NetV.js: A Large-Scale Graph Visualization Library

Dongming Han, Jiacheng Pan, Rusheng Pan, Jiehui Zhou, 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

Network data wildly exists in the world. Network visualization plays an important role in many fields, such as visualizing fraud transactions in financial data analysis [1], exploring information propagation in social media network [2], and showing protein-protein-interaction in biological network [3]. As the development of modern browsers and open-sourced communities, a series of visualization generation tools and programming toolkits are employed to construct network visualization in a web page [4, 5]. In the meantime, as the data grows, attention has been paid to the large-scale network visualization [1]. A series of requirements and tasks requires users to explore and analyze large networks. A large-scale network visualization tool is necessary for users to rapidly and efficiently construct systems.

One main bottleneck of visualizing large-scale networks is the rendering performance. Conventional tools use DOM tree, SVG, or Canvas to construct graph visualization, such as D3.js [6], Cytoscape.js [7], and Echarts [8]. They can not handle a large number of elements. For example, the SVG performance test of D3.js indicates that rendering 2000 elements will cause a noticeable lack of smoothness (around 24 frames per second) [9]. For optimizing the effectiveness of rendering, D3.js supports Canvas in its fifth version. Echarts and Fabric.js also use Canvas as their rendering backend. However, the Canvas performance test shows that 10,000 elements will cause an obvious lack of smoothness [10]. Now, continuous research and tools focus on rendering data by using WebGL (GPU), such as Stardust.js [11], PixiJS [12] and P5 [13]. WebGL takes a remarkable efficiency improvement in rendering, but the challenge of WebGL programming difficulty also comes. Users need to master the knowledge of graphics pipeline and shader programming. The steer learning curve and complex API lead to the second challenge. It is difficult for users to construct network visualizations easily. Existing WebGL-based visualization generation tools have already encapsulated a series of API about bottom interfaces. And they aim to generate general visualization components. As a result, they are inevitable to design complex and diverse API for adapting the requirements of different visualizations. Complex API will hinder users who have no experience of visualization in constructing network visualization applications. Sometimes, users still need to write some shader related codes. In the meantime, the goal of generality needs a lot of logic code and complex data structure. The rendering and interaction performance will be greatly reduced in a specified visualization task due to the redundancy in design. It is still a difficult task for users to construct large-scale network visualization rapidly and efficiently.

To address performance and accessibility challenges, we designed and developed NetV.js, an open-sourced[1], JavaScript-based, WebGL-based library for rapidly and easily construct large-scale network visualization. NetV.js leverages GPU processing power to render large-scale network and provide rich build-in interactions to explore networks. At the same time, NetV.js own friendly and concise programming interfaces for developers to rapidly construct network visualization applications.

0.1. Network Grammars and Frameworks

Many network grammars and frameworks have been provided to developers for designing network visualization

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

[1] netv.zjuvag.org

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 [6], ECharts [8], Vega [14], and Vega-Lite [15]. 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 [7], sigmajs [16] and Gephi [17] 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 [12], P5 [13] and Stardust [11] 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.

0.2. GPU-based Visualization Rendering

1. NetV.js Design

NetV.js aims to help users rapidly and efficiently construct network visualization applications. Specifically, NetV.js consists of three parts: the core engine, plugins, and library interface (Figure 4). The core engine contains the data manager for maintaining nodes and links and the renderer for leveraging GPU processing power to render a large-scale network. The plugins are employed to increase and expand more requirements and functions. The library interface aims to help developers rapidly construct applications with friendly and concise APIs.

1.1. Core Engine

The core engine aims to render large-scale network based on WebGL and maintains the primary network information for rapidly editing operations. It consists of the data manager and the renderer.

1.1.1. Data Manager

The data manager supports a series of interface corresponding network structure. It is used to achieve nodes or links operations, including adding, deleting, searching, and editing. Node-links and link-nodes mapping tables are also supported for accelerating search and locate. At the same time, the data manager has a build-in network data set for developers to get started quickly.

