Final Year Project Report

Full Unit - Final Report

Offline HTML5 Maps Application

George Honeywood

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Reuben Rowe

Department of Computer Science
Royal Holloway, University of London

March 23, 2023

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 12,889

Student Name: George Honeywood

Date of Submission: March 23, 2023

Signature: GH

Table of Contents

Abstract **4**

1 Introduction **4**

1.1 Literature review 6

1.2 Aims and objectives 6

1.3 Deliverables & timeline 7

1.3.1 Term one 8

1.3.2 Term two 8

2 Research **9**

2.1 Basic web technologies 9

2.2 Offline HTML5 applications 10

2.3 Using the HTML5 canvas 12

2.4 OpenStreetMap data sources 14

2.5 Projecting map data 15

3 Proof-of-concept development process **17**

3.1 Simple offline HTML5 app 17

3.2 Rendering geometry to a canvas 18

3.3 Adding interactivity — panning and zooming 19

3.4 Rendering tiled data from a Mapsforge file 21

3.4.1 The Mapsforge format 22

3.4.2 Writing a parser 24

3.4.3 Rendering the tiles 25

4 Final application development process: OSMO **26**

4.1 Dynamic tile loading 26

4.2 Caching dynamically loaded tiles 27

4.3 Downloading a region 28

4.4 Experimenting with WebGL 29

4.5 Svelte: using a frontend framework 30

4.6 Progressive web apps (PWAs) 31

4.7 Service worker quirks 32

4.8 Handling HiDPI displays 33

5 Software engineering **35**

5.1 Testing 35

5.1.1 Test suite coverage 37
5.2 GitLab CI 37
5.3 Version control 38
5.4 Documentation 38
5.5 Profiling and optimization 39
6 Professional issues 41
7 Conclusion & evaluation 44
8 Appendix 46
8.1 `diary.md` 46
Bibliography 51

Abstract

Although online web maps are commonplace, offline maps are a valuable niche that are useful in certain situations, such as on mobile devices with limited data, or when roaming abroad. There are limited options in this space that are cross-platform, which I hope my project can resolve through the nature of it being provided as a progressive web app (PWA).

In this project I aim to build an offline maps application based on OpenStreetMap data. The user should be able to download map data for their area of interest, then view it by panning and zooming, like a traditional online slippy map. Time permitting, I may also add additional features that require an internet connection, like routing and Wikipedia integration. Through this project I hope to learn about how you project map data into a rendered map. It will also teach me how to successfully develop a medium-sized application.

1: Introduction

The OpenStreetMap project began in 2004, with the aim of creating a free world map [1]. Since then, it has met and surpassed its goals, becoming a mature, global dataset, edited by both volunteers and corporations alike. As of late 2022, an average of 6,000 contributors edit the map daily, with more than 9 million registered in total [2]. In contrast to how cartography is traditionally carried out, OSM contributors tend to not use any specialized equipment, instead using tools like GPS and aerial imagery to create the map. The barrier of entry is low by design, making it easy for new mappers to get started. This sometimes results in inaccurate, or even vandalistic edits being made, but the community usually spots and reverts bad edits quickly.

I have personally been involved in the OpenStreetMap project since creating an account in May 2019. This was initially as I was using an OSM based map, and noticed that a new-build estate near my house was missing. Hence, I added it, and this led me down the rabbit hole of wanting to add all the missing features around me. Since then, I have been a regular contributor, with currently almost 2,000 changesets made over 341 separate days. I have also made some code contributions to the Every Door mobile editor, principally adding support for viewing the history of elements [3].

Although online OpenStreetMap-based web maps are very popular, offline maps are much less of an explored field. On the Android platform, you have a few options for offline maps, such as the venerable OsmAnd [4], Organic Maps (FOSS fork of Maps.me [5]), and

the proprietary Magic Earth [6]. On desktop, the available options are more sparse. KDE Marble is one option [7], but it seems like the offline experience is an afterthought. It allows you to download pre-rendered Mapnik Carto style tiles, but this is limited to above zoom level 16, so fine details are not visible (openstreetmap.org renders tiles down to zoom level 19).

This approach of downloading rendered raster tiles is also not very scalable, as there quickly becomes a huge number of images that need to be downloaded, and these images take up significant storage space. For example, I attempted to download the tiles for Cornwall, UK, in KDE Marble, and it would have had to fetch 58,928 tiles just for between zoom levels 11 and 16, equating to an estimated download size of 749 MB. This method is also explicitly prohibited by the OpenStreetMap Foundation's Tile Usage Policy, which states that "In particular, downloading an area of over 250 tiles at zoom level 13 or higher for offline or later usage is forbidden." [8]. This is because rendering map tiles is computationally expensive, and the OSMF is run with a limited budget.

Another option is the OpenStreetMap API itself [9]. This is more promising for offline usage, as it allows you to download vector data that can be stored more efficiently than raster images. However, the API is not designed for this use case — it is an interface specifically for map editor programs to use. As such, you can only download small geographical areas at a time, and the data is not stored in an optimal format for rendering. For example, complex structures, such as buildings with internal courtyards, are represented in OSM with multipolygon relations, which are difficult to parse and render correctly when using raw OSM data. An additional barrier to using the editing API is that its Terms of Use explicitly prohibit read only applications [10].

Cruiser follows a more promising approach [11]. Instead of downloading rendered image tiles, it uses prebuilt tiled vector map files, in the Mapsforge binary format [12]. This is much more space efficient — the whole of England is an 805 MB download. It is also much less difficult to host, as you don't need a powerful server to render the raster map tiles on the fly. One possible issue with this approach is that this may be too much data to store in a HTML5 web application, and this is something I will have to explore further.

Offline maps are a niche market, as desktop computers tend to always have an internet connection, and on mobile devices data is cheap enough to allow downloading some small vector or raster map tiles. Hence, the main use-case for an offline HTML5 map would be for when an internet connection is either prohibitively expensive or not available at all. This could be when roaming abroad, or when in an area without LTE coverage. As such, it makes sense to make support for mobile devices a priority, as they are the most likely to be in these situations.

From this project I hope to learn how map data actually becomes a rendered map, as my

current understanding of this process is limited. I also hope to learn more about offline HTML applications, as I think that this type of web app has lots of room to become popular in the coming years. There are lots of apps on my phone that I think could be replaced by a PWA (progressive web app).

1.1 Literature review

Whilst researching, my primary source for information about the OpenStreetMap project was *OpenStreetMap — Using and Enhancing the Free Map of the World*, by Ramm, Topf and Childon [1]. This provided a good foundation of knowledge that supplemented information that I have picked up over the years from contributing to the project. Some sections were a little out of date, especially the sections on editors and tools for mappers. Notably the online editor referred to here, Potlatch, is no longer available, being superseded by iD in 2013.

For more up to date or specific information I often relied on the OpenStreetMap Wiki [13], which provides a helpful reference for both OSM specific information and other GIS adjacent topics. These include general information about map projections [14], and details about the Z/X/Y tiling scheme that is common for web maps [15].

When it came to implementing the project, MDN Web Developer documentation proved invaluable [16]. They provide an excellent reference on how to use many web APIs, with detailed usage guides included. In particular, the information about the Canvas API [17], and Service Workers [18] was very useful, as these were technologies that I was not familiar with.

1.2 Aims and objectives

Here I will list some specific features that I would like to implement in the project. These are not fixed, and some may not be implemented, or others added in their place:

- Download vector map data for a user-provided region (preferably at least as large as a UK county), allowing the user to browse the map offline.
- Allow the user to pan and zoom the map. They should be able to zoom out to the view the full extent of the downloaded data.
- Provide a search functionality, using the Nominatim API
- Allow the user to route between two points, which could be implemented using OSRM, GraphHopper or Valhalla.

- When an OSM element has been tagged with a reference to a Wikipedia article, it should show a description from Wikipedia.
- Allow the user to save and name markers for later use.
- When online, the application should allow the user to browse a map without having to first download any data. This could be done using raster tiles.

1.3 Deliverables & timeline

Following a predetermined timeline will help my project proceed without any major unexpected delays, and will give me targets to aim for. At the beginning of term 1 I focused on exploring any risky areas or technologies that I was unsure about. This helped answer any large questions early on in the process.

1. Report on offline HTML5 technologies. Used this to discover whether it is possible to download & store a large amount of vector map data (>100 MB) for later rendering.
2. Proof of concept basic offline HTML5 app. Used this to discover any limit of how files can be stored for offline use.
3. Report on the different ways that the program could get OpenStreetMap data. One possibility is the Mapsforge format, or it could use the OSM editing API. This helped mitigate risk of using a technology that is not suitable for the project.
4. Report about how map projection works. Specifically the mathematics behind projecting the data that is produced by OpenStreetMap. This ensured that I understood this key concept.
5. Proof of concept that took some way made up of latitude longitude pairs and draws a line onto a canvas.
6. Make proof of concept 5 interactive, by allowing the user to pan and zoom the map. This should be done with the scroll wheel on desktop and pinch zooming on mobile.
7. Proof of concept that loads some actual OSM data using the technology that I decided upon in deliverable 3.
8. Thoroughly test the application on mobile, as this will likely be the main use case for an offline map.
9. Add online search functionality, using the Nominatim API.
10. Add online routing functionality, using OSRM, GraphHopper or Valhalla.
11. Show point of interest information from Wikipedia, when an OSM element has been tagged to allow this.
12. Add the ability to save and name markers for later use.
13. Allow the user to browse the map without first downloading data when online. This could be done using raster tiles.

1.3.1 Term one

- **Week 3 (2022/10/03)**: Report 1.
- **Week 4 (2022/10/10)**: Proof of concept 2 & report 3.
- **Week 5 (2022/10/17)**: Report 4 & proof of concept 5.
- **Week 6–7 (2022/10/24)**: Proof of concept 6.
- **Week 8–9 (2022/11/07)**: Proof of concept 7.
- **Week 10–11 (2022/11/21)**: Prepare for the interim report, and presentation.

1.3.2 Term two

- **Week 1–2 (2023/01/09)**: Integration of the above proof of concepts into the final program.
- **Week 3–4 (2023/01/23)**: Deliverable 8.
- **Week 5 (2023/02/06)**: Prepare an initial draft for the final report.
- **Week 6–7 (2023/02/13)**: Add support for further features, such as deliverables 9, 10, 11, 12 and 13.
- **Week 8 (2023/02/27)**: Evaluate the solution so far, and decide whether to extend the project further, if time permits.
- **Week 9–11 (2023/03/06)**: Prepare for the final report.

2: Research

Here I present the research reports I conducted throughout the development process. These helped me to discover which technologies would be most appropriate for my project, and how I could implement them in my proof of concepts.

2.1 Basic web technologies

Websites are commonly made up of 3 main components: HTML, CSS, and JavaScript [19]. HTML is used to create text, images, videos, and other non-interactive content. CSS is responsible for styling, colours, sizing, and other visual effects. JavaScript is used to add interactivity.

While I will need some HTML and CSS, my project is focused on creating an interactive map. Therefore, much of my work will be done in JavaScript, specifically heavily utilising the Canvas API [17]. Instead of using vanilla JavaScript, I have chosen to use TypeScript, which is a superset of JavaScript that adds a compilation/stripping step, where types are statically checked to prevent runtime type issues (this is discussed more in Section 5).

In Listing 2.1 you can see an HTML example that does little other than create a canvas element, and load an external script. This script can then get a reference to the `map` canvas element in the DOM (e.g., with `document.querySelector("#map")`), and use this to create and issue calls to the canvas rendering context.

