INTERNET ARCHIVE
Wayback Machine
https://www.originate.com/thinking/multidimensional-snapshot-testing
Go
AUG SEP OCT
1 capture
20 Sep 2022
20 2021 2022 2023
About this capture

Multidimensional Snapshot Testing

By allenwu

Our mobile team recently built a pair of content-driven iOS and Android apps for a large mixed-media company. Given how rich and extensive their media collection is, the designers provided us *hundreds* of cell layouts and rules for the media types as the basis for a dynamic and lively browsing experience. This presented us with an interesting architectural and design challenge.

We'll focus on the iPad app in this post and how a few simple techniques helped us build and test our cells.

The Challenge

The app revolves around several primary types of media content—photos, videos, audio clips, and articles (think: digital magazine). These pieces of content are collected and curated by the backend server, which our app consumes and presents to users as a feed of cells arranged in grids.

Based on several parameters that we'll be exploring, the cells need to adapt to a wide variety of physical form-factors and variations. We'll refer to these as *dimensions* in this post.

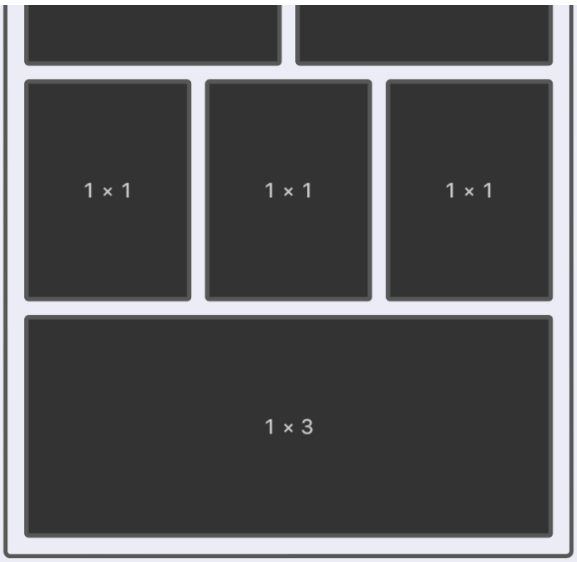
Dimension #1: Media Type

A cell can represent one of a various number of media types (photo, video, article, etc...).

Dimension #2: Cell Size

Cells must occupy one of a handful of specific sizes (dimensions denoted in *rows* × *cols* notation) as they are laid out on a `3 x 3` grid that spans the screen. Some combinations of `r x c` are invalid.

Here's an example of a screen full of cells:



Dimension #3: Cell Display

Cells also have a *display* property, which specifies a particular design variation of a cell.

For example, a photogallery cell might be rendered in several different ways: with two thumbnails side-by-side, or with 1 large thumbnail on the left and 2 smaller thumbnails on the right, or with 3 thumbnails etc... in all cases, the cells represent the same underlying content.

Most media types support a handful of display variations. However, some, such as a textual quote, may not support any variations.

