined your `Drawable` in XML, save the file in the `res/drawable/` directory of your project. The following example shows the XML that defines a `TransitionDrawable` resource, which inherits from `Drawable`:

```
<!-- res/drawable/expand_collapse.xml -->
<transition xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@drawable/image_expand">
    <item android:drawable="@drawable/image_collapse">
</transition>
```

Then, retrieve and instantiate the object by calling `Resources.getDrawable()`, and passing the resource ID of your XML file. Any `Drawable` subclass that supports the `inflate()` method can be defined in XML and instantiated by your app. Each drawable class that supports XML inflation utilizes specific XML attributes that help define the object properties. The following code instantiates the `TransitionDrawable` and sets it as the content of an `ImageView` object:

```
Resources res = mContext.getResources();
TransitionDrawable transition =
    (TransitionDrawable) res.getDrawable(R.drawable.expand_collapse);

ImageView image = (ImageView) findViewById(R.id.toggle_image);
image.setImageDrawable(transition);

// Then you can call the TransitionDrawable object's methods
transition.startTransition(1000);
```

For more information about the XML attributes supported, refer to the classes listed above.

Shape drawables

A `ShapeDrawable` object can be a good option when you want to dynamically draw a two-dimensional graphic. You can programmatically draw primitive shapes on a `ShapeDrawable` object and apply the styles that your app needs.

`ShapeDrawable` is a subclass of `Drawable`. For this reason, you can use a `ShapeDrawable` wherever a `Drawable` is expected. For example, you can use a `ShapeDrawable` object to set the background of a view by passing it to the `setBackgroundDrawable()` method of the view. You can also draw your shape as its own custom view and add it to a layout in your app.

Because `ShapeDrawable` has its own `draw()` method, you can create a subclass of `View` that draws the `ShapeDrawable` object during the `onDraw()` event, as shown in the following code example:

```
public class CustomDrawableView extends View {
  private ShapeDrawable mDrawable;

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

    int x = 10;
    int y = 10;
    int width = 300;
    int height = 50;

    mDrawable = new ShapeDrawable(new OvalShape());
    // If the color isn't set, the shape uses black as the default.
    mDrawable.getPaint().setColor(0xff74AC23);
    // If the bounds aren't set, the shape can't be drawn.
    mDrawable.setBounds(x, y, x + width, y + height);
  }

  protected void onDraw(Canvas canvas) {
    mDrawable.draw(canvas);
  }
}
```

You can use the `CustomDrawableView` class in the code sample above as you would use any other custom view. For example, you can programmatically add it to an activity in your app, as shown in the following example:

```
CustomDrawableView mCustomDrawableView;

protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  mCustomDrawableView = new CustomDrawableView(this);

  setContentView(mCustomDrawableView);
}
```

If you want to use the custom view in the XML layout instead, then the `CustomDrawableView` class must override the `View(Context, AttributeSet)` constructor, which is called when the class is inflated from XML. The following example shows how to declare the `CustomDrawableView` in the XML layout:

```
<com.example.shapedrawable.CustomDrawableView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        />
```

The `ShapeDrawable` class, like many other drawable types in the `android.graphics.drawable` package, allows you to define various properties of the object by using public methods. Some example properties you might want to adjust include alpha transparency, color filter, dither, opacity, and color.

You can also define primitive drawable shapes using XML resources. For more information, see Shape Drawable in Drawable Resource Types.

NinePatch drawables

A `NinePatchDrawable` graphic is a stretchable bitmap image that you can use as the background of a view. Android automatically resizes the graphic to accommodate the contents of the view. An example use of a NinePatch image is the background used by standard Android buttons—buttons must stretch to accommodate strings of various lengths. A NinePatch graphic is a standard PNG image that includes an extra 1-pixel border. It must be saved with the `9.png` extension in the `res/drawable/` directory of your project.

Use the border to define the stretchable and static areas of the image. You indicate a stretchable section by drawing one (or more) 1-pixel wide black line(s) in the left and top part of the border (the other border pixels should be fully transparent or white). You can have as many

stretchable sections as you want. The relative size of the stretchable sections stays the same, so the largest section always remains the largest.

