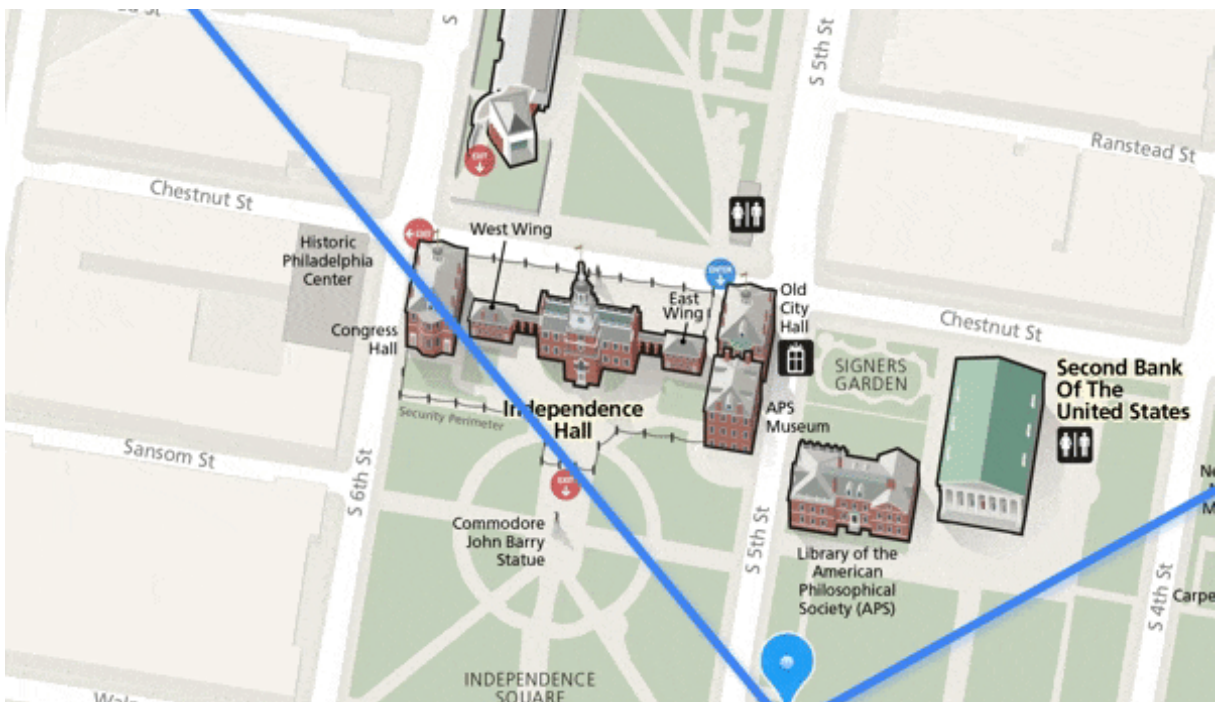

release v3.0.1

This project isn't maintained anymore. It is now recommended to use <https://github.com/peterLaurence/MapView>.

MapView is maintained by Peter, one of our main contributors. MapView is an highly optimized Kotlin version (which can be used from Java projects) that's very cool and fast. It's 100% production ready and open source. Peter uses this widget in his app at <https://github.com/peterLaurence/TrekMe/>.

TileView: Version 4

The TileView widget is a subclass of ViewGroup renders and positions bitmap tiles to compose a larger, original image, often one too large to display normally.



Vesion 4 is effectively the promotion of version 3 from beta to production-ready, fixing the issues brought up by you, the users. While normally this'd probably be done using a series of minor version bumps, and this might be 3.4.11, there was a change to the base package name, and this requires a major change according to semver, since it's a breaking change for past versions.

Also of note, the universal (2-D) ScrollView class, and related classes like ScalingScrollView (which is a subclass of the 2D universal ScrollView, but also manages scaling and scale gestures) is now it's own repository: <https://github.com/moagrius/ScrollView>, and available with `implementation 'com.moagrius:scrollview:1.0.3'`. It is now included in the TileView project using normal gradle

dependency operations, specifically using `api` rather than `implementation` but otherwise similar to other dependencies and is being pulled down from jcenter and reconciled normally.

Demos for `ScrollView` are in the `ScrollView` repo. Demos for `TileView` are in this repository.

Feel free to use `ScrollView` as a standalone widget if you don't need image tiling.

Change Log

4.0.7 (most recent)

- Fixed bug with `MarkerPlugin` where markers clipped when scaled beyond 100%.

4.0.5 (most recent)

- Save and restore instance state properly implemented.
- Removed a bug that produced an increasing number of delayed callbacks from a `Handler`. This is a serious memory leak and all users on versions 4.0.3 and 4.0.4 should immediately upgrade to 4.0.5.

