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

Masterarbeit

Institut für Informatik

Universität Augsburg

vorgelegt von

Alfred Melch

Matrikelnummer xxx

Augsburg, August 2019

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

Abstract

Abstract goes here

Contents

1 Introduction

Compression of polygonal data structures is the task of simplifying geometries while preseving 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. This thesis aims to compare and classify these solutions by various heuristics. Performance and compression rate are quantitative heuristic used. Positional, length and area errors will also be measured to quantify simplification errors. Qualitative heuristics will be determined by a user study. With the rising trend of moving desktop applications to the web platform also geographic information systems (GIS) have experienced the shift towards web browsers [example ESRI Web Gis]. 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. A privilege thus far only granted to the Javascript programming language. 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 [web-source wasm]. The integration into the web platform brings portability to a wide range of platforms like mobile and internet of things (IoT). 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 an example 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 [source]. The mainly targeted high-level languages for compilation are C and C++ [wasm-specs]. Also a compiler for Rust has been developed [rust-wasm working group]. It will be explored how existing implementations could easily be adopted when using a compiler.

1.2 Performance as important factor for web applications

Performance is one of the factors users complain the most about in websites. [Some study] shows that insufficient UI-performance is the main reason for negative user experience. [Another study] states that users will immediately leave websites after only 2 seconds of unresponsiveness. There has been a rapid growth of complex applications running in web-browsers [source]. These so called progressive web apps (PWA) combine the fast reachability of web pages with the feature richness of locally installed applications. Even though these applications can grow quire complex, the requirement for fast page loads and short time to user interaction still remains. One way to cope this need is the use of compression algorithms to reduce the amount of data transmitted and processed. Compression can be lossless. This is often used for the purpose of data transmission. Web servers use lossless compression algorithms like gzip to deflate data. Browsers that implement these algorithms can then fully restore the requested ressources resulting in lower bandwidth usage. Another form of compression removes information of the data in a way that cannot be restored. This is called lossy compression. The most common usage 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 svg transformation used to display topology on the web [source]. Implementing simplification algorithms for use on the web platform can lead to smoother user experience when working with large geodata sets.

1.4 Structure of this thesis

This thesis is structured into a theoretical and a practical component. First the theoretical principles will be reviewed. Topology of polygonal data will be explained as how to store geodata. Also the fundamentals of LineString simplification will be covered.
Then a number of algorithms will be introduced. In this section the each algorithm will be dissected by complexity, characteristics and the possible influence to the heuristics mentioned above.
In the fourth chapter the practical implementation will be presented. This section will dig deeper in several topics important to web development. Such as single page applications, WebAssembly and how web workers will be used for asynchronous execution. The developed application will aim to implement modern best practices in web development such fast time to first user interaction and deferred loading of modules.
The fifth chapter explains how performance will be measured in the web application. After presenting the results the concluion chapter will finish the thesis.

2 Principles

2.1 Polygon basics

2.1.1 Topological aspects

2.2 LineString simplification

2.2.1 Positional errors

2.2.2 Length errors

2.2.3 Area Errors

2.3 Runtimes on the Web

2.3.1 Webassembly

2.4 Coordinate representation

in Javascript

in C

in C++

2.5 Data Formats

2.6 GeoJSON

2.7 TopoJSON

3 Algorithms

Compression algorithms.

3.1 n-th point algorithm

3.2 Random-point routine

3.3 Radial distance algorithm

3.4 Perpendicular distance algorithm

3.5 Reumann-Witkam simplification

3.6 Opheim simplification

3.7 Lang simplification

3.8 Douglas-Peucker simplification

3.8.1 with reduction parameter

3.9 Jenks simplification

3.10 Visvalingam-Whyatt simplification

3.11 Zhao-Saalfeld simplification

3.12 Summary

4 Running the algorithms on the web platform

4.1 Introduction to Webassembly

Present WebAssembly

4.1.1 Existing compilers

Languages from which to compile

emscripten

assemblyscript

rust

4.1.2 Technical hurdles

Managing memory

passing arrays

4.1.3 Benefits of WebAssembly

Why are people going through the hassle of bringing machine code to a platform with a working scripting engine. Is javascript really that aweful. 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. The engine observes the execution of running javascript code and will perform optimizations that can be compared to optimizations of compilers.

