Master Thesis

Subtitle

Alfred Melch

A thesis presented for the degree of
Master of Science

Department Name
University Name
Country
Date

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

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. It was extracted from Leaflet, the "leading open-source JavaScript library for mobile-friendly interactive maps". 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.

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 Polylines in nested array form. Luckily since the library is small and written in javascript any skilled webdeveloper 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 a performance 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 as it is seen in listing. 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 form 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 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/`

```
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

`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 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 wasm.

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 entrypoint is not the `main`-function but a function called simplify. This is specified to the compiler as can be seen in `lib/simplify-wasm/Makefile`. Furthermore the functions malloc and free from the standard library are made available for the host environment. Compling the code through emscripten produces a wasm file and the glue code in javascript format. These files are called simplify.wasm and simplify.js respectively. An example usage can be seen in `lib/simplify-wasm/example.html`. Even through 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` the a further abstraction to the emscripten emitted

code was realised. The exported function `Simplify.wasm` handles module instantiation, memory access and the correct call to the exported wasm code. 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.

```
1  export async function simplifyWasm(coords, tolerance,
       highestQuality) {
2    const module = await getModule()
3    const buffer = storeCoords(module, coords)
4    const resultInfo = module._simplify(
5      buffer,
6      coords.length * 2,
7      tolerance,
8      highestQuality
9    )
10   module._free(buffer)
11   return loadResultAndFreeMemory(module, resultInfo)
12 }
```

../lib/simplify–wasm/index.js

Listing 5.2 shows the function `Simplify.wasm`. Further explanaition will follow regarding the functions `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`[1] was written to turn the module factory into a Javascript Promise that resolves on finished compilation. The result from this promise can be cached in a global variable. The usage of this function can be seen in listing 3.

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

Listing 3: My Caption

Next clarification is provided about how coordinates will be passed to this module and how the result is returned. Emscripten offers multiple views on the module memory. These correspond to the available WebAssembly datatypes (e.g. HEAP8,

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

HEAPU8, HEAPF32, HEAPF64, ...)[2]. As Javascript numbers are always represented as a double-precision 64-bit binary[3] (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 4 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 4: The storeCoords function

Check for coords length ¡ 2

Now we dive int C-land. Listing 5 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 n 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

Listing 6 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 appro-

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

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

```
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 5: Entrypoint in the C-file

priate 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 6: 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 user input 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 a full application it will be possible to test a variety of use cases on multiple end-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.

The dynamic aspects of the web page will be built in javascript to make it run in the browser. Webpack[4] will be used to bundle the application code and use compilers like babel[5] 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, which not a requirement in this case. By refraining from this practice there will also be no unintentional impact on the application performance. Libraries in use are Benchmark.js[6] for statistically significant benchmarking results, React[7] for the building the user interface and Chart.js[8] for drawing graphs.

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.

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 5.3 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.

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`.

[4] https://webpack.js.org/
[5] https://babeljs.io/
[6] https://benchmarkjs.com/
[7] https://reactjs.org/
[8] https://www.chartjs.org/

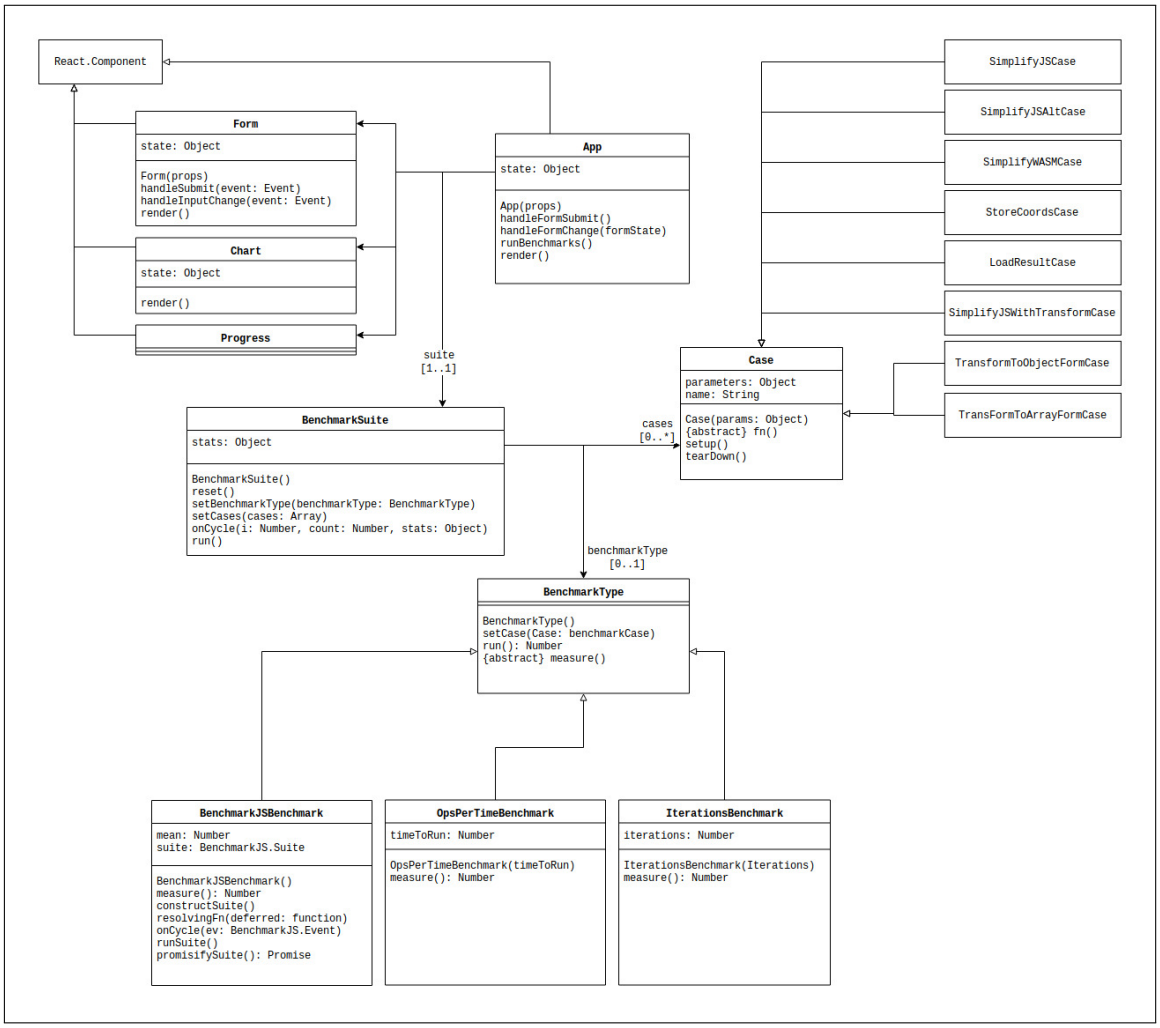

Figure 1: UML diagram of the benchmarking application

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.

- **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.

BenchmarkType

BenchmarkSuite

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

6.1 State of the art: psimpl

6.2 Compiling to webassembly

6.2.1 Introduction to emscripten

6.3 Preserving topology GeoJSON vs TopoJSON

object form vs array form

6.4 The implementation

7 Results

7.1 Benchmark results

7.2 Comparing the results of different algorithms

8 Conclusion

Enhancement: Line Smoothing as preprocessing step

Listings