Performance comparison of simplification algorithms for polygons in the context of web applications

Masterarbeit

Institut für Informatik

Universität Augsburg

vorgelegt von

Alfred Melch

Augsburg, August 2019

1. Gutachter:	Prof. Dr. Jörg Hähner
2. Gutachter:	Prof. Dr. Sabine Timpf
Betreuer:	Prof. Dr. Jörg Hähner

Abstract

In this thesis the performance of polyline simplification in web browsers is evaluated. Based on the JavaScript library Simplify.js a WebAssembly solution is built to increase performance. The solutions implement the Douglas-Peucker polyline simplification algorithm with optional radial distance preprocessing. The format for polylines that Simplify.js expects differs from the representation used in major geodata formats. This discrepancy is obvious in another JavaScript library, Turf.js, where it is overcome by format transformations on each call. A slight variant of Simplify.js is proposed in this thesis that can operate directly on the format used in GeoJSON and TopoJSON. The three approaches, Simplify.js, Simplify.js variant and Simplify.wasm are compared across different browsers by creating a web page, that gathers various benchmarking metrics. It is concluded that WebAssembly performance alone supersedes JavaScript performance. A drop-in replacement that includes memory management however bears overhead that can outweigh the performance gain. To fully utilize WebAssembly performance more effort regarding memory management is brought to web development. It is shown that the method used by Turf.js is unfavorable in most cases. Merely one browser shows a performance gain under special circumstances. In the other browsers the use of the Simplify.js variant is preferable.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Simplification of polygonal data structures is the task of reducing data points while preserving topological characteristics. The simplification often takes the form of removing points that make up the geometry. There are several solutions that tackle the problem in different ways. With the rising trend of moving desktop applications to the web platform geographic information systems have experienced the shift towards web browsers too (Alesheikh, Helali, and Behroz 2002). Performance is critical in these applications. Since simplification is an important factor to performance the solutions will be tested by constructing a web application using a technology called WebAssembly.

1.1 Binary instruction sets on the web platform

The recent development of WebAssembly allows code written in various programming languages to be run natively in web browsers. So far JavaScript was the only native programming language on the web (Reiser and Bläser 2017). The goals of WebAssembly are to define a binary instruction format as a compilation target to execute code at native speed and taking advantage of common hardware capabilities (Haas et al. 2017). The integration into the web platform brings portability to a wide range of platforms like mobile and internet of things. The usage of this technology promises performance gains that will be tested by this thesis. The results can give conclusions to whether WebAssembly is worth a consideration for web applications with geographic computational aspects. WebGIS is only one technology that would benefit greatly of such an advancement. Thus far WebAssembly has been shipped to the stable version of the four most used browser engines (Wagner 2017). The mainly targeted high-level languages for compilation are C and C++. Also a compiler for Rust and a TypeScript subset has been developed.

1.2 Performance as important factor for web applications

There has been a rapid growth of complex applications running in web-browsers. These so called progressive web apps combine the fast reachability of web pages with the feature richness of locally installed applications. Even though these applications can grow quite complex, the requirement for fast page loads and instant user interaction still remains. One way to cope with this need is the use of compression algorithms to reduce the amount of data transmitted and processed. In a way simplification is a form of data compression. Web servers use lossless compression algorithms like gzip to deflate data before transmission. Browsers that implement these algorithms can then fully restore the requested ressources resulting in lower bandwidth usage. The algorithms presented here however

remove information from the data in a way that cannot be restored. This is called lossy compression. The most common usage for this on the web is the compression of image data.

1.3 Topology simplification for rendering performance

While compression is often used to minimize bandwidth usage, the compression of geospatial data can particulary influence rendering performance. The bottleneck for rendering often is the transformation to scalable vector graphics used to display topology on the web. Implementing simplification algorithms for use on the web platform can lead to smoother user experience when working with large geodata sets.

1.4 Related work

There have been previous attempts to speed up applications with WebAssembly. They all have seen great performance benefits when using this technology. Results show that over several source languages the performance is predictably consistent across browsers (Surma 2019). Reiser and Bläser even propose to cross-compile JavaScript to WebAssembly. Through their developed library Speedy.js one can compile TypeScript, a JavaScript superset, to WebAssembly. The performance gains of critical functions reaches up to a factor of four (Reiser and Bläser 2017).

Shi and Cheung analyzed several different polyline simplification algorithms in 2006 regarding their performance and quality (Shi and Cheung 2006). In this thesis the algorithms will also be introduced. The performance benchmarking however will be limited to only the most effective algorithm that is used on the web.

1.5 Structure of this thesis

This thesis is structured into a theoretical and a practical component. First the theoretical principles will be reviewed. A number of algorithms will be introduced in this section. Each algorithm will be dissected by complexity and characteristics. Topology of polygonal data will be explained as how to describe geodata on the web. An introduction to WebAssembly will follow.

In chapter 3 the practical implementation will be presented. A web application will be developed to measure the performance of three related algorithms used for polyline simplification.

The results of the above methods will be shown in chapter 4. After discussion of the results a conclusion will finish the thesis.

2 Theory

In this chapter the theory behind polygon simplification will be explained. The simplification process is part of generalization in cartography. It will be clarified which goals drive the reduction of data quantity, especially in the context of web applications. Then several different simplification algorithms will be introduced. The leading data formats that represent geodata on the web will be explained. From there a closer look can be taken how the algorithms run on the web platform. For that the technology WebAssembly will be presented.

2.1 Generalization in cartography

In map generalization one aims to reduce the data presented appropriate to the scale and/or purpose of the map (Brophy 1973). This selection has been a manual process for a long time, such that geographic generalization has been developed into an art that can only be learned by years of apprenticeship and practice (Brassel 1990). When using automation one could be concerned about a lower quality of maps. This is why many talk about computer assisted generalization where only subprocesses can be fully automated.

Polyline simplification is the most basic topic in map generalization (Ai et al. 2017). The problems of geographic cartography also apply here. So a number of algorithms have been developed to describe the work of cartographers as abstract, computer automatable processes. A selection of these algorithms will be explained in chapter 2.2.

Cartography does not halt before digitalization. In the era of big data there is a large volume of map data available. Many come from collaborative projects like OpenStreetMap[1] (OSM) where volunteers submit freely available geographic information. To deliver this mass of data over the internet one can make use of the simplification processes described in this thesis. This is particularly useful as the information provided has usually no scale description. Automated simplification can bring appropriate data sizes while maintaining data usability (Ai et al. 2017).

2.2 Polyline simplification

In this chapter several algorithms for polyline simplification will be explained. For each algorithm a short summary of the routine will be given. At the end a comparison will be drawn to determine the method in use for benchmarking.

[1] `https://www.openstreetmap.org/`

n-th point algorithm This algorithm is fairly simplistic. It was described in 1966 by Tobler. The routine is to select every n-th coordinate of the polyline to retain. The larger the value of n is, the greater the simplification will be. (Clayton 1985)

The Random-point routine is derived from the n-th point algorithm. It sections the line into parts containing n consecutive positions. From each section a random point is chosen to construct the simplified line. (Shi and Cheung 2006)

Radial distance algorithm Another simple algorithm to reduce points clustered too closely together. The algorithm will sequentially go through the line and eliminate all points whose distance to the current key is shorter than a given tolerance limit. As soon as a point with greater distance is found, it becomes the new key. (Koning 2011)

Perpendicular distance algorithm Again a tolerance limit is given. The measure to check against is the perpendicular distance of a point to the line connecting its two neighbors. All points that exceed this limit are retained. (Koning 2011)

Reumann-Witkam simplification As the name implies this algorithm was developed by Reumann and Witkam. In 1974 they described the routine that constructs a "corridor/search area" by placing two parallel lines in the direction of its initial tangent. The width of this corridor is user specified. Then the successive points will be checked until a point outside of this area is found. Its predecessor becomes a key and the two points mark the new tangent for the search area. This procedure is repeated until the last point is reached. (Reumann and Witkam 1974)

Zhao-Saalfeld simplification This routine, also called the sleeve-fitting polyline simplification, developed in 1997 is similar to the Reumann-Witkam algorithm. Its goal is to fit as many consecutive points in the search area. The corridor is however not aligned to the initial tangent but rather to the last point in the sequence. From the starting point on, successors get added as long as all in-between points fit in the sleeve. If the constraint fails a new sleeve will be started from the last point in the previous section. (Zhao and Saalfeld 1997)

The Opheim simplification Opheim extends the Reumann-Witkam algorithm in 1982 by constraining the search area. To do that two parameters dmin and dmax are given. From the key point on the last point inside a radial distance search region defined by dmin is taken to form the direction of the search corridor. If there is no point inside this region

the subsequent point is taken. Then the process from the Reumann-Witkam algorithm is applied with the corridor constrained to a maximum distance of dmax. (Opheim 1982)

Lang simplification Lang described this algorithm in 1969. The search area is defined by a specified number of points to look ahead of the key point. A line is constructed from the key point to the last point in the search area. If the perpendicular distance of all intermediate points to this line is below a tolerance limit, they will be removed and the last point is the new key. Otherwise the search area is shrunk by excluding this last point until the requirement is met or there are no more intermediate points. All the algorithms before operated on the line sequentially and have a linear time complexity. This one also operates sequentially, but one of the critics about the Lang algorithm is that it requires too much computer time (Douglas and Peucker 1973). The complexity of this algorithm is $\mathcal{O}(m^n)$ with m being the number of positions to look ahead. (Lang 1969)

Douglas-Peucker simplification David H. Douglas and Thomas K. Peucker developed this algorithm in 1973 as an improvement to the by then predominant Lang algorithm. It is the first global routine described here. A global routine considers the entire line for the simplification process and comes closest to imitating manual simplification techniques (Clayton 1985). The algorithm starts with constructing a line between the first point (anchor) and last point (floating point) of the feature. The perpendicular distance of all points in between those two is calculated. The intermediate point furthest away from the line will become the new floating point on the condition that its perpendicular distance is greater than the specified tolerance. Otherwise the line segment is deemed suitable to represent the whole line. In this case the floating point is considered the new anchor and the last point will serve as floating point again (DP). The worst case complexity of this algorithm is $\mathcal{O}(nm)$ with $\mathcal{O}(n \log m)$ being the average complexity (Koning 2011). The m here is the number of points in the resulting line which is not known beforehand. (Douglas and Peucker 1973)

Visvalingam-Whyatt simplification This is another global point routine. It was developed in 1993. Visvalingam and Wyatt use a area-based method to rank the points by their significance. To do that the "effective area" of each point has to be calculated. This is the area the point spans up with its adjoining points (Shi and Cheung 2006). Then the points with the least effective area get iteratively eliminated, and its neighbors effective area recalculated, until there are only two points left. At each elimination the point gets stored in a list alongside with its associated area. This is the effective area of that point or the associated area of the previous point in case the latter one is higher. This

way the algorithm can be used for scale dependent and scale-independent generalizations. (Visvalingam and Whyatt 1993)

2.2.1 Summary

The algorithms shown here are the most common used simplification algorithms in cartography and geographic information systems. The usage of one algorithm stands out however. It is the Douglas-Peucker algorithm. In "Performance Evaluation of Line Simplification Algorithms for Vector Generalization" Shi and Cheung conclude that "the Douglas-Peucker algorithm was the most effective to preserve the shape of the line and the most accurate in terms of position" (Shi and Cheung 2006). Its complexity however is not ideal for web-based applications. The solution is to preprocess the line with the linear-time radial distance algorithm to reduce point clusters. This solution will be further discussed in section 3.1.

2.3 Geodata formats on the web

Here the data formats that are used through this theses will be explained.

The JavaScript Object Notation (JSON) Data Interchange Format was derived from the ECMAScript Programming Language Standard (Bray 2014). It is a text format for the serialization of structured data. As a text format it is well suited for the data exchange between server and client. Also it can easily be consumed by JavaScript. These characteristics are ideal for web based applications. It does however only support a limited number of data types. Four primitive ones (string, number, boolean and null) and two structured ones (objects and array). Objects are an unordered collection of name-value pairs, while arrays are simply ordered lists of values. JSON was meant as a replacement for XML as it provides a more human readable format. Through nesting, complex data structures can be created.

The GeoJSON Format is a geospatial data interchange format (Butler et al. 2016). As the name suggests it is based on JSON and deals with data representing geographic features. There are several geometry types defined to be compatible with the types in the OpenGIS Simple Features Implementation Specification for SQL (Open GIS Consortium et al. 1999). These are Point, MultiPoint, LineString, MultiLineString, Polygon, Multipolygon and the heterogeneous GeometryCollection. Listing 1 shows a simple example of a GeoJSON object with one point feature. A more complete example can be viewed in the file `data/example-7946.geojson`.

```
{
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [125.6, 10.1]
  },
  "properties": {
    "name": "Dinagat Islands"
  }
}
```

Listing 1: An example for a GeoJSON object

The feature types differ in the format of their coordinates property. A position is an array of at least two elements representing longitude and latitude. An optional third element can be added to specify altitude. While the coordinates member of a Point-feature is simply a single position, a LineString-feature describes its geometry through an Array of at least two positions. More interesting is the specification for Polygons. It introduces the concept of the linear ring as a closed LineString with at least four positions where the first and last positions are equivalent. The Polygon's coordinates member is an array of linear rings with the first one representing the exterior ring and all others interior rings, also named surface and holes respectively. At last the coordinates member of MultiLineStrings and MultiPolygons is defined as a single array of its singular feature type.

GeoJSON is mainly used for web-based mapping. Since it is based on JSON it inherits its strengths. There is for one the enhanced readability through reduced markup overhead compared to XML-based data types like GML. Interoperability with web applications comes for free since the parsing of JSON-objects is integrated in JavaScript. Unlike the Esri Shapefile[2] format a single file is sufficient to store and transmit all relevant data, including feature properties.

To its downsides count that a text based format cannot store the geometries as efficiently as it would be possible with a binary format. Also only vector-based data types can be represented. Another disadvantage can be the strictly non-topologic approach. Every feature is completely described by one entry. However, when there are features that share common components, like boundaries in neighboring polygons, these data points will be encoded twice in the GeoJSON object. This further poses concerns about data size. Also it is more difficult to execute topological analysis on the data set. Luckily there is a related data structure to tackle this problem.

[2] https://doc.arcgis.com/en/arcgis-online/reference/shapefiles.htm

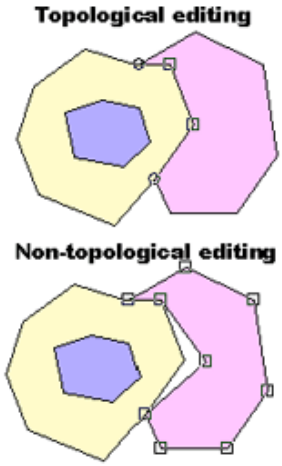

Figure 1: Topological editing (top) vs. Non-topological editing (bottom) (Theobald 2001)

TopoJSON is an extension of GeoJSON and aims to encode datastructures into a shared topology (Bostock 2017). It supports the same geometry types as GeoJSON. It differs in some additional properties to use and new object types like "Topology" and "GeometryCollection". Its main feature is that LineStrings, Polygons and their multiplicitary equivalents must define line segments in a common property called "arcs". The geometries themselves then reference the arcs from which they are made up. This reduces redundancy of data points. Another feature is the quantization of positions. To use it, one can define a "transform" object which specifies a scale and translate point to encode all coordinates. Together with delta-encoding of position arrays one obtains integer values better suited for efficient serialization and reduced file size.

