NetV.js: A Large-Scale Graph Visualization Library

Dongming Han, Jiacheng Pan, Xiaodong Zhao[a,b], Wei Chen[a,*]

[a]State Key Lab of CAD&CG, Zhejiang University, Hangzhou, Zhejiang, China
[b]Zhejiang Lab, hangzhou, zhejiang, China

Abstract

Network visualization plays an important role in many fields, such as social media networks, protein-protein-interaction networks, and traffic networks. In the meantime, a dozen of visualization designer tools and programming toolkits are widely used in implementing network visualization applications. With network data growth, exiting tools can not help users rapidly and efficiently construct large-scale network visualization. In this paper, we present NetV.js, an open-sourced, JavaScript-based, WebGL-based library that supports the rapid large-scale network rendering, interaction, and visualization.

Keywords: Network visualization, Web-based visualization

1. Introduction

Visualization of graph data plays an important role in many fields, such as showing fraud transactions in financial data analysis [1], exploring information propagation in social media network [2], and visualizing protein-protein-interaction in biological network [3]. A series of visualization authoring tools have been developed to facilitate visualization generation. Notably, the proposal of d3.js [4] reduces the difficulty of web-based visualization authoring and enriches the visualization community. Thus, a great part of graph visualization tools are also web-based. Node-link diagrams are widely used among many graph visualization solutions, because they reveal topology and connectivities [5].

The node-link diagrams visualization has its speciality. For example, one link in a node-link diagram connects two nodes. User only need to control the nodes' positions and related links change their positions correspondingly; node-link diagrams strongly rely on layout algorithms; interactions on nodes such as clicking and dragging are frequently needed. Developers need to pay efforts to deal with these specialities using general visualization authoring tools such as d3.js [4], p5 [6], and stardust [7]. For example, when a developer uses d3.js to create a node-link diagram, s/he needs to map data elements to graphical marks using the data-drive-documents idea. The developer directly handles the visual elements and there is no hook/handler for s/he to handle the underlying graph model. It may lead to bugs. Some graph visualization authoring tools such as Cytoscape.js [8] and Sigma.js [9] support graph visualization efficiently. Compared to general visualization tools, tools for graph visualization encapsulate related interfaces of graph visualization, hide some unrelated interfaces (e.g. controlling the position of a link) for developers, and expose some graph-related interfaces (e.g. accessing neighborhoods of a node). They improve the usability through full-fill requirements of node-link diagrams.

With the growth of data scale, graph visualization authoring tools need to handle more graphical marks (e.g. nodes and links). However, most of existing tools are unable to handle a large number of graphical marks. According to our experiments in Section 4, existing graph visualization authoring tools have delays in rendering large-scale graph data (with more than ...). It reduces the user accessibility.

To best of our knowledge, no existing tool can address developer usability and user accessibility in the same time. We explored the design requirements of the node-link diagram visualization, designed and developed NetV.js, a web-based high-performance node-link diagram visualization library. It improves developer usability through a serials of node-link diagram related functions and interfaces and increases user accessibility by utilizing the high-performance rendering ability of the GPU. We also evaluated the usability and accessibility of NetV.js through several comparative experiments. NetV.js is now open-sourced (`https://netv.zjuvag.org/`) for developers to access and contribute their own code.

2. Related Work

Many network grammars and frameworks have been provided to developers for designing network visualization applications. In the beginning, developers use conventional programming languages to construct network visualization applications, such as C Sharp, C++, Javascript, and Python. Developers need to have proficient programming skills and understand the implementation mechanism. In the meantime, they also spend a lot of time developing and debugging. To make it easier for developers to program and develop quickly, visualization grammars and frameworks give granular control of visual channels of visualizations such as D3.js [4], ECharts [10],

*Corresponding author
Email address: chenvis@zju.edu.cn (Wei Chen)

Vega [11], and Vega-Lite [12]. Developers can use more concise tools to construct visualizations.