Listing 2.1: Basic HTML to run an external script with some basic styling

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
5      <title>Proof of concept: 1</title>
6      <style>
7        body {
8          font-family: sans;
9        }
10     </style>
11   </head>
12   <body>
13     <h1>Proof of concept: 1</h1>
14     <canvas id="map"></canvas>
15     <script src="dist/bundle.js"></script>
16   </body>
```

```
17 </html>
```

To run your multiple JavaScript files in the browser, you can either use plain ES Modules [20], or add a bundling step, where all the code is combined into a single file. Using ES Modules in the browser allows you to avoid a build step, but has the negative that it requires n round-trips to the server. For example, if `main.js` imports `map.js` and `map.js` imports `util.js`, your browser will need to make 3 separate requests to the server, and script execution will be blocked until the last dependency has been fetched and evaluated [21].

Bundling, in its simplest form, involves concatenating all the source files together into a single file. This neatly avoids the round trip problem, whilst still allowing you to store your source code in separate files. Advanced bundlers also support features like minification, where variables are renamed to be shorter, whitespace is removed, and unused functions are purged. `esbuild` is a high-performance modern JavaScript/TypeScript bundler, which is written in Go [22].

2.2 Offline HTML5 applications

In order to create an offline HTML5 map application, I will first have to research how offline HTML5 applications work, and what technologies I'll need to become familiar with.

There are various different APIs that one can use to store data in an offline HTML5 application, such as IndexedDB, the File and Directory Entries API [23] [24] and `localStorage`. `localStorage` will likely not be appropriate for my use case, as it is designed for only small amounts of data. There is also the Application Cache, but this has been deprecated, and support has been removed from all major browsers [25] [26]. Therefore, the Application Cache will not be suitable for a new application.

If I am to process and use a large precompiled map data format like mbtiles [27] or Mapsforge [12], using the File and Directory Entries API will likely be more appropriate. For optimal performance, it may be necessary to have a loading step I transform from one of these formats into the IndexedDB, for usage by the app. Hopefully this step can be avoided, as it would not be ideal to have multiple copies of the data, particularly on space constrained devices.

> "The File and Directory Entries API simulates a local file system that web apps can navigate around. You can develop apps that can read, write, and create files and directories in a sandboxed, virtual file system."
>
> — MDN Contributors. *Introduction to the File and Directory Entries API*. 2022. URL:

https://developer.mozilla.org/en-US/docs/Web/API/File_and_Directory_Entries_API/Introduction (visited on 12/10/2022)

Another benefit of the File and Directory Entries API [23], is that it has the ability to act like a local file system. This would allow my app to have a flow like:

- User opens the webapp.
- Selects map to download from list (or as an enhancement by zooming in on desired region on a low detail world map).
- App downloads the map file to sandboxed filesystem.
- User goes offline.
- Map rendered from this internally stored file.

A flow like this makes sense from a user's perspective, as it follows a similar paradigm to other offline map viewers, like Organic Maps or OsmAnd (as discussed in the initial plan).

If there are issues with using the File and Directory Entries API, for example file size limits in certain browsers, it should be possible to use the plain File API instead [28]. The File API allows access to read single files from a user's local filesystem. Importantly for my use case of loading a large map file from a user's disk, we cannot store the entire file in RAM at once. Usefully, the File API provides a `.slice()` method, which allows you to work with a smaller subsection of the file. This will likely be important to avoid out-of-memory (OOM) errors.

Using the plain file API will, however, come at a detriment to the user flow. Instead of being able to select and download a map within the app, the user will have to download the map file themselves, then point the app to the map file they have downloaded. This is not a major issue, but it would be preferable to have this process handled without user interaction.

Unfortunately the Files and Directory Entries API has limited support in Firefox [29]. Critically `window.requestFileSystem()` is not supported, and this is the function that you call to gain access to a virtual filesystem. In Firefox this API can only be used through an `<input>` element, or drag and drop. This is an issue as for my application, the app would need to be able to create the virtual filesystem programmatically.

> "Content scripts can't create file systems or initiate access to a file system."
>
> — MDN Contributors. *File and Directory Entries API support in Firefox: Limitations.* 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/File_and_Directory_Entries_API/Firefox_support#limitations_in_firefox (visited on 17/10/2022)

Hence, if I wish to use the File and Directory Entries API, which would provide a smoother user journey, I would not be able to support Firefox. For my application, I think it

is more important to support Firefox than have perfect UX — so I will settle on the compromise of the user having to select the desired map file on each start, via the file picker. This will be an irritation, but unfortunately there is no widely compatible way to use a sandboxed/virtual filesystem in the browser.

One benefit of storing the map file outside the browser is that my application will not have to deal with the browser not allocating it enough storage quota for the file. For example, in Chrome you must use the Quota Management API to request persistent storage space to use [30], which will prompt the user to accept the request.

Another option is to use the Cache API. The Cache API is part of the Service Worker specification [31] [18], and has good cross-browser support. It has a simpler interface that should work for the application — a pair of a request and a response object are stored, and you get a cached version of the response by providing a similar request.

To determine which will be the best choice for the application, I will produce proof of concepts that employ the File API and the Cache API.

2.3 Using the HTML5 canvas

The HTML5 Canvas provides another dimension to web applications, where bitmap animations or other data can be displayed interactively.

> "HTML5 Canvas is an immediate mode bitmapped area of the screen that can be manipulated with JavaScript. Immediate mode refers to the way the canvas renders pixels on the screen. HTML5 Canvas completely redraws the bitmapped screen on every frame by using Canvas API calls from JavaScript."
>
> — Steve Fulton and Jeff Fulton. *HTML5 Canvas: Native Interactivity and Animation for the Web*. O'Reilly Media, 2013. ISBN: 9781449335885. URL: `https://books.google.co.uk/books?id=zLUyKvtdCQwC`

So, in my application, the canvas is what the map will be "drawn" to. The canvas API provides different interfaces that you can use, known as rendering contexts [17]. For simple 2D drawing, there is the `CanvasRenderingContext2D` interface. For more intensive 2D and 3D workloads, there is the GPU accelerated `WebGLRenderingContext`, which uses an OpenGL-like system of vertex and fragment shaders [33]. This is what some existing vector web map libraries use, like Mapbox's GL JS [34].

In the long run, the `WebGLRenderingContext` would be the best choice in terms of efficiency and performance, as it is hardware accelerated. However, the `CanvasRendering-`

`Context2D` has a far simpler API, so it makes sense to use it here.

The simplest operation I will need to use the canvas for is drawing the outlines of polygonal shapes. Note that on the canvas, `(0, 0)` is at the top left. To begin drawing, you first need to get an instance of `CanvasRenderingContext2D`, which is done by calling `const ctx = canvas.getContext("2d")` on some canvas element in the DOM. To start on the polygon, you must call `ctx.beginPath()` to create a new path, that you then add data to using the `ctx.lineTo(x, y)` method. If required, you can also call `ctx.moveTo(x, y)` to have the line jump from one spot to the next. Once you have made your `.moveTo()` and `.lineTo()` calls, you then execute `ctx.stroke()` to render the line out.

Another operation I will need to perform to create a styled map, is creating "filled" polygons. For example, openstreetmap.org renders buildings with a brown infill and a darker outline. This is achieved on the canvas in a very similar way to the polygon outline, but instead of calling `ctx.stroke()` to render the path, you call `ctx.fill()`. To control the colour, you set `ctx.fillStyle` to some desired value.

To render names of points of interest and ways to the map, I will need a method to write text to the canvas. This can be done using `ctx.fillText('string', x, y)` [35]. Unfortunately, this does not make provisions for drawing text at an angle. This would be useful for drawing names of roads or boundaries, where the name follows the curve of the geometry. However, you can use the `ctx.rotate(angle)` to achieve a similar result, albeit with some more complexity [36].

Another useful method for drawing a map will be the `ctx.measureText('string')` method, which returns the pixel size that the text will be drawn at. This information can be used to prevent labels on the map overlapping, by storing each of the previously drawn labels' boundary boxes, and preventing new labels from being drawn if they are colliding.

Finally, the `Path2D` interface may be useful for my application [37]. It allows you to save a path, meaning you can stroke and fill some geometry, without having to draw the path twice. Hence, it should give an efficiency boost. You use it like so:

```
1 const ctx = canvas.getContext('2d');
2
3 let path = new Path2D();
4 path.rect(5, 5, 20, 20);
5
6 ctx.stroke(path);
7 ctx.fill(path);
```

2.4 OpenStreetMap data sources

In order to produce a map using OpenStreetMap data, you first need to decide which source/format of data to use. Traditionally, for this project, students use the OpenStreetMap editing API. This however, has a number of limitations, some of which I have already discussed in the project plan:

- It is designed for map editing applications, and the terms of service explicitly prohibit read-only uses [8].
- You can only download a relatively limited geographical area, usually only around 2 km^2. This has the effect that you will not be able to zoom out to see a large area, such an entire country [38].
- Complex structures (like buildings with internal courtyards) represented in OpenStreetMap as multipolyons require extensive parsing and validation to correctly display.

For online maps, it is common to use either raster or vector “map tiles”. Raster map tiles are usually 256×256 `.png` files, rendered by a Mapnik server. These tiles are named according to their zoom level, and an x and y value, where x and y are offsets from the top left most tile [15]. Raster tiles, however, are not particularly appropriate for offline usage, as you have to request an extremely large amount of tiles, especially when you approach high zoom levels. For example, if you downloaded 4 tiles at zoom 10, you would need to download 16 at zoom 11, 64 at zoom 12, and 262144 at zoom 18. This behaviour is unacceptable as rendering tiles is computationally expensive for the OpenStreetMap Foundation, and the size of these tiles adds up.

Vector tiles use a similar scheme, with tiles also being served at Z/X/Y addresses. The key differentiator is that instead of rendered images, vector data is served to clients, and it is up to the clients to render this data into a map. This gives the clients flexibility in how they can choose to display the data, such as the colours, labels, and line thicknesses. This is usually done through stylesheets. As vector data does not pixelate like raster images, you can “overzoom” on vector tiles, meaning you do not have to serve tiles to such high zoom levels. One of the most popular formats for vector map tiles are Mapbox Vector tiles. These use the Google Protobuf format to store the vector data.

Vector tiles make a more appropriate offline format than raster tiles, due to their ability to “overzoom”, and that vector data can be stored more efficiently. Unfortunately, it is still not particularly suited to offline usage, as with raster tiles, you need to download many tiles to cover a large region.

Therefore, it makes sense to use a dedicated storage format, that is designed to provide map data for offline applications. The OpenStreetMap Wiki provides a number of possible

options for this purpose [39]. The most popular choices are mapsforge [40], which is used by many applications [41], and MBTiles (by Mapbox) [42]. Unfortunately, as the MBTiles format is based on an SQLite database, it can't easily be used in a HTML5 app.

Mapsforge, on the other hand, uses a custom binary data format [12]. Mapsforge only provide libraries for the Java language and the Android platform. There is a partial implementation available of a parser written in JavaScript, which only supports decoding the headers and other metadata. Hence, to use this format for my project, I would need to implement my own parser, which would add substantial complexity. I may also need to tackle issues with memory usage, as parsing and rendering the file will require random access within it — meaning loading the whole file into an `ArrayBuffer` or some similar structure. Using the `Blob.slice()` method in JavaScript may allow me to work around this issue to some degree.

If parsing the Mapsforge binary files is too complex, I can scale back the project to using the OSM Editing API. This will require less parsing work, but will still have plenty of issues to overcome in terms of rendering.

2.5 Projecting map data

The core of this project is being able to render a map. This, in its most basic form, means being able to take coordinates, which represent a position on the sphere-like body that is the earth, and placing these on a two-dimensional plane. This process is known as map projection [43, p. 5].

OpenStreetMap uses the WGS84 coordinate reference system (EPSG:4326) to represent the positions of the nodes that make up its data [44]. GPS uses this CRS, and it is a popular standard [45].

If you were to naively place coordinates from the WGS84 system on to a graph, you would get a "plate carrée" projected map. This projection is a form of the equirectangular projection, where the standard parallel $\phi_1 = 0$. In plate carrée, lines of longitude are straight, vertical and equidistant, and lines of latitude are similar, albeit horizontal. Although all map projections are compromises between conformality (preservation of angle) and preservation of area, plate carré does not preserve either, meaning it is not particularly well-used.

Hence, `openstreetmap.org` presents a map in the "Web Mercator" projection (also known as Spherical Mercator, WGS 84/Pseudo-Mercator, Google Web Mercator, EPSG:3857). This projection is almost conformal, meaning that (local) angles on the map are the same

as angles on the ground, even if lengths are not preserved [46]. It deviates from normal Mercator in that its calculations are based on the earth being a true sphere, rather than ellipsoidal. The benefit of this is that the calculations for converting between WGS84 latitude/longitude coordinates, and Web Mercator are simpler than normal Mercator [47].

So, to convert from latitude/longitude coordinates to Web Mercator, we can use equations 7–1a & 7–2a from Snyder [48, p. 41], where λ is longitude and ϕ is latitude, both in degrees:

$$x = \frac{\pi R(\lambda^\circ - \lambda_0^\circ)}{180^\circ}$$

$$y = R \ln \tan \left(45^\circ + \frac{\phi^\circ}{2} \right)$$

3: Proof-of-concept development process

As part of the development process, I have created a number of proof of concepts, that each build on each other. These have helped me to understand, test, and evaluate various technologies that I could use to build the final application.

3.1 Simple offline HTML5 app

Source in `proof-of-concepts/1-offline-html5`

My first proof of concept was a basic offline HTML5 application, employing the techniques that I researched in Section 2.2. Since the `Application Cache` has been deprecated, the accepted way to create an offline HTML5 app is to use Service Workers [31]. The concept behind this is that the Service Worker intercepts all HTTP requests that are made on the website, and can then respond to this request with a result from a prepopulated cache.

In addition to creating an offline HTML5 app, I also learned how to store a large binary blob. This will be necessary for my project, as this is how the map data will be stored. To this end I looked at using both the Service Worker cache, and the File API.

The File API works like a traditional file picker — allowing the user to choose some file from their local filesystem [28]. This is useful, but does not provide an excellent user experience. For example, the user would have to download a map file to some location on their computer, then select it in the file picker every time they want to view a map. This violates Nielsen's "Recognition, not recall" heuristic [49], as the application should work without the user having to remember where they saved the map file.

Therefore, it is preferred to use the Service Worker cache method, where you store the file inside the web browser's cache. You can store binary blobs in the same way as any other website resources:

Listing 3.1: Using the Service Worker cache

