The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume XLVIII-4/W1-2022
Free and Open Source Software for Geospatial (FOSS4G) 2022 – Academic Track, 22–28 August 2022, Florence, Italy

MAPLIBRE-RS: TOWARD PORTABLE MAP RENDERERS

Maximilian Ammann[1,*], Antoine Drabble[2], Jens Ingensand[2], Bertil Chapuis[2]

[1] MapLibre project - max@maxammann.org
[2] University of Applied Sciences and Arts (HES-SO/HEIG-VD), Switzerland - (antoine.drabble, jens.ingensand, bertil.chapuis)@heig-vd.ch

Commission IV, WG IV/4

KEY WORDS: Geospatial, Map, Vector Tiles, Rust, WebGPU, WebAssembly.

ABSTRACT:

Map renderers play a crucial role in Web, desktop, and mobile applications. In this context, code portability is a common problem, often addressed by maintaining multiple code bases: one for the Web, usually written in JavaScript, and one for desktop and mobile, often written in C/C++. The maintenance of several code bases slows down innovation and makes evolution time-consuming. In this paper, we review existing open-source map renderers, examine how they address this problem, and identify the downsides of the current strategies. With a proof of concept, we demonstrate that Rust, WebAssembly, and WebGPU are now sufficiently mature to address this problem. Our new open-source map renderer written in Rust runs on all platforms and showcases good performance. Finally, we explain the challenges and limitations encountered while implementing a modern map renderer with these technologies.

1. INTRODUCTION

Map renderers play a crucial role in various applications deployed in Web, desktop, mobile, and embedded applications. For instance, we rely on them to travel, commute, find the best hotels and restaurants, and locate others. Their adoption drives innovation in various areas, such as urban planning, transportation, solar panel placement, pandemic monitoring, etc. (Biljecki et al., 2015). Beyond digital environments, maps also get printed in books, reports, or pieces of urban furniture. Despite major advancements in mapping solutions during the last 20 years, there is still room for improvement. While embed maps are already part of our everyday life, software developers still face significant challenges when creating artifacts that use maps. For example, these challenges include styling maps or self-hosting geo data. In this paper, we focus on the problem of code portability (i.e., the ability to use a single code base on various platforms).

Code portability is a common problem in map renderers. For instance, several map renderers maintain a JavaScript codebase for the Web (e.g., mapbox-gl-js, maplibre-gl-js, and tangram) and a C++ codebase for native platforms (e.g., mapbox-gl-native, maplibre-gl-native, and tangram-es). The JavaScript codebases allow their renderers to run in all major browsers using the WebGL graphics API. The C++ codebases allow running on desktop and mobile environments, servers (e.g., for headless rendering), in cars, planes, or in embedded settings. Guarantying that these renderers behave similarly and produce the same outputs on all these platforms is challenging and costly, and it slows down the ability of development teams to innovate and improve renderers.

In this paper, we present emerging solutions to the code portability problem. We then review the most popular open-source map renderers from a portability point of view. We show that the existing codebases currently fail at producing a portable map renderer in at least one area. Based on the emerging technologies identified, we study the feasibility of creating a portable map renderer. We present a proof-of-concept, released under the terms of the MIT, that can render maps natively and in the browser. We describe its overall architecture and highlight the challenges we encountered while implementing it. Finally, we present our future work and explore possible ameliorations. Overall, this review and feasibility study gives an exciting glimpse into a possible future for map renderers.

2. BACKGROUND

This section introduces fundamental concepts that explain the rest of the paper. Firstly, we present modern portable technologies that can be used natively or in a browser. Secondly, we provide a bird's-eye view of portable graphics APIs.

2.1 Code Portability

A code is portable when it can execute on multiple platforms. To some extent, most programming languages are portable. They can achieve portability with interpreters, compilers, or a combination of both: interpreters translate source code into machine code at runtime; compilers convert source code into machine code before the execution; some interpreters compile parts of the code just-in-time (JIT) to achieve better performance. Here, we recall widespread approaches to code portability for Web and native environments.

JavaScript is an interpreted language originally designed for the Web. The V8 engine, a cross-platform interpreter and JIT compiler written in C++, is widely used to create portable JavaScript applications that run in browsers, on servers, and in other environments (V8, 2008). Outside of browsers, access to graphics APIs is possible but difficult, which limits the use cases. WebAssembly or Wasm is a binary instruction format that can run in browsers with a low-performance overhead compared to native (Haas et al., 2017). It is a popular compilation target for low-level and high-performance programming languages like

* Corresponding author

C, C++, and Rust. An increasing number of higher-level programming languages, such as Go and C#, also compile to Wasm, hence achieving greater portability. The Wasm text format (WAT) is a textual representation of Wasm's binary instruction, which is intended to be read by humans. The C and C++ languages have been popular among graphics programmers, as they enable low-level memory access. Many GIS and CAD applications, such as QGIS, ArcGIS, or AutoCAD, use these languages for performance-critical components. C and C++ can both be compiled to Wasm with the LLVM compiler (Lattner and Adve, 2004) and Emscripten (Zakai, 2011). For instance, the QGIS rendering engine was successfully compiled for the Web (Dobias, 2022). Rust is a high-level programming language designed for safety and high performance. The project started at Mozilla and is now developed by the Rust Foundation. Its compiler targets various architectures, such as x86 and ARM. The Rust compiler can also target Wasm with a code generation backend based on LLVM (Rust and WebAssembly, 2018). This enables Rust applications to run natively and in Web applications.

2.2 Graphics Portability

Graphics rendering depends significantly on the hardware and drivers supplied by platform vendors. Here, we recall low-level and platform-dependent APIs, portable APIs initially developed for the Web, and higher-level APIs aimed at simplifying the task of rendering graphics.

