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

1. Introduction

图数据在现实世界广泛存在。图数据的可视化在很多领域都有其重要作用，比如在金融数据可视分析中可视化欺诈交易，探索社交媒体网络中的信息传播，以及展示生物网络中的蛋白质互相作用等等。 为了提升用户和开发者构建可视化的效率，一系列可视化构建工具被提出。特别是d3的提出，降低了基于web的可视化构建的难度，丰富了可视化社区。因此，很多图数据可视化工具也基于web进行开发。其中，节点链接图是最为广泛使用的可视化形式。^{pan}

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].^{pan}

节点链接图可视化拥有其特点，比如每一条链接往往链接了两个节点，用户只需控制节点的位置，其链接的位置会相应变化；节点链接图依赖于布局算法，从图数据到可视化的映射，需要用户通过布局算法来处理节点的位置摆放；针对节点的交互（比如点击和拖拽）非常普遍。开发者在使用通用的可视化工具进行开发时，需要pay effort to 处理这些问题。比如在使用D3.js进行图数据可视化时，开发者需要使用其数据驱动文档的思想将对应的数据映射到对应的可视化元素上，开发者直接面向可视化元素进行开发，没有抽象的模型来支持他控制整个图数据，可能会导致编程错误的产生。 一些图可视化构建工具，比如xxxx，良好地支持了对图数据的可视化，相对于通用的可视化工具而言，他们通过解决节点链接图的特殊需求，封装图可视化的相关接口，隐藏一部分开发者不关心的接口（比如对边的位置控制），暴露一部分图可视化特性接口（对相邻节点的访问），来提升自身的易用性。^{pan}

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.^{pan}

随着数据规模的增长，图可视化也需要处理更多graphical marks（比如节点和链接）。然而，现有大部分图可视化工具难以处理有较多graphical marks。根据我们的实验，

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

它们在渲染较大规模数据集上存在延迟，这将会降低用户的accessibility。pan

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.pan

据我们所知，尚未存在一款工具，能够解决开发者易用性以及用户可访问性的问题。我们探索了图可视化的相关设计需求，设计并实现了NetV，一款基于web的高性能图可视化工具。其通过设计一系列图可视化相关的功能和接口来提高开发者易用性，并调用了GPU的高性能渲染能力来提高渲染效率以增加用户的可访问性。 通过和其他工具的对比实验，我们验证了NetV的accessibility和usability。我们对该工具进行了开源以便开发者访问和贡献代码。pan

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.pan

2. Related Work

2.1. Web-based Visualization Authoring Tools

2.2. Graph Visualization Authoring Tools

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], 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.

2.3. Network Grammars and Frameworks

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], 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.

2.4. GPU-based Visualization Rendering

3. NetV.js Design

3.1. Design Requirements of NetV.js

为了探索NetV.js的设计空间，我们采访了3个图可视化相关的专家，调研了一系列图可视化的工具，包括Gephi，Cytoscape.js，Sigma.js，GraphViZ，总结了如下高性能节点链接图可视化的设计需求:pan

To explore the design space of NetV.js, we interviewed 3 graph visualization experts, investigated 6 graph visualization tools including Gephi [13], Pajek [15], SNAP [16], Sigma.js [9], GraphViZ [17], and Cytoscape.js [8]. We summarized the following design requirements for high-performance node-link diagram visualization: pan

- **需要对拥有大量可视化元素的节点链接图提供高刷新率**: 根据我们对三位图可视化专家的采访，他们认为【对超过10万元素的大规模图有30fps以上的渲染速度】才能认为其对大规模节点链接图提供了高性能渲染的能力。pan
- **需要有抽象的图模型来帮助控制图可视化**: 为了简化开发者对于可视化的操作，NetV.js需要一个抽象图模型来控制图可视化而非直接控制图可视化的元素。该模型需要支持图的以下几点特性：pan
  - 链接关联节点: pan
  - 邻节点和邻接边的可访问性: pan
  - 基本图度量的计算: pan
  - 基本图论算法的支持: pan
- 支持不同节点和链接的样式:pan
- 需要提供多种布局功能和自定义布局插件：pan
- 需要提供基本的文本标签渲染和自定义标签：pan
- 需要提供基础交互模型：pan

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.

3.2. 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.

3.2.1. 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.

3.2.2. 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 $O(1)$.