```
1  self.addEventListener('install', (event) => event.waitUntil(
2          addResourcesToCache(['/index.html', '/blob.map']) // prepopulate cache
3  ));
4
5  const cacheFirst = async ({ request }) => {
6      const responseFromCache = await caches.match(request);
7      if (responseFromCache) { // try to get the resource from the cache
8          return responseFromCache;
9      }
```

```
10
11      return new Response('could not retrieve from cache', {
12              status: 408,
13              headers: { 'Content-Type': 'text/plain' },
14          });
15  };
16
17  // the fetch event will intercept all HTTP requests made by the website
18  // such as JavaScript fetch() calls, or HTML <img> tags
19  self.addEventListener('fetch', (event) => event.respondWith(
20          cacheFirst({request: event.request})
21  ));
```

3.2 Rendering geometry to a canvas

Source in `proof-of-concepts/2-rendering-a-way`, online demo at `files.george.honeywood.org.uk/2-rendering-a-way/`, or demo video.

My second proof of concept was to draw some geographical data onto the canvas. As a first step, I had to acquire some data to project. As this was a simple proof of concept I decided to use the GeoJSON format, which is a popular standard for representing geodata, in a fairly human-readable format. The main reason I chose it was that it is easier to parse in JavaScript than the XML data that the OpenStreetMap API provides [9]. For my first test, I attempted to render the boundary of Egham, which is currently way 666914693. To convert XML data from OpenStreetMap I used the online `geojson.io` tool, which allows you to import OSM XML and export GeoJSON [50].

The main challenge I wanted to solve in this proof of concept was projecting the data. This stage is necessary, as OpenStreetMap data comes in the WGS84 CRS, and while this can be naively plotted on a graph as (x, y) coordinates, this would result in the “plate carrée” projection, as detailed in Section 2.5. The “plate carrée” projection is not particularly desirable, as it does not have any useful properties like conformality (preservation of angle) or preservation of area. Hence, for my application I chose to use Web Mercator, which preserves conformality, and is the de facto standard for web maps.

Initially, I attempted to use the algorithm from Snyder [48, p. 41], as shown in Section 2.5. Unfortunately, I had some issues with getting these equations to produce reasonable results. Therefore, I turned to the OpenStreetMap Wiki, which helpfully provides a reference for transforming WGS84 data into the Web Mercator projection [14]. I translated the C example into TypeScript, and it correctly projected the data — see Listing 3.2 for the implementation.

Listing 3.2: Projecting to Web Mercator

```
1 const RADIANS_TO_DEGREES = 180 / Math.PI;
2
3 // calculate x/long
4 const x = long;
5 // calculate y/lat
6 const y = Math.log(Math.tan(
7     (lat / RADIANS_TO_DEGREES) / 2 + Math.PI / 4
8 )) * RADIANS_TO_DEGREES;
```

Figure 3.1: Rendering some GeoJSON data to the canvas

This proof of concept was entirely static, with the viewport and zoom level hardcoded into the program. To make the implementation simpler I wrote a function that scaled all the coordinates to be around (0,0), so that I didn't have to handle offsetting the viewport.

3.3 Adding interactivity — panning and zooming

Source in `proof-of-concepts/3-panning-and-zooming`, online demo at `files.george.honeywood.org`.

uk/3-panning-and-zooming/, or demo video.

The third proof of concept heavily built upon the second — with the additional goal of adding interactivity in the form of panning and zooming. Since being mobile friendly is a priority for my project, I made sure to add touch controls once I had it working with mouse events.

The first part I tackled was the zooming. This turned out to be as simple as multiplying each projected coordinate by some scale factor, which I was already doing in the previous proof of concept. To modify this scale factor, the user can use the +/- buttons, the mouse scroll wheel, or a pinch gesture on a touch device. Pinch gestures were the most challenging to implement, as you have to handle and interpret `Touch` events for each separate finger on the screen.

Panning also turned out to be relatively easy — my implementation involved adding some offsets to the projected and scaled coordinates in the latitude and longitude axes. More involved was zooming the map about some arbitrary point. For example, on desktop, you expect a map to zoom into the position of your mouse cursor, when on mobile you expect the middle of your pinch gesture. To achieve this, I update the `x` and `y` offsets whenever you zoom, with a smaller offset change when zooming into the top left, and a larger one for the bottom right. Another issue I had to handle is that at a low zoom level, zooming was relatively quick, but once you reached higher zoom levels, it got slower and slower. To compensate for this I made the scale factor logarithmic.

Figure 3.2: An interactive world map, using Natural Earth data [51]

For testing the zooming between low and high zoom, I needed some "large scale" data to supplement the localized OSM data that I had downloaded for the previous proof of concept. Natural Earth Cultural Vector data seemed suitable, as it is readily available as GeoJSON, is released in the public domain, and is suitably low-detail for my purpose [51].

Once again I used `geojson.io` [50], this time to merge these two sets of data.

To begin with, I was simply redrawing the map every time a `Touch` or `Mouse` event fired. This worked, but these events would often fire very rapidly (>60 times/second), resulting in a slow panning experience, where rendering was occurring unnecessarily. My solution for this was `requestAnimationFrame()`, which allows you to have some function executed at a regular interval by the browser [52]. Therefore, I could modify the state of the map as often as necessary, then have it smoothly rendered at regular intervals to reflect any changes to zoom or offsets.

Unfortunately, this change meant the canvas re-rendered 60 times a second at all times, even when no state change had occurred. This is a waste of processing power and energy, especially on mobile devices. To remedy this I introduced a `dirty` flag, which I set whenever a state change had occurred. I then check this `dirty` flag at the beginning of the `render()` function, and return early if the internal state is unchanged — as shown in Listing 3.3. The end result was a map that updated smoothly 60 times a second when state changes were occurring, and not at all when the map was static.

Listing 3.3: Only rendering when the map state is `dirty`

```
1  public render() {
2      // if nothing has changed, don't bother re-rendering
3      if (!this.dirty) {
4          requestAnimationFrame(() => this.render());
5          return;
6      }
7      this.dirty = false;
8      [...]
9  }
```

3.4 Rendering tiled data from a Mapsforge file

Source in `proof-of-concepts/4-rendering-osm-data`, online demo at `files.george.honeywood.org.uk/4-rendering-osm-data/`, mobile demo video, or desktop demo video.

My final proof of concept revolved around reading tiled vector OSM data. This allows for a map viewer that is performant at a wide range of zoom levels, as it can switch to more detailed data as you zoom in, and less detailed as you zoom out. It also solves the issue of which data to render — my previous proof of concepts rendered all the data, all the time, even if it was off canvas. With tiled data, you only have to render the tiles that are currently within the viewport.

The bulk of the work for this proof of concept came in the form of writing a parser for the

Mapsforge format [12]. I decided to use this format based on my report on OpenStreetMap data sources, in Section 2.4. This was because it seemed simpler to parse than MBTiles, which is the other popular choice for tiled vector map files. It is a binary format that is designed to space-efficiently encode geographical data, such that it can be rendered on low-power mobile devices.

3.4.1 The Mapsforge format

Here I will explain the main concepts of the Mapsforge format, that I had to understand whilst writing the parser. This information is mostly as per the specification [12], supplemented by what I learnt as I progressed.

Points of Interest (PoIs)

These represent tagged OpenStreetMap nodes, such as shops, bus stops, or any other objects that have been mapped as a node. They are also used for city/town/place labels at low zoom levels.

Ways

Ways are an abstraction over OpenStreetMap ways and relations, presenting both as a single type. OSM represents complex structures such as buildings with internal courtyards as multiple ways, for the outer and inner parts. These are then linked by a multipolygon relation that labels each part as inner or outer [53]. These are non-trivial to efficiently interpret, especially with large amounts of data, so it makes sense to push this complexity to the map-writer software.

Indexes

In order to retrieve a specific tile of data, we need to know where in the file it is stored. This is achieved through a set of indexes for each sub file. Each index entry stores a pointer to the start of a tile. The indexes are stored as row major 5 byte integers, with the first value being for the first tile (x, y) $(0, 0)$ in the sub-file. You can then read the $(1, 0)$ index by adding a 5 byte offset, or the $(0, 1)$ index by adding 5 $\times$ the amount of x tiles in the sub-file.

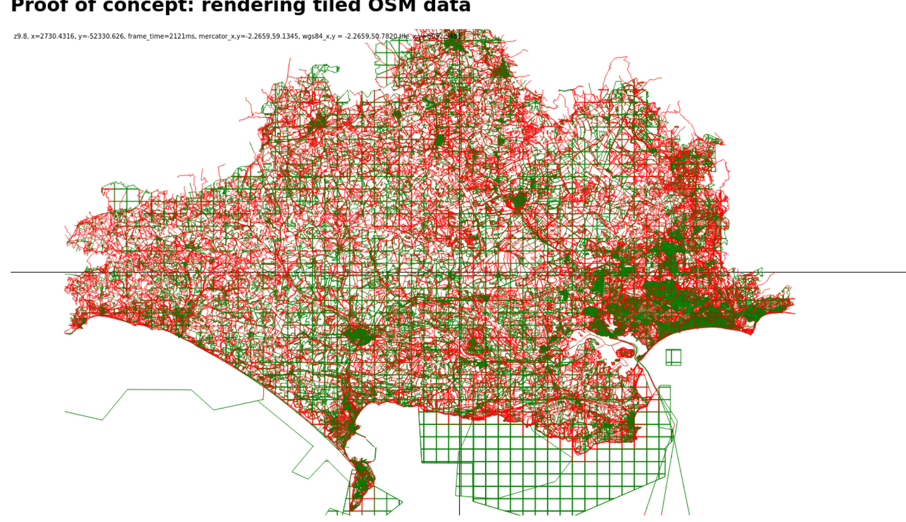

Figure 3.3: Raw tiled data from zoom 14 base tiles

Sub-files

Sub-files store map data for a range of zoom levels. For example, the file `proof-of-concepts/4-rendering-osm-data/data/ferndown.map` is compromised of three sub-files, one for z1-z7, based at z5, another for z8-z11, based at z10, and a final one for z12-z21, based at z14. This system provides a compromise between storage space and geographical correctness — if, say, we want data for z18, we "over-zoom" the z14 sub-file, and vice versa for z1, we "under-zoom" the z5 sub-file. Therefore, we don't need to store much duplicated data, compared to storing each zoom level separately.

The over-zoom/under-zoom functionality is supported by each sub-file containing a "zoom table", which contains how many PoIs/Ways need to be read for the tile at the requested zoom level. For example, when under-zooming a z14 tile to z12, you will be showing around 112 z14 base tiles (on a 1080p screen). This is too many tiles to be able to render all of their features with acceptable performance. To enable this zoom-table feature, the Way and PoI data in each tile is sorted by "feature importance", meaning that data to be shown at low zoom levels, like place labels or important roads, comes first, and less important geometries like houses are placed last. Therefore, you can read that there should be `x` amount of PoIs/Ways for this tile, for the zoom level you are decoding, read the specified amount, then skip over the rest.

Variable length integers

To save file space, coordinates and other numbers are encoded in a custom format. This means that both large and small numbers can use the same representation, whilst requiring

a minimal amount of bytes to store them. The format is based around the idea that the first bit of a byte is used as a continuation bit — i.e., in `1000 0000` the continuation bit is set. If the continuation bit is set, you continue to read the next byte as part of the same number, until you reach an unset continuation bit.

The format is little-endian, meaning that the first byte of the number is the least significant[1]. Hence, you can read the whole number by shifting each of the subsequent numbers by $7n$, then applying a bitwise OR (represented by `|` in JavaScript) against the current total. I've provided the commented implementation in Listing 3.4, as the code is easier to understand than my prose explaination.

Listing 3.4: Parsing variable length integers

