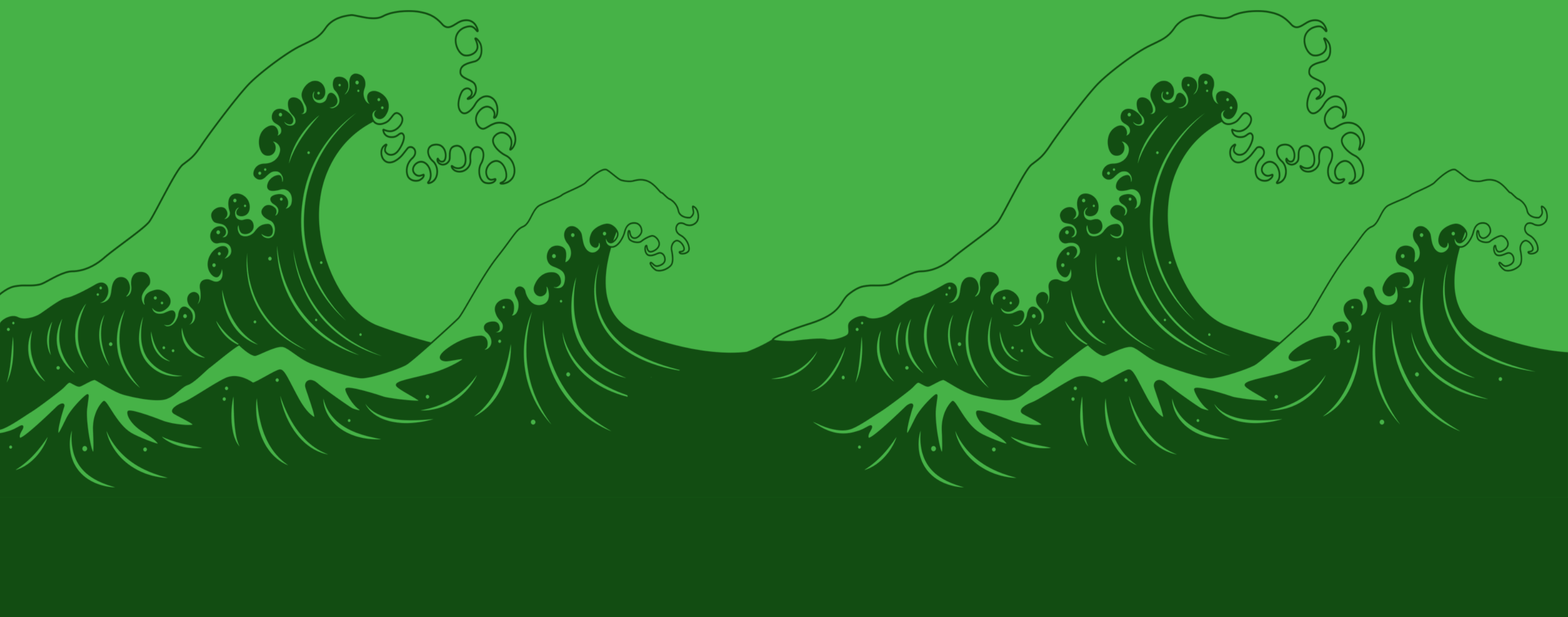
Bitesize Android KitKat

a review of Android KitKat for developers, in 6 bite-sized chunks



shinobicontrols

by Sam Davies

Bitesize Android KitKat

a review of Android KitKat for developers, in 6 bite-sized chunks

Sam Davies

This book is for sale at http://leanpub.com/bitesizekitkat

This version was published on 2014-04-01

Also By Sam Davies

iOS7 Day by Day

Contents

Full Screen Apps

Introduction

Full-screen apps allow the user to get fully engrossed with whatever content they are viewing or interacting with. Examples include e-book readers, video players, games and drawing apps. Before KitKat, although there was support for a fullscreen mode, KitKat introduces 2 new immersive modes.

In this chapter we'll take a look at 3 different full-screen modes, and discuss times when you might want to use them. There's an accompanying app which demonstrates how to use the different modes and what they might be used for. The source code is available on Github github.com/ShinobiControls/bitesize-kitkat[1], and is a gradle project which has been tested on Android Studio 0.4.4.