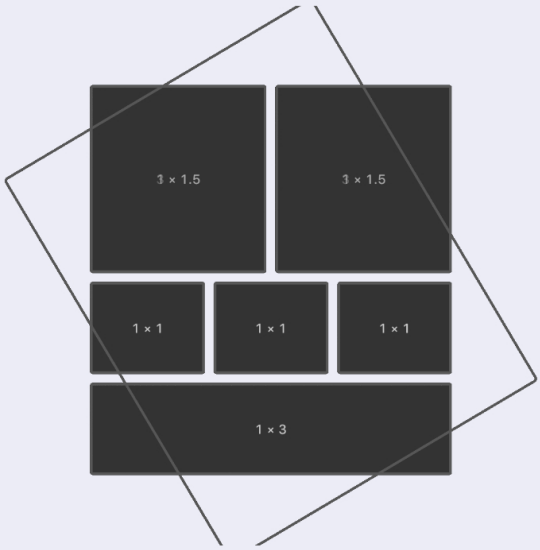
display: double

Dimension #4: Device Orientation

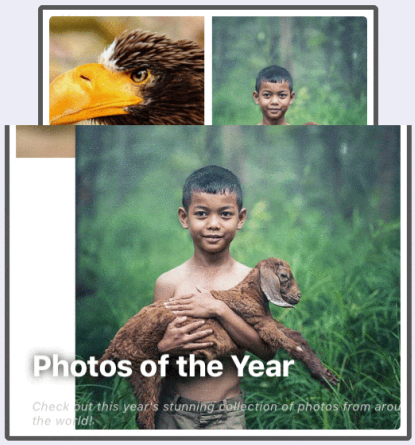
Because iPad users tend to use their devices in both portrait *and* landscape (moreso than iPhone users), we also need to consider the device orientation before we can render the cell.

Some display types, depending on the cell size they're embedded in, may undergo further layout changes depending on the device orientation.

In the following animation, notice how the *(1 × 1.5)* cell morphs into a *(3 × 1.5)* cell to fill the screen height once rotated in landscape. Because of the significantly altered aspect ratio, it makes sense for the cell to consider the device orientation.



Not all `display` variations consider the device orientation. The photogallery cell does, but the quote cell doesn't (simple font scaling and image stretching doesn't warrant any additional logic).



Dimension #5: Device Size

And finally, the actual device being used can introduce our final bit of variation. The iPad Pro 12.9" is significantly larger than the iPad 9.7", giving us extra screen real-estate to work with.

Goals

Given the apparent design complexity of our cells, we needed to come up with an efficient process to verify that all

our cells are built correctly, ideally as part of our CI and QA process.

With our problem space now defined, we have a few goals in mind.

1. How can we easily verify that all the cell variations look correct?
2. How can we reduce the complexity of handling these variations?

Our Solution, Snapshot Testing

We'll break down the problem a bit and examine our dimensions closely. For any given cell, we have a combination of:

```
1. media
    - [ article | music | photogallery | quote | soundbite | tweet | videoclip ]
2. display
    - [ standard | double | tripleLeft | tripleRight | tripleTop | quad1st4th | quad2nd3rd ]
3. cell size
    - [ 1x1 | 1x1.5 | 1x2 | 1x3 | 2x2 | 2x3 | 3x1.5 | 3x3 ]
4. device orientation
    - [ portrait | landscape ]
5. device size
    - [ iPad_9.7, iPadPro_10.5, iPadPro_12.9 ]
```

If every combination is possible, this results in $7 \times 7 \times 8 \times 2 \times 3 = 2{,}352$ unique cells to build! Some combinations aren't valid though, so we can reduce the total set down to a few hundred—still a non-trivial problem.

Furthermore, our cells are highly visual in nature. Exact designer-specified proportions, fonts, dimensions, and subview layouts need to be built for our 5 dimensions. UI tests could be written to assert, *for each combination*, that the title label contains the right text, the background is the correct color, the thumbnail is the correct size, etc... but this would be an exhausting, expensive, and error-prone effort.

Enter: Snapshot Testing!

Using a concept called snapshot testing, we can replace code-based layout tests with visual tests performed by our eyes—or even better—a machine's "eyes". ????

How It Works

The initial execution of a test case creates a screenshot of the view we're interested in. It's then saved to disk as a PNG "reference image". Subsequent executions of the same test case compare a newly generated screenshot with the saved reference image. The test case fails if the images aren't identical.

(Though this is generally referred to as snapshot testing, *screenshot* testing might be a more accurate term.)

Within our test suite, we utilized a well-crafted loop to iterate through all combinations of the 5 cell parameters, so that we could quickly and *exhaustively* generate views for all valid combinations, and utilize iOSSnapshotTestCase to take screenshots of the fully-rendered views.

Note: This exhaustivity is an important piece of the puzzle! Using Sourcery, a Swift code generation tool, we are provided a compiler-guaranteed mechanism for iterating through all enumeration cases, which ensures that any new scenarios we add are automatically tested and accounted for.