```
1  // decode a variable length unsigned integer as a number
2  getVUint() {
3      // if the first bit is 1, need to read the next byte rest of the 7 bits
4      // are the numeric value, starting with the least significant
5      let value = 0;
6      let shift = 0;
7
8      // check if we need to continue
9      while ((this.data.getUint8(this.offset) & 0b1000_0000) != 0) {
10         // if this not the first byte we've read, each bit is worth more
11         value |= (this.data.getUint8(this.offset) & 0b0111_1111) << shift
12         this.offset++
13         shift += 7
14     }
15
16     // read the seven bits from the last byte
17     value |= (this.data.getUint8(this.offset) << shift)
18     this.offset++
19     return value
20 }
```

Signed integers follow a similar scheme with the continuation bit, but also sacrifice the second bit of the last byte to indicate the sign of the number, e.g., `0100 0000`. There is no concept of a floating point number in the format. Instead, you divide numbers when it is required, such as for coordinates, which are stored in the file in microdegree units.

3.4.2 Writing a parser

In my first iteration of this proof of concept, I was decoding the file inline, using a `DataView`, manually keeping track of offsets within the blob. This worked, but led to a lot of duplicate code that was just handling offsets, and a number of errors caused by

[1] Note that the bytes themselves are still stored in big-endian format

me making some trivial mistake when incrementing the offset by the wrong amount. To remedy this smell, I refactored the code to use the abstraction of a `Reader`. You construct the `Reader` with a blob, then call methods like `.getVUint()`, or `.getBigUint64()`. These methods increment an offset internal to the `Reader` class.

3.4.3 Rendering the tiles

Proof of concept: rendering tiled OSM data

Figure 3.4: Styling of data from Mapsforge file

When rendering the map, I first convert the current zoom level to the corresponding base zoom level of one of the sub-files. Once I have this I then calculate the coordinates at the top-left and bottom-right of the screen, then convert these into Z/X/Y tile numbers, which I can then fetch from the file. To improve rendering performance, only the PoIs and Ways specified in the zoom table for that zoom level are shown.

I handled styling fairly simplistically, setting a number of booleans based on way tags, as the tiles load in. This moves the more complicated (and slow) logic out of the rendering hot path. Inside the `render()` function I then check these booleans and choose whether a stroke or fill should be used for that way, and what colour it should be. This is shown in Figure 3.4.

To make sure that the map always draw labels on top, I make two passes through the data. In the first, I render out all the areas and lines, then in the second, the way labels and PoIs. Before I implemented this, areas from other tiles would often overlap labels.

4: Final application development process: OSMO

Source in `final-deliverable/`, online demo at `files.george.honeywood.org.uk/final-deliverable/`, or watch the demo video.

In the second term, I worked on the final application, named OSMO. This started off as a direct clone of the final proof of concept — as discussed in section 3.4.

4.1 Dynamic tile loading

The final proof of concept (section 3.4) rendered data from a preloaded Mapsforge file, requiring the whole map to be downloaded into RAM before any tiles could be drawn. This works fine for small map files, but is not reasonable for larger ones — the map file that covers England is around 800 MB — which is too much data to download and store in RAM, especially on mobile devices.

To circumvent this issue, we can use HTTP range requests, which allow us to request only certain chunks of a file from a web server [54]. Range requests are natively supported by popular web servers like Apache and nginx. This approach is a little unusual, typically vector/raster tile based online maps utilise a dedicated server application known as a tile server. The tile server takes a web request shaped like `example.org/z/x/y`, and then responds to this with data for the requested tile.

Listing 4.1: Using `curl` to make a HTTP range request

```
% curl -v 'https://files.george.honeywood.org.uk/final-deliverable/data/england.map' \
    -H 'range: bytes=0-23'

> GET /final-deliverable/data/england.map HTTP/2
> Host: files.george.honeywood.org.uk
> range: bytes=0-23

< HTTP/2 206
< server: nginx/1.18.0 (Ubuntu)
< content-length: 24
< content-range: bytes 0-23/844019294

[... 24 bytes of data ...]
```

The first 24 bytes of the file contain the magic bytes, then length of the rest of the header. Once we have this, we can then request the rest of the header, and based on this, the tile indexes. We then use these parts of the Mapsforge file to calculate which tiles the file

contains, and their offsets within the file.

The Mapsforge format is not specifically designed for this type of use. Instead, data is typically read from a locally stored file by a native application. This range request approach was inspired by the PMTiles project, which stores tiles in "pyramids built on compressed Hilbert ordering", allowing for efficient random access to tiles [55]. This specifically designed PMTiles format saves initial network traffic compared to using the Mapsforge format, but the overhead is relatively low for my use case. For the 800 MB England map, the file header is only 8.0 KB, and the tile indexes are just over 1.1 MB. The PMTiles approach makes more sense for larger world scale maps, as they will have many more tiles, and hence larger tile indexes.

Additionally, PMTiles is intended as a general purpose map library, whereas OSMO is more specifically designed as a standalone offline map application. This means that we can assume that the indexes will only need to be fetched once, and from then will be cached in the service worker.

The dynamic tile loading implementation is transparent to the parsing code, as it loads the required bytes via an abstracted `fetchBytes(start, end)` function. Depending on how the `MapsforgeParser` was constructed, it will then either read data from a pre-downloaded blob, or dynamically via an HTTP range request.

4.2 Caching dynamically loaded tiles

As discussed in the previous section, the final proof of concept read data from a pre-downloaded map blob. This made offline use quite simple, as the entire blob could be stored in the service worker cache. Working with only partial range requested data, however, requires a more specialized approach, as the service worker cache cannot natively store partial data [31, see Section 5.4.5: `Cache.put(request, response)`].

To work around this limitation I instead decided to store the data with an extra key appended to the URL, which contains the byte range that is stored within that response. For example, if we had stored the two byte ranges in the cache (that make up the file header), we would have two keys in the cache: `england.map?bytes=0-23` and `england.map?bytes=24-8021`. As the same tile will stay at the same byte offset, later, when we are offline, we can use the same key to retrieve the tile data, with `cache.match(url)`.

4.3 Downloading a region

Caching the partially downloaded map file became more complicated when I wanted to add the download region feature. This would allow the user to click the "Download" button, which recursively downloads all the tiles below the current map viewport. The use case for this is that a disk space restricted user will be able to download a subset of the map, without needing to download the entire 800 MB `england.map` file.

My initial, naïve downloading approach was as follows:

1. Calculate which tiles are contained within the current viewport.
2. Generate the byte ranges which these tiles are stored at within the file.
3. Call `fetchBytes()` to load these bytes into the service worker cache for later offline use.

While functional, this approach was less than ideal. For one, this meant storing thousands of individual byte ranges within the cache — an area the size of Cornwall contains about 11,000 tiles at z14. The other main problem was that dispatching hundreds of HTTP range requests to the server was very slow, and it makes much more sense to download larger chunks of the data at once. Conveniently to this end, the Mapforge format stores the tile data for the (x, y) $(0, 0)$ and $(1, 0)$ tiles in a contiguous range of bytes. This means that instead of downloading each tile individually, we can download a whole row of tiles at once.

Taking this request chunking approach meant that the service worker code had to be altered, to allow a partial read of data from within a larger cached range. For example, if we had the range `england.map?bytes=100-200`, we not only need to be able to read bytes 100 to 200, but also any arbitrary subset of that range. The implementation for this was fairly simple (see Listing 4.2).

Listing 4.2: Reading a range from a cached response

```
1  const requested_range = {
2      start: [...] // parsed from Range header in request
3      end: [...]
4  };
5
6  [...] // elided logic to choose which cached range to use
7  cached_range = {
8      start: [...] // parsed from bytes= query parameter in cache key
9      end: [...]
10 };
11
12 return new Response(
```

```
13      (await responseFromCache.blob())
14          .slice(
15              requested_range.start - cached_range.start,
16              // NOTE: ranges in the cache are end exclusive,
17              // so need add one here to get the final byte
18              (requested_range.end - cached_range.start) + 1,
19          ),
20      { status: 206 }
21  );
```

One minor complication caused by allowing the partial reads of cached ranges was that we could no longer use the `cache.match(url)` method, as this only matches an exact request URL. Instead, I had to create my own implementation, where I looped over the keys in the cache (using `cache.keys()`) and manually checked if the requested byte range was a subset of one of the cached ranges. This was done with a linear search, which isn't ideal, but the runtime should remain reasonable so long as the number of cached ranges is small.

4.4 Experimenting with WebGL

Whilst profiling the `render()` function of my application, it became apparent that the draw calls on the `<canvas>` were the limiting factor in my applications' performance (see 5.5). Instead of using `CanvasRenderingContext2D`, popular vector map rendering libraries like Mapbox GL use the `WebGL2RenderingContext`. This allows for GPU accelerated rendering, even providing enough power to render complex 3D graphics. This would have been the best way to reduce the time needed to render a frame, and make the application more responsive to panning and zooming.

During the beginning of term two, I decided to experiment with using WebGL for rendering. WebGL2 provides a very low level interface to the GPU, employing the concepts of fragment shaders and vertex shaders. These allow you to draw triangles that you can then use to make up more complex shapes — for example, a square would be made up of two equilateral triangles. You have to handle all of this work yourself, unless you choose to employ a library.

Following a tutorial, I was able to implement basic drawing using the `WebGL2RenderingContext` [56]. This simple experiment rendered a million one pixel wide lines to the screen, which required over 150 lines of code. The implementation can be seen on the `feat/web-gl` branch. This was in stark contrast to the relative simplicity of the `CanvasRenderingContext2D` API. Granted, the performance was very impressive, but it would have taken a significant effort to implement the same functionality as my existing application using the simpler 2D rendering context. Therefore, I decided to continue using the `CanvasRenderingContext2D` and instead focus on adding more functionality to my

application.

4.5 Svelte: using a frontend framework

The search box was the first feature I added that wasn't primarily using the canvas for display and interaction. I initially implemented it in vanilla JavaScript, which worked fine, but I found myself writing a lot of error-prone boilerplate code to update the DOM. In addition to this, I was following the antipattern of creating references to the DOM, then passing around these references to other functions. This made it difficult to reason about the state of the application, and harder to refactor the code.

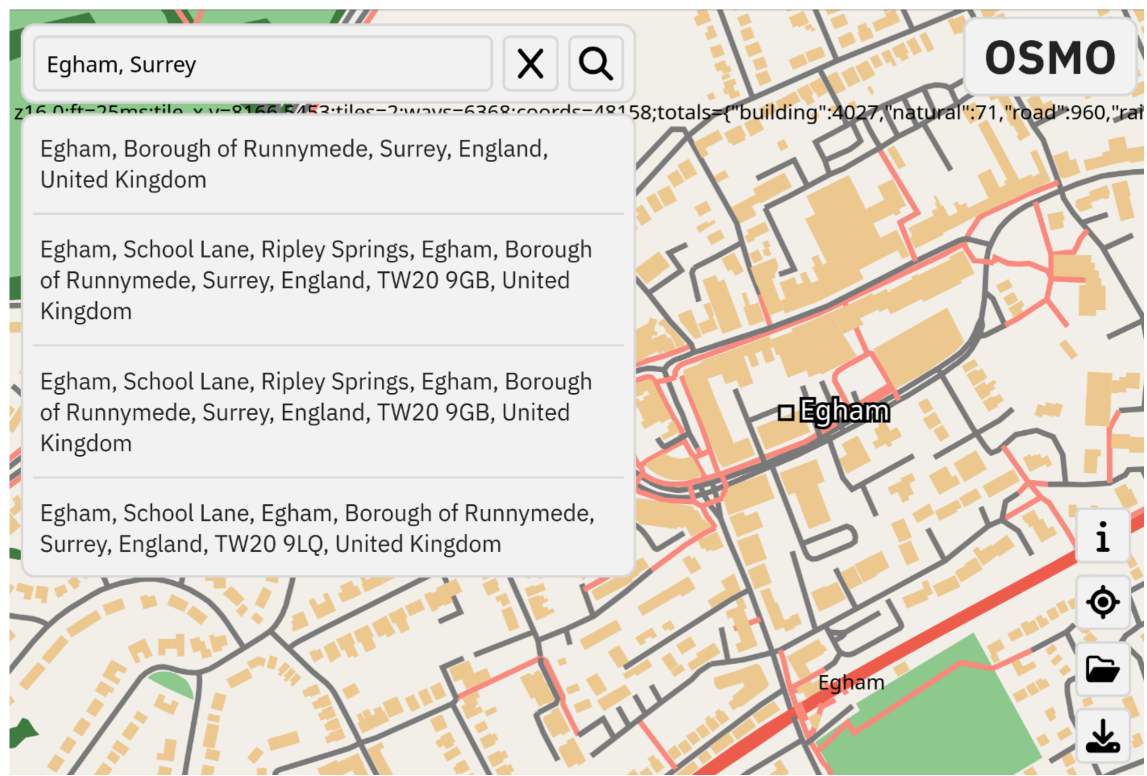

Figure 4.1: Searching for "Egham" in OSMO

To remedy this I decided to start using a frontend framework. This would allow me to write declarative code, and abstract away the DOM manipulation. I chose to use Svelte, as it is relatively lightweight & performant, and I've used it before. Svelte also solved my issue of passing DOM references, by allowing them to be created next to the point of use. This allowed me to write self-contained components, which each manage their own state.

Another benefit of Svelte is that it provides a simple interface to allow you to write animations that aren't normally possible with CSS. For example, in the search box, when

the result box appears, it slides in from the top — see Figure 4.1. This is cannot be done with pure CSS, as CSS animations only trigger when the properties of the element are changed. For example, when setting el.style.height = "200px". In reality, you don't know the height that the results box will be, so this method is untenable. Svelte allows you to overcome this limitation by providing means to write animations that are triggered by changing the state of the component.

Svelte also provides scoped CSS, which allows you to write styles that will only apply to the component. This means you don't have to be careful about giving your DOM elements unique IDs or class names, as they will only be used within the component. This then means you can often get away with just styling a plain div, span or p element, without having to give it a class name.

4.6 Progressive web apps (PWAs)

PWAs are web apps that are able to present themselves as and have similar features to native applications [57]. For example, a PWA could include offline support, the ability to "install" the website on mobile platforms, and features like push notifications. In order to make web app installable, you have to write a web manifest, and have a service worker that provides offline content. The manifest provides metadata about your application, allowing you to control how it will be displayed in the user's app list. In my manifest I provide icons to use, long and short names to use, and a description.

To ensure my application was a valid PWA that could be installed on mobile devices, I used Google Chrome's Lighthouse testing tool. Unfortunately, this did not catch an issue I experienced, which was that my app could be "added to home screen" but it was not installed into the application drawer. When using the Chrome remote debug functionality to see the console output on my phone, I saw a message: Failed to install WebAPK for '<url>'.

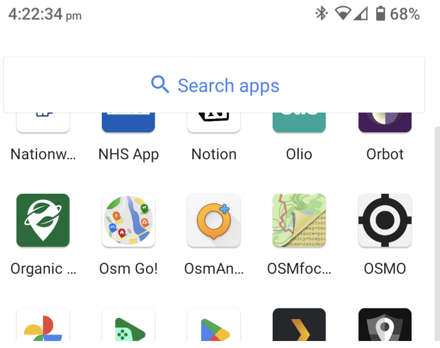

Figure 4.2: OSMO installed in the app drawer of an Android phone

After some searching, I found that PWA installation on Android has a few indirections, principally that clicking "Install" sends a request to a Google server to bundle the website into a WebAPK. Bundling as a WebAPK means your app is treated like a native app, and hence can handle URL ranges or custom URI schemes[1]. Unfortunately, this terse message did not provide any detail into why the installation failed. Eventually after considerable debugging, I found that the issue was that I had provided my icon in the manifest as an SVG, instead of a PNG. See Figure 4.2 for a screenshot of OSMO installed on my phone.

4.7 Service worker quirks

Near the end of the project I spent some time polishing the offline user experience. One of the principal issues was due to a quirk in service worker registration behaviour. When a new service worker installs, it will not control the page that registered it until the next page refresh. This creates the unexpected behaviour that if you download a region on the initial page load, then go offline, you will not be able to see the downloaded region after a refresh. This is because when the service worker is not controlling, it cannot intercept `fetch` events, and so the cache will not be populated. Luckily some escape hatches are provided, such as `skipWaiting()` and `clients.claim()`. These allow you to force the service worker to take control of the page, and intercept requests immediately.

However, this led to a race condition issue — that critical resources would be loaded before the service worker had finished installing. For example, on initial page load, the header & tile index would be loaded before the service worker had claimed the page, meaning it wouldn't get cached.

The simplest way to remedy this would have been to force a page reload when the service worker finished installing, but this would have been a poor user experience. Instead, I added another event listener, for the `controllerchange` event, which is fired when `clients.claim()` is called. Once this event fires, I then instantiate the Svelte app, ensuring that the service worker is ready to intercept and cache the header & tile indexes — see Listing 4.3.

Listing 4.3: Waiting for the service worker to claim the page before starting the app