The OpenGL, Vulkan, DirectX, and Metal specifications handle low-level graphics-related tasks. The Vulkan and OpenGL specifications both aim at being vendor independent and cross platform. DirectX and Metal are vendor-dependant and respectively target Windows and macOS. WebGL enables JavaScript applications to render 2D and 3D graphics in all major Web browsers (WebGL, 2011). The WebGL API is similar to OpenGL ES, a subset of the OpenGL API targeting embedded and mobile systems. As of the time of writing, browsers have adopted versions 1 and 2 of WebGL. The WebGPU specification is considered the successor of WebGL (WebGPU, 2021). Inspired by Vulkan, Metal, and DirectX 12, its API gives low-overhead access to modern graphics hardware. The GPU for the Web Community Group develops the W3C specification with engineers from Apple, Microsoft, Mozilla, Google, and others. Contrary to WebGL, which was solely designed for the Web, WebGPU implements a standard header file (webgpu.h) that allows it to work across platforms and be available in native environments. The most prominent implementations of this specification are wgpu and dawn (Wgpu, 2019, Dawn, 2017). Using low-level APIs for rendering graphics in 2D and 3D requires in-depth knowledge and can be challenging. Higher-level APIs and frameworks, ranging from lightweight rendering libraries (e.g., Skia, bgfx, Filament, etc.) to full-featured game engines (e.g., Bevy, Unity, Unreal Engine, etc.), address this problem. Some of these engines enable physically based rendering.

3. STATE-OF-THE-ART

TerraVision was one of the first applications to use tiles to load terrain data at multiple resolution levels (Leclerc et al., 1995). The tiling of geospatial data became very popular in the early 2000s due to the availability of global datasets and the emergence of well-known Web mapping solutions, such as Google Maps and OpenLayers (Haklay et al., 2008). Figure 1 depicts a hypothetical tile scheme and some tile coordinates. Over time,

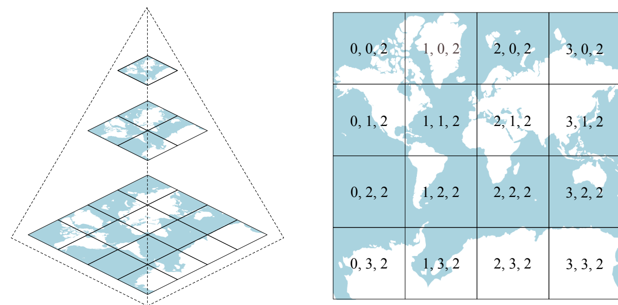


Figure 1. A multi-resolution pyramid of tiles (zoom levels 0, 1, and 2) and tile coordinates (x, y, zoom level).

many specifications, such as the tiling scheme utilized by OpenStreetMap (Haklay and Weber, 2008), have been established to describe these notions. Recently, the Open Geospatial Consortium (OGC) defined them more formally in the OGC TMS standard (OGC TMS, 2019).

Raster tiles encode geospatial data with image formats, such as TIFF or PNG. The advancements in the industry regarding the creation and utilization of raster tile mapping systems have led to the establishment of the OGC WMTS standard for tiled raster services (OGC WMTS, 2010). Cloud services and government agencies have widely adopted this standard to disseminate geospatial data. WMTS offers the possibility of defining a custom tiling scheme (e.g., with different non-regular zoom levels) and projection systems that optimize the display of specific raster data.

The Mapbox Vector Tile Specification has had a great success (Mapbox Vector Tile Specification, 2014) and became a de facto standard for vector tiles. This specification defines several aspects of vector tiling, such as: the format (i.e., protocol buffers); the tiling scheme (which allows for only one coordinate system); the simplification of input geometries snapped on a grid (which reduces the size of the tiles but does not keep the topology); and the addressing of the tiles. The OGC recently drafted a specification within the frame of the new OGC API standards family that defines the tiling scheme and the utilization of different coordinate systems. It keeps the choice of a data format and attributes handling open (OGC API, 2017). Most software capable of producing vector tiles focuses on the Mapbox Vector Tile Specification, but server-side implementations of the OGC API Tiles Specification recently emerged (e.g., PyGeoAPI and GeoServer).

The OGC 3D Tiles Specification defines a way to tile 3D geospatial data (OGC 3D Tiles, 2019). This standard corresponds to the Cesium 3D Tiles Specification (Cesium 3D Tiles, 2015). The tiling scheme is irregular in this specification: it is weight-based (e.g., the number of vertices). In other words, the extent of one tile depends on its weight (e.g., compared to WMTS, OGC API tiles, and MapBox vector tiles) to optimize the loading of tiles. Consequently, each tile's location needs to be communicated to the client and cannot be deduced automatically by the viewport.

4. REVIEW OF MAP RENDERERS

To review map renderers exhaustively, we started by listing them as broadly as possible based on our field knowledge. We included renderers that: target different types of environments

