Dash Sylvereye: A WebGL-powered Library for Dashboard-driven Visualization of Large Street Networks

ALBERTO GARCIA-ROBLEDO[1], **MAHBOOBEH ZANGIABADY**[2]
[1]Conahcyt, CentroGeo, Querétaro, México (e-mail: agarcia@centrogeo.edu.mx)
[2]University of Twente, Enschede, the Netherlands (e-mail: m.zangiabady@utwente.nl)

Corresponding author: Alberto Garcia-Robledo (e-mail: agarcia@centrogeo.edu.mx).

ABSTRACT State-of-the-art open network visualization tools like Gephi, KeyLines, and Cytoscape are not suitable for studying street networks with thousands of roads since they do not support simultaneously polylines for edges, navigable maps, GPU-accelerated rendering, interactivity, and the means for visualizing multivariate data. To fill this gap, the present paper presents Dash Sylvereye: a new Python library to produce interactive visualizations of primal street networks on top of tiled web maps. Thanks to its integration with the Dash framework, Dash Sylvereye can be used to develop web dashboards around temporal and multivariate street data by coordinating the various elements of a Dash Sylvereye visualization with other plotting and UI components provided by the Dash framework. Additionally, Dash Sylvereye provides convenient functions to easily import OpenStreetMap street topologies obtained with the OSMnx library. Moreover, Dash Sylvereye uses WebGL for GPU-accelerated rendering when redrawing the road network. We conduct experiments to assess the performance of Dash Sylvereye on a commodity computer when exploiting software acceleration in terms of frames per second, CPU time, and frame duration. We show that Dash Sylvereye can offer fast panning speeds, close to 60 FPS, and CPU times below 20 ms, for street networks with thousands of edges, and above 24 FPS, and CPU times below 40 ms, for networks with dozens of thousands of edges. Additionally, we conduct a performance comparison against two state-of-the-art street visualization tools. We found Dash Sylvereye to be competitive when compared to the state-of-the-art visualization libraries Kepler.gl and city-roads. Finally, we describe a web dashboard application that exploits Dash Sylvereye for the analysis of a SUMO vehicle traffic simulation.

INDEX TERMS Data visualization, data analysis, software libraries, component architectures, complex networks, graphical user interfaces, graphics, vehicle dynamics

ONE of the primary objects of interest of urban researchers and planners are street networks. They study street networks for a variety of applications such as traffic engineering, transportation, and urban planning. Recently, academics from seemingly unrelated fields, such as Networks Science and Computer Science, have joined to study the complexity of large street networks and develop efficient algorithms to process them.

With the advent of the OpenStreetMap (OSM) project [11] the street topology of virtually any city in the world became publicly available for analysis. Tools like OSMnx [5], [6] make it easy for any researcher to download OSM street network data with a simple query. However, the availability of such networks has also revealed the limitations of current tools like graph visualization.

Urban researchers need tools to make sense of multivariate data associated with street networks. These data are hardly static: vehicle counts, vehicle positions, traffic bottlenecks, and other urban data change over time. Dashboards have become a standard visual analytics tool when trying to make sense of multivariate data. Prominent dashboard tools in the industry include Tableau [3] and Google Data Studio[1]. However, these kinds of open tools are too general to support the practical analytical needs of real-world urban applications [26].

On the other hand, the street network of large cities is made of dozens of thousands of nodes and edges. This imposes the need to push the processing capabilities of graphics adapters to render such large structures. These complex visualizations should also allow for user interactivity by enabling navigation, panning, zooming, and clicking.

The use of web technologies for developing visualization solutions is currently a tendency among practitioners. Open-source programming libraries like the Dash framework en-

[1]https://datastudio.google.com/

able data analysts to develop their own rich and interactive web dashboards by exploiting a variety of coordinated web plotting and UI components. Such dashboards can be displayed in any modern web browser and be easily deployed on the web.

State-of-the-art graph visualization tools like Gephi, KeyLines, and Cytoscape are not suitable for studying city-scale street networks since they do not support simultaneously polyline[2] drawing for edges, navigable maps, interactivity, the means for visualizing multivariate data, and GPU-accelerated rendering.

In this context, the research question we tackle is: how to fill this gap and exploit state-of-the-art visual analytics techniques and web technologies to produce interactive visualizations of street networks on a city scale, along with its multivariate data, making this technology easily available to researchers and practitioners alike?

To answer the posed question, this paper presents a new Python library called Dash Sylvereye which produces interactive visualizations of primal street networks on top of tiled web maps. Thanks to its integration with the Dash framework, Dash Sylvereye can be easily exploited to develop web dashboards around temporal and multivariate urban data by coordinating the various elements of a Dash Sylvereye visualization with other Dash plotting and UI components.

Dash Sylvereye can render large city-scale interactive street networks as well as thousands of interactive markers in commodity computers with the help of the system's GPU through WebGL.

The core contributions of this paper are as follows:

- A library tool for Python that generates street network visualizations that can draw atop web tile maps and that is designed from the ground up to be compatible with the widely used Dash dashboard visualization framework.
- A library tool that allows for the customization of colors, sizes, transparency, and visibility of individual street network elements as well as markers. Visual properties can be also automatically scaled based on the values found in the street network's data.
- A library tool that provides fast software acceleration and exploits hardware acceleration for redrawing, showing panning speeds of close to 60 FPS, and CPU times below 20 ms, for street networks with thousands of edges.

The rest of this paper is structured as follows. Section I provides additional background on topics that are relevant to the proposed solution. Section II offers a review of the state-of-the-art on street graph visualization. Section III lists the requirements we identified were needed to meet and presents details on the internal design of the Dash Sylvereye library. Section IV offers the reader a quick grasp of how coding with the Dash Sylvereye library feels. In Section V we assess the animation performance of Dash Sylvereye in terms of frames per second, frame duration, and CPU time. In Section VI we present a comparison of the animation performance among Dash Sylvereye and other two state-of-the-art road network visualization libraries. In Section VII we describe a non-trivial example of a dashboard that uses Dash Sylvereye as its central component. Finally, in Section VIII we offer final conclusions and future work.

[2]A sequence of connected segments that describe a curve.

I. BACKGROUND

A. STREET NETWORKS

This paper is concerned with primal street graphs. A primal street graph is a non-planar directed multi-graph with loops allowed where nodes represent street intersections or junctions, and edges represent street segments [6]. A street network is a kind of spatial network [1]: a graph that models natural, sociological, or technological phenomena where the elements of the graph are mapped to the spatial dimension, usually to geographical coordinates. Urban networks have become the focus of many works in recent years. An example of such works is [14], which makes use of a new model for analyzing urban network structures, combining them with the information provided by taxi trajectory data.

B. WEB-BASED VISUALIZATION

The wide availability of web browsers has turned them into an all-pervasive execution platform. Recently, an increasing number of web-based visualization applications have been proposed motivated by the new technologies offered by modern browsers [19]. The HTML5 standard gives programmers an array of options to render graphics: the HTML canvas, SVG graphics, CSS animations, and WebGL. WebGL is a standardized JavaScript API for rendering GPU-accelerated graphics in web browsers. A WebGL application consists of two parts: control code written in JavaScript and shader code written in the GLSL language. WebGL has particularly attracted the interest of the data visualization community since it allows programmers to exploit the GPU processor regardless of the vendor.

A good example of a work that exploits state-of-the-art web visualization technologies for graph analysis is ContraNA. ContraNA [8] is a visual analytics framework that exploits machine learning to compare two networks for learning the main specifications of one network with respect to the other. Such comparison is challenging due to the complex structure of large graphs. The authors developed ContraNA as a web application. The back-end is developed in Python whereas the front-end uses a combination of HTML5, JavaScript, D3.js, and WebGL. WebSockets are used for back-end and front-end communication. Examples of works that exploit WebGL to produce visualizations of large graphs include [7], [12], [17], [22].

JavaScript has become the *lingua franca* for front-end web development. There already exists a mature ecosystem of open-source JavaScript libraries which are being exploited for data visualization. Prominent examples of such libraries are D3.js and Three.js. More recently, WebAssembly has enabled developers to write high-performance code that rivals in speed with native applications written in C and C++. This technology opens new possibilities for efficiently running compute-intensive algorithms in the browser, such as graph layout algorithms.

C. DASHBOARD VISUALIZATION

Dashboards are one of the most common use cases for data visualization [21]. Interest in developing web dashboards has recently increased in governments, universities, research centers, and health institutions due to the need of sharing real-time information about the state of the COVID-19 pandemic in an open and accessible manner.

In urban studies, dashboards are being used to visualize real-time urban data from a variety of sources to provide an easy-to-understand tool to decision-makers [9]. Dashboards can be used to visually assess urban performance to support the sustainable development of smart cities [15], and for transparent and accountable decision-making [18]. A good example of a work that exploits dashboards for city analytics purposes is [20], which proposes a dashboard-driven visual tool for analyzing traffic accident and casualty trends.

Python is one of the most-used languages among developers who identify themselves as data scientists [13]. There is a relatively new ecosystem of frameworks that are attracting the attention of data science practitioners in need of developing web dashboard applications entirely in Python, without the need of learning front-end web languages like HTML, CSS, and JavaScript. One such library is the Plotly Dash framework[3].

The Dash framework is built around the concept of *Dash component*. A Dash component is a Python class that provides an abstraction for a web UI element: from a single HTML tag to more complex elements such as a slider, a chart, a gauge meter, or a navigation bar. A Dash component has properties that can be set, read, and updated. Under the hood, Dash components are Python wrappers for components written with the widely-used React.js front-end UI framework[4]. This enables programmers to build their Dash components in JavaScript.

Dash applications are composed of two parts. The first part is the *layout* of the dashboard, which describes the application's appearance. It is specified as a tree of Dash components. The second part is the *callbacks*, which defines the interactivity of the application. Dash callbacks are Python functions that are automatically triggered when the properties of Dash components change.

A callback receives the values of the changed properties as input and returns new values for other properties as output. Every property of a Dash component can be updated through a callback. A special kind of input is the states: input parameters that do not trigger a callback, only store the state of a parameter at the moment the callback is triggered by another input parameter.

[3]https://plotly.com/dash/
[4]https://reactjs.org/

II. RELATED WORK

Many graph visualization tools have been developed over the last few years to generate graph visualizations. In consequence, the landscape of such tools has become extensive. We focus our state-of-the-art review on both open tools[5] and academic works that propose practical contributions with any of the two following features: 1) support for the development of visualization dashboards around the reported tool, 2) some sort of geospatial visualization support, such as the ability to render graphs on top of maps, and 3) rendering of large road networks. We discuss in detail tools that provide any kind of dashboard support in Section II-C.

A. TOOLS IN LITERATURE

The following are works in the academic literature that report graph visualization libraries for a variety of programming languages. We focus our review on tools that can render large road networks (through GPU hardware acceleration).

ccNetViz [22] is an open-source WebGL-based JavaScript library for network visualization. It supports animation features (nodes and links). Node colors, size, and transparency can be manipulated in real-time. Similarly, the animation of edges can be used to display information transmission. Animation features can be specified dynamically.

Carina [7] is a visualization tool that helps researchers to explore and visualize large graphs with millions of nodes. Carina supports fast graph drawing through WebGL and supports both desktop (Electron) and mobile platforms. An outstanding feature of Carina is it does not save the whole graph in RAM, enabling the tool to handle networks as big as 69 million edges.

Authors in [12] developed a visualization tool for large graphs called NetV.js. It is a WebGL-based JavaScript library that supports up to 50 thousand nodes and 1 million edges. It exploits the GPU to enhance the drawing performance and create an interface for manipulating graph components.