```
navigator.serviceWorker.register('./sw.js', { scope: './' }).then(() => {
    return new Promise<void>(resolve => {
        if (navigator.serviceWorker.controller) {
            resolve();
        } else {
```

[1] In the case of my application, registering to handle geo URIs would have allowed users to open links to specific locations [58]. Unfortunately I did not have time to implement this.

```
        navigator.serviceWorker.addEventListener('controllerchange', () => {
            resolve();
        });
    }
  });
}).then(() => {
  console.log(`starting app, service worker is ready, took {performance.now() -
      start_time}ms`)
  new App({
      target: document.getElementById('app'),
  })
})
```

4.8 Handling HiDPI displays

One long-standing bug was the map rendering slightly blurry on devices with HiDPI screens, such as mobile phones or 4K monitors. This is due to how the `px` unit works in CSS — they are just a unit of length, not a representation of the physical pixels on the screen. This means that on a HiDPI screen, if the `canvas.width` and `height` are set to the canvas width in CSS pixels, the canvas pixels have will be stretched across multiple physical pixels.

Therefore, to render at the correct resolution, you must use the `window.devicePixelRatio` property to determine how many physical pixels there are per CSS pixel. You can then use this to set the `canvas.width` and `height` to be the same as the screen's physical pixel height and width. As you are now rendering more pixels, you'll then need scale the canvas back down to the size it is rendering at, which will remove the blur.

Unfortunately, as many of the rendering calculations assumed that the canvas size would be the same as the size in CSS `px`, I had to fix some subtle bugs. One example was that extra tiles were being rendered off the edge of the canvas, resulting in performance losses. This was challenging to debug, as the extra tiles could not be seen — I ended up translating the whole canvas across, then scaling it down, allowing me to see where the extra tiles were being drawn (see Figure 4.3).

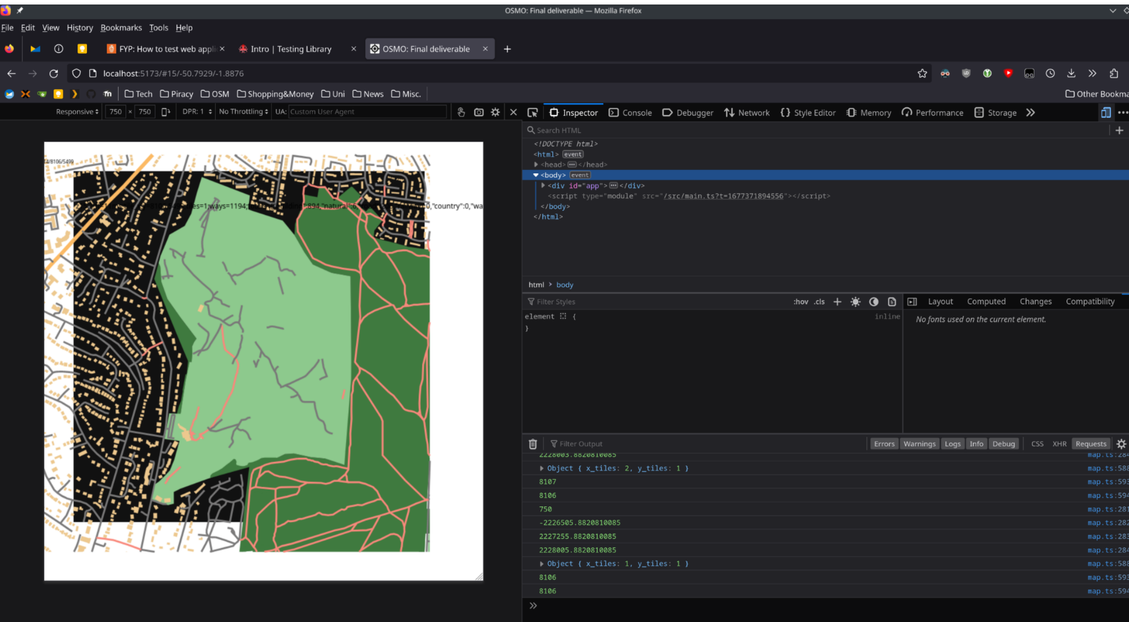

Figure 4.3: Scaling and translating the canvas to debug tile rendering — the black area represents the scaled down viewport

5: Software engineering

In order to successfully deliver a project, it is important to follow good software engineering practices. These should help you create something that has minimal technical debt, is correct, is maintainable, and can be easily understood by other members of a team.

I decided to write my project in TypeScript over plain JavaScript, as it provides an extremely helpful layer to help solve runtime type issues. It also allows my IDE, Visual Studio Code, to provide far more useful suggestions, compared to writing JS. This is because it is aware what types variables are at all times, instead of using a fallible heuristic approach.

Conveniently, the JavaScript bundler I chose, `esbuild`, has built in support for stripping TypeScript annotations, meaning I only have a single build step. It doesn't have the ability to check types itself, but this was not an issue for me, as VS Code performs this function natively in the editor. `esbuild` also has a convenient "dev-server" mode, where it bundles up the latest code as the requests come in, removing the need for a file-watcher.

5.1 Testing

Testing is critical to producing a piece of software that works as expected. It helps you to define what your function should return before you write it, and ensures that the function returns exactly what you expect.

In my project a significant amount of the work is rendering data to the HTML5 canvas. This portion of the project is very difficult to test — at least while development is still occurring. Therefore, I directed my testing efforts towards the non-visual portions of the application, such as parsing the Mapsforge file, geometry operations, and projecting data.

I implemented testing using the Jest framework. Initially I had some issues setting it up to be able to understand tests and code written in TypeScript. Once it was set up, it was irritatingly slow, taking 5+ seconds to run with only a single test. The cause of this was two-fold; I was using the `ts-jest` package, which uses Babel to transpile TypeScript to JS (which is slow), and I had written my `jest.config.ts` in TypeScript. This meant that when I ran `jest test`, it had to spin up `ts-node` to interpret the file, which is also relatively slow. To fix this I rewrote the config file in plain JavaScript, and switched Jest to use `@swc-node/jest` instead of `ts-jest` to transpile the JavaScript, which reduced the time taken to under 2 seconds for the whole suite.

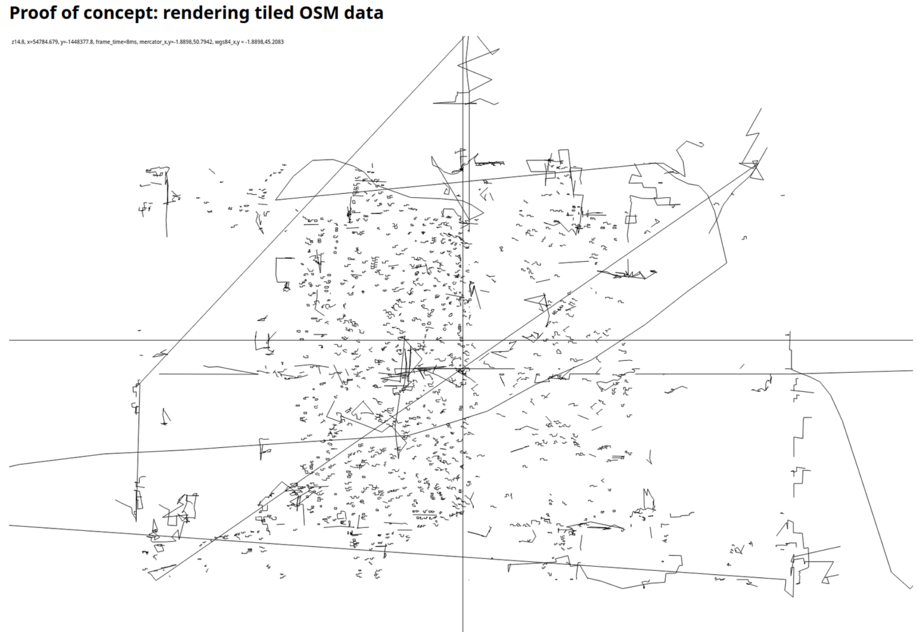

Figure 5.1: Rendering incorrectly parsed geometries

In particular, the tests for the `Reader` class proved very useful. At first, I didn't write any formal tests for the `.getVSint()` method — instead running it and checking if the output seemed reasonable. This got me so far, with it returning some well-formed data, but when it came to rendering the geometries were all corrupted, as is shown in Figure 5.1. During the debugging process, I wrote exhaustive table tests for this function (see Listing 5.1), and I realized that I was incrementing my offset through the file on the wrong line.

These tests made it much easier to refactor the code, without worrying about breaking functionality. This is because I could just run `pnpm test` to ensure I had not introduced any regressions, and did not need to manually check various different cases.

Listing 5.1: Output from the `.getVSint()` test suite

```
should be able to decode signed variable ints
    pass: 1 byte max negative: [01111111].getVSint() == -63
    pass: 1 byte max positve: [00111111].getVSint() == 63
    pass: 2 byte min: [10000000,00000001].getVSint() == 128 (1 ms)
    pass: 2 byte max: [11111111,01111111].getVSint() == -8191
    pass: 3 byte min: [10000000,10000000,01000001].getVSint() == -16384
    pass: 3 byte max: [11111111,11111111,01111111].getVSint() == -1048575
    pass: 4 byte min: [10000000,10000000,10000000,00000001].getVSint() == 2097152
```

5.1.1 Test suite coverage

Test coverage is the percentage of lines of code that are executed by the test suite. This can be a useful tool to determine which parts of the codebase your tests are covering, and which cases you had not considered. You can then use this information to write more tests to cover the untested parts of the program. However, chasing 100% coverage is usually not a productive use of time, as testing every single trivial case or condition is often necessary, and adds noise to the test suite.

Listing 5.2: Test suite coverage

