České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Róbert Čepa**

Studijní program: Softwarové technologie a management
Obor: Web a multimedia

Název tématu: **Vizualizace informace na webu**

Pokyny pro vypracování:

Seznamte se s nástroji pro vizualizaci informace na webu (např. D3.js, ManyEyes, Processing.js, Tableau Public). Na základě analýzy porovnejte vybrané nástroje pro vizualizaci informaci na webu z hlediska vizualizace různých typů dat (skalární data, n-rozměrná data, relační data, apod.), možností interakce s daty, možností rozšiřitelnosti a dostupnosti/ceny. Vyberte jeden z nástrojů a implementujte v něm aplikaci pro vizualizaci adresářové struktury z Google drive. Aplikace bude umožňovat přehlednou vizualizaci atributů (velikost, typ, vlastník, apod.) souborů a adresářů. Aplikace bude používat alespoň dva propojené pohledy (linked/connected views) na stejná data, ve kterých bude možné s daty interagovat.

Seznam odborné literatury:

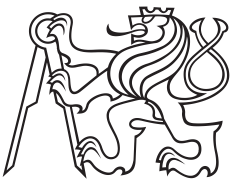
C. Ware. Information visualization: perception for design. Elsevier, 2012.

Vedoucí: Ing. Ladislav Čmolík, Ph.D.

Platnost zadání: do konce zimního semestru 2017/2018

V Praze dne 29. 2. 2016

Bachelor's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Visualizing Information on the Web

Róbert Čepa
Software Engineering and Management - Web and Multimedia

May 2016
Supervisor: Ing. Ladislav Čmolík, Ph.D.

Acknowledgement / Declaration

First and foremost, I have to thank my research supervisor, Ing. Ladislav Čmolík, Ph.D. Without his assistance and dedicated involvement in every step throughout the process, this work would have never been accomplished. I would also like to show gratitude to my parents, Ivana and Ladislav, for their never-ending support.

I declare that I have written the presented research thesis by myself without undue help from a second person and that I stated all the information sources in accordance with the methodological guideline of the ethical principles in the preparation of university theses.

Prague, 27th May 2016

.....

Abstrakt / Abstract

Cieľom tejto práce je zoznámiť sa s dostupnými nástrojmi pre vizualizáciu dat na webe, vykonať nad nimi analýzu a porovnať tieto nástroje z hľadiska vizualizácie rôznych typov dát, možností interakcie s datami, možnosti rozšíriteľnosti a dostupnosti/ceny. Na základe analýzy si jeden z nástrojov vybrať a implementovať v ňom aplikáciu pre vizualizáciu adresárovej štruktúry z Google Drive. Aplikácia umožňuje priehľadnú vizualizáciu atributov súborov a adresárov. Aplikácia obsahuje dva prepojené pohľady na rovnaké data, v ktorých je možné s datami interagovať.

Kľúčové slová: bakalárska práca, JavaScript, HTML5, CSS3, SVG

The aim of this thesis is to investigate available tools for visualizing data on the web, to analyze and evaluate these tools by visualization possibilities of various data types, interaction possibilities with the data, extension options, and availability/price. Based on the analysis to select one (or more) of these tools and to implement the application for visualizing filesystem on Google Drive. Application is able to visualize attributes of files and directiories. Application has two connected views to same data, interacting with each other.

Keywords: bachelor thesis, JavaScript, HTML5, CSS3, SVG

Contents /

Tables / Figures

Chapter 1
Introduction

Data visualization is not as young discipline as many might assume. The idea of using pictures to better and faster understand huge amounts of data has been around for centuries. From maps and graphs to pie charts invented by William Playfair in 1801, we can clearly conclude that people knew about advantages of eye-brain connection for a very long time. Old Japanese proverb "a hundred listenings do not compare with one look" underlines the fact. The reason is how human visual system works. According to the study of Koch et al. [1], approximate bandwidth of human retinas is about 8960 kilobits per second. Once that data arrive to a brain, they are processed very fast by parts of brain which excel at edge detection, shape recognition and pattern matching. Not only data visualization helps people to process lots of data, it also helps to see patterns which are not clear by seeing raw numbers in tables or reports. Figure 1.1 provides a good example.

