Android canvas draw shadow

Create. A path, adding some elements to it suggests BlurMaskFilter at a Paint pull a path with dx, thigh shade offset unlocked mask filter drawing a path again with no. Offset Material Design is filled with fancy shapes and shadows, but not all of these things are implemented and ready to use. There are a good number of 3rd-party libraries, but their quality varies. Let's go through the options we have when it comes to blinking and shadows. ClippingHere 'clipper' means 'limiting the drawing of a view's contents to a specific form'. Material design introduced rounded corners for buttons and charts. Floating Action button was a special, circular type button. Currently, with Material Design 2, we have more rounded and cut angles, diagonal slices and other interesting shapes. If the clipped view is a layout, clipping is also applied to its children. This is especially important when talking about CardView and thoroughbred images. ViewOutlineProviderStarting with Android Lollipop there is a native way to clip views using view lines. This method is hardware accelerated, very quickly, quite easy to use, correctly clipping the layout's contents and producing antialiased outlines. This should be the default method used in API 21+ and the only method used by system components. However, it has two drawbacks. The first one is pretty obvious - you can't use it on KitKat and below. In such a case, you should use one of the other options or just skip token shadows altogether. Older phones are usually slower, so this can be a good solution. The lack of blinking on pre-Lollipop systems leads to tons of issues related to CardView, where blinking is a necessity. Therefore, CardView forces an ugly paddle or refuses to round up corners on older platforms. A map with its angles correctly rounded on API 14The other one is more problematic. ViewOutlineProvider cannot blink to shapes other than rectangles, rounded rectangles, and circles. Even ovals are not supported. Neither do roads, even convexes. It's even more surprising when you realize that a lot of recent Material Design ideas require a cut-off angle or a cradle in a view. The second problem introduced a lot of friction between developers and designers unaware of that limitation. Especially when the clipped cinema/plane ticket shapes became very popular on Dribbble and Behance. Unfortunately, there's no way to support such forms using ViewOutlineProvider from today. View.setBackground() and DrawableIf the view's shape is simple, then instead of clipping it, it can use a background tokenable forging of the clipper, for example, a rounded rectangle with a bitmap, a gradient, or a solid color. This technique can be used to achieve the correct shape of a button or an ImageView used to show profile pictures. A view's background will its contents don't blink, but it's easy to prepare, work on all platforms, and is very quick to draw. Drawable supports all concave shapes, so it's possible to draw stars, theater tickets and other things to clip using ViewOutlineProvider.The Design Support Library uses tokenables for its components to support rounded corners of Button and CardView on pre-Lollipop platforms. Therefore, CardView acts on older phones as described in the previous paragraph. Canvas.clipPath()Each view uses a Canvas instance to draw itself. It's possible to set a clip shape on those Canvas and manually draw the clipped content. This method requires a small drawing code change in each component we want to clip. This kind of clipper is fast, works on all platforms and supports any form. There are two drawbacks. The first one is that there is no antialiasing, so the outline is very sharp. The other one is that Canvas.clipPath() does not accelerate hardware on API 14 through 17. I suppose that with the current market share this is not a game changer. Nevertheless, it's important to remember that certain operations force the drawing code to switch to the software rendering pipeline, resulting in slower drawing and less FPS. Paint.setXferemode()This method is the most complicated and computational-heavy, but gives the best results on all platforms: it supports all shapes, is hardware accelerated and the outline is antialiased. The idea is familiar to artists as 'masking'. All we need is a separate layer to mask new content, a mask, and a compilation code. The algorithm is quite easy: Create a new layer for masking operations using Canvas.saveLayer(). Record the view. Switches the drawing mode to either cleaning unmasked or holding masked parts. Draw the mask.Compose layers using Canvas.restore(). ShadowsMaterial Design shadows are dynamic and take into account the caster's shape, position and opacity. Shadows have a bunch of roles — they help show importance, purpose and relationships between components. Android Lollipop has introduced a native mechanism for delivering nice, hardware-accelerated shadows. This part of the framework was not officially backed up and is not available for older platforms. The Design Support Library uses custom drawings to draw shadows. ViewOutlineProviderAgain, the official method uses the view's outline and works on API 21+. It's hardware accelerated, fast and looks great. There is a small difference in supported forms - when drawing shadows, the outline can be any convex shape, so it is possible to use custom paths. This leads to strange situations where you can get nice shadows for diagonally cut views, but you need to clip them using another method. ViewOutlineProvider supports colored shadows from API 28. This means a lot of waiting until this feature is used more widely. Koncave forms are not supported, so theater tickets mentioned earlier cannot throw shadows using this method. Although it's a reasonable case, there is also a more common one. The BottomAppBar component introduced in Material Design 2 has a cradle for a button, which makes its shape non-convex. As a result, BottomAppBar does not throw shadows using the official method, but Paint.setShadowLayer(). Paint.setShadowLayer()This method uses blurb to draw a shadow under anything drawn using that Paint object. It's easy to use, gives pretty good results and works with any shape and color. The Design Support Library provides the MaterialDrawableShape class, which uses Paint.setShadowLayer() internally - you can find it under the following link: There is one problem with Paint.setShadowLayer() - to draw things other than text, it is accelerated from API 28. This disadvantage makes this method pretty much unused as it can lead to problems with component drawing and animation. You can find more information about hardware-accelerated token operations in the docs. There is a table with these operations and the minimum supported API level for each of them. View.setBackground() and DrawableIt are, of course, possible to draw shadows using proper loadables - dynamic, 9 spots, or just images. In the HTML world, this solution has been widely used for a long time. When using a background signable for drawing shadows, we should remember that the shade becomes part of the view. This means that the view needs some padding inside to correctly align its contents. ScriptIntrinsicBlurThis is a RenderScript shade that we can use to generate shadows dynamically. These are the most complex and the slowest of the discussed methods, but it works in all cases, on all platforms and can be used with hardware layers without any problems. Here's the algorithm: Drawing a view's black shape to an offscreen bitmap. Blur it. Draw the blurred form under the actual view. Colored shadows generated using ScriptIntrisincBlur's generated shade can be set as the view's background with some padding, drawn by the view or drawn by the view's parent. Shadows can be chested and reused to reduce time needed for fading. Personally, I use a method based on hardware that fades with some optimizations. The most fancy trick is to generate a 9-patch instead of the full form of a view. This significantly reduces fad time and supports the size of animations for free. Share code, notes, and snippets immediately. You can't perform that action right now. You signed in with another tab or window. Restart to refresh your session. You signed out in another tab or window. Restart to refresh your session. We use optional third-party analytics cookies to understand how you can GitHub.com up so we can build better products. Learn more. We use optional third-party analytics cookies to understand how you can GitHub.com up so we can build better products. You can always update your choice by clicking cookie preferences at the bottom of the page. For more information, see our Privacy Statement. We use essential cookies to make essential website features Eg. Learn more we use analytics cookies to understand how you use our websites so that we can make them better, e.g. Eg. accomplish a task. Learn more