```
---------------|---------|----------|---------|---------|-------------------
File           | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---------------|---------|----------|---------|---------|-------------------
All files      |   92.73 |    75.23 |   94.82 |    92.4 |
map            |     100 |      100 |     100 |     100 |
geom.ts        |     100 |      100 |     100 |     100 |
map/mapsforge  |   92.42 |    75.23 |   93.75 |   92.06 |
mapsforge.ts   |   91.25 |    63.38 |     100 |   90.93 | 117,137-166,192,[..snip..]
objects.ts     |    90.9 |      100 |      85 |   89.36 | 196-202,212
reader.ts      |     100 |      100 |     100 |     100 |
util.ts        |     100 |      100 |     100 |     100 |
---------------|---------|----------|---------|---------|-------------------
Tests:       38 passed, 38 total
Snapshots:   0 total
Time:        1.125 s
```

5.2 GitLab CI

In industry, it is common to run tests and deploy code automatically, as commits are pushed to the repository — this is known as CI/CD (continuous integration/continuous deployment). I set up GitLab CI to automatically run tests when new commits were pushed. The advantage of this is that if I accidentally push code that breaks the tests, I get an email to alert me (see Figure 5.2). As the CIM GitLab doesn't provide any CI runners, I had to set up my own "self-managed" runner. There are a number of options for how the job should be run (known as "executors"), such as in a Docker container, or over SSH. As my use case is fairly simple, without any complicated dependencies or required state, I decided to use the shell executor, which simply downloads the Git repository, and runs the shell code specified in the `.gitlab-ci.yml` file.

When working in a team it is good practice to have a centrally managed deployment system (CD), where team members can initiate or view the current state of a deployment of an application. For my project, as I am the sole developer, it is acceptable to have a

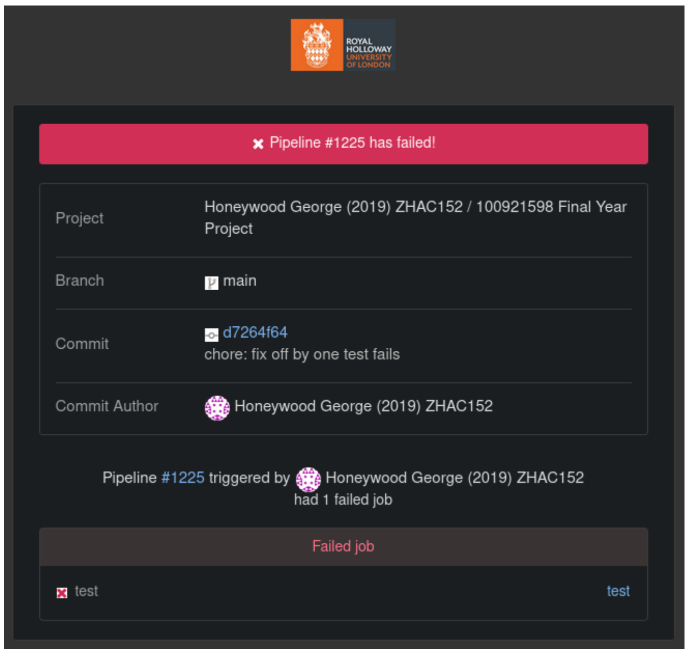

Figure 5.2: GitLab CI failure notification

simple deploy script that copies files to my web server using `rsync`.

5.3 Version control

Another important software engineering practice is making good use of version control. Throughout this project I have made use of Git branches, primarily to develop each of my proof of concepts. This has helped make sure that the code in the `main` branch is always in a working state.

I have rigorously followed the *Conventional Commits* specification, prefixing all commits with a type, such as `feat:`, `fix:`, `test:` or `report:` [59]. This indicates the basic purpose of a commit at a glance, and encourages me to split up large changes into smaller atomic units of a single type. When changes are complicated or have subtle side effects, I have written a longer commit body, to supplement the information in the commit name.

5.4 Documentation

Regarding user documentation, so far my proof of concepts have been simple enough that detailed documentation is not yet necessary. For each proof of concept I have written a `README.md` that details how it can be run (or a link to it hosted online), and any

dependencies or extra files required.

In terms of developer documentation, where appropriate I have written JSDoc comments, particularly for functions that provide a *public* API. These explicitly detail the purpose of the function, how it should be used, and what the arguments should be set to. In addition to these JSDoc comments, where the code itself is not completely clear or intentions aren't obvious, I have written comments inline.

5.5 Profiling and optimization

The `render()` function is critically important to the responsiveness of the application. As the `requestAnimationFrame()` callback fires 60 times a second, we should aim to have the `render()` function run in under $1000/60 = 16.666$ ms, so that we have a new frame to show with each refresh of the screen.

The first tool I implemented to work towards achieving this was adding a frame time metric to the debug information at the top of the canvas. This allowed me to test various different configurations and code changes to see which resulted in the best performance. Measuring the frame time is helpful for gauging performance overall, but it is not very precise — you cannot see exactly what is consuming the CPU time within the `render()` call.

Figure 5.3: Using the profiler

This is where the profiler comes in. It is a tool that lets you visualize how long different function calls within your code take to execute. Both Firefox and Chrome include one within their Developer Tools. I marginally prefer the one Firefox provides (for its Flame Graph, see Figure 5.3), but it is important to test in both browsers, as their performance

characteristics are surprisingly different. This led me to discover that the vast majority of the time spent in the `render()` function is in calls on the `CanvasRenderingContext2D`, to either `stroke()`, `fill()`, or draw a `lineTo()`. Therefore, the most important factor to optimize in this scenario is to reduce the amount of geometries that need to be rendered.

6: Professional issues

Accessibility is a particularly relevant concern for my project. People with disabilities still need to be able to be able to interpret geographic data, and may have different priorities to able-bodied users. Those with visual impairments need larger map label sizes, and wheelchair users may need extra context about sloped kerbs. My application should accommodate as many of these needs as possible.

Through assistive technologies like screen readers or magnification, most text-based web content is accessible to partially sighted or blind users. Android and iOS both have built-in screen-readers — TalkBack and VoiceOver respectively. These tools assist users by providing a full alternate means of interaction that is more complete and usable than simply reading out the text on screen. TalkBack disables the usual touch/swipe gestures on the phone, instead allowing the user to swipe left or right to navigate through tab stops in the interface. At each tab stop TalkBack announces the type and content of the selected screen element, and the user can double-tap to interact with it if necessary. The system navigation gestures can be accessed through two-finger swipes, instead of the usual single finger ones.

Talkback works for not only native apps, but also web based content. This is enabled through use of semantic HTML elements, supplemented by ARIA tagging where this is not possible [60]. The idea behind semantic HTML is that the HTML markup should convey meaning, instead of providing information purely visually through CSS. For example, UI buttons should be tagged as `<button>` elements, over plain `<div>`s, so that a screen reader can relay this context to the user. There are numerous HTML elements that have semantics, such as basics like `<ul>` and `<h1>`, and others that convey more complex concepts, like `<nav>` for navigation menus [61].

Even most non-text based media can be made accessible to blind people, through `alt` tags on images and subtitling on videos. Unfortunately, this is simply not possible for web maps, as they convey meaning in a visual form that cannot be textually described easily. MDN advises that, in general, web developers should use semantic HTML over the `<canvas>` element, as the canvas is opaque bitmapped data that cannot be interpreted by a screen reader [62]. Notwithstanding this, blind and visually impaired people still need to navigate the world. Grierson, Zelek and Carnahan [63] found that a “Tactile Way-finding Belt”, could help facilitate navigation for those with “visual impairment or Alzheimer’s disease”. They based their study on a system of four vibrating motors, with one for each cardinal direction, which provide signals directing the wearer to their destination. By design this format is more limited than a map, but it does work towards making one of their main uses, navigation, more accessible.

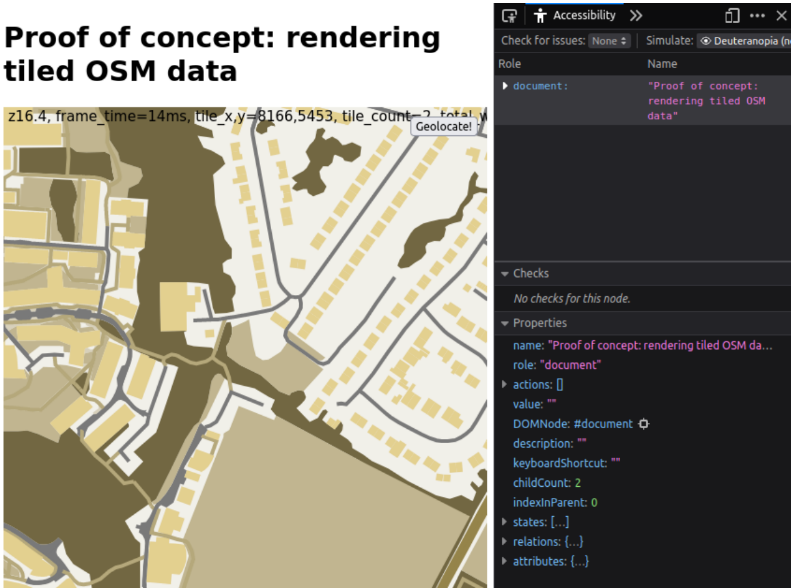

Figure 6.1: Simulating deuteranopia, in Firefox Developer Tools

A common form of visual impairment is colour vision deficiency (colour blindness). According to the NHS [64], the red-green form affects around 1 in 12 men and 1 in 200 women. This can be more easily accommodated than other forms of visual impairment by ensuring that text and backgrounds have a suitable contrast ratio, described in the Web Content Accessibility Guidelines [65]. It is also important to visually check colour contrast through simulators. The Developer Tools in Firefox provide one such simulator, which can visualize various forms of colour vision deficiency — see Figure 6.1.

Physical disabilities can also be accommodated in an OpenStreetMap-based map. In its simplest form, a point of interest can be tagged with `wheelchair=(yes|limited|no)`. This tag indicates to what degree an amenity can be accessed by a wheelchair user. `wheelchair=*` tagging is quite well-used, though not ubiquitous, with around 2.5 million uses worldwide according to Taginfo [66]. Routing software for wheelchair and walking frame users must avoid features like stairs (`highway=steps`) and full-height kerbs. Adding this tagging is well-supported in "on-the-go" map editing apps like StreetComplete, which has a "quest" dedicated to adding this adding kerb details — see Figure 6.2 [67].

There are specialized map viewers, specifically designed to make viewing wheelchair tagging simple — most popularly Wheelmap, which uses green, orange, or red to indicate wheelchair accessibility [68]. Wheelmap can also be used to add this tagging to amenities, through a simple form, that lowers the barrier of entry for contributing this information to OSM. `smoothness=*` tagging is also valuable to wheelchair users [69]. This key has textual values ranging from `excellent` to `very_horrible`, with wheelchair users unlikely to be able to traverse anything at the level of `smoothness=bad` or worse. Routers can use this information to inform the paths they choose.

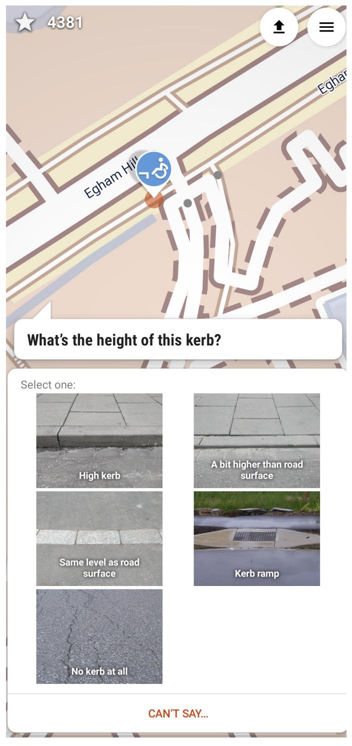

Figure 6.2: Adding kerb height data using the StreetComplete app

One accessibility consideration that I have made whilst developing this project is to include multiple alternative interaction gestures where possible. For example, when zooming the map, you can pick from either the two finger "pinch" zoom gesture, or alternatively a single finger tap and hold. This allows people with the ability to use only one hand, such as amputees or walking stick users, to make full use of the application.

7: Conclusion & evaluation

Overall the project has been a success. OSMO is capable of decoding and rendering Mapsforge files, and can sparsely download map regions for offline use. A novel aspect of the project is the usage of HTTP range requests to fetch only the required bytes from the Mapsforge file, meaning that no specialized map tile server is required. I am also happy with the UX on mobile, where the web app can be "installed" as a PWA, giving a native-like experience.

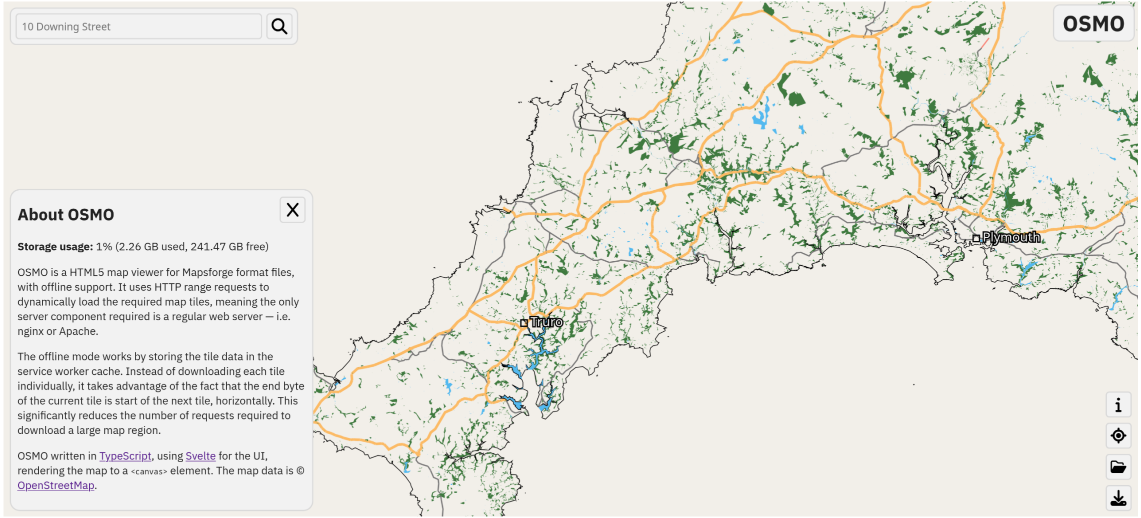

Figure 7.1: OSMO rendering Cornwall

In term one I successfully followed the timeline I set out in subsection 1.3.1. I mildly deviated from the plan in that I added an extra research report. This was on the basics of web technologies, to provide a basis of understanding for the rest of my project — as I had written my other reports assuming quite a high level of base knowledge. The proof of concepts were useful, helping me to understand key technologies and discover how is best to build the final deliverable.

In term two the plan was less useful, as the way that I had built the proof of concepts on one another meant that no integration work was necessary. I also allocated a generous amount of time for mobile design & testing, which wasn't required as I conducted this through the development of proof of concepts. In addition, I did not allow time specifically for working on the UI, when in reality I spent some time working adding a frontend framework (see section 4.5).

In term two I also spent some time refining the tile loading system to use HTTP range requests, which was not planned for. This had the knock-on effect of complicating offline support with the service worker, taking up extra time. Other than that, however, the

rest of the deliverables were non-blocking to the project, meaning I had some flexibility in what I worked on. I ended up not implementing deliverables 10 (online routing) and 11 (Wikipedia integration), instead prioritizing polishing the offline user experience.

In retrospect, I should have put some more thought into alternative rendering strategies — the renderer is definitely the weakest part of the project. Both mobile and desktop performance is broadly acceptable until you zoom out past zoom level 13, at which point the frame time becomes >100ms (<10fps). At zoom 7, where you can see the whole of the UK, the frame time is >500ms (<2fps). This is unacceptable, especially for an app targeted at mobile devices with limited CPU power & battery life.

At the start of the project I should have spent time researching alternative rendering strategies that would have allowed for a more responsive experience. One way I could have achieved this would have been to use WebGL, which is GPU accelerated — see section 4.4 for more detail. Or, instead of WebGL, I could have adopted a hybrid raster approach, in which the tiles are each rendered once for each zoom level, then reused, instead of re-rendering everything on every frame. The drawbacks of this approach would be additional complexity, and that you would only be able to view the map at discrete zoom levels.

Due to this ever-present need to keep the frame time down, I was apprehensive to add advanced rendering features, like way labels that follow road curvature — as they would make the process even slower. In addition, I would have liked to add a label collision prevention algorithm. While the Mapsforge format handles some of this issue, mostly by reducing the number of city/town labels, a more sophisticated solution would have been useful for areas with a high density of labels, such as town centres.

Finally, it would have been valuable to implement some form of theming system. Most map rendering libraries provide some sort of interface that can be used to customize how features render. This could be used to provide a dark theme, or customize the cartography of the map for a specific use.

I am pleased with the implementation of online/offline mode switching, where if a user loads any tile that has not already been cached, it will be stored in the service worker for later offline use. The user can also choose to download the tiles currently below the viewport, if they know that they will need access to that specific area when offline later.

8: Appendix

8.1 `diary.md`