Nonetheless, it's technology, that made the visualization the hot topic of today. Even the application I have created as outcome of this thesis wouldn't be possible before the introduction of the first public working draft of HTML5 [2]. Computers make it possible to process huge amounts of data and output their graphical representation - and how to output the data is complicated topic for discussion. Use of color, shadows and shapes in data visualization is addressed by scientists and designers continually. There are lots of trade-offs between computer and humans limitations, and also display limitations– every pixel is important. I will briefly touch some of them in this work.

Data visualization as an utility, has the potential to transform a large part of the industry. Companies will be able to understand their data quickly, discover emerging trends–both in the business and in the market–which can give them edge over the competition, and identify relationships and patterns. It allows access to challenging data sets which would otherwise require hours to identify. However, to visualize the data, one must understand them and their structure, determine what needs to be visualized and what kind of information needs to be displayed. Therefore, there is need to know an audience and to understand how it processes visual information–topics of accessibility emerge. Finally, use a visualization which expresses the information in the simplest and the most understandable form.

	A	B	C	D
1	**Item**	**Color**	**Date**	**Amount**
2	T-Shirt	#ff6633	15.1.2015	4
3	Hat	#ccffaa	15.1.2015	3
4	Hat	#ff6633	16.1.2015	3
5	Hoodie	#ff0066	16.1.2015	5
6	Cup	#ccffaa	16.1.2015	2
7	T-Shirt	#ff6633	17.1.2015	12
8	Cup	#aaaa33	17.1.2015	1
9	Hoodie	#ff0066	17.1.2015	4
10	Hat	#ff6633	17.1.2015	5
11	Hoodie	#aaaa33	18.1.2015	5
12	T-Shirt	#ff6633	18.1.2015	3
13	Cup	#ccffaa	18.1.2015	8
14	Hat	#ff0066	18.1.2015	5
15	T-Shirt	#ff6633	19.1.2015	20
16	Hat	#ccffaa	19.1.2015	1
17	...	...	...	...

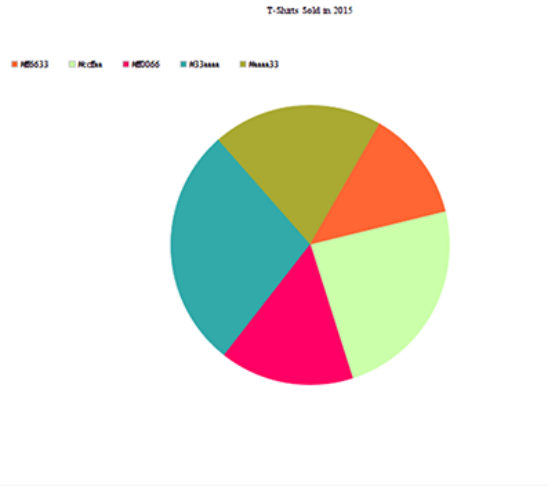


Figure 1.1. Two different pictures of the same data. In the second one, it is clear that #33aaaa colored T-Shirt is the best seller, so company should focus on making more of those. Spreadsheet requires a lot of time to figure this out, and computed data can be harder to remember than a large bluish cone.

1.1 Aim of thesis

Typically, administrator manages Google Drive service which the business pays for. His job is to share/unshare files to company employees, track space usage, or to see how many files with given types are present in specific directory. Graphical interface provided by Google is very similar to typical GUI provided in Windows or OSX–enabling to navigate through hierarchy and do some operations with files. However, if we want to get specific information about group of files or file system as whole, we need to do it manually, or run some filter script on top of it to see the results. One of the specific use cases of administrator managing the drive is to find some old multimedia files which are not needed anymore and take too much space. This is not a problem when the file system content is small. But as the file system content grows, it becomes more problematic to get the information we want and even worse to remember it, even for a short amount of the time. Visualization can help with these problems.

The aim of this work is to create an application which will help from administrators in large corporations to common household users to spot information and patterns that they seek in their Google Drive file system. In this work I choose and implement the most suitable visualization techniques for the following generalized tasks:

- To navigate through hierarchy and see file system structure
- To spot files by their size
- To spot files by their types
- To spot files by their owners and permissions to these files

In addition, visualizations are in single views and these views interact with each other. The interactions are following:

- Changing the color of an attribute in one view translates to another and vice versa
- Changing the depth of visualized hierarchy in one view translates to another view and vice versa (depends on user)
- Actual node selected in one view may translate to another view and vice versa (depends on user)

The implementation of the application is based on the analysis of available tools for visualizing data on the web and choosing the most suitable visualization techniques for the given tasks.