	Platforms						Input formats							
GitHub Repository	**Web**	**Linux**	**Windows**	**macOS**	**iOS**	**Android**	**Raster tiles**	**Vector tiles**	**Other**	**3D map**	**Lang**	**License**	**Entity**	**Year**
a-b-street/abstreet	✔	✔	✔	✔					✔		Rust	Apache-2.0	A/B Street	2018
aliflux/vectortilerenderer		✔	✔	✔	✔	✔	✔	✔			C#	MIT	Individual	2018
cartodb/mobile-sdk			✔	✔	✔	✔	✔	✔		✔	C/C++	BSD-3	CARTO	2016
cesiumgs/cesium	✔						✔		✔	✔	JS	Apache-2.0	Cesium	2012
dfyz/osm-renderer		✔	✔	✔					✔		Rust	MIT	Individual	2018
enzet/map-machine		✔	✔	✔					✔		Python	MIT	Individual	2015
framstag/libosmscout		✔	✔	✔	✔	✔			✔		C++	LGPL	Individual	2015
geoserver/geoserver		✔	✔	✔			✔	✔	✔		Java	GPL-2.0	GeoServer	2001 W
heremaps/harp.gl	✔						✔	✔	✔	✔	TS	Apache-2.0	HERE	2018
hijiangtao/glmaps	✔						✔		✔	✔	JS	MIT	Individual	2019
itowns/itowns	✔						✔	✔	✔	✔	JS	Cecill-B/MIT	iTowns	2015
josm/josm		✔	✔	✔			✔	✔	✔		Java	GPL-2.0/3.0	OSM DE	2010
karimnaaji/vectiler		✔	✔	✔				✔		✔	C++	MIT	Individual	2015
leaflet/leaflet	✔						✔	✔	✔		JS	BSD-2	Leaflet	2010
mapbox/mapbox-gl-js	✔						✔	✔	✔	✔	JS	Source-avail.	Mapbox	2013
maplibre/maplibre-gl-js	✔						✔	✔	✔	✔	JS	BSD-3	MapLibre	2020
maplibre/mablibre-gl-native		✔	✔	✔	✔	✔	✔	✔	✔	✔	C++	BSD-2	MapLibre	2020
mapnik/mapnik		✔	✔	✔			✔		✔		C++	LGPL-2.1	Community	2011
mapserver/mapserver		✔	✔	✔			✔	✔	✔		C/C++	MIT	OSGeo	1994 W
mapsforge/mapsforge		✔	✔	✔		✔			✔		Java	LGPL-3.0	Community	2014
mapsme/omim		✔	✔	✔	✔	✔			✔		C++	Apache-2.0	Maps.me	2015
mousebird/whirlyglobe					✔	✔	✔	✔	✔	✔	C++	Apache-2.0	Mousebird	2012
nasa-ammos/3dtilesrendererjs	✔						✔	✔	✔	✔	JS	Apache-2.0	NASA	2020
nils-hamel/eratosthene-suite		✔							✔	✔	C	GPL-3.0	Individual	2017
openlayers/openlayers	✔						✔	✔	✔		JS	BSD-2	OpenLayers	2006 W
openmobilemaps/maps-core					✔	✔	✔		✔		C++	MPL-2.0	Ubique	2021
orbisgis/orbismap		✔	✔	✔					✔		Java	GPL-3.0	OrbisGIS	2017
organicmaps/organicmaps		✔	✔	✔	✔	✔			✔		C++	Apache-2.0	Organic Maps	2020
potree/potree	✔								✔	✔	JS	BSD-2	Potree	2012
felixpalmer/procedural-gl-js	✔						✔		✔	✔	JS	MPL-2.0	Individual	2020
protomaps/protomaps.js	✔						✔	✔	✔		TS	BSD-3	Protomaps	2021
qgis/qgis		✔	✔	✔			✔	✔	✔	✔	C++	GPL-2.0	QGIS	2002 W
tangrams/tangram	✔						✔	✔	✔	✔	JS	MIT	Mapzen	2013
tangrams/tangram-es		✔	✔	✔	✔	✔	✔	✔	✔	✔	C++	MIT	Mapzen	2014
tordanik/osm2world		✔	✔	✔					✔	✔	Java	LGPL-3.0	Individual	2012
tumic0/qtpbfimageplugin		✔	✔	✔	✔	✔	✔	✔			C++	LGPL-3.0	Individual	2018
visgl/deck.gl	✔						✔	✔	✔	✔	JS	MIT	Vis.gl	2015

Table 1. Review of open-source and source-available map renderers.

(e.g., desktop computers, browsers, etc.); run on servers (i.e., they may eventually run in the browser thanks to Wasm); are written with a high-level programming language (i.e., it is often justified to prioritize user experience over portability); support different data formats (e.g., raster tile, vector tile, etc.); are part of a greater feature set (e.g., GIS application, editor, etc.). We then expanded our list using online lists and search keywords, which allowed us to find map renderers implemented in various programming languages (e.g., C#, Python, Kotlin, Java, etc.). Finally, we used the following criteria to filter our list and reduce it to a reasonable size: updated during the last two years; aimed at rendering navigable world maps; not dependent on a full-featured game engine (e.g., Unreal Engine, Unity, etc.); open-source or source-available. We apologize if we missed some important map renderers during our selection process, or if we miss-classified some of their features. Nevertheless, we believe that the resulting list gives an accurate picture of the open-source and source-available map renderers available.

For each map renderer, we gathered the following information: the *platforms* (i.e., if it runs on the Web, Linux, Windows, macOS, iOS, or Android); the supported *formats* (i.e., if it can display raster tiles, vector tiles, or other formats); the ability of the renderer to display a *3D map*; the main programming *language* used to develop the renderer; the *license* used (i.e., whether it is open-source or source-available); the *entity* behind the development of the project (i.e., company, foundation, individual, etc.); the *year* of inception observed in the GitHub repository (e.g., the year of the first commit), or on Wikipedia.

Table 1 summarizes the information we collected, and we make the following observations. Firstly, we notice that none of the reviewed open-source map renderers achieve Web and native portability. Some of them, including the QGIS rendering engine, experimented with Wasm and demonstrated that it is possible to port a native renderer to the Web. However, none of these initiatives has been massively adopted by end-users yet. As of the time of writing, Google Earth and CartoType are the only notable closed-source map renderers written in C++ that achieve portability with Wasm (Beck and Mears, 2020, Asher, 2022). We expect many map renderers to follow in their footsteps soon. Secondly, we observe that JavaScript is the most popular language for renderers targeting the Web. Similarly, C++ is the most popular language for native environments. Several entities (e.g., Mapbox, MapLibre, and Tangram) maintain compatible JavaScript and C++ codebases to target the Web and native environments. Thanks to the ability to compile C++ to Wasm, many JavaScript renderers (e.g., maplibre-gl-js) may eventually be replaced by their C++ counterparts (e.g., maplibre-gl-native). However, this approach may produce large Wasm binaries due to legacy code and dependencies. Finally, we notice that map renderers tend to support more formats as they get more mature. For instance, raster tiles, commonly used in 2D maps, can be loaded in 3D globes. Similarly, glTF objects, commonly used in 3D globes, can now be displayed in renderers that were initially intended to display 2D maps.

