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 simplifyJS. 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 webpage is built to produce benchmarking insights to compare the two approaches performance wise.

5.1 State of the art: simplifyJS

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 has currently 20,066 weekly downloads while the Turf.js derivate @turf/simplify has 30,389.
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 lisencing 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`.

```
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

```
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

Just like the simplify-js library the webassembly solution requires the data to be transformed for processing. Meant with that is the storing and loading of bytes into and from the module heap. This transformations however are intensive ones and not as easy to overcome. In a larger project the data may already be managed in a webassembly module. So the raw execution time might be relevant as well. To make assumptions about the real-world usage of WebAssembly in this case there will be seperate measurements for storing and loading of data and the execution.

5.3 The implementation

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