1.1.2. Renderer

The renderer aims to render basic elements: nodes and links. It focuses on the efficiency of rendering massive data on the browser platform. As the highest performance graphics rendering API of browser platform, NetV.js uses WebGL as the bottom rendering. However, WebGL programming is still hard and complex. The WebGL API is encapsulate In order to reduce the program execution time as much as possible, the basic WebGL API is only encapsulated to meet the most basic data processing and rendering. Specifically, the renderer uses three strategies to improve rendering efficiency.

- **Batch**: The renderer contains a batch drawing element instance. Considering that most of the elements of the network are the same, but the location information is different. The renderer creates an instance of an element and draws it in the batch process to reduce the consuming time of the rendering process.

- **Shader**: The renderer uses Shader function to control the shape render. Each node's shape usually needs to be defined as a circle, square, ellipse, and so on. However, the complex shape will cause a serious effect on rendering performance. NetV.js exploits the powerful Shader function to define and render shape in GPU with batch processing.

- **Modify as needed**: The renderer supports manual rendering function for developers. The main attributes of elements, including positions, color, and texture, are stored into different buffers of GPU. When attributes of elements need to be Modified, the renderer can refresh corresponding buffers rather getting attributes from the data manager. And then, developers can refresh the network by supported function. The time consuming of getting attributes from the data manager can be omitted.

- **Element positioning**: The renderer has an elements positioning function for supporting elements search and interaction. For improving user interaction efficiency, the renderer uses the WebGL Texture to record the screen pixel position of each element. The time consuming of elements positioning has great improvement compared with index search and spatial index tree. The time complexity of the function is $O(1)$.

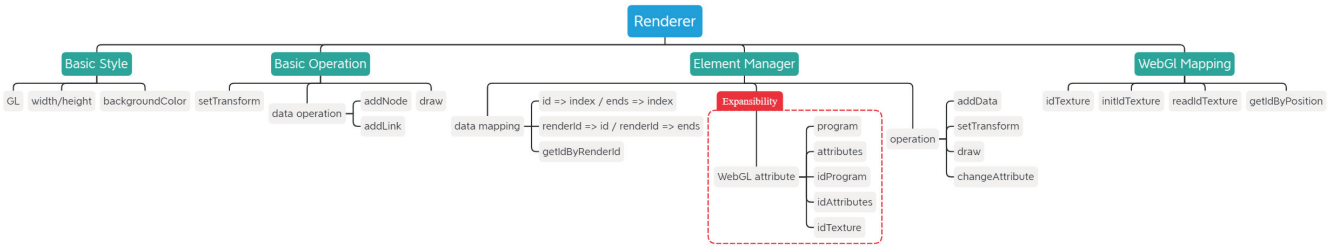

Figure 1: NetV.js designs: NetV.js consists of three parts: core engine, plugins, and library interface.

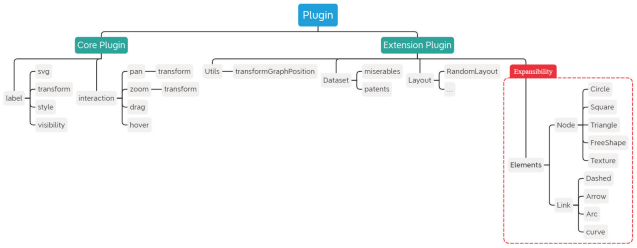

Figure 2: NetV.js designs: NetV.js consists of three parts: core engine, plugins, and library interface.

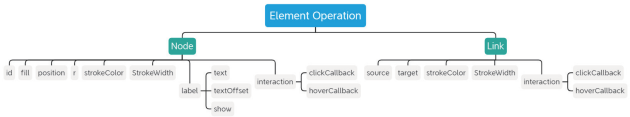

Figure 3: NetV.js designs: NetV.js consists of three parts: core engine, plugins, and library interface.

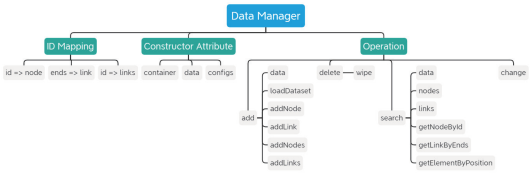

Figure 4: NetV.js designs: NetV.js consists of three parts: core engine, plugins, and library interface.

