Visual Informatics 2 (2018) 136–146

Contents lists available at ScienceDirect

Visual Informatics

journal homepage: www.elsevier.com/locate/visinf

ECharts: A declarative framework for rapid construction of web-based visualization☆

Deqing Li [a], Honghui Mei [b], Yi Shen [a], Shuang Su [a], Wenli Zhang [a], Junting Wang [a], Ming Zu [a], Wei Chen [b,*]

[a] Baidu Inc., China
[b] State Key Lab of CAD&CG, Zhejiang University, China

ARTICLE INFO

Article history:
Received 6 March 2018
Received in revised form 24 April 2018
Accepted 26 April 2018
Available online 17 May 2018

MSC:
00-01
99-00

Keywords:
Information visualization
Web-based visualization

ABSTRACT

While there have been a dozen of authoring systems and programming toolkits for visual design and development, users who do not have programming skills, such as data analysts or interface designers, still may feel cumbersome to efficiently implement a web-based visualization.

In this paper, we present ECharts, an open-sourced, web-based, cross-platform framework that supports the rapid construction of interactive visualization. The motivation is driven by three goals: easy-to-use, rich built-in interactions, and high performance. The kernel of ECharts is a suite of declarative visual design language that customizes built-in chart types. The underlying streaming architecture, together with a high-performance graphics renderer based on HTML5 canvas, enables the high expandability and performance of ECharts. We report the design, implementation, and applications of ECharts with a diverse variety of examples. We compare the utility and performance of ECharts with C3.js, HighCharts, and Chart.js. Results of the experiments demonstrate the efficiency and scalability of our framework. Since the first release in June 2013, ECharts has iterated 63 versions, and attracted over 22,000 star counts and over 1700 related projects in the GitHub. ECharts is regarded as a leading visualization development tool in the world, and ranks the third in the GitHub visualization tab.

1. Introduction

With the boosting of data, there is a dire demand on presenting and analyzing the data (Wang et al., 2016), eliciting the rapid construction tools of data visualization. While there have been a dozen of authoring systems and programming toolkits for visual design and development (Mei et al., 2018), it is still cumbersome for users, such as data analysts or interface designers, to rapidly implement a web-based and interactive visualization (Grammel et al., 2010).

A recent trend is to enable visualization construction in graphical user interfaces (GUI), without textual programming (Satyanarayan and Heer, 2014). Typically, these tools lack of expressiveness, especially in specifying interactions. Meanwhile, the design of graph grammar (Ichikawa et al., 2013) is essential for navigating the design space (e.g., Lyra (Satyanarayan and Heer, 2014) is built upon the visualization grammar Vega (Satyanarayan et al., 2016)).

Declarative languages such as D3.js (Bostock et al., 2011) and Vega (Satyanarayan et al., 2016) are popular tools for building visualizations. With the encapsulation of underlying data transformations and control flow exposed to users, these declarative languages allow users to focus on the visual design. However, users have to be very skilled at web development. For example, D3.js requires users to be familiar with HTML, CSS, SVG and DOM. Similarly, Vega requires users to master a new set of graphics syntaxes. These requirements make the development non-trivial.

We argue that the flexibility and complexity of visual design should not be limited by the requirement on programming skills (Heer et al., 2008). The essential motivation of this work is to fill this gap through a declarative object **option** and composable visualization components, which are modeled with the user-configurable declarative object **option**. When users create

☆ Wei Chen is supported by National 973 Program of China (2015CB352503), National Natural Science Foundation of China (61772456, 61761136020).

* Corresponding author.

E-mail addresses: lideqing@baidu.com (D. Li), meihonghui@zju.edu.cn (H. Mei), shenyi01@baidu.com (Y. Shen), sushuang@baidu.com (S. Su), zhangwenli01@baidu.com (W. Zhang), wangjunting@baidu.com (J. Wang), zuming@baidu.com (M. Zu), chenwei@cad.zju.edu.cn (W. Chen).

Peer review under responsibility of Zhejiang University and Zhejiang University Press.

https://doi.org/10.1016/j.visinf.2018.04.011

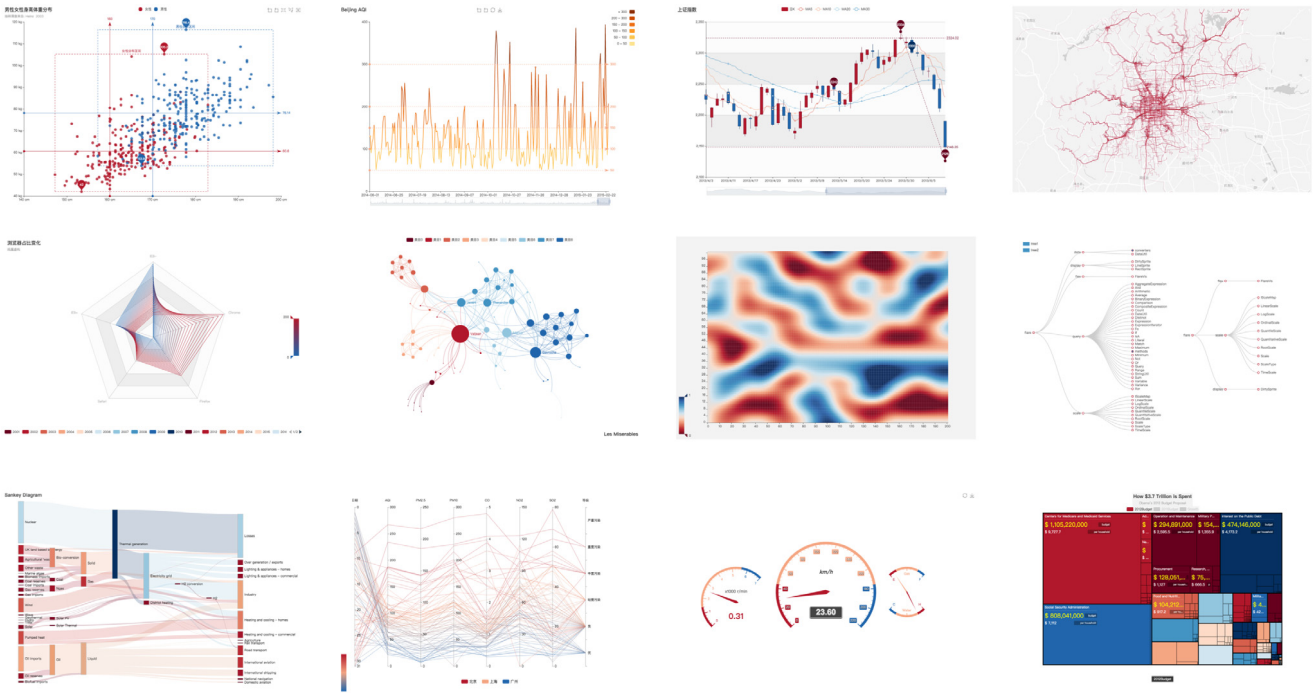

Fig. 1. Examples of ECharts chart types. From top to down, left to right: scatterplot, line chart, candle-stick charts, geomap, radar chart, node-link graph, heatmap, tree diagram, sankey diagram, parallel coordinates, gauge chart, treemap.

visualizations, they do not have to be proficient in web-relevant programming. Instead, they only need to take a few minutes to be familiar with the provided visual components, and configure the components by specifying data, visual encoding, annotation and visual style.

