MapLibre Native Metal Status Report: August 2023

This is the sixth status report for the MapLibre Native Metal project. For a deeper dive into the project's background, consult the first report.

Overall the status is **Green.** We are on track for a Metal implementation by the end of Q3, and to finish the work at the end of November.

Background

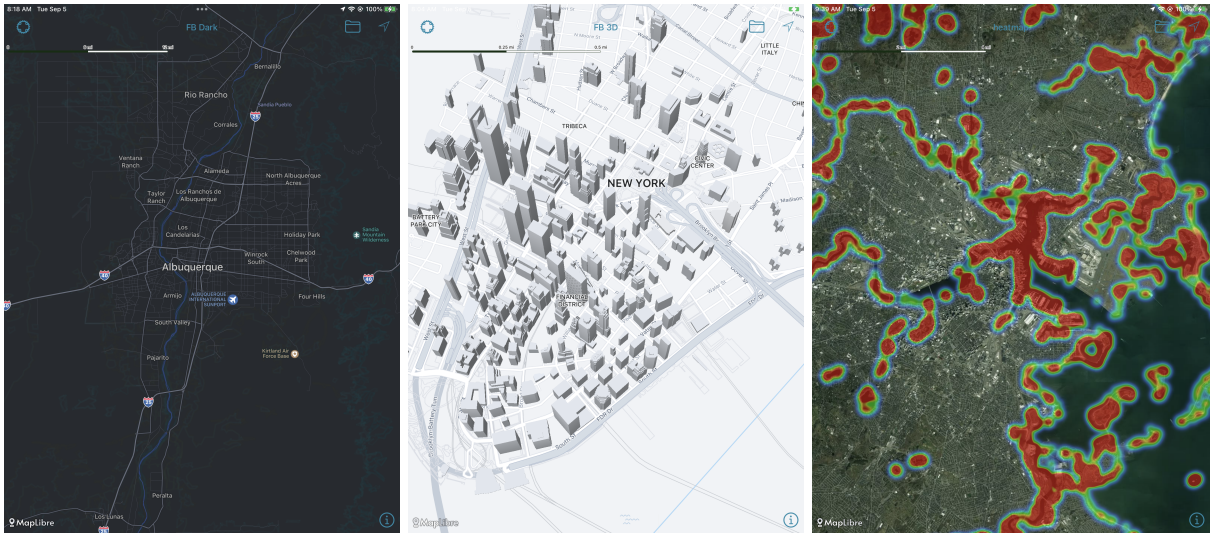
As a quick reminder, we're updating the MapLibre Project to support Metal for iOS and generally modernize the renderer.

We've wrapped up the first phase, Renderer Modularization with the acceptance of the merge by the community. The team has made great progress on the second phase, the Metal Port. We'll dig into the details below.

Status

We're **green** and will comfortably meet the Q3 goals, which are discussed later.

The theme for August's report is "Steady on".



Most layers are now working in Metal. There are a few exceptions, which we'll mention, and a substantial amount of work in optimization and in passing the rendering tests.

[In reference to Alan from Grab] We're excited both to have Grab contribute successfully as well as having an external developer come in at this point and help prove out the architecture.

Q3 Goals

From the June report we introduced a few Q3 goals. Let's see how we're doing.

- **Finish the Mega Merge**
  Accepted by the community and done.

- **Direct Version of Metal Port**
  Nearing functional completeness.

- **Equal Performance**
  Pretty close.

- **Try it out with a big user**
  We'll start engaging with potential users shortly.

We're on track to meet all of the Q3 goals.

Mega Merge

We went through a fairly elaborate process to merge the results of the first phase into the MapLibre repo. If you're interested, that's detailed in the last two reports. The work is now done and the changes accepted to the main branch by the maintainers.

At some point soon we'll turn on the new rendering path and rip out the old rendering code. The new rendering is now passing all the relevant tests, so that decision will largely be driven by the maintainers.

Optimization

When the project started we agreed to a number of metrics, of which two are essential.

- Don't make the toolkit slower
- Don't make the toolkit bigger

These aren't hard and fast, of course. A working Metal port that's slightly bigger is not the end of the world. But we do take them seriously and we're putting in work to address them.

First was frame rate. We didn't want to be any slower, so we revived the rendering speed tests and found that we were slower, by a factor of 4 in some cases. That suggested some room for improvement.

In the examples below, the "**Old Renderer**" is the OpenGL version from before we started and "**New Renderer**" is the OpenGL renderer we built in the modularization phase. The test cases focus on dense areas, mostly cities, and start out like so in the legacy source code.