5. PROOF OF CONCEPT

In this section, we present maplibre-rs, a new portable map rendering library[1]. We lay out its overall architecture and design. We describe its features and the challenges that we foresee in releasing a full-featured portable map renderer written with Rust. Finally, we describe some issues encountered while building and packaging it for different targets. This project was released on GitHub in the MapLibre organization and published with the MIT license.

5.1 Rendering Architecture

Rust, Wasm, WebGPU, and wgpu provide unparalleled portability while guaranteeing good performance on all supported platforms. WebGPU represents a novel and modern way to define a Web-first, cross-platform graphics API. Compared to its predecessor, WebGL, standalone runtimes that support WebGPU are available from the start. Figure 2 depicts the overall architecture of maplibre-rs and shows how this technology is leveraged to create a portable map renderer. The maplibre-rs library depends on wgpu; furthermore, a cross-platform graphics API written in Rust is also used by Firefox internally as a WebGPU implementation. Rust applications integrating wgpu can compile to native targets, in which case, a hardware abstraction layer (HAL) renders graphics using low-level APIs, like Vulkan, Metal, DirectX, or OpenGL. If maplibre-rs is compiled to a Wasm binary for the Web target, either WebGL or WebGPU can be used to render graphics.

Because WebGL is only available in Web browsers, it cannot be used on a non-Web-based Wasm runtime. The Deno runtime for JavaScript, TypeScript, and Wasm currently supports the WebGPU specification in a headless mode. WebGL and WebGPU implementations can differ between browsers. While Firefox uses wgpu, Chrome ships with Dawn, a different implementation of WebGPU (Wgpu, 2019, Dawn, 2017).

5.2 Design

Several requirements shaped the design of maplibre-rs. The library must be: (i) able to run across platforms, (ii) interactive, (iii) performant, (iv) extensible, (v) modern, and (vi) lightweight.

The cross-platform support is addressed by the architecture introduced in Section 5.1. By leveraging the WebGPU graphics API and the Rust programming language, it is possible to reach high performance and make the map renderer highly interactive. This means that users can interact with the map in real-time. Because the Rust programming language is a systems programming language, low-level control over the memory is possible. Extensibility is achieved by keeping the design of maplibre-rs modular, as shown in Figure 3. Building on top of modern Rust libraries and APIs makes it possible to keep maplibre-rs lightweight. In practice, we never had to depend on C/C++ libraries. For instance, pure Rust implementations were available for TLS or PNG decoding. This makes the build process and the resulting binary self-contained and stand-alone. In the following, we introduce the core modules of maplibre-rs.

5.2.1 schedule A *schedule* is responsible for preparing new frames, handling input, requesting data, preparing GPU resources, and queuing up render commands. These steps are encapsulated

[1] `https://github.com/maplibre/maplibre-rs`

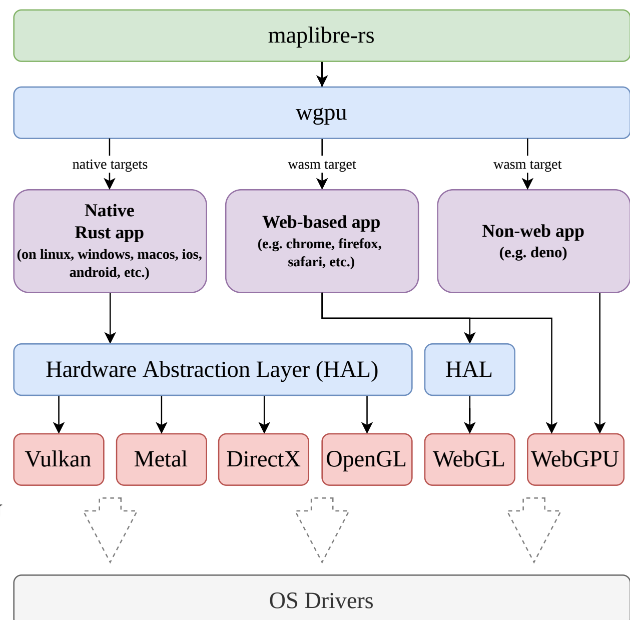


Figure 2. Rendering architecture of maplibre-rs. The library is depicted as a green box. Components of the wgpu project are shown in blue. Target environments are colored in violet. Low-level and Web graphics specifications are shown in red. The underlying OS-dependant drivers are pictured at the bottom.

in *stages*, which consist of executable tasks with a run method. Performance optimizations can be performed in-between *stages*. The *stages* are executed by the *schedule* for each frame. This architecture enables maplibre-rs to be extensible and yet performant by allowing optimizations.

5.2.2 data pipeline A *data pipeline* is used to describe how data is fetched, transformed, and cached. Expensive operations should not block the thread that is responsible for rendering the map; therefore, the steps of a *pipeline* are executed asynchronously. Some steps, such as fetching data from disk or remote servers, are common to most *pipelines*. Some other steps are dependent on the format. For instance, while vector tiles require a tessellation step that triangulates geometric data, raster tiles need an image decoding step.

5.2.3 renderer A *renderer* takes input data and displays it on the screen. In the case of wgpu, resources like buffers, textures, and shaders need to be initialized, and render commands have to be prepared. In order to design an extensible and performant *renderer*, an abstraction must be introduced that allows implementing various rendering techniques (Williams, 1978, Schneider and Klein, 2007, Trapp and Dollner, 2019). The *renderer* of maplibre-rs uses a render graph in order to resolve dependencies between different render passes (O'Donnell, 2017). This rendering architecture creates an abstraction on top of wgpu. Before a render pass is executed, GPU-allocated resources need to be prepared, and render commands need to be queued. The resources are resolved during the render pass, and the commands are executed. We did not build on top of a high-level rendering engine, because we wanted to be able to optimize the renderer. By not using an off-the-shelf render engine, we were able to create a domain-specific renderer, which also fulfills the requirement of being lightweight. Usually, existing render engines come with many dependencies.

https://doi.org/10.5194/isprs-archives-XLVIII-4-W1-2022-35-2022 | © Author(s) 2022. CC BY 4.0 License.