Other than the reduced data duplication topological formats have the benefit of topological analysis and editing. When modifying adjacent Polygons for example by simplification one would prefer TopoJSON over GeoJSON. Figure 1 shows what this means. When modifying the boundary of one polygon, one can create gaps or overlaps in non-topological representations. With a topological data structure however the topology will be preserved. (Theobald 2001)

Coordinate representation Both GeoJSON and TopoJSON represent positions as an array of numbers. The elements depict longitude, latitude and optionally altitude in that order. For simplicity, this thesis will deal with two-dimensional positions only. A polyline is described by creating an array of these positions as seen in listing 2.

There is however one library in this thesis which expects coordinates in a different format. Listing 3 shows a polyline in the sense of this library. Here one location is

```
1 [[102.0, 0.0], [103.0, 1.0], [104.0, 0.0], [105.0, 1.0]]
```

Listing 2: Polyline coordinates in nested-array form

represented by an object with x and y properties.

```
1 [{x: 102.0, y: 0.0}, {x: 103.0, y: 1.0}, {x: 104.0, y: 0.0}, {x: 105.0,
     y: 1.0}]
```

Listing 3: Polyline in array-of-objects form

To distinguish these formats in future references the first first format will be called nested-array format, while the latter will be called array-of-objects format.

2.4 Running the algorithms on the web platform

JavaScript was traditionally the only native programming language of web browsers (Reiser and Bläser 2017). With the development of WebAssembly, there seems to be an alternative on its way. This technology, its benefits and drawbacks, will be explained in this chapter.

2.4.1 Introduction to WebAssembly

WebAssembly[3] started in April 2015 with an W3C Community Group[4] and is designed by engineers from the four major browser vendors (Mozilla, Google, Apple and Microsoft). It is a portable low-level bytecode designed as compilation target of high-level languages. By being an abstraction over modern hardware it is language-, hardware-, and platform-independent. It is intended to be run in a stack-based virtual machine. This way it is not restrained to the Web platform or a JavaScript environment. Some key concepts are the structuring into modules with exported and imported definitions and the linear memory model. Memory is represented as a large array of bytes that can be dynamically grown. Security is ensured by the linear memory being disjoint from code space, the execution stack and the engine's data structures. Another feature of WebAssembly is the possibility of streaming compilation and the parallelization of compilation processes. (Haas et al. 2017)

[3] https://webassembly.org/

[4] https://www.w3.org/community/webassembly/

The goals of WebAssembly have been well defined. Its semantics are intended to be safe and fast to execute and to bring portability by language-, hardware- and platform-independence. Furthermore, it should be deterministic and have simple interoperability with the web platform. For its representation, the following goals are declared. It shall be compact and easy to decode, validate and compile. Parallelization and streamable compilation are also mentioned. (Haas et al. 2017)

These goals are not specific to WebAssembly. They can be seen as properties that a low-level compilation target for the web should have. In fact there have been previous attempts to run low-level code on the web. Examples are Microsoft's ActiveX, Native Client (NaCl) and Emscripten, each having issues complying with the goals stated. Java and Flash are examples for managed runtime plugins. Their usage is declining however not at least due to falling short on the goals mentioned above. (Haas et al. 2017)

It is often stated that WebAssembly can bring performance benefits. It makes sense that statically typed machine code beats scripting languages performance wise. It has to be observed however, if the overhead of switching contexts will neglect this performance gain. JavaScript has made a lot of performance improvements over the past years. Not at least Googles development on the V8 engine has brought JavaScript to an acceptable speed for extensive calculations. Modern engines observe the execution of running JavaScript code and will perform optimizations that can be compared to optimizations of compilers. (Clark 2017)

The JavaScript ecosystem has rapidly evolved the past years. Thanks to package managers like Bower, npm and Yarn it is simple to pull code from external sources into ones codebase. Initially thought for server sided JavaScript execution the ecosystem has found its way into front-end development via module bundlers like browserify, webpack and rollup. In course of this growth many algorithms and implementations have been ported to JavaScript for use on the web. With WebAssembly this ecosystem can be broadened even further. By lifting the language barrier, existing work of many more programmers can be reused on the web. Whole libraries exclusive for native development could be imported by a few simple tweaks. Codecs not supported by browsers can be made available for use in any browser supporting WebAssembly. (Surma 2018)

The Emscripten toolchain There are various compilers with WebAssembly as compilation target. In this thesis the Emscripten toolchain is used. Other notable compilers are wasm-pack[5] for Rust projects and AssemblyScript[6] for a TypeScript subset. This latter compiler is particularly interesting as TypeScript, itself a superset of JavaScript,

[5] `https://rustwasm.github.io/`

[6] `https://github.com/AssemblyScript/assemblyscript`

```c
#include <stdio.h>
int main()
{
  int sum = 0;
  for (int i = 1; i <= 100; i++)
    sum += i;
  printf("1+...+100=%d\n", sum);
  return 0;
}
```

```
define i32 @main() {
  %1 = alloca i32, align 4
  %sum = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 0, i32* %1
  store i32 0, i32* %sum, align 4
  store i32 1, i32* %i, align 4
  br label %2

; <label>:2
  %3 = load i32* %i, align 4
  %4 = icmp sle i32 %3, 100
  br i1 %4, label %5, label %12

; <label>:5
  %6 = load i32* %i, align 4
  %7 = load i32* %sum, align 4
  %8 = add nsw i32 %7, %6
  store i32 %8, i32* %sum, align 4
  br label %9

; <label>:9
  %10 = load i32* %i, align 4
  %11 = add nsw i32 %10, 1
  store i32 %11, i32* %i, align 4
  br label %2

; <label>:12
  %13 = load i32* %sum, align 4
  %14 = call i32 (i8*, ...)*
        @printf(i8* getelementptr inbounds
          ([14 x i8]* @.str, i32 0, i32 0),
          i32 %13)
  ret i32 0
}
```

```
function _main() {
  var __stackBase__  = STACKTOP;
  STACKTOP += 12;
  var __label__ = -1;
  while(1) switch(__label__) {
    case -1:
      var $1 = __stackBase__;
      var $sum = __stackBase__+4;
      var $i = __stackBase__+8;
      HEAP[$1] = 0;
      HEAP[$sum] = 0;
      HEAP[$i] = 0;
      __label__ = 0; break;
    case 0:
      var $3 = HEAP[$i];
      var $4 = $3 <= 100;
      if ($4) { __label__ = 1; break; }
      else    { __label__ = 2; break; }
    case 1:
      var $6 = HEAP[$i];
      var $7 = HEAP[$sum];
      var $8 = $7 + $6;
      HEAP[$sum] = $8;
      __label__ = 3; break;
    case 3:
      var $10 = HEAP[$i];
      var $11 = $10 + 1;
      HEAP[$i] = $11;
      __label__ = 0; break;
    case 2:
      var $13 = HEAP[$sum];
      var $14 = _printf(__str, $13);
      STACKTOP = __stackBase__;
      return 0;
  }
}
```

Figure 2: Example code when compiling a C program (left) to asm.js (right) through LLVM bytecode (middle) without optimizations. (Zakai 2011)

is a popular choice among web developers. This reduces the friction for WebAssembly integration as it is not necessary to learn a new language.

Emscripten[7] started with the goal to compile unmodified C and C++ applications to JavaScript. This is achieved by acting as a compiler backend to LLVM assembly. High level languages compile through a frontend into the LLVM intermediate representation. Well known frontends are Clang and LLVM-GCC. From there it gets passed through a backend to generate the architecture specific machine code. Emscripten hooks in here to generate asm.js, a performant JavaScript subset. In figure 2 one such example chain can be seen. On the left is the original C code which sums up numbers from 1 to 100. The resulting LLVM assembly can be seen in the middle. It is definitely more verbose, but easier to work on for the backend compiler. Notable are the allocation instructions, the labeled code blocks and code flow moves. The JavaScript representation on the right is the nearly one to one translation of the LLVM assembly. The branching is done via a switch-in-for loop, memory is implemented by a JavaScript array named HEAP and LLVM assembly functions calls become normal JavaScript function calls like `_printf()`. Through optimizations the code becomes more compact and only then more performant. (Zakai 2011)

It is in fact this project that inspired the creation of WebAssembly. It was even called

[7] `https://webassembly.org/`

the "natural evolution of asm.js"[8]. As of May 2018 Emscripten changed its default output to WebAssembly[9] while still supporting asm.js. Currently the default backend named `fastcomp` generates the WebAssembly bytecode from asm.js. A new backend however is about to take its place that compiles directly from LLVM (Zakai 2019).

The compiler is only one part of the Emscripten toolchain. Part of it are various APIs, for example for file system emulation or network calls, and tools like the compiler mentioned.

[8] `https://groups.google.com/forum/#!topic/emscripten-discuss/k-egXO7AkJY/discussion`
[9] `https://github.com/emscripten-core/emscripten/pull/6419`

3 Implementation of a performance benchmark

In this chapter the approach to improve the performance of a simplification algorithm in a web browser via WebAssembly will be explained. The go-to library for this kind of operation is Simplify.js. It is the JavaScript implementation of the Douglas-Peucker algorithm with optional radial distance preprocessing. The library will be rebuilt in the C programming language and compiled to WebAssembly with Emscripten. A web page is built to produce benchmarking insights to compare the two approaches performance wise.

3.1 State of the art: Simplify.js

Simplify.js calls itself a "tiny high-performance JavaScript polyline simplification library"[10]. It was extracted from Leaflet, the "leading open-source JavaScript library for mobile-friendly interactive maps"[11]. Due to its usage in leaflet and Turf.js, a geospatial analysis library, it is the most common used library for polyline simplification. The library itself currently has 20,066 weekly downloads on the npm platform while the Turf.js derivate @turf/simplify has 30,389. Turf.js maintains an unmodified fork of the library in its own repository. The mentioned mapping library Leaflet is downloaded 189,228 times a week.

The Douglas-Peucker algorithm is implemented with an optional radial distance preprocessing routine. This preprocessing trades performance for quality. Thus the mode for disabling this routine is called highest quality.

Interestingly the library expects coordinates to be a list of objects with x and y properties. GeoJSON and TopoJSON however store coordinates in nested array form (see chapter 2.3). Luckily since the library is small and written in JavaScript any skilled web developer can easily fork and modify the code for his own purpose. This is even pointed out in the library's source code. The fact that Turf.js, which can be seen as a convenience wrapper for processing GeoJSON data, decided to keep the library as is might indicate some benefit to this format. Listing 4 shows how Turf.js calls Simplify.js. Instead of altering the source code the data is transformed back and forth between the formats on each call. It is questionable if this practice is advisable at all.

Since it is not clear which case is faster, and given how simple the required changes are, two versions of Simplify.js will be tested. The original version, which expects the coordinates to be in array-of-objects format and the altered version, which operates on nested arrays. Listing 5 shows an extract of the changes performed on the library. Instead

[10] `https://mourner.giformthub.io/simplify-js/`
[11] `https://leafletjs.com/`

```
function simplifyLine(coordinates, tolerance, highQuality) {
    return simplifyJS(coordinates.map(function (coord) {
        return {x: coord[0], y: coord[1], z: coord[2]};
    }), tolerance, highQuality).map(function (coords) {
        return (coords.z) ? [coords.x, coords.y, coords.z] : [coords.x,
    coords.y];
    });
}
```

Listing 4: Turf.js usage of simplify.js

of using properties, the coordinate values are accessed by index. Except for the removal of the licensing header the alterations are restricted to these kind of changes. The full list of changes can be viewed in `lib/simplify-js-alternative/simplify.diff`.

```
13,14c4,5
<     var dx = p1.x - p2.x,
<         dy = p1.y - p2.y;
---
>     var dx = p1[0] - p2[0],
>         dy = p1[1] - p2[1];
```

Listing 5: Snippet of the difference between the original Simplify.js and alternative

3.2 The WebAssembly solution

In scope of this thesis a library will be created that implements the same procedure as Simplify.js in C code. It will be made available on the web platform through WebAssembly. In the style of the model library it will be called Simplify.wasm. The compiler to be used will be Emscripten as it is the standard for porting C code to WebAssembly.

As mentioned, the first step is to port Simplify.js to the C programming language. The file `lib/simplify-wasm/simplify.c` shows the attempt. It is kept as close to the JavaScript library as possible. This may result in C-untypical coding style but prevents skewed results from unexpected optimizations to the procedure itself. The entry point is not the `main`-function but a function called `simplify`. This is specified to the compiler as can be seen in listing 6.

Furthermore, the functions `malloc` and `free` from the standard library are made available for the host environment. Another option specifies the optimisation level. With `O3` the highest level is chosen. The closure compiler minifies the JavaScript glue code. Compiling the code through Emscripten produces a binary file in wasm format and the glue code as JavaScript. These files are called `simplify.wasm` and `simplify.js` respectively.

```
OPTIMIZE="-O3"

simplify.wasm simplify.js: simplify.c
  emcc \
    ${OPTIMIZE} \
    --closure 1 \
    -s WASM=1 \
    -s ALLOW_MEMORY_GROWTH=1 \
    -s MODULARIZE=1 \
    -s EXPORT_ES6=1 \
    -s EXPORTED_FUNCTIONS='["_simplify", "_malloc", "_free"]' \
    -o simplify.js \
    simplify.c
```

Listing 6: The call to compile the C source code to WebAssembly in a Makefile

An example usage can be seen in `lib/simplify-wasm/example.html`. Even though the memory access is abstracted in this example the process is still unhandy and far from a drop-in replacement of Simplify.js. Thus in `lib/simplify-wasm/index.js` a further abstraction to the Emscripten emitted code was written. The exported function `simplifyWasm` handles module instantiation, memory access and the correct call to the exported wasm function. Finding the correct path to the wasm binary is not always clear when the code is imported from another location. The proposed solution is to leave the resolving of the code-path to an asset bundler that processes the file in a preprocessing step.

```
export async function simplifyWasm(coords, tolerance, highestQuality) {
  const module = await getModule()
  const buffer = storeCoords(module, coords)
  const resultInfo = module._simplify(
    buffer,
    coords.length * 2,
    tolerance,
    highestQuality
  )
  module._free(buffer)
  return loadResultAndFreeMemory(module, resultInfo)
}
```

Listing 7: The top level function to invoke the WebAssembly simplification.

Listing 7 shows the function `simplifyWasm`. Further explanation will follow regarding the abstractions `getModule`, `storeCoords` and `loadResultAndFreeMemory`.

Module instantiation will be done on the first call only but requires the function to be asynchronous. For a neater experience in handling Emscripten modules, a utility function named `initEmscripten`[12] was written to turn the module factory into a JavaScript `Promise` that resolves on finished compilation. The usage of this function can be seen in listing 8. The resulting WebAssembly module is cached in the variable `emscriptenModule`.

```
1 let emscriptenModule
2 export async function getModule() {
3   if (!emscriptenModule)
4     emscriptenModule = initEmscriptenModule(wasmModuleFactory, wasmUrl)
5   return await emscriptenModule
6 }
```

Listing 8: Caching the instantiated Emscripten module

Storing coordinates into the module memory is done in the function `storeCoords`. Emscripten offers multiple views on the module memory. These correspond to the available WebAssembly data types (e.g. HEAP8, HEAPU8, HEAPF32, HEAPF64, ...)[13]. As Javascript numbers are always represented as a double-precision 64-bit binary[14] (IEEE 754-2008), the HEAPF64-view is the way to go to not lose precision. Accordingly the datatype double is used in C to work with the data. Listing 9 shows the transfer of coordinates into the module memory. In line 3 the memory is allocated using the exported `malloc`-function. A JavaScript TypedArray is used for accessing the buffer such that the loop for storing the values (lines 5 - 8) is trivial.