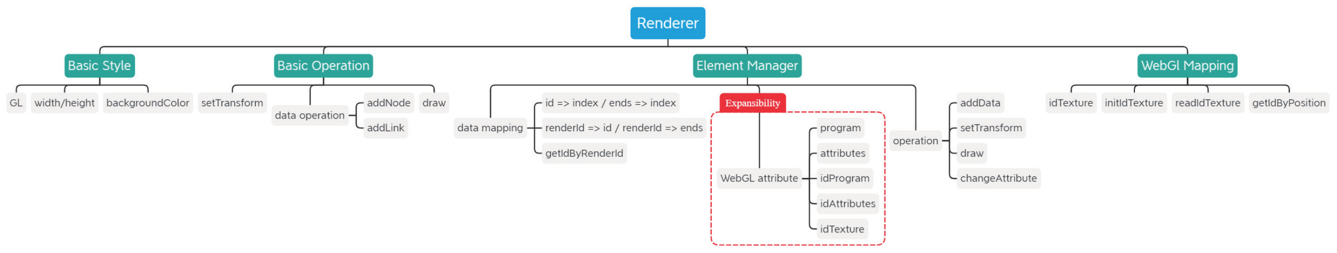


Figure 1: NetV.js designs: NetV.js consists of three parts: core engine, plugins, and library interface.

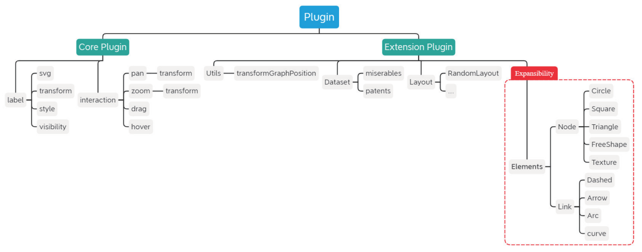


Figure 2: NetV.js designs: NetV.js consists of three parts: core engine, plugins, and library interface.

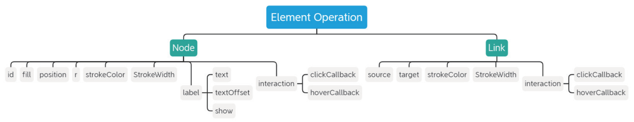


Figure 3: NetV.js designs: NetV.js consists of three parts: core engine, plugins, and library interface.

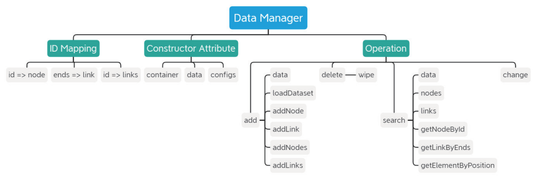


Figure 4: NetV.js designs: NetV.js consists of three parts: core engine, plugins, and library interface.

3.3. 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.

3.4. 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.

3.5. 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.

3.5.1. 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()
```

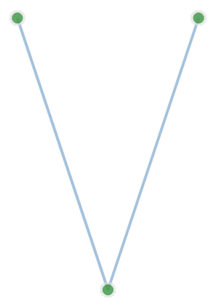


Figure 5: Basic network rendering.

3.5.2. *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()
```

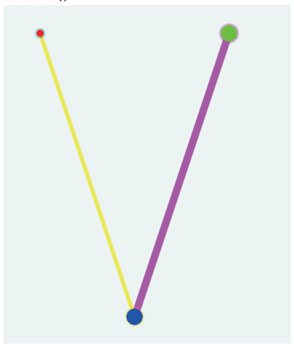


Figure 6: Customized style.

3.5.3. *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()
```

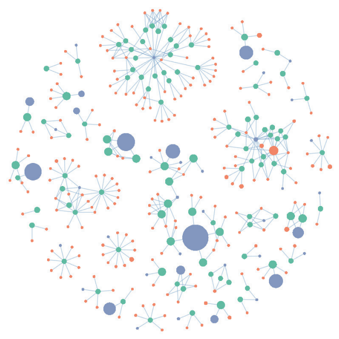


Figure 7: Build-in datasets.

3.5.4. *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)

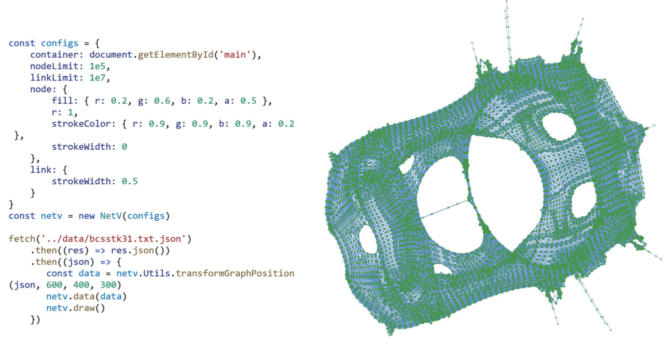
Figure 8: Layout. (a) Combining with D3.js. (b) Using NetV.js plugins.

3.5.5. *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.

3.5.6. *Label*

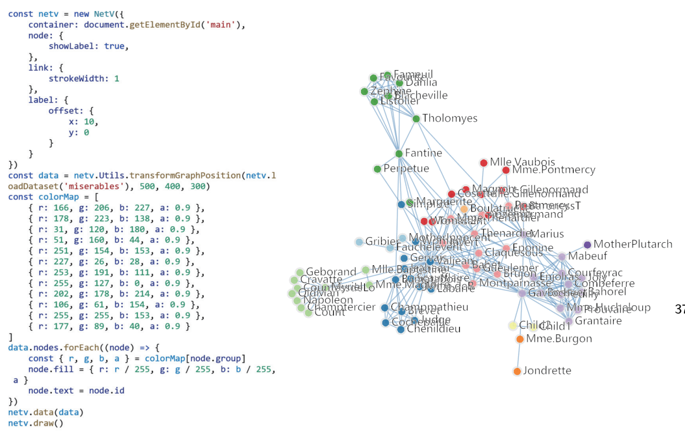
NetV.js supports label rendering with different drawing techniques such as SVG, Canvas, and WebGL. Figure 10 shows the network rendering with labels.



