Introduction to animations

Animations can add visual cues that notify users about what's going on in your app. They are especially useful when the UI changes state, such as when new content loads or new actions become available. Animations also add a polished look to your app, which gives it a higher quality look and feel.

Android includes different animation APIs depending on what type of animation you want, so this page provides an overview of the different ways you can add motion to your UI.

To better understand when you should use animations, also see the material design guide to motion.

Note: For guidance on Animations in Compose, see the Compose Animation documentation.

Animate bitmaps

When you want to animate a bitmap graphic such as an icon or illustration, you should use the drawable animation APIs. Usually, these animations are defined statically with a drawable resource, but you can also define the animation behavior at runtime.

Figure 1. An animated drawable

For example, animating a play button transforming into a pause button when tapped is a nice way to communicate to the user that the two actions are related, and that pressing one makes the other visible.

For more information, read Animate Drawable Graphics.

Animate UI visibility and motion

When you need to change the visibility or position of views in your layout, you should include subtle animations to help the user understand how the UI is changing.

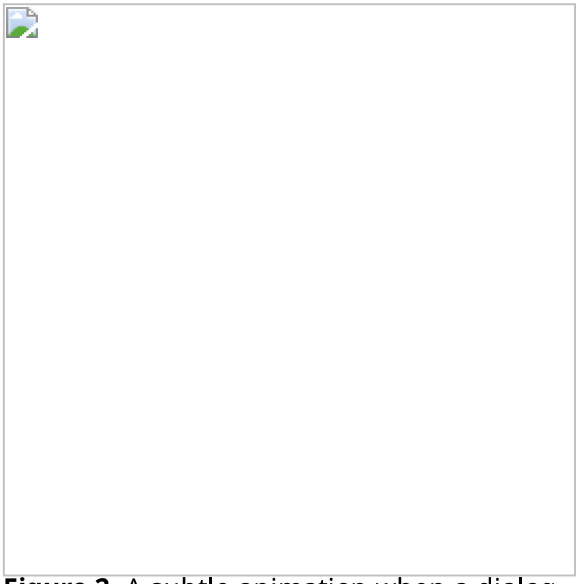

Figure 2. A subtle animation when a dialog appears and disappears makes the UI change less jarring

To move, reveal, or hide views within the current layout, you can use the property animation system provided by the android.animation package, available in Android 3.0 (API level 11) and higher. These APIs update the properties of your View objects over a period of time, continuously redrawing the view as the properties change. For example, when you

change the position properties, the view moves across the screen, or when you change the alpha property, the view fades in or out.

To create these animations with the least amount of effort, you can enable animations on your layout so that when you simply change the visibility of a view, an animation applies automatically. For more information, see Auto Animate Layout Updates.

To learn how to build animations with the property animation system, read the Property Animation Overview. Or see the following pages to create common animations:

- Change a view visibility with a crossfade
- Change a view visibility with a circular reveal
- Swap views with a card flip
- Change the view size with a zoom animation