To read the result back from memory one has to look at how the simplification is returned in the C code. Listing 10 shows the entry point for the C code. This is the function which gets called from JavaScript. As expected, arrays are represented as pointers with corresponding length. The first block of code (line 2 - 6) is only meant for declaring needed variables. Lines 8 to 12 mark the radial distance preprocessing. The result of this simplification is stored in an auxiliary array named `resultRdDistance`. In this case, `points` will have to point to the new array and the length is adjusted. Finally, the Douglas-Peucker procedure is invoked after reserving enough memory. The auxiliary array can be freed afterwards. The problem now is to return the result pointer and the

[12] /lib/wasm-util/initEmscripten.js

[13] `https://emscripten.org/docs/api_reference/preamble.js.html#type-accessors-for-the-memory-model`

[14] `https://www.ecma-international.org/ecma-262/6.0/#sec-4.3.20`

```
1  export function storeCoords(module, coords) {
2    const flatSize = coords.length * 2
3    const offset = module._malloc(flatSize * Float64Array.
       BYTES_PER_ELEMENT)
4    const heapView = new Float64Array(module.HEAPF64.buffer, offset,
       flatSize)
5    for (let i = 0; i < coords.length; i++) {
6      heapView[2 * i] = coords[i][0]
7      heapView[2 * i + 1] = coords[i][1]
8    }
9    return offset
10 }
```

Listing 9: The storeCoords function

array length back to the calling code. The fact that pointers in Emscripten are represented by 32bit will be exploited to return a fixed size array of two integers containing the values. We can now look back at how the JavaScript code reads the result.

Listing 11 shows the code to read the values back from module memory. The result pointer and its length are acquired by dereferencing the `resultInfo`-array. The buffer to use is the heap for unsigned 32-bit integers. This information can then be used to align the Float64Array-view on the 64-bit heap. Construction of the appropriate coordinate representation, by reversing the flattening, can be looked up in the same file. It is realised in the `unflattenCoords` function. At last it is important to actually free the memory reserved for both the result and the result-information. The exported method `free` is the way to go here.

3.3 File sizes

For web applications an important measure is the size of libraries. It defines the cost of including the functionality in terms of how much the application size will grow. When it gets too large, especially users with low bandwidth are discriminated as it might be impossible to load the app at all in a reasonable time. Even with fast internet, loading times are relevant as users expect a fast time to first interaction. Also users with limited data plans are glad when developers keep their bundle size to a minimum.

The file sizes in this chapter will be given as the gzipped size. gzip is a file format for compressed files based on the DEFLATE algorithm. It is natively supported by all browsers and the most common web server software. So this is the format that files will be transmitted in on production applications.

For JavaScript applications there is also the possibility of reducing filesize by code minification. This is the process of reformating the source code without changing the

```
int* simplify(double * points, int length, double tolerance, int
    highestQuality) {
    double sqTolerance = tolerance * tolerance;
    double* resultRdDistance = NULL;
    double* result = NULL;
    int resultLength;

    if (!highestQuality) {
        resultRdDistance = malloc(length * sizeof(double));
        length = simplifyRadialDist(points, length, sqTolerance,
    resultRdDistance);
        points = resultRdDistance;
    }

    result = malloc(length * sizeof(double));
    resultLength = simplifyDouglasPeucker(points, length, sqTolerance,
    result);
    free(resultRdDistance);

    int* resultInfo = malloc(2);
    resultInfo[0] = (int) result;
    resultInfo[1] = resultLength;
    return resultInfo;
}
```

Listing 10: Entrypoint in the C-file

```
export function loadResultAndFreeMemory(module, resultInfo) {
  const [resultPointer, resultLength] = new Uint32Array(
    module.HEAPU32.buffer,
    resultInfo,
    2
  )
  const simplified = new Float64Array(
    module.HEAPF64.buffer,
    resultPointer,
    resultLength
  )
  const coords = unflattenCoords(simplified)
  module._free(resultInfo)
  module._free(resultPointer)
  return coords
```

Listing 11: Loading coordinates back from module memory

functionality. Optimization are brought for example by removing unnecessary parts like spaces and comments or reducing variable names to single letters. Minification is often done in asset bundlers that process the JavaScript source files and produce the bundled application code.

For the WebAssembly solution there are two files required to work with it. The `.wasm` bytecode and JavaScript glue code. The glue code is already minified by the Emscripten compiler. The binary has a size of 3.8KB while the JavaScript code has a total of 3.1KB. Simplify.js on the other hand will merely need a size of 1.1KB. With minification the size shrinks to 638 bytes.

File size was not the main priority when producing the WebAssembly solution. There are ways to further shrink the size of the bytecode. As of now it contains the logic of the library but also necessary functionality from the C standard library. These were added by Emscripten automatically. The bloat comes from using the memory management functions `malloc` and `free`. If the goal was to reduce the file size, one would have to get along without memory management at all. This would even be possible in this case as the simplification process is a self-contained process and the module has no other usage. The input size is known beforehand so instead of creating reserved memory one could just append the result in memory at the location directly after the input feature. The function would merely need to return the result size. After the call is finished and the result is read by JavaScript the memory is not needed anymore. A test build was made which renounced from memory management. The size of the wasm bytecode shrunk to 507 byte and the glue code to 2.8KB. By using vanilla JavaScript API one could even ditch the glue code altogether (Surma 2019).

For simplicity the memory management was left in as the optimizations would require more careful engineering to ensure correct functionality. The example above shows however, that there is enormous potential to cut the size. Even file sizes below the JavaScript original are possible.

3.4 The implementation of a web framework

The performance comparison of the two methods will be realized in a web page. It will be built as a frontend web application that allows the user to specify the input parameters of the benchmark. These parameters are: the polyline to simplify, a range of tolerances to use for simplification and if the so called high quality mode shall be used. By building this application it will be possible to test a variety of use cases on multiple devices. Also the behavior of the algorithms can be researched under different preconditions. In the scope of this thesis a few cases will be investigated. The application structure will now

be introduced.

3.4.1 External libraries

The dynamic aspects of the web page will be built in JavaScript. Webpack[15] will be used to bundle the application code and use compilers like babel[16] on the source code. As mentioned in section 3.2, the bundler is also useful for handling references to the WebAssembly binary as it resolves the filename to the correct download path to use. There will be intentionally no transpiling of the JavaScript code to older versions of the ECMA standard. This is often done to increase compatibility with older browsers. Luckily this is not a requirement in this case and by refraining from this practice there will also be no unintentional impact on the application performance. Libraries in use are Benchmark.js[17] for statistically significant benchmarking results, React[18] for the building the user interface and Chart.js[19] for drawing graphs.

3.4.2 The application logic

The web page consists of static and dynamic content. The static parts refer to the header and footer with explanation about the project. Those are written directly into the root HTML document. The dynamic parts are injected by JavaScript. Those will be further discussed in this chapter as they are the main application logic.

The web app is built to test a variety of cases with multiple datapoints. As mentioned, Benchmark.js will be used for statistically significant results. It is however rather slow as it needs about 5 to 6 seconds per datapoint. This is why multiple types of benchmarking methods are implemented. Figure 3 shows the corresponding UML diagram of the application. One can see the UI components in the top-left corner. The root component is `App`. It gathers all the internal state of its children and passes state down where it is needed.

3.4.3 Benchmark cases and chart types

In the upper right corner the different Use-Cases are listed. These cases implement a function `"fn"` to benchmark. Additional methods for setting up the function and clean up afterwards can be implemented as given by the parent class `BenchmarkCase`. Concrete cases can be created by instantiating one of the benchmark cases with a defined set of

[15] https://webpack.js.org/
[16] https://babeljs.io/
[17] https://benchmarkjs.com/
[18] https://reactjs.org/
[19] https://www.chartjs.org/

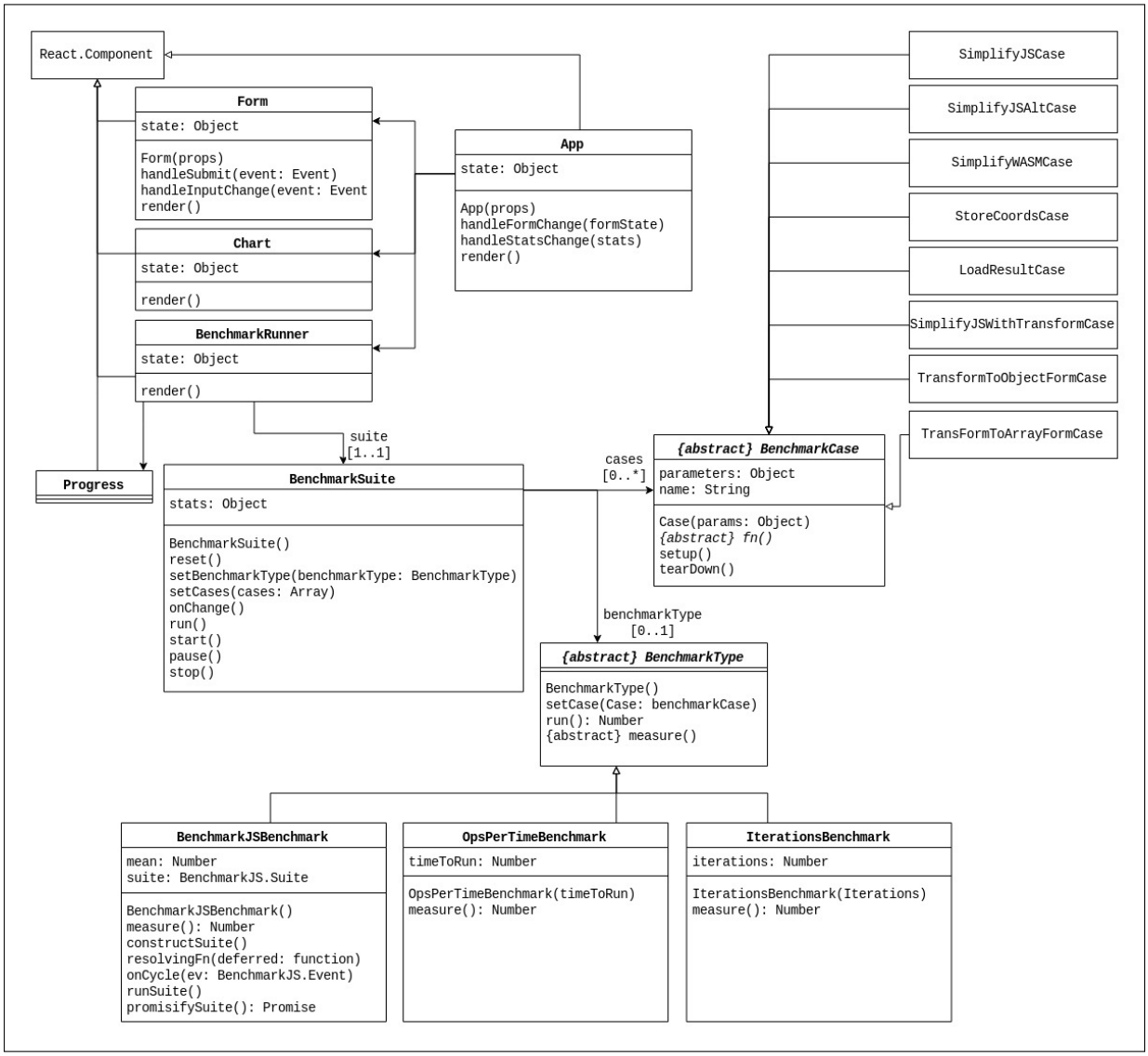

Figure 3: UML diagram of the benchmarking application

parameters. There are three charts that will be rendered using a subset of these cases. These are:

- **Simplify.js vs Simplify.wasm** - This Chart shows the performance of the simplification by Simplify.js, the altered version of Simplify.js and the newly developed Simplify.wasm.
- **Simplify.wasm runtime analysis** - To further gain insights to WebAssembly performance this stacked barchart shows the runtime of a call to Simplify.wasm. It is partitioned into time spent for preparing data (`storeCords`), the algorithm itself and the time it took for the coordinates being restored from memory (`loadResult`).

- **Turf.js method runtime analysis** - The last chart will use a similar structure. This time it analyses the performance impact of the back and forth transformation of data used in Truf.js.

3.4.4 The different benchmark types

On the bottom the different types of Benchmarks implemented can be seen. They all implement the abstract `measure` function to return the mean time to run a function specified in the given `BenchmarkCase`. The `IterationsBenchmark` runs the function a specified number of times, while the `OpsPerTimeBenchmark` always runs a certain amount of milliseconds to run as much iterations as possible. Both methods got their benefits and drawbacks. Using the iterations approach one cannot determine the time the benchmark runs beforehand. With fast devices and a small number of iterations one can even fall in the trap of the duration falling under the accuracy of the timer used. Those results would be unusable of course. It is however a very fast way of determining the speed of a function. And it holds valuable for getting a first approximation of how the algorithms perform over the span of datapoints. The second type, the operations per time benchmark, seems to overcome this problem. It is however prune to garbage collection, engine optimizations and other background processes. (Mathias Bynens 2010)

Benchmark.js combines these approaches. In a first step it approximates the runtime in a few cycles. From this value it calculates the number of iterations to reach an uncertainty of at most 1%. Then the samples are gathered. (Hossain 2012)

3.4.5 The benchmark suite

For running multiple benchmarks the class `BenchmarkSuite` was created. It takes a list of `BenchmarkCases` and runs them through a `BenchmarkType`. The suite manages starting, pausing and stopping of going through list. It updates the statistics gathered on each cycle. By injecting an `onCycle` method, the `Runner` component can give live feedback about the progress.

Figure 3.4.5 shows the state machine of the suite. Based on this diagram the user interface component shows action buttons so the user can interact with the state. While running, the suite checks if a state change was requested and acts accordingly by pausing the benchmarks or resetting all statistics gathered when stopping.

