https://www.utpjournals.press/doi/pdf/10.3138/cart.54.4.2018-0014 - Gábor Farkas <gfarkas@gamma.ttk.pte.hu> - Thursday, December 05, 2019 10:38:24 AM - IP Address:84.1.165.145

ARTICLE

Hardware-Accelerating 2D Web Maps: A Case Study

Gábor Farkas
Department of Cartography and Geoinformatics / University of Pécs / Pécs / Hungary

ABSTRACT

Recent trends show that developers behind some of the most popular web mapping libraries put excessive work into creating custom hardware-accelerated rendering engines. Other libraries focus on functionality rather than visualization. From the perspective of the developer using these libraries an important question arises: is it necessary to use a WebGL-powered library for 2D web mapping? The answer was found through the implementation and evaluation of a simple WebGL renderer for the open source Web mapping library OpenLayers. It extends the previous, texture-based implementation with line-string, polygon, and label-rendering capabilities. Through various benchmarks, the benefits of using a WebGL rendering engine over the traditional, but nowadays widely supported and – in most cases – hardware-accelerated HTML5 Canvas renderer are assessed. Contrary to the current trends in Web mapping, results suggest that using the Canvas Application Programming Interface (API) is sufficient for smaller Web maps (up to around 2000 features and 60,000 vertices) using static vector data. WebGL only gives a noticeable performance boost with maps using large vector layers, such as Web GIS clients.

Keywords: GIS, Web GIS, Web development, software development, cartography, data visualization

RÉSUMÉ

Les tendances récentes indiquent que les concepteurs de certaines des bibliothèques de cartographie interactive les plus populaires consacrent énormément d'efforts à la création de moteurs de rendu personnalisé accéléré par matériel. D'autres bibliothèques sont axées sur la fonctionnalité plutôt que la visualisation. Pour le concepteur qui utilise ces bibliothèques, une importante question se pose : est-il nécessaire de recourir à une bibliothèque alimentée par WebGL pour la cartographie Web en 2D ? La réponse est issue de la mise en application et de l'évaluation d'un simple moteur de rendu WebGL pour la bibliothèque de cartographie Web à code source libre OpenLayers. Elle élargit la mise en application antérieure basée sur la texture au moyen de capacités de rendu en cordes linéaires, polygones et étiquettes. Au moyen de divers points de repère sont analysés les avantages de l'utilisation d'un moteur de rendu WebGL par rapport au rendu traditionnel, mais bénéficiant aujourd'hui d'un large soutien et — dans la plupart des cas — accéléré par matériel HTML5 Canvas. Contrairement aux tendances actuellement observées en cartographie Web, les résultats semblent indiquer que l'utilisation de l'interface de programmation d'application (API) Canvas est suffisante pour les cartes Web plus petites (d'un maximum d'environ 2 000 caractéristiques et 60 000 points) faisant appel aux données vectorielles statiques. Le WebGL ne permet une augmentation notable de la performance que dans le cas des cartes utilisant de grandes couches de vecteurs, comme celui des clients du SIG Web.

Mots clés : cartographie, développement de logiciels, développement Web, SIG, SIG Web, visualisation de données

Introduction

With the spread of hardware-accelerated technologies on the Web, numerous geospatial data visualization libraries using them have been developed. Of those technologies, more and more projects utilize a recent one called WebGL. WebGL is a subset of the Open Graphics Library (OpenGL) API, which allows developers to write programs executed on the GPU. This technology is often associated with 3D graphics, and some of the libraries (e.g., Cesium, NASA Web World Wind) were indeed created to show spatial phenomena on a 3D virtual globe. However, as proven by Google Maps (Nogueira 2012), WebGL not only is a tool for creating 3D visualizations in the geospatial industry, but also can significantly enhance user experience in 2D Web maps (Lienert and others 2012).

According to recent trends in Web mapping, harnessing the capabilities of this technology will be inevitable for Web mapping libraries striving to give the best user experience. For instance, one of the latest trends inf Web mapping is using vector tiles (Antoniou, Morley, and Haklay 2009). These are a well-recognized (Springmeyer 2015)

https://www.utpjournals.press/doi/pdf/10.3138/cart.54.4.2018-0014 - Gábor Farkas <gfarkas@gamma.ttk.pte.hu> - Thursday, December 05, 2019 10:38:24 AM - IP Address:84.1.165.145

specification for using vector data instead of traditional raster base maps. This technology opens a variety of possibilities for creating richer and more responsive Web maps (Antoniou et al. 2009; Agafonkin 2016), however, CPU-based rendering is not feasible for creating representation models from such amount of data. This only rules out early Web 2.0 Web mapping libraries that manipulate browsers' native document object model (DOM) – which is used for rendering Web page elements – to create dynamic Web maps. As DOM is not as scalable as hardware-accelerated rendering, it limits the usability of the library and prevents it from becoming the basis for a universal Web GIS system.

Using WebGL in 2D Web mapping libraries is becoming a common solution. Some proprietary libraries, such as Google Maps (Nogueira 2012) and ArcGIS API for JavaScript, have full implementations of WebGL rendering. In the open source sector, Mapbox GL JS, Tangram, and deck.gl are recent libraries with complete WebGL rendering engines, while OpenLayers (formerly OpenLayers 3) has offered partial support since its first release. Other popular libraries capable of Web mapping (Farkas 2017) utilize only CPU-based rendering (i.e., DOM), with some exceptions using another hardware-accelerated technology: the HTML5 Canvas API.

The Canvas API has better compatibility than WebGL, as its hardware-accelerated nature is optional. If a browser or the underlying hardware does not allow the Canvas API to use the GPU, it simply changes to a CPU-based rendering mode. By design, it is a programmatic drawing surface, providing a 2D context, which can be manipulated using its methods (Mowery and Shacham 2012). Consequently, it offers a convenient API and fast rendering, although the rendering pipeline cannot be customized as thoroughly as in the case of the WebGL API.

While it seems trivial to use a WebGL-powered library in every Web mapping application, there are two drawbacks to doing so. The trivial one is supporting legacy browsers incapable of using WebGL, while the more substantial one can be found in the capabilities offered. Recent open source libraries have great rendering capabilities, while older libraries have more GIS features. While choosing the best stack for an application, one may wonder if performance should be chosen over features, or if a hardware-accelerated Canvas renderer is enough for 2D Web maps. In this article, the benefits of using a 2D WebGL map renderer are assessed with the implementation and benchmarking of a WebGL-based vector rendering engine, which can be used for 2D geospatial data visualization.

A Rendering Overview

As Internet mapping is almost as old as the Internet (Farkas 2017), the first Web-based map viewers relied on early Web standards. These DOM- and scalable vector graphics (SVG)-based Web applications (Peng and Zhang 2004) later evolved into the dynamic web mapping libraries of the early Web 2.0 era. These libraries (Farkas 2017) could create representation models based on raw vector data, images, and image tiles. With the rapid development of Web technologies, they became more capable; however, their rendering mechanism did not change.

The possibilities of native (i.e., not plugin-based) Web mapping libraries with next generation rendering engines came with the HTML5 standard's <canvas> element and the API bound to it. It was capable of pixel manipulation with unparalleled speed; however, it was not as efficient as SVG with vector graphics (Sauerwein 2010; Jayathilake and others 2011). On the other hand, it was not its performance that made it appealing as a rendering engine for Web maps, but its visualization capabilities, which could not be achieved with traditional DOM-based rendering (Jayathilake and others 2011). Later, as browsers became capable of hardware-accelerating traditional 2D canvas contexts (i.e., not WebGL contexts), software using the Canvas API became faster than traditional DOM rendering engines. A very interesting, nontrivial topic, on the other hand, is the degree of hardware acceleration in 2D canvas contexts. While the Canvas API is standardized, the underlying renderer can vary between browser engines. This makes them act like a black box from the perspective of Web developers. Nevertheless, most browser engines are open source. On the other hand, it can be assumed that every browser engine strives for the best performance when it decides if a step should be hardware-accelerated in the rendering pipeline.

Although directly comparing different rendering engines for assessing Web technologies (Figure 1) would be scientifically questionable, running some basic measurements could still give some interesting insight into current technology. While Web mapping libraries are sensitive to the number of features, they can handle a single feature with numerous vertices fairly well (especially using an SVG path element). On the other hand, it does not matter if a DOM or Canvas-based renderer is used; rendering larger datasets – which can easily hold thousands of features in practice – decreases the performance of the libraries rapidly. To accurately measure the benefits of a WebGL-based 2D map renderer over using the Canvas API, a capable library had to be chosen that has full Canvas support and can be extended with a WebGL rendering engine. For the basis of the analysis, OpenLayers seemed to be the ideal candidate, as it was created with WebGL in mind, and it was already capable of rendering images and vector points with WebGL, besides its adequate Canvas renderer.