1.2 Structure

Following chapter presents analysis of available JavaScript tools for visualizing the information on the web. I document features and limits of given libraries/frameworks and choose the most suitable one for my work.

Chapter 3 presents building the application. In the beginning I investigate Google Drive API to see how a file meta data objects look like and what type of information I can work with, then I choose the most suitable visualization techniques for the given tasks. I also design user interface and describe other tools I use for front-end and back-end part of the application. In the end of the chapter I provide implementation details of the chosen visualizations.

Chapter 4 presents future work, related work and conclusions.

This work also contains well-commented source code of the application.

Chapter 2
Analysis of libraries

2.1 Introduction

This chapter presents the analysis of available JavaScript libraries and frameworks for data visualization. I outline the most popular and relevant JavaScript tools which are free to use. I investigate what the given tools are able to do, what kind of data are they able to visualize, if they are still maintained, if the code is readable and easy to extend and finally, if they are easy to learn.

In the end I choose one tool and analyze it in detail.

2.2 Basic analysis of libraries

2.2.1 Arbor.js

`Arbor`[1]) is a graph visualization library built with web workers and jQuery. Rather than trying to be an all-encompassing framework, arbor provides an efficient, force-directed layout algorithm plus abstractions for graph organization and screen refresh handling. Developer can decide between canvas, SVG or positioned HTML elements.

- Suitable for hierarchical, relational and tabular data visualization. As shown in Figure 2.1, Arbor.js is suitable for creating interactive file system visualizations.
- Canvas, SVG and HTML manipulation. Data interaction via collapsible nodes and manipulating with nodes position.
- Implemented gravity and physics for graphs.
- Extensive and readable documentation.
- MIT license. Possible to extend the code.
- Not readable and uncommented code. Hard to change the code.
- Depends on `jQuery`[2]).
- Last github commit in May 2012. The project probably stopped to develop.

[1]) `http://arborjs.org`
[2]) `https://jquery.com/`

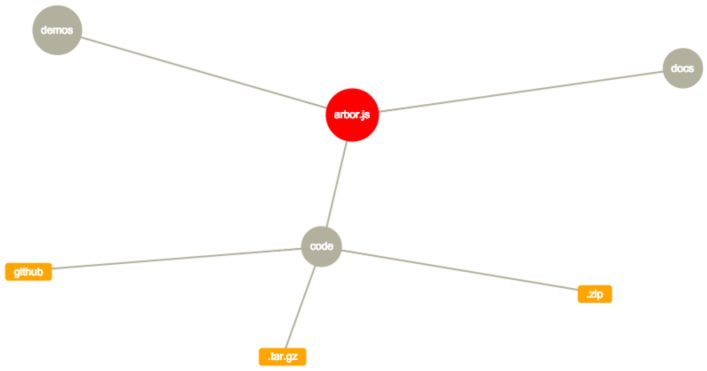


Figure 2.1. Hierarchical data visualization with Arbor.js

2.2.2 D3.js

`D3`[1]) is a JavaScript framework for manipulating documents based on data. D3 helps to bring data to life using HTML, SVG, and CSS. It puts emphasis on web standards and has full modern browsers support. Despite its similarity of syntax and semantics with jQuery, tools are completely different. With jQuery, developer directly manipulates elements, unlike with D3, developer provides the data to the library, defines callbacks and D3 manipulates elements. With D3 we typically bind data to some shape, define how it should be rendered, and D3 makes DOM updates. Note that we always have access to data, so we do not have to store the information to the ids, classes or data attributes of the elements for the future access. D3 uses update selections, making it quite efficient in the business of keeping the UI in sync with data changes.

- Virtually able to visualize any kind of data. Figure 2.2 shows scatter plot matrix.
- Unlimited interaction possibilities.
- Huge, scalable framework with variety of layouts for graphs, trees, charts, diagrams, dendograms, treemaps, parallel coordinates, `and so on`[2]).
- Active github, lots of tutorials and big community.
- BSD license. Possible to extend the code.