3.4.6 The user interface

The user interface has three sections. One for configuring input parameters. One for controlling the benchmark process and at last a diagram of the results. Figure 5 shows

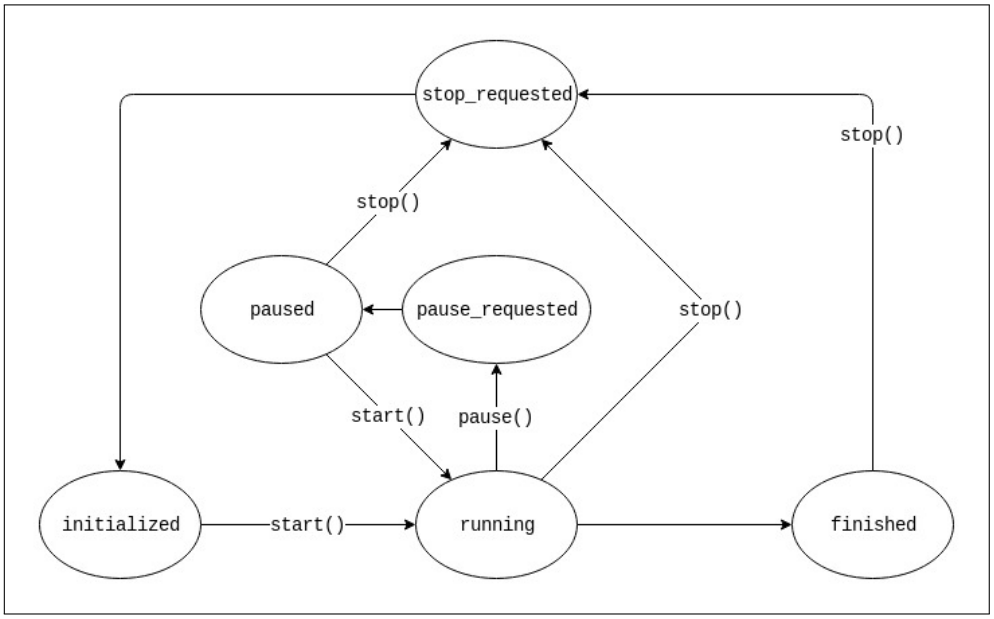

Figure 4: The state machine for the benchmark suite

the user interface.

Settings At first the input parameters of the algorithm have to be specified. For that there are some polylines prepared to choose from. They are introduced in chapter 3.5. Instead of testing a single tolerance value the user can specify a range. This way the behavior of the algorithms can be observed in one chart. The high quality mode got its name from Simplify.js. If it is enabled there will be no radial-distance preprocessing step before applying the Douglas-Peucker routine. The next option determines which benchmarks will be run. The options are mentioned in chapter 3.4.3. One of the three benchmark methods implemented can be selected. Depending on the method chosen additional options will show to further specify the benchmark parameters. The last option deals with chart rendering. Debouncing limits the rate at which functions are called. In this case the chart will delay rendering when datapoints come in at a fast rate.

Run Benchmark This is the control that displays the status of the benchmark suite. Here benchmarks can be started, stopped, paused and resumed. It also shows the progress of the benchmarks completed in percentage and absolute numbers.

Chart The chart shows a live diagram of the results. The title represents the selected chart. The legend gives information on which benchmark cases will run. Also the algorithm parameters (dataset and high quality mode) and current platform description can be found here. The tolerance range maps over the x-Axis. On the y-Axis two scales can

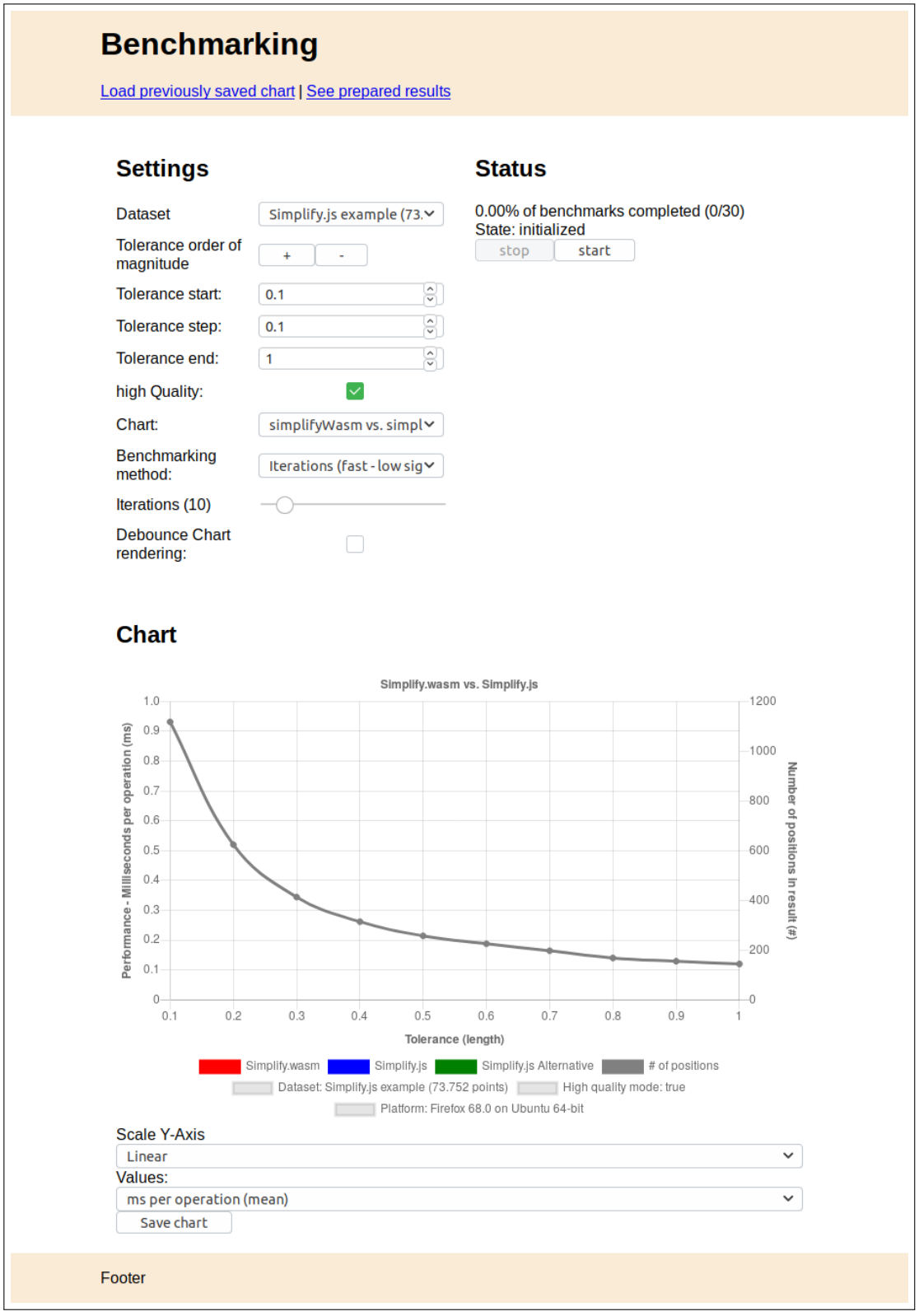

Figure 5: The user interface for benchmarking application.

be seen. The left hand shows by which unit the performance is displayed. This scale corresponds to the colored lines. Every chart will show the number of positions in the result as a grey line. Its scale is displayed on the right. This information is important for selecting a proper tolerance range as it shows if a appropriate order of magnitude has been chosen. Below the chart additional control elements are placed to adjust the visualization. The first selection lets the user choose between a linear or logarithmic y-Axis. The second one changes the unit of measure for performance. The two options are the mean time in milliseconds per operation (ms) and the number of operations that can be run in one second (hz). These options are only available for the chart "Simplify.wasm vs. Simplify.js" as the other two charts are stacked bar charts where changing the default options won't make sense. Finally the result can be saved via a download button. A separate page can be fed with this file to display the diagram only.

3.5 The test data

Here the test data will be shown. There are two data sets chosen to operate on. The first is a testing sample used in Simplify.js the second one a boundary generated from the OpenStreetMap (OSM) data.

Simplify.js example This is the polyline used by Simplify.js to demonstrate its capabilities. Figure 6 shows the widget on its homepage. The user can modify the parameters with the interactive elements and view the live result. The data comes from a 10.700 mile car route from Lisboa, Portugal to Singapore and is based on OpenStreetMap data. The line is defined by 73,752 positions. Even with low tolerances this number reduces drastically. This example shows perfectly why it is important to generalize polylines before rendering them.

Bavaria outline The second polyline used for benchmarking contains 116,829 positions. It represents the outline of a german federate state, namely bavaria. It was extracted from the OSM dataset by selecting administrative boundaries. On the contrary to the former polyline this one is a closed line, often used in polygons to represent a surface. The plotted line can be seen in figure 7.

Simple line There is a third line used in the application to choose from. This one is however not used for benchmarking since it contains only 8 points. It is merely a placeholder to prevent the client application to load a bigger data sets from the server on

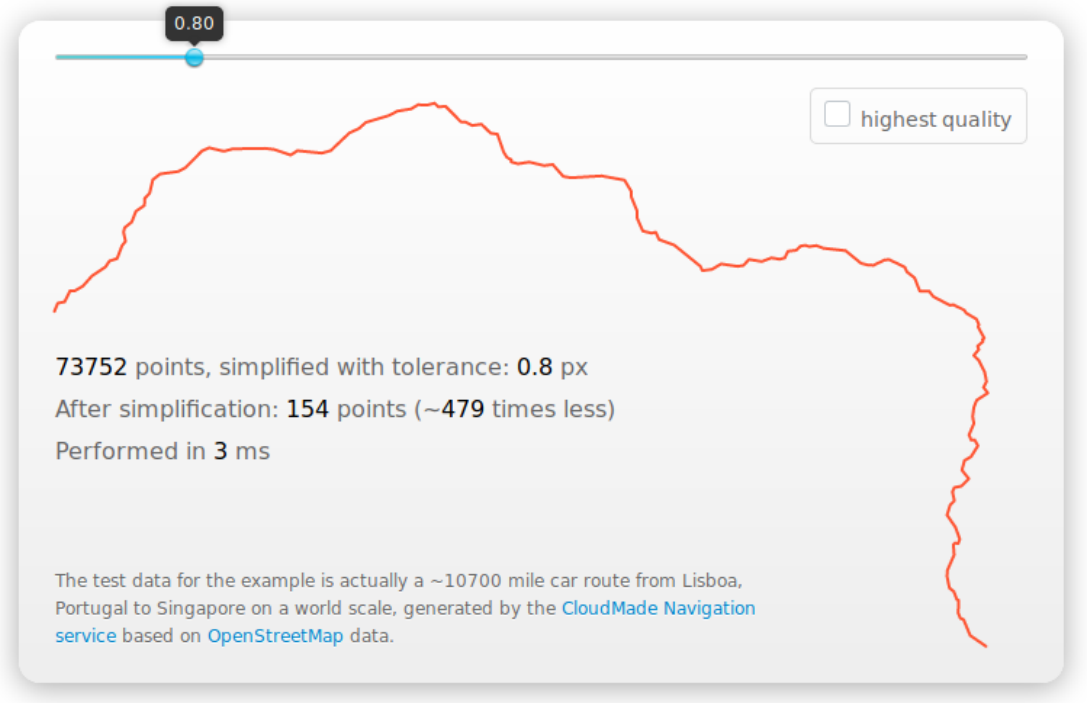

Figure 6: The Simplify.js test data visualized

page load. This way the transmitted data size will be reduced. The larger lines will only be requested when they are actually needed.

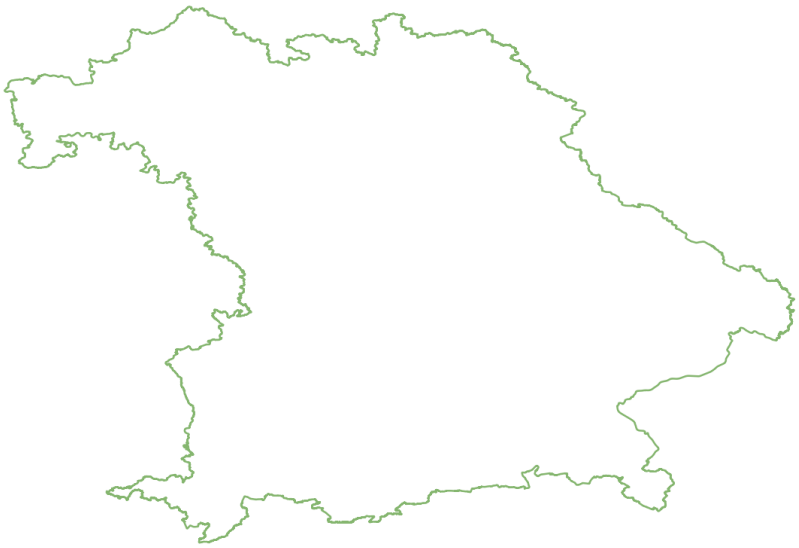

Figure 7: The Bavaria test data visualized

4 Benchmark results

In this chapter the results are presented. There were a multiple tests performed. Multiple devices were used to run several benchmarks on different browsers and under various parameters. To decide which benchmarks had to run, first all the problem dimensions were clarified. Devices will be categorized into desktop and mobile devices. The browsers to test will come from the four major browser vendors which were involved in WebAssembly development. These are Firefox from Mozilla, Chrome from Google, Edge from Microsoft and Safari from Apple. For either of the two data sets a fixed range of tolerances is set to maintain consistency across the diagrams. The other parameter "high quality" can be either switched on or off. The three chart types are explained in chapter 3.4.3.

All benchmark results shown here can be interactively explored at the web page provided together with this thesis. The static files lie in the `build` folder. The results can be found when following the "show prepared results"-link on the home page.

Each section in this chapter describes a set of benchmarks run on the same system. A table in the beginning will indicate the problem dimensions chosen to inspect. After a description of the system and a short summary of the case the results will be presented in the form of graphs. Those are the graphs produced from the application described in chapter 3.4. Here the results will only be briefly characterized. A further analysis will follow in the next chapter.

4.1 Case 1 - WebAssembly vs JavaScript in different browsers

Device:	Desktop		Mobile	
Browser:	Firefox	Chrome	Edge	Safari
Dataset:	Simplify.js example		Bavaria outline	
High Quality:	On		Off	
Charts:	Simplify.js vs. Simplify.wasm	Simplify.wasm runtime analysis	Turf.js runtime analysis	

Table 1: Problem dimensions of Case 1

At first it will be observed how the algorithms perform under different browsers. The chart to use for this is the "Simplify.js vs Simplify.wasm" chart. For that a Windows system was chosen as it allows to run benchmarks under three of the four browsers in question. The dataset is the Simplify.js example which will be simplified with and without the high quality mode.

The device is a HP Pavilion x360 - 14-ba101ng[20] convertible. It contains an Intel® Core™ i5-8250U Processor with 4 cores and 6MB cache. The operating system is Windows 10 and the browsers are on their newest versions with Chrome 75, Firefox 68 and Edge 44.18362.1.0.

Table 1 summarizes the setting. For each problem dimension the chosen characteristics are highlighted in green color. The number of benchmark diagrams in a chapter is determined by the multitude of characteristics selected. In the case here there are three browsers tested each with two quality options resulting in six diagrams to be produced.