The different full-screen experiences

We're going to take a look at 3 different full-screen modes:

- **Leanback** This is the mode that was available pre-KitKat and is suited to video players.
- **Immersive** New to KitKat. Suitable for eBook reader - most of the time is spent in full-screen mode, but occasionally it's necessary to interact with other UI elements.
- **Sticky Immersive** New to KitKat. Suitable for games or a drawing app, where nearly all the workflow happens in full-screen mode.

First up, leanback mode.

Leanback

In leanback mode, the chrome such as the status bar, the activity bar and any UI controls will disappear after a period without user interaction. Whilst they are not visible then any interaction with the screen will cause them to reappear, and the touches won't be passed on to the underlying view. This means that any user interaction can only occur whilst the UI chrome is visible. This behavior is exactly the behavior you might expect from a video player - the user doesn't need to be able to interact with the content, instead they will expect some controls to appear when they touch the screen.

Android doesn't provide a UI visibility mode which performs exactly as we would want here, so we have to create it ourself.

We set the mode using a combination of flags and the `setSystemUiVisibility()` method on the decor view. The following flags are of interest for leanback mode:

[1]https://github.com/ShinobiControls/bitesize-kitkat

- `SYSTEM_UI_FLAG_LAYOUT_STABLE` provide a stable layout size to `fitSystemWindows()` method. This means that as the appearance of transient chrome changes then the layout of the underlying content should remain stable.
- `SYSTEM_UI_FLAG_FULLSCREEN` hide the status bar, action bar and other non-critical screen adornments.
- `SYSTEM_UI_FLAG_HIDE_NAVIGATION` if the device has navigation buttons (e.g. back, home etc) then hide them.
- `SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION` lay the screen out as if the navigation is hidden, but don't necessarily hide it. This means that when the navigation does become hidden then the layout won't have to redraw itself.
- `SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN` in a similar manner to the previous flag, lay the screen out as if it is in fullscreen mode.

The following method is used to enable/disable fullscreen mode:

```
protected void enableFullScreen(boolean enabled) {
    int newVisibility =  View.SYSTEM_UI_FLAG_LAYOUT_STABLE
                       | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
                       | View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN;

    if(enabled) {
        newVisibility |= View.SYSTEM_UI_FLAG_FULLSCREEN
                      |  View.SYSTEM_UI_FLAG_HIDE_NAVIGATION;
    }

    // Set the visibility
    getDecorView().setSystemUiVisibility(newVisibility);
}
```

Irrespective of whether we're 'leaning back', or allowing the user to interact with the app, we need to set the layout to assume we're hiding the navigation and using full-screen. This means that when we enter leanback mode, there isn't a noticeable jerk as the view re-lays itself out.

In the case where we are enabling fullscreen mode (i.e. leaning back) then we want to actually hide the navigation and enable full-screen.

We set the new visibility on the decor view, which is supplied by the `getDecorView()` method:

```
private View getDecorView() {
    return getWindow().getDecorView();
}
```

Now, once this has been called (with `true` as an argument) then the chrome will disappear. If the user interacts with the screen, then it'll automatically cause the chrome to reappear, the initial touch being absorbed by the OS. In order to get the leanback experience of the UI disappearing again after a period of time then we we will listen for UI visibility changes and start a hide timer when one occurs:

```java
public class LeanBackActivity extends AbstractFullScreenLayoutActivity
                         implements View.OnSystemUiVisibilityChangeListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        getDecorView().setOnSystemUiVisibilityChangeListener(this);
    }
    ...
}
```

and provide an implementation for the `onSystemUiVisibilityChange()` method:

```java
@Override
public void onSystemUiVisibilityChange(int visibility) {
    if((mLastSystemUIVisibility & View.SYSTEM_UI_FLAG_HIDE_NAVIGATION) != 0
                && (visibility & View.SYSTEM_UI_FLAG_HIDE_NAVIGATION) == 0) {
        resetHideTimer();
    }
    mLastSystemUIVisibility = visibility;
}
```