Since OpenLayers initially offered three different rendering engines (the DOM renderer has been depreciated), it was built with the necessary abstractions for implementing a comparable WebGL renderer. From the abstract classes the renderers can not only inherit, but also define

https://www.utpjournals.press/doi/pdf/10.3138/cart.54.4.2018-0014 - Gábor Farkas <gfarkas@gamma.ttk.pte.hu> - Thursday, December 05, 2019 10:38:24 AM - IP Address:84.1.165.145

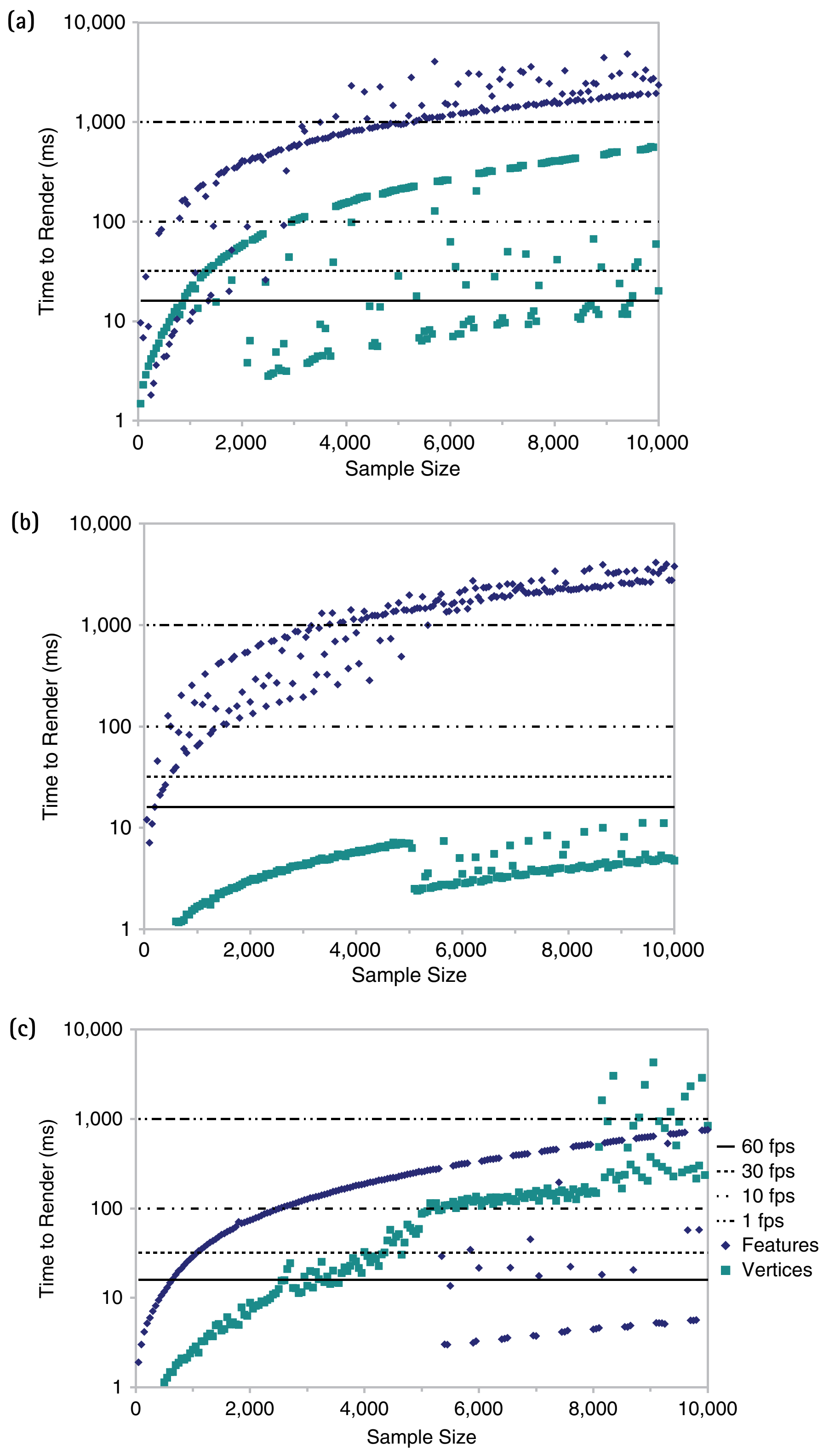

Figure 1. Benchmarks of rendering layers having pseudo-random line string geometries with different numbers of features and vertices using three popular Web mapping libraries: (a) Leaflet with Canvas, (b) OpenLayers 2 with DOM, and (c) OpenLayers with Canvas
Note: Measured with Google Chrome utilizing hardware-accelerated Canvas. In the case of vertices, there is only a single feature, while in the case of features, every geometry consists of five vertices. The rendering process was measured with a tool developed for this purpose (Farkas 2018b), which uses Benchmark.js for every test.

https://www.utpjournals.press/doi/pdf/10.3138/cart.54.4.2018-0014 - Gábor Farkas <gfarkas@gamma.ttk.pte.hu> - Thursday, December 05, 2019 10:38:24 AM - IP Address:84.1.165.145

their own ways of managing the rendering pipeline of the given context (see Figure A1). They can use features processed by the library and implement their rendering logic. In OpenLayers, vector layers use render classes to draw different types of geometries. There are two different vector rendering methods: immediate and replay. Replay is the main method for rendering geometries. Instead of rendering the whole visible extent in every frame, it tries to optimize the rendering calls; therefore it caches everything it can. When it comes to rendering a frame, function calls are done in two phases: preparation and composition. Preparation includes processing styles and raw coordinates (e.g., relative to eye correction; Cozzi and Ring 2011) and then creating a drawable – and mostly cacheable – model. Functions in the composition phase take this model and draw it on the map canvas. While in drawing a frame both of the phases are executed only once, during an animation (e.g., panning or zooming), this model can be used to speed up rendering by preparing only once, and composing the prepared data in fast successions.

Implementation

As OpenLayers already had a minimal WebGL template, it could easily be extended. The main difficulty was writing the actual rendering routines (e.g., line and polygon triangulating algorithms). As Web applications use JavaScript code on the client side, there are fewer rendering utility libraries to choose from, while they are also less matured and recognized than their desktop counterparts (e.g., GLU, cairo, freetype). Moreover, as JavaScript is an interpreted language, which is farther from the hardware than low-level languages (Crockford 2008), performance is more of an issue than in the case of the aforementioned C and C++ libraries (Di Benedetto and others 2010). This increases the need for customized implementations instead of using general libraries with some overhead. At the time of the implementation, three different rendering modules were needed to assess the most important rendered features of a Web map: a line, a polygon, and a text renderer.

RENDERING LINES

Triangulating line strings is a common and trivial task, where one only has to find the lines parallel to the original one at a fixed distance (i.e., the buffered lines). For the sake of simplicity, and to make the algorithm as fast as possible, the whole triangulation was written to run on the GPU. The triangulation can be parallelized by providing three consecutive pairs of coordinates (two in the case of endpoints), p_0, p_1, and p_2, a direction, d, and an instruction, i, for every vertex in the line string. While only the most important part of the algorithm is presented (Algorithm 1), the full implementation can be found in the OpenLayers code base (OpenLayers Contributors 2019).[1]

Algorithm 1. Algorithm for offsetting a single point of line string during triangulation

```
Require: $p_0, p_1, p_2, d, i$
 1: $w \leftarrow$ width                    ▷ Provided as a uniform variable
 2: $u \leftarrow p_1 - p_0$
 3: $v \leftarrow p_2 - p_1$
 4: if $i = simple$ then   ▷ Offset $p_1$ along rotated $\hat{u}$ (or $\hat{v}$)
 5:     $p_1 \leftarrow p_1 + w \div 2 \times (-\hat{u}_y, \hat{u}_x) \times d$
 6: else if $i = miter$ then
 7:     $k \leftarrow \hat{u} + \hat{v}$
 8:     $l \leftarrow w \div 2 \div \hat{k} \cdot (-\hat{v}_y, \hat{v}_x)$
 9:     $p_1 \leftarrow p_1 + (-\hat{k}_y, \hat{k}_x) \times l \times d$
10: end if
```

Besides the core functionality of offsetting lines in two directions, the renderer can collapse sharp line joins (miters) into flat line joins (bevels) or round them. It can also create the three most popular line endings in vector graphics: butt, square, and rounded. As it is a very simple implementation, it has some limitations. For example, it can only use a single colour for a line string (no gradients or textures) and it cannot create dashed lines. Furthermore, sharp line joins are also handled (intersecting triangles, artifacts produced by too long miters). While the implementation is not prone to producing artifacts, some can be observed, especially with old, integrated graphics cards. Finally, anti-aliasing was not implemented due to OpenLayers' choice of relying on browsers' native multisample anti-aliasing technique (Liktor and Dachsbacher 2013). Overall, in most cases, the implementation creates visually adequate results, comparable with those for the Canvas renderer.

RENDERING POLYGONS

Since WebGL can handle the three basic primitive types (points, lines, and triangles), the challenge in a polygon renderer is writing an efficient triangulation algorithm, which can slice up the polygon for the GPU. There are dozens of universal desktop implementations for this problem with respect to the vector graphic libraries (e.g., cairo) and GUI frameworks (e.g., Qt). Although the theoretical background of such triangulation algorithms (Meisters 1975; Seidel 1991; Shewchuk 1996) is the same as for the Web, there are only a few JavaScript libraries to choose from. From those libraries, the ones capable of triangulating complex and topologically incorrect polygons that are also under constant maintenance or development are Mapbox's earcut and delaunator and libtess.

Of those libraries, libtess is a direct port of OpenGL Utility Library's (GLU) tessellator function. Consequently, it can be considered the most matured, most stable, and most

https://www.utpjournals.press/doi/pdf/10.3138/cart.54.4.2018-0014 - Gábor Farkas <gfarkas@gamma.ttk.pte.hu> - Thursday, December 05, 2019 10:38:24 AM - IP Address:84.1.165.145

universal approach to creating meshes using the trapezoidal decomposition (Seidel 1991) technique. However, according to Mapbox's claim (Mapbox 2015), it also performs rather poorly. Of the other two libraries, one is a Delaunay triangulation algorithm, which creates the best-quality tessellations and has worse performance than ear clipping (Held 2001). As the implementation's sole purpose is rendering polygons, performance is more important than maximizing the polygons' internal angles, which makes earcut an obvious choice. The earcut library uses an ear clipping technique adapted from Martin Held's fast industrial-strength triangulation (FIST) algorithm (Held 2001). Its only downside is it does not support self-intersecting polygons, which are common in real-life GIS datasets, and should be supported by universal Web mapping libraries. To overcome this problem, but also take advantage of the ear clipping technique's performance, another FIST adaptation was implemented, which handles self-intersecting polygons by inserting Steiner points at self-intersections. Similarly to earcut, it does not guarantee correct triangulation of very complex geometries; however, it was designed to be reliable with real-life data. As having the best performance with topologically correct polygons is crucial, it has a penalizing structure (Algorithm 2); it has a fast mode for simple polygons (e.g., it does not check for intersections), while it starts to use more time-consuming methods as the deficiencies of the processed polygon add up.