[1]) `http://d3js.org/`
[2]) `https://github.com/mbostock/d3/wiki/Gallery`

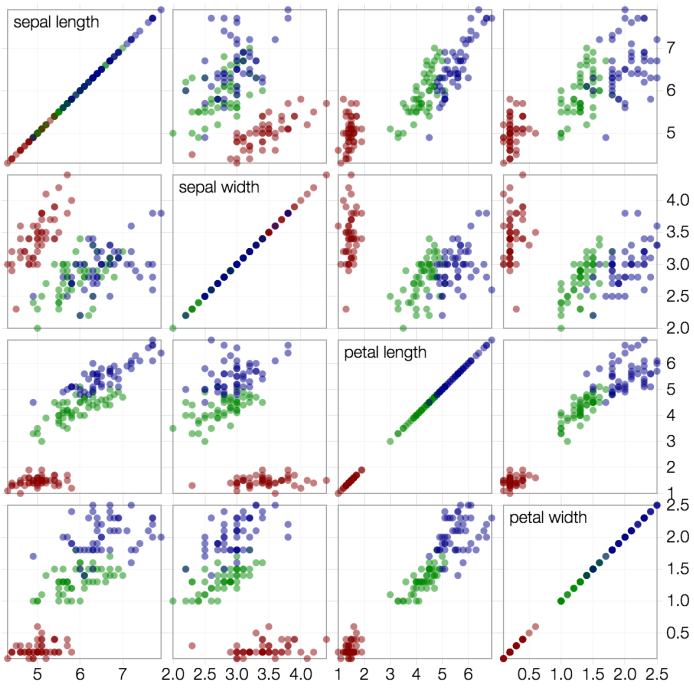


Figure 2.2. Scatter plot matrix is good way to roughly determine if we have a linear correlation between multiple variables.

2.2.3 Cola.js

`Cola`[1]) is an open-source library for arranging HTML5 documents and diagrams using constraints. It alters the behavior of other data visualization libraries like D3, Cytoscape or svg.js. The core layout is based on a complete rewrite in JavaScript of the C++ libcola library. Cola also contains D3 force layout adapter which achieves i.e. more stable layout than D3 and provides constraints such as alignments or grouping. In terms of learning curve, programmers can develop in Cola as they were developing in D3. However, there are few minor differences.

- Easy to learn and use for D3 developer.
- Project is alive on github and continues to develop.
- It is not a standalone tool, more like an extension layer above either D3, Cytoscape or svg.js.
- MIT license. Possible to extend the code.

2.2.4 Cubism.js

`Cubism`[2]) is a D3 plugin for visualizing time series. It can be used to create real-time dashboards. Cubism reduces server load by polling the most recent values. Despite loading is asynchronous, rendering is synchronous, thus it improves both performance and readability. It has built-in support for fetching data with `Graphite`[3]) and `Cube`[4]),

[1]) `http://marvl.infotech.monash.edu/webcola/`
[2]) `http://square.github.io/cubism/`
[3]) `http://graphite.wikidot.com/`
[4]) `http://square.github.io/cube/`

and can be readily extended to fetch data from other sources. Figure 2.3 shows time series of scalar numeric data and basic operations with them.

- Suitable for data visualization based on time.
- No interactivity options unless additional D3 code is used.
- It is D3 plugin, thus D3 knowledge is required.
- Last github commit on April 2014. Project probably stopped to be developed, although it still works well with no deprecation warnings.
- Apache license. Possible to extend the code.

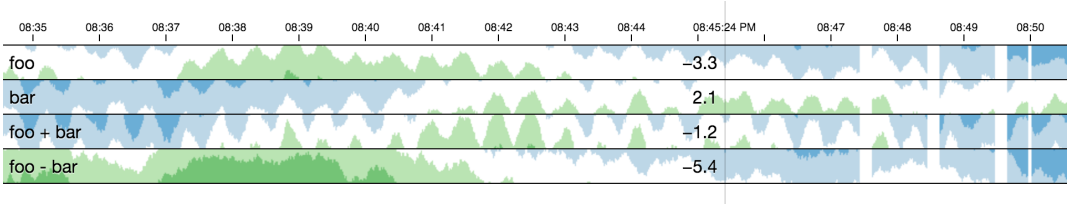


Figure 2.3. Time series visualization with Cubism.js. Positive values are green and negative values are blue. With mouse over the drawing we can inspect the values.