1.2. Plugin module

The goal of the plugin modular is to enhance future expansions. For supporting more features and requirements, NetV.js designs the plugin modular to employ new improvements or functions. At the same time, it can isolate the core modular from the rest. For now, the plugin module includes:

- **Interaction:** It supports binding interaction events and callback functions of elements. Interaction events contain basis events of edges and nodes such as Hover, MouseDown, Click, etc.
- **Layout:** It contains build-int layouts and allows users to use custom layouts.

1.3. Library Interface

The library interface aims to support concise and efficient API for users to build large-scale network visual analysis applications quickly. Users do not need to touch the underlying WebGL rendering programming. Humanized and simple API can be called to config data and render network. Specifically, it includes three parts:

- **Global:** It is used to define custom configs, such as the mount node, the default style of the canvas, and the default style of elements.
- **Element:** It includes the data adding of nodes of edges and the attributes changing of elements.
- **Plugin:** It aims to set up different plugins, such as layouts and interactions.

1.4. Examples

Users can use NetV.js to create network visualization such as designing network visualization, analyzing networks by layout, exploring large-scale networks, and so on. In this section, we show diverse network visualization examples to illustrate the expressiveness and usability of NetV.js.

1.4.1. Basic newtrok rendering

The Figure 5 shows a Basic network rendering. It can be regard as three parts: initialization, loading data, and rendering. The 'testData' illustrates the network data format. In this example, the initialization part hangs the canvas to the document 'main'.

```
const testData = {
    nodes: [
        {
            id: '0',
            x: 300,
            y: 100
        },
        {
            id: '1',
            x: 500,
            y: 100
        },
        {
            id: '2',
            x: 400,
            y: 400
        }
    ],
    links: [
        {
            source: '0',
            target: '2'
        },
        {
            source: '1',
            target: '2'
        }
    ]
}

const netv = new NetV({
    container: document.getElementById('main')
})
netv.data(testData)
netv.draw()
```

Figure 5: Basic network rendering.

1.4.2. Customized style

In network visualization, the most important function is to draw a network with different styles, such as color, stroke, radius, and position of elements. In the Figure 6, this example shows the setting of customized style by using NetV.js. Developers can customize the style of each element and set the default style in the initialization part. In particular, initialize configuration items are set in the 'configs'.

```
const testData = {
    nodes: [
        {
            id: '0',
            x: 300,
            y: 100,
            r: 5,
            fill: { r: 1, g: 0, b: 0, a: 1 },
            strokeColor: { r: 0, g: 1, b: 1, a: 0.3 }
        },
        {
            id: '1',
            x: 500,
            y: 100,
            fill: { r: 0, g: 1, b: 0, a: 1 },
            strokeColor: { r: 1, g: 0, b: 1, a: 0.3 }
        },
        {
            id: '2',
            x: 400,
            y: 400,
            fill: { r: 0, g: 0, b: 1, a: 1 },
            strokeColor: { r: 1, g: 1, b: 0, a: 0.3 }
        }
    ],
    links: [
        {
            source: '0',
            target: '2',
            strokeWidth: 4,
            strokeColor: { r: 1, g: 1, b: 0, a: 1 }
        },
        {
            source: '1',
            target: '2',
            strokeWidth: 8,
            strokeColor: { r: 1, g: 0, b: 1, a: 1 }
        }
    ]
}
```

```
const configs = {
    container: document.getElementById('main'),
    node: {
        r: 10,
        fill: { r: 1, g: 0, b: 0, a: 0.8 }
    },
    width: 800,
    height: 600,
    backgroundColor: { r: 0.95, g: 0.98, b: 0.98, a:
1 },
    nodeLimit: 1000
}
const netv = new NetV(configs)
netv.data(testData)
netv.draw()
```

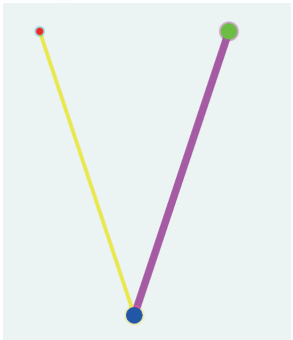

Figure 6: Customized style.