Algorithm 2. Pseudo-code of the triangulating algorithm

Require: Polygon P with each vertex classified as convex or reflex based on convexity with respect to the neighboring vertices

```
1:  procedure TRIANGULATE($P$)
2:  while $|P| > 3$ do
3:      if $P$ is simple then
4:          if there are valid ears in $P$ then
5:              $T \leftarrow$ Clipped ears using fast rules
6:          else if $P$ can be reclassified then
7:              $P \leftarrow$ Reclassified $P$
8:          else ▷ $P$ can have touching segments
9:              $P \leftarrow P$ with resolved self-intersections
10:         end if
11:     else
12:         if there are valid ears in $P$ then
13:             $T \leftarrow$ Clipped ears using precise rules
14:         else if $P$ can be reclassified then
15:             $P \leftarrow$ Reclassified $P$
16:         else if There are self-intersections then
17:             $P \leftarrow P$ with resolved self-intersections
18:         else if $P$ is simple again then
19:             $P \leftarrow P$ with inverted orientation
20:         else
21:             Split $P$ into two parts and triangulate
22:         end if
23:     end if
24: end while
25: $T \leftarrow$ T + last ear in $P$
26: end procedure
```

For fast traversal across the polygon's segments, the implementation creates a linked-list structure and fills it with the polygon's segments, holding references to the classified vertices. The other data structure for speeding up intersection checks is an R-Tree: a bounding volume hierarchy (BVH) storing the axis-oriented bounding boxes of the geometries. The ear clipping procedure also adapts the FIST algorithm (Held 2001); thus, three consecutive vertices ($v1v2v3$) of polygon P form an ear if and only if

- P is simple
- v_2 is convex
- there are no points of P in $\Delta v_1 v_2 v_3$ (except for v_1, v_2, v_3)

or

- v_2 is convex
- segment $v_1 v_3$ does not intersect with any segments of P (except in v_1, v_3)
- segment $v_1 v_3$ is completely inside P

With the most important rules emphasized above, the rest of the algorithm matches FIST (Held 2001). While the full code is too tedious to show in this paper, similarly to the line renderer, it can be found in the OpenLayers code base. As the most crucial problems have been fixed since the polygon renderer's first release, and as it uses the line renderer to draw outlines, it produces decent visual output in most of the cases. However, it is important to note that, as can be seen in the demo application, there are still some unresolved problems. Notably, colours are mixed at the lowest zoom levels on every machine, and vertices are mixed up on the smartphone used for benchmarking. On the other hand, as the process did not produce any JavaScript or WebGL errors, it was not considered an issue for measuring performance.

RENDERING LABELS AND IMAGES

The final part of the rendering engine was the text renderer. Adapting to computers' increased vulnerabilities on the Web, browsers' increased security prevents them accessing system files through JavaScript. This makes using advanced text rendering techniques harder, as every supported font need to be served. Consequently, the only general solution providing access to system fonts is making the browser serve them, which is an image-based approach. As text rendering is image-based, it

https://www.utpjournals.press/doi/pdf/10.3138/cart.54.4.2018-0014 - Gábor Farkas <gfarkas@gamma.ttk.pte.hu> - Thursday, December 05, 2019 10:38:24 AM - IP Address:84.1.165.145

was implemented on the top of the image renderer, the first and only part of the library's WebGL engine prior to this work. As the WebGL API can transform both regular images and canvas elements to textures, OpenLayers uses it for both rendering images generated by spatial servers (e.g., GeoServer) in a browser-friendly format (e.g., PNG, JPEG), and using pre-rendered markers on canvas elements. The label renderer extends this part of the rendering engine and uses internal canvas elements for storing labels, similarly to the point renderer, when the point symbolizer is a shape. To speed up drawing, it also utilizes the library's atlas manager implementation and uses it as a glyph atlas to store multiple characters on the same canvas.

One could argue that using the Canvas API in the WebGL rendering pipeline makes benchmarking unnecessary, as it will be always slower than the native Canvas approach. This is not entirely true. The Canvas API is only used for creating initial textures, which are cached and drawn by the GPU. Therefore, the WebGL engine can show a performance boost increasing with the number of repetitions. Since the implementation uses a glyph atlas, this is especially true for labels, as every character in a font type is generated only once.

Benchmarks

To explore the necessity of a WebGL rendering engine in creating 2D Web maps, the rendering times of different vector layers were benchmarked. The benchmarks were performed on a Dell Inspiron 7567 with an Intel Core i7-7700HQ CPU, 8 GB DDR4 RAM, an NVIDIA GeForce GTX 1050 Ti dedicated GPU (through the proprietary NVIDIA driver), an Intel HD Graphics 630 integrated GPU, and a 39.6-cm (15.6-inch) display with a resolution of 1920×1080 pixels. The browser hosting the test application was a 64-bit Chromium with hardware-accelerated Canvas running on a Debian 9 OS.

Due to the necessity of representing weaker, more common devices, the same benchmarks were also carried out on a Lenovo A536 smartphone. The device has Quad-core 1.3 GHz Cortex-A7 CPU, 1 GB RAM, an ARM Mali-400 MP2 GPU, and a 12.7-cm (5-inch) display with a resolution of 480×854 pixels. The browser used on the smartphone was a 32-bit Chrome running on Android 4.4.2, while the measurements were conducted via remote debugging. The phone also has proper hardware-accelerated Canvas and WebGL support.

While multiple browsers were considered to run these benchmarks on, Chromium was chosen as the only platform. It was deemed unfeasible to run the tests on multiple browsers, as the additional results would have added little value compared with the number of excess data. From the more popular browsers, Chromium was chosen for its superior developer tools.

METHODS

The benchmarked application (Farkas 2018a; Figure 2) was developed to represent both Web cartography and GIS; therefore it has two sets of layers (groups). Both groups consist of statically loaded GeoJSON layers; therefore there

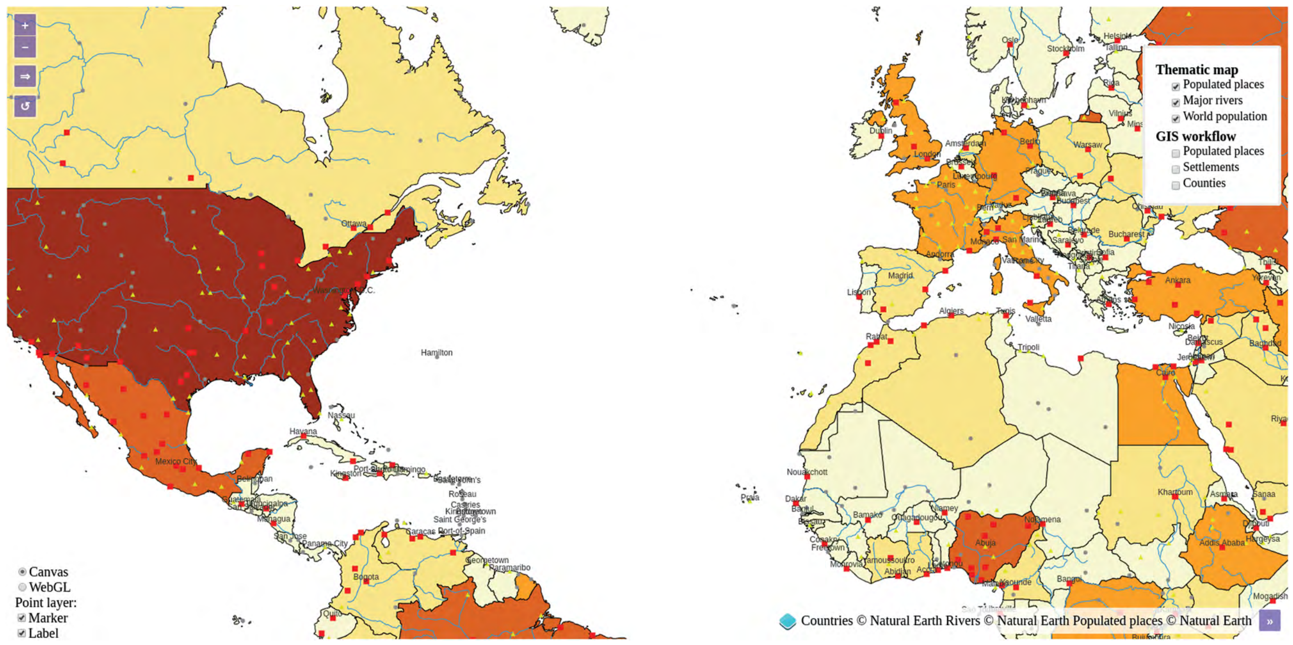

Figure 2. The benchmarked Web map showing its thematic layer group
Source: Countries, rivers, and populated places © Natural Earth.

https://www.utpjournals.press/doi/pdf/10.3138/cart.54.4.2018-0014 - Gábor Farkas <gfarkas@gamma.ttk.pte.hu> - Thursday, December 05, 2019 10:38:24 AM - IP Address:84.1.165.145

was no loading time during measurements. Furthermore, the initial loading time was not included in any benchmarks, as rendering time was the main focus of the study.