Get chart and source of js performance

Source for V8 performance observation

The javascript ecosystem has rapidly evolved the past years. Thanks to package managers like bower, npm and yarn it is super simple to pull code from external sources into ones codebase. In course of this growth many algorithms and implementations have been ported to javascript for use on the web. After all it is however not more then that. A port splits communities and contradicts the DRY principle. With WebAssembly existing work of many programmers can be reused as is for usage on the web. This is the second benefit proposed by the technology. 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. One example could be the promising AV1 codec

more about av1

To summarize the two main benefits that are expected from WebAssembly are perfomance and integration. In this thesis these two benefits will be tested.

4.2 Two test cases - performance and integration

The benefits that WebAssembly promises shall be tested in two seperate Webpages. One for the performance measurements and one to test the integration of existing libraries.

Performance As it is the most applicated algorithm the Douglas-Peucker algorithm will be used for measuring performance. A Javascript implementation is quickly found. simplifyJS. It is the package used by Turf, the most used for geospatial calculations. To produce comparable results the implementation will be based on this package. Since WebAssembly defines a compilation goal several languages can be used for this test.

source for simplify JS

source for turf

Integration An existing implementation of several simplification algorithms has found in the C++ ecosystem. psimpl implements x algorithms distributed as a single header file. It also provides a function for measuring positional errors making it ideal for use in a quality analysis tool for those algorithms.

5 Implementation of a performance benchmark

In this chapter I will explain the approach to improve the performance of a simplification algorithm in a web browser via WebAssembly. 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.

5.1 State of the art: Simplify.js

Simplify.js calls itself a "tiny high-performance JavaScript polyline simplification library"[1]. It was extracted from Leaflet, the "leading open-source JavaScript library for mobile-friendly interactive maps"[2]. 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 while the Turf.js derivate @turf/simplify has 30,389. Turf.js maintains an unmodified fork of the library in its own repository.

So numbers can be added

leaflet downloads

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 object with x and y properties. GeoJSON and TopoJSON however store coordinates in nested array form. 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 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 1 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.

reference object vs array form

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 2 shows an extract of the changes performed on the library. Instead of using properties, the coordinate values are accessed by index. Except for

[1]https://mourner.github.io/simplify-js/

[2]https://leafletjs.com/

```javascript
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 1: Turf.js usage of simplify.js

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`.

```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 2: Snippet of the difference between the original Simplify.js and alternative

5.2 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 use 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 3.

More about the compiler call

Furthermore the functions malloc and free from the standard library are made available for the host environment. 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.
An example usage can be seen in `lib/simplify-wasm/example.html`. Even through

```
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 3: The compiler call

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 however 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)
}
```

../lib/simplify–wasm/index.js

Listing 5.2 shows the function `simplifyWasm`. Further explanaition 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`[3] 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 4. The resulting WebAssembly module is cached in the variable `emscriptenModule`.

```
let emscriptenModule
export async function getModule() {
  if (!emscriptenModule)
    emscriptenModule = initEmscriptenModule(wasmModuleFactory,
   wasmUrl)
  return await emscriptenModule
}
```