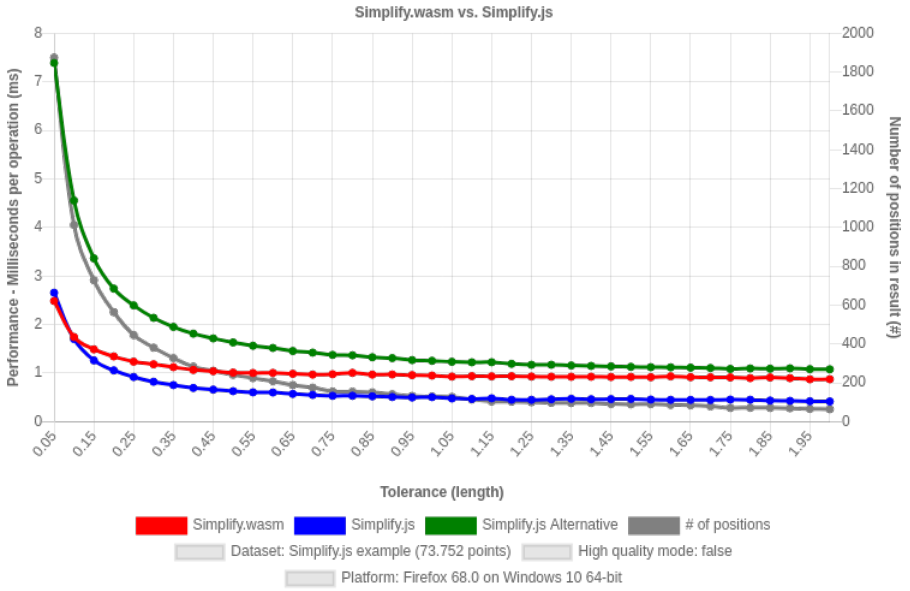

Figure 8: Simplify.wasm vs. Simplify.js benchmark result of Windows device with Firefox browser on dataset "Simplify.js example" without high quality mode.

The first two graphs (figure 8 and 9) show the results for the Firefox browser. Here and in all subsequent charts of this chapter the red line indicates the performance of Simplify.wasm, the blue line represents Simplify.js and the green line its alternative that operates on coordinates as nested arrays. The gray line represents the number of positions that remain in the simplified polyline.

Simplify.js runs without the high quality mode per default. Here at the smallest tolerance chosen the WebAssembly solution is the fastest method. It is overtaken immediately by the original JavaScript implementation where it continues to be the fastest one of the three methods. The alternative is slowest in every case.

In the case of the high quality mode enabled however the original and the WebAssembly solution switch places. Here Simplify.wasm is always faster. The Simplify.js alterna-

[20] https://support.hp.com/us-en/product/hp-pavilion-14-ba100-x360-convertible-pc/16851098/model/18280360/document/c05691748

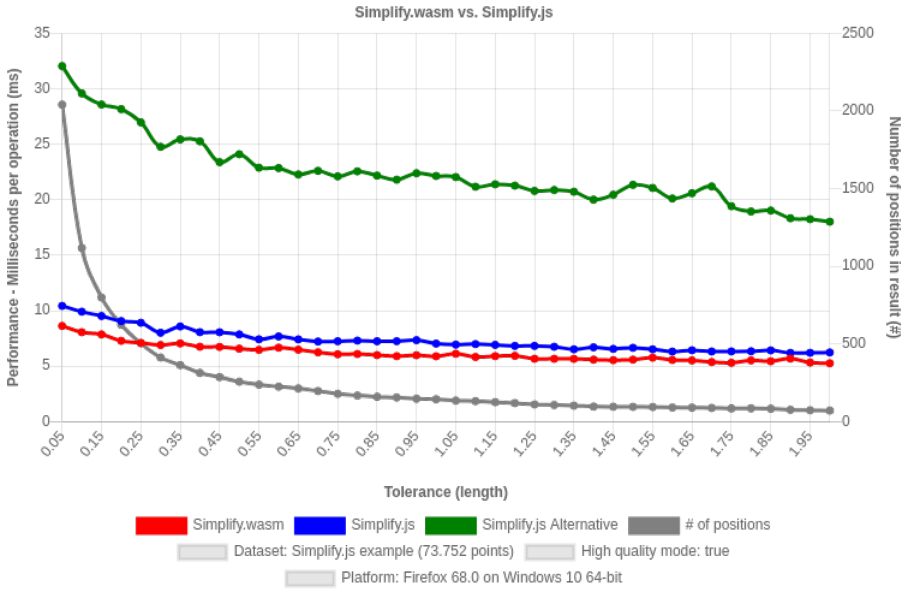

Figure 9: Simplify.wasm vs. Simplify.js benchmark result of Windows device with Firefox browser on dataset "Simplify.js example" with high quality mode.

tive clearly separates itself by being much slower than the other two. It does however have a steeper curve as the original and the WebAssembly solution have pretty consistent performance through the whole tolerance range.

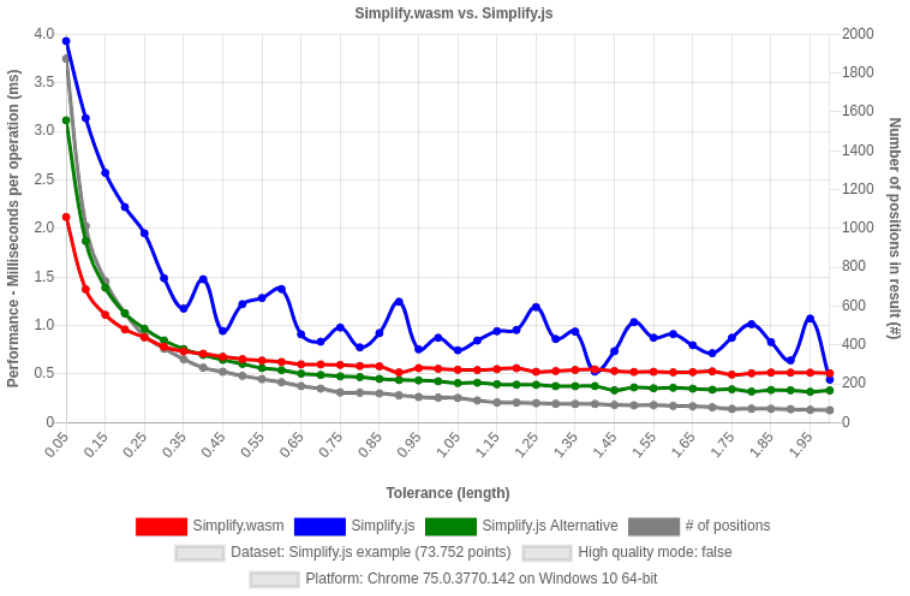

Figure 10: Simplify.wasm vs. Simplify.js benchmark result of Windows device with Chrome browser on dataset "Simplify.js example" without high quality mode.

Figure 10 and 11 show the results under Chrome for the same setting. Here the performance seem to be switched around with the original being the slowest method

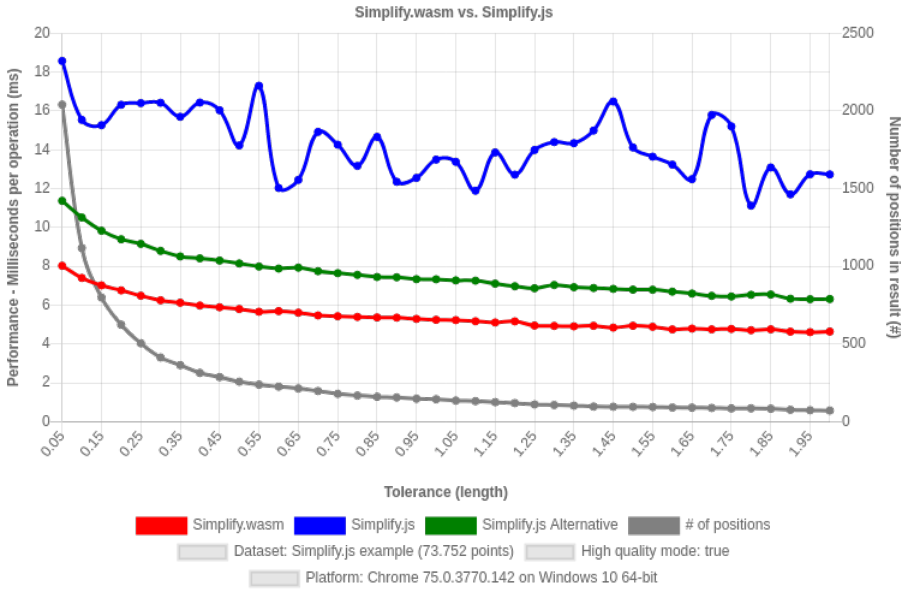

Figure 11: Simplify.wasm vs. Simplify.js benchmark result of Windows device with Chrome browser on dataset "Simplify.js example" with high quality mode.

in both cases. This version has however very inconsistent results. There is no clear curvature which indicates some outside influence to the results. Either there is a flaw in the implementation or a special case of engine optimization was hit.

Without high quality mode the Simplify.wasm gets overtaken by the Simplify.js alternative at 0.4 tolerance. From there on the WebAssembly solution stagnates while the JavaScript one continues to get faster. With high quality enabled the performance gain of WebAssembly is more clear than in Firefox. Here the Simplify.js alternative is the second fastest followed by its original.

Interestingly, in the Edge browser the two JavaScript algorithms perform more alike when high quality disabled. As can be seen in figure 12, the turning point where WebAssembly is not the fastest is at around 0.45 to 0.6. When turning high quality on, the graph in figure 13 resembles the chart from Chrome only with more consistent results for the original implementation.

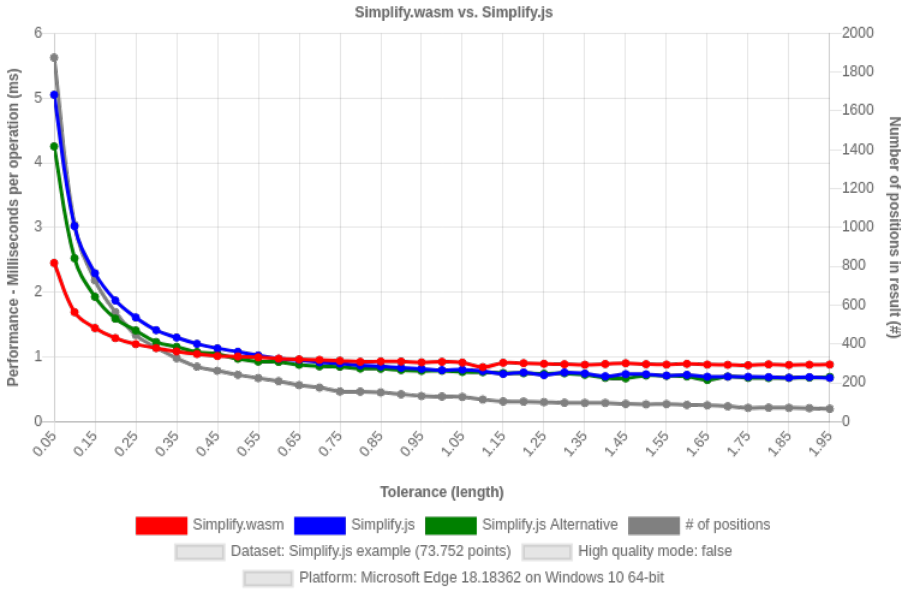

Figure 12: Simplify.wasm vs. Simplify.js benchmark result of Windows device with Edge browser on dataset "Simplify.js example" without high quality mode.

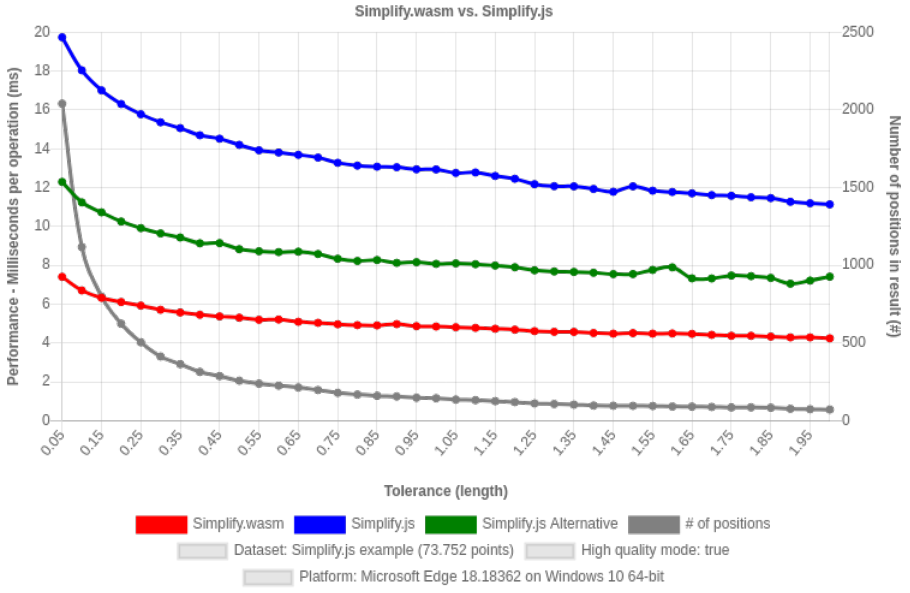

Figure 13: Simplify.wasm vs. Simplify.js benchmark result of Windows device with Edge browser on dataset "Simplify.js example" with high quality mode.

4.2 Case 2 - Simplify.wasm runtime analysis

For this case the same device as in the former case is used. To compare the results of the two cases the same dataset is used. Under the Edge browser the Simplify.wasm runtime analysis was measured. Table 2 summarizes this.

The bar charts visualize where the time is spent in the Simplify.wasm implementation.

Device:	Desktop		Mobile	
Browser:	Firefox	Chrome	Edge	Safari
Dataset:	Simplify.js example		Bavaria outline	
High Quality:	On		Off	
Charts:	Simplify.js vs. Simplify.wasm	Simplify.wasm runtime analysis	Turf.js runtime analysis	

Table 2: Problem dimensions of Case 2

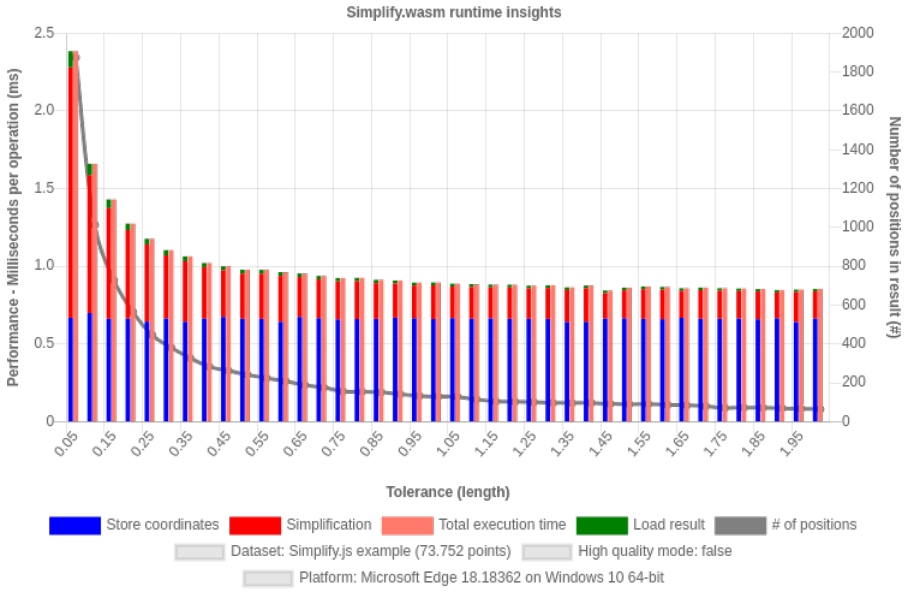

Figure 14: Simplify.wasm runtime analysis benchmark result of Windows device with Edge browser on dataset "Simplify.js example" without high quality mode.

Each data point contains a stacked column to represent the proportion of time spent for each task. The blue section represents the time spent to initialize the memory, the red one the execution of the compiled WebAssembly code. At last the green part will show the time spent for getting the coordinates back in the right format.

