COMPOSE / JETPACK / TESTING

Compose: UI Screenshot Testing

July 9, 2021 Mark Allison No comment

UI testing on Android has been tricky. However, Jetpack Compose makes it much easier. In a recent post we looked at how we can easily test adaptive layouts. But we can take this further. In another recent post, we created a strikethru animation overlay. In this post, we'll look at how we can test the animation using UI screenshots.

Before we begin, I must point out that I take no credit for this technique. I have based everything here on some tests from the Jetpack Compose Rally sample app. However, we'll make some adaptations along the way, and look at some gotchas, plus some useful techniques.

I understand that the original code in the Rally sample app was written by Jose Alcérreca. Many thanks to Jose for providing a great example of how to do this, and some code that we can copy and adapt.

The fundamental principle that screenshot UI testing is based upon is taking a screenshot of the current UI state as a bitmap. We compare this to a known good image pixel by pixel. If they match, the test passes. If they don't match the test fails.

For this technique to work, your UI must behave in a deterministic manner. That is, for a given start state, followed by a given set of events, the UI should be in a consistent state. If there are any random elements then it will be non-deterministic so the tests will be flakey. We'll touch on this more later!

ScreenshotComparator

As I have already mentioned, the foundation for this is `ScreenshotComparator.kt` from the Rally sample app. This renders the UI under test to a bitmap, saves it to the test device, and then compares it to a 'golden' file that is stored within `androidTest/assets` in your project. When you run this for the first time, you won't have a golden reference image. But after running and failing, the test device will have a set of screenshot images stored. These can be copied back to your development machine and saved to the `assets` folder. Now if you run the tests a second time they should pass.

This actually works quite nicely, but I had some additional requirements.

Firstly, I wanted to be able to have discrete test suites, each with its own set of golden images.

Secondly, I have multiple test devices which have different screen densities. Images capture from each will be of different sizes, so we cannot compare them. Therefore I need multiple golden images for different screen densities.

Thirdly, I want to manage space on my test devices a little better. The current implementation keeps adding images on the test device each time a test suite runs. For a comprehensive suite of UI tests on a large project, this may soon fill up the device.

Modifications

My modified version of `SnapshotComparator` looks like this:

```
/*
 * Copyright 2020 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.stylingandroid.compose.strikethru

import android.graphics.Bitmap
import android.graphics.BitmapFactory
import android.os.Build
import androidx.annotation.RequiresApi
import androidx.compose.ui.graphics.asAndroidBitmap
import androidx.compose.ui.test.SemanticsNodeInteraction
```

Categories
Select Category

Archives
Select Month

Social Media

Mark Allison at Mastodon

Recent Posts
Hiatus
Compose: UI Screenshot Testing
Compose: Strikethru Animation
Bluetooth Pairing
Compose: List / Detail – Testing part 2
Compose: List / Detail – Testing part 1
Compose – List / Detail: Foldables
Compose – List / Detail: Basics
Gradle: Version Catalogs
Jetpack Compose

Useful Links
Android Open Source Project
Android Weekly
Official Android Developer Site

Blogroll
Android UI Design Patterns
Cyril Mottier's Android blog
Grokking Android
Martin van Zuilekom's blog
Official Android Developers Blog
Reto Meier's Blog
Richard Hyndman's blog

Meta
Log in
Entries feed
Comments feed
WordPress.org

```
import androidx.compose.ui.test.captureToImage
import androidx.test.platform.app.InstrumentationRegistry
import java.io.File
import java.io.FileOutputStream

/**
 * Simple on-device screenshot comparator that uses golden images present in
 * `androidTest/assets`. It's adapted from
 * https://github.com/android/compose-samples/blob/main/Rally/app/src/androi
 *
 * Minimum SDK is O.
 *
 * Screenshots are saved on device in `/data/data/{package}/files`.
 *
 * Screenshot names will have the bitmap size included. This allows for diffe
 * to be used for different screen densities. You will need to ensure that go
 * appropriate size in the name is included for all supported densities.
 */
@RequiresApi(Build.VERSION_CODES.O)
fun assertScreenshotMatchesGolden(
    folderName: String,
    goldenName: String,
    node: SemanticsNodeInteraction
) {
    val bitmap = node.captureToImage().asAndroidBitmap()

    // Save screenshot to file for debugging
    saveScreenshot(
        folderName,
        "${goldenName}_${bitmap.width}x${bitmap.height}_${System.currentTimeM
        bitmap
    )
    val golden = InstrumentationRegistry.getInstrumentation()
        .context.resources.assets.open(
            "$folderName/${goldenName}_${bitmap.width}x${bitmap.height}.webp"
        )
        .use { BitmapFactory.decodeStream(it) }

    golden.compare(bitmap)
}

private fun saveScreenshot(folderName: String, filename: String, bmp: Bitmap)
    val path = File(InstrumentationRegistry.getInstrumentation().targetConte
    if (!path.exists()) {
        path.mkdirs()
    }
    FileOutputStream("$path/$filename.webp").use { out ->
        bmp.compress(Bitmap.CompressFormat.WEBP_LOSSLESS, 100, out)
    }
    println("Saved screenshot to $path/$filename.webp")
}

private fun Bitmap.compare(other: Bitmap) {
    if (this.width != other.width || this.height != other.height) {
        throw AssertionError("Size of screenshot does not match golden file
    }
    // Compare row by row to save memory on device
    val row1 = IntArray(width)
    val row2 = IntArray(width)
    for (column in 0 until height) {
        // Read one row per bitmap and compare
        this.getRow(row1, column)
        other.getRow(row2, column)
        if (!row1.contentEquals(row2)) {
            throw AssertionError("Sizes match but bitmap content has differen
        }
    }
}

private fun Bitmap.getRow(pixels: IntArray, column: Int) {
    this.getPixels(pixels, 0, width, 0, column, width, 1)
}

internal fun clearExistingImages(folderName: String) {
    val path = File(InstrumentationRegistry.getInstrumentation().targetConte
    path.deleteRecursively()
}
```