You can also define an optional drawable section of the image (effectively, the padding lines) by drawing a line on the right and a line on the bottom. If a `View` object sets the NinePatch graphic as its background and then specifies the view's text, it stretches itself so that all the text occupies only the area designated by the right and bottom lines (if included). If the padding lines aren't included, Android uses the left and top lines to define this drawable area.

To clarify the difference between the lines, the left and top lines define which pixels of the image are allowed to be replicated in order to stretch the image. The bottom and right lines define the relative area within the image that the contents of the view are allowed to occupy.

Figure 1 shows an example of a NinePatch graphic used to define a button:

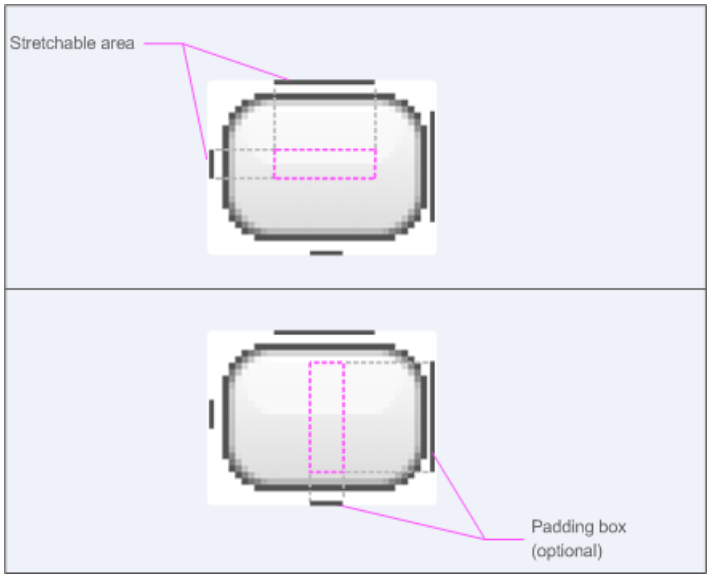

Figure 1: Example of a NinePatch graphic that defines a button

This NinePatch graphic defines one stretchable area with the left and top lines, and the drawable area with the bottom and right lines. In the top image, the dotted grey lines identify the regions of the image that are replicated in order to stretch the image. The pink rectangle in the bottom image identifies the region in which the contents of the view are allowed. If the contents don't fit in this region, then the image is stretched to make them fit.

The Draw 9-patch tool offers an extremely handy way to create your NinePatch images, using a WYSIWYG graphics editor. It even raises warnings if the region you've defined for the stretchable area is at risk of producing drawing artifacts as a result of the pixel replication.

The following sample layout XML demonstrates how to add a NinePatch graphic to a couple of buttons. The NinePatch image is saved to `res/drawable/my_button_background.9.png`.

```
<Button id="@+id/tiny"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerInParent="true"
        android:text="Tiny"
        android:textSize="8sp"
        android:background="@drawable/my_button_background"/>

<Button id="@+id/big"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_centerInParent="true"
        android:text="Biiiiiiig text!"
        android:textSize="30sp"
        android:background="@drawable/my_button_background"/>
```

Note that the `layout_width` and `layout_height` attributes are set to `wrap_content` to make the button fit neatly around the text.

Figure 2 shows the two buttons rendered from the XML and NinePatch image shown above. Notice how the width and height of the button

varies with the text, and the background image stretches to accommodate it.

Figure 2: Buttons rendered using an XML resource and a NinePatch graphic

Vector drawables

A vector drawable is a vector graphic defined in an XML file as a set of points, lines, and curves along with its associated color information. The Android framework provides the `VectorDrawable` and `AnimatedVectorDrawable` classes, which support vector graphics as drawable resources.

Apps that target Android versions lower than 5.0 (API level 21) can use the Support Library version 23.2 or higher to get support for vector drawables and animated vector drawables. For more information about using the vector drawable classes, see Vector Drawable.