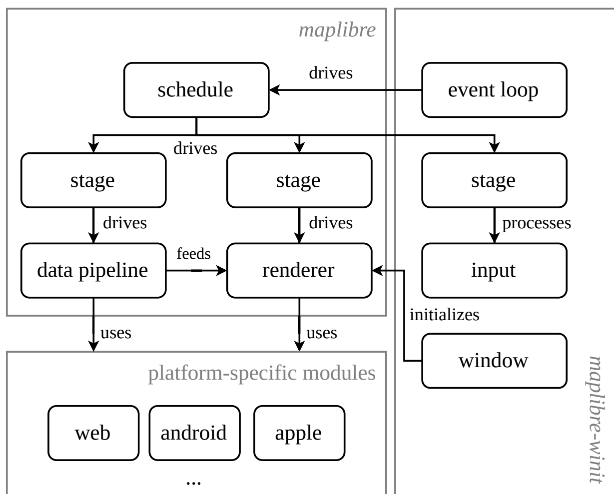


Figure 3. Design of maplibre-rs. The maplibre-rs library consists of a core component, which is called *maplibre*. The *schedule* feature drives multiple *stages*, which have different responsibilities like preparing *data*, *rendering* data, or handling *inputs*. The *maplibre-winit* component implements the management of an *event loop*, *inputs*, and *windows*. It is based on the popular `winit` library. In order to host *platform-specific code*, several platform-specific modules are available.

5.2.4 window and event loop The *window* module prepares the surface for the *renderer* in order to draw the map. This can either be an entire window or just a portion of a window. On mobile platforms, maps usually need to be displayed on a portion of the screen. In the case of headless rendering, no window is required. It renders on a GPU-allocated texture instead. We introduced an abstraction to support these different render targets and use cases. Depending on the use case and the platform that provides the surface or window, a different *event loop* implementation is required. On Web-based platforms, the event loop is driven by a method called `requestAnimationFrame()` (HTML Standard, 2022). This API requests a new frame from maplibre-rs depending on the refresh rate of the user's display. If maplibre-rs is used headlessly without a proper window, the event loop can be implemented differently based on the required use case.

5.2.5 input Different platforms offer different methods of interacting with maps. For example, in desktop environments, the map can be controlled with a pointing device, such as a mouse. On mobile platforms, touch inputs with gestures are used. At the time of writing, maplibre-rs processes raw inputs from the underlying platform: keyboard inputs, mouse inputs, and cursor positions. While this is a simple solution, it lags in terms of user experience: gestures may work slightly differently depending on the device. For instance, the double-tap gesture is implemented differently on iOS and Android. Therefore, a future version of this module will use platform-specific SDKs to process gestures.

5.2.6 platform specific modules Each platform (e.g., Web, Android, iOS, macOS) targeted by maplibre-rs requires slightly different implementations for rendering or scheduling asynchronous tasks. For example, we use a different asynchronous runtime on the Web and in native targets. In order to separate concerns, each target platform use a separate module that implemented platform-specific code.

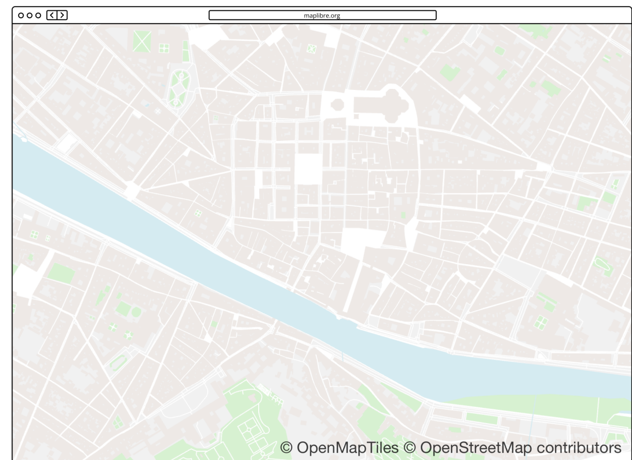


Figure 4. A map rendered using maplibre-rs. Text rendering is not included as it is not yet fully integrated in the renderer.

5.3 Features

This section describes the main features that we evaluated in our proof-of-concept. They were not all implemented in the main branch of maplibre-rs, but their feasibility was demonstrated with the architecture described in Section 5.1. Figure 4 displays a screenshot of maplibre-rs running on macOS.

5.3.1 Vector tile tessellation The Mapbox Vector Tile Specification describes an efficient encoding format for geographic vector data (Mapbox Vector Tile Specification, 2014). A vector tile usually contains multiple layers. Each layer consists of a name and a list of features, which belong semantically together (e.g., buildings, roads, etc.). Each feature contains a geometry that can either be a point, a line-string, or a polygon. The containment and direction (e.g., clockwise or counterclockwise) of polygon rings are used to draw multi-polygons with holes. Additionally, each feature contains tag references (i.e., a dictionary-like structure is used to store tag values and avoid redundancy). Using a tessellation algorithm, we convert the 2D geometries into a set of triangles, called a mesh, that can then be transferred to the GPU for rendering. In maplibre-rs, we use Lyon, a powerful and portable tessellation library written in Rust (Silva, 2016).