Both the captured images and the golden images are within distinct folders. The caller specifies the folder name. We load the golden images from matching folders. This fixes my first requirement of having different golden images sets for different test suites.

The saved screenshot filename includes the dimensions of the captured bitmap. These golden image filenames also contain the dimensions. When the tests run on different screen densities, t comparison uses the appropriate golden images. This solves my second requirement of tests running on different test devices. However, it does mean we need that golden images for all supported densities.

Finally, there is an additional function that will delete a named directory and its entire contents. This satisfies my storage space requirement.

Managing Storage

In our Test Suite, we add a @BeforeClass function which will run once, before any of the tests:

```
class StrikethruIconTest {

    @get:Rule
    val composeTestRule = createAndroidComposeRule()
```

```
    companion object {
        private const val testTagName = "StrikethruIcon"

        @BeforeClass
        @JvmStatic
        fun clearExistingImagesBeforeStart() {
            clearExistingImages(testTagName)
        }
    }
    .
    .
    .
}
```

The `TEST_TAG_NAME` will be used in multiple places in the following tests. We'll use this for the folder name for the golden and saved images. The `clearImagesBeforeStart()` method removes any existing images before we start.

We can use different folder names for different test suites. By making each suite responsible for its own housekeeping, we can run them separately, and they won't interfere with each other's stored images.

A simple snapshot test

Let's start with a test that about as simple as it gets:

```
@Test
fun GivenAStrikethruIcon_WhenItHasDefaultStates_ThenTheStateHasNoStrikethru()
    composeTestRule.setContent {
        StrikethruIcon()
    }

    assertScreenshotMatchesGolden(TEST_TAG_NAME, "no_strikethru", composeTest
}
```

This sets up the content as a single `StrikethruIcon` and calls the screenshot matcher.

Using testTag

Next, we'll look at a couple of slightly more complex tests which validate the UI state following specific events:

```
@Test
fun GivenAStrikethruIcon_WhenWeTapOnItOnce_ThenTheStateChangesToStrikethru()
    composeTestRule.setContent {
        StrikethruIcon(modifier = Modifier.testTag(TEST_TAG_NAME))
    }

    composeTestRule.onNode(hasClickAction() and hasTestTag(TEST_TAG_NAME))
        .performClick()

    assertScreenshotMatchesGolden(TEST_TAG_NAME, "full_strikethru", composeTe
}

@Test
fun GivenAStrikethruIcon_WhenWeTapOnItTwice_ThenTheStateChangesToNoStrikethru
    composeTestRule.setContent {
        StrikethruIcon(modifier = Modifier.testTag(TEST_TAG_NAME))
    }

    composeTestRule.onNode(hasClickAction() and hasTestTag(TEST_TAG_NAME))
        .performClick()
    composeTestRule.onNode(hasClickAction() and hasTestTag(TEST_TAG_NAME))
        .performClick()

    assertScreenshotMatchesGolden(TEST_TAG_NAME, "no_strikethru", composeTest
}
```

These verify whether the strikethru is showing after a single tap, and two taps. What is interesting here is the use of the `testTag` modifier on each instance of `StrikethruIcon`. This makes finding modes within the UI hierarchy much easier. The `hasTestTag(TEST_TAG_NAME)` matcher finds any UI components which have the specified tag set.