Argo Lite [17] is an interactive network visualization tool for web browsers. Users are enabled to modify the characteristics of nodes (size, shape, colors), links (colors), and labels (size and length). It uses WebGL to draw graphs fast. Users can import graph data from CSV, GEXF, and TSV files.

Authors in [23] developed a web-based application to visualize detailed information of transportation networks for mobility analytics by exploiting reachability maps. It is powered by GLSL.

Urban Network Analysis (UNA) [24] is a full-fledged toolbox that can be used to visualize spatial networks, as well as computing network measurements. It is provided as an extension for ArcGIS and Rhinoceros 3D. Support for GPU acceleration is not explicitly mentioned in the paper nor

[5]Tools that are publicly available as either open-source, free, or commercial products.

TABLE 1. Feature comparison of state-of-the-art network visualization tools. To grant support for a given feature, it should be provided out-of-the-box or via plug-ins/extensions. A N/A in the *Language* column means that the work is not a library but a web or desktop application. Notes: **(1)** Support of the feature is granted only if the tool can draw polylines for edges. **(2)** OSMnx can generate static Web visualizations with Folium. **(3)** By using the TimestampedGeoJson Folium plugin. **(4)** By using the Cytoscape.js library. **(5)** By using the Cytoscape.js Dash component. **(6)** A KeyLines visualization can be embedded in a Kibi dashboard. **(7)** There is a Tableau extension for embedding Kepler.gl visualizations. **(8)** By combining the Dash component with, for example, a Dash slider component that implements a timeline. **(9)** Since the tools are built on top of the Leaflet.js library, they may be able to display non-OSM tilemaps. **(10)** It supports OpenCL for accelerating compute-intensive tasks such as graph layout calculation. Cytoscape.js is not powered by WebGL. **(11)** With Gephi Toolkit. **(12)** With Cytoscape.js. **(13)** With PyGraphistry and GraphistryJS. **(14)** GPU acceleration is used for computing graph layouts. **(15)** It supports non-georeferenced polylines. **(16)** It supports JavaScript if the standalone React component is used. However, dashboard framework integration (and by extension timeline support) will not be available. **(17)** With graph-app-kit. **(18)** Support for GPU acceleration is mentioned neither in the paper nor webpage. Nonetheless, given that UNA is an extension, GPU acceleration support could be provided through the base packages (ArcGIS and Rhinoceros 3D) which support OpenGL. **(19** Python support is provided through the keplergl module.

Tool	Language	Polyline drawing (1)	Rendering on top of a map layer	GPU-accelerated rendering	Programmable interactivity	Timeline support	Dashboard framework integration
city-roads	JavaScript	Yes	No	WebGL	No	No	No
Folium	Python	Yes (2)	OSM, Mapbox, Stamen (9)	No	No	Yes (3)	No
Gephi	Java (11)	No	No	OpenGL	No	Yes	No
OSMnx	Python	Yes	OSM (2)	No	No	No	No
Cytoscape	JavaScript (12)	No	No	No (10)	Yes (4)	Yes (8)	Yes (5)
KeyLines	JavaScript	No	OSM (9)	WebGL	Yes	Yes	Yes (6)
Kepler.gl	Python, JavaScript	Yes	Mapbox	WebGL	Yes	Yes	Yes (7)
Sigma.js	JavaScript	No	No	WebGL	Yes	No	No
VivaGraphJS	JavaScript	No	No	WebGL	Yes	No	No
Graphistry	Python, JavaScript (13)	No	No	WebGL	No	Yes (17)	Yes (17)
ReGraph	JavaScript (19)	No	OSM (9)	WebGL	Yes	Yes	No
Ogma	JavaScript	No	OSM (9)	WebGL	Yes	Yes	No
G6	JavaScript	No (15)	No	No (14)	Yes	No	No
yFiles	JavaScript	No (15)	OSM (9)	WebGL	Yes	Yes	No
El Grapho	JavaScript	No	No	GLSL	Yes	No	No
ngraph.pixel	JavaScript	No	No	GLSL	Yes	No	No
react-force-graph	JavaScript	No	No	WebGL	Yes	No	No
ccNetViz	JavaScript	No	No	WebGL	Yes	No	No
Carina	N/A	No	No	WebGL	No	No	No
NetV.js	JavaScript	No	No	WebGL	Yes	No	No
Argo Lite	N/A	No	No	WebGL	No	No	No
Schoedon et al. 2019	N/A	Yes	Yes	GLSL	No	No	No
UNA	N/A	Yes	Yes	? (18)	No	No	No
D. Sylvereye	**Python (16)**	**Yes**	**OSM (9)**	**WebGL**	**Yes**	**Yes (8)**	**Yes**

webpage. Nonetheless, given that UNA is an extension, GPU acceleration support could be provided through ArcGIS or Rhinoceros 3D.

B. OPEN TOOLS

The following are tools that are made available openly through code repositories across the web. We limited our review on programming libraries that are still active, that show the aforementioned mentioned features, or with an associated programming library. For the sake of comparison with the library reported in this paper, we gave priority to JavaScript and Python libraries, but we also covered the widely-used Gephi tool, which is Java-based.

Folium[6] is an open-source Python tool that allows users to visualize data on an interactive Leaflet.js map. Users can zoom or click on the map to analyze the geo-referenced data.

OSMnx [5], [6] is an open-source Python library to easily download, visualize, and analyze urban street networks. It is built upon three widely used Python libraries, namely GeoPandas, NetworkX, and Matplotlib. It allows the user to extract street data from OSM for different transport modes such as walking, cycling and driving with a single line of code. OSMnx can also visualize isochrone maps.

Cytoscape [25] is an open-source graph visualization tool originally developed for biological network analysis. Cytoscape provides visualization functions that make it easy for researchers to interactively analyze complex graph datasets. However, it doesn't scale to high-volume graphs. Nonetheless, it supports offloading computationally intensive processing on a GPU, multi-core CPU, or multi-processor card by using OpenCL.

Gephi [2] is an open-source visualization tool for users who seek to generate static visualizations of graphs. It is a desktop application that supports a wide catalog of plug-ins. It is simple to use for beginners. Also, it makes it easy to create CSV files from the network's data. Graphs can be exported to a variety of formats. It is powered by OpenGL.

[6]https://github.com/python-visualization/folium

Gephi provides the Gephi Toolkit[7], a standalone Java library that programmers can use to generate visualizations programmatically.

Anvaka's city-roads[8] is an open-source visualization web tool written in JavaScript that extracts data from OSM to draw all the streets within a city. It is powered by WebGL.

Sigma.js[9] is an open-source JavaScript library that supports HTML canvas and WebGL renderers for graph visualization, as well as mouse and touch support. Thanks to its plug-in architecture, the library is extensible. It can import Gephi graphs in GEXF format.

VivaGraphJS[10] is an open-source JavaScript library that supports WebGL, Canvas, and SVG renderers for graph visualization. It is built on top of the ngraph[11] graph algorithms library.

ReGraph[12] is a commercial WebGL-powered React library for graph visualization by Cambridge Intelligence. It implements two visualization components: a chart and a time bar. It can render graphs on top of Leaflet.js web maps. Other geospatial features supported are geo-fencing, overlays, and multiple coordinate reference systems.

Ogma[13] is a commercial graph visualization JavaScript library by Linkurious. Ogma is powered by WebGL, but it also supports HTML5, Canvas, and SVG renderers. Inserting a custom UI on top of Ogma is possible. Geographical mode allows the programmer to display the graph on top of a web map from different map providers.

G6[14] is a graph visualization JavaScript library. It supports drawing polylines for edges. However, it does not support rendering graphs on top of maps. GPU acceleration is supported for computing graph layouts.

El Grapho[15] is an open-source JavaScript library for graph visualization that exploits GLSL shaders for quickly generating graph renderings of large graphs. The rendered graphs can be zoomed and panned. It supports multiple graph layout algorithms. Graph renderings in El Grapho are interactive.

ngraph.pixel[16] is an open-source JavaScript library by the creator of city-roads for visualizing non-road graphs. As city-roads, ngraph.pixel is powered by WebGL. Unlike city-roads, ngraph.pixel allows the programmer to listen to graph change events.

react-force-graph[17] is an open-source WebGL-powered library for graph visualization. react-force-graph is implemented as a React library. Its graph renderer is based on ShaderMaterial from the Three.js 3D JavaScript library[18]. It supports both 2D and 3D graph rendering.

C. TOOLS WITH DASHBOARD SUPPORT

The following are graph visualization tools that provide some kind of integration with dashboard visualization frameworks.

Cytoscape.js[19] is a JavaScript library for visualizing and interacting with graphs. It provides a rich set of features and APIs for creating graph visualizations, performing graph analysis, and implementing custom graph algorithms. Cytoscape.js allows the creation and manipulation of nodes and edges, apply various layout algorithms, customizing visual styles, and the handling of user interactions. Cytoscape.js can be integrated into Dash dashboards by exploiting the Dash Cytoscape component[20]. It does not provide WebGL support.

KeyLines[21] is a commercial JavaScript toolkit for visualizing and interacting with network and graph data. It is powered by WebGL. Since it is neither free nor open-source, users must purchase a license to use it. It supports events to react to user actions such as mouse clicks and drag-and-drop. Kibi, now known as Kibana, is an open-source data exploration and visualization platform primarily built for Elasticsearch. Kibana provides its own set of visualization components and plugins for creating dashboards and exploring data. KeyLines visualizations can be integrated into a Kibi dashboard by utilizing custom development and integration techniques. This may involve embedding KeyLines visualizations within Kibana's dashboard panels or incorporating KeyLines as a separate component within a Kibi dashboard.

Kepler.gl is an open-source geospatial data visualization library. Kepler.gl has the ability to display millions of data points representing numerous trips and perform real-time spatial aggregations by exploiting WebGL. By presenting geospatial data within a unified interface, Kepler.gl enables users to validate concepts and extract insights from these visualizations. Users have the flexibility to visualize spatial datasets with various map layers and explore the data through filtering, animation, and aggregation. The Kepler.gl Tableau extension integrates a Kepler.gl map visualization directly into the Tableau Desktop App, allowing users to interact with the map using the same user interface found in the Kepler.gl demo app. Additionally, the map can be configured to interact with other Tableau charts.

Graphistry is a commercial graph-based analysis tool. It supports WebGL acceleration and provides a Python library called PyGraphistry[22] which acts as a client to extract, transform, and load graphs into Graphistry. An alternative Graphistry client is the GraphistryJS JavaScript library[23].

[7]https://gephi.org/toolkit/
[8]https://github.com/anvaka/city-roads
[9]http://sigmajs.org/
[10]https://github.com/anvaka/VivaGraphJS
[11]https://github.com/anvaka/ngraph
[12]https://cambridge-intelligence.com/regraph/
[13]https://doc.linkurio.us/ogma/latest/
[14]https://g6.antv.vision/en
[15]https://www.elgrapho.com/
[16]https://github.com/anvaka/ngraph.pixel
[17]https://github.com/vasturiano/react-force-graph
[18]https://threejs.org/
[19]https://js.cytoscape.org/
[20]https://dash.plotly.com/cytoscape
[21]http://www.keylines.com
[22]https://github.com/graphistry/pygraphistry
[23]https://github.com/graphistry/graphistry-js

Graphistry provides the graph-app-kit [24], which is a toolkit designed to help build custom graph analytics applications and dashboards. More specifically, the graph-app kit provides integration with a dashboard environment based on the Streamlit library that can be deployed on the cloud. graph-app-kit provides a set of reusable components and utilities to assist with the integration of Graphistry's graph visualization and analysis capabilities into dashboard applications.

D. DISCUSSION

Table 1 shows a comparison of the discussed network visualization tools and academic works in terms of the following features: 1) programming language (for libraries), 2) support for polyline drawing for edges, 3) support for rendering graphs on top of a map layer, 4) support for GPU-accelerated rendering, 5) support for programmable interactivity, 6) support for a timeline and 7) support for integration with a dashboard framework. We believe that these are the feature a street network visualization tool or library should possess to make it useful for real-world urban street analysis.