```
# Final Year Project Diary

## Term 1

### Week 2 (2022-09-26)

Worked on the initial project plan. Read through "OpenStreetMap -- Using and Enhancing the
 Free Map of the World".

### Week 3 (2022-10-03)

Got feedback on initial plan from my supervisor (Reuben Rowe), and made alterations based
on it. Specifically, I shorted the abstract by moving content to an introduction section,
wrote a general "Aims & objectives" section, and added a "Deliverables" section,
referencing these deliverables in the timeline. I also polished the "Risks & mitigations"
section, and changed inline links to references.

### Week 4 (2022-10-10)

Worked on the first report, about offline HTML5 applications (deliverable 1).

### Week 5 (2022-10-17)

Worked on the offline HTML5 proof of concept (deliverable 2). Implemented a demo offline
app using service workers, with the ability to load a large file either from the local
disk using the File API, or from the service worker cache with the Cache API.

Completed report on OpenStreetMap data sources (deliverable 3). Completed report on map
projections (deliverable 4).

### Week 6 (2022-10-24)

Implemented the proof of concept that renders some array of wgs84 coordinates to a
Mercator projected map (deliverable 5, proof-of-concepts/2-rendering-a-way). Allows you to
 render the boundaries of three different towns -- with manually chosen zoom factor.

Started a report on the HTML5 canvas.

Started work on the panning and zooming proof of concept (deliverable 6, proof-of-concepts
/3-panning-and-zooming). Got WASD panning, and scrollwheel zoom working.

### Week 7 (2022-10-31)
```

Continued work on the panning and zooming proof of concept. Implemented click panning and touch controls: pinch zoom & single finger panning. Also made zooming work around the mouse/pinch gesture instead of the centre of the screen.

Made the map linkable, i.e. you can send a link to a position on the map. This is done by writing the current map centre position and zoom level to the end of the URL.

Add esbuild (JavaScript bundler). This means I can write JavaScript in multiple files, but have them merged together be served to the user. I can also use newer JavaScript features, and have them transpiled into older syntax.

Week 8 (2022-11-07)

Began work on the Mapsforge file parser. Added testing using Jest to help develop the Mapsforge parser. Finished decoding the file header, which contains metadata and other information needed to parse the rest of the file.

Week 9 (2022-11-14)

More Mapsforge parser work. Specifically decoding the tile data itself. First reading PoIs from the file, then Ways. Used the Mapsforge map file creator to generate map files with debug information, to make writing the parser easier. Created a `Reader` abstraction over the JavaScript `DataView` API, storing and updating the offset that the data is being read from, making the code much cleaner.

Got a single tile of data to appear in the canvas map, but there is some issue with the transformation of the coordinate data.

Week 10 (2022-11-21)

Worked on the interim report in preparation for supervisor meeting, then for the interim submission. Fixed a tricky bug that was a result of an incorrect interpretation of the signed integer values. This was discovered and fixed by writing some tests that I should have written in the first place.

Got multiple tiles to appear on the map, being loaded dynamically as you pan and zoom. Currently always loads the high resolution subfile, at z14, so zooming out quickly becomes slow. Fixing this is my next goal.

Week 11 (2022-11-28)

Worked on getting the correct base tiles to render as you zoom in/out, significantly improving performance, and letting you get a high level overview of the map.

Added styling for Ways and PoIs, with some zoom-conditional rendering logic. Continuing with the interim report.

Week 12 (2022-12-05)

Improved multipolygon handling, rendering each ring separately. Still not perfect, as it currently in-fills everything inside the outer way, when it should only fill up to the inner ways.

Worked on handling v5 files. These have variable tag values, so data other than the hardcoded tags supported by the format can be stored.

Added an online mode, where instead of fetching the whole map blob in one go, we fetch only the chunks we need, using HTTP range requests. This allows you to interactively use huge map files, like the whole of England (~800 MB), as it only fetches the necessary bits of the file for your current viewport.

Term 2

Week 18 (2023-01-16)

Added geolocation support, using `geolocation.watchPosition()`, which updates with your position. Also added double tap + hold gesture for zooming with a single finger on mobile.

Worked on the professional issues' section of the report, talking about accessibility.

Week 19 (2023-01-23)

Finished off the professional issues section. Talked about visual impairments and physical disabilities, and how they impact usage of an OSM-based map. Also discussed how mappers can help make OSM accessible by using certain tags, like `wheelchair=*` or `smoothness=*`. Debating writing a conclusion paragraph for it.

Worked on map fling --- where map continues to scroll after fast movement. Only for mouse controls at the moment.

Started work on search (geocoding), using the Nominatim API hosted by the OSMF. Improved styling, by making the canvas fullscreen, and overlaying the title atop it.

Week 20 (2023-01-30)

Added the ability to bind the search to the current map viewport. Made search results show up beneath the search bar, where you can click them to pan the map to the location.

UI improvements. Made search bar responsive, and always visible. Replaced text with icons where possible.

Week 21 (2023-02-06)

Big refactor to use Svelte front-end framework. This made all the UI code much cleaner, with no more passing DOM references through the JavaScript code. Now there is a big `Map.svelte` component, which holds all the Touch/Mouse/Keyboard event handlers. Also allowed me to scope the CSS to each component, instead of having a single massive chunk of CSS in `index.html`.

Using a proper UI framework made it much easier to make the search form reactive. Worked on making it resettable. Also added a nice dropdown animation for when the results pop in.

Week 22 (2023-02-13)

Improved offline handling, swapping the search bar for an offline icon when there is no internet access. New fancy favicon.

Added support for opening a local map file, on the user's device. Set up Gitlab CI, with self-hosted runner. Spent a while debugging why tests were taking 3 minutes, when locally they take 2 seconds. Turned out to be [this bug](https://gist.github.com/charlyie/76ff7d288165c7d42e5ef7d304245916). Also upgraded dependencies.

Week 23 (2023-02-20)

Added support for Hi-DPI screens. Previously the canvas size was always just the window.innerWidth/Height value, which is in CSS pixels -- which aren't necessarily correlated to real screen pixels. Now the canvas renders at a higher resolution to look crisp on screens with a higher DPI.

Week 24 (2023-02-27)

Grammar and spelling fixes in report. Fixed bug where labels were duplicated on tile borders. Tested using map inside an iframe -- this is a common map usecase, i.e. embedded on a business website.

Added support for exact range requests in the service worker cache -- this means we can do offline mode with the dynamic range requests. Added a download/precache button that lets you download a whole area in advance.

Week 25 (2023-03-06)

Added a PWA webmanifest, to make the website "installable" on mobile phones. Did some refactoring on the touch event handling code, to remove some necessary helpers. Made it so loading/not found tiles are rendered crossed out.

Improved partial service worker caching, so that it can retrieve smaller chunks of larger files. This makes downloading and storing many map tiles much more efficient, as (x,y) (x+1,y) tiles are contiguous bytes in the mapsforge file. Adapted prefetch code to fetch these contiguous ranges. Added loading spinner, and an "about" page.

Week 26 (2023-03-13)

Fixed bug with inconsistent behaviour with dynamic mode and blob mode -- turned out to be due to HTTP range requests including the last requested byte, i.e. `Range: bytes=0-10` will return 11 bytes, whereas `slice(0, 10)` will return 10 bytes. Fixed another bug in the handling of files with debug info.

Made it so that the service worker takes control on the first page load. Before it was only "controlling" the page after a refresh. This meant that tiles were not cached until a

```
 reload occurred, which is unexpected behaviour. Also set up proper resource precaching,
to ensure we get all files we need.