4.0.4

- Update `ScalingScrollView` to version 1.0.10, which provides additional methods and method exposure (many `private` access methods became `protected`)

4.0.3

- You can scale greater than 1f now without flicker, and without clipping.
- Remote images should decode now with a much higher rate of success, nearing 100%.
- Marker plugin API has been updated and should now work properly (see `TileViewDemoAdvanced` Activity).
- `LowFidelityBackgroundPlugin` now scaled and positioned appropriately with the `TileView` host.
- The `COVER` and `CONTAIN` `MinimumScaleModes` should work properly now.
- Upgraded from `com.moagrius:scrollview:1.0.4` to `1.0.9`

TileView

Very large images in Android will often result in an `OutOfMemoryError`. Memory is finite, and bitmaps take a great deal of it. `TileView` solves this by stitching together small pieces of the image (tiles) and displaying them reconstructed in the area of the screen visible to your user.

Out of the box, the `TileView` will manage the tiled image, including subsampling when needed, flinging, dragging, scaling, and multiple levels of detail.

Additional plugins are provided to allow adding overlaying Views (markers), info windows, hot spots, path drawing, and coordinate translation.

Version 4 is fairly young. If you need a stable version, the last release of version 2 is 2.7.7, and is available as a branch, here: <https://github.com/moagrius/TileView/tree/version-2.7.7>, or from the releases page. You can also get it from gradle using the old namespace: `implementation 'com.qozix.tileview:2.7.7'`

No further work will be done on version 2.

This is truly open source. I'm very happy to accept improvements or bug fixes - no caveats; create an issue, branch, and create a PR. While I'm not super interested in finding bugs and don't want the issues page to become a wasteland (I already know what most of them are, and will start to document them as this make its way into the wild), I am very interested in fixing those bugs - if you have the time to locate and correct a bug, please open a PR.

Installation

Add this to your app module's `build.gradle`.

```
1 implementation 'com.moagrius.tileview:4.0.7'
```

Quick Setup

1. Tile an image into image slices of a set size, e.g., 256x256 instructions // TODO
2. Name the tiles by the column and row number, e.g., 'tile-1-2.png' for the image tile that would be at the 2nd column from left and 3rd row from top.
3. Create a new application with a single activity ('Main').
4. Save the image tiles to your assets directory.
5. Add the latest version to your gradle dependencies.
6. In the Main Activity, use this for `onCreate`:

```
1 @Override
2 protected void onCreate(Bundle savedInstanceState) {
3     super.onCreate(savedInstanceState);
4     TileView tileView = new TileView.Builder(this)
5         .setSize(3000, 3000)
6         .defineZoomLevel("tile-%d-%d.png")
7         .build();
8     setContentView(tileView);
```

That's it. You should have a tiled image that only renders the pieces of the image that are within the current viewport, and pans and zooms with gestures.

Note that String replacements for rows and columns is not required - you can supply literally any Object instance to a DetailLevel, and a BitmapProvider implementation can use that Object to generate a Bitmap instance however you want.

What Changed

As a user, the biggest things you'll notice are:

1. You no longer need redundant tile sets or detail levels. If your image doesn't change the details (e.g., show different images or labels at different zoom levels), you don't need to create tile sets besides the original, full size one. The program will now use subsampling to do this work for you without any setup on your part: one call to `defineDetail(anyObject)` is sufficient.
1. No more `BitmapProvider`. We're just doing way too much management of Bitmaps, including re-use and caching, to allow any stray Bitmap to wander in. This may change in the future, but the replacement for now is `StreamProvider` - basically the same thing but you just return an `InputStream` instead of a `Bitmap`, and we take care of the rest.
1. Greatly improved bitmap management. For details, see [How It Works](#).
1. Touch events are now managed appropriately, rather than blindly consumed. We now use the same logic as the framework's `ScrollView`, so you can assign normal `OnClickListeners` or similar gesture management devices on things like markers and callouts, without interrupting drag or fling operations.
1. Plugin architecture. Only sign up for things you need - if you don't need markers, don't install the plugin, and keep things snappy and simple. Check [TileViewDemoSimple](#) for a very bare implementation, and [TileViewDemoAdvanced](#) for a more dressed up version.
1. Much smaller codebase. Almost all the magic now happens in either `Tile` or `TileView` - if you understand those 2 classes, you understand the majority of the project.
1. Decomposition. This takes a little explanation. There are now 3 major, public widgets: `ScrollView`, `ScalingScrollView`, and `TileView`. Each inherits from the last. You'll notice the demo module has `Activities` for each of these classes.

ScrollView `com.moagrius.widgets.ScrollView` is a clone of AOSP's `ScrollView` and `HorizontalScrollView`, with some inspiration taken from `RecyclerView` and `GestureDetector`. That means the APIs will be very familiar to anyone who's used the framework-provided `ScrollView`. The big difference here is that `com.moagrius.widgets.ScrollView` functions along both axes, without configuration. For example, if you have a layout that matches parent width and expands vertically, `ScrollView` will scroll it vertically - the converse holds. If

you have a layout that larger in both directions than the parent, `ScrollView` will scroll in any direction.