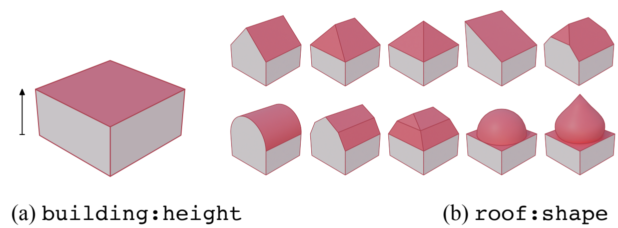
5.3.2 Vector tile styling The Mapbox Style Specification defines the visual appearance of a map (Mapbox Style Specification, 2014). This is a complex specification that covers many aspects, such as: the main properties associated with the map (e.g., version, name, metadata, etc.); the configuration of the viewer when opening the map (e.g., its center, zoom, pitch, and bearing); the definition of data sources (e.g., vector, raster, GeoJSON, etc.); the layers and styling rules applied to the data (e.g., fill colors, outline colors, legends, icons, etc.). Expressions play an essential role in the specification. They allow defining rich formulas for computing properties dynamically based on feature properties and current zoom. With maplibre-rs, it is possible to define simple styles. However, given its extent, we do not yet support the full Mapbox Style Specification. In the future, we plan to improve the support for it to load style files dynamically and allow the user to update them interactively. Advanced styling capabilities enable exciting use cases, such as: color-blindness and night modes support; contextual mapping (i.e., the ability to style the map depending on a context).

5.3.3 Coordinate systems and tile matrix The maplibre-rs library currently supports the Web Mercator projection and a quad-tree tile structure where each zone of a tile is divided into four children. This approach (also adopted by other renderers) drastically simplifies the development and allows for efficient visualizations (i.e., the data is not reprojected before being displayed). However, this approach also has disadvantages, as some data sets are not available in the Web Mercator projection. Furthermore, it limits the number of use cases, as specific coordinate systems may be required for advanced types of utilization like the land registry or military applications. In the existing WMTS standard, as well as in the future OGC-API Tiles standard, it is possible to define a custom tile matrix (i.e., determining custom zoom levels and custom coordinate systems) (OGC WMTS, 2010, OGC API, 2017). Therefore, we intend to support other coordinate systems and custom tile matrices in the long term.

5.3.4 Portable text rendering During the design and development of maplibre-rs, we considered several approaches for rendering text on a map. While Web-based renderers can theoretically use the canvas API to display text, it is unavailable on native targets, and its performance may vary between browsers (HTML Standard, 2022). Therefore, it is not an option for maplibre-rs. Traditionally, bitmap fonts are used to render text. Modern font formats feature vector glyphs consisting of quadratic or cubic Bézier curves. In contrast to bitmap fonts, vector fonts require a rasterization step. Bitmap fonts are resolution-dependent, whereas vector fonts can be rendered independently of the resolution. Being independent of the resolution is essential because fonts should be clear and sharp regardless of their size. Rasterizing vector fonts is a computing-intensive task. Therefore, a new approach based on signed distance fields was introduced in 2007 (Green, 2007). This approach simplifies the rasterization and makes it independent of the resolution. It also works for simple glyphs but yields artifacts when rendering complex glyphs at a high resolution. More recent approaches rasterize Bézier curves directly on the GPU (Dobbie, 2016, Lengyel, 2017). Lengyel's approach is patented and incompatible with an open-source project (Lengyel, 2017). A more straightforward yet efficient approach creates a tessellated mesh for each glyph and makes the edges on the GPU more smooth (Wallace, 2016). The maplibre-rs project implements this method as well as signed distance field rendering as a proof-of-concept.

5.3.5 Portable networking The interface for networking is different in Web browsers and operating systems. Operating systems like Android, iOS, macOS, Linux, or Windows provide a feature-rich and low-level API for sending and receiving data over the internet protocol. Browsers only offer a limited API designed around Web primitives. We created a uniform interface to download vector tiles to address this issue. This interface selects the proper implementation at compile time depending on the targeted architecture. For the Web platform, we use bindings provided by `wasm-bindgen`. (Rust and WebAssembly, 2018). When running natively in an operating system, we use the `reqwest` crate. Depending on the operating system, `reqwest` simplifies the portability problem with different implementations of TLS. For Linux, macOS, iOS, and Windows, `reqwest` uses OpenSSL. Because we cross-compile for Android, we use the `rustls` implementation to simplify compilation.

5.3.6 Geometry extrusion The Vector Tile Specification can include tags alongside 2D geometries. Depending on the dataset used to create the vector tiles, these tags convey meaning in the third dimension, and we can infer 3D information from it. As illustrated in Figure 5, the `building:height` and `roof:shape` attributes are used in OpenStreetMap to respectively describe the height of buildings and the shape of rooftops. We experimented with a simple extrusion approach to create 3D objects: we took a 2D polygon, copied it further along its normal axis, and filled the sides with new faces. A more advanced approach may take roof shapes into account in the future. Additional properties, such as wall materials and facade colors, may also be used. Extrusion can also be applied to other 2D objects, such as roads, trees, railway bridges, etc.



(a) `building:height` (b) `roof:shape`

Figure 5. Extrusion based on OpenStreetMap attributes.

5.3.7 Headless mode The maplibre-rs library can be operated in headless mode. This means that instead of rendering the map onto a window or surface, the map is rendered onto an in-memory texture, which can then be transferred to the CPU and written to the disk as a raster file (e.g., PNG). In the future, we will improve how we handle the particularities of the headless mode. For instance, a common use case consists of efficiently transforming vector tiles into raster tiles. In this use case, the viewport should load a single tile from the memory instead of loading several tiles from the disk or network.

5.4 Web Platform Limitations

During the development of maplibre-rs, we encountered several Web platform limitations. These limitations represent barriers that specification designers and browser manufacturers deliberately impose. The design of maplibre-rs carefully takes them into account. While some of these limitations are temporary because of missing features, most are permanent.

5.4.1 Raster tile decoding Raster tiles consist of image data encoded in a specific format. While a decoder can be implemented in pure Rust, state-of-the-art implementations for decoders are usually written in C/C++. Therefore, it is beneficial to use the image decoding API of the browser, which converts compressed images to bitmaps. Unfortunately, the implementation of WebGPU available in Firefox is not yet able to upload these bitmaps as a texture to the GPU (Johns, 2021). It is only a matter of time until this feature gets supported.