Mid-animation state

We can go further still. The previous tests are validating static UI states. But what if we want to test that the animation is actually running? It's actually possible to get a snapshot at a specific point during the animation:

```
@Test
fun GivenAStrikethruIcon_WhenWeTapOnItOnce_ThenTheStateIsCorrectAfter50ms() {
    composeTestRule.setContent {
        StrikethruIcon(modifier = Modifier.testTag(TEST_TAG_NAME))
    }

    composeTestRule.mainClock.autoAdvance = false
    composeTestRule.onNode(hasClickAction() and hasTestTag("StrikethruIcon"))
        .performClick()
    composeTestRule.mainClock.advanceTimeBy(50)
```

```
    assertScreenshotMatchesGolden(TEST_TAG_NAME, "half_strikethru", composeTe
}
```

The highlighted lines are the special sauce here. Animations are essentially values that change over time. `composeTestRule.mainClock` allows us to control the system clock used to drive these animations. By default, it runs normally, but by turning off `autoAdvance` we can control it manually. Here we perform a click and then advance time by 50ms.

When we perform the screenshot match, the animation is 50ms in. It will look something like this:

This is actually one of the golden images from my test suite. The strikethru is only partially drawn. That's how far it got after 50ms. However, this is deterministic. We know precisely how the UI should look after 50ms. The golden image reflects this.

Non-deterministic states

I originally got this code working with Compose `1.0.0-beta08` and things worked perfectly. I then updated to `1.0.0-beta09` and all of my tests broke, as detailed below. I think that there may have been a bug introduced in `1.0.0-beta09`, because the behaviour went back to normal in `1.0.0-rc01`.

I'll leave my original write-up below as it still makes a valid point about screenshot testing. If your UI behaves in a non-deterministic way, then screenscot tests will be flakey.

UPDATE: Actually it's not quite that simple. Next week's post will go in to more detail, and offer a couple of solutions.

~~Beware the ripple~~

~~I need to share one real gotcha. I had developed these tests using Compose `1.0.0-beta08`. They were all working fine, and passing without flakiness. Then I updated to Compose `1.0.0-beta09` and they started breaking.~~

~~The cause of the issue was actually a bug in beta08. Material ripples were not working in beta08. They were fixed in beta09, and broke my tests! The issue was that the ripples themselves appear to be non-deterministic. Their appearance is not consistent for the same inputs. I suspect that this may have something to do with the sparkles added in Android 12 Betas which may have a random element to them.~~

~~This inconsistent appearance caused the pixel by pixel comparison to fail. Even trying tricks like advancing the `mainClock` to a time long after the ripple should have finished didn't help.~~

~~After a fair bit of head-scratching, I decided on removing the theme wrapper from the test UI. The reason this works is that the ripple is part of the `MaterialTheme` that my theme was extending. By removing the Material Theme, the ripple no longer appears. For those that were wondering, that's why my golden image is black, it is not themed.~~

~~Once again, this shows how Compose makes life much easier for us. I had previously included `StrikethruTheme {}` wrapping the `StrikethruIcon` in my tests. Simply removing this enabled me to strip out the theming~~

Understanding what we're testing

~~This raises an important issue. It is important to understand that they are testing the strikethrough behaviour independently of any theming. Whereas they previously tested that the theming was also correct. If we still want to test that, then we can create additional tests which will use the theme, but not perform actions that create ripples in the snapshots. However, having dedicated theme tests might be better than do it at component level.~~

Although I had to temporarily remove the theming while using Compose `1.0.0-beta09`, it has now been re-instated. So my tests are more complete as a result. Not only do they test the strikethrough states, and the animation. But they also verify that the theme is being appleid correctly. Specifically the tinting of the icon is being tested.

Conclusion

This kind of testing would have been much harder using the traditional `View` system. Although we could have used the concept of screnshot testing, things like controlling the animation though the `mainClock` are simply not possible. Also, stripping out the theme was a case of deleting a few lines here, but that would have been much harder using `Views`.

Many thanks to Jose Alcérreca for inspiring this. And to Nick Butcher for offereing some comments.

suggestions, and improvements. Also, thanks to Stojan Anastasov for suggesting using `deleteRecursively()` instead of walking the file tree manually to delete the existing screenshots.

The source code for this article is available here.

Share this:

Related

Compose: List / Detail – Testing part 1
June 11, 2021
In "Compose"

Compose: List / Detail – Testing part 2
June 18, 2021
In "Compose"

Jetpack Compose
May 14, 2021
In "Compose"

ANDROIDDEV

« Previous Post
Compose: Strikethru Animation

Next Post »
Hiatus

About Mark Allison

You Might Also Like

Bluetooth Pairing
June 25, 2021

Compose: List / Detail – Testing part...
June 18, 2021

Compose: List / Detail – Testing part...
June 11, 2021

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *

Name *

Email *

Website

Notify me of follow-up comments by email.

Notify me of new posts by email.

Post Comment

This site uses Akismet to reduce spam. Learn how your comment data is processed.