As shown in Table 1, to the best of our knowledge, Dash Sylvereye is the only tool written for Python that generates street network visualizations that can draw atop web tile maps, that supports programmable user interactivity, that exploits hardware acceleration, and, most importantly, that is designed from the ground up to be compatible with a larger dashboard visualization framework.

When ignoring the target programming language, Kepler.gl is the one tool that holds the most similarities with Dash Sylvereye's feature set. However, unlike Dash Sylvereye, Kepler.gl is not written with dashboard integration as one of its core features.

III. DASH SYLVEREYE DESIGN

A. REQUIREMENTS

We aim to provide a flexible and accessible tool that allows for the visualization of large road networks with associated multivariate data on commodity systems. This aim involves a series of design requirements:

- *R1. Support for polyline drawing on top of web tilemaps.* An edge in a road network is defined as a sequence of coordinates that represent its shape in the actual geography. Visualizations should be able to show edges as a sequence of lines given the sequence of coordinates. Also, the road network should be rendered on top of an interactive web tilemap such as those provided by Mapbox or OSM, which allow the user to navigate through the map by panning and zooming.
- *R2. Support for markers.* Street network visualization is useful for practical applications insofar as it allows for the graphical representation of events that happen around the street network itself, such as traffic warnings, car accidents, and bottleneck spots, as well as places of interest (POI). A common practice in the industry to represent such information in products such as Google Maps and Waze is the use of markers. With this in mind, a street network visualization tool should provide support for drawing customizable markers on top of the map and the road network.
- *R3. Good frame rate for large street networks on commodity hardware.* The visualization tool should provide an animation frame rate of 24 FPS[25] or higher, for street networks with thousands of nodes and edges, to provide the user a responsive experience when navigating through the visualization (zooming and panning). Such responsive experience should be achievable without the need for a high-end GPU, on commodity hardware such as a laptop computer with a commodity integrated graphics processor (e.g. Intel HD Graphics and AMD Ryzen with Radeon graphics).
- *R4. Styles for nodes, links, and markers.* The tool should enable the user to customize the visual styles (e.g. color and size) of individual nodes, edges, and markers. Also, it should facilitate the use of the data associated with the street network for styling.
- *R5. Interactions.* The tool should allow the programmer to listen for events triggered when the user interacts with the elements of the visualization to define custom behavior such as retrieving and showing the data of a clicked node, or showing a popup with custom data on top of a clicked marker.
- *R6. Support for nodes, edges, and markers to store arbitrary data.* The tool should enable the user to associate arbitrary data with individual elements of the visualization. For example, edges obtained from OSM should be able to store its length, its road type (bridge, highway), maximum speed, etc.
- *R7. Integration with a dashboard framework.* Most importantly, the tool should enable the street network visualization to work natively with a well-known dashboard framework to allow for the creation of dashboard visualizations of multivariate urban data that complement and enrich the street network visualization.

Requirements R3, R4, and R5 have been previously identified by the authors in [12] as relevant for high-performance complex graph visualization after interviewing experts in the field and reviewing a series of state-of-the-art tools. Authors in [14] also acknowledge that, when it comes to studying urban networks with trajectory data, "the approach needs to handle a large number of city streets and massive trajectory data." Regarding requirement R6, authors in [10] note that "node properties and edge weights play a fundamental role in the field of multivariate network visualization," in the context of multifaceted graph visualization.

To address the aforementioned requirements, we developed Dash Sylvereye, a visual analytics library for generating graph-based and interactive visualizations of large street

[24]https://github.com/graphistry/graph-app-kit

[25]The standard minimum speed needed to experience realistic motion [16].

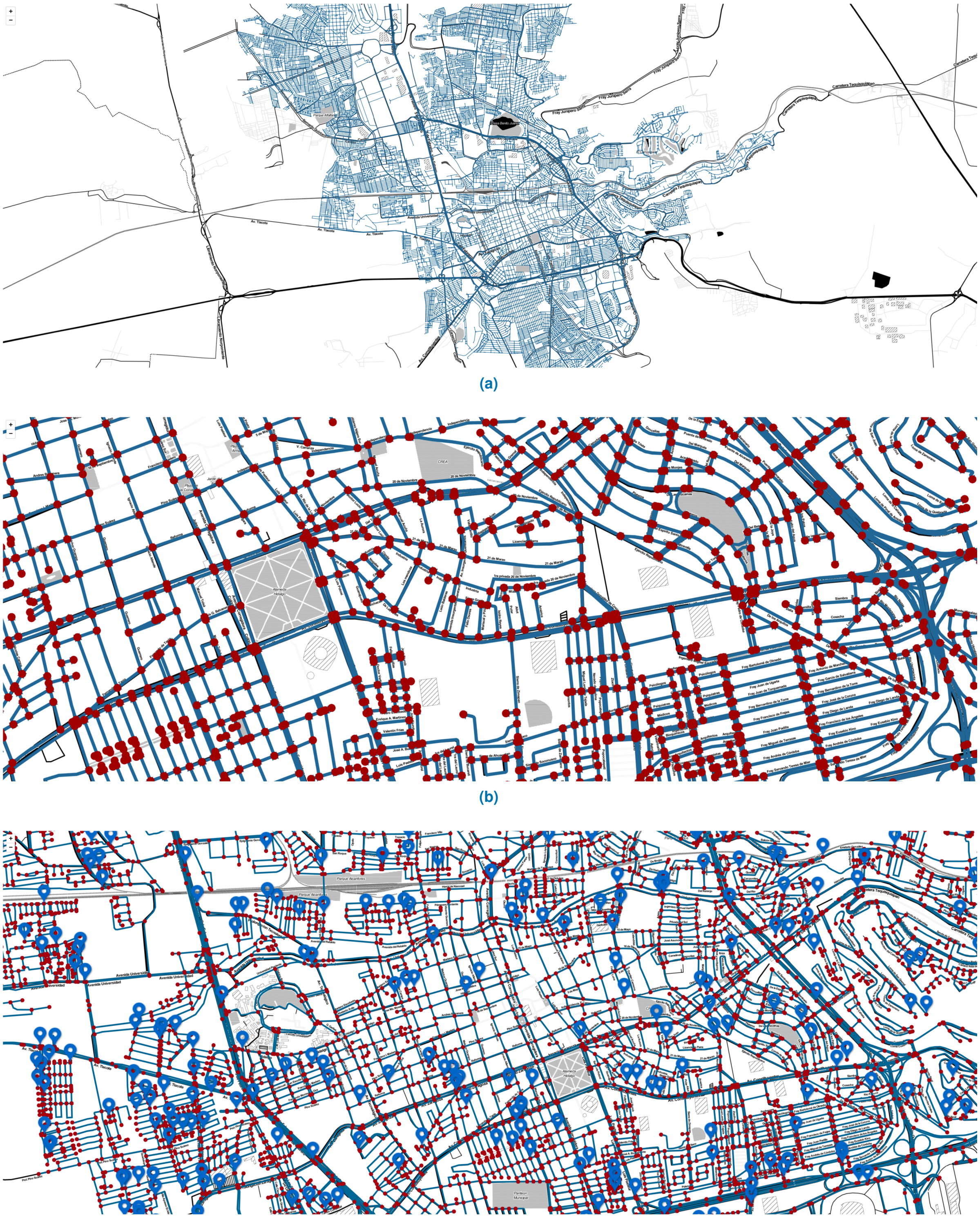


FIGURE 1. Screenshots of a Dash Sylvereye visualization displaying the street network of Queretaro City, Mexico, on top of an OSM tilemap layer, at different zoom levels and by showing different visualization layers. The visualized street network has 20,385 nodes and 49,137 edges.

networks and their associated multivariate data. It offers the following solutions to the identified requirements:

- *R1.* GPU-accelerated rendering of nodes to represent junctions and street road polylines on top of Leaflet.js interactive web maps.
- *R2.* GPU-accelerated rendering of fully customizable markers. Popups with custom text are supported.
- *R3.* Nodes, edges, and markers are rendered with WebGL for responsive and "smooth" navigation on networks with thousands of elements on any graphics adapter supported by modern web browsers.
- *R4.* Nodes, edges, and markers styles are customizable: color, size/width, transparency, and visibility. In the case of markers, the user can also customize the marker icon by providing a custom image. Color scales are supported and computed by the library.
- *R5.* Dash callback triggering when clicking individual nodes, edges, and markers. The user can also listen for changes in the map zoom level and any other property of the visualization component.
- *R6.* Individual nodes, edges, and markers can be associated with any arbitrary data. Functions are provided to load not only street network topologies but the data associated with them from OSM. The library uses simple list-of-dictionaries data structures for easy loading of networks from any other source.
- *R7.* The library is implemented as a component of the widely-used Dash framework. This enables Dash Sylvereye visualizations to be natively embedded into custom dashboards. In this way, Dash Sylvereye allows for the display of multivariate data with the help of the plotting components available in Dash, such as bar plots, line plots, and scatter plots. Dash integration also allows the user to coordinate Dash Sylvereye visualizations with a variety of Dash UI elements such as buttons, sliders, dropdown lists, etc.

A fully working and complete version of Dash Sylvereye for the Python programming language has been implemented. The following sections describe its design and implementation.

B. DESIGN

1) Layers

Dash Sylvereye is implemented as a Dash framework component. A Dash Sylvereye visualization is made of four layers:

1) *Tile layer.* Displays a zoomable and pannable web map generated by joining dozens of individually requested images in real time. Dash Sylvereye is built on top of Leaflet.js, enabling the user to select the tilemap provider of his/her preference (e.g. OSM and Mapbox).
2) *Edge layer.* Displays a clickable polygon for each edge in the street network. It also displays a direction arrow sprite for each edge. It can display edges with different widths, transparency, and color.
3) *Node layer.* Displays a clickable sprite for each node in the street network. It can display nodes with different sizes, transparency, and color.
4) *Marker layer.* Displays a clickable sprite for each marker. It can display markers with different sizes, transparency, color, and icon.

Fig. 1 shows screenshots of a Dash Sylvereye visualization displaying the street network of Queretaro City, Mexico, on top of an OSM tilemap, at different zoom levels and with different layers activated. The user can navigate through the visualization by panning and zooming it.

2) Data loading

Dash Sylvereye provides various convenient routines for loading street networks out of NetworkX graphs and GraphML files generated by the OSMnx library. In this way, the user can retrieve the street network of any city from OSM for visualization with a simple query in a single line of code.

3) Styling

The style of individual nodes, edges, and markers is customizable, allowing for the programmatic manipulation of colors, sizes, transparency, and visibility of individual graph elements. The user can also instruct Dash Sylvereye to automatically scale the size, color, and transparency based on values found in the street network's data. When using this coloring option, the user can decide whether to use a predefined or a custom color scale. Markers can show custom popup messages and the default marker's icon can be replaced by a custom SVG image.

4) Interactivity and coordination

As previously mentioned, nodes, edges, and markers are clickable, allowing for the definition of custom behavior at the user interaction. In addition, the callback architecture of the Dash framework enables the interaction between a Dash Sylvereye visualization and other Dash components. More specifically, any of the visual styles, the street network data, and the street network itself can be updated at runtime as a reaction to events emitted by other Dash components, such as time sliders and buttons. In this way, for example, the transparency and color of edges can be scaled to their vehicle count at different points in time selected via a Dash slider. This specific example would add support for the time dimension to Dash Sylvereye, allowing for the visualization of dynamic events in the street network.

5) Software and GPU acceleration

Dash Sylvereye exploits synchronous software (CPU) and GPU acceleration for displaying a large graph on a tiled web map as follows.