Listing 4: My Caption

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, ...)[4]. As Javascript numbers are always represented as a double-precision 64-bit binary[5] (IEEE 754-2008) the HEAP64-view is the way to go to not lose precision. Accordingly the datatype double is used in C to work with the data. Listing 5 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.

```
 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 5: The storeCoords function

Check for coords length ¡ 2

[3] /lib/wasm-util/initEmscripten.js
[4] https://emscripten.org/docs/api_reference/preamble.js.html#type-accessors-for-the-memory-model
[5] https://www.ecma-international.org/ecma-262/6.0/#sec-4.3.20

To read the result back from memory we have to look at how the simplification will be returned in the C code. Listing 6 shows the entry point for the C code. This is the function that 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 array length back to the calling code. The fact that pointers in Emscripten are represented by an integer will be exploited to return a fixed size array of two containing the values. A hacky solution but it works. We can now look back at how the JavaScript code reads the result.

Fact check. evtl unsigned

```
 1  int* simplify(double * points, int length, double tolerance, int
       highestQuality) {
 2      double sqTolerance = tolerance * tolerance;
 3      double* resultRdDistance = NULL;
 4      double* result = NULL;
 5      int resultLength;
 6
 7      if (!highestQuality) {
 8          resultRdDistance = malloc(length * sizeof(double));
 9          length = simplifyRadialDist(points, length, sqTolerance,
       resultRdDistance);
10          points = resultRdDistance;
11      }
12
13      result = malloc(length * sizeof(double));
14      resultLength = simplifyDouglasPeucker(points, length,
       sqTolerance, result);
15      free(resultRdDistance);
16
17      int* resultInfo = malloc(2);
18      resultInfo[0] = (int) result;
19      resultInfo[1] = resultLength;
20      return resultInfo;
21  }
```

Listing 6: Entrypoint in the C-file

Listing 7 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. Constructing 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.

```
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 7: Loading coordinates back from module memory

5.3 The implementation of a web framework

The performance comparison of the two methods will be realized in a web page. It will be a built as a front-end 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.

5.3.1 External libraries

The dynamic aspects of the web page will be built in JavaScript to make it run in the browser. Webpack[6] will be used to bundle the application code and use compilers like babel[7] on the source code. As mentioned in section 5.2 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

[6]https://webpack.js.org/
[7]https://babeljs.io/

impact on the application performance. Libraries in use are Benchmark.js[8] for statistically significant benchmarking results, React[9] for the building the user interface and Chart.js[10] for drawing graphs.

5.3.2 The framework

The web page consist 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.