Here we contribute ECharts, an easy-to-use framework to construct interactive visualization. The main contribution confirm to three goals.

Easy-to-use. There are some difficulties for users to learn the visual representations if a declarative language is employed. It is desirable to allow users to focus on the design of the visualization rather than on the use of some tools.

Rich built-in interactions. Efficient data exploration and analysis demand a wealth of configurable interactions. ECharts designs and implements rich built-in interactions that are attached to each chart type, minimizing the requirement of customization of user.

High performance. By introducing a streaming system architecture and incremental rendering mode, high performance is achieved with ECharts, even when handling millions of data points.

2. Related work

2.1. Grammars of visualization

Charting tools such as Excel (Microsoft Excel, 2017) and ManyEyes (Viegas et al., 2007) support rapid generation of charts by selecting appropriate forms from predefined visual templates. One main drawback of this scheme is that the expressiveness of visualization is fully bound by the provided templates.

Wilkinson (Wilkinson, 2005) introduces *The Grammar of Graphics* for a more wide range of graphical specification and *"shun chart typology"*. It has far-reaching influence that formal languages are designed to describe the rules of generating graphics. Following this idea, many softwares and frameworks are implemented to bring more customization for users, such as Tableau (Tableau software, 2017) (formerly Polaris (Stolte et al., 2002)), ggplot2 (Wickham, 2009, 2010), and ggivs (ggvis 0.4 overview, 2017). Some visualization frameworks abstract low-level graphics drawing to achieve more concise specification, such as InfoVis Toolkit (Fekete, 2004), Improvise (Weaver, 2004), prefuse (Heer et al., 2005), and Flare (Flare, 2017). In these frameworks, inheritable visualization widgets or composable operators are introduced to ease the users' burden. On the other side, those *declarative, domain specific languages* (DSL) for information visualization, including ProtoVis (Bostock and Heer, 2009; Heer and Bostock, 2010), D3.js (Bostock et al., 2011), and Vega (Satyanarayan et al., 2016), allow users to specify visualizations by directly mapping data to visual elements without computational details. Based on this, Vega-lite (Satyanarayan et al., 2017) abstracts more over data models, graphical marks, visual encodings and other detailed specifications. Such a high-level abstraction enables rapid construction of visual forms by leveraging partial specification that omits low-level details and resolves ambiguities with default values.

However, complex visual design always stands opposite to simplicity and efficiency. The design of ECharts seeks to achieve a good balance between rapid construction and expressive visual design.

2.2. Graphics libraries

With the drastic development of web technology, such as Cascading Style Sheets (CSS), Java applets, JavaScripts, and AJAX, an increasing number of applications are deployed through web sites. For visualization tools, web-based approaches provide a simple solution to support cross-platform deployment and enable easy sharing and communication between collaborators and audiences. These web tools make users focus on core issues (Mwalongo et al., 2016) without maintenance.

Early construction tools for web-based visualization utilize Java and Flash. Processing (Processing, 2017; Reas and Fry, 2003, 2005) provides support of Java applet in its first edition. Flare (Flare, 2017) is an ActionScript library for creating visualizations that runs in the Adobe Flash Player. Later, visualization construction tools turn to use JavaScript and Scalable Vector Graphics (SVG), including Raphaël (Raphaël—JavaScript Library, 2017), JavaScript

InfoVis Toolkit (JIT) (JavaScript InfoVis Toolkit, 2017), ProtoVis and D3.js. Essentially, SVG can produce the best quality of 2D drawings, and JavaScript provides flexible and easy-to-use callback of user input events received by DOM elements when specifying user interactions.

Despite its convenience, using SVG has a low performance because the DOM structures of all SVG elements and corresponding styles and events need to be maintained in the web browser. An alternative scheme is to utilize HTML5 canvas for 2D drawing. For instance, using canvas in Vega (Satyanarayan et al., 2016) achieves a 2× to 4× performance speedup over SVG. Other tools such as iVisDesigner (Ren et al., 2014), iVolVER (Méndez et al., 2016) and Data-Driven Guides (Kim et al., 2017) employ canvas for high performance. However, as interaction specification for individual graphical objects is not natively supported by HTML5 canvas, tools using canvas must design an own event handling mechanism.

ECharts implements a 2D vector drawing library named ZRender, which supports display of visual forms in HTML5 canvas and manages graphic elements, rendering as well as events. Details of ZRender can be found in Section 5.2.

2.3. Customization of visual design

The most popular web-based visualization design engine, D3.js, fully exploits the capability of web presentation by directing link data to native web presentation, i.e., SVG. This leads to a high expressiveness. However, users may still feel cumbersome when a visual design contains complex visual mappings to the "d" attribute of SVG path elements. For that, D3.js provides modules to encapsulate common visual tasks including shapes, scales, layouts, and interactions. This provides a flexible choice subject to users' expertise, but still causes confusion when high-level and low-level declarations are mixed. Vega overcomes this problem by modeling complex forms as the combination of basic shapes that are arranged by predefined layouts.

Recently proposed Data-Driven Guides (DDG) (Kim et al., 2017) supports convenient creation of charts and flexible visual design by means of a *data guide* mechanism. The guides are simple shapes with data mapped to one of their attributes, for example, a series of parallel lines whose length is bound to one of the data attributes. Based on such guides, designers can then draw SVG shapes on the top of data guides.

To provide flexibility while preserving simplicity, ECharts provides a special component **series** which allows users to modify a predefined chart by changing its rendering process.

One of the most complex tasks in creating visualization is to specify user interactions. One common solution is to attach event listeners. Event listeners are callback functions that are invoked when the specified events occur and react to user inputs. Based on this, D3.js introduces event handlers called *behaviors* for the reuse of interaction techniques. In Vega, events are extracted from the input stream as *signals* and trigger *predicates* which can affect visual mappings. On the other hand, a reactive programming method (Satyanarayan et al., 2014) avoids the dreaded "callback hell" but is hard for novices to design. Likewise, Vega-Lite allows users to simply define a new interaction by *transforming* predefined ones. However, only the selection operation is supported by Vega-Lite.

In ECharts, rich interactions are automatically attached to generated charts, and new interactions can still be specified by means of an easy-to-use event system.

3. Declarative visualization design

ECharts employs an all-in-one JSON format **option** to declare the components, styles, data and interactions, resulting in a logic-less and stateless mode. The main advantage of JSON format lies in that it is safe to store, transmit and execute, and is easy to do further validation.

Following the conventions of well-known tools like Microsoft Excel, ECharts uses **series** to abstract a group of graphic elements that are mapped and encoded from data. For example, all lines or bars in a cartesian coordinate system form a series. Similarly, pie chart, treemap, graph or other chart types can be abstracted as a series. In other words, a series is an instance of a type of chart. Besides, ECharts uses **component** to name the functional unit like data zooming, visual encoding and toolbox.