Inspecting figures 14 and 15 one immediately notices that the time spent for the memory preparation does not vary in either of the two cases. Also very little time is needed to load the result back from memory especially as the tolerance gets higher. Further analysis of that will follow in chapter 5 as mentioned.

In the case of high quality disabled, the results show a very steep curve of the execution time. Quickly the time span for preparing the memory dominates the process. In the second graph it can be seen that the fraction is significantly lower due to the execution time being consistently higher.

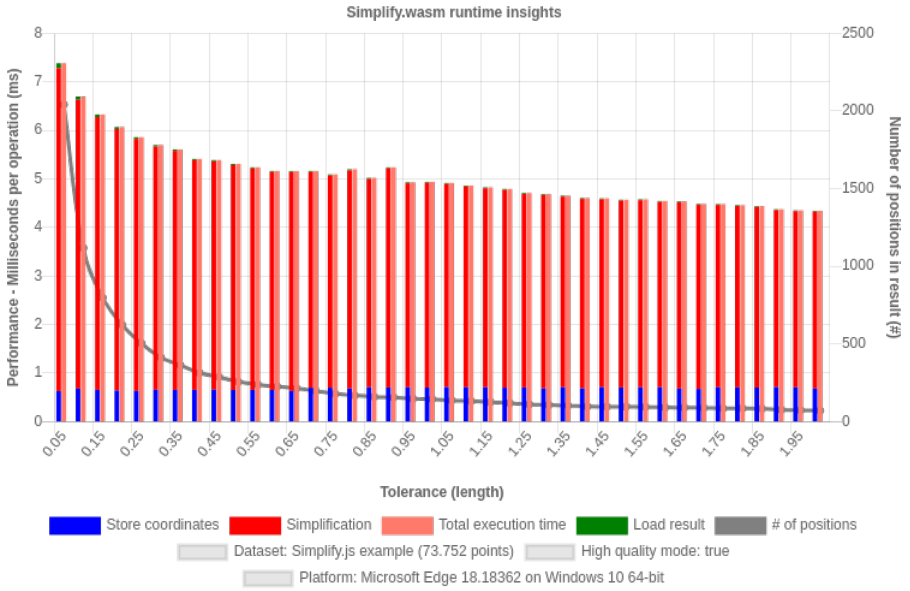

Figure 15: Simplify.wasm runtime analysis benchmark result of Windows device with Edge browser on dataset "Simplify.js example" with high quality mode.

4.3 Case 3 - Benchmarking Safari on MacOS

Device:	Desktop		Mobile	
Browser:	Firefox	Chrome	Edge	Safari
Dataset:	Simplify.js example		Bavaria outline	
High Quality:	On		Off	
Charts:	Simplify.js vs. Simplify.wasm	Simplify.wasm runtime analysis	Turf.js runtime analysis	

Table 3: Problem dimensions of Case 3

A 2018 MacBook Pro 15" will be used to test the safari browser. For comparison the benchmarks will also be held under Firefox on MacOS. This time the bavarian boundary will be simplified with both preprocessing enabled and disabled.

At first figure 16 and 17 show the setting under Firefox. And indeed they are comparable to the results from chapter 4.1. In the case of high quality disabled WebAssembly is fastest for lower tolerances. After a certain point the original is faster while the alternative comes close to WebAssembly performance but without intersection. When enabling the high quality mode the original is more close to Simplify.wasm without being faster. The JavaScript alternative is still trailing behind.

The results of the Safari browser with high quality disabled (figure 18) resembles the figure 12 where the Edge browser was tested. Both JavaScript versions with similar performance surpass the WebAssembly version at one point. Unlike the Edge results the

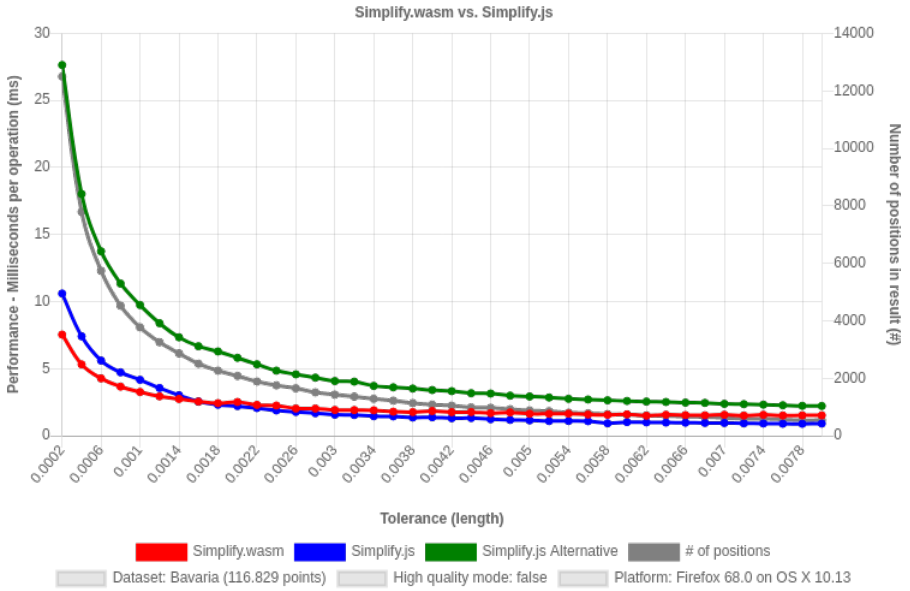

Figure 16: Simplify.wasm vs. Simplify.js benchmark result of MacBook Pro device with Firefox browser on dataset "Bavaria outline" without high quality mode.

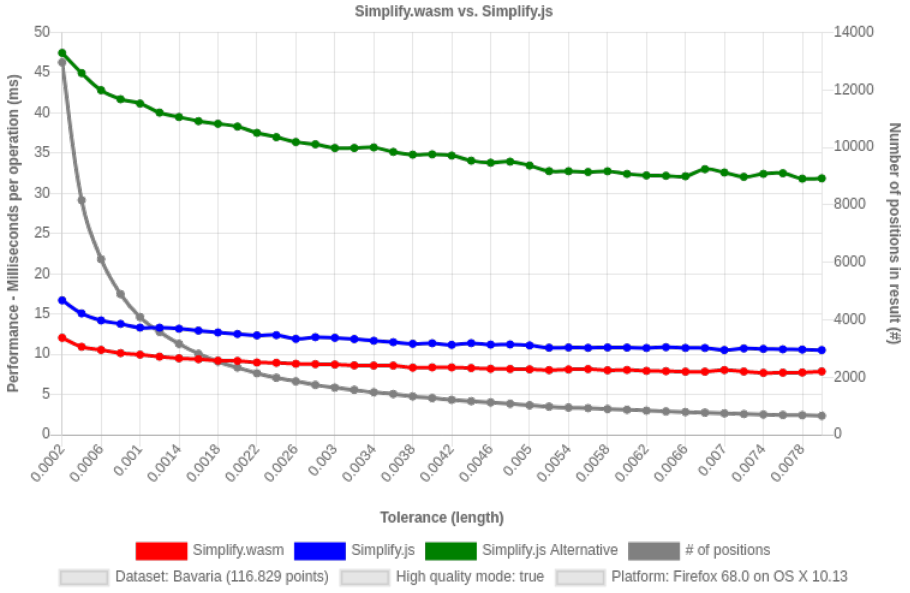

Figure 17: Simplify.wasm vs. Simplify.js benchmark result of MacBook Pro device with Firefox browser on dataset "Bavaria outline" with high quality mode.

original implementation is slightly ahead.

When turning on high quality mode the JavaScript implementations still perform alike. However, Simplify.wasm is clearly faster as seen in figure 19. Simplify.wasm performs here about twice as fast as the algorithms implemented in JavaScript. Those however have a steeper decrease as the tolerance numbers go up.

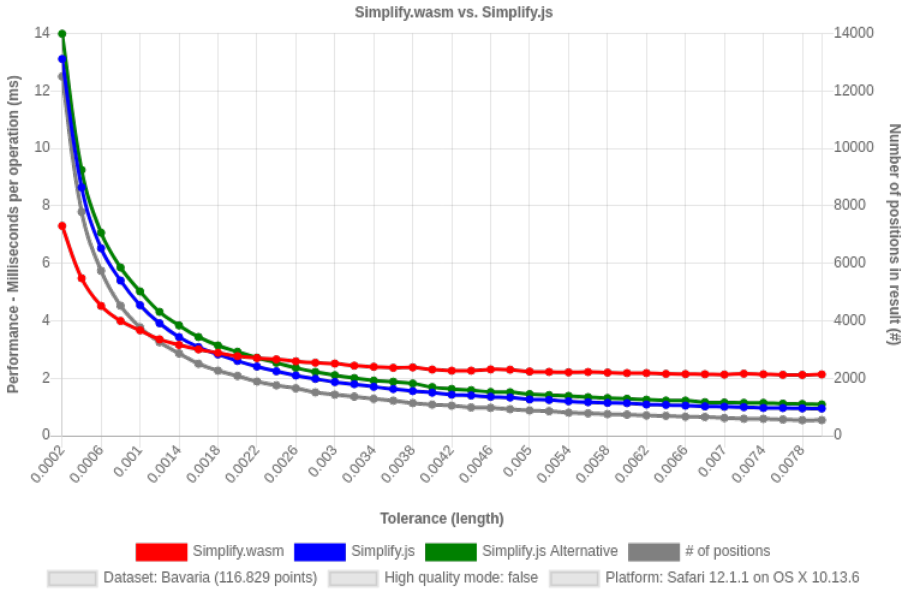

Figure 18: Simplify.wasm vs. Simplify.js benchmark result of MacBook Pro device with Safari browser on dataset "Bavaria outline" without high quality mode.

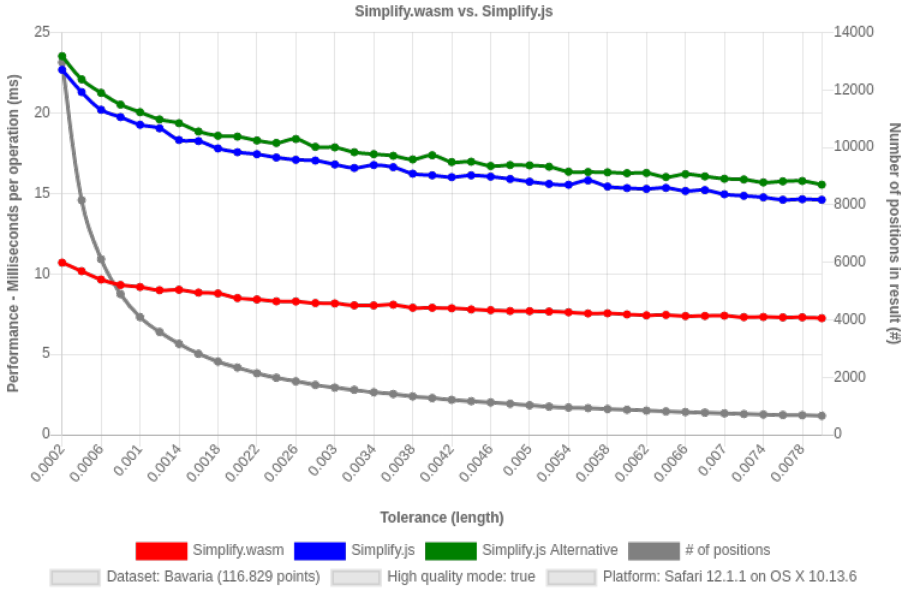

Figure 19: Simplify.wasm vs. Simplify.js benchmark result of MacBook Pro device with Safari browser on dataset "Bavaria outline" with high quality mode.

4.4 Case 4 - Measuring the Turf.js method

In this case the system is a Lenovo Miix 510 convertible with Ubuntu 19.04 as the operating system. Again the bavarian outline is used for simplification with both quality settings. It will be observed if the Turf.js implementation is reasonable. The third kind of chart is in use here, which is similar to the Simplify.wasm insights. There are also stacked bar

Device:	Desktop		Mobile	
Browser:	Firefox	Chrome	Edge	Safari
Dataset:	Simplify.js example		Bavaria outline	
High Quality:	On		Off	
Charts:	Simplify.js vs. Simplify.wasm	Simplify.wasm runtime analysis	Turf.js runtime analysis	

Table 4: Problem dimensions of Case 4

charts used to visualize the time spans of subtasks. The results will be compared to the graphs of the Simplify.js vs. Simplify.wasm chart. As the Turf.js method only makes sense when the original version is faster than the alternative, the benchmarks are performed in the Firefox browser.

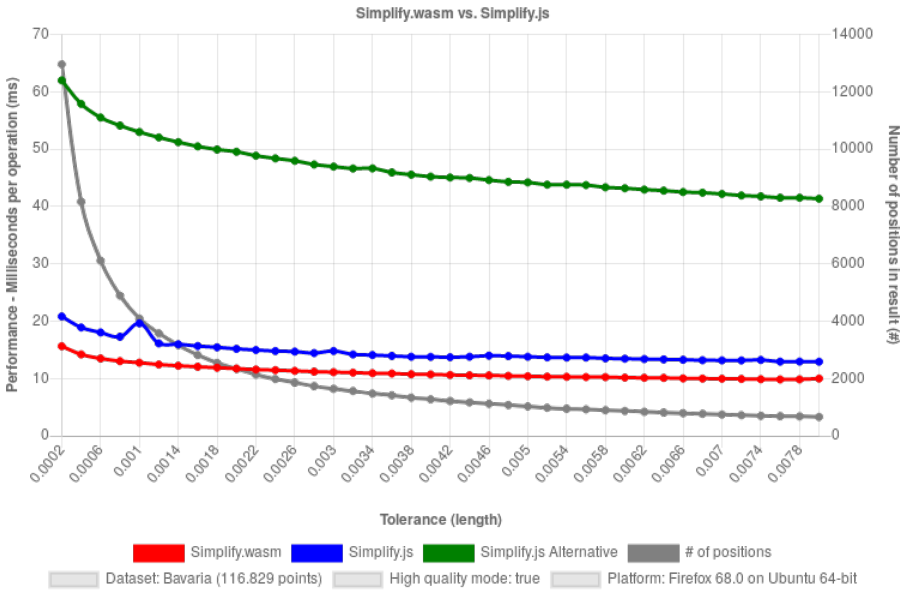

Figure 20: Simplify.wasm vs. Simplify.js benchmark result of Ubuntu device with Firefox browser on dataset "Bavaria outline" with high quality mode.

Figure 20 shows how the JavaScript versions perform with high quality enabled. Here it is clear that the original version is prefereable. In figure 21 one can see the runtime of the Turf.js method. The red bar here stands for the runtime of the Simplify.js function call. The blue and green bar is the time taken for the format transformations before and after the algorithm. Again the preparation of the original data takes significantly longer than the modification of the simplified line. When the alternative implementation is so much slower than the original it is actually more performant to transform the data format. More analysis as mentioned follows in the next chapter.