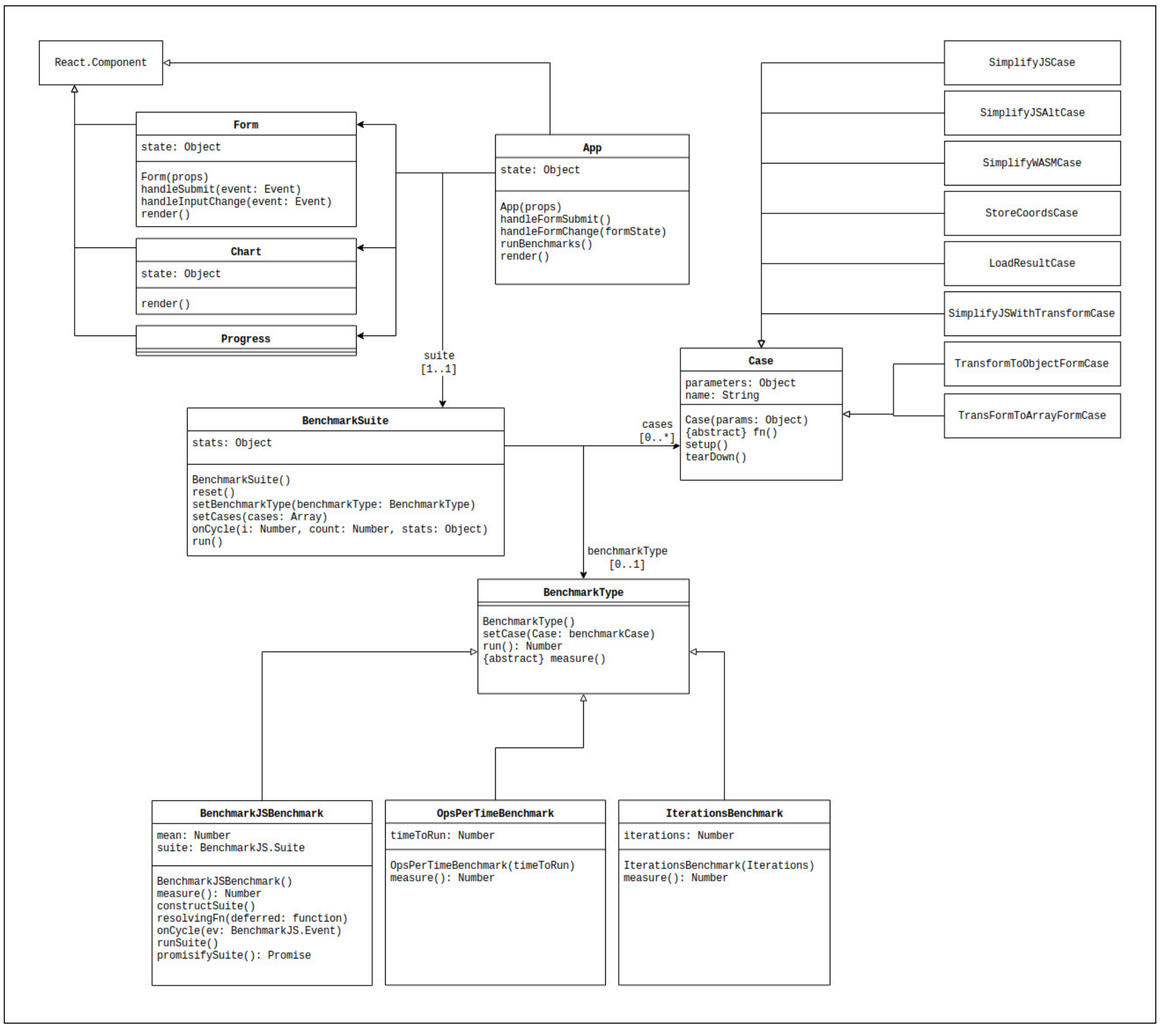


Figure 1: UML diagram of the benchmarking application

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

[8]https://benchmarkjs.com/
[9]https://reactjs.org/
[10]https://www.chartjs.org/

rather slow as it needs about 5 to 6 seconds per datapoint. This is why multiple types of benchmarking methods are implemented. Figure 5.3.2 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.
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 BenchmarkCases with a defined set of 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. Cases
- **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. Cases

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 tun 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. [11]
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. more

[12]

BenchmarkType

BenchmarkSuite

5.3.3 The user interface

[11] https://calendar.perfplanet.com/2010/bulletproof-javascript-benchmarks/
[12] http://monsur.hossa.in/2012/12/11/benchmarkjs.html

6 Compiling an existing C++ library for use on the web

In this chapter I will explain how an existing C++ library was utilized compare different simplification algorithms in a web browser. The library is named *psimpl* and was written in 2011 from Elmar de Koning. It implements various Algorithms used for polyline simplification. This library will be compiled to WebAssembly using the Emscripten compiler. Furthermore a Web-Application will be created for interactively exploring the Algorithms. The main case of application is simplifying polygons, but also polylines will be supported. The data format used to read in the data will be GeoJSON. To maintain topological correctness a intermediate conversion to TopoJSON will be applied if requested.

6.1 State of the art: psimpl

psimpl is a generic C++ library for various polyline simplification algorithms. It consists of a single header file `psimpl.h`. The algorithms implemented are *Nth point*, *distance between points*, *perpendicular distance*, *Reumann-Witkam*, *Opheim*, *Lang*, *Douglas-Peucker* and *Douglas-Peucker variation*. It has to be noted, that the *Douglas-Peucker* implementation uses the *distance between points* routine, also named the *radial distance* routine, as preprocessing step just like Simplify.js (Section 5.1). All these algorithms have a similar templated interface. The goal now is to prepare the library for a compiler.