ECharts provides a method **setOption** to create or update the components and series from the accepted option. If the setOption is to update the data, ECharts employs a key-based *diff* algorithm to find the difference of data and use proper transitions to display it.

3.1. Declarative options

The declarative option is a hierarchical JSON object. On the top level, all components, series and global **settings** are declared. In particular, global settings, such as color palette, font and animation, are universal settings that are posed on different series and components. ECharts checks whether the key in the top level is registered as a component, then creates all components. Otherwise the values are set as global settings.

Components like **legend**, **tooltip**, **brushing**, **visualMap** etc., are all optional. Different components can be composed for different visualization purposes.

On the component level, properties are configured subject to associated components, including layout, styles and states. Most states in ECharts components are stored in the global option and change synchronously when a user interaction happens. After getting the current global option including components states, the view can be easily repeated in another environment. This is very useful for debugging, replay and automated testing.

There is also a **series** field that contains one or more series of datasets and their chart types. Most of the chart types have a coordinate system, like cartesian, polar and geographic. A coordinate system is also defined as a component on the top level. For a series, there are three ways to index the corresponding coordinate system: index, ID and name, which are common for indexing any other component.

Fig. 2 presents an outline of a basic option.

3.2. Chart types

To make it amenable for various scenarios, ECharts provides a variety of chart types, which may be divided into three categories, including *built-in*, *customized*, and *extended*.

Specifically, built-in chart types provide a convenient way to create charts that are most commonly used; if a specific layout or chart type is required, which is not provided with built-in chart types, users can use a customized series to specify the layout; the extended mode serves as an alternative to make customized chart types, and has more control over the rendering process and interactions.

Generally speaking, the built-in mode is the easiest one. Otherwise, if a chart type can be reused for other cases, it is recommended to be formulated as extension. Nevertheless, the customization mode has the maximum flexibility.

```
{
    // Define xAxis and yAxis of cartesian
    // coordinate system.
    xAxis: {},
    yAxis: {},
    // Define interactive components
    legend: {
        data: [...]
    },
    // Define series
    series: [{
        type: 'line',
        // Indexing coordinate system
        xAxisIndex: 0,
        yAxisIndex: 0,

        data: [...]
    }]
}
```

Fig. 2. The structure of a basic option.

3.2.1. Built-in chart types

Built-in chart types include general-purposed chart types like scatterplots, line charts, bar charts, pie charts, geomap, and candlestick charts.

ECharts supports 19 built-in chart types. Some of them are shown in Fig. 1.

3.2.2. Customized chart types

With the customized chart types, users only need to concentrate on the rendering logic, without implementing details like creating or releasing graphic elements, or transition animation, or visual map. Section 4.2 presents more details on the customization process.

Fig. 3 shows examples of customized chart types.

3.2.3. Extended chart types

ECharts provides an extension mechanism to support adding new features (Section 4.3). Fig. 3(e) shows an example of tag cloud extension.

3.3. Coordinates systems

In ECharts, the coordinate systems can be configured within the declarative object model. The basic coordinate systems are **Cartesian coordinate system** and **Polar coordinate system**. The former can be used through the configuration **xAxis** and **yAxis**; while the latter is specified with: **polar, radiusAxis** and **angleAxis**; Both coordinate systems can receive more than two axes, which are identified by the index. ECharts Cartesian coordinate system supports three kinds of axes: value, category and time.

ECharts provides two special coordinate systems: calendar and geomap. In addition, ECharts automatically maps temporal or spatial information in the data to corresponding coordinate systems. These systems can receive arbitrary graphic elements targeted at specific positions. That means, other charts, such as pie chart, can be drawn within the coordinate system.

3.4. Components

Each component in ECharts contains three types of functions: **visual encoding**, **guide**, and **interaction**. Corresponding modules are integrated into different stages in the pipeline. Some components like tooltip and markPoint focus on the guide. Some components like **dataZoom** focus on the interaction. Most components are responsible for more than two functions.

3.4.1. Visual encoding

Each chart is a combination of data-driven visual encodings, like height in bar, position in scatterplots, area in pie, etc. Visual encoding components are used for this purpose. Visual channels like color, color lightness, size, glyph are composable and can be encoded in these components from different levels of data. For example, a **legend** component encodes color for different series. Based on this, a **visualMap** component can modify the color lightness to indicate the value comparison in each series. In particular, the alpha channel of color is used to encode the data selection status if **brushing** component is included.

Most visual encoding components are interactive, like selecting the series with **legend** components, filtering data in range with **visualMap** components.

3.4.2. Guide

The Guide function provides descriptive information over the visualization. Components can use labels, guide lines, glyphs to label out the particular items or show additional information.

Most components in ECharts play the role of guide **components**. Coordinate system uses labels and ticks to show the range of data and helps reading the value. **Legend** uses labels with encoded color to show what the series is of each element in the view. **VisualMap** can be used as a **color bar**.

3.4.3. Interactions

ECharts provides a set of interactive components to support interactions operations like: panning, zooming, selecting. In particular, **legend** components can unselect unnecessary series. **dataZoom** components can filter data along a specified dimension and **brushing** components can highlight selected data within the specified area. Each **interactions** component triggers an event; These events are necessary in linking multiple charts instances. The properties in the events like selected data indices are useful when performing statistics on data or showing detailed information are needed.

4. Customization of visualization design

4.1. Interaction specification

The declarative object model supports event listeners: callback functions that receive input events targeted at specific graphic element. ECharts provides this feature for the entire mouse event types and event types built by ECharts. To keep the consistency with other callback functions in ECharts, the function receives an object `params` that includes the information of the targeted graphic element, to support data-driven interaction. The data information includes `componentType`, `seriesType`, `data` and `dataIndex`.

Both internal and external procedures control the interactive behavior of the visualization view. This ensures that there is only one mechanism that handles interactions, rather than one for each. This mechanism works by dispatching ECharts' built-in events to registered event listeners when some elements are triggered, or manipulating related data and updating visualization.

