Hardware Acceleration

In this document

See also

Beginning in Android 3.0 (API level 11), the Android 2D rendering pipeline supports hardware acceleration, meaning that all drawing operations that are performed on a `View`'s canvas use the GPU. Because of the increased resources required to enable hardware acceleration, your app will consume more RAM.

Hardware acceleration is enabled by default if your Target API level is >=14, but can also be explicitly enabled. If your application uses only standard views and `Drawable`s, turning it on globally should not cause any adverse drawing effects. However, because hardware acceleration is not supported for all of the 2D drawing operations, turning it on might affect some of your custom views or drawing calls. Problems usually manifest themselves as invisible elements, exceptions, or wrongly rendered pixels. To remedy this, Android gives you the option to enable or disable hardware acceleration at multiple levels. See Controlling Hardware Acceleration.

If your application performs custom drawing, test your application on actual hardware devices with hardware acceleration turned on to find any problems. The Unsupported drawing operations section describes known issues with hardware acceleration and how to work around them.

Controlling Hardware Acceleration

You can control hardware acceleration at the following levels:

- Application
- Activity
- Window
- View

Application level

In your Android manifest file, add the following attribute to the `<application>` tag to enable hardware acceleration for your entire application:

```
<application android:hardwareAccelerated="true" ...>
```

Activity level

If your application does not behave properly with hardware acceleration turned on globally, you can control it for individual activities as well. To enable or disable hardware acceleration at the activity level, you can use the `android:hardwareAccelerated` attribute for the `<activity>` element. The following example enables hardware acceleration for the entire application but disables it for one activity:

```
<application android:hardwareAccelerated="true">
    <activity ... />
    <activity android:hardwareAccelerated="false" />
</application>
```

Window level

If you need even more fine-grained control, you can enable hardware acceleration for a given window with the following code:

```
getWindow().setFlags(
    WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED,
    WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED);
```

> **Note**: You currently cannot disable hardware acceleration at the window level.

View level

You can disable hardware acceleration for an individual view at runtime with the following code:

```
myView.setLayerType(View.LAYER_TYPE_SOFTWARE, null);
```

> **Note**: You currently cannot enable hardware acceleration at the view level. View layers have other functions besides disabling hardware acceleration. See View layers for more information about their uses.

Determining if a View is Hardware Accelerated

It is sometimes useful for an application to know whether it is currently hardware accelerated, especially for things such as custom views. This is particularly useful if your application does a lot of custom drawing and not all operations are properly supported by the new rendering pipeline.

There are two different ways to check whether the application is hardware accelerated:

- `View.isHardwareAccelerated()` returns `true` if the `View` is attached to a hardware accelerated window.
- `Canvas.isHardwareAccelerated()` returns `true` if the `Canvas` is hardware accelerated

If you must do this check in your drawing code, use `Canvas.isHardwareAccelerated()` instead of `View.isHardwareAccelerated()` when possible. When a view is attached to a hardware accelerated window, it can still be drawn using a non-hardware accelerated Canvas. This happens, for instance, when drawing a view into a bitmap for caching purposes.

Android Drawing Models

When hardware acceleration is enabled, the Android framework utilizes a new drawing model that utilizes *display lists* to render your application to the screen. To fully understand display lists and how they might affect your application, it is useful to understand how Android draws views without hardware acceleration as well. The following sections describe the software-based and hardware-accelerated drawing models.

Software-based drawing model

In the software drawing model, views are drawn with the following two steps:

1. Invalidate the hierarchy
2. Draw the hierarchy

Whenever an application needs to update a part of its UI, it invokes `invalidate()` (or one of its variants) on any view that has changed

content. The invalidation messages are propagated all the way up the view hierarchy to compute the regions of the screen that need to be redrawn (the dirty region). The Android system then draws any view in the hierarchy that intersects with the dirty region. Unfortunately, there are two drawbacks to this drawing model:

- First, this model requires execution of a lot of code on every draw pass. For example, if your application calls `invalidate()` on a button and that button sits on top of another view, the Android system redraws the view even though it hasn't changed.
- The second issue is that the drawing model can hide bugs in your application. Since the Android system redraws views when they intersect the dirty region, a view whose content you changed might be redrawn even though `invalidate()` was not called on it. When this happens, you are relying on another view being invalidated to obtain the proper behavior. This behavior can change every time you modify your application. Because of this, you should always call `invalidate()` on your custom views whenever you modify data or state that affects the view's drawing code.

> **Note**: Android views automatically call `invalidate()` when their properties change, such as the background color or the text in a `TextView`.