Case	**Old Renderer**
Paris	59.9 fps
Paris2	59.7 fps
Alps	59.8 fps
US East	59.8 fps
Greater LA	59.9 fps
SF	41.3 fps
Oakland	59.8 fps
Germany	59.7 fps
Average	*57.5 fps*

The FPS numbers come from a very old iPad, otherwise they'd max out at 60fps and be useless, though we do have ways of scaling performance back on newer devices. At this phase of the project we're just looking to not make things worse. We'll highlight performance in a different way for Metal.

Most of these test cases aren't terribly useful, so we focus on the ones that might tell us something. In this case San Francisco.

Case	**Old Renderer**	**New Renderer**
San Francisco (default)	41.3 fps	9.5 fps
Average	57.5 fps	16.8 fps

Not good! We began optimizing with a hand from the rest of the team and got us back up to par.

San Francisco (default)	41.3 fps	43.0 fps
Average	*57.5 fps*	*57.9 fps*

Good enough to turn on the new renderer, I think. But the call went out for complex style sheets and he found a few other cases to look at. Facebook's style is well known for containing a lot of layers and focusing heavily on POIs.

Oakland (Facebook)	59.8 fps	49.1 fps
SF (Americana)	41.1 fps	35.9 fps
Paris (Americana)	59.4 fps	40.0 fps

Those three jump out as interesting data points beyond the average FPS, which only drop from 60fps to around 56 or 57 for the complex styles. The Instrument traces point to some optimizations which would benefit both OpenGL and Metal renderers, so we'll circle back to this later.

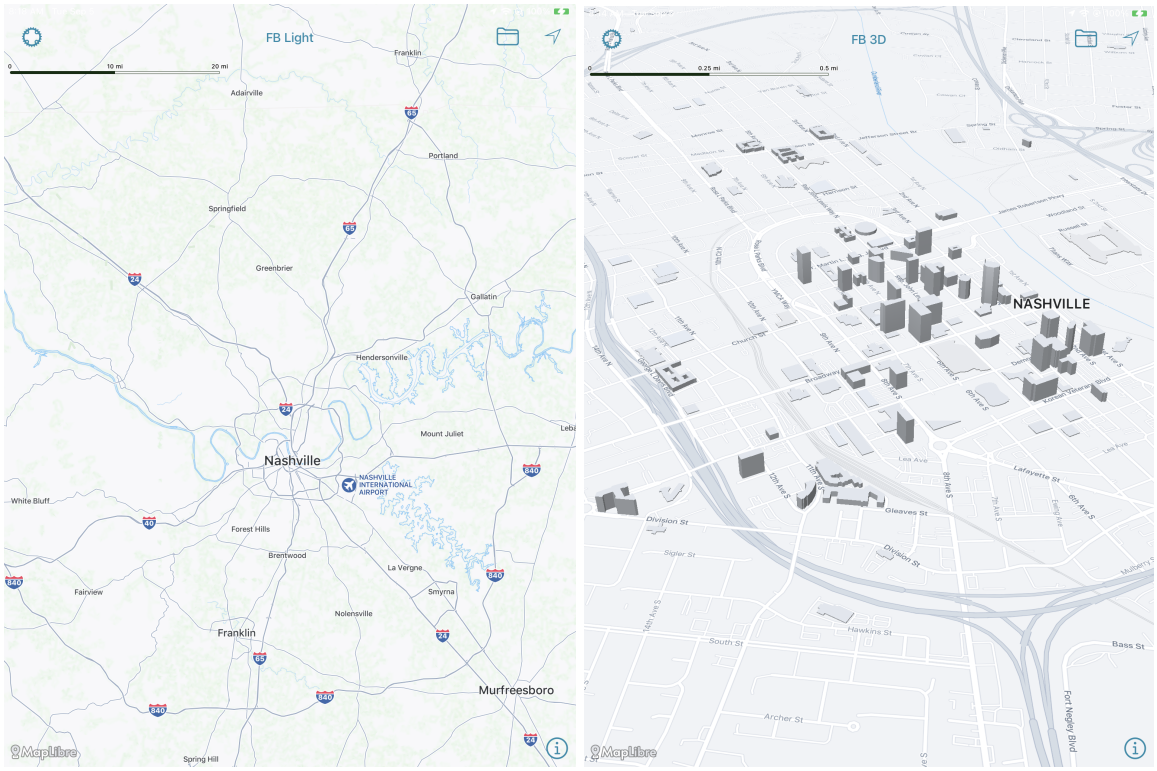
If you're curious about the details, we continue to look at the results in this Issue.