However, when developers construct network visualizations, they need to carefully select complex and different API to construct a network, because these APIs are designed for full visualization rather than network visualization applications. Cytoscape.js [8], sigmajs [9] and Gephi [13] are used to construct network visualizations. They encapsulate a series of API which are elaborated for the network data. Developers can use them to construct network visualizations quickly. Moreover, large-scale data brings new challenges. These grammars and tools can only render thousands of elements. To address this issue, PixiJS [14], P5 [6] and Stardust [7] used GPU-based acceleration technology to render a large number of elements. However, these GPU-based tools are also not designed for network visualizations. They still have redundancy and complex API that have nothing to do with network visualizations. It leads to a decrease in rendering efficiency and an increase in learning costs.

Our NetV.js focus on large-scale network visualization. It uses a GPU-based rendering engine to support large-scale data and design-friendly concise programming interfaces for network visualization construction.

3. Design of NetV.js

3.1. Design Requirements of NetV.js

To explore the design space of NetV.js, we interviewed three graph visualization experts, investigated five graph visualization tools including Gephi [13], Cytoscape.js [8], SNAP [15], Sigma.js [9], and GraphViZ [16]. We summarized the following design requirements for high-performance node-link diagram visualization:

R1 An abstract graph model to manipulate graph data. In a node-link diagram, visual elements (graphical marks) are one-to-one corresponding to data elements (nodes and links). Developers only needs to manipulate graph data elements rather than to access visual elements. To simplify the manipulation of origin data and visualization, developers need an abstract graph model. Several features should be supported:

- **R1.1 The node-link connection**. The basic characteristic of graph is that each link connects two nodes. Developers only need to focus on modifiying the position of nodes and ignore the position of links when drawing a node-link diagram. The position of links should be automatically determined.
- **R1.2 The accessibility of neighbors**. Another characteristic of graph is the connections among one node to its neighbor nodes and neighbor links. We found that many graph visualization systems support highlighting neighbor nodes and links of the focused node. All three experts agreed with that the graph model needs to support a node to access its neighbors.
- **R1.3 Basic graph theory algorithms**. Graph algorithms play important roles in some senarios. For example, some importance measurements (e.g. node degree, node centrality) can be used to highlight POIs (points of interest) and shortest paths finding algorithms can help users to trace the association between two nodes.

R2 High frame rate for large-scale node-link diagrams. The frame rate can affect the user experience. The user interface will respond slowly or even stop responding with a low frame rate. Base on our interview with three experts, a rule of thumb for high-performance rendering is "*reaching 30 FPS (Frames Per Second) for rendering node-link diagrams with more than 100 thousand visual elements*".

R3 Basic styles for nodes and links. Developers often needs to encode information on node-link diagrams such as encoding degree with node size. Although circular nodes and straight links are popular in large-scale node-link diagrams, there are also nodes and links with diverse shapes and styles. The most common node shapes includes circles, rectangles and triangles and the most common link shapes include straight lines and curves.

R4 Layout algorithms. The dependence of node-link diagrams on layouts is self-evident. Thus, NetV.js should provide some basic layout algorithms. Because of the diversity of layout algorihtms, NetV.js should provide interfaces for developers to customize their own layout algorihmts.

R5 Custom labels . Labels show basic information of nodes and links. Although drawing labels for all elements can lead to visual clutter in large-scale node-link diagrams, developers may adopt some strategies to select a part of elements to draw their labels. NetV.js should provide interfaces for developers to draw custom labels.

R6 Basic interactions. Based on our interview with experts and investigation on graph visualization authoring systems, we found interactions of node-link diagrams can be decomposed into interactions on nodes, links, and the canvas. Mouse events like mouseover, mousemove, mouseout, mouseup and mousedown should be supported to construct more complex interactions such as dragging nodes. Selection interactions such as lasso should also be supported.

3.2. Design Details of NetV.js

To fullfill the proposed requirements, we designed and implemented NetV.js. NetV.js consists of three main parts: *Graph Model Manager*, *Rendering Engine*, and *Interaction Manager* (Figure 1). We isolate **R4**, **R5**, and selection interactions (**R6**) into three different plugins to reduce the code length of NetV.js.