1.4.3. Build-in datasets

NetV.js supports build-in datasets for users to construct a network visualization (Figure 7) quickly. The build-in datasets also support the attribute and the position of each node in the network. Users can the radius and the color of nodes to encode different attributes of nodes.

```
const configs = {
    container: document.getElementById('main'),
    nodeLimit: 1e5,
    linkLimit: 1e7,
    node: {
        strokeWidth: 0
    },
    link: {
        strokeWidth: 0.5
    }
}
const netv = new NetV(configs)
const data = netv.Utils.transformGraphPosition(netv.l
oadDataset('patents'), 500, 400, 300)
const colorMap = {
    patent: { r: 102, g: 194, b: 165, a: 1 },
    inventor: { r: 252, g: 141, b: 98, a: 1 },
    assignee: { r: 141, g: 160, b: 203, a: 1 }
}
const radius = (x) => {
    const transformer = (n, k) => 0.5 * Math.max(3, k
*n**0.5)
    switch (x.type) {
        case 'patent':
            return transformer(x.numCitations, 0.15)
        case 'inventor':
            return transformer(x.numPatents, 0.3)
        case 'assignee':
            return transformer(x.numPatents, 0.1)
    }
}
data.nodes.forEach((node) => {
    const { r, g, b, a } = colorMap[node.type]
    node.fill = { r: r / 255, g: g / 255, b: b / 255,
 a }
    node.r = radius(node)
})
netv.data(data)
netv.draw()
```

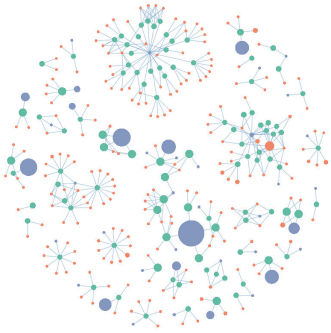

Figure 7: Build-in datasets.

1.4.4. Layout

NetV.js supports various network layout algorithms. Moreover, developers can implement by combining with external layouts such as D3.js (Figure 8 (a)), or by using NetV.js plugins (Figure 8 (b)). When combining with external layouts, NetV.js acts as a renderer to draw network by layout results. When using NetV.js layout plugins, it supports developers to controllers of all layout stages.

```
netv.data(data)
const simulation = d3.forceSimulation(data.nodes)
    .force(
        'link',
        d3.forceLink(data.links).id((d) => d.id)
    )
    .force(
        'collide',
        d3.forceCollide((d) => {
            return 2.4 * radius(d)
        })
    )
    .force('center', d3.forceCenter(width / 2, height
 / 2))
    .force('x', d3.forceX(width / 2).strength(0.13))
    .force('y', d3.forceY(height / 2).strength(0.13))

simulation.on('tick', () => {
    data.nodes.forEach((n) => {
        const node = netv.getNodeById(n.id)
        node.x(n.x)
        node.y(n.y)
    })
    netv.draw()
})
```

(a)

```
const layout = new NetV.Layouts.RandomLayout(netv)
layout.time(1000)
layout.onStart(() => {
    console.log('random layout start')
})
layout.onTick(() => {
    console.log('iteration...')
})
layout.onStop(() => {
    console.log('random layout done')
})
layout.start()
```

(b)

Figure 8: Layout. (a) Combining with D3.js. (b) Using NetV.js plugins.

1.4.5. Large-scale network

NetV.js aims to render large-scale networks. Figure 9 shows a large-scale network visualization result with 35,590 nodes, and 572,915 links. With the powerful WebGL performance, NetV.js maximum supports for drawing millions of elements.

1.4.6. Label

NetV.js supports label rendering with different drawing techniques such as SVG, Canvas, and WebGL. Figure 10 shows the network rendering with labels.