Dash Sylvereye uses PixiJS to draw the network. In a regular multimedia application (e.g. a videogame) written with PixiJS the main job of the GPU is to draw each frame efficiently to give the feeling of a smooth animation. In Dash

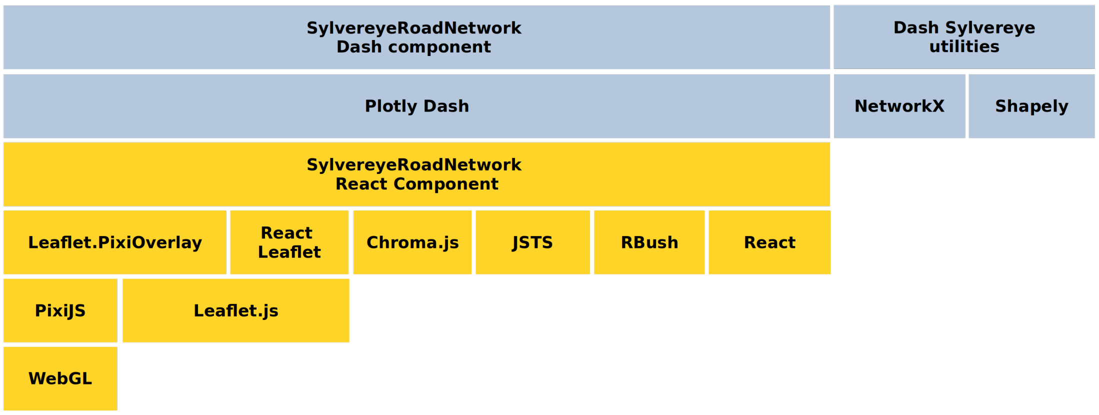


FIGURE 2. Stack of the main libraries used to build the Dash Sylvereye library. The diagram can be read in the top-down direction as follows: the library in a given layer uses the libraries in the layer located immediately below. The boxes in blue are Python libraries. Boxes in yellow are JavaScript libraries.

Sylvereye the drawing of a road network and the markers represents the redrawing of a single frame of such an animation. Thus, the GPU function is to render that animation frame as fast as possible. To that end, Dash Sylvereye uses Leaflet.PixiOverlay which allows to draw over a Leaflet.js overlay with PixiJS, which in turn uses WebGL for GPU-accelerated drawing of thousands of objects.

Due to the use of Leaflet.PixiOverlay, the GPU acceleration is involved when: (1) drawing the road network and markers for the first time and (2) redrawing the road network and markers after the user has interacted with it. More specifically, drawing/redrawing (and thus GPU acceleration) is triggered at three specific times:

1) At startup (first draw).
2) After panning the map, i.e. after releasing the left mouse button after panning.
3) After panning the map, i.e. after zooming in or zooming out.

The rest of the time (i.e. during the panning and zooming animations) the CPU handles the drawing work with software acceleration since the road network was already rendered.

C. SOFTWARE STACK

The Dash Sylvereye library is built on top of the following open-source JavaScript and Python libraries:

JavaScript libraries:

- *PixiJS*[26]: Cross-device 2D rendering library accelerated by WebGL for creating interactive graphics on web browsers. It acts as an abstraction layer for the WebGL API.
- *Leaflet.js*[27]: Mapping library for rendering interactive tiled web maps hosted on public servers with (optional) tiled overlays. Supports HTML5 and CSS3. It can create interactive layers.
- *Leaflet.PixiOverlay*[28]: Overlay class for Leaflet.js for WebGL-accelerated drawing on top of tiled web maps using PixiJS.
- *Chroma.js*[29]: Library for computing color conversions and color scales in the web browser.
- *JSTS*[30]: Library of spatial predicates and functions for processing geometries in web browsers. It is a JavaScript port of the JTS Java library.
- *RBush*[31]: Library for 2D spatial indexing of points and rectangles in web browsers. It is built around a custom R-tree data structure with bulk insertion support.
- *React.js*: Component-driven front-end library for building UI components maintained by Facebook.
- *React Leaflet*[32]: Bindings between React.js and Leaflet.js. Exposes Leaflet.js layers as React components.

Python libraries:

- *Plotly Dash*: User interface library for creating data-driven web applications around dashboard visualizations entirely in Python.
- *NetworkX*[33]: Social Network Analysis library for network reading, creation, generation, manipulation, measuring, and visualization.
- *Shapely*[34]: Library for manipulating geometric objects in the Cartesian plane.

Fig. 2 shows the library stack used to develop Dash Sylvereye. Leaflet.js provides a layer of tiled web maps as well as zooming and panning capabilities, whereas the PixiJS library provides WebGL-powered street network drawing primitives (polygons and sprites). This is done by using Leaflet.PixiOverlay which provides a Leaflet.js overlay where PixiJS can draw.

[26]https://www.pixijs.com/
[27]https://leafletjs.com/
[28]https://github.com/manubb/Leaflet.PixiOverlay
[29]https://gka.github.io/chroma.js/
[30]https://bjornharrtell.github.io/jsts/
[31]https://github.com/mourner/rbush
[32]https://react-leaflet.js.org/
[33]https://networkx.org/
[34]https://github.com/Toblerity/Shapely

Dash Sylvereye also makes use of other third-party JavaScript libraries, such as JSTS for defining edge-hit polygons, RBush for efficiently finding edge-hit polygons that have been clicked by the user, and Chroma.js for computing color scales for edges, nodes, and markers.

React Leaflet is used to bring everything together: the Leaflet.js map, the tilemap layer, and the road network visualization layer that exploits Leaflet.PixiOverlay. All these elements are encapsulated into the *SylvereyeRoadNetwork* React Component. The React component is then wrapped to produce the *SylvereyeRoadNetwork* Dash component by using the toolchain provided by Dash.

On the Python side, Dash Sylvereye network loading routines make use of NetworkX and Shapely, enabling Dash Sylvereye to import street networks from the OSM project via OSMnx or from OSMnx-generated GraphML files.

IV. USAGE EXAMPLE

This section presents a simple usage example to illustrate what programming with Dash Sylvereye looks like. The example is separated into three parts, namely initialization, interactivity, and styling.

In Fig. 3, the street network of Queretaro City is retrieved from OSM with the OSMnx library and then transformed to Dash Sylvereye's list-of-dictionaries data structure by using the utility function `load_from_osmnx_graph()`. Dash Sylvereye also provides the function `load_from_osmnx_graphml()` to load a street network from a graph file in GraphML format generated by OSMnx.

To insert a street network visualization in a Dash dashboard, the programmer only has to insert an instance of the class `SylvereyeRoadNetwork` in the dashboard application layout. The street network topology (and data) is provided via the `nodes_data` and `edges_data` parameters. Apart from the road network, the user can provide information about the map and the web tile layer by using an interface similar to that of Leaflet.js.

Table 2 shows the list of currently supported properties in the `SylvereyeRoadNetwork` class. These properties allow the user to set up the tilemap (e.g. tilemap provider and attribution), the road network data (e.g. nodes and edges), node/edge/marker style options, the layer visibility, and the map itself (e.g. zoom level and center). Recall that any of these parameters can be updated at runtime, triggering the automatic update of the visualization when changed. For example, if the user wants to update the street network topology, it is enough to update the `nodes_data` and the `edges_data` parameters in a callback.

Fig. 4 shows an example on the use of callbacks for reacting to user interaction by using the `clicked_node` and `clicked_edge` callback parameters listed in Table 2. Every time the user clicks a node, a callback provided by the programmer is triggered to update an `H2` Dash label component with the node's coordinates. Likewise, every time the user clicks an edge the provided callback is triggered to update an `H2` label component label with the edge's polyline coordinates.

The programmer can fine-tune the visuals of the street network visualization on an element-by-element basis by filling option dictionaries available for nodes, edges, and markers. Table 3 lists the currently supported style option methods. The user only needs to: 1) get an options dictionary pre-filled with default settings, 2) customize the options dictionary by selecting and setting up one or more visual option methods listed in Table 3, and 3) pass the dictionary to the Dash Sylvereye component. Again, if the user passes an updated options dictionary to Dash Sylvereye at runtime, the visualization will update accordingly in an automatic fashion.

In Fig. 5, the transparency level (alpha) of all nodes is set to 0.25 to make them translucent. Also, the size method is set to `NodeSizeMethod.SCALE` in order to set the diameter of all nodes in proportion to their weight. As for the visuals of edges, both the edge width and edge color methods are also set to `EdgeWidthMethod.SCALE` and `EdgeColorMethod.SCALE`, respectively, in order to be scaled in proportion to edge weights. Fig. 6 shows the resulting web dashboard when putting together the code provided in Figs. 3-5.

V. ANIMATION PERFORMANCE ASSESSMENT

We quantitatively assessed how "responsive" is to the user interaction with Dash Sylvereye visualizations on a commodity computer for a set of OSM street networks of varying sizes when exploiting software acceleration.

Panning[35] of a web map is an important operation since, in our case, it lets the user navigate the road network and explore its elements. We, therefore, assessed how smooth is the panning of a network visualization by measuring the screen refresh rate of a web page in terms of the animation frames per second (FPS). The CPU time and frame duration can offer insights for explaining the observed FPS.

We conducted the assessment on a commodity computer with a dual-core AMD A9-9425 processor at 3.1 GHz, with an Integrated AMD Radeon R5 (Stoney Ridge) GPU, and 7.2 GiB in RAM. The computer was running Linux Ubuntu 18.04.4 LTS 64-bits. Note that the processor used in the experiments is a mid-end mobile CPU with an integrated GPU that can be found in budget laptops.

The assessment methodology consisted of two main stages. In the first stage, we retrieved the data of street networks from OSM by running OSMnx with the query strings listed in Table 4 for four cities. We used the OSM website to get initial map center coordinates to open the test dashboards and then choose the final map centers and zoom levels. Final map centers and zoom levels were chosen in such a way that the whole street network was visible.

In the second stage, we conducted the following experiment for each street network. We used the performance tab

[35]Panning consists in holding the left button of the mouse and moving the mouse to navigate on the map.

```python
import osmnx as ox
from dash import Dash
from dash_html_components import Div
from dash_sylvereye import SylvereyeRoadNetwork
from dash_sylvereye.utils import load_from_osmnx_graph

OSMNX_QUERY = 'Santiago de Queretaro, Queretaro, Mexico'
TILE_LAYER_URL = '...'
TILE_LAYER_SUBDOMAINS = '...'
TILE_LAYER_ATTRIBUTION = '...'
MAP_CENTER = [20.5858171, -100.3888608]
MAP_ZOOM = 14
MAP_STYLE = {'width': '60%', 'height': '500px'}

road_network = ox.graph_from_place(OSMNX_QUERY, network_type='drive')
nodes_data, edges_data = load_from_osmnx_graph(road_network)

app = Dash()
app.layout = Div([
    SylvereyeRoadNetwork(id='sylvereye-roadnet',
                         tile_layer_url=TILE_LAYER_URL,
                         tile_layer_subdomains=TILE_LAYER_SUBDOMAINS,
                         tile_layer_attribution=TILE_LAYER_ATTRIBUTION,
                         map_center=MAP_CENTER,
                         map_zoom=MAP_ZOOM,
                         map_style=MAP_STYLE,
                         nodes_data=nodes_data,
                         edges_data=edges_data
                        )
])

if __name__ == '__main__':
    app.run_server()
```