The first group contains thematic layers showing the population of the world, using freely available data from the Natural Earth 1:50,000,000 dataset. It is a composition following cartographic guidelines to measure a real-world use case. It has a choropleth base map, major rivers as simple lines, and a thematic point layer. The latter shows different marker symbols with settlements in different population intervals, while it also shows labels for capital cities. The rationale behind this group is drawing manageable amount, but styled vector data.

The GIS layer group does not have custom symbology; however, it consists of layers produced by a basic GIS workflow, which thus contain gradually more features and vertices than the thematic group (Table 1). The example workflow uses freely available data to calculate the population density of each county in Hungary. It uses OpenStreetMap (OSM) data for settlement and county polygons and an extract of the Free Gazetteer Data product of GeoNames, which contains – amongst a lot of additional inseparable features – a point for each settlement with its population. Assuming that both of the layers contain every settlement, and there is a common column in them (e.g., name), the task can be achieved by taking the following steps:

(1) Join the attributes of the OSM settlement layer to the GeoNames layer using the common field.
(2) Extract settlements from the GeoNames layer by selecting features where the common field is not null.
(3) Aggregate populations of the settlements residing in each zone using the features of the OSM county layer as zones.
(4) Calculate population density from the aggregated population data and the areas of county polygons.

By following these steps, one can have at most three different layers rendered at a time: a point layer containing populated places and two polygon layers of different administrative boundaries.

Table 1. Feature and vertex counts of the benchmarked layers

	Type	Features	Vertices
Thematic group			
Countries	Polygon	241	99,566
Rivers	LineString	460	25,629
Towns	Point	1249	1,249
Capitals	Label	200	200
GIS group			
Settlements	Polygon	3170	428,803
Counties	Polygon	20	61,770
Places	Point	9839	9,839

As the thematic group has all the different vector types a Web mapping library must be able to render, the performance of each component could be measured individually, and the whole group as a composition. Additionally, the GIS group was measured to assess the need of a WebGL rendering engine for a smaller GIS workflow in a Web-based environment.

Perceived performance has other factors besides the raw metrics of the layers (Table 1) in OpenLayers, namely the generalization algorithms on smaller scales and the BVH used for filtering out features outside the rendered extent on larger scales. Consequently, the benchmark measured the rendering engines' performance on the two layer stacks using different scales starting from the largest one enclosing the whole group. This made it possible to measure the direct benefits of choosing a WebGL-powered Web mapping library over one using the Canvas API in different circumstances.

Chromium's Developer Tools and the Performance Timeline API (Grigorik, Mann, and Wang 2016) were used to benchmark the application in different circumstances. From the different builds (high-end: NVIDIA, mid-class: Intel, low-end: ARM Mali), the appropriate ones were always used. For example, the thematic group was measured with all of the builds. On the other hand, as a smartphone is not expected to handle GIS workloads, the GIS group was measured with the Intel and NVIDIA builds only.

Every result was averaged from at least 10 successive measurements. This, however, greatly depended on the measurement's type and the variance of data points. For example, redraws produced a fixed number of data points every time, while the numbers of values from animations were functions of frame rate. Continuous measurements were executed with manual filtering of outliers until the mean converged to a value. The minimal 10 data points were only enough for a few cases; the typical number of measurements behind a result was around 70–100.

THEMATIC GROUP

In the first phase, frame rates of the two renderers were measured at different zoom levels (Table 2). Besides benchmarking the drawing performance of the library, the animating speed was also assessed, as animation frames are more frequent than complete redraws. The maximum observed performance with both the WebGL and the Canvas renderer, and with both the Intel and the NVIDIA GPU, was about 60 frames per second (fps). As some of the measurements reached this frame rate – and with 60-Hz displays it can be considered optimal – cases reaching this frame rate were not investigated further. Furthermore, as Intel and NVIDIA builds had similar tendencies, but the NVIDIA GPU is strong enough to smooth out some of

https://www.utpjournals.press/doi/pdf/10.3138/cart.54.4.2018-0014 - Gábor Farkas <gfarkas@gamma.ttk.pte.hu> - Thursday, December 05, 2019 10:38:24 AM - IP Address:84.1.165.145

Table 2. Metrics and properties of the benchmarked maps

Zoom level	Scale[a]	Centre[b]	Features	Vertices
Thematic group				
1	1:295,829,355	0; 0	2150	37,753
2	1:147,914,678	0; 0	2150	57,284
3	1:73,957,339	0; 2,200,000	2118	78,940
4	1:36,978,669	7,300,000; 5,500,000	1210	60,882
5	1:18,489,335	3,400,000; 6,200,000	513	43,820
GIS group				
8	1:2,311,167	2,200,000; 5,970,000	13,029	243,724
9	1:1,155,583	2,100,000; 5,970,000	10,458	279,139
10	1:577,792	2,050,000; 5,900,000	3,541	128,998
11	1:288,896	1,980,000; 5,920,000	1,144	55,362
12	1:144,448	1,980,000; 5,920,000	359	22,097

[a] Scales were calculated from the laptop screen's DPI value and the map's resolution.
[b] Centres are coordinates projected using the Web Mercator projection (EPSG:3856).

the differences, only the Intel, and the ARM Mali results are presented in more detail. Finally, triangulation was disabled for these measurements. The reason behind this decision is the nature of the triangulation algorithm. It was found that the whole process takes most of the rendering time, while it does not need to run so frequently. The triangulated mesh can be cached, and only recalculated when the given polygon is modified.

The thematic composition (Table 3) was handled by both of the renderers acceptably, but only with the two stronger builds. Since animation frames use cached data, drawing is generally slower than animating. The increment in animating performance is much higher in the WebGL engine, although it is also high enough in the Canvas engine to make overall performance tolerable (>16 fps) in the case of layers with up to 60,000 vertices. In drawing performance, the Canvas renderer noticeably outperformed the WebGL implementation, mostly due to the high polygon count of the country layer. While the WebGL renderer's drawing function caused the application to lag, as it went far below 16 fps, a threshold value for perceiving subsequent frames as a continuous animation (Mengeringhausen and Witherell 1962), it still gave a better overall experience. During panning the map it was fast and responsive. On the other hand, the Canvas engine gave a more balanced performance, with slight lags at the third zoom level. However, during animations, this lag could only be observed on the integrated GPU.

By investigating the results further (Figure 3), it was found that both of the renderers' drawing performance follows the number of vertices very closely. Additionally, the number of vertices is mostly influenced by the generalization algorithm and the BVH used by the library. At the first three zoom levels, the generalization algorithm allows more vertices to be seen, while at the last two, the BVH filters out numerous features. As the polygon layer contains about 10 times the line layer's vertices, and about one hundred times

Table 3. Rendering performance (fps) of the thematic layer group

	Zoom level				
	1	2	3	4	5
NVIDIA GPU					
WebGL animating	56.35	56.82	57.53	56.00	54.19
Canvas animating	29.24	23.64	22.45	32.59	46.37
WebGL drawing	14.75	10.92	10.26	12.18	15.37
Canvas drawing	22.94	18.59	14.23	24.33	33.00
Intel GPU					
WebGL animating	58.99	55.34	52.59	59.01	59.72
Canvas animating	26.07	20.21	15.17	16.77	22.83
WebGL drawing	10.91	8.37	7.17	7.72	8.03
Canvas drawing	13.57	12.92	10.28	10.71	14.66
ARM Mali GPU					
WebGL animating	12.67	12.08	10.66	16.51	21.83
Canvas animating	1.19	1.22	1.87	3.69	5.84
WebGL drawing	1.25	0.94	0.82	1.45	2.27
Canvas drawing	1.13	0.96	1.32	2.51	4.05

Note: Lags are emphasized by underlining, and severe lags by framing.

https://www.utpjournals.press/doi/pdf/10.3138/cart.54.4.2018-0014 - Gábor Farkas <gfarkas@gamma.ttk.pte.hu> - Thursday, December 05, 2019 10:38:24 AM - IP Address:84.1.165.145