Describe the error statistics function of psimpl

6.2 Compiling to WebAssembly

As in the previous chapter the compiler created by the Emscripten project will be used. This time the code is not directly meant to be consumed by a web application. It is a generic library. There are no entry points defined that Emscripten can export in WebAssembly. So the entry points will be defined in a new package named psimpl-js. It will contain a C++ file that uses the library, the compiled code and the JavaScript files needed for consumption in a JavaScript project. *psimpl* makes heavy use of C++ template functions which cannot be handled by JavaScript. So there will be entry points written for each exported algorithm. These entry points are the point of intersection between JavaScript and the library. Listing 8 shows one example. They all follow the same procedure. First the pointer given by JavaScript is interpreted as a double-pointer in line 2. This is the beginning of the coordinates array. *psimpl* expects the first and last point of an iterator so the pointer to the last point is calculated (line 3). The appropriate function template from psimpl is instantiated and called with the other given parameters (line 5). The result is stored in an intermediate vector.

```
1 val douglas_peucker(uintptr_t ptr, int length, double tol) {
2     double* begin = reinterpret_cast<double*>(ptr);
3     double* end = begin + length;
4     std::vector<double> resultCoords;
5     psimpl::simplify_douglas_peucker<2>(begin, end, tol, std::
    back_inserter(resultCoords));
6     return val(typed_memory_view(resultCoords.size(), &resultCoords
    [0]));
7 }
```

Listing 8: One entrypoint to the C++ code

Since this is C++ the the capabilities of Emscripten's Embind can be utilized. Embind is realized in the libraries `bind.h`[13] and `val.h`[14]. `val.h` is used for transliterating JavaScript to C++. In this case it is used for the type conversion of C++ Vectors to JavaScript's Typed Arrays as seen at the end of listing 8. On the other hand `bind.h` is used for for binding C++ functions, classes, or enumerations to from JavaScript callable names. Aside from providing a better developer experience this also prevents name mangling in cases where functions are overloaded. Instead of listing the exported functions in the compiler command or annotating it with `EMSCRIPTEN_KEEPALIVE` the developer gives a pointer to the object to bind. Listing 9 shows each entry point bound to a readable name and at last the registered vector datatype. The parameter `my_module` is merely for marking a group of related bindings to avoid name conflicts in bigger projects.

```
EMSCRIPTEN_BINDINGS(my_module) {
    function("nth_point", &nth_point);
    function("radial_distance", &radial_distance);
    function("perpendicular_distance", &perpendicular_distance);
    function("reumann_witkam", &reumann_witkam);
    function("opheim", &opheim);
    function("lang", &lang);
    function("douglas_peucker", &douglas_peucker);
    function("douglas_peucker_n", &douglas_peucker_n);
    register_vector<double>("vector<double>");
}
```

Listing 9: Emscripten bindings

[13]https://emscripten.org/docs/api_reference/bind.h.html#bind-h
[14]https://emscripten.org/docs/api_reference/val.h.html#val-h

Compiler call (–bind)

The library code on JavaScript side is similar to the one in chapter 5.2. This time a function is exported per routine.

More about javascript glue code with listing callSimplification.

6.3 The implementation

The implementation is just as in the last chapter a web page and thus JavaScript is used for the interaction. The source code is bundled with Webpack. React is the UI Component library and babel is used to transform JSX to JavaScript. MobX[15] is introduced as a state management library. It applies functional reactive programming by giving the utility to declare observable variables and triggering the update of derived state and other observers intelligently. To do that MobX observes the usage of observable variables so that only dependent observers react on updates. In contrast to other state libraries MobX does not require the state to be serializable. Many existing data structures can be observed like objects, arrays and class instances. It also does not constrain the state to a single centralized store like Redux[16] does. The final state diagram can be seen in listing 2. It represents the application state in an object model. Since this has drawbacks in showing the information flow the observable variables are marked in red, and computed ones in blue.
On the bottom the three main state objects can be seen. They are implemented as singletons as they represent global application state. Each of them will now be explained.