Here, we're checking to find out whether the change in visibility represents the appearance of the navigation (i.e. from hidden to visible), and if it is then we start a timer to make the chrome hide again:

```java
private void resetHideTimer() {
    // First cancel any queued events - i.e. resetting the countdown clock
    mLeanBackHandler.removeCallbacks(mEnterLeanback);
    // And fire the event in 3s time
    mLeanBackHandler.postDelayed(mEnterLeanback, 3000);
}
```

where `mLeanBackHandler` and `mEnterLeanback` are defined as follows:

```java
private final Handler mLeanBackHandler = new Handler();
private final Runnable mEnterLeanback = new Runnable() {
    @Override
    public void run() {
        enableFullScreen(true);
    }
};
```

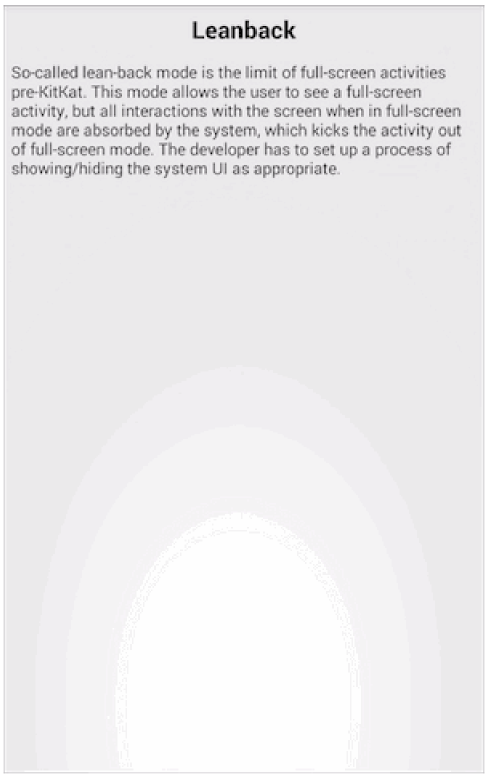
If the user interacts with the UI then we don't want the timer firing and hiding the controls away from underneath them. If you have some custom buttons then you could add a call to `resetHideTimer()` as part of the tap handler, but since we don't have any controls here, we'll add a `ClickListener` to the main content view:

```
protected void onCreate(Bundle savedInstanceState) {
    ...
    getMainView().setOnClickListener(this);
}
```

And reset the timer when a click is performed:

```
@Override
public void onClick(View v) {
    // If the `mainView` receives a click event then reset the leanback-mode c\
lock
    resetHideTimer();
}
```

This approach completes the leanback mode, and if you run it up you'll see behavior like the following:


Leanback

So-called lean-back mode is the limit of full-screen activities pre-KitKat. This mode allows the user to see a full-screen activity, but all interactions with the screen when in full-screen mode are absorbed by the system, which kicks the activity out of full-screen mode. The developer has to set up a process of showing/hiding the system UI as appropriate.

Leanback mode example

Immersive mode

Immersive view is new to KitKat, and allows users to interact with the view whilst in full-screen mode, in contrast to leanback mode, which shows the system UI as soon as interaction begins. This approach is ideal for ebook readers, where whilst reading the user would want to be able to see the entire page of content, but will need to be able to get back at the controls to change books and the suchlike.

The same flags are of interest - with the addition of:

- `SYSTEM_UI_FLAG_IMMERSIVE` enter immersive mode, where the view is fullscreen, and touches are sent through to the content - i.e. are not intercepted and treated as visibility change triggers.