Hardware accelerated drawing model

The Android system still uses `invalidate()` and `draw()` to request screen updates and to render views, but handles the actual drawing differently. Instead of executing the drawing commands immediately, the Android system records them inside display lists, which contain the output of the view hierarchy's drawing code. Another optimization is that the Android system only needs to record and update display lists for views marked dirty by an `invalidate()` call. Views that have not been invalidated can be redrawn simply by re-issuing the previously recorded display list. The new drawing model contains three stages:

1. Invalidate the hierarchy
2. Record and update display lists
3. Draw the display lists

With this model, you cannot rely on a view intersecting the dirty region to have its `draw()` method executed. To ensure that the Android system records a view's display list, you must call `invalidate()`. Forgetting to do so causes a view to look the same even after it has been changed.

Using display lists also benefits animation performance because setting specific properties, such as alpha or rotation, does not require invalidating the targeted view (it is done automatically). This optimization also applies to views with display lists (any view when your application is hardware accelerated.) For example, assume there is a `LinearLayout` that contains a `ListView` above a `Button`. The display list for the `LinearLayout` looks like this:

- DrawDisplayList(ListView)
- DrawDisplayList(Button)

Assume now that you want to change the `ListView`'s opacity. After invoking `setAlpha(0.5f)` on the `ListView`, the display list now contains this:

- SaveLayerAlpha(0.5)
- DrawDisplayList(ListView)
- Restore
- DrawDisplayList(Button)

The complex drawing code of `ListView` was not executed. Instead, the system only updated the display list of the much simpler `LinearLayout`. In an application without hardware acceleration enabled, the drawing code of both the list and its parent are executed again.

Unsupported Drawing Operations

When hardware accelerated, the 2D rendering pipeline supports the most commonly used `Canvas` drawing operations as well as many less-used operations. All of the drawing operations that are used to render applications that ship with Android, default widgets and layouts, and common advanced visual effects such as reflections and tiled textures are supported.

The following table describes the support level of various operations across API levels:

	First supported API level
Canvas	
drawBitmapMesh() (colors array)	18
drawPicture()	23
drawPosText()	16
drawTextOnPath()	16
drawVertices()	✗
setDrawFilter()	16
clipPath()	18
clipRegion()	18
clipRect(Region.Op.XOR)	18
clipRect(Region.Op.Difference)	18
clipRect(Region.Op.ReverseDifference)	18
clipRect() with rotation/perspective	18
Paint	
setAntiAlias() (for text)	18
setAntiAlias() (for lines)	16
setFilterBitmap()	17
setLinearText()	✗
setMaskFilter()	✗
setPathEffect() (for lines)	✗
setRasterizer()	✗
setShadowLayer() (other than text)	✗
setStrokeCap() (for lines)	18
setStrokeCap() (for points)	19
setSubpixelText()	✗
Xfermode	
PorterDuff.Mode.DARKEN (framebuffer)	✗
PorterDuff.Mode.LIGHTEN (framebuffer)	✗
PorterDuff.Mode.OVERLAY (framebuffer)	✗
Shader	
ComposeShader inside ComposeShader	✗
Same type shaders inside ComposeShader	✗
Local matrix on ComposeShader	18

Canvas Scaling

The hardware accelerated 2D rendering pipeline was built first to support unscaled drawing, with some drawing operations degrading quality significantly at higher scale values. These operations are implemented as textures drawn at scale 1.0, transformed by the GPU. In API level <17, using these operations will result in scaling artifacts increasing with scale.

The following table shows when implementation was changed to correctly handle large scales:

Drawing operation to be scaled	First supported API level
drawText()	18
drawPosText()	✗
drawTextOnPath()	✗
Simple Shapes*	17
Complex Shapes*	✗
drawPath()	✗
Shadow layer	✗

> **Note**: 'Simple' shapes are `drawRect()`, `drawCircle()`, `drawOval()`, `drawRoundRect()`, and `drawArc()` (with useCenter=false) commands issued with a Paint that doesn't have a PathEffect, and doesn't contain non-default joins (via `setStrokeJoin()` / `setStrokeMiter()`). Other instances of those draw commands fall under 'Complex,' in the above chart.

If your application is affected by any of these missing features or limitations, you can turn off hardware acceleration for just the affected portion of your application by calling `setLayerType(View.LAYER_TYPE_SOFTWARE, null)`. This way, you can still take advantage of hardware acceleration everywhere else. See Controlling Hardware Acceleration for more information on how to enable and disable hardware acceleration at different levels in your application.

View Layers

In all versions of Android, views have had the ability to render into off-screen buffers, either by using a view's drawing cache, or by using `Canvas.saveLayer()`. Off-screen buffers, or layers, have several uses. You can use them to get better performance when animating complex views or to apply composition effects. For instance, you can implement fade effects using `Canvas.saveLayer()` to temporarily render a view into a layer and then composite it back on screen with an opacity factor.