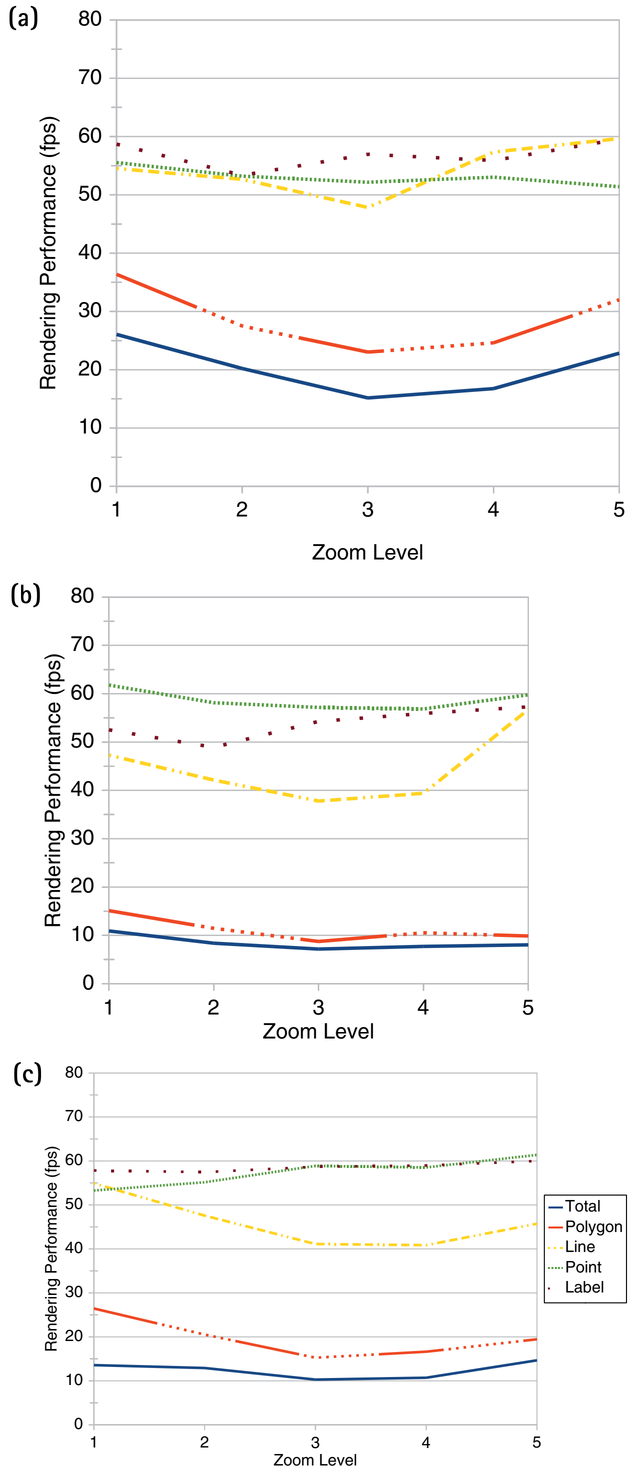

Figure 3. Detailed results of the thematic layer group's rendering tests on the Intel GPU: (a) Canvas animating performance, (b) WebGL drawing performance, and (c) Canvas drawing performance

the point and label layers' vertices at every scale, it is the main contributor to the weak drawing performance. The reason behind WebGL animation frames could give a stable near-60-fps performance, while Canvas animation frames produce noticeable lag is the amount of cached data. While the WebGL engine is capable of caching the entire buffer, the Canvas engine has limited means of optimization, as it cannot know if the geometries will change between animation frames. Moreover, low frame rates produced by processing several tens of thousands of vertices are not surprising, especially as the polygon renderer includes line rendering in both of the engines. Finally, as the WebGL line renderer is heavily GPU-based, a line layer with a few thousand vertices caused measurable performance drop in drawing lines, although it also affected the Canvas renderer.

Results from the ARM Mali GPU (Table 3) showed similar trends, but on a lower performance at a first glance. Most of the scenarios resulted in severe lags (< 5 fps). In case of this build, the WebGL engine could give a perceivable performance boost, although the experience was far from enjoyable due to the slow drawing performance on the first three zoom levels (i.e. unresponsive map between two pans). Detailed measurements (Figure 4) on the other hand show different trends. It seems like on weaker GPUs, polygons largely contribute to the overall performance. While numerous polygons with large amounts of vertices (33,459; 51,888; 71,578; 54,751; and 40,022 on the first five zoom levels) have a serious performance impact, both of the engines give a decent animating performance with only a few ten thousands of vertices. This cannot be told about drawing performance, where the maximum amount of polygon vertices resulting in a responsive map without lagging is lower by an order of magnitude than on the Intel GPU. However, this has a minor impact on user experience, as it can only be perceived as minor lagging between interactions.

GIS GROUP

By placing a heavy load on the rendering engines, additional trends could be identified. Rendering the GIS layer group (Table 4) with over one hundred thousand vertices at the first three zoom levels resulted in a switched trend in the two engines' performance. While drawing speed was slow with both of them, the naive WebGL implementation could overtake the hardware-accelerated Canvas engine when the vertex number was very high. Animating performance was still balanced in the Canvas renderer. On the other hand, the WebGL renderer showed a decline in animating with the integrated GPU, proving there is a practical limit to a smooth WebGL GIS experience on weaker video cards.

Since the GIS layer group has a large point layer (9839, 7878, 2575, 801, and 242 vertices at zoom levels 8, 9, 10, 11, and 12, respectively), this benchmark also gave the opportunity to assessing the impact of textures on rendering performance. Such numbers did not affect the WebGL renderer's animating functionality; the layer was animated at about 60 fps. The same cannot be said about the other assessed functions (Figure 5). While the overall performance was tolerable in most cases, the performance decline was still measurable. Lags were only experienced

https://www.utpjournals.press/doi/pdf/10.3138/cart.54.4.2018-0014 - Gábor Farkas <gfarkas@gamma.ttk.pte.hu> - Thursday, December 05, 2019 10:38:24 AM - IP Address:84.1.165.145

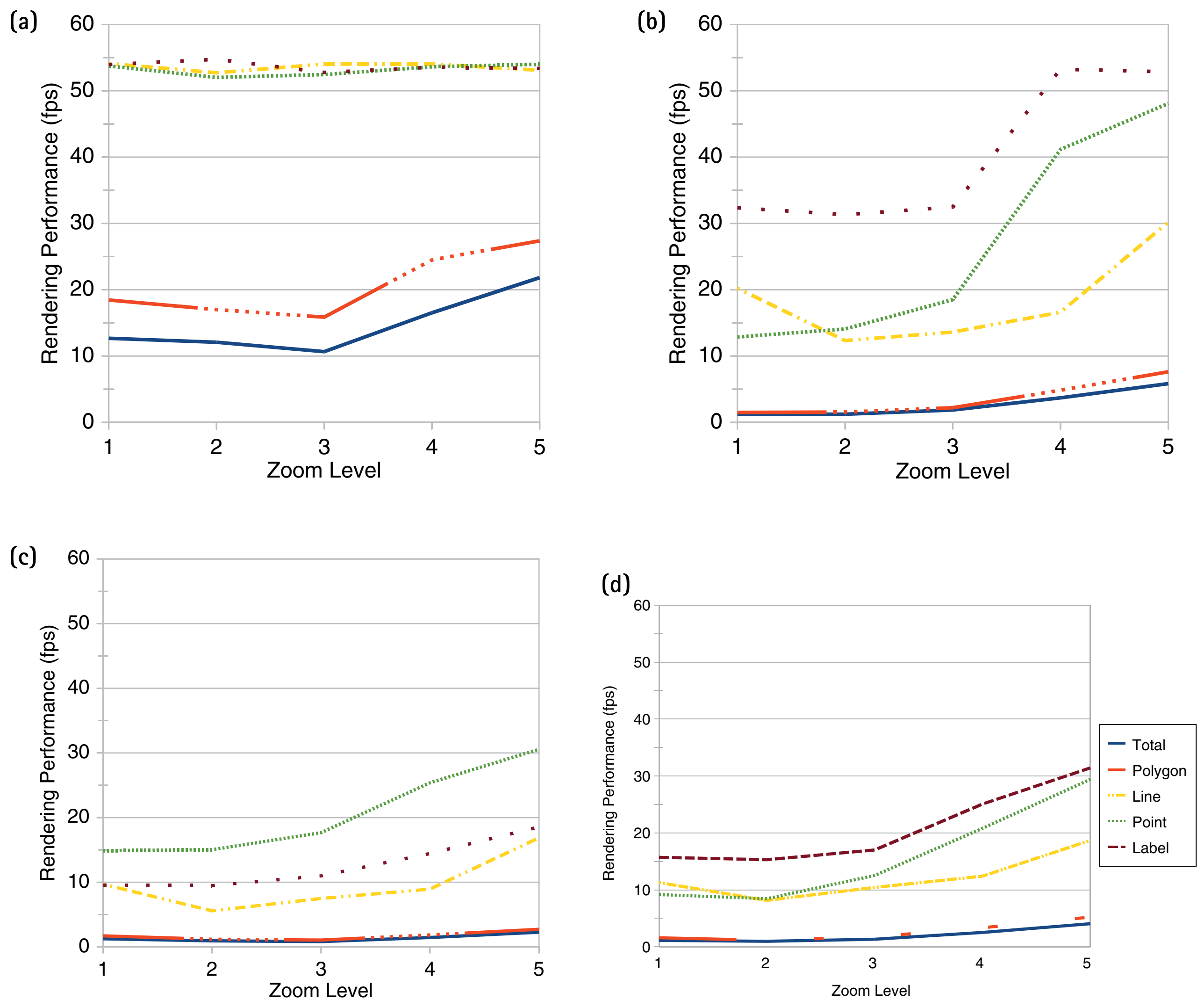

Figure 4. Detailed results of the thematic layer group's rendering tests on the ARM Mali GPU: (a) WebGL animating performance, (b) Canvas animating performance, (c) WebGL drawing performance, and (d) Canvas drawing performance

Table 4. Rendering performance (fps) of the GIS layer group

	Zoom level				
	8	9	10	11	12
NVIDIA GPU					
WebGL animating	59.05	57.28	57.76	57.76	58.69
Canvas animating	3.43	4.03	8.34	15.54	28.00
WebGL drawing	4.10	3.53	7.29	13.01	28.99
Canvas drawing	2.67	2.90	7.18	13.31	23.54
Intel GPU					
WebGL animating	17.33	24.83	44.85	51.64	56.30
Canvas animating	2.75	2.41	6.14	11.00	18.85
WebGL drawing	2.97	3.05	5.59	9.91	16.08
Canvas drawing	2.04	2.13	5.29	8.96	14.49

Note*:* Lags are emphasized by underlining, and severe lags by framing.

https://www.utpjournals.press/doi/pdf/10.3138/cart.54.4.2018-0014 - Gábor Farkas <gfarkas@gamma.ttk.pte.hu> - Thursday, December 05, 2019 10:38:24 AM - IP Address:84.1.165.145