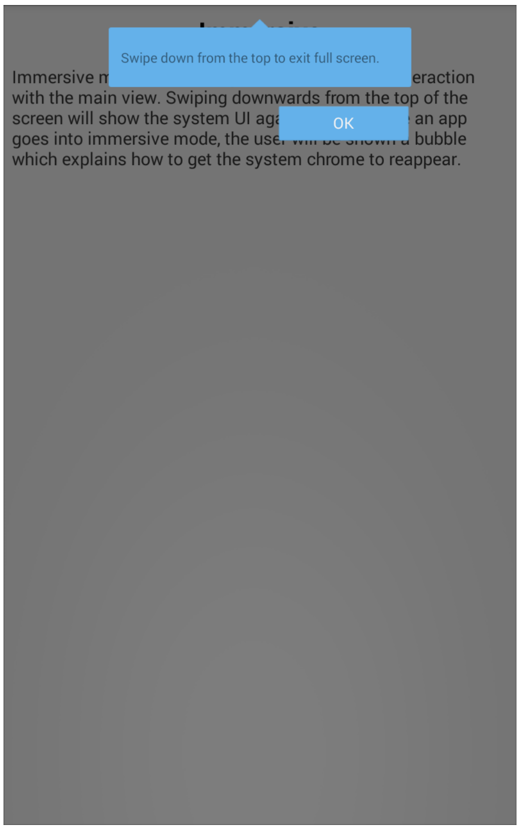
The following demonstrates the use of these flags to set visibility:

```java
protected void enableFullScreen(boolean enabled) {
    int newVisibility =  View.SYSTEM_UI_FLAG_LAYOUT_STABLE
                       | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
                       | View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN;

    if(enabled) {
        newVisibility |= View.SYSTEM_UI_FLAG_FULLSCREEN
                      | View.SYSTEM_UI_FLAG_HIDE_NAVIGATION
                      | View.SYSTEM_UI_FLAG_IMMERSIVE;
    }

    getDecorView().setSystemUiVisibility(newVisibility);
}
```

Calling this method with `enabled` set as `true` will cause the navigation buttons, the status bar and the action bar to animate out. In order to get them to reappear, the user must swipe downwards from the top of the screen, and as such, the first time immersive mode is entered in an app, the user will be presented with the following bubble:


Swipe down from the top to exit full screen.

Immersive m eraction with the main view. Swiping downwards from the top of the screen will show the system UI ag an app goes into immersive mode, the use bubble which explains how to get the system chrome to reappear.

Immersive Bubble

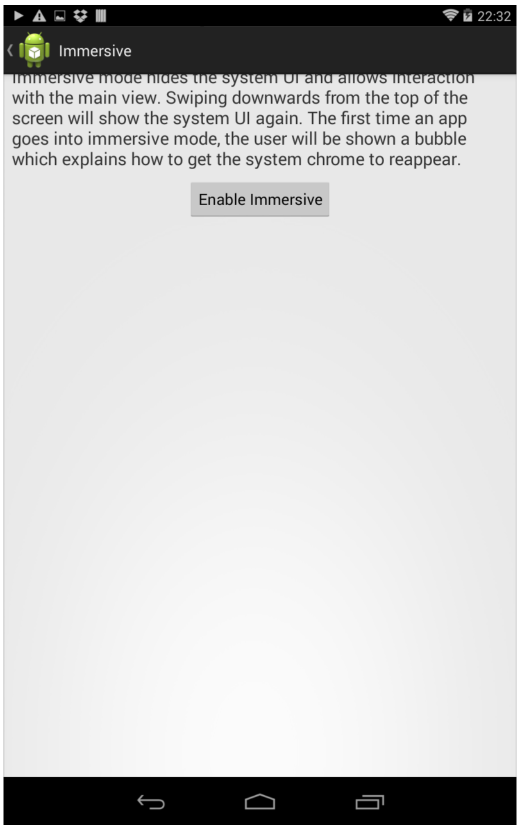
This disappears after a couple of seconds, and allows the user to interact with the full screen display as they might want to.

Once the user swipes down from the top, then immersive display is canceled, and the system UI chrome will reappear. This will involve a call to `onSystemUiVisibilityChanged()` of the relevant listener. In the accompanying demo app, we use this as an opportunity to display a button which allows the user to re-enter immersive mode:

```java
@Override
public void onSystemUiVisibilityChange(int visibility) {
    Button immersiveButton = (Button)findViewById(R.id.enableImmersiveButton);
    if((visibility & View.SYSTEM_UI_FLAG_HIDE_NAVIGATION) != 0) {
        // Hide button
        immersiveButton.setVisibility(View.INVISIBLE);
    } else {
        immersiveButton.setVisibility(View.VISIBLE);
    }
}
```

With the click handler simply calling our `enableFullScreen()` method:

```java
public void immersiveButtonClickHandler(View view) {
    enableFullScreen(true);
}
```



Immersive mode hides the system UI and allows interaction with the main view. Swiping downwards from the top of the screen will show the system UI again. The first time an app goes into immersive mode, the user will be shown a bubble which explains how to get the system chrome to reappear.