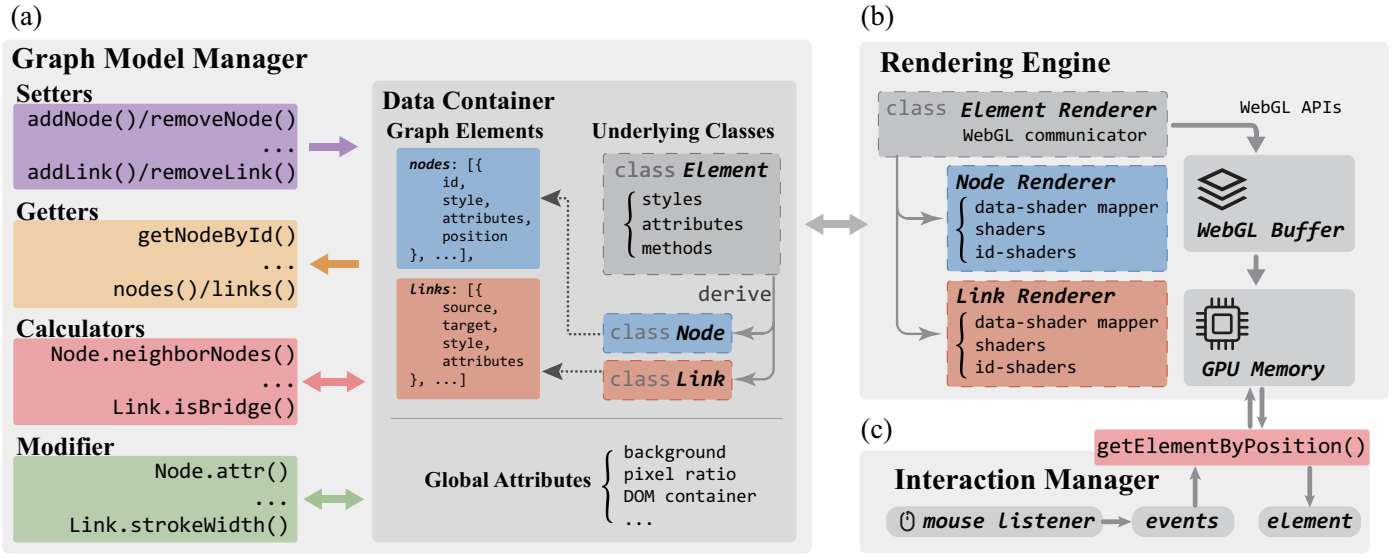

Figure 1: NetV.js designs: NetV.js consists of three main parts: (a) *Graph Model Manager*, (b) *Rendering Engine*, and (c) *Interaction Manager*.

3.2.1. Graph Model Manager

To help developers manipulate data and visualization in a unified way (**R1**), we designed *Graph Model Manager* in NetV.js (Figure 1(a)). The core of *Graph Model Manager* is an *Data Container*. It stores data elements including nodes and links. Each node contains a unique identifier (**`id`**). Its styles are stored in **`style`** attribute and coordinates are stored in **`position`** attribute. Other information is stored in **`attributes`**. Similarly, each link contains a **`source`** and a **`target`** which represent two nodes the link connects. Styles and other information are stored in **`style`** and **`attributes`** in the same. In NetV.js, nodes and links are stored in objects instanced from classes **`Node`** and **`Link`**. They are derived from **`Element`**.

We support developers to add, remove, modify, query and calculate data elements or visual elements. *Data Setters* are designed to add and remove data elements. *Data Getters* are designed to query elements (e.g. querying nodes by **`id`**s). Developers can modify the style and attributes of elements through *Data Modifier*. Several comman graph algorithms are supported in NetV.js. Because they are not strongly related to graph visualization, we only implemented several interfaces such as neighbor nodes/links querying (**R1.2**) and shortest paths finding (**R1.3**). Besides, *Graph Model Manager* can also manipulate global configurations such as backgrond color of the canvas and the DOM container.

3.2.2. Rendering Engine

To fullfill **R2**, *Rendering Engine* utilizes the high-performance rendering ability of GPU. The workflow of *Rendering Engine* consists of: 1) mapping styles and positions of elements into WebGL shader variables; 2) loading corresponding data into the WebGL buffer using WebGL APIs; 3) transfering data in the buffer into GPU memory and than drawing elements on the screen. In this way, *Rendering Engine* transfers data to the CPU memory of the browser, and the GPU memory in the end.