5.4.2 Multi-processing and multi-threading Parallelism is an important aspect of rendering engines. Computing-intensive work should not be done in the rendering thread, which is expected to stay interactive. Instead, work should be offloaded to a separate thread or process that runs asynchronously to the rendering thread. While operating systems provide threads and processes as multi-core primitives, Web browsers only provide WebWorkers (HTML Standard, 2022). WebWorkers behave like lightweight processes that allow message passing through a channel. Therefore, WebWorkers follow the multi-processing paradigm. Recently, a new way of communication and synchronization between these workers has been established. WebWorkers can share memory via the `SharedArrayBuffer` API

(ECMAScript Language Specification, 2022). Using this shared memory is especially interesting when combined with Wasm, as the Rust standard library offers compatible synchronization primitives and atomic operations. Sharing memory lifts WebWorkers from following the multi-processing paradigm to the multi-threading paradigm. WebWorkers can synchronize using mutexes, semaphores, or barriers. Unfortunately, the usage of shared memory was restricted with the disclosure of the Spectre vulnerabilities in 2018 (Kocher et al., 2019). Since then, browsers have required users of `SharedArrayBuffer` to opt-in to cross-origin isolation, which would limit the ability to integrate maplibre-rs in third-party websites. In maplibre-rs, we provide a uniform task scheduling API, which allows the execution of asynchronous tasks independently of the environment. While using shared memory offers performance and simplicity benefits, we have to implement a fallback for non-cross-origin isolated websites in the future.

5.4.3 Wasm Memory Management The practical implications of memory management in Wasm are far-reaching. Firstly, the design of Wasm does not guarantee the success of initial or future memory allocations. The implementations of Wasm available in browsers behave differently when specifying the initially required memory and when growing the memory requirement (Wasm Needs a Better Memory Management Story (#1397), 2021). For instance, some of them require developers to pre-allocate the maximum amount of memory, while others pre-allocate little memory and grow it dynamically over time. This behavior may also differ in mobile and desktop environments. Secondly, Wasm does not offer a way to deallocate memory and does not support virtual memory, making memory fragmentation more likely. Therefore, developers of Wasm applications need to keep these facts in mind and potentially use custom memory allocators. Thirdly, when using shared memory in a multi-threading context, there is no way to grow the memory, and the developer must estimate the maximum memory consumption. In the context of maplibre-rs, we have not yet faced these memory-related issues, because our memory requirements are low.

5.5 Library Packaging

A *target* is an environment, typically characterized by a CPU architecture and an operating system, in which maplibre-rs can be used. The major challenge when supporting different targets is to provide developer-friendly libraries. For each target, a library needs to be created to link the maplibre-rs binary distribution and cross-language boundaries with bindings (e.g., JNI bindings on Android) and simplify its usage with target-specific code. For instance, the Android, iOS, and Web targets, respectively, require Kotlin, Swift, and JavaScript libraries. The maplibre-rs project uses CI/CD pipelines in order to test, compile, package, and distribute target-specific binaries. This ensures that the binaries stayed compatible with every target and that downstream projects could benefit from fixes and new features early. The maplibre-rs binaries are linked either statically or dynamically by target-specific libraries. Static linking typically happens during the compilation, whereas dynamic linking happens when the binary distribution is dynamically loaded at runtime. In a system-level language, such as C++ or Rust, maplibre-rs can be linked statically or dynamically. In higher-level languages, such as Kotlin or JavaScript, maplibre-rs has to be loaded dynamically. At the time of writing, our proof-of-concept has a binary distribution for all major targets as well as experimental bindings. In the following paragraphs, we describe the status of several targets.

5.5.1 Web Package EcmaScript modules can be used to package and distribute libraries for the Web (ECMAScript Language Specification, 2022). The maplibre-rs library provides a Wasm binary with JavaScript and TypeScript wrappers that load the Wasm binary, initiate WebWorkers, and render the output to an HTML canvas (HTML Standard, 2022). The Wasm binary of maplibre-rs weighs around one megabyte, and it weighs half a megabyte after compression without further optimization. As a comparison, MapLibre GL JS v1.15.2 weighs 706KB (191KB after compression). Although it is bigger than its JavaScript counterpart, maplibre-rs remains Web-friendly (i.e., lightweight to load and easy to cache). Support for Web frameworks like Angular or ReactJS can be added by building on top of the Web library target.

5.5.2 Android SDK Android libraries are distributed as AAR files. We use a Gradle project with plugins for the Android SDK and NDK to package an Android library. A further Gradle plugin simplifies the creation of a dynamic library from the Rust crate. The binary distribution of maplibre-rs is packaged in an AAR file, so it can be dynamically loaded at runtime by the Java virtual machine.

5.5.3 iOS and macOS SDKs We provide a XCode project for iOS and macOS application developers. The XCode project compiles the Rust crate and bundles static libraries for different architectures.

6. CONCLUSION AND FUTURE WORK

In this paper, we recalled some of the challenges associated with code portability and presented modern solutions to this problem. Our review showed that only a few map renderers successfully leverage Wasm to address code portability with a single code base. Our feasibility study introduced maplibre-rs, a new open-source map renderer that uses Rust, Wasm, and wgpu to achieve portability on all major platforms. The design of this renderer demonstrates that writing a cross-platform and feature-complete map renderer in Rust is feasible despite a few known limitations. Therefore, we are confident that this ecosystem will play an essential role in accelerating innovation in map renderers in the future. A pitfall that may affect maplibre-rs is the burden of legacy: innovation often requires selectively ignoring prior work. For instance, Mapbox and Cesium both introduced specifications to address the challenges associated with the visualization of spatial data in the browser (i.e., vector tiles, 3D tiles). As the industry heads toward 3D maps, essential questions to ask are: should maplibre-rs reach feature parity with maplibre-js and maplibre-native; should maplibre-rs selectively ignore some parts of its legacy? The answers to these questions will impact our future work.