Enable immersive button

Note that the control flow of the system UI visibility change listener will hide the immersive button as the app enters immersive mode.

Sticky Immersive mode

The second new visibility mode added to KitKat is sticky immersive, and as you might expect it's very closely related to immersive mode. Whereas immersive mode is canceled when the user swipes down from the top of the screen, sticky immersive mode will automatically re-enter immersive mode after a short period of time. In this respect it is very much like a cross between immersive and lean-back. This makes it ideally suited for apps which may occasionally require the user to navigate around, but the main focus of the app is very much on the content - for example games or a drawing app.

Immersive sticky introduces one more visibility flag:

- `SYSTEM_UI_FLAG_IMMERSIVE_STICKY` enables immersive sticky mode. Importantly this flag will not be removed from the visibility mask when the user swipes down, and it will return to immersive mode after a short time.

Therefore, our `enableFullScreen()` method becomes the following:

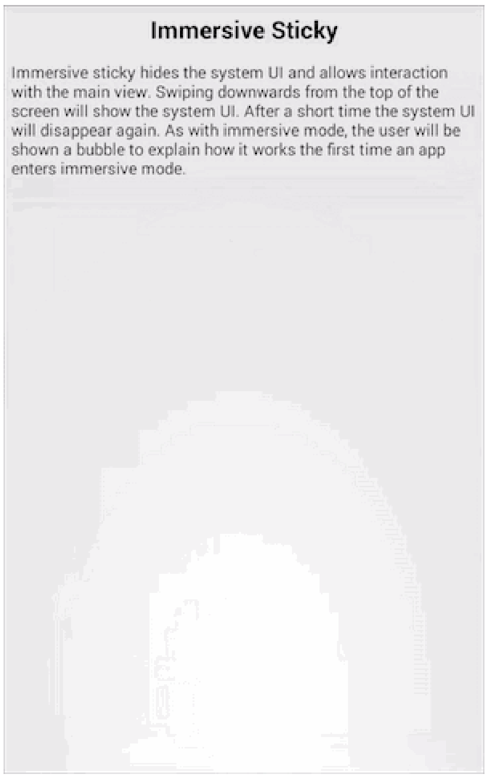
```
@Override
protected void enableFullScreen(boolean enabled) {
    int newVisibility =  View.SYSTEM_UI_FLAG_LAYOUT_STABLE
                       | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
                       | View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN;

    if(enabled) {
        newVisibility |= View.SYSTEM_UI_FLAG_FULLSCREEN
                      | View.SYSTEM_UI_FLAG_HIDE_NAVIGATION
                      | View.SYSTEM_UI_FLAG_IMMERSIVE_STICKY;
    }

    getDecorView().setSystemUiVisibility(newVisibility);
}
```

Calling this method with `true` as the `enabled` argument will enter sticky immersive, and this will remain until the system's UI visibility is changed elsewhere.

It has the following appearance:



Immersive Sticky

Immersive sticky hides the system UI and allows interaction with the main view. Swiping downwards from the top of the screen will show the system UI. After a short time the system UI will disappear again. As with immersive mode, the user will be shown a bubble to explain how it works the first time an app enters immersive mode.

Sticky Immersive

Conclusion

KitKat introduces a couple of new approaches for displaying full-screen content which makes it easier to get the behavior users expect from different app experiences.

The app that we've built here demonstrates how to use 3 popular full-screen modes, and you can play around with them to decide which will work best for your app. The code is available on Github github.com/ShinobiControls/bitesize-kitkat[2] and if you have any questions/comments then let me know below, on Github or via twitter @iwantmyrealname[3].

[2]https://github.com/ShinobiControls/bitesize-kitkat

[3]https://twitter.com/iwantmyrealname