```
const configs = {
    container: document.getElementById('main'),
    nodeLimit: 1e5,
    linkLimit: 1e7,
    node: {
        fill: { r: 0.2, g: 0.6, b: 0.2, a: 0.5 },
        r: 1,
        strokeColor: { r: 0.9, g: 0.9, b: 0.9, a: 0.2
 },
        strokeWidth: 0
    },
    link: {
        strokeWidth: 0.5
    }
}
const netv = new NetV(configs)

fetch('../data/bcsstk31.txt.json')
    .then((res) => res.json())
    .then((json) => {
        const data = netv.Utils.transformGraphPosition
(json, 600, 400, 300)
        netv.data(data)
        netv.draw()
    })
```

Figure 9: Large-scale network.



```
const netv = new NetV({
    container: document.getElementById('main'),
    node: {
        showLabel: true,
    },
    link: {
        strokeWidth: 1
    },
    label: {
        offset: {
            x: 10,
            y: 0
        }
    }
})
const data = netv.Utils.transformGraphPosition(netv.l
oadDataset('miserables'), 500, 400, 300)
const colorMap = [
    { r: 166, g: 206, b: 227, a: 0.9 },
    { r: 178, g: 223, b: 138, a: 0.9 },
    { r: 31, g: 120, b: 180, a: 0.9 },
    { r: 51, g: 160, b: 44, a: 0.9 },
    { r: 251, g: 154, b: 153, a: 0.9 },
    { r: 227, g: 26, b: 28, a: 0.9 },
    { r: 253, g: 191, b: 111, a: 0.9 },
    { r: 255, g: 127, b: 0, a: 0.9 },
    { r: 202, g: 178, b: 214, a: 0.9 },
    { r: 106, g: 61, b: 154, a: 0.9 },
    { r: 255, g: 255, b: 153, a: 0.9 },
    { r: 177, g: 89, b: 40, a: 0.9 }
]
data.nodes.forEach((node) => {
    const { r, g, b, a } = colorMap[node.group]
    node.fill = { r: r / 255, g: g / 255, b: b / 255,
 a }
    node.text = node.id
})
netv.data(data)
netv.draw()
```

Figure 10: Label.

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 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.

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

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

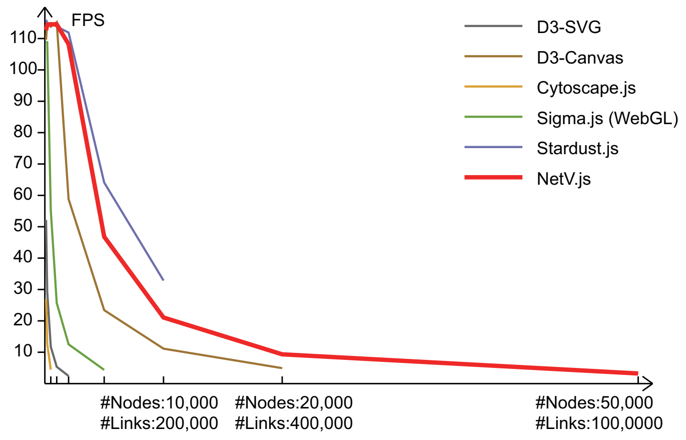


Figure 11: The Frame per Second (FPS) experiment.

components and visual analysis algorithms for analyzing network data.

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] 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] V. Batagelj, A. Mrvar, Pajek, in: Encyclopedia of Social Network Analysis and Mining, 2014, pp. 1245–1256. doi:10.1007/978-1-4614-6170-8_310.
URL https://doi.org/10.1007/978-1-4614-6170-8_310

[16] 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.

[17] 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.