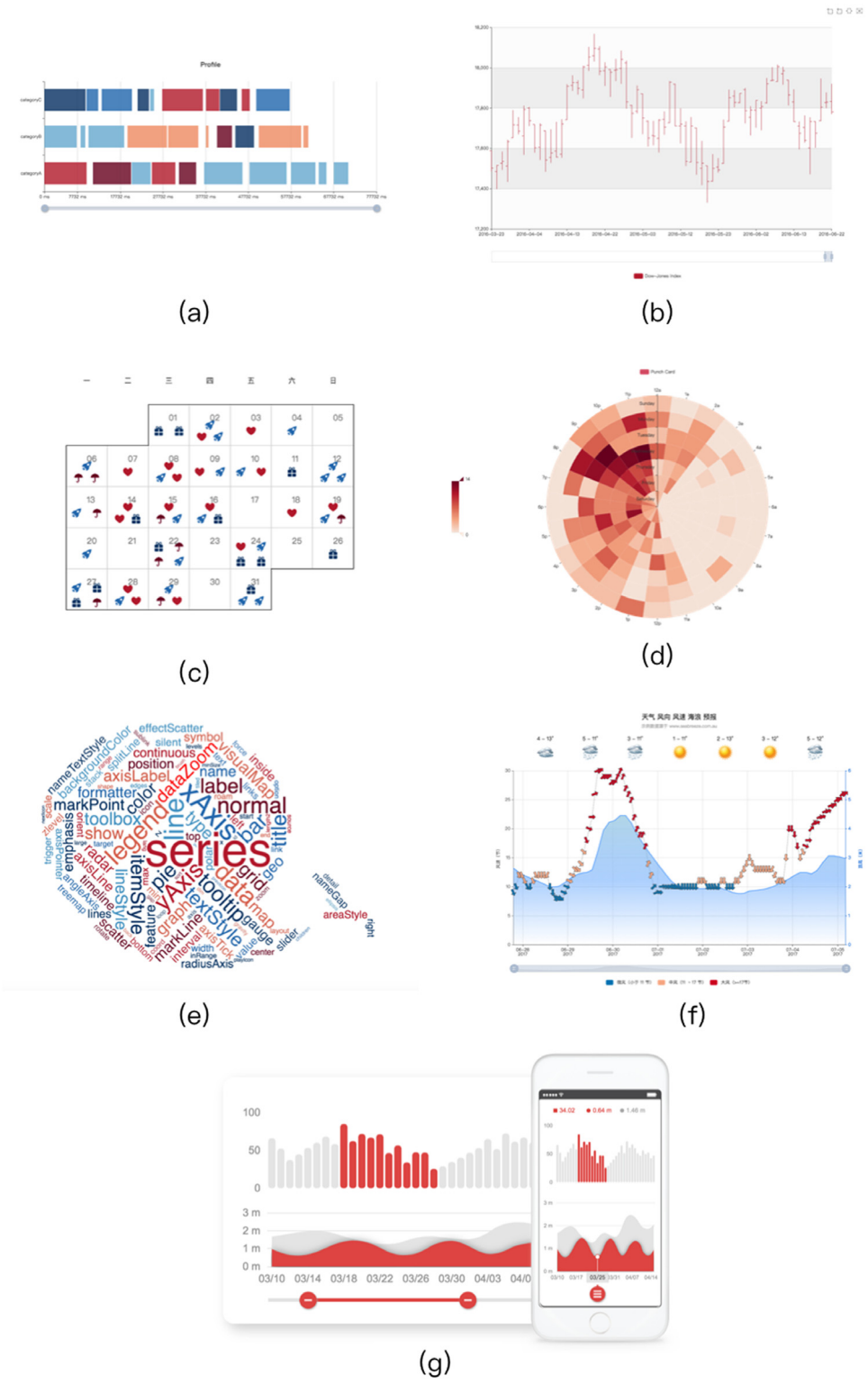

Fig. 3. Example of custom chart types: (a) *x*-range plot, (b) OHLC plot, (c) calendar, (d) polar heatmap, (e) tag cloud, (f) map drawn with ArcGIS API. (g) shows the responsive design of charts on PC (left) and mobile (right).

We propose an extensional architecture that focuses on data streaming to enable direct reuse of defined interactions of existing components (e.g. the hover event on a legend). This can reduce the workload when developing a new chart type for EChart. This is feasible because the handled data can be pulled into a separate module and each component has an individual processor, making the interactions on charts and components separate.

4.2. Customization series

In the scope of data visualization, chart types are not enumerable. It is always expected to be able to implement new visual forms with the help of low-level libraries like D3.js. When creating a new type of chart, the features provided by the existing charts and components (e.g., zooming, tooltip, layout, visual encoding, user

configuration, platform compatibility) should be adopted almost transparently. Consequently, the developers need to be proficient in underlying libraries and take much workload to implement and debug the logic, which is unnecessary in most cases.

We take OHLC[1] chart as an example, which is a type of stock chart used in US as shown in Fig. 3(b). To create a new chart type of OHLC together with the feature of "zoom" and "tooltip", developers have to consider the creation of graphic elements and the layout in a coordinate system. Moreover, a set of interactions need to be specified: the hover interaction to show and hide "tooltip", the "zoom" interaction of data or graphic elements correspondingly, and the animation of triggered visual elements. Moreover, new chart types and existing chart types are usually needed to be composed in the same coordinates.

To resolve the aforementioned issues, ECharts introduces "Customization Series", with the rationale of decomposing the correlated complicated features and orchestrating them in framework level for reuse.

New chart types like OHLC chart and its variants are not provided as built-in chart types, and the graphic elements are different from existing chart types, which inevitably cause new implementation work. But the logic of layout in coordinates, tooltip, animation and zooming, can be reused transparently or with few configurations.

Benefited from the data-driven stream architecture (Fig. 5), features like "zoom", "visual encoding" and "tooltip" can be decoupled in different stages of the pipeline and be reused transparently. Only the rendering stage that is to be exposed as an extension point, called `renderItem`, and is provided by developers, is responsible for transforming data to the definitions of graphic elements.

Some layout utilities are provided `renderItem`, such as `api.coord(datum)`, which can map data points into the declared coordinates. Some visual encoding utilities are also provided, such as `api.style()` that retrieves the current visual mapping result, where the mapping is performed in the previous stage. The "zoom" works in a similar fashion. The transform animation of graphic elements is adapted implicitly with the same mechanism despite the difference of chart types.

In this way, only few lines of code are needed to create a new chart type. An example in Section 6.1 illustrates more details.

4.3. ECharts extension

Based on the design of declarative options and extensional architecture, ECharts allows users to import plugins written by other users. This facilitates to create relatively complex charts or combine ECharts with other libraries. For instance, ECharts can work together with an online map library (e.g. ArcGIS used in Fig. 3(g)) to create elements drawn on a map. The plugins not only work with coordinates, but also other components such as visual encoding (e.g. Fig. 3(f) shows an example of node-link graph which colors of nodes are assigned by the plugin corresponding to the community detection result of the graph).

Moreover, ECharts extensions also provide support for complex chart types (Section 3.2.3), statistical computing, web framework combination (e.g. with AngularJs or Vue), and generating ECharts options with other programming languages (e.g. Python or R).

4.4. Cross-platform presentation

The cross-platform compatibility of ECharts ensures that charts behave similarly on various platforms and support adaptations on certain platforms when necessary.

This means that charts should have the same presentation and interaction on different platforms, and sometimes a few adaptations (like layout changes) may be required. This is so-called *responsive design*, which is described in Section 4.4.2.

4.4.1. Universal appearance and behavior

Charts created with ECharts are typically embedded in Web pages, and may be presented in different Web browsers, Operating Systems, and devices. ECharts provides rendering charts with Canvas, SVG, and VML, which have advantages on different platforms.