We adapted a passive buffer modifying strategy: when developers modify styles of elements, *Graph Model Manager* will not call *Rendering Engine* to modify the WebGL buffer but stores the modified elements into a cache pool. When developers try to refresh the canvas, *Rendering Engine* will traverse elements in the cache pool to get related attributes and cover their corresponding buffer content. The relative strategy is to forwardly change the WebGL buffer when developers modify styles of elements. It is more intuitive because the data flows forward. But every time one element is modified, *Graph Model Manager* will call the related interface of *Rendering Engine* to change the buffer. It deepens the call stack and reduces the efficiency of rendering.

We forbid developers to modify the position of links directly (**R1.1**), but links will be cached in to the pool when developers modify the position of their connected nodes. *Rendering Engine* will update the WebGL buffer of links when the canvas is refreshed.

Rendering Engine supports rendering several basic styles of nodes and links (**R3**). Nodes can be rendered with four different shapes: circles, rectangles, triangles, and crosses. Developers can control the radius (**`r`**) of circular nodes , the **`width`** and the **`height`** of rectanglar nodes, the verteces (**`vertexAlpha`**, **`vertexBeta`**, and **`vertexGamma`**) of triangular nodes , and the **`thickness`** and the **`size`** of cross nodes. Developers can set fill color (**`fill`**), the border color (**`strokeColor`**) and the border width (**`strokeWidth`**) of all kinds of nodes. Two different links are supported (the straight line and the curve). Developers can control the **`curveness`** of curve edges. Both two kinds of links support setting the color (**`strokeColor`**), the width (**`strokeWidth`**), and the dash array (**`dashInterval`**).

3.2.3. Interaction Manager

Interaction Manager provides a seriels of basic interactions (**R6**). NetV.js listens several mouse events (mouseover, mouseout, mousemove, mousedown, mouseup and mousewheel) on

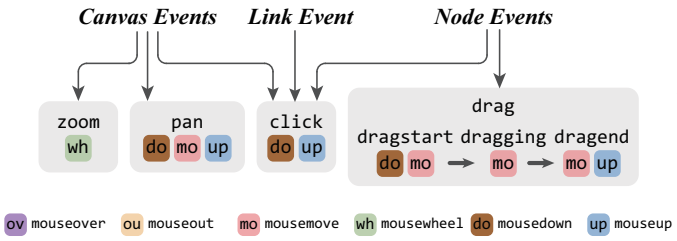

Figure 2: NetV.js interactions decomposition. Canvas events include zoom, pan and click. Link event contains only click. And node events include click and drag.

the canvas. All these events are supported on nodes, links and the canvas. We implemented several complex interactions by combining different events. For example, NetV.js supports the zoom+pan interaction by listening and combining mousewheel, mousedown, mousemove, and mouseup. Node dragging can be decomposed into several parts: dragstart, dragging, and dragend. When users press the mouse and then move the mouse, dragstart will be triggered. Then dragging is triggered. When users release the mouse, dragend and mouseup are triggered one by one. Figure 2 shows events on nodes, links and the canvas.

To determine which element is interacted by the mouse, *Rendering Engine* generates a hidden canvas which encodes the unique identifiers of elements into their corresponding pixels. Developers can use the interface **`getElementByPosition()`** to get the identifier encoded on the mouse-focused pixel so that the element on the mouse-focused pixel can be gathered by developers immediately.

3.2.4. Layout Plugin

Visualizing a node-link diagram relys on a layout algorithm to give the relative position of nodes. Different layout algorithms have different interfaces, our plugin intends to design a set of unified interfaces to facilate developer calling (**R4**):

- **`start()`**: calling **`start()`** interface brings the algorithm into processing. To improve the frontend performance, NetV.js creates another thread (using asynchronous timer) to calculate the layout result.
- **`stop()`**: calling **`stop()`** interface stops the layout thread immediately.
- **`onTick()`**: calling **`onTick()`** enables developers to visit the intermediate status of the layout algorithm with its callback. Some layout algorithms do not provide intermediate status. NetV.js generates intermediate status using interpolation functions. It make the layout process more smooth.
- **`onStop()`**: calling **`onStop()`** raises the stopping status to developers and returns the layout result. It will be triggered when the algorithm finishes or developers call the **`stop()`** interface.