The next two figures show the case when high quality is disabled. In figure 22 two algorithms seem to converge. And when looking at figure 23 one can see that the data

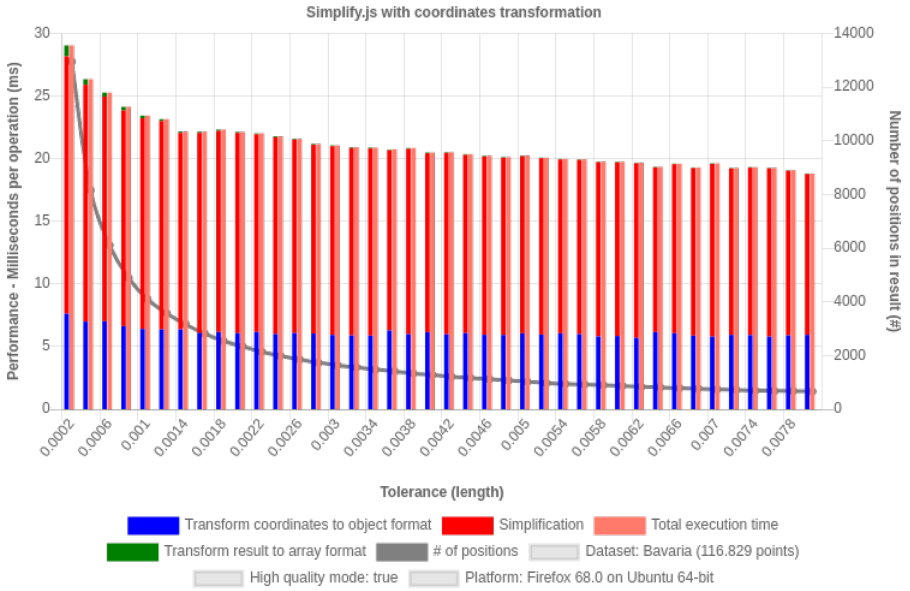

Figure 21: Turf.js simplify benchmark result of Ubuntu device with Firefox browser on dataset "Bavaria outline" with high quality mode.

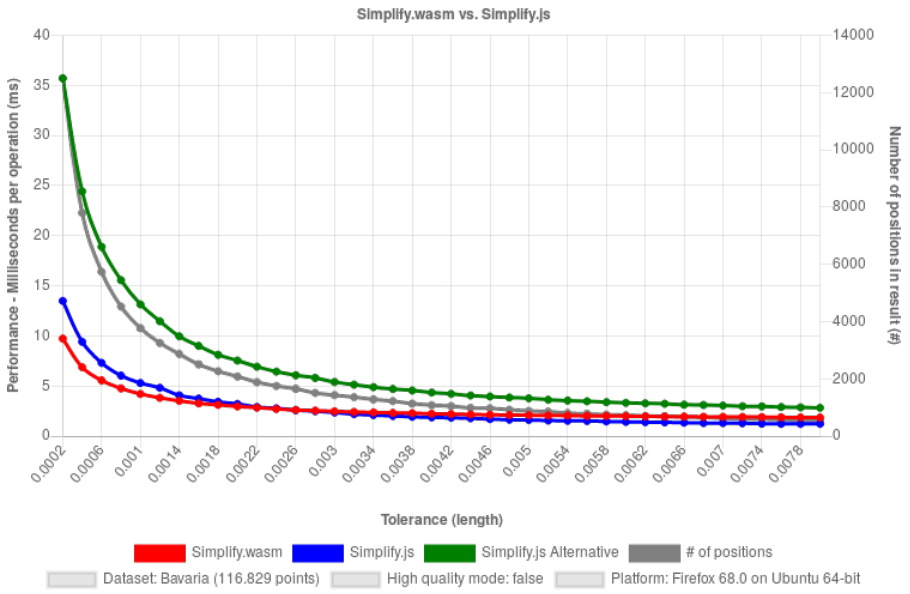

Figure 22: Simplify.wasm vs. Simplify.js benchmark result of Ubuntu device with Firefox browser on dataset "Bavaria outline" without high quality mode.

preparation gets more costly as the tolerance rises. From a tolerance of 0.0014 on the alternative Simplify.js implementation is faster than the Turf.js method.

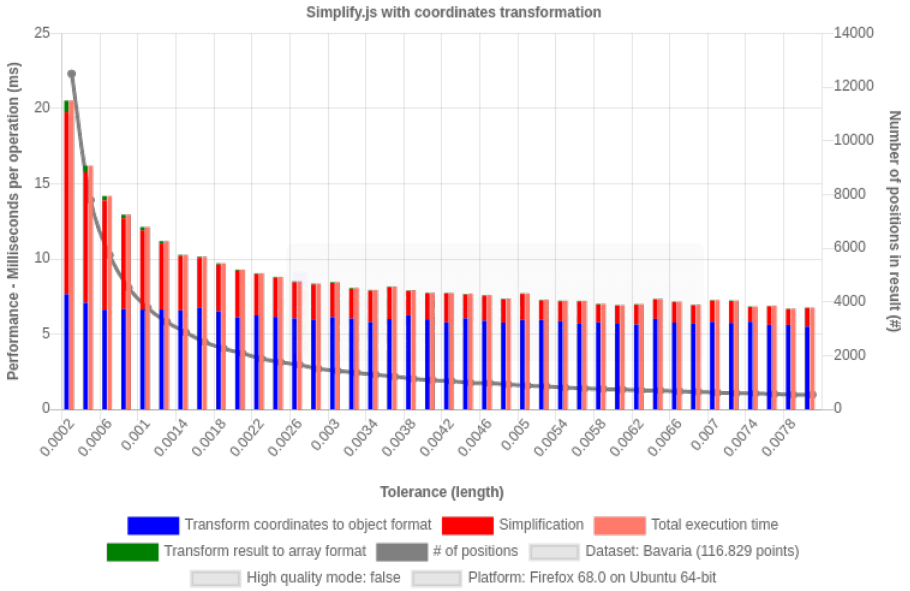

Figure 23: Turf.js simplify benchmark result of Ubuntu device with Firefox browser on dataset "Bavaria outline" without high quality mode.

4.5 Case 5 - Mobile benchmarking

Device:	Desktop		Mobile	
Browser:	Firefox	Chrome	Edge	Safari
Dataset:	Simplify.js example		Bavaria outline	
High Quality:	On		Off	
Charts:	Simplify.js vs. Simplify.wasm	Simplify.wasm runtime analysis	Turf.js runtime analysis	

Table 5: Problem dimensions of Case 5

At last the results from a mobile device are shown. The device is an iPad Air with iOS version 12.4. The Simplify.js example is being generalized using Safari and the Firefox browser. Again both quality settings are used for the benchmarks.

When the high quality parameter is left in its default state the WebAssembly solution is fastest on low tolerance numbers (figure 24). As seen before the JavaScript versions are getting faster when the tolerance increases. The original Simplify.js version surpasses the WebAssembly performance while the alternative tangents it. As it was the case on the desktop system the algorithms perform similarly when high quality is set to `true`. Figure 25 shows that Simplify.wasm is also here the faster method.

Interestingly the results in figure 26 and 27 show the exact same results as the Safari results. In chapter 5 this will be further examined.

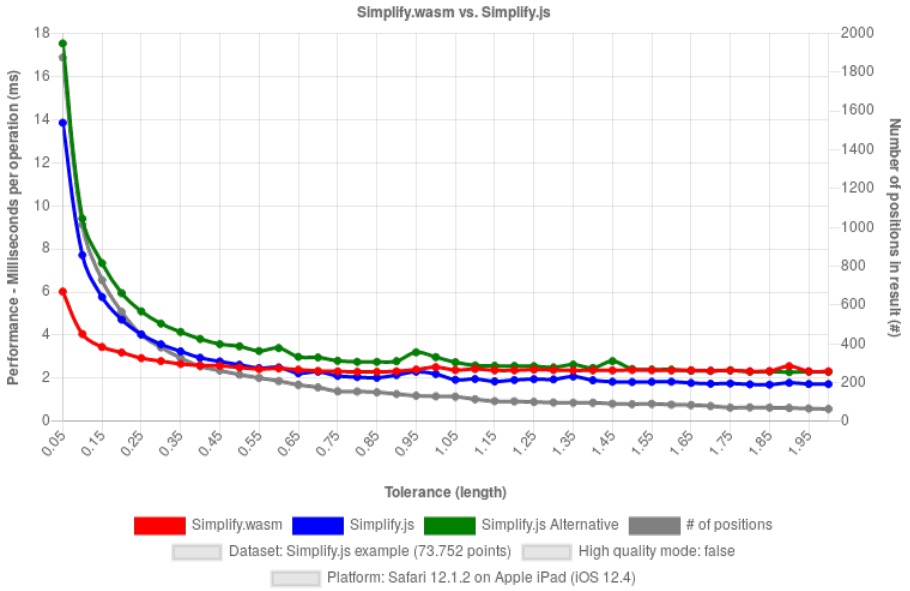

Figure 24: Simplify.wasm vs. Simplify.js benchmark result of iPad device with Safari browser on dataset "Simplify.js example" without high quality mode.

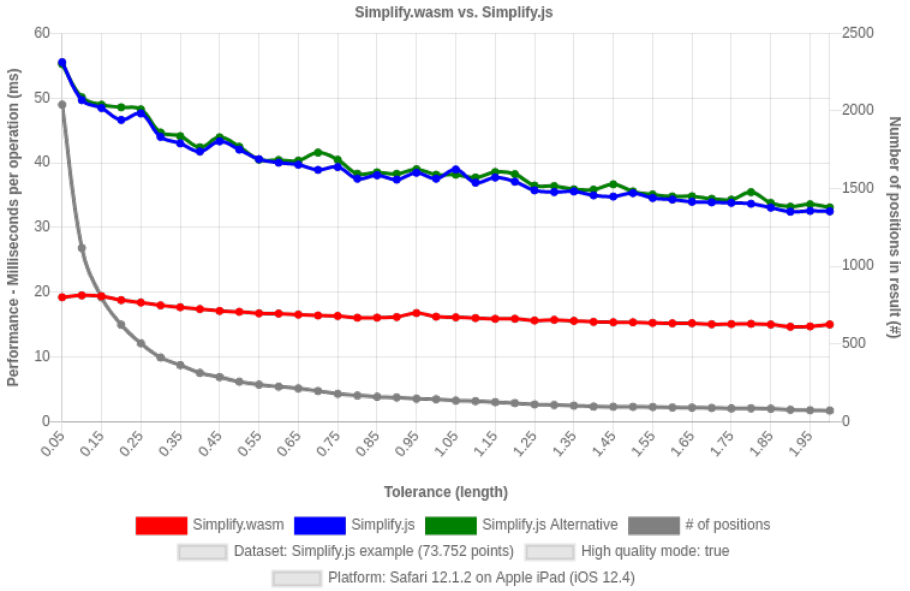

Figure 25: Simplify.wasm vs. Simplify.js benchmark result of iPad device with Safari browser on dataset "Simplify.js example" with high quality mode.

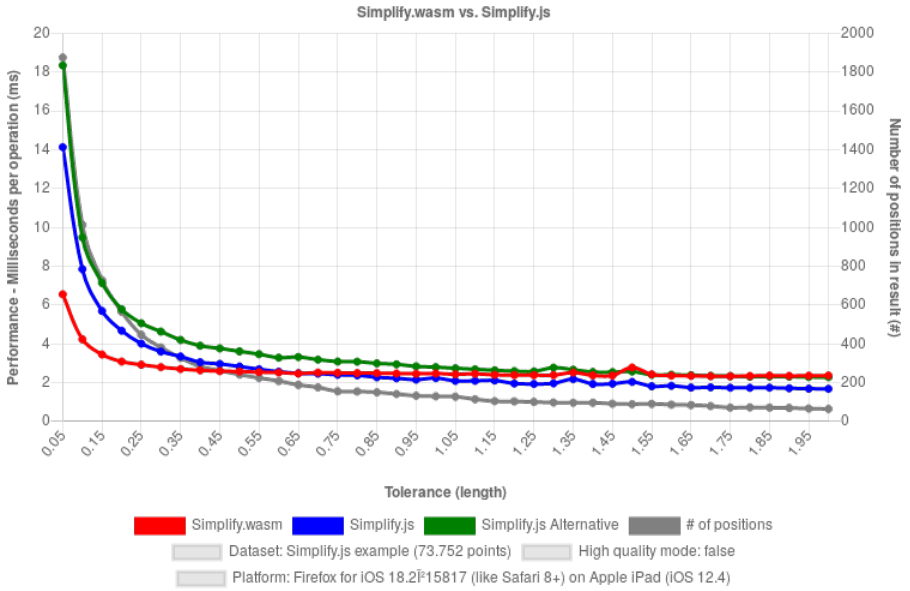

Figure 26: Simplify.wasm vs. Simplify.js benchmark result of iPad device with Firefox browser on dataset "Simplify.js example" without high quality mode.

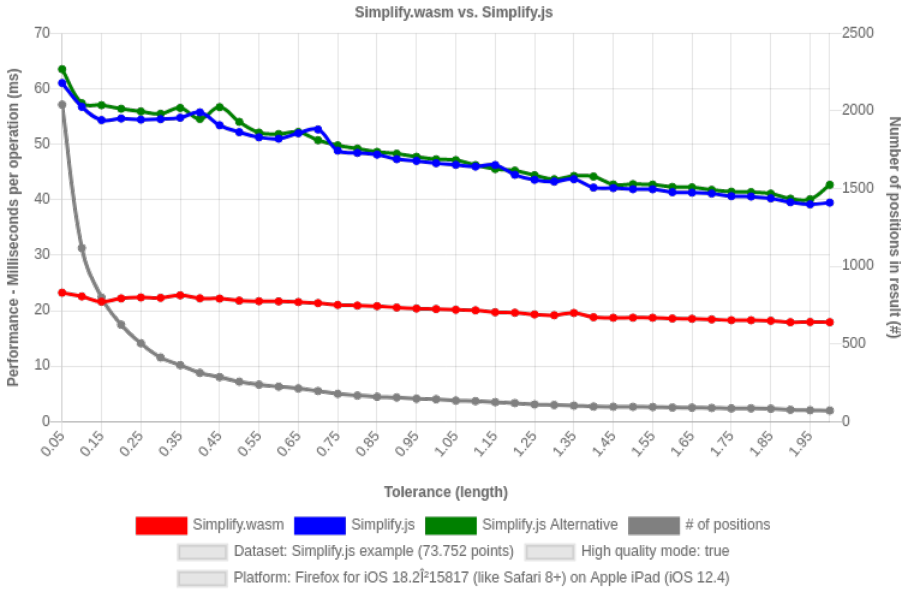

Figure 27: Simplify.wasm vs. Simplify.js benchmark result of iPad device with Firefox browser on dataset "Simplify.js example" with high quality mode.

5 Discussion

In this section the results are interpreted. This section is structured in different questions to answer. First it will be analyzed what the browser differences are. One section will deal with the performance of the pure JavaScript implementations while the next will inspect how Simplify.wasm performs. Then further insights to the performance of the WebAssembly implementation will be given. It will be investigated how long it takes to set up the WebAssembly call and how much time is spent to actually execute the simplification routines. Next the case of Turf.js will be addressed and if its format conversions are reasonable under specific circumstances. Finally, the performance of mobile devices will be evaluated.