FIGURE 3. Example showing how to embed a SylvereyeRoadNetwork component in a minimal Dash dashboard to display a street network obtained with OSMnx.

```python
from dash_html_components import H2
...

app = Dash()
app.layout = Div([
    SylvereyeRoadNetwork(id='sylvereye-roadnet', ...),
    H2(id='h2-node-coords'),
    H2(id='h2-edge-coords')
])

@app.callback(
    Output('h2-node-coords', 'children'),
    [Input('sylvereye-roadnet', 'clicked_node')])
def update_h2_node_coords(clicked_node):
    if clicked_node:
        return f'Clicked node coords: {clicked_node["data"]["lat"]},\
                                      {clicked_node["data"]["lon"]}'

@app.callback(
    Output('h2-edge-coords', 'children'),
    [Input('sylvereye-roadnet', 'clicked_edge')])
def update_h2_edge_coords(clicked_edge):
    if clicked_edge:
        return f'Clicked edge coords: {clicked_edge["data"]["coords"]}'

...
```

FIGURE 4. Example showing how to use Dash callbacks to react to mouse clicks on the street network's nodes and edges.

```
from dash_sylvereye.enums import (NodeSizeMethod,
                                  EdgeWidthMethod,
                                  EdgeColorMethod)
from dash_sylvereye.defaults import (get_default_node_options,
                                     get_default_edge_options)
...

node_options = get_default_node_options()
node_options["alpha_default"] = 0.25
node_options["size_method"] = NodeSizeMethod.SCALE
node_options["size_scale_field"] = "weight"

edge_options = get_default_edge_options()
edge_options["width_method"] = EdgeWidthMethod.SCALE
edge_options["width_scale_field"] = "weight"
edge_options["color_method"] = EdgeColorMethod.SCALE
edge_options["color_scale_field"] = "weight"
edge_options["color_scale_left"] = 0xcbdbff
edge_options["color_scale_right"] = 0x06696

app = Dash()
app.layout = Div([
    SylvereyeRoadNetwork(id='sylvereye-roadnet',
                         node_options=node_options,
                         edge_options=edge_options,
                         ...
                        ),
    ...
])

...
```

FIGURE 5. Example showing how to configure the visual styles of nodes (transparency and size) and edges (width and color scale) in a Dash Sylvereye visualization.

of the Chrome DevTools console to record the dashboard while manually panning the whole visualization by performing circular dragging movements. We used Google Chrome v85.0.4183.121.

Next, we manually registered the frame duration, frame FPS, and frame CPU time of 31 recorded animation frames from the Chrome DevTools performance tab to obtain statistically valid results. We repeated this experiment 10 times for each street network.

Fig. 7 shows the median frame FPS, duration, and CPU time for each experiment and each city. Fig. 8 shows the median values when merging all experiments for each city. We use the median because it is less sensitive to outliers than the average. Cities are sorted from smaller to larger from left to right.

Figures show that lower FPS values are associated with larger CPU times and frame durations. This might be explained by the fact that the more the CPU has to work the more the duration of an animation frame, negatively impacting the FPS in that animation frame.

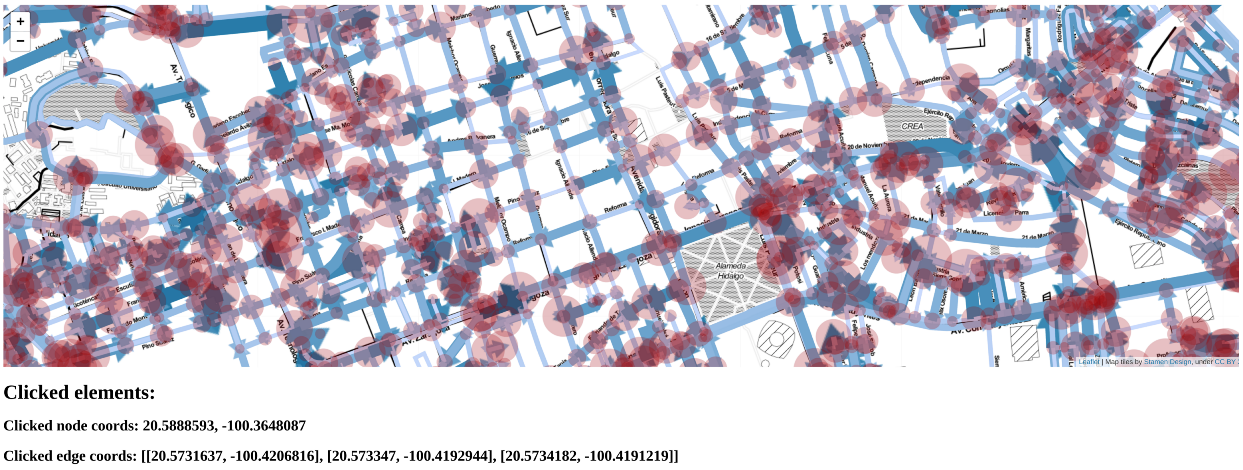
From the FPS perspective, figures show that the larger the city the lower the FPS, ranging from around 60 FPS for the Alameda city to around 10 FPS for the Beijing city. Nonetheless, Queretaro city, with 20k nodes and 49k edges, shows an FPS of above 24 FPS, suggesting that Dash Sylvereye can smoothly handle the panning of networks with dozens of thousands of nodes and edges on the experiment machine.

VI. ANIMATION PERFORMANCE COMPARISON

We also present a performance comparison between Dash Sylvereye and other state-of-the-art visualization libraries that can render large road networks: Kepler.gl and city-roads. We quantitatively measured and compared the responsiveness to the user interaction of the three tools on a commodity computer for the Alameda, Enschede, Queretaro, and Beijing road networks.

The hardware setup and the two-stage methodology were the same as in Section V. We conducted the 10 experiments for each tool sequentially and continuously in time, without interruptions (no computer reset, no login-logout, etc.) to get numbers as accurate as possible. There were no other apps and tabs other than the web browser was open. The full city map was always visible during the movements in all experiments. All circular movements were performed manually, and clockwise in all experiments. We used the same manual movements with regard to speed and diameter as humanly possible. The center and zoom level in Kepler.gl and Dash Sylvereye were adjusted programmatically whereas for city-roads we had to set the center and zoom level manually. For this experimentation, we used Google Chrome v87.0.4280.88 on Ubuntu 20.04.1 LTS (64-bit).

Fig. 9 shows the median frame FPS, duration, and CPU time for each experiment and each city for the three tools.



Clicked elements:

Clicked node coords: 20.5888593, -100.3648087

Clicked edge coords: [[20.5731637, -100.4206816], [20.573347, -100.4192944], [20.5734182, -100.4191219]]

FIGURE 6. Screenshot of the resulting dashboard example after putting together the code snippets shown in Figs. 3-5.

TABLE 2. The SylvereyeRoadNetwork Dash component supports an array of properties classified as follows: 1) data properties, 2) (style) option properties, 3) show/hide properties, map properties, 4) tile layer properties, and 5) callback properties. All but the callback properties are provided and updated by the user to set up and tune the street network visualization. Callback properties, on the other hand, are updated by Dash as a reaction to user-click interaction.

Property	Category	Brief description	Observations
`nodes_data`	Data	List of the road network's nodes	Nodes can hold arbitrary data in the 'data' field
`edges_data`	Data	List of the road network's edges	Edges can hold arbitrary data in the 'data' field
`markers_data`	Data	List of map markers	Markers can hold arbitrary data in the 'data' field
`node_options`	Options	Visual options dictionary for nodes	-
`edge_options`	Options	Visual options dictionary for edges	-
`marker_options`	Options	Visual options dictionary for markers	-
`show_nodes`	Show/hide	If false, all nodes will be hidden	Hidden nodes will cease to be interactive
`show_edges`	Show/hide	If false, all edges will be hidden	Hidden edges will cease to be interactive
`show_arrows`	Show/hide	If false, all direction arrows will be hidden	-
`show_markers`	Show/hide	If false, all markers will be hidden	Hidden markers will cease to be interactive
`map_center`	Map	Map center coordinates	In (latitude, longitude) format
`map_zoom`	Map	Map zoom level	As specified by Leaflet.js
`map_min_zoom`	Map	Minimum allowed map level	As specified by Leaflet.js
`map_max_zoom`	Map	Maximum allowed map level	As specified by Leaflet.js
`map_style`	Map	Map CSS styles	Provided in dictionary form: {'style': 'value'}
`tile_layer_url`	Tile layer	Tile layer URL template	As specified by Leaflet.js
`tile_layer_subdomains`	Tile layer	Tile layer attribution HTML text	As specified by Leaflet.js
`tile_layer_attribution`	Tile layer	Tile layer subdomains	As specified by Leaflet.js
`tile_layer_opacity`	Tile layer	Tile layer opacity	A value between 0 and 1
`clicked_node`	Callback	Used to invoke a callback when a node is clicked	Data will be available as the property's value
`clicked_edge`	Callback	Used to invoke a callback when an edge is clicked	Data will be available as the property's value
`clicked_marker`	Callback	Used to invoke a callback when a marker is clicked	Data will be available as the property's value

TABLE 3. Style methods available for nodes, edges, and markers. For example, there are three color methods for nodes: NodeColorMethod.DEFAULT, NodeColorMethod.SCALE, and NodeColorMethod.CUSTOM. DEAFAULT methods use the predefined settings provided by Dash Sylvereye, which can be customized. SCALE methods, on the other hand, scale visual style values in proportion to a weight field. CUSTOM methods allow styling based on the data associated with individual nodes, edges, and markers. For the case of visibility methods, ALWAYS instructs Dash Sylvereye to turn the visibility of all elements on. Some other style methods are more specific to a given kind of element, such as the ORIGINAL method for marker icons, which makes Dash Sylvereye use the original color of the SVG image specified as an icon.

Style	Node methods	Edges methods	Marker methods
Color	`DEAFAULT`, `SCALE`, `CUSTOM`	`DEAFAULT`, `SCALE`, `CUSTOM`	`DEAFAULT`, `SCALE`, `CUSTOM`, `ORIGINAL`
Size	`DEAFAULT`, `SCALE`, `CUSTOM`	N/A	`DEAFAULT`, `SCALE`, `CUSTOM`
Alpha	`DEAFAULT`, `SCALE`, `CUSTOM`	`DEAFAULT`, `SCALE`, `CUSTOM`	`DEAFAULT`, `SCALE`, `CUSTOM`
Visibility	`ALWAYS`, `CUSTOM`	`ALWAYS`, `CUSTOM`	`ALWAYS`, `CUSTOM`
Width	N/A	`DEAFAULT`, `SCALE`, `CUSTOM`	N/A
Icon	N/A	N/A	`DEFAULT`, `CUSTOM`

TABLE 4. Details of the OSM street networks and the tilemap configuration used for the animation performance assessment.

City name	OSMnx query string	Number of nodes	Number of edges	Center (lat, lon)	Zoom level
Alameda, US	Alameda, Alameda County, CA, USA	1,830	4,842	37.7618235, −122.2429843	15
Enschede, NL	Enschede, Overijssel, Netherlands, The Netherlands	5,337	13,587	52.2271595, 6.9046205	12
Queretaro, MX	Santiago de Querétaro, Querétaro, México	20,385	49,137	20.6025256, −100.3886302	12
Beijing, CN	Beijing, China	63,347	153,120	39.9116304, 116.4010405	9

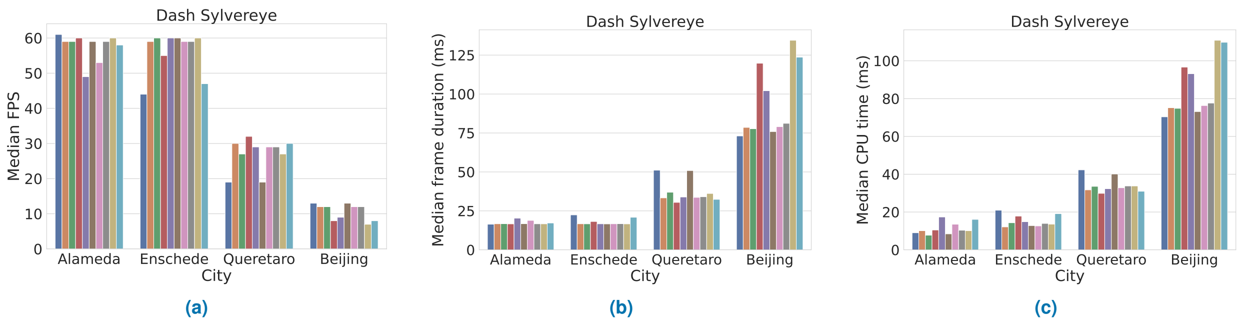


FIGURE 7. Dash Sylvereye's median frame FPS, duration, and CPU time for each experiment and each city. Cities are sorted from smaller to bigger from left to right.