To resolve **R4**, we implemented a ratial tree layout and a force-directed layout based on these interfaces. In the future, we will increase more layout algorithms to facilate visualization. Developers can also implemente their own layout algorithms using our interfaces.

3.2.5. Label Plugin

We implemented a label drawing plugin to facilate developers creating labels for elements (**R5**). When drawing a large-scale node-link diagram, drawing labels for all elements will cause serious visual clutter, so we assume that developers will only render labels for a few elements, for example, selecting some POI nodes or areas to draw labels. So we choosed SVG as label rendering tools to improve the expansibility. Besides the default textual label, developers can also customize labels by creating label templates.

3.2.6. Selection Plugin

Besides interactions mentioned in Section 3.2.3, we implemented a selection plugin to facilate elements selection. Common selection interactions include the box selection and the lasso selection. Because of the flexibility of the lasso selection, we implemente it for demonstration. Developers can get the selected structure (including nodes and links) in the callback function.

4. Experiment

We conducted the frame per second (FPS) experiment[1] to test the rendering performance of our NetV.js with other popular tools and libraries which are supports network rendering, including D3-SVG, D3-Canvas, Cytoscape.js, Sigma.js (WebGL), and Stardust.js. In particular, NetV.js, Sigma.js, and Stardust.js use WenGL to render data. To simulate real-world network data, we set the density of the network as 20, it means the ratio of the number of edges to the number of nodes is 1 to 20. The display refresh rate is 144Hz, the GPU is GTX 1060 with 6G.

From the line chart (Figure 3), NetV.js, Stardust.js, and D3-Canvas can render around 100,000 elements. Our NetV.js can render more than 1 million elements with FPS greater than 1.

5. Conclusion

This paper presents NetV.js, an open-sourced, JavaScript-based, WebGL-based library that supports the rapid large-scale network rendering, interaction, and visualization. In the future, we plan to extend NetV.js to support heterogeneous networks. We also plan to explore more visualization components and visual analysis algorithms for analyzing network data.

[1] https://github.com/ZJUVAG/NetV.js/tree/benchmarks/benchmarks

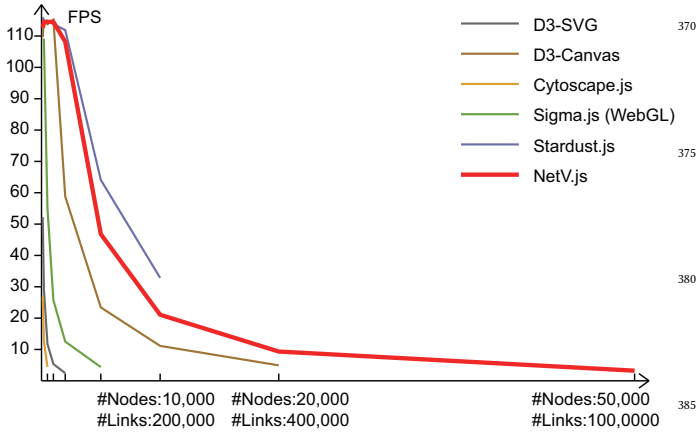

Figure 3: The Frame per Second (FPS) experiment.

References

[1] W. Chen, F. Guo, D. Han, J. Pan, X. Nie, J. Xia, X. Zhang, Structure-based suggestive exploration: A new approach for effective exploration of large networks, IEEE Trans. Vis. Comput. Graph. 25 (1) (2019) 555–565. doi:10.1109/TVCG.2018.2865139.
URL https://doi.org/10.1109/TVCG.2018.2865139

[2] M. A. Smith, B. Shneiderman, N. Milic-Frayling, E. M. Rodrigues, V. Barash, C. Dunne, T. Capone, A. Perer, E. Gleave, Analyzing (social media) networks with nodexl, in: J. M. Carroll (Ed.), Proceedings of the Fourth International Conference on Communities and Technologies, C&T 2009, University Park, PA, USA, June 25-27, 2009, ACM, 2009, pp. 255–264. doi:10.1145/1556460.1556497.
URL https://doi.org/10.1145/1556460.1556497

[3] N. T. Doncheva, Y. Assenov, F. S. Domingues, M. Albrecht, Topological analysis and interactive visualization of biological networks and protein structures, Nature protocols 7 (4) (2012) 670.