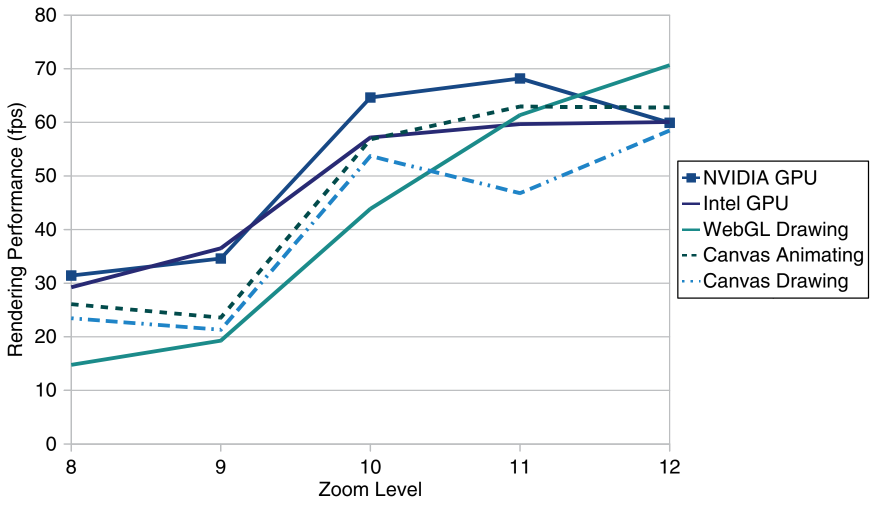

Figure 5. Rendering performance of the GIS point layer in various circumstances

with the Canvas renderer's drawing functionality (on both of the GPUs), while during animations, it was able to draw even 10,000 points with an acceptable frame rate. The WebGL renderer took the task easily, as it drew the marker textures smoothly on every zoom level.

Gaining Additional Insight by Profiling

Profiling the application's rendering pipeline served two purposes. First, the different browser tasks (e.g., painting, scripting) could be measured using the layers of the thematic map individually. Then, using the profiler's call stack, outlying function calls were identified, and the two rendering engines were compared based on their current and possible performance. Since the ratio of different calls depends on the sample size according to their time complexity, this measurement took account of the entire layers, and was used to separate significant parts (i.e., functions with the greatest time complexity) of the pipeline from insignificant. Second, profiling was also used to study the first set of measurements in greater depth. Examining some of the more interesting cases from the first benchmark made it possible to compare them with the reference data collected from the individual layers.

For this part of the analysis, Chromium's profiler was used. To every profile a few seconds delay was added, as occasional behaviour adding some irrelevant calls to the start of every benchmark was observed. From the relevant part of the timeline, idle time was subtracted, and the rest of the profile was analyzed. Since thematic and GIS layers were both profiled, the NVIDIA GPU was used for this task.

Following the profiler's categories, the following tasks were differentiated: scripting, rendering, painting, and other. The "other" category aggregates calls that cannot be grouped into any of the other categories, and their nature is not detailed by the profiler. As the scripting category (JavaScript calls) was dominant in every measurement, the ratio of the two rendering stages (preparation, composition) was also recorded. This holds valuable information, as the preparation stage can be completely cached between animation and interaction frames (e.g., during a pan or a zoom) if new geometries and images outside of the original rendered extent are not required until the animation is finished. Finally, the most dominant JavaScript calls were identified – if any – to predict more accurate differences in an optimal environment.

TRENDS OF RENDERING SIMPLE GEOMETRIES

The point and line renderers (Table 5) describe the two rendering engines really well. They do not have any significant JavaScript calls, nor great differences in their ratios of distinct types of calls. These results confirm one's general expectation about a WebGL engine; as it works at a lower level than the 2D Canvas API (no matter if it is hardware-accelerated), it must be faster. In the case of points – and therefore images – the greatest benefit of using the WebGL renderer is the large number of cached calls during animations and interactions, due to the simplicity of rendering textures. During animation frames the rendering time – as the textures are already prepared and only need to be drawn – decreased by about 60–80%, while with a Canvas renderer this gain was only about 40–50%. Line rendering does not have such significant differences. The main benefit of using a WebGL renderer is the performance boost. As line rendering is simple enough for most of it to be implemented using GLSL (OpenGL Shading Language), it can be boosted significantly compared with Canvas methods. On the other hand, the ratios of preparation calls in the two engines

https://www.utpjournals.press/doi/pdf/10.3138/cart.54.4.2018-0014 - Gábor Farkas <gfarkas@gamma.ttk.pte.hu> - Thursday, December 05, 2019 10:38:24 AM - IP Address:84.1.165.145

made a similar 60–70% boost possible during animations and interactions.

Labels (Table 6), on the other hand, contradict general expectations. While calls of the Canvas implementations are evenly distributed in execution time, their WebGL counterparts have some outliers explaining the swapped trend in rendering performance. Label rendering is seemingly more efficient in the Canvas engine. However, there is one part that cannot be represented with only one measurement: the glyph atlas. While the Canvas renderer draws every label separately, the WebGL renderer becomes more efficient with less variation in label styles and characters. The reason it is slower than the Canvas renderer can be found in the most costly function it uses: getTextSize. As the WebGL renderer was implemented with compatibility in mind, it uses the Canvas API for drawing characters on glyph atlases. This process needs the space of every drawn character reserved, which also needs the Canvas API. By using some other methods (e.g., signed distance fields), the rendering time could be reduced by at least 30% at the expense of label style restrictions and the need to serve the supported fonts.

Table 5. Ratio of different calls when points and lines are rendered

	Point		LineString	
	Canvas	WebGL	Canvas	WebGL
Drawing time (ms)	19.10	9.66	46.65	21.20
Scripting (%)	79.01	89.86	87.16	78.49
Rendering (%)	0.42	1.14	0.24	0.61
Painting (%)	2.57	3.31	0.71	2.36
Other (%)	18.22	5.69	11.83	18.49
Preparation (%)	40.31	62.01	66.05	58.77
Composition (%)	26.65	7.76	17.43	11.79

TRENDS OF COMPUTATIONALLY EXPENSIVE CALLS

Polygon rendering (Table 6) is another different scenario. Most of the rendering time is spent in the scripting phase, making every other phase proportionally negligible. However, expressed in milliseconds, those phases are actually comparable to the Canvas engine. By investigating the outlying function calls, it can be deduced that most of the scripting time is spent on triangulation. The most dominant calls of the process (e.g., ear clipping, classifying) are responsible for about 80% of the total rendering time. However, by triangulating only once, and caching the triangulation, the rendering time for a single frame would decrease seriously. On the other hand, there would be no dramatic change in the distribution of different calls, since there is another outlier taking 10% of the original's, and 57% of the cached version's time: the line renderer. As the polygon renderer uses the line renderer for drawing outlines for the polygons, the execution time of the otherwise efficient line renderer is added to the polygon renderer's total time. This makes it less efficient than its Canvas counterpart for smaller maps.

Finally, to get better insight into the switched trends with larger layers, additional profiling was performed on the GIS dataset. The first scale (zoom level 8) gave a large enough difference between the two engines on the NVIDIA GPU. Consequently, point (populated places layer) and polygon (settlements layer) drawing calls were analyzed (Table 7) at the eighth zoom level, in addition to the thematic profiling. In the WebGL, engine triangulation was disabled. When greater numbers of points were rendered, the distribution of different calls remained mostly the same, except that the other category rose at the expense of the scripting calls' ratio. The same can be observed with polygons, but with a stronger tendency. As time spent in functions of the browser's Canvas API implementation grows disproportionately with the number of vertices, the cacheable proportion of the total

Table 6. Ratio of different calls when rendering labels and polygons

	Label		Polygon		
	Canvas	WebGL	Canvas	WebGL	WebGL-c[a]
Drawing time (ms)	11.47	21.16	37.27	283.84	54.03
Scripting (%)	78.5	63.52	59.67	99.63	98.04
Rendering (%)	0.87	0.80	0.32	0.04	0.22
Painting (%)	4.01	3.07	0.78	0.10	0.54
Other (%)	17.00	32.80	39.31	0.22	1.17
Preparation (%)	46.21	49.76	32.28	95.59	76.81
Composition (%)	11.33	3.59	23.10	3.34	17.53

[a] In this scenario, triangulation was disabled in the application.

https://www.utpjournals.press/doi/pdf/10.3138/cart.54.4.2018-0014 - Gábor Farkas <gfarkas@gamma.ttk.pte.hu> - Thursday, December 05, 2019 10:38:24 AM - IP Address:84.1.165.145

Table 7. Ratio of different calls when rendering the GIS group

	Point		Polygon	
	Canvas	WebGL	Canvas	WebGL
Drawing time (ms)	84.48	36.28	337.79	248.76
Scripting (%)	78.61	95.78	28.04	97.16
Rendering (%)	0.09	0.55	0.05	0.05
Painting (%)	0.37	0.83	0.15	0.10
Other (%)	20.98	2.78	71.76	2.67
Preparation (%)	49.49	86.99	14.56	68.28
Composition (%)	26.31	4.82	12.90	27.57

drawing time decreases. This makes the perceived performance of the Canvas API significantly worse than that of the WebGL engine with large layers.

Discussion

To put the findings of this study in context, a few real world examples are investigated in this section. When a developer needs to create an application, the first task is finding the most fitting technologies for the project. In the case of Web mapping or Web GIS development, this involves balancing spatial data flow between the server side and the client side. While handling spatial data on the client side is more flexible, it outsources the required processing resources to the users' machines, and increases network traffic. By targeting modern computers, this would not be a constraint; however, considering the variability of spatial data and current Web mapping technologies, balancing the application is still a relevant problem.