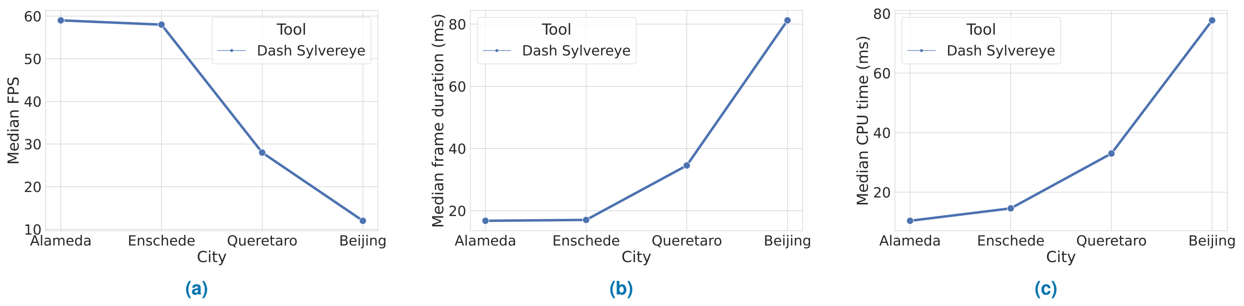


FIGURE 8. Dash Sylvereye's median frame FPS, duration, and CPU time when merging all experiments for each city. Cities are sorted from smaller to bigger from left to right.

Fig. 10 shows the median values when merging all experiments for each city for the three tools.

Figures show that, for the three tools, lower FPS values are associated with larger CPU times and frame durations. However, Figures also show that, unlike Dash Sylvereye, Kepler.gl and city-roads seem to be unaffected by the road network size. This suggests that these tools might exploit hardware acceleration during the panning process. In contrast, recall that Dash Sylvereye exploits hardware acceleration only after panning (and zooming) for redrawing.

Nonetheless, note in Fig. 9 that Kepler.gl showed FPSs lower than 10 in one experiment for Alameda, one experiment for Enschede, and two experiments for Queretaro. In contrast, for Alameda, Enschede, and Queretaro, Dash Sylvereye's FPS was higher than 20 whereas the duration and CPU remained low. For the largest city, Beijing, Dash Sylvereye's FPS was higher than 20 for three experiments.

Overall, Figures 9 and 10 show that Dash Sylvereye's performance was inversely proportional to the road network size. However, Fig. 9 also shows that Dash Sylvereye FPS outperformed Kepler.gl in three out of four cities (Alameda, Enschede, and Queretaro), whereas it outperformed city-roads in one out of four cities (Alameda). Additionally, as city-roads, Dash Sylvereye showed FPSs above 24 for three out of four cities (Alameda, Enschede, and Queretaro). With these results, Dash Sylvereye showed to be competitive when compared to both Kepler.gl and city-roads for road networks with dozens of thousands of nodes and edges.

VII. DASHBOARD EXAMPLE: QUERETARO CITY TRAFFIC SIMULATION

This section presents the design and implementation of an example dashboard application written with the Dash framework that exploits Dash Sylvereye for the analysis of post-mortem simulation data on the street network of Queretaro City, Mexico. For simulations, we made use of the SUMO

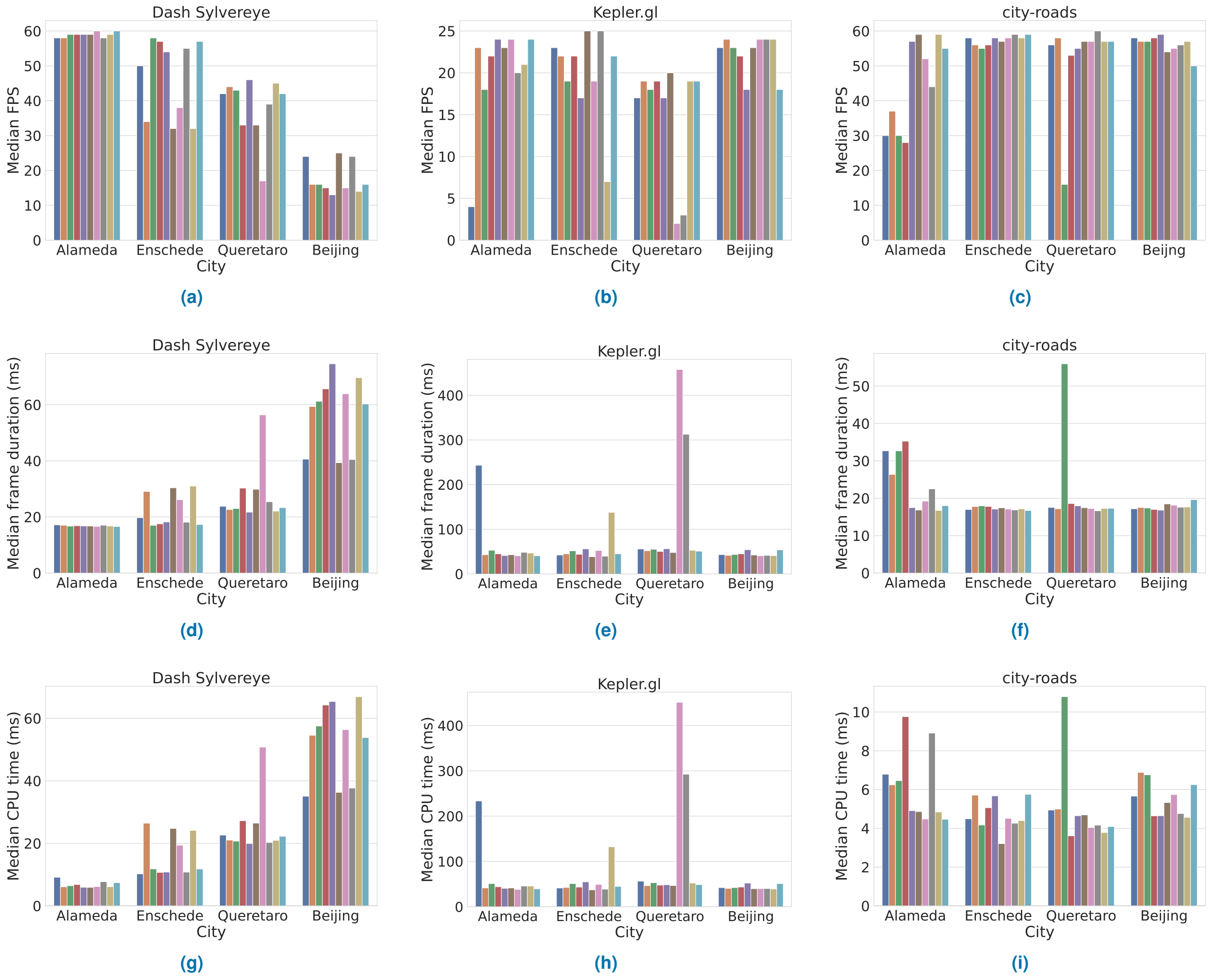


FIGURE 9. Median frame FPS, duration, and CPU time shown by Dash Sylvereye, Kepler.gl, and city-roads on each experiment and each city. Cities are sorted from smaller to bigger from left to right.

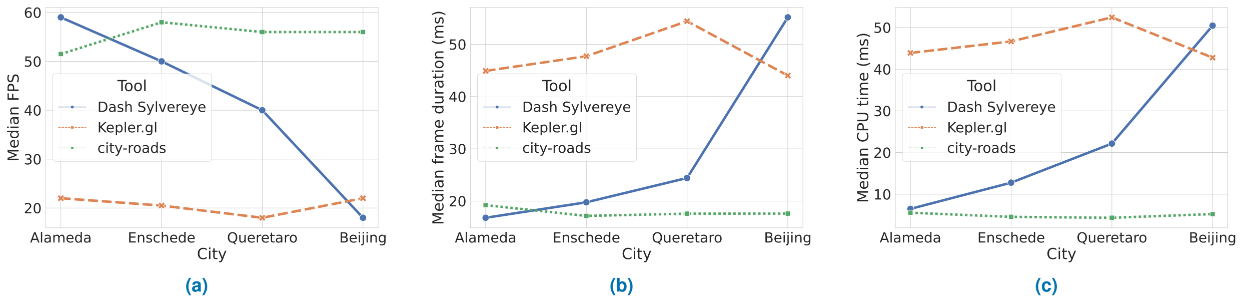


FIGURE 10. Dash Sylvereye's median frame FPS, duration, and CPU time shown by Dash Sylvereye, Kepler.gl, and city-roads when merging all experiments for each city. Cities are sorted from smaller to bigger from left to right.

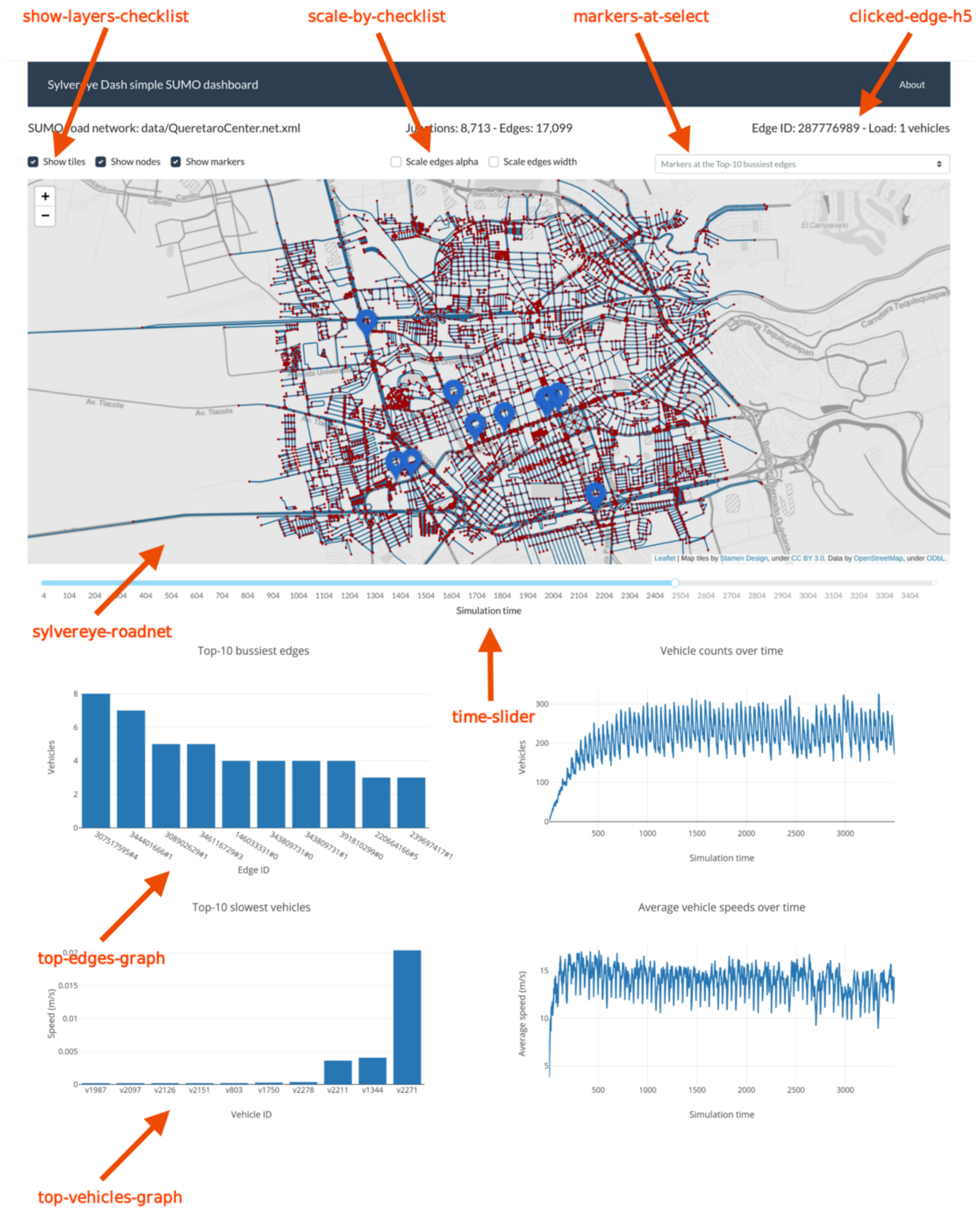


FIGURE 11. Layout of the dashboard made with Dash and the Dash Sylvereye library for analyzing a SUMO traffic simulation. Labels in orange are the Dash component identifiers referred to in Fig. 12.

urban traffic simulator [4], a well-known simulator in the field of urban analysis.