Again, no configuration is required - there is no `orientation` attribute - it does it's best to figure out what makes sense.

ScalingScrollView `ScalingScrollView` extends `ScrollView` and adds scaling APIs, as well as built-in scaling gesture managent (pinch and double tap to zoom). By default, it will visually scale its content, but that can be disabled with a single setter `setShouldVisuallyScaleContents` (`false`), and you can manage your content with a porportional scaled value. Similarly, it will lay itself out to the current scale value, unless that too is disabled `setWillHandleContentSize` (`true`)

At some point, `ScalingScrollView` will be it's own repository, and will be a dependency of `TileView`.

`TileView` inherits from `ScalingScrollView`, and adds tiling functionality.

From here down may be a bit dry but might be interesting to people interested in contributing to the process.

How It Works

Problem You have an image too large to display with an `OutOfMemoryError`.

Solution Chop the image into tiles and reconstruct them to fill the viewport, and only the viewport. As the user scrolls (or scales), aggressively recycle any bitmap data that is not visible to the user at any given time, and render any new "tiles" that are not within the viewable area.

Basic operation Use the width and height of the view that contains this image, plus it's scroll position, to determine space we call the viewport.

Divide the corners of the viewport (left, top, right, bottom) by the size of the cell, which provide the start and end row and column. Round down the left and top and round up the right and bottom to make sure there are not empty patches - as long as any pixel of a tile is visible in the viewport, it must be rendered (one reason - among many - to carefully consider tile size, and not default to as large as possible).

From start column to end column and start row to end row are the tiles (imagine a 2D array) that should be rendered at that moment.

For example, let's consider...

- image is 10,000 square
- viewport is 1080 by 1920
- viewport is at scroll position (1000, 500)
- so our computed viewport is (1000, 500 - 2080, 2420)
- out tiles are 256 square

so grid is

1. start column $1000 / 256 = 3$
2. end column is $2080 / 256 = 9$
3. start row is $500 / 256 = 2$
4. end row is $2420 / 256 = 10$

Iterate between those points...

```
1 for (int i = grid.columns.start; i < grid.columns.end; i++) {  
2   for (int j = grid.rows.start; j < grid.rows.end; j++) {
```

And your grid looks like this:

```
1 (03, 02), (04, 02), (05, 02), (06, 02), (07, 02), (08, 02), (09, 02)  
2 (03, 03), (04, 03), (05, 03), (06, 03), (07, 03), (08, 03), (09, 03)  
3 (03, 04), (04, 04), (05, 04), (06, 04), (07, 04), (08, 04), (09, 04)  
4 (03, 05), (04, 05), (05, 05), (06, 05), (07, 05), (08, 05), (09, 05)  
5 (03, 06), (04, 06), (05, 06), (06, 06), (07, 06), (08, 06), (09, 06)  
6 (03, 07), (04, 07), (05, 07), (06, 07), (07, 07), (08, 07), (09, 07)  
7 (03, 08), (04, 08), (05, 08), (06, 08), (07, 08), (08, 08), (09, 08)  
8 (03, 09), (04, 09), (05, 09), (06, 09), (07, 09), (08, 09), (09, 09)  
9 (03, 10), (04, 10), (05, 10), (06, 10), (07, 10), (08, 10), (09, 10)
```

As long as there's only the one size, scale is uniformly applied - just multiple tile size in the previous math by the current scale, and your grid remains intact.

Scaling At every half way point, the image is at 50% or smaller, so subsample it to save memory: <https://developer.android.com/topic/performance/graphics/load-bitmap#load-bitmap>

Remember that each subsample (representing half the size) will actually quarter the amount of memory required for the bitmap. Do this at every half: 50%, 25, 12.5, 6.25, 3.625, etc.

When you add another detail level, things get a little more complicated, but not terribly...

Detail/Zoom Levels

-
- Subsampling starts over at the last provided detail level, so it's no longer directly taken from the scale
 - Your grid now has to be "unscaled" - now it's tile size * scale (as it was before), but now must be unscaled by an amount equal to the emulation of the detail level.
 - This unscaled value is effectively the inverse of the scale, so 50% scale would be times 2, 25% would be times 4, 12.5% would be times 8
 - However, this is a constant value, so anything from 50% to 26% would be times 2, so a better descriptor might be "zoom left shift one" or (zoom « 1)
 - This works fairly well as is at this point, but on large screens with high density, you end up with *lots* of very small tiles, unless you provide a ton of detail levels.
 - If you have a single detail level and let the program subsample for you, the total memory footprint stays (about) the same, but the number of times the disk or caches are hit is very high, and this can appear very slow
 - To remedy this, we use "patching". With patching, things get a lot more complicated.