Beginning in Android 3.0 (API level 11), you have more control on how and when to use layers with the `View.setLayerType()` method. This API takes two parameters: the type of layer you want to use and an optional `Paint` object that describes how the layer should be composited. You can use the `Paint` parameter to apply color filters, special blending modes, or opacity to a layer. A view can use one of three layer types:

- `LAYER_TYPE_NONE`: The view is rendered normally and is not backed by an off-screen buffer. This is the default behavior.
- `LAYER_TYPE_HARDWARE`: The view is rendered in hardware into a hardware texture if the application is hardware accelerated. If the application is not hardware accelerated, this layer type behaves the same as `LAYER_TYPE_SOFTWARE`.
- `LAYER_TYPE_SOFTWARE`: The view is rendered in software into a bitmap.

The type of layer you use depends on your goal:

- **Performance**: Use a hardware layer type to render a view into a hardware texture. Once a view is rendered into a layer, its drawing code does not have to be executed until the view calls `invalidate()`. Some animations, such as alpha animations, can then be applied directly onto the layer, which is very efficient for the GPU to do.
- **Visual effects**: Use a hardware or software layer type and a `Paint` to apply special visual treatments to a view. For instance, you can draw a view in black and white using a `ColorMatrixColorFilter`.
- **Compatibility**: Use a software layer type to force a view to be rendered in software. If a view that is hardware accelerated (for instance, if your whole application is hardware acclerated), is having rendering problems, this is an easy way to work around limitations of the hardware rendering pipeline.

View layers and animations

Hardware layers can deliver faster and smoother animations when your application is hardware accelerated. Running an animation at 60 frames per second is not always possible when animating complex views that issue a lot of drawing operations. This can be alleviated by using hardware layers to render the view to a hardware texture. The hardware texture can then be used to animate the view, eliminating the need for the view to constantly redraw itself when it is being animated. The view is not redrawn unless you change the view's properties, which calls `invalidate()`, or if you call `invalidate()` manually. If you are running an animation in your application and do not obtain the smooth results you want, consider enabling hardware layers on your animated views.

When a view is backed by a hardware layer, some of its properties are handled by the way the layer is composited on screen. Setting these properties will be efficient because they do not require the view to be invalidated and redrawn. The following list of properties affect the way the layer is composited. Calling the setter for any of these properties results in optimal invalidation and no redrawing of the targeted view:

- `alpha`: Changes the layer's opacity
- `x`, `y`, `translationX`, `translationY`: Changes the layer's position
- `scaleX`, `scaleY`: Changes the layer's size
- `rotation`, `rotationX`, `rotationY`: Changes the layer's orientation in 3D space

- `pivotX`, `pivotY`: Changes the layer's transformations origin

These properties are the names used when animating a view with an `ObjectAnimator`. If you want to access these properties, call the appropriate setter or getter. For instance, to modify the alpha property, call `setAlpha()`. The following code snippet shows the most efficient way to rotate a viewiew in 3D around the Y-axis:

```
view.setLayerType(View.LAYER_TYPE_HARDWARE, null);
ObjectAnimator.ofFloat(view, "rotationY", 180).start();
```

Because hardware layers consume video memory, it is highly recommended that you enable them only for the duration of the animation and then disable them after the animation is done. You can accomplish this using animation listeners:

```
View.setLayerType(View.LAYER_TYPE_HARDWARE, null);
ObjectAnimator animator = ObjectAnimator.ofFloat(view, "rotationY", 180);
animator.addListener(new AnimatorListenerAdapter() {
    @Override
    public void onAnimationEnd(Animator animation) {
        view.setLayerType(View.LAYER_TYPE_NONE, null);
    }
});
animator.start();
```

For more information on property animation, see Property Animation.

Tips and Tricks

Switching to hardware accelerated 2D graphics can instantly increase performance, but you should still design your application to use the GPU effectively by following these recommendations:

Reduce the number of views in your application

The more views the system has to draw, the slower it will be. This applies to the software rendering pipeline as well. Reducing views is one of the easiest ways to optimize your UI.

Avoid overdraw

Do not draw too many layers on top of each other. Remove any views that are completely obscured by other opaque views on top of it. If you need to draw several layers blended on top of each other, consider merging them into a single layer. A good rule of thumb with current hardware is to not draw more than 2.5 times the number of pixels on screen per frame (transparent pixels in a bitmap count!).

Don't create render objects in draw methods

A common mistake is to create a new `Paint` or a new `Path` every time a rendering method is invoked. This forces the garbage collector to run more often and also bypasses caches and optimizations in the hardware pipeline.

Don't modify shapes too often

Complex shapes, paths, and circles for instance, are rendered using texture masks. Every time you create or modify a path, the hardware pipeline creates a new mask, which can be expensive.

Don't modify bitmaps too often

Every time you change the content of a bitmap, it is uploaded again as a GPU texture the next time you draw it.

Use alpha with care

When you make a view translucent using `setAlpha()`, `AlphaAnimation`, or `ObjectAnimator`, it is rendered in an off-screen buffer which doubles the required fill-rate. When applying alpha on very large views, consider setting the view's layer type to `LAYER_TYPE_HARDWARE`.