The purpose of this section is twofold. Firstly, we intend to better illustrate the usefulness of Dash Sylvereye in assisting a traffic analyst to observe how traffic bottlenecks build up with time in a busy transportation area made of thousands of street roads and junctions through a dashboard visualization centered around Dash Sylvereye. Secondly, we intend to offer the reader more details about how the Dash Slyvereye component can be integrated into a non-trivial dashboard by coordinating it with other charts and UI controls for multivariate data visualization.

A. STREET NETWORK RETRIEVAL AND SIMULATION

A street network from the center of the metropolitan area of Queretaro City, Mexico, was manually selected and downloaded in OSM format by taking advantage of the export features of the OSM website. The resulting street network had 8,713 nodes and 17,099 edges. The OSM network was converted to SUMO's XML format by using SUMO's netconvert tool. Finally, the SUMO XML network was converted to Dash Sylvereye's list-of-dictionaries format with the help of the Sumolib Python library[36].

To create synthetic vehicle trip data, a simulation was run by using SUMO as follows: random trips for vehicles

[36]https://sumo.dlr.de/docs/Tools/Sumolib.html

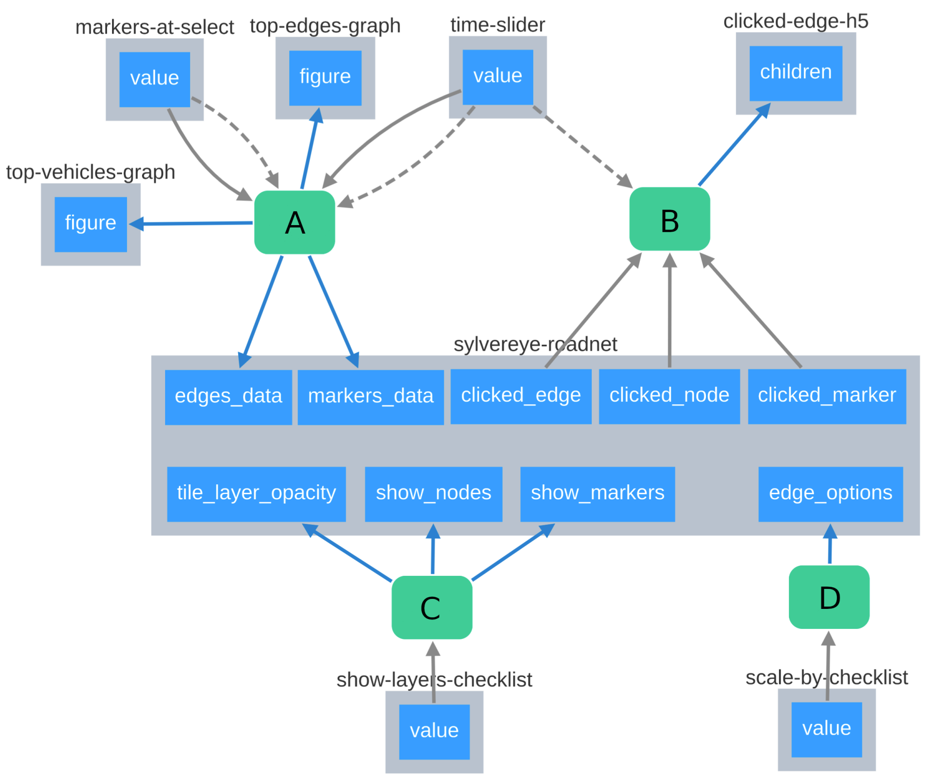


FIGURE 12. Callback graph of the SUMO simulator dashboard example, as generated by the Dash Dev Tools. Gray boxes represent Dash components. Labels on top of gray boxes are the Dash component identifiers. Green rounded boxes represent callback functions. Blue boxes represent the input and output properties. Solid gray arrows pointing to green rounded boxes come from input callback parameters. Solid blue arrows going out of green rounded boxes point to output callback parameters. Dashed gray arrows pointing to green circles represent states.

were generated with SUMO's `randomTrips.py` script. A SUMO simulation was run for 3,500 timesteps with the `--fcd-output` flag to save the Floating Car Data (FCD) of all timesteps in XML format. The produced FCD data was then processed to get CSV files that could be conveniently imported into the Dash dashboard application. CSV files included the vehicle count for edges at each timestep, the speed of vehicles at each timestep, total vehicle counts for each timestep, and average vehicle speed at each timestep.

B. LAYOUT DESIGN

Recall from Section I that a Dash dashboard application is composed of two parts: 1) the layout which describes what the application looks like and 2) the callbacks that define the interactivity of the application. Fig. 11 shows a screenshot of the resulting dashboard layout.

The dashboard includes a Dash Sylvereye visualization as its main element (`sylvereye-roadnet`). The user can select which layers are visible through the `show-layers-checklist` checklist. Markers are displayed at either the middle of edges with the highest vehicle counts or atop the slowest vehicles, depending on the option selected by the user in the `markers-at-select` selection list.

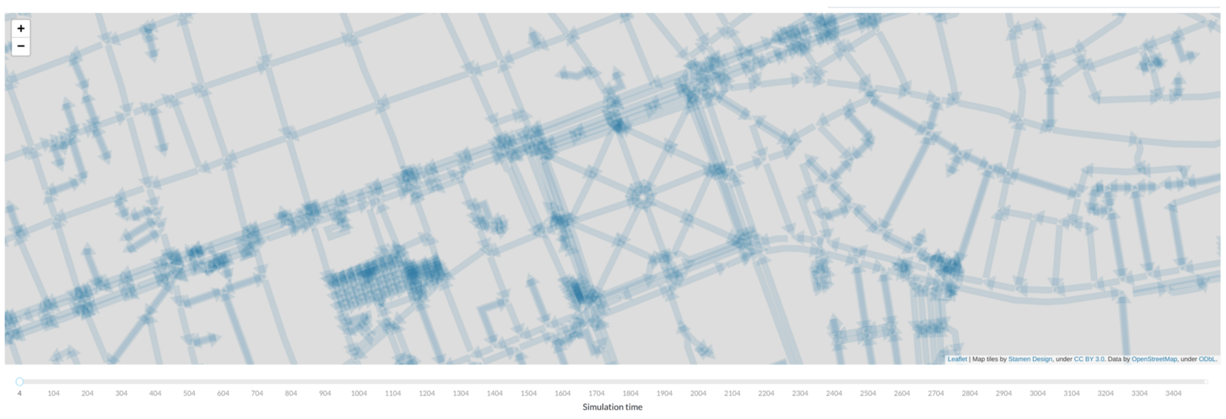
The user can also select which visual attributes of edges (transparency, width) to scale in proportion to the edge vehicle count through the `scale-by-checklist` checklist. The dashboard shows a slider to allow the user to select the desired simulation time to display (`time-slider`). When the user changes the simulation time, a callback is triggered to update:

- The network edges, width, and transparency.
- The position and the popup texts of markers.
- A bar plot of the top-10 edges with the highest vehicle count in the network (`top-edges-graph`)
- A bar plot of the speed for the top-10 slowest vehicles (`top-vehicles-graph`)

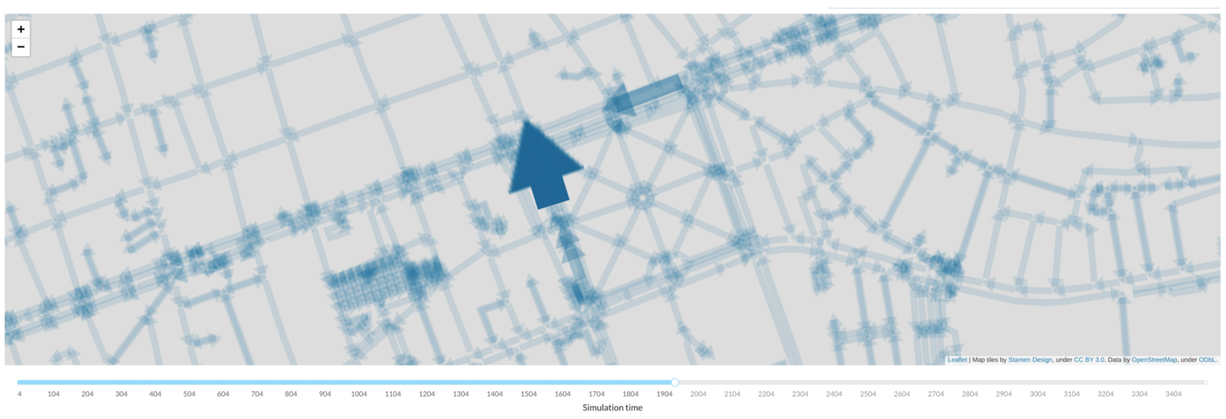
When the user clicks either an edge, a node, or a marker, data about the clicked element is shown in the label at the top-right corner (`clicked-edge-h5`). Finally, the dashboard also shows a line plot showing the vehicle count over time and a line plot showing the average vehicle speed over time. However, these two plots are static in the sense that they do not need to change as a result of the interaction of the user with the dashboard.