It is essential for ECharts to provide a universal appearance of the charts, and means of interactions. Besides, this job should be done implicitly, which means that users do not need to do extra work to ensure that. For example, it is expected that a chart looks the same in a Google Chrome Web browser of Macintosh Operating System and in a Internet Explorer Web browser of Windows Operating System.

For that, ECharts uses a rendering engine called ZRender to manage rendering elements and render to different platforms in a universal way. Please see Section 5.2 for more information.

4.4.2. Responsive design

Visualizations on mobile devices have a different design from that on PC, as shown in Fig. 3(g). This is because mobile devices have a small screen size.

In this case, ECharts uses a policy similar to CSS Media Queries,[2] with which users need to set rules (**option**) for each device requirement (**query**). Fig. 4 shows an example.

The **baseOption** sets the overall options, so that most options do not need to be repeated in **media**. And users can set the rule according to device width, height, or aspect ratio, and set special options for each rule respectively.

In this way, it is ensured that the least code is required for the common parts, and special rules can be applied for certain device sizes.

5. Architecture

5.1. Streaming architecture

A modern universal charting library is required to be componentized, extensible and interactive. To achieve this goal, ECharts introduces a streaming architecture (Fig. 5). In a complicated visualization instance there are usually multiple visual components cooperating with each others, responsible for performing different types of layout, visual encoding, user interaction and rendering. Some of those jobs are dependent on other jobs, and some may be conflicted with others. The simplest example is that both "legend" components and some components dedicated in visual mapping are able to control the appearance of chart elements dynamically.

Here, at least two issues should be considered. Firstly, for extendibility consideration, components that have dependency relationship should not know each other, neither the existence of instance nor the component type. ECharts uses a universal abstraction of data as the source and target of each process to build the relationship of components but keep them independent. Secondly, the priority of processes is needed to be ruled, where

[1] OHLC chart, https://en.wikipedia.org/wiki/Open-high-low-close_chart.

[2] CSS Media Queries, www.w3.org/TR/css3-mediaqueries.

```
option = {
    baseOption: { // Overall option
        title: {...},
        legend: {...},
        series: [{...}, {...}, ...],
        ...
    },
    media: [ // Rules for different screen sizes
        {
            query: { // Rule 1
                minWidth: 200,
                maxWidth: 800,
                maxHeight: 300,
                minAspectRatio: 1.3
            },
            option: { // Works if meeting with Rule 1
                legend: {...},
                ...
            }
        },
        {
            query: {...},   // Rule 2
            option: { // Works if meeting with Rule 2
                legend: {...},
                ...
            }
        },
        ...
    ]
};
```

Fig. 4. Media Query for Responsive Design.

stage mechanism is introduced. Thus ECharts is designed as a data-driven streaming pipeline with stages of data processing, visual encoding and rendering, which produces graphic elements finally. The flow is unidirectional, that is, any user interactions can only modify the raw option or data, and run the pipeline from the beginning. Moreover, each stage can be exposed to developers as an extension point.

The main advantage of the designed architecture, is that both human interaction and the interaction from program are implemented in the same way. This enables programming interface to take control of generating charts and allows users to create custom components or extensions.

5.1.1. Progressive visualization

Visualizing millions of data points usually takes several seconds to transmit data from server to browser. Users always need to wait a long time for data processing and rendering before see and interact with the visualizations. Additionally, when performing updates caused by user interactions, the main UI thread is blocked and cannot react to concurrent animations and interactions. To address this problem, we introduce incremental rendering techniques based on our streaming architecture. As shown in Fig. 6, data can be loaded and split into several small chunks. Chunks are pushed into the pipeline one by one, and then be processed and rendered.

When data is loading, or changed causing by user interactions, ECharts will create tasks for operating the split data chunks. These tasks include data filtering, visual encoding, graphic elements creation, etc. Created tasks are sorted by their priorities and the indexes of corresponding data chunks before being executed. Only a limited number of tasks can be executed in each frame to ensure that the execution time is less than 16 ms. Then the `requestAnimationFrame` is called and remaining tasks are paused and wait to be executed until next frame. If a new interaction happens during the process, all running tasks are discarded and new tasks are created.

In this way, the main UI thread of ECharts is never blocked. Users can always interact with the visualization smoothly with immediate feedback.

5.1.2. Multi-Thread rendering

When running all tasks in the main UI thread, Canvas drawing must wait until the data processing and vice versa (Fig. 7). Even after optimization, drawing on canvas still costs much time, especially when drawing complex shapes like a circle. The next task can only start after the latest task is finished and waste much time on I/O blocks. To further improve the performance of ECharts, we implement a multi-thread mode that separate data processing and canvas drawing in different threads (Fig. 8).

The web worker enables scripts to run in a background thread. It provides a possibility of multi-thread rendering. To make ECharts run in the worker, a mock canvas is created and used in the ECharts instance. This mock canvas records all operations, like changing fillStyle, filling or stroking a path, drawing a text, during the rendering process. These operations and corresponding parameters are stored in a Float32Array commands list. After finishing one task, the command list is transmitted to the main thread with

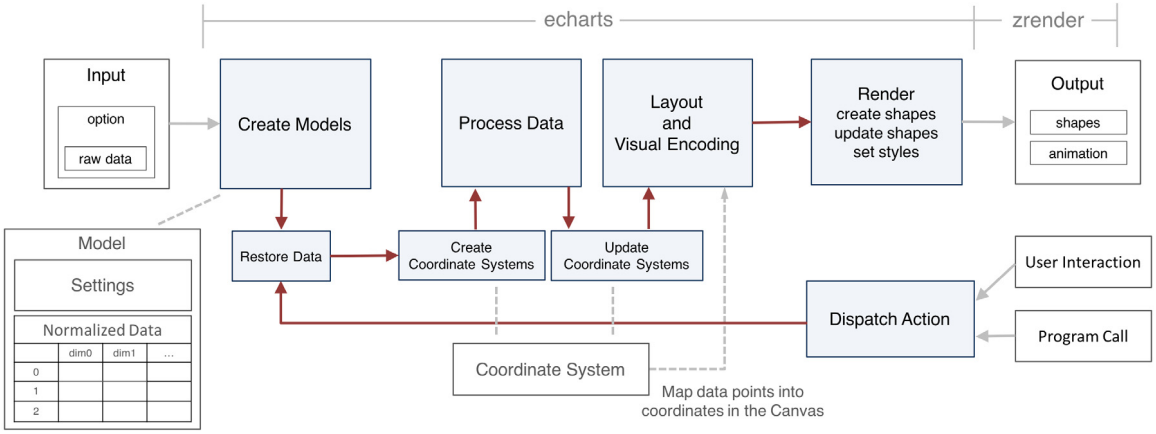

Fig. 5. The design of the data-driven architecture, where the raw data and the settings are modeled, and go through the stages of processing, layout, visual encoding, and are rendered as graphic elements finally. User interactions or programming call can trigger the pipeline from the beginning.