From an architectural point of view, a very typical case is when the developer has partial control over the server side, or does not have control at all. Such a scenario can occur if the developer chooses to publish a Web site using a pre-installed content management system such as WordPress, or uses a service such as GitHub Pages. In this case, the developer can only affect the client side of the application by scripting the Web page and uploading static files. This architecture limits the developer to using rasterized base maps and static vector data. Such a Web map can still tolerate global thematics irrespective of the rendering engine used, but only with sufficiently generalized geometry. A very detailed map (such as Natural Earth 1:10m) must be constrained to a smaller extent, or displayed using a WebGL rendering engine. Obviously, not only geometric details (number of vertices) affect the performance, but also the number of thematics (layers) displayed. Furthermore, as more complex styling is applied, the rendering becomes more difficult, making the application even slower.

The second case using the same approach is when the developer has complete control over the server side, and can deploy different spatial server applications at will. In this case, the developer has more options and is less dependent on the Web mapping library used on the client side. That is, performance issues from the size and complexity of the visualized data can be mitigated by using several techniques. One of them is server side rendering. This is a basic capability of spatial servers (e.g., GeoServer, MapServer), but one can make further server side optimizations by utilizing a tile caching application (e.g., GeoWebCache, MapCache). If client side styling is a criterion, then by using the right tools, scale-dependent geometry generalization is also possible. Combining this with a tiling scheme, the developer can use vector tiles, a modern, standardized way to serve large vector datasets. This is an optimal solution, although it restricts the client side, as the Web mapping library needs to recognize the format. By using any of these techniques, a developer can prepare spatial data to be used efficiently with a Canvas renderer.

Another aspect is the type of visualization. While a simple map (i.e., interactive representation of a digital map) places little constraint on the visualization technology, this is not the case with animated maps. Simple animations, such as random clouds or water, could be used with a Canvas renderer with some optimizations; however, animations involving massive calculations need a WebGL engine. Such animations include geometric distortions along a new dimension. This can be momentum (e.g., wind-flow maps), time (temporal datasets), or a spatial dimension (e.g., extruded building footprints). Additionally, while the Canvas technology is capable of displaying 3D scenes to a limited extent, real 3D Web maps involve too many calculations for a smooth experience.

The final issue investigated in this section is the type of application. While Web mapping is often used to visualize large spatial datasets, this type of application has a very important advantage. The data used by Web maps are usually either static or calculated in a predefined way; therefore the developer can consider their size during the development process. There is another type of application, on the other hand, that can use arbitrary data: Web GIS. The problem with developing a Web GIS client is twofold. First, one has to prepare the client application for digesting an arbitrary amount of data from the user. This can be mitigated by using a server side component, which stores and generalizes uploaded data before serving them to the user, making a Canvas-based renderer viable. Second, a Web GIS needs more functionality than a Web map. This problem by itself limits the developer in choosing between different client-side technologies, as only a few of the current Web-mapping libraries are capable of providing GIS functionality. Moreover, the two most popular Web-mapping libraries with extensive built-in or third-party-provided GIS

https://www.utpjournals.press/doi/pdf/10.3138/cart.54.4.2018-0014 - Gábor Farkas <gfarkas@gamma.ttk.pte.hu> - Thursday, December 05, 2019 10:38:24 AM - IP Address:84.1.165.145

capabilities (OpenLayers and Leaflet) both have mature and well-tested rendering engines built on the HTML5 Canvas technology. However, both of them have WebGL engines capable of rendering vector features with simple styling.

To sum up, the fuzziness in the number of GIS features needed in a Web-mapping application and in the complexity of required visualization techniques makes choosing the right Web-mapping library a hard task. The problem can be imagined in a triangle, which has GIS features, advanced visualization techniques, and large numbers of data as its points. If the number of data can be controlled, and no computationally intensive animations are required, a Canvas renderer is sufficient, which makes the number of GIS features a less relevant point. Similarly, if large numbers of data need to be visualized, but simple styling is acceptable, the WebGL engines of feature-rich Web-mapping libraries can be used. If the goal is only visualization, but with a large or arbitrary number of data, then a WebGL library focusing solely on spatial data visualization should be chosen. The most problematic situation is that in which an arbitrary number of data need to be visualized with advanced techniques in an application with extensive GIS functionality. In this case, the extensibility of the chosen library and the experience of the developer with it become important, as extending the library's capabilities becomes unavoidable.

Conclusions

In this paper the necessity of using a WebGL engine for rendering 2D maps has been explored. While WebGL offers peerless freedom in customizing the rendering pipeline, it also has a major drawback: one cannot access matured and de facto standard libraries (e.g., freetype, cairo) for hardware-accelerated rendering in browsers. Furthermore, as WebGL is a relatively new interface to a mature technology, most of the preprocessing algorithms and structures (e.g., glyph atlases, triangulation) need to be implemented; there are only a few JavaScript libraries to choose from. On top of that, as JavaScript is a high-level, interpreted language, these algorithms need more optimizations to offer performance similar to that of their compiled counterparts. This makes these few general libraries inadequate without their being tailored to the given application. In contrast, the Canvas API's rendering functions are written in the browser's native language and compiled with it, allowing better optimizations and higher performance at the expense of weaker customizability.

Based on the findings of this study, the following conclusions can be drawn:

- Using the hardware-accelerated Canvas API for smaller maps (up to around 2000 features and 60,000 vertices) either is faster, or the difference is negligible.
- The Canvas API is less scalable than a custom WebGL implementation; therefore a WebGL map slowly outperforms a Canvas map with an increasing number of vertices.
- While both of the engines' drawing performance becomes slow when large layers are rendered, WebGL buffers can be cached during animations. This makes WebGL maps appear to be smooth, while their drawing performance is still only a few frames per second.
- A WebGL renderer has a serious implementation overhead over the stable Canvas API, and also some restrictions. For example, either a glyph atlas uses the Canvas API, or font files need to be served. When performance and custom capabilities (e.g., interactions, format handling, and controls) are both needed, it is usually faster and more straightforward to implement the missing features. With that said, a Canvas-based Web-mapping library can easily outperform a WebGL-based one in terms of rendering capabilities, since the same functionality can be implemented with less effort.
- A WebGL engine should be used when its customizability is needed (e.g., animated vectors) or an arbitrary number of features need to be rendered at once (e.g., GIS applications). WebGL-powered libraries offer adequate performance for complete Web GIS solutions. A further optimization could be rendering smaller chunks of data asynchronously to minimize GPU starvation.
- For general Web maps, especially when the number of rendered vertices can be controlled (e.g., vector tiles), the Canvas API is sufficient. Therefore, most developers should not worry about performance, but stick with the most capable or most convenient library. On weaker handheld devices, the significance of the performance gain from a WebGL renderer to other, more trivial optimizations is questionable (e.g., using fewer, more generalized polygons).

Acknowledgements

The project has been supported by the European Union, co-financed by European Social Fund Grant EFOP-3.6.1.-16-2016-00004, entitled by Comprehensive Development for Implementing Smart Specialization Strategies at the University of Pécs. The author would like to express his personal gratitude to all of the reviewers of this article. From the numerous constructive critiques received, this paper was shaped to its current, greatly enhanced form. The author would especially like to thank one anonymous reviewer, who took the time to manually outline errors and leave a great number of comments on a printed draft

https://www.utpjournals.press/doi/pdf/10.3138/cart.54.4.2018-0014 - Gábor Farkas <gfarkas@gamma.ttk.pte.hu> - Thursday, December 05, 2019 10:38:24 AM - IP Address:84.1.165.145

Author Information

Gábor Farkas is a PhD candidate at the Institute of Geography, University of Pécs, Hungary. He holds a diploma in geography from the same institution. He became a self-taught software developer in his early studies, which still determines his research. His main research interests focus on Web cartography, Web GIS application development, and other information system development. E-mail: gfarkas@gamma.ttk.pte.hu.

Note

1. The full line string GLSL shader can be found at OpenLayers Contributors 2019. The other shaders can be found in their appropriate folders in OpenLayers' GitHub repository.

References

Agafonkin, V. 2016. "How WebGL Vector Maps Work." In FOSS4G Bonn 2016, 20. FOSS4g and Open Source Geospatial Foundation. https://doi.org/10.5446/20352.

Antoniou, V., J. Morley, and M. Haklay. 2009. "Tiled Vectors: A Method for Vector Transmission over the Web." In *International Symposium on Web and Wireless Geographical Information Systems*, Lecture Notes in Computer Science 5886, ed. J.D. Carswell, A.S. Fotheringham, and G. McArdle, 56–71. Berlin: Springer.

Cozzi, P., and K. Ring. 2011. *3D Engine Design for Virtual Globes*. Boca Raton, FL: CRC Press.

Crockford, D. 2008. *JavaScript: The Good Parts*. Sebastopol, CA: O'Reilly Media.

Di Benedetto, M., F. Ponchio, F. Ganovelli, and R. Scopigno. 2010. "SpiderGL: A JavaScript 3D Graphics Library for Next-Generation WWW." In *Proceedings of the 15th International Conference on Web 3D Technology*, ed. M.K. Zuffo and Association for Computing Machinery, 165–74. New York: Association for Computing Machinery.

Farkas, G. 2017. "Applicability of Open-Source Web Mapping Libraries for Building Massive Web GIS Clients." *Journal of Geographical Systems* 19(3): 273–95. https://doi.org/10.1007/s10109-017-0248-z.

Farkas, G. 2018a. OpenLayers Visual Bbenchmarking Application. Available at https://gaborfarkas.github.io/rendering_pub/profile/.

Farkas, G. 2018b. Web Mapping Library Benchmarking Application. Available at https://gaborfarkas.github.io/rendering_pub/bench/.