[4] M. Bostock, V. Ogievetsky, J. Heer, D^3 data-driven documents, IEEE Trans. Vis. Comput. Graph. 17 (12) (2011) 2301–2309. doi:10.1109/TVCG.2011.185.
URL https://doi.org/10.1109/TVCG.2011.185

[5] M. Ghoniem, J.-D. Fekete, P. Castagliola, A comparison of the readability of graphs using node-link and matrix-based representations, in: Proceedings of IEEE Symposium on Information Visualization, 2004, pp. 17–24. doi:10.1109/INFVIS.2004.1.

[6] J. K. Li, K. Ma, P5: portable progressive parallel processing pipelines for interactive data analysis and visualization, IEEE Trans. Vis. Comput. Graph. 26 (1) (2020) 1151–1160. doi:10.1109/TVCG.2019.2934537.
URL https://doi.org/10.1109/TVCG.2019.2934537

[7] D. Ren, B. Lee, T. Höllerer, Stardust: Accessible and transparent GPU support for information visualization rendering, Comput. Graph. Forum 36 (3) (2017) 179–188. doi:10.1111/cgf.13178.
URL https://doi.org/10.1111/cgf.13178

[8] M. Franz, C. T. Lopes, G. Huck, Y. Dong, S. O. Sümer, G. D. Bader, Cytoscape.js: a graph theory library for visualisation and analysis, Bioinform. 32 (2) (2016) 309–311. doi:10.1093/bioinformatics/btv557.
URL https://doi.org/10.1093/bioinformatics/btv557

[9] J. Coene, sigmajs: An R htmlwidget interface to the sigma.js visualization library, J. Open Source Softw. 3 (28) (2018) 814. doi:10.21105/joss.00814.
URL https://doi.org/10.21105/joss.00814

[10] D. Li, H. Mei, Y. Shen, S. Su, W. Zhang, J. Wang, M. Zu, W. Chen, Echarts: A declarative framework for rapid construction of web-based visualization, Vis. Informatics 2 (2) (2018) 136–146. doi:10.1016/j.visinf.2018.04.011.
URL https://doi.org/10.1016/j.visinf.2018.04.011

[11] A. Satyanarayan, R. Russell, J. Hoffswell, J. Heer, Reactive vega: A streaming dataflow architecture for declarative interactive visualization, IEEE Trans. Vis. Comput. Graph. 22 (1) (2016) 659–668. doi:10.1109/TVCG.2015.2467091.
URL https://doi.org/10.1109/TVCG.2015.2467091

[12] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, J. Heer, Vega-lite: A grammar of interactive graphics, IEEE Trans. Vis. Comput. Graph. 23 (1) (2017) 341–350. doi:10.1109/TVCG.2016.2599030.
URL https://doi.org/10.1109/TVCG.2016.2599030

[13] M. Bastian, S. Heymann, M. Jacomy, Gephi: An open source software for exploring and manipulating networks, in: E. Adar, M. Hurst, T. Finin, N. S. Glance, N. Nicolov, B. L. Tseng (Eds.), Proceedings of the Third International Conference on Weblogs and Social Media, ICWSM 2009, San Jose, California, USA, May 17-20, 2009, The AAAI Press, 2009.
URL http://aaai.org/ocs/index.php/ICWSM/09/paper/view/154

[14] C. G. I. Graphics, Learn pixi. js.

[15] J. Leskovec, R. Sosič, Snap: A general-purpose network analysis and graph-mining library, ACM Transactions on Intelligent Systems and Technology (TIST) 8 (1) (2016) 1.

[16] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, G. Woodhull, Graphviz and dynagraph – static and dynamic graph drawing tools, in: GRAPH DRAWING SOFTWARE, Springer-Verlag, 2003, pp. 127–148.