Physics-based motion

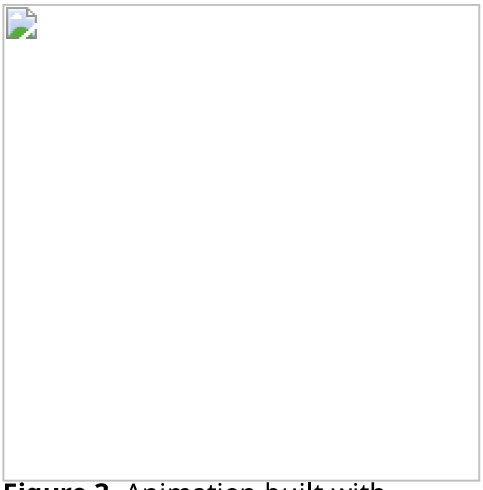

Figure 3. Animation built with ObjectAnimator

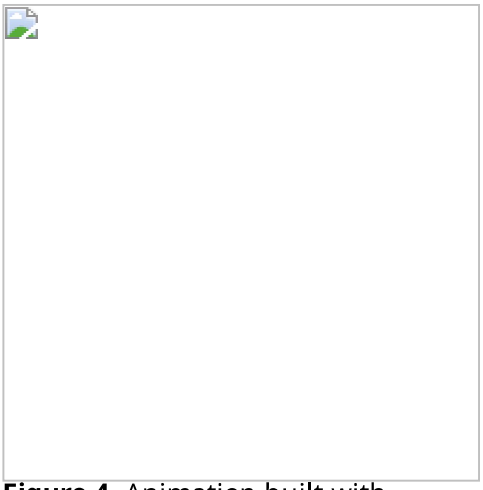

Figure 4. Animation built with physics-based APIs

Whenever possible, your animations should apply real-world physics so they are natural-looking. For example, they should maintain momentum when their target changes, and make smooth transitions during any changes.

To provide these behaviors, the Android Support library includes physics-based animation APIs that rely on the laws of physics to control how your animations occur.

Two common physics-based animations are the following:

- Spring Animation
- Fling Animation

Animations not based on physics—such as those built with ObjectAnimator APIs—are fairly static and have a fixed duration. If the target value changes, you need to cancel the animation at the time of target value change, re-configure the animation with a new value as the new start value, and add the new target value. Visually, this process creates an abrupt stop in the animation, and a disjointed movement afterwards, as shown in figure 3.

Whereas, animations built by with physics-based animation APIs such as DynamicAnimation are driven by force. The change in the target value results in a change in force. The new force applies on the existing velocity, which makes a continuous transition to the new target. This process results in a more natural-looking animation, as shown in figure 4.

Animate layout changes

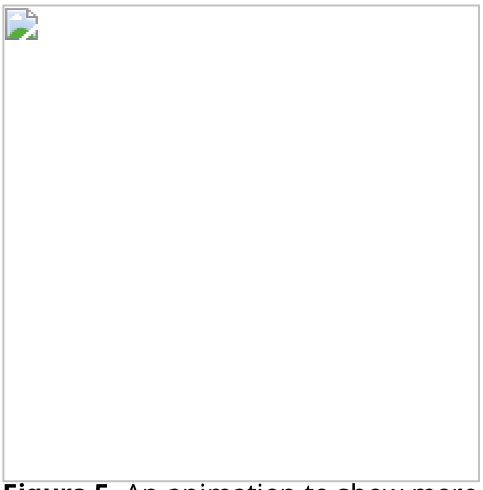

Figure 5. An animation to show more details can be achieved by either changing the layout or starting a new activity

On Android 4.4 (API level 19) and higher, you can use the transition framework to create animations when you swap the layout within the current activity or fragment. All you need to do is specify the starting and ending layout, and what type of animation you want to use. Then the system figures out and executes an animation between the two layouts. You can use this to swap out the entire UI or to move/replace just some views.

For example, when the user taps an item to see more information, you can replace the layout with the item details, applying a transition like the one shown in figure 5.

The starting and ending layout are each stored in a Scene, though the starting scene is usually determined automatically from the current layout. You then create a Transition to tell the system what type of animation you want, and then call TransitionManager.go() and the system runs the animation to swap the layouts.

For more information, read Animate Between Layouts Using a Transition. And for sample code, check out BasicTransition.

Animate between activities

On Android 5.0 (API level 21) and higher, you can also create animations that transition between your activities. This is based on the same transition framework described above to animate layout changes, but it allows you to create animations between layouts in *separate activities*.

You can apply simple animations such as sliding the new activity in from the side or fading it in, but you can also create animations that transition between shared views in each activity. For example, when the user taps an item to see more information, you can transition into a new activity with an animation that seamlessly grows that item to fill the screen, like the animation shown in figure 5.

As usual, you call startActivity(), but pass it a bundle of options provided by ActivityOptions.makeSceneTransitionAnimation(). This bundle of options may include which views are shared between the activities so the transition framework can connect them during the animation.

For all the details, see Start an Activity with an Animation. And for sample code, check out ActivitySceneTransitionBasic.