MapState holds state relevant for the map display. An array of TileLayers defines all possible background layers to choose from. The selected one is stored in `selectedTileLayerId`. The other two variables toggle the display of the vector layers to show.

AlgorithmState stores all the information about the simplification algorithms to choose from. The class `Algorithm` acts as a generalization interface. Each algorithm defines which fields are used to interact with its parameters. These fields hold their current value, so the algorithm can compute its parameters array at any time. The fields also define additional restrictions in their `props` attribute like the number range from which to choose from. An integer field for example, like the n value

[15]`https://mobx.js.org/`
[16]`https://redux.js.org/`

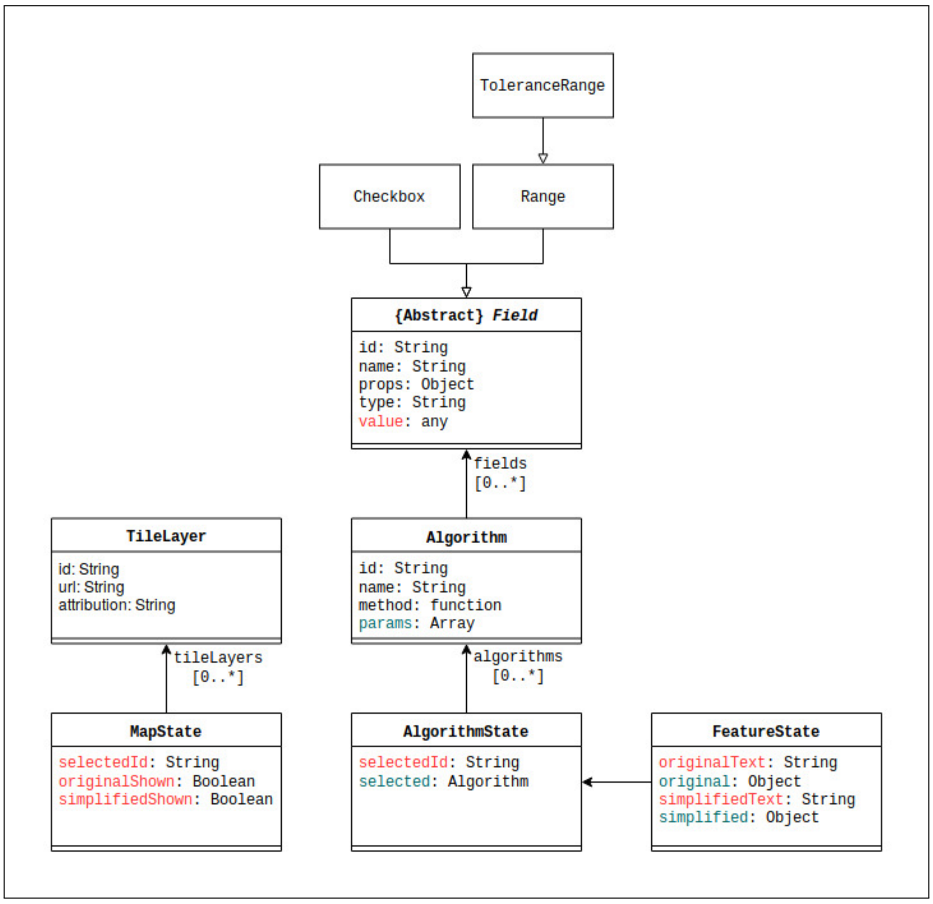


Figure 2: The state model of the application

in the *Nth point* algorithm, would instantiate a range field with a step value of one. The `ToleranceRange` however, which is modeled as its own subclass due to its frequent usage, allows for smaller steps to represent decimal numbers.