5.1 Browser differences for the JavaScript implementations

The first thing to see from the results of chapter 4.1 and 4.3 is that there is actually a considerable performance difference in the two versions of Simplify.js. So here we take a closer look at the JavaScript performance of the browsers. Interestingly, a clear winner between the similar algorithms cannot be determined as the performance is inconsistent across browsers. While the original version is faster in Firefox and Safari, the altered version is superior in Chrome and Edge. This is regardless of whether the high quality mode is switched on or not. The difference is however more significant when the preprocessing step is disabled.

In figure 11 and 13 one can see how similar Chrome and Edge perform with high quality mode enabled. When disabled however, the algorithms perform similar in Edge, while in Chrome the alternative version still improves upon the original.

In Firefox the result is very different. Without the high quality mode the original version performs about 2.5 times faster than the alternative. When disabling the preprocessing, the performance gain is even higher. The original performs constantly 3 times faster.

The same results can be reproduced under Firefox on macOS with the "Bavarian outline" dataset. Interestingly, under Safari the algorithms perform similarly with a small preference to the original version. This applies to either case tested.

With so much variance it is hard to determine the best performing browser regarding the JavaScript implementation. Under the right circumstances Chrome can produce the fastest results with the alternative implementation. Safari is consistently very fast. Even while it falls short to Firefox's results with the original algorithm when high quality is turned on. The greatest discrepancy was produced by Firefox with high quality enabled.

There the alternate version produced the slowest results while the results with Simplify.js can compete with Chrome's results with the Simplify.js alternative. Edge lies between these two browsers with not too bad, but also not the fastest results.

5.2 Browser differences for Simplify.wasm

So diverse the results from last chapter are, so monotonous they will be here. The performance of the Simplify.wasm function is consistent across all browsers tested. This is a major benefit brought by WebAssembly often described as predictable performance.

The variance is very low when the preprocessing is turned off through the high quality mode. The browsers produce about the same runtimes under the same conditions. When high quality is off the Chrome browser got its nose ahead with a mean run time of 0.66ms. Edge follows with 1.02ms and Firefox takes an average 1.10ms. The results of chapter 4.3 show that Safari is a bit faster at the high quality mode than Firefox but slower without.

5.3 Insights into Simplify.wasm

So far, when the performance of Simplify.wasm was addressed, it meant the time spent for the whole process of preparing memory to running the algorithm as WebAssembly bytecode to loading back the result to JavaScript. This makes sense when comparing it to the JavaScript library with the motive to replace it one by one. It does however not produce meaningful comparisons of WebAssembly performance in contrast to the native JavaScript runtime. Further insights to Simplify.wasm call will be provided here.

First the parts where JavaScript is run will be examined. Chapter 4.2 shows that there is as good as no variance in the memory initialization. This is obvious due to the fact that this step is not dependent on any other parameter than the polyline length. Initial versions of the library produced in this thesis were not as efficient in flattening the coordinate array as the final version. By replacing the built-in `Array.prototype.flat`-method with a simple `for` loop, a good optimization was achieved on the JavaScript side of the Simplify.wasm process. The `flat` method is a rather new feature of ECMAScript and its performance might be enhanced in future browser versions. This example shows that when writing JavaScript code one can quickly deviate from the "fast path" even when dealing with simple problems.

On the other side of the process exists the function `loadResult`. It is dependent on the size of the resulting polyline. Since the result is often very small, the green bar can rarely be seen in figures 14 and 15. Merely at low tolerance values the influence is visible. The maximum fraction is at tolerance value 0.05 where the operation takes 4.26% of the

total execution time.

Now when comparing the two graphs one can clearly see that the influence of the JavaScript portions is much greater when the high quality mode is turned off. The time taken for preparing the memory in both cases is about 0.67ms. The execution time of the algorithms is so low in the first case, that it comes down to making up only 24.47% when taking the median values. In case where high quality is enabled the results do not look as drastic. The median value of execution time is 4.31ms and with that much greater than preparation time. Whether JavaScript is at an advantage in the first case, and the high execution time justifies the switch of runtimes in the latter, will be examined in the next chapter.

5.4 Comparison Simplify.wasm vs Simplify.js

The results from chapter 4.1 and 4.3 have shown that Simplify.wasm is always faster when the high quality mode is enabled. The Firefox and Chrome browser are able to optimize at least one version of the JavaScript routines in a way that they come close to the performance of the WebAssembly based library. In Edge, the original version is three times, the alternative two times slower. In Safari, both take about twice the time than Simplify.wasm. On the other hand with preprocessing enabled, there is often one JavaScript version that surpasses the Simplify.wasm performance. In Edge and Safari its even both JavaScript versions that perform faster at higher tolerance values.

This shows that it is not always ideal to replace a library with a WebAssembly based approach. The cost of the overhead might exceed the performance gain when the execution time is low. In section 5.3 it is pointed out, that the pure execution time of the simplification algorithm is fastest with WebAssembly. When preparing the geodata beforehand, for example by serializing it in a binary representation, one could immediately call the bytecode. This poses further effort regarding memory management to the web developer. One has to weigh up the complexity overhead to the performance benefit when considering such approaches.

5.5 Analysis of Turf.js implementation

In this section the method used by Turf.js is evaluated. As seen when using the Chrome or Edge browser, the original library is the slower JavaScript method for simplification. There the data transformation is definitely unfavorable. In Safari, where the JavaScript versions perform equally, the overhead will still lead to worse run times. Lastly the Firefox browser will be examined. The results from chapter 4.4 show that there are indeed cases

where the method from Turf.js achieves better performance than running the alternative Simplify.js library directly. These are the ones where the execution time is large enough to justify the overhead. Namely when high quality is enabled or low tolerance values when high quality is disabled.

Those conclusions are similar to the Simplify.wasm results, where overhead costs also played a role. Here however, one has to bear in mind that it is only one browser that is affected by a performance gain in certain circumstances. In the three other browsers the method is definitely disadvantageous.

5.6 Mobile device analysis

Here the results from the iPad benchmarks in chapter 4.5 are discussed. It stands out that the two browsers show identical results. This is due to Apple Inc.'s security guidelines concerning the iOS App Store. It is specifically restricted for web browsers to use any other engine than WebKit (see Apple Inc. 2019, section 2.5.6). Because of this, Firefox for iOS cannot use the Gecko engine developed by Mozilla. This explains why the two browsers perform equally.

The results from the two browsers lead to similar conclusions as the benchmarks of Safari under macOS did. Simplify.wasm is always fastest in high quality mode. With preprocessing, the JavaScript implementations outperform the WebAssembly based approach at higher tolerance ranges. Both JavaScript versions perform equally.

The mobile device has lower hardware capabilities than the MacBook Pro tested in 4.3. That is why it is not surprising, that the runtimes are higher on the iPad. The concrete results of chapter 4.3 and 4.5 are not directly comparable as different data sets were used. So the focus lies on the behavior of the algorithms. In the case of high quality enabled, the algorithms take about twice as long on the mobile device. This affects both, JavaScript and WebAssembly, equally. With high quality disabled, one can see that the JavaScript performance of the desktop device gets gradually better where at the mobile device the performance stagnates.

6 Conclusion

In this thesis, the performance of simplification algorithms in the context of web applications was analyzed. The dominant library for this task in the JavaScript ecosystem is Simplify.js. It implements the Douglas-Peucker algorithm with optional radial distance preprocessing. By using a technology called WebAssembly, this library was recreated with the goal to achieve a better performance. This recreation was called Simplify.wasm. Also a JavaScript alternative to Simplify.js was tested that operates on a different representation of polylines. To perform several benchmarks on different devices a website was built. The results were gathered by using the library Benchmark.js which produces statistically relevant benchmarks.

It was shown that the WebAssembly based library showed more stable results across different web browsers. The performance of the JavaScript based ones varied greatly. Not only did the absolute run times vary. There were also differences in which JavaScript variant was the faster one. Generally it can be said that the complexity of the operation defines if Simplify.wasm is preferable to Simplify.js. This comes from the fact that there is an overhead of calling Simplify.wasm. To call the WebAssembly code the coordinates will first have to be stored in a linear memory object. With short run times this overhead can exceed the performance gain through WebAssembly. The pure algorithm run time was always shorter with WebAssembly.

The alternative Simplify.js version was created because another major library, Turf.js, implemented an odd routine for simplification. To call Simplify.js the data format of the polyline was transformed back and forth. It could be shown that this process has negative impact to performance in most browsers. Merely one browser showed faster runtimes with this method when the run time of the algorithm was high.

The integration of a WebAssembly module requires more effort than a JavaScript one. Especially as this is a rather recent technology. Hurdles are the sparse tooling, the limited support by older browsers and the necessity to bring in another programming language. While the first two points will develop to the better over time the latter one is tackled by promising projects like AssemblyScript which compiles a subset of TypeScript to WebAssembly.

6.1 Improvements and future work

The library created in this thesis can be improved in a few aspects. First, there is the excessive file size produced by the Emscripten compiler. Section 3.3 already mentions this issue. A solution is proposed to reduce the size of the byte code to about 500 bytes using

gzip. This optimization is achieved by not using standard library functions. Only then will the library be contestable to the JavaScript original in this regard.

Another improvement can be made by changing the abstractions implemented in JavaScript. These were constructed with the goal to achieve a similar experience to Simplify.js. The whole memory management is encapsulated in these abstractions. Each call leads to allocating and freeing the memory for the polyline. One could provide a better interface to the memory management where the user of the library can preload a polyline and execute the algorithm on the prepared memory. Another approach could be to make use of serialized geodata. Whole feature sets could be represented in a binary encoding and simplified in one WebAssembly call.

The geodata types mentioned in this thesis, namely GeoJSON and TopJSON, allow for three dimensional coordinates. The third value often represents altitude. The library Simplify.js provides alternate source code to operate on those types of coordinates. The library created here did not implement a solution for them. If provided, Simplify.wasm will ignore the third coordinate value and run the algorithm on the two dimensional polyline. The functionality could be extended to support calculations on three dimensional positions.

As mentioned, WebAssembly gives the ability to bring code from other programming languages to the web. A library was found that implements several different simplification algorithms in C++. This library can be compiled to WebAssembly. A successful build was developed in the early stages of this thesis. The outcome was not as appropriate for a performance analysis as the direct port of the JavaScript library. In a future work however, this ported library can be used for quality analysis of the different algorithms.

The main goal of projects like WebAssembly is to bring the web platform up to speed with native applications. Especially in the beginning of JavaScript the code that could run in web browsers was slow compared to those. Since then JavaScript engines have evolved and brought huge performance gains, for example by just-in-time compilation. WebAssembly could be a way to reduce the gap to native execution even further. It will be interesting to see how much the cost of running a virtual machine in the browser really is. The code from Simplify.wasm can easily be compiled by general C compilers. A comparison of the native execution to the results from this thesis would be interesting.

References

Ai, Tinghua et al. (2017). "Envelope generation and simplification of polylines using Delaunay triangulation". In: *International Journal of Geographical Information Science* 31.2, pp. 297–319.

Alesheikh, Ali Asghar, Hussein Helali, and HA Behroz (2002). "Web GIS: technologies and its applications". In: *Symposium on geospatial theory, processing and applications*. Vol. 15.

Apple Inc. (June 3, 2019). *App Store Review Guidelines*. URL: `https://developer.apple.com/app-store/review/guidelines/` (visited on 08/15/2019).

Bostock, Mike (2017). *TopoJSON*. URL: `https://github.com/topojson/topojson-specification`.

Brassel, K (1990). "Computergestützte Generalisierung". In: *Schweizerische Gesellschaft für Kartographie,(Ed.) Kartographisches Generalisieren. Zürich, Orell Füssli Graphische Betriebe*, pp. 37–48.

Bray, Tim (2014). "The javascript object notation (json) data interchange format". In: URL: `https://tools.ietf.org/html/rfc8259`.

Brophy, M (1973). "An automated methodology for linear generalization in thematic cartography". In: *proceedings of American congress of surveying and mapping*, pp. 300–314.

Butler, Howard et al. (2016). "The geojson format". In: *RFC 7946; The Internet Engineering Task Force*. URL: `https://tools.ietf.org/html/rfc7946`.

Clark, Lin (Feb. 28, 2017). *What makes WebAssembly fast?* URL: `https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/` (visited on 08/15/2019).

Clayton, Victoria H (1985). "Cartographic generalization: a review of feature simplification and systematic point algorithms". In:

Douglas, David H and Thomas K Peucker (1973). "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature". In: *Cartographica: the international journal for geographic information and geovisualization* 10.2, pp. 112–122.

Haas, Andreas et al. (2017). "Bringing the web up to speed with WebAssembly". In: *ACM SIGPLAN Notices*. Vol. 52. 6. ACM, pp. 185–200.

Hossain, Monsur (Dec. 11, 2012). *benchmark.js: how it works*. URL: `http://monsur.hossa.in/2012/12/11/benchmarkjs.html` (visited on 08/15/2019).

Koning, Elmar de (2011). "Polyline Simplification". In:

Lang, T (1969). "Rules for the robot draughtsmen". In: *The Geographical Magazine* 42.1, pp. 50–51.

Mathias Bynens, John-David Dalton (Dec. 23, 2010). *Bulletproof JavaScript benchmarks*. URL: https://calendar.perfplanet.com/2010/bulletproof-javascript-benchmarks/ (visited on 08/15/2019).

Open GIS Consortium et al. (1999). "OpenGIS simple features specification for SQL". In: *URL: http://www. opengeospatial. org/docs/99-054. pdf*. URL: https://portal.opengeospatial.org/files/?artifact_id=829.

Opheim, Harold (1982). "Fast data reduction of a digitized curve". In: *Geo-processing* 2, pp. 33–40.

Reiser, Micha and Luc Bläser (2017). "Accelerate JavaScript applications by cross-compiling to WebAssembly". In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. ACM, pp. 10–17.

Reumann, K and APM Witkam (1974). *Optimizing Curve Segmentation in Computer Graphics. International Computing Symposium*.

Shi, Wenzhong and ChuiKwan Cheung (2006). "Performance evaluation of line simplification algorithms for vector generalization". In: *The Cartographic Journal* 43.1, pp. 27–44.

Surma, Das (Mar. 2018). *Emscripting a C library to Wasm*. URL: https://developers.google.com/web/updates/2018/03/emscripting-a-c-library (visited on 08/15/2019).

— (Feb. 2019). *Replacing a hot path in your app's JavaScript with WebAssembly*. URL: https://developers.google.com/web/updates/2019/02/hotpath-with-wasm (visited on 08/15/2019).

Theobald, David M (2001). "Understanding topology and shapefiles". In: *Arc-User (April-June)*. URL: https://www.esri.com/news/arcuser/0401/topo.html.

Visvalingam, Maheswari and James D Whyatt (1993). "Line generalisation by repeated elimination of points". In: *The cartographic journal* 30.1, pp. 46–51.

Wagner, Luke (Feb. 28, 2017). *WebAssembly consensus and end of Browser Preview*. URL: https://lists.w3.org/Archives/Public/public-webassembly/2017Feb/0002.html (visited on 08/15/2019).

Zakai, Alon (2011). "Emscripten: an LLVM-to-JavaScript compiler". In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, pp. 301–312.

— (July 1, 2019). *Emscripten and the LLVM WebAssembly backend*. URL: https://v8.dev/blog/emscripten-llvm-wasm (visited on 08/15/2019).

Zhao, Zhiyuan and Alan Saalfeld (1997). "Linear-time sleeve-fitting polyline simplification algorithms. In". In: *Proceedings of AutoCarto 13*. Citeseer.