7. ACKNOWLEDGMENTS

We would like to thank (in no particular order) Mapbox for revolutionizing the state-of-the-art of interactive maps, Fabian Wildgrube for researching text rendering possibilities, Brandon Liu and Pirmin Kalberer for early feedback on the project, Luke Seelenbinder for design feedback and contributing an existing codebase, Yuri Astrakhan and the MapLibre organization for hosting the maplibre-rs project, and the open-source community for its contributions to maplibre-rs.

REFERENCES

Asher, G., 2022. CartoType. https://www.cartotype.com/. Visited on 2022-06-01.

Beck, J., Mears, J., 2020. Google Earth comes to more browsers, thanks to WebAssembly. https://medium.com/google-earth/google-earth-comes-to-more-browsers-thanks-to-webassembly-1877d95810d6. Visited on 2022-06-01.

Biljecki, F., Stoter, J., Ledoux, H., Zlatanova, S., Çöltekin, A., 2015. Applications of 3D City Models: State of the Art Review. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences (ISPRS)*, 2842–2889.

Cesium 3D Tiles, 2015. https://github.com/CesiumGS/3d-tiles. Visited on 2022-06-01.

Dawn, 2017. https://dawn.googlesource.com/dawn. Visited on 2022-06-01.

Dobbie, W., 2016. GPU text rendering with vector textures. https://wdobbie.com/post/gpu-text-rendering-with-vector-textures/. Visited on 2022-06-01.

Dobias, M., 2022. QGIS and WebAssembly. https://lists.osgeo.org/pipermail/qgis-developer/2022-March/064589.html. Visited on 2022-06-01.

ECMAScript Language Specification, 2022. https://tc39.es/ecma262/. Visited on 2022-06-01.

Green, C., 2007. Improved alpha-tested magnification for vector textures and special effects. *Special Interest Group on Computer Graphics (SIGGRAPH)*, ACM, San Diego, California, 9.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J., 2017. Bringing the web up to speed with WebAssembly. *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, ACM, New York, NY, USA, 185–200.

Haklay, M., Singleton, A., Parker, C., 2008. Web Mapping 2.0: The Neogeography of the GeoWeb. *Geography Compass*, 2011–2039.

Haklay, M., Weber, P., 2008. OpenStreetMap: User-Generated Street Maps. *Pervasive Computing*, 12–18.

HTML Standard, 2022. https://html.spec.whatwg.org/. Visited on 2022-06-01.

Johns, B., 2021. WebGPU <img>, <canvas>, and <video> texture best practices. https://toji.github.io/webgpu-best-practices/img-textures.html. Visited on 2022-06-01.

Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y., 2019. Spectre Attacks: Exploiting Speculative Execution. *Proc. of the Symposium on Security and Privacy (SP)*, IEEE, San Francisco, CA, USA, 1–19.

Lattner, C., Adve, V., 2004. LLVM: A compilation framework for lifelong program analysis amp; transformation. *Proc. of the International Symposium on Code Generation and Optimization (CGO)*, IEEE, 75–86.

Leclerc, Y. G., Jr, S. Q. L., Center, A., 1995. TerraVision: A Terrain Visualization System. *SRI International*, 20.

Lengyel, E., 2017. GPU-Centered Font Rendering Directly from Glyph Outlines. *Journal of Computer Graphics Techniques*, 17.

Mapbox Style Specification, 2014. https://docs.mapbox.com/mapbox-gl-js/style-spec/. Visited on 2022-06-01.

Mapbox Vector Tile Specification, 2014. https://github.com/mapbox/vector-tile-spec. Visited on 2022-06-01.

O'Donnell, Y., 2017. FrameGraph: Extensible Rendering Architecture in Frostbite. https://www.gdcvault.com/play/1024612/FrameGraph-Extensible-Rendering-Architecture-in. Visited on 2022-06-01.

OGC 3D Tiles, 2019. https://www.ogc.org/standards/3DTiles. Visited on 2022-06-01.

OGC API, 2017. https://ogcapi.ogc.org/. Visited on 2022-06-01.

OGC TMS, 2019. https://www.ogc.org/standards/tms. Visited on 2022-06-01.

OGC WMTS, 2010. https://www.ogc.org/standards/wmts. Visited on 2022-06-01.

Rust and WebAssembly, 2018. https://rustwasm.github.io/docs/book/. Visited on 2022-06-01.

Schneider, M., Klein, R., 2007. Efficient and Accurate Rendering of Vector Data on Virtual Landscapes. *International Conferences in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, 15, 59–65.

Silva, N., 2016. Lyon. https://github.com/nical/lyon. Visited on 2022-06-01.

Trapp, M., Dollner, J., 2019. Real-time Screen-space Geometry Draping for 3D Digital Terrain Models. *Proc. of the International Conference Information Visualisation (IV)*, IEEE, Paris, France, 281–286.

V8, 2008. https://v8.dev/. Visited on 2022-06-01.

Wallace, E., 2016. Easy Scalable Text Rendering on the GPU. https://medium.com/@evanwallace/easy-scalable-text-rendering-on-the-gpu-c3f4d782c5ac. Visited on 2022-06-01.

Wasm Needs a Better Memory Management Story (#1397), 2021. https://github.com/WebAssembly/design/issues/1397. Visited on 2022-06-01.

WebGL, 2011. https://www.khronos.org/webgl/. Visited on 2022-06-01.

WebGPU, 2021. https://www.w3.org/TR/webgpu/. Visited on 2022-06-01.

Wgpu, 2019. https://gfx-rs.github.io/2019/03/06/wgpu.html. Visited on 2022-06-01.

Williams, L., 1978. Casting Curved Shadows on Curved Surfaces. *Special Interest Group on Computer Graphics (SIGGRAPH)*, 270–274.

Zakai, A., 2011. Emscripten: An LLVM-to-JavaScript compiler. *Proc. of the International Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM, Portland, Oregon, USA, 301.