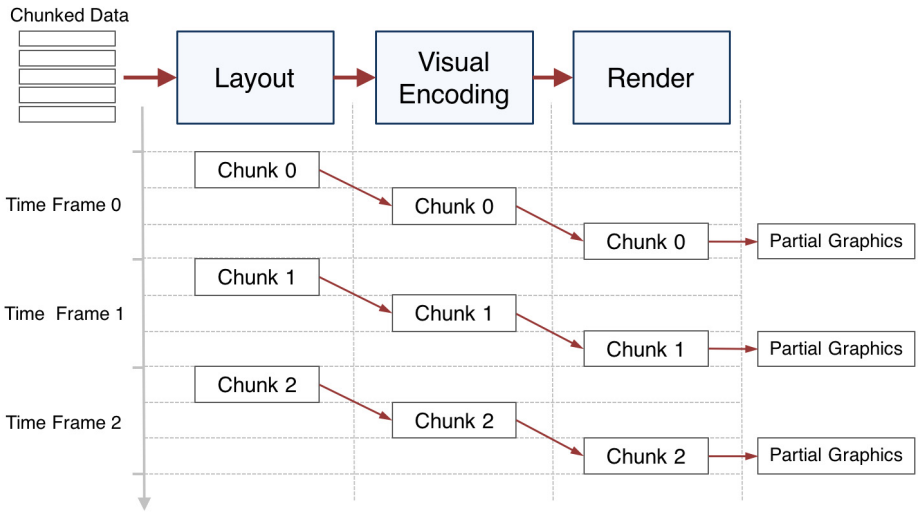

Fig. 6. The flowchart of progressive visualization.

Fig. 7. The flowchart of the single-thread mode in ECharts.

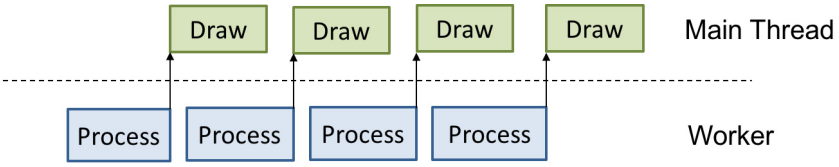

Fig. 8. The flowchart of the multi-thread mode in ECharts.

`postMessage` function. The real Canvas created in the main thread repeats the received commands and draw the graph on the screen. Meanwhile, in the worker thread, ECharts does not have to wait for the drawing to be finished and moves to the next task immediately.

5.2. ZRender

We additionally design and implement a 2D vector library named ZRender for graphic elements management, renderer management and event system. It supports multiple rendering backends, including Canvas, SVG and VML.

5.2.1. Graphic elements management

In ZRender, graphic elements like rectangle, circle, text and image are stored in a tree structure, which is similar to a DOM tree. In the tree, leaf nodes are called **displayable**, which can be texts, images, paths and drawn on the canvas. Internal nodes are called **group**. **Group** stores other groups or **displayables** as children. Each node has scale, rotation, position properties. The transformations of a group node are applied on its children nodes and accumulated in a top-down manner before visualization.

There are three types of **displayable** in ZRender, namely, **text**, **image** and **path**. **Text** and **image** basically wrap the interface of canvas. For **path**, a proxy is generated to store the path commands data in a Float32Array. This data will be used in the hit test of the event system. It is also useful for rebuilding the path data during the rendering process.

5.2.2. Renderer management

An operation on the elements will trigger ZRender to refresh updates in the next frame. For each render, ZRender traverses the tree, updates the transformations of all nodes, identifies all **displayables** needed to be drawn and sends them into a render queue. The object-wise culling is performed by using a rectangle-based bounding box for each shown **displayable**.

Thereafter, the queues are sorted along **z** and **zlevel**. **Displayables** are drawn sequentially along the order. If Canvas is employed, **zlevel** determines which canvas is to be drawn. The **displayables** on the same **zlevel** are not redrawn if they are not changed.

5.2.3. Event system

Unlike DOM, **displayables** drawn in Canvas do not trigger any mouse events. To solve this problem, we implemented an event system which supports mouse event detection and event bubbling. For mobile devices, we also simulate pinch events so that the view can zoomed through two finger pinch operations.

When a mouse moving event happens, ZRender traverses all the **displayables** in the renderer queue, which is updated in the last renderer, in a reversed order. For each **displayable**, we first do a fast check if the mouse position (x, y) is inside its bounding box to accelerate the hit test. For text and image **displayables**, the bounding box test will be adequate to return the found element directly. For path, it needs to further hit test to determine if the point is in the actual drawn area. After finding the hit **displayable**, it will throw a proper event, which can be *click*, *mousemove*, *mouseover*, *mouseout*, and bubbles to the root node.

```
function renderItem(params, api) {
    var categoryIndex = api.value(0);
    var start = api.coord([api.value(1), categoryIndex]);
    var end = api.coord([api.value(2), categoryIndex]);
    var height = api.size([0, 1])[1] * 0.6;
    return {
        type: 'rect',
        shape: echarts.graphic.clipRectByRect({
            x: start[0],
            y: start[1] - height / 2,
            width: end[0] - start[0],
            height: height
        }, {
            x: params.coordSys.x,
            y: params.coordSys.y,
            width: params.coordSys.width,
            height: params.coordSys.height
        }),
        style: api.style()
    };
}
```

Fig. 9. The implementation of the "renderItem".

The entire event system including the hit test is implemented in pure JavaScript and only depends on the **displayable** data. Thus it works well on Canvas, SVG, VML or any other graphic interfaces.

6. Examples

Since the first release of ECharts 1.0 in June 2013, ECharts has been updated to 3.0 in Dec. 2017. A vast number of examples can be found in our official website (http://echarts.baidu.com) and gallery (http://gallery.echartsjs.com), together with basic applications in the tutorial for users. Here we describe one kind of examples to illustrate the usage of customization series and the extension capabilities.

The source code of ECharts is available on GitHub: https://github.com/ecomfe/echarts.

6.1. Customization series

As mentioned above, customization series brings the capabilities of creating new chart types with few codes but still powered by common features such as data zooming, visual encoding, tooltip. Here we take the first customized chart in Fig. 3 as an example.

This kind of chart is usually used to present performance profile, but is not implemented as a built-in chart type of ECharts. It can be placed in cartesian coordinates, with the X axis representing timeline and the Y axis listing categories. To implement this chart type, the only work is to provide a simple function `renderItem` (Fig. 9), in which the value of each datum is retrieved and converted to positions in canvas based on cartesian coordinates and then graphic elements are declared.

To fulfill the implementation, more sophisticated features are needed. It should be noted that, zooming is required to check the detailed information in a tiny duration. Meanwhile, animation is needed for the smooth transitions of graphic elements when zooming or sliding the data window. Other features like visual encoding and tooltip are also needed. As shown in Fig. 9, all these tasks can be easily accomplished with few lines of codes empowered with ECharts.

7. Discussions and comparisons

The declarative options allow users to create visual charts in a simple way and focus on the design of the visualization. In addition, rich built-in components with configurable options provide the flexible specification of interactions. The extensional architecture enables customization for advanced users.