FeatureState encapsulates the state of the vector features. Each layer is represented in text form and object format of the GeoJSON standard. The text form is needed as a serializable form for detecting whether the map display needs to update on an action. As the original features come from file or the server, the text representation is the source of truth and the object format derives from it. The simplified features are asynchronously calculated. This process is outsourced to a debounced reaction that updates the state upon finish.

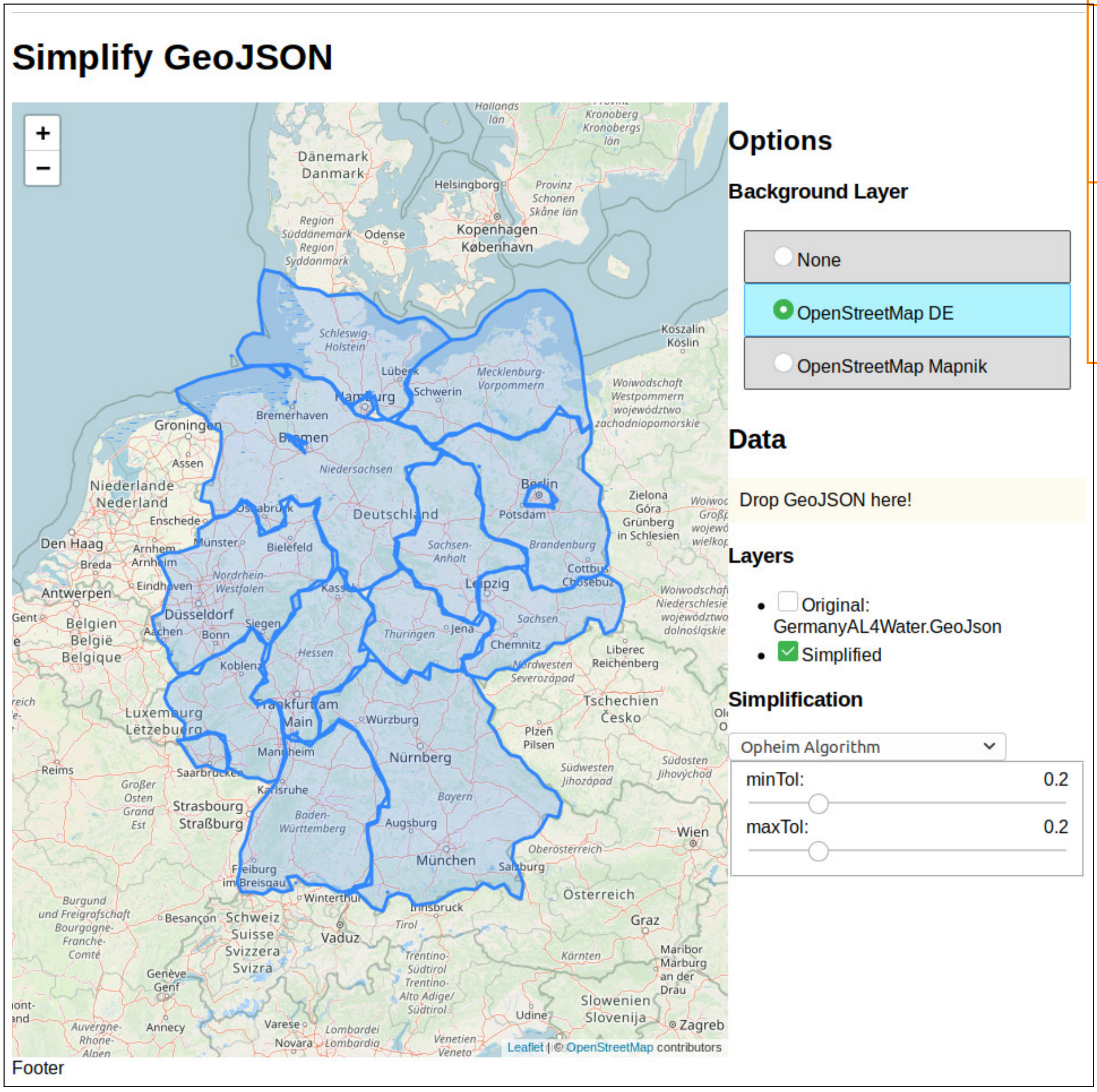
6.4 The user interface

After explaining the state model the User Interface (UI) shall be explained. The interface is implemented in components which are modeled in a shallow hierarchy. They represent and update the application state. In listing 3 the resulting web page is shown. The labeled regions correspond to the components. Their behavior will be explained in the following.

Insert final picture.

Red boxes around regions

Make ui fit description


Simplify GeoJSON

Options

Background Layer

None
[✓ OpenStreetMap DE]
OpenStreetMap Mapnik

Data

Drop GeoJSON here!

Layers

- ☐ Original: GermanyAL4Water.GeoJson
- ✓ Simplified

Simplification

Opheim Algorithm

minTol: 0.2

maxTol: 0.2

Footer

Figure 3: The user interface for the algorithm comparison.

Leaflet Map The big region on the left marks the Leaflet map. Its main use is the visualization of Features. The layers to show are one background tile layer, the original and the simplified features. Original marks the user specified input features for simplification. These are marked in blue with a thin border. The simplified features are laid on top in a red styling. Aside from the default control for zooming on the top left the map contains a info widget showing the length of the currently specified tolerance on the top right.

Background Layers Control The first component in the Options panel is a simple radio button group for choosing the background layer of the map or none at all. They are provided by the OpenStreetMap (OSM) foundation[17]. By experience the layer "OpenStreetMap DE" provides better loading times in Germany. "OpenStreetMap Mapnik" is considered the standard OSM tile layer[18].