Patching

- "Patching" is basically grabbing as many subsampled tiles as needed to create a full sized tiles, stitching them together into a single bitmap, and stuffing that in the cache
- A very important number here is the "image subsample". this is distinct from the "zoom sample" described above
- The image subsample is derived from the *distance* you are from the last provided detail level. so if you have only 1 detail level (0) and you're at 20%, you're at image subsample 4 (50% would be 2, 25% would be 4, 12.5% would be 8, etc)
- To do that, your grid math has to change a little. you now have to round down to the nearest image subsample on the west and north, and round up to the nearest image subsample on the south and east, and skip a number of columns and rows equal to the image subsample
- So now your viewport computation becomes very different:

```
1 for (int i = grid.columns.start; i < grid.columns.end; i += imageSample) {  
2     for (int j = grid.rows.start; j < grid.rows.end; j += imageSample) {
```

And for each of those tiles, we grab the last good detail level and fill in the blanks, so tile (0,4) would draw it's neighbors:

```
1 (0,0), (0,1), (0,2), (0,3)  
2 (1,0), (1,1), (1,2), (1,3)  
3 (2,0), (2,1), (2,2), (2,3)  
4 (3,0), (3,1), (3,2), (3,3)
```

This allows us to then save the “patched” bitmap tile to both a memory and a disk cache, so we’re not decoding very large files and discarding excess data after the first time.

The next thing that comes up is when changing between zoom levels. If we just swap out levels, the screen will be blank for a second, regardless of whether that’s a defined, supplied detail level, or just subsampling the tiles. Since `View.onDraw` uses a `Canvas`, we need to redraw the entire thing each time invalidation occurs. That means if you switch between zoom level, the canvas will clear the old tiles before drawing the new tiles. This behavior is generally good (and what ensures that we’re generally only going to drawing as much bitmap data as we need to fill the screen, even if we do a bad job of cleaning up), but in this case the visual effect is jarring.

To remedy this, we:

- Populate a `Region` instance with the virtual viewport (scaled)
- For each fully decoded tile in the current (the detail level we just entered), punch a hole in that `Region`. What’s left are pixels that will remain unfilled after we draw current tiles.
- For each fully decoded and drawn tiles from the previous detail level - the detail level we’re exiting - we use `Region.quickReject` to determine if that previous tile intersects any part of the virtual viewport that will not be filled with current tile pixel data.
- If a previous tile does intersect this unfilled region, we need to draw it.
- If a previous tile does *not* intersect this unfilled region, that means we have completely covered that area with pixel data from the new level, and can both destroy the previous tile (and it’s bitmap), as well as remove it from the list of previous tiles we’re testing.
- Since we remove every previous tile that’s covered by current tiles, this set will become empty once the current viewport is filled with current pixel data.

So all the preceding “works” pretty well, but decoding the same data over and over is a lot of work that we can probably skip if we’re smart about it.

Bitmap Caching and Re-use Emphasis on **and reuse** - the reuse bit is as important as (maybe even more than) the traditional caching piece.

The default providers assume we’re reading tiles from disk already, so the only thing that goes into disk-cache are the patched tiles, mentioned above (this behavior can be modified, if for example you’re reading tiles from a server). Everything, however, is subject to a memory cache. The default is one-quarter of available RAM, but the more you can spare for this, the better your performance will be. This serves as both a straight bitmap cache, and a pool of re-usable objects: **When we first go to decode a tile, we check if that exact bitmap is in the memory cache - if it is, great, use it - if it’s not, grab the oldest bitmap in the cache and re-use the underlying bitmap (`BitmapFactory.Options.inBitmap`)**

Reusing bimap prevents frequent, large memory allocation and de-allocation, and can greatly improve the perceived performance of your app. I’ll let Colt explain: https://www.youtube.com/watch?v=_ioFW3cyRV0

Threading Threading is accomplished using a dedicated `ThreadPoolExecutor`. It's basically a fixed size pool with a pool size equal to the number of cores on the device (so 2 or 4 commonly, on newer phones as much as 8). There's a 0 keep-alive and tasks are queued with a `LinkedBlockingQueue`. A custom factory provides `Thread` instances that use `Thread.MIN_PRIORITY` so we don't consume resource the main thread needs to perform nice scrolls and flings. Going further on this, each `Tile` (which is a `Runnable` submitted to the `ThreadPoolExecutor` also calls `Process.setThreadPriority(Process.THREAD_PRIORITY_LOWEST)`; which gives a larger range than `Thread.setPriority`, and this is actually very noticeable. On a real Nexus 6, the jank without call to `setThreadPriority` isn't terrible but is obvious, and immediately remedied with the call just mentioned.

This general approach is very similar to what you'd get with `Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors())`.