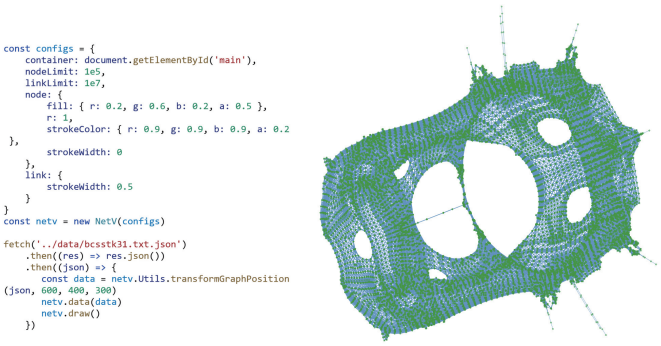

Figure 9: Large-scale network.

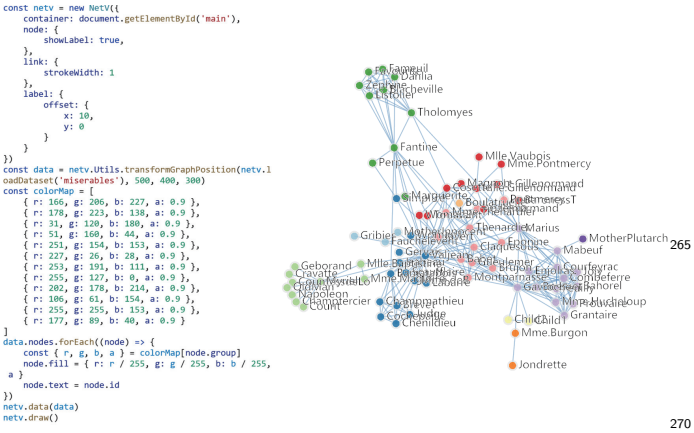

Figure 10: Label.

2. Experiment

We conducted the frame per second (FPS) experiment[2] 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 11), 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.

3. 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.

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

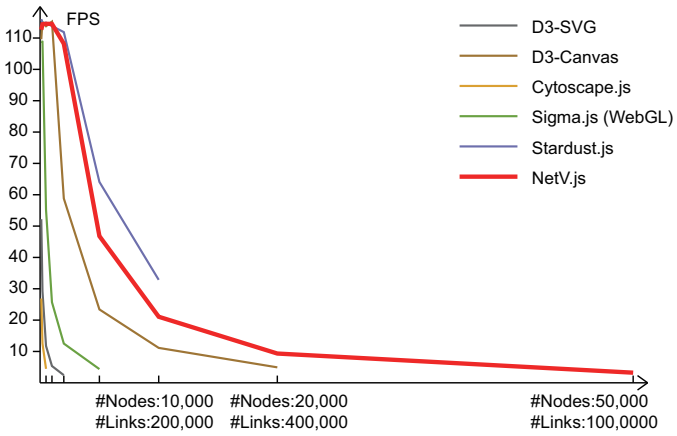

Figure 11: The Frame per Second (FPS) experiment.

References

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] A. Srinivasan, H. Park, A. Endert, R. C. Basole, Graphiti: Interactive specification of attribute-based edges for network modeling and visualization, IEEE Trans. Vis. Comput. Graph. 24 (1) (2018) 226–235. doi:10.1109/TVCG.2017.2744843.
URL https://doi.org/10.1109/TVCG.2017.2744843

[5] A. Bigelow, C. Nobre, M. Meyer, A. Lex, Origraph: Interactive network wrangling, in: R. Chang, D. A. Keim, R. Maciejewski (Eds.), 14th IEEE Conference on Visual Analytics Science and Technology, IEEE VAST 2019, Vancouver, BC, Canada, October 20-25, 2019, IEEE, 2019, pp. 81–92. doi:10.1109/VAST47406.2019.8986909.
URL https://doi.org/10.1109/VAST47406.2019.8986909

[6] 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

[7] 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

[8] 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
[9] M. Romero, Svg performance test, http://bl.ocks.org/mjromper/95fef29a83c43cb116c3/ Accessed June 9, 2020.
[10] Html5 canvas performance test, http://clockmaker.jp/labs/120202_html5_performance/canvas.html/ Accessed June 9, 2020.
[11] 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
[12] C. G. I. Graphics, Learn pixi. js.
[13] 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
[14] 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
[15] 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
[16] 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
[17] 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