C. CALLBACK DESIGN

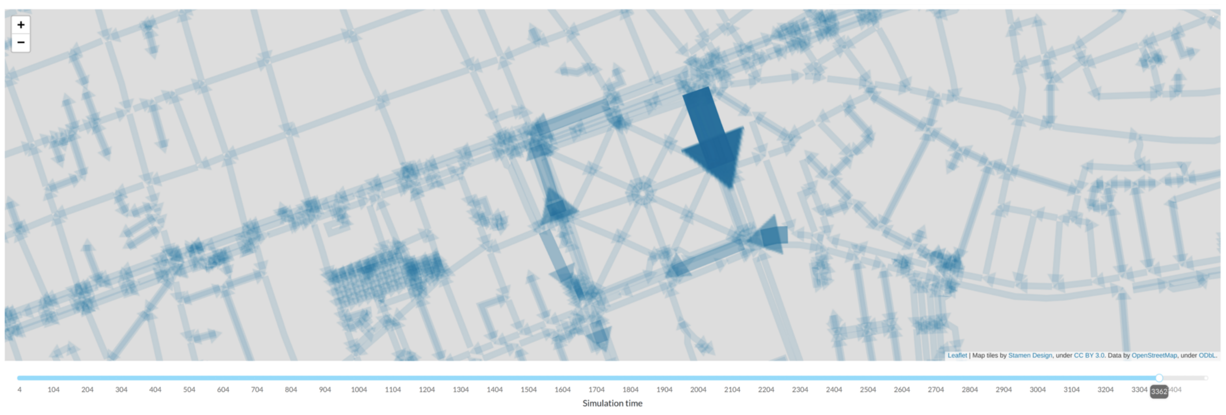
Fig. 12 shows the callback graph of the SUMO simulator dashboard example, as generated by the Dash Dev Tools. The application contains four main callbacks, callbacks A



(a)



(b)



(c)

FIGURE 13. Radiography-like visualizations centered at the "Alameda Hidalgo" park at three arbitrarily selected but consecutive simulation time steps. The visualization style was configured through the GUI components of the SUMO traffic dashboard developed with Dash and the Dash Sylvereye library. Note that the large blue arrows are street roads rendered by Dash Sylvereye (images were not edited).

to D, which together define the interactivity of the whole application.

Callback A triggers when either:

1) The value of `markers-at-select` changes its value, in which case the callback outputs a new set of markers.
2) The `time-slider` changes its value, in which case the callback updates the markers and both bar plots `top-vehicles-graph` and `top-edges-graph` to reflect the data at the new time step.

Callback B triggers when any of the click attributes of the Dash Sylvereye component changes its value as the result of a user clicking on a node, an edge, or a marker. The callback updates the label `clicked-edge-h5` with info about the clicked element.

Callback C triggers when `show-layers-checklist` changes its value because the user selected a different set of layers to show. The callback updates the show/hide properties of the Dash Sylvereye component accordingly.

Finally, callback D triggers when the component `scale-by-checklist` changes its value because the user selected different scale options. The callback updates the `edge_options` attribute of the Dash Sylvereye component accordingly to update the edge's alpha and width style methods.

D. VISUALIZATION INSIGHTS

Fig. 13 shows screenshots of the Dash Sylvereye component at three arbitrarily selected simulation time steps. The tilemap is centered at the "Alameda Hidalgo" park, a centric place where traffic bottlenecks build up in real life. Edges transparency and width were scaled to the vehicle count by checking the corresponding checkboxes in the dashboard. The map tile layer, as well as the nodes and markers layers, were hidden by unchecking the corresponding checkboxes. The result was a radiography-like visualization of the vehicle traffic. The "radiography" in Fig. 13 clearly shows that, even with random trips, vehicle traffic builds up on the main street roads surrounding the park.

VIII. CONCLUSION

This paper presented Dash Sylvereye, a new Python library for generating web-based visualizations of large street networks, delivered as a component for the widely-used Dash framework. To the best of our knowledge, Dash Sylvereye is the first tool written for Python that generates street network visualizations atop web tile maps that supports programmable user interactivity, that is designed as a component of a dashboard framework from the ground up, and that supports WebGL. Dash Sylvereye can be combined with other Dash UI and chart components to enable the development of interactive dashboard visualizations around street network data.

We showed that Dash Sylvereye can offer fast response speeds (close to 60 FPS) for street networks with thousands of edges. We also found Dash Sylvereye to be competitive when compared to the state-of-the-art visualization libraries Kepler.gl and city-roads for road networks with dozens of thousands of nodes and edges. With the help of a dashboard application example, we explored how Dash Sylvereye can be utilized as a convenient tool for interactively analyzing multivariate traffic data.

Visualization generation time is an important factor that impacts the experience of the end-user. Even with WebGL acceleration, we have observed that the visualization first drawing and redrawing of very large graphs in Dash Sylvereye may take non-negligible time on a commodity system, an overhead not present in other libraries like Kepler.gl and city-roads. This overhead includes the time needed for the generation of the graphics (sprites and polygons) of the street network and the computation of hit polygons for edge click detection. Future work includes methodologically assessing visualization generation times on commodity computers and evaluating optimization options.

Similar to other web-based visualization tools, one of Dash Sylvereye's main drawbacks is that the size of a street network the library can handle is limited by the system's physical memory and the GPU memory capacity. In this regard, an interesting research venue is to study efficient graph coarsening algorithms for edge bundling that 1) allow the tool to handle very large networks and 2) help the researcher's cognitive process of making sense of such complex structures.

We plan to release Dash Sylvereye under an open-source license, enabling anyone to use it for their specific street network visualization needs.

REFERENCES

[1] T. Anderson and S. Dragićević, "Complex spatial networks: Theory and geospatial applications", Geography Compass, vol. 14, no. 9, 2020. Available: 10.1111/gec3.12502.

[2] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An Open Source Software for Exploring and Manipulating Networks", ICWSM, vol. 3, no. 1, Mar. 2009.

[3] S. Batt, O. Harmon and P. Tomolonis, "Learning Tableau: A Data Visualization Tool", SSRN Electronic Journal, 2019. Available: 10.2139/ssrn.3438993.

[4] M. Behrisch, L. Bieker, J. Erdmann and D. Krajzewicz, "SUMO - Simulation of Urban MObility - an Overview", in The Third International Conference on Advances in System Simulation, Barcelona, Spain, 2011, pp. 55-60.

[5] G. Boeing, "OSMnx: A Python package to work with graph-theoretic OpenStreetMap street networks", The Journal of Open Source Software, vol. 2, no. 12, p. 215, 2017. Available: 10.21105/joss.00215.

[6] G. Boeing, "OSMNX: New Methods for Acquiring, Constructing, Analyzing, and Visualizing Complex Street Networks", SSRN Electronic Journal, 2016. Available: 10.2139/ssrn.2865501.

[7] D. Fang, M. Keezer, J. Williams, K. Kulkarni, R. Pienta and D. Horng Chau, "Carina: Interactive Million-Node Graph Visualization using Web Browser Technologies", in 26th International Conference on World Wide Web, 2017, pp. 775–776.

[8] T. Fujiwara, J. Zhao, K. Ma, F. Chen and K. Ma, "A Visual Analytics Framework for Contrastive Network Analysis", in 2020 IEEE Conference on Visual Analytics Science and Technology (VAST), 2020, pp. 48-59.

[9] S. Gray, O. O'Brien and S. Hügel, "Collecting and Visualizing Real-Time Urban Data through City Dashboards", Built Environment, vol. 42, no. 3, pp. 498-509, 2016. Available: 10.2148/benv.42.3.498.

[10] S. Hadlak, H. Schumann and H. Schulz, "A Survey of Multi-faceted Graph Visualization", in Eurographics Conference on Visualization (EuroVis) - STARs, 2015, pp. 1-20.
[11] M. Haklay and P. Weber, "OpenStreetMap: User-Generated Street Maps", IEEE Pervasive Computing, vol. 7, no. 4, pp. 12-18, 2008. Available: 10.1109/mprv.2008.80.
[12] D. Han, J. Pan, X. Zhao and W. Chen, "NetV.js: A web-based library for high-efficiency visualization of large-scale graphs and networks", Visual Informatics, vol. 5, no. 1, pp. 61-66, 2021. Available: 10.1016/j.visinf.2021.01.002.
[13] B. Hayes, "Usage of Programming Languages by Data Scientists: Python Grows while R Weakens", Businessoverbroadway.com, 2020. [Online]. Available: https://businessoverbroadway.com/2020/06/29/usage-of-programming-languages-by-data-scientists-python-grows-while-r-weakens/. [Accessed: 30-Apr-2021].
[14] X. Huang, Y. Zhao, C. Ma, J. Yang, X. Ye and C. Zhang, "TrajGraph: A Graph-Based Visual Analytics Approach to Studying Urban Network Centralities Using Taxi Trajectory Data", IEEE Transactions on Visualization and Computer Graphics, vol. 22, no. 1, pp. 160-169, 2016. Available: 10.1109/tvcg.2015.2467771.
[15] C. Jing, M. Du, S. Li and S. Liu, "Geospatial Dashboards for Monitoring Smart City Performance", Sustainability, vol. 11, no. 20, p. 5648, 2019. Available: 10.3390/su11205648.
[16] M. Kuperberg, M. Bowman, R. Manton and A. Peacock, A guide to computer animation. Oxford, U.K.: Focal Press, 2002.
[17] S. Li et al., "Argo Lite: Open-Source Interactive Graph Exploration and Visualization in Browsers", in 29th ACM International Conference on Information & Knowledge Management, 2020, pp. 3071–3076.
[18] R. Matheus, M. Janssen and D. Maheshwari, "Data science empowering the public: Data-driven dashboards for transparent and accountable decision-making in smart cities", Government Information Quarterly, vol. 37, no. 3, p. 101284, 2020. Available: 10.1016/j.giq.2018.01.006.
[19] F. Mwalongo, M. Krone, G. Reina and T. Ertl, "State-of-the-Art Report in Web-based Visualization", Computer Graphics Forum, vol. 35, no. 3, pp. 553-575, 2016. Available: 10.1111/cgf.12929.
[20] A. Sakib, S. Ismail, H. Sarkan, A. Azmi and O. Yusop, "Analyzing Traffic Accident and Casualty Trend Using Data Visualization", in IRICT 2018: Recent Trends in Data Science and Soft Computing, 2018, pp. 84-94.
[21] A. Sarikaya, M. Correll, L. Bartram, M. Tory and D. Fisher, "What Do We Talk About When We Talk About Dashboards?", IEEE Transactions on Visualization and Computer Graphics, vol. 25, no. 1, pp. 682-692, 2019. Available: 10.1109/tvcg.2018.2864903
[22] A. Saska et al., "ccNetViz: a WebGL-based JavaScript library for visualization of large networks", Bioinformatics, vol. 36, no. 16, pp. 4527-4529, 2020. Available: 10.1093/bioinformatics/btaa559.
[23] A. Schoedon, M. Trapp, H. Hollburg, D. Gerber and J. Döllner, "Web-based Visualization of Transportation Networks for Mobility Analytics", in 12th International Symposium on Visual Information Communication and Interaction, 2019, pp. 1-5.
[24] A. Sevtsuk and M. Mekonnen, "Urban network analysis. A new toolbox for ArcGIS", Revue Internationale de Géomatique, vol. 22, no. 2, pp. 287-305, 2012. Available: 10.3166/rig.22.287-305.
[25] P. Shannon, "Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks", Genome Research, vol. 13, no. 11, pp. 2498-2504, 2003. Available: 10.1101/gr.1239303.
[26] Y. Zheng, W. Wu, Y. Chen, H. Qu and L. Ni, "Visual Analytics in Urban Computing: An Overview", IEEE Transactions on Big Data, vol. 2, no. 3, pp. 276-296, 2016. Available: 10.1109/tbdata.2016.2586447.

ALBERTO GARCIA-ROBLEDO holds a MSc. and a Ph.D. in Computer Science from the Center for Research and Advanced Studies of the National Polytechnic Institute (Mexico). He was a tech lead at the Geospatial Data Center of the Massachusetts Institute of Technology (US). Currently, he is a Conacyt Research Fellow at the Center for Research in Geography and Geomatics (Mexico). His current research interests include HPC, Big Data, Graph Analytics, and Visual Analytics.

MAHBOOBEH ZANGIABADY is a lecturer at the Design and Analysis of Communication Systems (DACS) group at the University of Twente. She holds a Ph.D. in computer science from the Centre for Research and Advanced Studies of the National Polytechnic Institute. Her research interests cover network virtualization, QoS, resource management, machine learning, and Network Functions Virtualization, Software-Defined Networks (NFV/SDN).