Grigorik, I., J. Mann, and Z. Wang. 2016. Performance Timeline Level 2. W3C Editor's Draft 15. Available at https://w3c.github.io/performance-timeline.

Held, M. 2001. "FIST: Fast Industrial-Strength Triangulation of Polygons." *Algorithmica* 30(4): 563–96. https://doi.org/10.1007/s00453-001-0028-4.

Jayathilake, D., S. Perera, S. Bandara, H. Wanniarachci, and L. Herath. 2011. "A Technical Insight into Community Geographic Information Systems for Smart-Phones." In *2011 IEEE International Conference on Computer Applications and Industrial Electronics (ICCAIE)*, ed. IEEE, 379–84. Piscataway, NJ: IEEE.

Lienert, C., B. Jenny, O. Schnabel, and L. Hurni. 2012. "Current Trends in Vector-Based Internet Mapping: A Technical Review." In *Online Maps with APIs and WebServices*, ed. M.P. Peterson, 23–36. Heidelberg: Springer.

Liktor, G., and Dachsbacher, C. 2013. "Decoupled Deferred Shading on the GPU." In *GPU Pro 4: Advanced Rendering Techniques*, ed. W. Engel, 81–98. Boca Raton, FL: CRC Press.

Mapbox. 2015. "Earcut." Available at https://github.com/mapbox/earcut.

Meisters, G.H. 1975. "Polygons Have Ears." *American Mathematical Monthly* 82(6): 648–51. https://doi.org/10.1080/00029890.1975.11993898.

Mengeringhausen, H.C., and W.R. Witherell. 1962. "A Nonstandard Use of 16mm to Meet the 8mm Print Cost Challenge." *Journal of the SMPTE* 71(8): 566–68. https://doi.org/10.5594/j09333.

Mowery, K., and H. Shacham. 2012. "Pixel Perfect: Fingerprinting Canvas in HTML5." In *Proceedings of W2SP*, ed. M. Fredrikson. 1–12. IEEE. Available at http://modul.repo.mercubuana-yogya.ac.id/modul/files/openjournal/OpenJournnalOfIndustry/canvas.pdf.

Nogueira, E.T. 2012. "WebGL: A New Standard for Developing 3D Applications." *Virtual Reality and Scientific Visualization Journal* 5(2): 40–60.

OpenLayers Contributors. 2019. LineStringReplay Default Shader. Available at https://github.com/openlayers/openlayers/blob/v5.3.3/src/ol/render/webgl/linestringreplay/defaultshader.glsl.

Peng, Z.R., and C. Zhang. 2004. "The Roles of Geography Markup Language (GML), Scalable Vector Graphics (SVG), and Web Feature Service (WFS) Specifications in the Development of Internet Geographic Information Systems (GIS)." *Journal of Geographical Systems* 6(2): 95–116. https://doi.org/10.1007/s10109-004-0129-0.

Sauerwein, T. 2010. "Evaluation of HTML5 for Its Use in the Web Mapping Client Open-Layers." M.S. thesis, Hochschule Kaiserslautern University of Applied Sciences, Kaiserslautern.

Seidel, R. 1991. "A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons." *Computational Geometry* 1(1): 51–64. https://doi.org/10.1016/0925-7721(91)90012-4.

Shewchuk, J. R. 1996. "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator." In *Applied Computational Geometry: Towards Geometric Engineering*, ed. M.C. Lin and D.N. Manocha, 203–22. New York: Springer.

Springmeyer, D. 2015. "Mapbox Vector Tile Specification Adopted by Esri." Available at https://www.mapbox.com/blog/vector-tile-adoption.

https://www.utpjournals.press/doi/pdf/10.3138/cart.54.4.2018-0014 - Gábor Farkas <gfarkas@gamma.ttk.pte.hu> - Thursday, December 05, 2019 10:38:24 AM - IP Address:84.1.165.145

Appendix: Code Structure

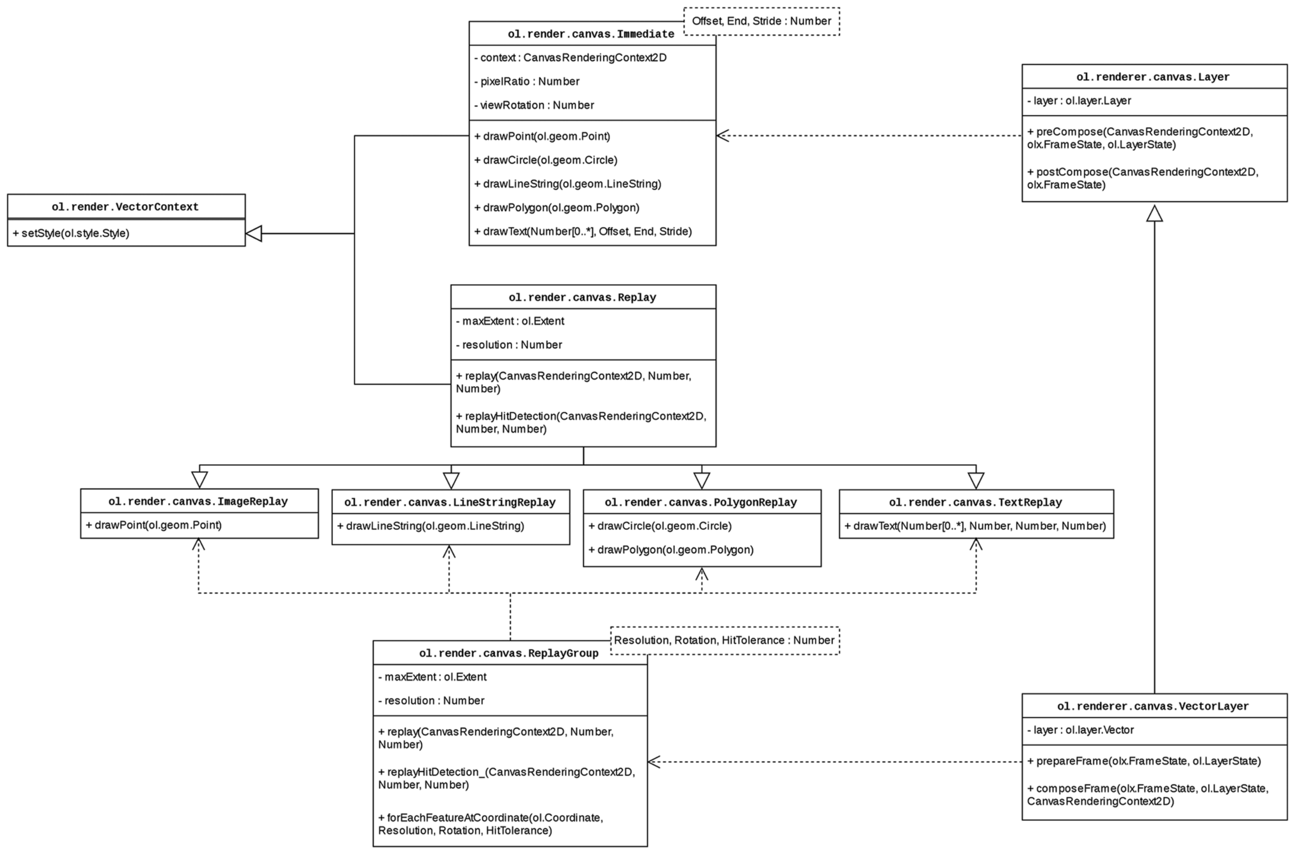
ol.render.canvas.Immediate
Offset, End, Stride : Number
- context : CanvasRenderingContext2D
- pixelRatio : Number
- viewRotation : Number
+ drawPoint(ol.geom.Point)
+ drawCircle(ol.geom.Circle)
+ drawLineString(ol.geom.LineString)
+ drawPolygon(ol.geom.Polygon)
+ drawText(Number[0..*], Offset, End, Stride)
ol.renderer.canvas.Layer
- layer : ol.layer.Layer
+ preCompose(CanvasRenderingContext2D, olx.FrameState, ol.LayerState)
+ postCompose(CanvasRenderingContext2D, olx.FrameState)
ol.render.VectorContext
+ setStyle(ol.style.Style)
ol.render.canvas.Replay
- maxExtent : ol.Extent
- resolution : Number
+ replay(CanvasRenderingContext2D, Number, Number)
+ replayHitDetection(CanvasRenderingContext2D, Number, Number)
ol.render.canvas.ImageReplay
+ drawPoint(ol.geom.Point)
ol.render.canvas.LineStringReplay
+ drawLineString(ol.geom.LineString)
ol.render.canvas.PolygonReplay
+ drawCircle(ol.geom.Circle)
+ drawPolygon(ol.geom.Polygon)
ol.render.canvas.TextReplay
+ drawText(Number[0..*], Number, Number, Number)
ol.render.canvas.ReplayGroup
Resolution, Rotation, HitTolerance : Number
- maxExtent : ol.Extent
- resolution : Number
+ replay(CanvasRenderingContext2D, Number, Number)
+ replayHitDetection_(CanvasRenderingContext2D, Number, Number)
+ forEachFeatureAtCoordinate(ol.Coordinate, Resolution, Rotation, HitTolerance)
ol.renderer.canvas.VectorLayer
- layer : ol.layer.Vector
+ prepareFrame(olx.FrameState, ol.LayerState)
+ composeFrame(olx.FrameState, ol.LayerState, CanvasRenderingContext2D)

Figure A.1. UML diagram of OpenLayers' rendering pipeline using the Canvas rendering engine as an example. The diagram shows only the most important classes for rendering vectors with their most important parameters and methods