7.1. Discussions

To achieve the optimum performance, ECharts leverages Canvas which does not have the time-consuming DOM manipulations. For that, implementing a lightweight DOM-like system is needed. Benefited from the JIT in modern browsers, the cost of manipulating the attributes on element graphic is quite small, and can be ignored. By using a dirty flag, we can batch all updates and redraw only once. The state change of canvas is costly because of value parsing and validation. Accordingly, for each redraw, we compare two adjacent displayables and only update the different style and transform. In this way, we can prune many unnecessary states changes in Canvas. As a result, animating 7000 rectangles in different colors can be done in 20 ms on chrome 62, with full mouse interactions like dragging and clicking.

There are some disadvantages to use Canvas. One is the memory cost. Memory usually matters in platforms like mobile devices. Too many canvas instances in one page will cause the browser out of memory and crash. Another disadvantage is that a large-sized canvas costs much time for the browser to do compositing with backgrounds, leading to an unsatisfying experience when user scrolling the page with many charts, especially on the mobile devices. To address this bottleneck, ECharts implements an SVG backend for the scenarios of drawing many charts on mobile devices.

7.2. Comparisons

In this section, we compare ECharts with other existing chart libraries, including HighCharts, Chart.js, and C3.

Four libraries support common charts, such as line charts, bar charts and pie charts. HighCharts uses declarative *options* and design of data *series*. However, the items of options and structures of the input data used in HighCharts are highly constrained. In contrast, ECharts provides a more flexible way to construct options by assembling components in need and to freely specify data attributes using `encode` keyword. Although HighCharts can also specify event listeners and import plugins, its expandability is limited due to its constrained options. For instance, HighCharts cannot work with online map libraries while ECharts can seamlessly integrate with online map like Baidu map. Chart.js and C3 do not offer the customization of visual design, and thus cannot generate complex visualizations.

It should be noted that, the extensional architecture of ECharts, including components and the further specification of event listeners, provides a flexible way to support customization of chart design and user interaction. These features are not supported by HighCharts, Chart.js and C3.

Benefited by the ZRender engine, ECharts can render charts with HTML5 canvas while remaining DOM-like manipulations. This enables a high performance as a large number of SVG elements on the web page is time- and memory-consuming. Chart.js also supports rendering with canvas, while HighCharts and C3 can only manipulate SVG.

Table 1 summarizes the differences between ECharts and other three libraries.

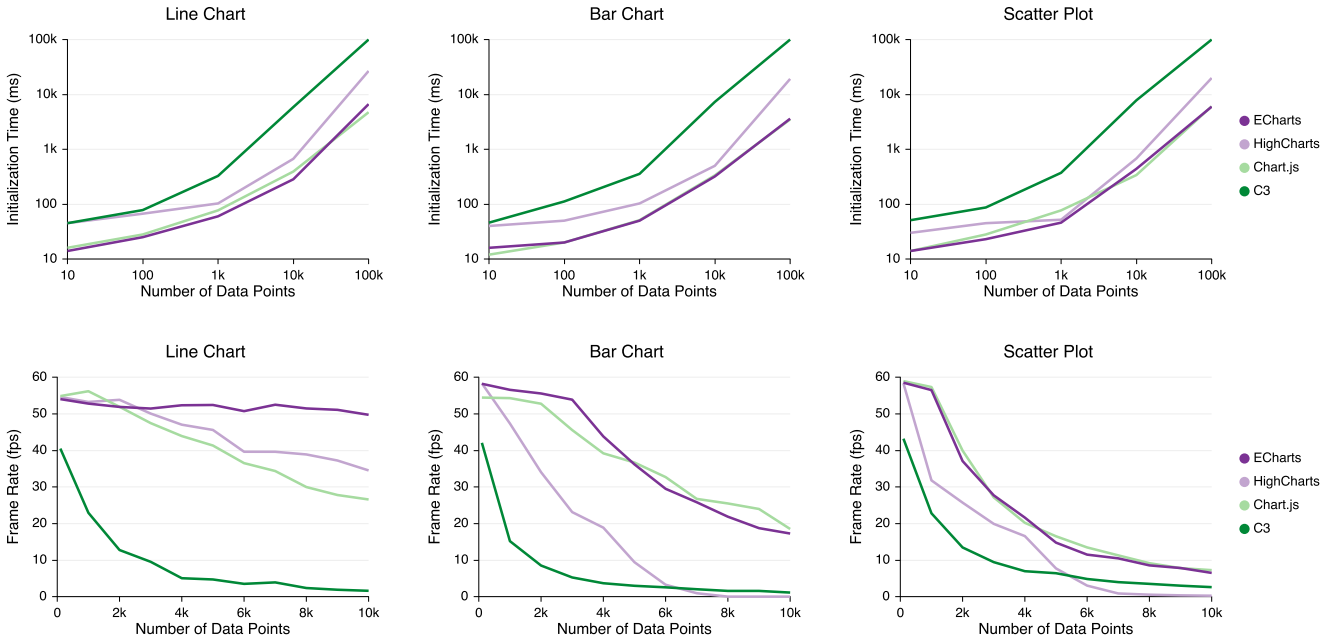

Fig. 10. Comparisons among ECharts, HighCharts, Chart.js, and C3 concerning the initialization time and animation framerate for different chart types.

Table 1
Comparison of ECharts and HightCharts, Chart.js, and C3.

Library	Extended charts	Custom interactions	Canvas
HighCharts	✗	✗	✗
Chart.js	✗	✗	✓
C3	✗	✗	✗
ECharts	✓	✓	✓

7.3. Performance comparison

In this section, we compare the performance of ECharts with C3.js, HighCharts, and Chart.js, which are widely employed charting library. The metrics include chart initialization time and animation framerate.

The initialization time is the duration from the creation to the rendering accomplishment of a chart. Fully rendered means all graphic elements have been drawn on the screen and users can interact with the page. ECharts draws the elements directly after setOption, but Chart.js runs it in the next tick. Thus we measure the ending time in a setTimeout operation and make sure the draw is actually finished. Because SVG in chrome is rendered asynchronously, we cannot get the accurate finish time, and have to run this part in Firefox Quantum.

The animation framerate indicates the performance of redrawing charts. This performance is essential for smooth user experiences during interactions. We enable transition animations and update the data in every 5 s. The transition duration is set to 5 s. Then we record the average FPS with Firefox Quantum development tool.

For each performance metric, we test the performance of four libraries when generating different types of charts, including line chart, scatterplots, and bar charts, at different sizes of datasets.

Fig. 10 reports the performance of four tools. ECharts and Chart.js have shorter initialization time compared to HighCharts and C3. This may caused by the heavy workload of creating the DOM tree of SVG elements. In the framerate tests, ECharts also performs well in all three types of charts. Benefited from using canvas, Chart.js has relatively high performance, while C3 has the lowest frame rates.

The results of performance tests show that ECharts compares favorably with other three tools in terms of performance.

8. Impact

We released the ECharts official version 1.0.0 on June 30, 2013, and has iterated 63 versions up to now. Till Jan. 2018, there are more than 22,000 star counts and over 1700 related projects in the GitHub, making ECharts the third in the GitHub visualization tab. In the meantime, there are nearly 7000 daily Baidu index[3] and 90,000 weekly active developers, and four thousand daily downloads.