At a high-level, our cell tests look like this:

```
import XCTest

import iOSSnapshotTestCase
class CellSnapshotTests: iOSSnapshotTestCase {


 func testCells() {
 Cell.allCombinations.forEach { media, display, cellSize, orientation, deviceSize in
 // 1. Instantiate and render cell view with these 5 dimensions
 // 2. Compare rendered view with saved reference image, if in test mode
 // or...
 // Save rendered view to disk as reference image, if in record mode
 // (This loop will iterate a few hundred times. One iteration for each valid combination.)
 }
 }
 }
```

And here's a collection of images automatically generated by our snapshot test for the completed photogallery cell:

05 / 34

media: photogallery
size: 1 × 1.5
display: triple-right
orientation: portrait

These snapshot tests help us answer the two main questions we posed earlier.

1. How can we easily verify that all the cell variations look correct?

1. *Quick visual inspection* — They allow the developers and designers to quickly see how all combinations of the cells look.
2. *Regression testing* — They act as the reference image (or golden file) for testing. Once we have built up a test suite of hundreds of cell snapshots, any code change that results in a visual diff of any snapshot image will be flagged as a failed test, allowing us to quickly identify visual bugs and confirm visual changes.
3. *Code review aid* — The reference images are checked into the repository. Therefore additions, removals, and edits of images are immediately visible to reviewers of a pull request. GitHub and other tools even provide visual diffs to highlight exactly what changed in the images, making code review a pleasant experience. It takes, say, **10 seconds** to look at a snapshot and comment that the colors and font size are way off, whereas pulling the feature branch, compiling, running the simulator, then poking around the app to find the right cell to comment on might take **5 minutes**! Multiply these savings across a large team building hundreds of cells, and we can save weeks of developer time over the course of a project.

2. How can we reduce the complexity of handling these variations?

1. *Shorten development cycles* — Setting aside the compilation time (which is already a bit of an issue in Swift), viewing incremental changes during iOS development generally requires a complete simulator refresh. This makes development cycles quite expensive, especially when dozens or hundreds of cells will be updated on the latest build cycle. By creating snapshots, we can visually inspect the cells directly (and in all their hundreds of variants), eliminating the lengthy time needed to actually look for the cells in the simulated app.
2. *Efficient cross-team iteration* — Designers could quickly scan through all .png files and pass along feedback to the development team without ever running the app.
3. *Keeping the development team in sync* — The snapshots also help the team stay in sync during development. Once a project reaches a certain size, it becomes increasingly difficult for all developers to know each and every feature built by the team. In addition to lowering the barrier for code review for the cells, the snapshots provide a near-frictionless way to see what the rest of the team is building.

Caveats

Snapshot testing, despite all the benefits it afforded us, was not foolproof. There were numerous occassions where views that looked a certain way in the app were rendered sufficiently different in the snapshots to make the test useless. These were warts with the implementation, and not the technique though. As the tooling and frameworks mature and our experience with the technique grows, snapshot testing will become more reliable and is already a worthy addition to any mobile project.

Sourcery, which was critical to enforcing the exhaustivity of our tests, had issues too. Some upstream bugs caused sporadic build failures, which was a source of frustration that we had to work around. The good news is that beginning with Swift 4.2, the main Sourcery feature we rely on is built into the language.

Conclusion

The "secret sauce" behind this post is really just ketchup mixed with mustard. We took two simple concepts—snapshot testing and exhaustively iterable types—to create a delightfully tangy system for managing complexity.

- We drastically minimized code review effort, saving precious time and keeping productivity and quality high.
- We created a system that helped all teams—developers, designers, and the product team. The snapshots were visible to all and each team could use them in their own ways as necessary.

Achieving this required some diligence by the team, but by being thoughtful and organized with our code and using a few third-party tools, we were able to successfully build an extensible and safe system to manage hundreds of cell layouts and ultimately ship a lively and beautiful product to the App Store! ????

Launch Your Journey

Complete this form to get in touch with our team. Let's talk about how Originate can work with you to improve the experiences you deliver across your entire company.

Name*
Last Name*
Email Address*
Phone Number
Role
Message

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.

SUBMIT

WHITE PAPER: 5 Key Initiatives To Transform Your Customer Experience in 2022

Download Now

Originate Privacy Policy (800) 352-2292 hello@originate.com

Copyright © 2022 Originate, Inc