### Week 27 (2023-03-20)

Fixed "installing" PWA on Android -- before it would add a shortcut to the homescreen, but
 not add the icon to the app drawer. This was due to providing an SVG icon in the
webmanifest, when a PNG was required.
```

Bibliography

[1] Frederik Ramm, Jochen Topf and Steve Childon. *OpenStreetMap — Using and Enhancing the Free Map of the World*. Note: A good summary of the basics of the OpenStreetMap project. Some content is a little out of date, especially the sections on editors and tools for mappers. Notably the online editor referred to here, Potlatch, is no longer available, having been superseded by iD in 2013. UIT Cambridge Ltd., 2011 (cit. on pp. 4, 6).

[2] Pascal Neis. *OSMStats — Statistics of the free wiki world map*. Note: Useful set of statistics about the OpenStreetMap project. This includes the number of registered users, the number of edits, and the number of objects in the database. URL: https://osmstats.neis-one.org/?item=members (visited on 29/09/2022) (cit. on p. 4).

[3] Ilya Zverev. *Every Door*. Note: A new mobile OSM editor, released in 2022, focussed on adding points of interest and other data best gathered on foot. In July 2022, I added the ability to view the history of an element to the app via a pull request. URL: https://every-door.app/ (cit. on p. 4).

[4] OsmAnd BV. *OsmAnd*. Note: Offline OSM map viewer for Android and iOS. The app is open source, and provides a swiss army knife of features. URL: https://osmand.net/ (cit. on p. 4).

[5] Maps.me (Cyprus) Limited. *Maps.me*. Note: Most popular OSM-based map app on the Android platform. Was historically open source, but was acquired by another company and made closed source. URL: https://maps.me/ (cit. on p. 4).

[6] Magic Earth. *Magic Earth*. Note: Proprietary OSM-based mobile map viewer. URL: https://www.magicearth.com/ (cit. on p. 5).

[7] KDE Marble Contributors. *KDE Marble*. Note: Open source map viewer for desktop, with cross-platform support. Has basic support for downloading OSM maps. URL: https://marble.kde.org/ (cit. on p. 5).

[8] OpenStreetMap Foundation. *Tile Usage Policy: Bulk Downloading*. URL: https://operations.osmfoundation.org/policies/tiles/%5C#bulk-downloading (visited on 29/09/2022) (cit. on pp. 5, 14).

[9] OpenStreetMap Wiki Contributors. *OpenStreetMap API*. URL: https://wiki.openstreetmap.org/wiki/API (visited on 29/09/2022) (cit. on pp. 5, 18).

[10] OpenStreetMap Wiki Contributors. *API: Terms of use*. URL: https://wiki.openstreetmap.org/wiki/API%5C#Terms_of_use (visited on 29/09/2022) (cit. on p. 5).

[11] devemux86. *Cruiser*. Note: A cross-platform offline first OSM map viewer. Is based on the Mapsforge library, but is not open source. URL: https://wiki.openstreetmap.org/wiki/Cruiser (cit. on p. 5).

[12] Mapsforge. *Mapsforge Binary Map File Format Specification*. Note: The specification for the Mapsforge binary map format. This format is quite popular, and is used by a number of applications. 2017. URL: https://github.com/mapsforge/mapsforge/blob/master/docs/Specification-Binary-Map-File.md (visited on 29/09/2022) (cit. on pp. 5, 10, 15, 22).

[13] OpenStreetMap Wiki Contributors. *OpenStreetMap Wiki: Main Page*. URL: https://wiki.openstreetmap.org/wiki/Main_Page (visited on 30/11/2022) (cit. on p. 6).

[14] OpenStreetMap Wiki Contributors. *Mercator*. URL: https://wiki.openstreetmap.org/wiki/Mercator (visited on 25/11/2022) (cit. on pp. 6, 18).

[15] OpenStreetMap Wiki Contributors. *Slippy map tilenames*. URL: https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames (visited on 20/10/2022) (cit. on pp. 6, 14).

[16] MDN Contributors. *MDN Web Docs*. URL: https://developer.mozilla.org/en-US/ (visited on 30/11/2022) (cit. on p. 6).

[17] MDN Contributors. *Canvas API*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API (visited on 02/11/2022) (cit. on pp. 6, 9, 12).

[18] MDN Contributors. *Service Worker API*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API (visited on 30/11/2022) (cit. on pp. 6, 12).

[19] MDN Contributors. *HTML: HyperText Markup Language*. URL: https://developer.mozilla.org/en-US/docs/Web/HTML (visited on 30/11/2022) (cit. on p. 9).

[20] MDN Contributors. *JavaScript modules*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules (visited on 30/11/2022) (cit. on p. 10).

[21] SitePoint Pty. Ltd. *Using ES Modules in the Browser Today*. URL: https://www.sitepoint.com/using-es-modules/ (visited on 30/11/2022) (cit. on p. 10).

[22] Evan Wallace. *esbuild*. URL: https://esbuild.github.io/ (visited on 30/11/2022) (cit. on p. 10).

[23] Joshua Bell. *File and Directory Entries API*. W3C Draft Community Group Report. W3C, Sept. 2022. URL: https://wicg.github.io/entries-api/ (cit. on pp. 10, 11).

[24] MDN Contributors. *Introduction to the File and Directory Entries API*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/File_and_Directory_Entries_API/Introduction (visited on 12/10/2022) (cit. on p. 10).

[25] WHATWG. *HTML Living Standard: 7.11.2 Application caches*. Tech. rep. Note: link to a snapshot, as this has since been removed from the standard. WHATWG, 2020. URL: https://html.spec.whatwg.org/commit-snapshots/27ca698a224a4fcf59b647be80a0c86c3c6abba5/#appcache (visited on 17/10/2022) (cit. on p. 10).

[26] MDN Contributors. *Window.applicationCache*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/applicationCache (visited on 17/10/2022) (cit. on p. 10).

[27] Mapbox. *MBTiles Specification*. 2018. URL: https://github.com/mapbox/mbtiles-spec (visited on 12/10/2022) (cit. on p. 10).

[28] Marijn Kruisselbrink. *File API*. W3C Editor's Draft. W3C, Oct. 2022. URL: https://w3c.github.io/FileAPI/ (cit. on pp. 11, 17).

[29] MDN Contributors. *File and Directory Entries API support in Firefox: Limitations*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/File_and_Directory_Entries_API/Firefox_support#limitations_in_firefox (visited on 17/10/2022) (cit. on p. 11).

[30] Google Developers. *Managing HTML5 Offline Storage*. 2018. URL: https://developer.chrome.com/docs/apps/offline_storage/ (visited on 17/10/2022) (cit. on p. 12).

[31] Jake Archibald and Marijn Kruisselbrink. *Service Workers Nightly: Caches*. W3C Editor's Draft. W3C, June 2022. URL: https://w3c.github.io/ServiceWorker/#cache-objects (cit. on pp. 12, 17, 27).

[32] Steve Fulton and Jeff Fulton. *HTML5 Canvas: Native Interactivity and Animation for the Web*. O'Reilly Media, 2013. ISBN: 9781449335885. URL: https://books.google.co.uk/books?id=zLUyKvtdCQwC (cit. on p. 12).

[33] MDN Contributors. *WebGL2RenderingContext*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebGL2RenderingContext (visited on 02/11/2022) (cit. on p. 12).

[34] Mapbox. *Mapbox GL JS*. 2022. URL: https://docs.mapbox.com/mapbox-gl-js/guides/ (visited on 02/11/2022) (cit. on p. 12).

[35] MDN Contributors. *Canvas API: Drawing text*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Drawing_text (visited on 02/11/2022) (cit. on p. 13).

[36] MDN Contributors. *Canvas API: Transformations: Rotating*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Transformations#rotating (visited on 21/11/2022) (cit. on p. 13).

[37] MDN Contributors. *Canvas API: Path2D*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/Path2D (visited on 21/11/2022) (cit. on p. 13).

[38] OpenStreetMap Wiki Contributors. *Downloading data: Choose your region*. URL: https://wiki.openstreetmap.org/wiki/Downloading_data#Choose_your_region (visited on 20/10/2022) (cit. on p. 14).

[39] OpenStreetMap Wiki Contributors. *Using OpenStreetMap offline: Frameworks*. URL: https://wiki.openstreetmap.org/wiki/Using_OpenStreetMap_offline#Frameworks (visited on 20/10/2022) (cit. on p. 15).

[40] Mapsforge. *Mapsforge Project Homepage*. URL: https://github.com/mapsforge/mapsforge (visited on 20/10/2022) (cit. on p. 15).

[41] Mapsforge. *Applications using Mapsforge software*. URL: https://github.com/mapsforge/mapsforge/blob/master/docs/Mapsforge-Applications.md (visited on 20/10/2022) (cit. on p. 15).

[42] Mapbox. *MBTiles*. URL: https://docs.mapbox.com/help/glossary/mbtiles/ (visited on 20/10/2022) (cit. on p. 15).

[43] Frank Canters. *Small-Scale Map Projection Design*. In Focus–Routledge Film Readers. CRC Press, 2002. ISBN: 9780203472095. URL: https://books.google.co.uk/books?id=RZDUDwAAQBAJ (cit. on p. 15).

[44] OpenStreetMap Wiki Contributors. *Converting to WGS84*. URL: https://wiki.openstreetmap.org/wiki/Converting_to_WGS84 (visited on 23/10/2022) (cit. on p. 15).

[45] EPSG. *EPSG:4326 — WGS84: World Geodetic System 1984, used in GPS*. URL: https://epsg.io/4326 (visited on 23/10/2022) (cit. on p. 15).

[46] Sarah E. Battersby et al. “Implications of Web Mercator and Its Use in Online Mapping”. In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 49.2 (2014), pp. 85–101. DOI: 10.3138/carto.49.2.2313 (cit. on p. 16).

[47] Utah Geospatial Resource Center. *The Earth is Not Round! Utah, NAD83 and WebMercator Projections*. URL: https://gis.utah.gov/nad83-and-webmercator-projections/ (visited on 23/10/2022) (cit. on p. 16).

[48] John P. Snyder. *Map Projections: A Working Manual*. Geological Survey Bulletin Series. U.S. Government Printing Office, 1987. ISBN: 9780160033605. URL: https://books.google.co.uk/books?id=nPdOAAAAMAAJ (cit. on pp. 16, 18).

[49] Jakob Nielsen. “Enhancing the explanatory power of usability heuristics”. In: *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. 1994, pp. 152–158 (cit. on p. 17).

[50] Mapbox. *geojson.io*. URL: https://geojson.io/ (visited on 25/11/2022) (cit. on pp. 18, 21).

[51] Natural Earth. *1:110m Cultural Vectors: Admin 0 — Countries*. URL: https://www.naturalearthdata.com/downloads/110m-cultural-vectors/ (visited on 26/11/2022) (cit. on p. 20).

[52] MDN Contributors. *Window.requestAnimationFrame()*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame (visited on 26/11/2022) (cit. on p. 21).

[53] OpenStreetMap Wiki Contributors. *Relation:multipolygon*. URL: https://wiki.openstreetmap.org/wiki/Relation:multipolygon (visited on 27/11/2022) (cit. on p. 22).

[54] MDN Contributors. *HTTP range requests*. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Range_requests (visited on 15/03/2023) (cit. on p. 26).

[55] Brandon Liu. *Protomaps*. 2023. URL: https://protomaps.com/ (visited on 15/03/2023) (cit. on p. 27).

[56] Gregg Tavares. *WebGL2 Fundamentals*. 2023. URL: https://webgl2fundamentals.org/ (visited on 17/03/2023) (cit. on p. 29).

[57] MDN Contributors. *Progressive web apps (PWAs)*. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps (visited on 22/03/2023) (cit. on p. 31).

[58] Christian Spanring and Alexander Mayrhofer. *A Uniform Resource Identifier for Geographic Locations ('geo' URI)*. RFC 5870. RFC Editor, June 2010. DOI: 10.17487/RFC5870. URL: https://www.rfc-editor.org/info/rfc5870 (cit. on p. 32).

[59] Conventional Commits. *Conventional Commits: Specification*. URL: https://www.conventionalcommits.org/en/v1.0.0/#specification (visited on 30/11/2022) (cit. on p. 38).

[60] James Nurthen, Michael Cooper and Peter Krautzberger. *Accessible Rich Internet Applications (WAI-ARIA) 1.3*. W3C Editor's Draft. W3C, Jan. 2023. URL: https://w3c.github.io/aria/ (cit. on p. 41).

[61] MDN Contributors. *HTML: A good basis for accessibility*. 2022. URL: https://developer.mozilla.org/en-US/docs/Learn/Accessibility/HTML (visited on 23/01/2023) (cit. on p. 41).

[62] MDN Contributors. *Element Reference: <canvas>: The Graphics Canvas element*. URL: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/canvas#accessibility_concerns (visited on 19/01/2023) (cit. on p. 41).

[63] Lawrence Grierson, John Zelek and Heather Carnahan. "The Application of a Tactile Wayfinding Belt to Facilitate Navigation in Older Persons". In: *Ageing International* 34 (Dec. 2009), pp. 203–215. DOI: 10.1007/s12126-009-9039-2 (cit. on p. 41).

[64] NHS. *Colour vision deficiency (colour blindness)*. 2019. URL: https://www.nhs.uk/conditions/colour-vision-deficiency/ (visited on 23/01/2023) (cit. on p. 42).

[65] Andrew Kirkpatrick et al. *Web Content Accessibility Guidelines (WCAG) 2.1*. W3C Recommendation. W3C, June 2018. URL: https://www.w3.org/TR/WCAG21/#contrast-enhanced (cit. on p. 42).

[66] Jochen Topf and Christian Topf. *Wheelchair | Keys | OpenStreetMap Taginfo*. 2023. URL: https://taginfo.openstreetmap.org/keys/wheelchair (visited on 24/01/2023) (cit. on p. 42).

[67] Tobias Zwick and Contributors. *StreetComplete*. 2023. URL: https://github.com/streetcomplete/StreetComplete (visited on 24/01/2023) (cit. on p. 42).

[68] Sozialhelden e.V. *Wheelmap*. 2023. URL: https://wheelmap.org/ (visited on 24/01/2023) (cit. on p. 42).

[69] OpenStreetMap Wiki Contributors. *Key:smoothness*. URL: https://wiki.openstreetmap.org/wiki/Key:smoothness (visited on 25/01/2023) (cit. on p. 42).