To our best knowledge, in the web front-end industry of China, ECharts acquires the recognition rate as high as 90% and the utilization rate as high as 74%. This means that ECharts is the most popular visualization toolkit in China. It has been employed by 90% Baidu's internal software product, and used by external agencies, such as the Chinese Foreign Ministry, China National Bureau of Statistics, China National Patent Offices, Alibaba, Tencent, Huawei and Lenovo.

In addition, universities and research institutions widely leverage ECharts for studying, research and development. We have made a thorough investigation and concluded that almost all universities that have visualization or statistics courses employ ECharts as basic visualization toolkits. Representatives include Zhejiang University, Beijing University, Wuhan University and CAS.

Besides being well known in China, we have numerous foreign users, some of which even contributed code to online ECharts community. Some international users gave us constructive feedback, like:

1. "Went with ECharts 3 at the end, it has almost all the functionality I need"
2. "This library looks amazingly powerful and complete"
3. "It looks amazing and easy to use for charts"

9. Conclusion

We introduce ECharts, an efficient web-based framework for rapid construction of cross-platform visualizations. ECharts is designed to provide easy-to-use visual specification that allows users

[3] Baidu index of ECharts, https://zhishu.baidu.com/?tpl=trend{&}word=ECharts.

who do not have programming skills to construct web-based visualizations. Users are allowed to freely configure components, styles, data and interactions through a declarative option. Such design reduces the workloads to take control of the constructing process and visual structures. Users can further specify visual effects, including novel visual designs and interactions, by utilizing well-designed interfaces. ECharts is built on a high-performance rendering and management system of HTML5 canvas, called ZRender. We present examples to illustrate the possibilities of our design, performance benchmarks of graph drawing, and the usability in real applications.

References

Bostock, M., Heer, J., 2009. Protovis: A graphical toolkit for visualization. IEEE Trans. Vis. Comput. Graphics 15 (6), 1121–1128.

Bostock, M., Ogievetsky, V., Heer, J., 2011. D^3 data-driven documents. IEEE Trans. Vis. Comput. Graphics 17 (12), 2301–2309.

Fekete, J.D., 2004. The infovis toolkit. In: IEEE Symposium on Information Visualization, pp. 167–174.

Flare, http://flare.prefuse.org/. (Last Accessed: Dec. 2017).

ggvis 0.4 overview, https://ggvis.rstudio.com/. (Last Accessed: Dec. 2017).

Grammel, L., Tory, M., Storey, M.-A., 2010. How information visualization novices construct visualizations. IEEE Trans. Vis. Comput. Graphics 16 (6), 943–952.

Heer, J., Bostock, M., 2010. Declarative language design for interactive visualization. IEEE Trans. Vis. Comput. Graphics 16 (6), 1149–1156.

Heer, J., Card, S.K., Landay, J.A., 2005. prefuse: A toolkit for interactive information visualization. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '05. ACM, New York, NY, USA, pp. 421–430.

Heer, J., Van Ham, F., Carpendale, S., Weaver, C., Isenberg, P., 2008. Creation and collaboration: Engaging new audiences for information visualization. In: Information Visualization. Springer, pp. 92–133.

Ichikawa, T., Jungert, E., Korfhage, R.R., 2013. Visual Languages and Applications. Springer Science & Business Media.

JavaScript InfoVis Toolkit, http://philogb.github.io/jit/, last Accessed: Dec. 2017.

Kim, N.W., Schweickart, E., Liu, Z., Dontcheva, M., Li, W., Popovic, J., Pfister, H., 2017. Data-driven guides: Supporting expressive design for information graphics. IEEE Trans. Vis. Comput. Graphics 23 (1), 491–500.

Mei, H., Ma, Y., Wei, Y., Chen, W., 2018. Design space of construction tools for information visualization: A survey. J. Vis. Lang. Comput. 44, 120–132.

Méndez, G.G., Nacenta, M.A., Vandenheste, S., 2016. iVoLVER: Interactive visual language for visualization extraction and reconstruction. In: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, CHI '16. ACM, pp. 4073–4085.

Microsoft Excel, https://products.office.com/excel/. (Last Accessed: Dec. 2017).

Mwalongo, F., Krone, M., Reina, G., Ertl, T., 2016. State-of-the-art report in web-based visualization. Comput. Graph. Forum 35, 553–575.

Processing, http://processing.org. (Last Accessed: Dec. 2017).

Raphaël—JavaScript Library, http://dmitrybaranovskiy.github.io/raphael/. (Last Accessed: Dec. 2017).

Reas, C., Fry, B., 2003. Processing: A learning environment for creating interactive web graphics. In: ACM SIGGRAPH 2003 Web Graphics. ACM, pp. 1–1.

Reas, C., Fry, B., 2005. Processing.org: A networked context for learning computer programming. In: ACM SIGGRAPH 2005 Web Program. ACM, p. 14.

Ren, D., Höllerer, T., Yuan, X., 2014. iVisDesigner: Expressive interactive design of information visualizations. IEEE Trans. Vis. Comput. Graphics 20 (12), 2092–2101.

Satyanarayan, A., Heer, J., 2014. Lyra: An interactive visualization design environment. Comput. Graph. Forum 33, 351–360.

Satyanarayan, A., Moritz, D., Wongsuphasawat, K., Heer, J., 2017. Vega-Lite: A grammar of interactive graphics. IEEE Trans. Vis. Comput. Graphics 23 (1), 341–350.

Satyanarayan, A., Russell, R., Hoffswell, J., Heer, J., 2016. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. IEEE Trans. Vis. Comput. Graphics 22 (1), 659–668.

Satyanarayan, A., Wongsuphasawat, K., Heer, J., 2014. Declarative interaction design for data visualization. In: Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology, UIST '14. ACM, New York, NY, USA, pp. 669–678.

Stolte, C., Tang, D., Hanrahan, P., 2002. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. IEEE Trans. Vis. Comput. Graphics 8 (1), 52–65.

Tableau software, https://www.tableau.com/, (Last Accessed: Dec. 2017).

Viegas, F.B., Wattenberg, M., Van Ham, F., Kriss, J., McKeon, M., 2007. ManyEyes: a site for visualization at Internet scale. IEEE Trans. Vis. Comput. Graphics 13 (6), 1121–1128.

Wang, X.-M., Zhang, T.-Y., Ma, Y.-X., Xia, J., Chen, W., 2016. A survey of visual analytic pipelines. J. Comput. Sci. Tech. 31 (4), 787–804.

Weaver, C., 2004. Building highly-coordinated visualizations in improvise. In: IEEE Symposium on Information Visualization, pp. 159–166.

Wickham, H., 2009. Ggplot2: Elegant Graphics for Data Analysis. Springer.

Wickham, H., 2010. A layered grammar of graphics. J. Comput. Graph. Statist. 19 (1), 3–28.

Wilkinson, L., 2005. The Grammar of Graphics. Springer.