At a certain point it will make more sense to focus on the GPU. More on that later.

For toolkit size, we have some initial data and we appear to have made the toolkit slightly bigger after the first phase. We'll know more in the coming month and will begin to address that in the remaining time. We have a number of options for cutting things down or making them optional.

Metal Implementation & Testing

We have a distinct "Ship of Theseus" problem. It just looks like MapLibre, only running on top of Metal.



Porting the layers was mostly an exercise in porting the shaders. The whole Drawable/Tweaker architecture worked to shield us from the details. There was, and will continue to be, some weirdness related to stencils and buffer handling, but nothing we can't sort out.

One outstanding problem is wide lines. OpenGL provides a simple implementation, but Metal does not. We're looking to reuse the line layer implementation, which opens up some interesting problems which I'll discuss later.

Moving Work to the GPU

Our Metal implementation is a naive Metal implementation. It works just the same as the OpenGL version. A fine way to do a port, but a waste of GPU power. We should move work to the GPU.

What work? Take the example of features fading in and out as you change the zoom level. Right now the CPU is furiously calculating opacity values for each feature that changes. In the terminology of modern GPU development this is "*really dumb*".

The proper way to do it is to give the shaders the functions that govern opacity and then let them calculate it as they need it. If you're thinking "but doesn't that duplicate…" no, it's not even a tiny factor. We should just do it.

This kind of thing holds for just about everything that changes based on time and zoom level in the map itself. The goal would be to stand up the geometry, tell it what the core values are (e.g. time, zoom) and not modify the geometry buffers until it's time to delete it. That's how we do it in WhirlyGlobe and it works quite well.

It will be easy in a few cases. Simple linear equations for opacity based on zoom level, for example. In other cases it won't, such as colors driven by external data values. We'll start with the easy ones and see where we wind up.

Secondary Goals

When the project began we had a number of secondary goals, some of which we'll try to address in the remaining time.

Custom Layers / Drawable Builders / Buckets

There is a class of layers implemented by the toolkit user. These do things like data overlays, user position tracking, and various other random things not well served by vector tiles or GeoJSON.

Right now these all have to be implemented from scratch. They can't readily share the same mechanisms we use to build up lines and polygons and so forth. We'd like to change that.

Reusing the Line Layer implementation in the Fill Layer outline implementation will be a good entry point into this.

Layer Addition, Replacement & Style Sheet Additions

It should be possible to add new layer types and replace existing layers from outside the toolkit. There's a central registry for these we may be able to extend for dynamic additions.

That also means allowing additions to the style sheet implementation. New fields and entirely new layer types. Right now all of this is constructed for the library at compile time but we may be able to bolt a "*yes, and"* mechanism on the side.

Threading

Currently most of the tile construction work is done on the main thread. It really ought to be possible to do most of this work on other threads, an approach we use to good effect in WhirlyGlobe.

It's not clear we'll get to this one.

Wins

From a functionality standpoint, Metal is largely working. There are a few things to chase down and we'll have to see how the tests look, but the scary part is over. We have something that looks like a map.

Taking a step back on the whole project, the renderer modularization worked. We front loaded the difficult part of understanding MapLibre. That made the actual Metal port much simpler. Presumably future rendering SDKs like Vulkan, WebGPU, or Direct3D will be easier as well.

Challenges

The big question in the coming months is how many of the secondary goals we can meet. We won't manage them all, but we'd like to leave the toolkit in a better place for expandability and performance going forward.

This is where the future of the toolkit comes into view. We're securing MapLibre Native's present, fixing a huge problem and dragging the architecture into the 2020's. But what about the future? AR? Vision Pro? 3D terrain and buildings? Using MapLibre as a platform for experimentation and development of whatever is next? That's still in flux.

Beyond that, we may have some trouble with toolkit size. We'll find out over the next month.

Insights

In the end, much of the process we laid out for the Mega Merge wasn't all that necessary. There weren't many participants beyond the maintainers so we could have just consulted directly with them.

As we've seen before, this is a project of great importance to people with very little free time.

Risks

We're past the major risks. We'll have a working Metal implementation. All that's in question is how many of the secondary goals we can meet.

Beyond that, there's the future of the toolkit. MapLibre is better positioned now to inherit the open source users. But new users who want to do bigger things are probably the key to the future.