Data Selection Here the input layer can be specified. Either by choosing one of the prepared data sets or by selecting a locally stored GeoJSON file. The prepared data will be loaded from the server upon selection by an Ajax call. Ajax stands for asynchronous JavaScript and XML and describes the method of dispatching an HTTP request from the web application without reloading the page. This way not all of the data has to be loaded on initial page load. On the other hand the user can select a file with an HTML input or via drag & drop. For the latter the external package "file-drop-element" is used[19]. It is a custom element based on the rather recent Custom Elements specification[20]. It allows the creation of new HTML elements. In this case it is an element called "file-drop" that encapsulates the drag & drop logic and provides a simple interface using attributes and events. Listing 10 shows the use of the element. The mime type is restricted by the `accept` attribute to GeoJSON files.

```html
<file-drop accept="application/geo+json">Drop area</file-drop>
```

Listing 10: The file-drop element in use

Layer Control This element serves the purpose of toggling the display of the vector layers. The original and the simplified features can be independently displayed or be hidden. If features have been loaded, the filename will be shown here.

Simplification Control The last element in this section is the control for the simplification parameters. At first the user can choose if a conversion to TopoJSON should be performed before simplification. Then the algorithm itself can be selected. The parameters change to fit the requirements of the algorithm. The update of one of the parameters trigger live changes in the application state so the user can get direct feedback how the changes affect the geometries.

[17] `https://wiki.osmfoundation.org/wiki/Main_Page`
[18] `https://wiki.openstreetmap.org/wiki/Featured_tile_layers`
[19] `https://github.com/GoogleChromeLabs/file-drop#readme`
[20] `https://w3c.github.io/webcomponents/spec/custom/`

6.5 Benchmark results

6.6 Comparing the results of different algorithms

7 Conclusion

Enhancement: Line Smoothing as preprocessing step

Listings