2.2.5 Google Chart Tools

`Google Chart Tools`[1]) is library for creating variety of charts. It is worth pointing out area charts, bar charts, bubble charts, donut charts, treemaps, maps, scatter charts and so on. Library is powerful and simple to use. It just requires to create library object with chart type name, and pass data with options to it.

- Perfect to create common visualizations in short time and with very little knowledge. Figure 2.4 shows pie chart created just with 3 lines of code.
- Not suitable if advanced and highly customized visualization with variety of interactions is needed. Tool has its boundaries which cannot be crossed.
- Can visualize scalar, vector, tabular and relational data.
- It is maintained by Google, always up to date, on par with web standards, and with full browser support.
- Free to use. API is provided minified, code is not readable and impossible to extend.

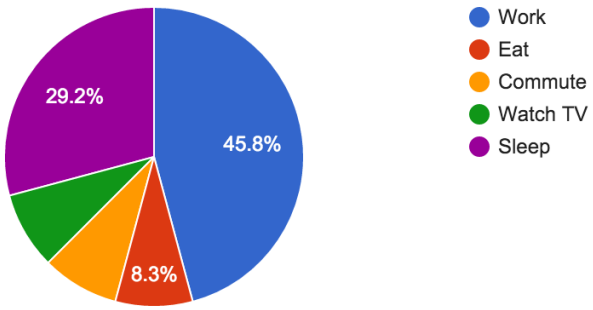


Figure 2.4. Pie chart showing day activities of a person, created with Google Chart Tools.

[1]) `https://developers.google.com/chart/`

2.2.6 Kartograph

`Kartograph`[1]) is a simple and lightweight framework for building interactive map applications without Google Maps or any other mapping service. It was created with the needs of designers and data journalists in mind. It offers versions for Python and JavaScript. It can cluster data in maps, visualize it in pie charts and so on. Figure 2.5 shows visualization of crime rate in the major cities in the USA.

- Can visualize tabular and relational data in geographic maps.
- Depends on jQuery and `Raphael`[2]).
- Project github is somewhat inactive. Although last commit is from July 2015, most of the code was committed in 2012.
- LGPL license. Possible to extend the code.

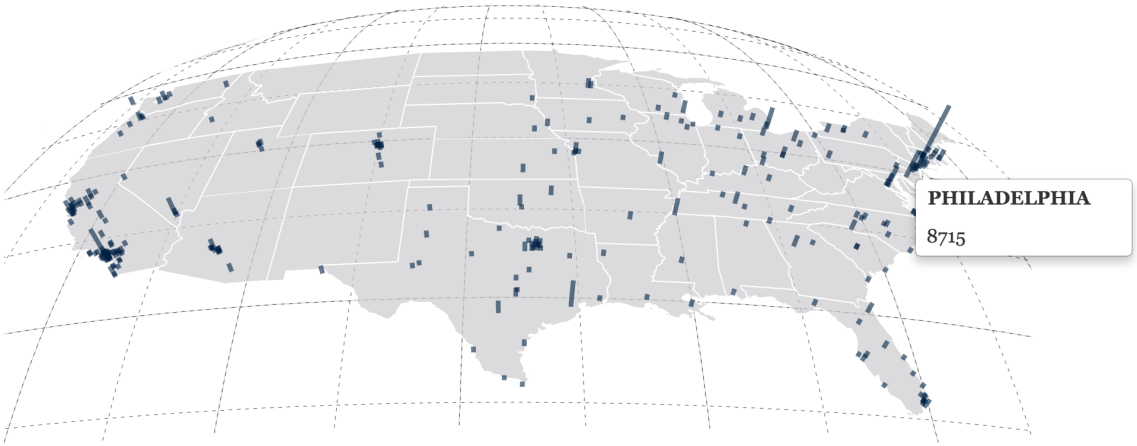


Figure 2.5. Interactive map created with Kartograph. Information is visualized via 3D bar charts.

2.2.7 NVD3.js

`NVD3`[3]) is a collection of re-usable chart components built with D3. Components are highly customizeable. Figure 2.6 shows example of stacked area chart.

- Collection of pie, line and bar charts with interaction.
- Great built-in interactivity.
- Usable for scalar, vector, tabular and relational data.
- Depends on D3.js.
- Active github.
- Apache license. Possible to extend the code.

[1]) `http://kartograph.org/`
[2]) `http://raphaeljs.com/`
[3]) `http://nvd3.org/`

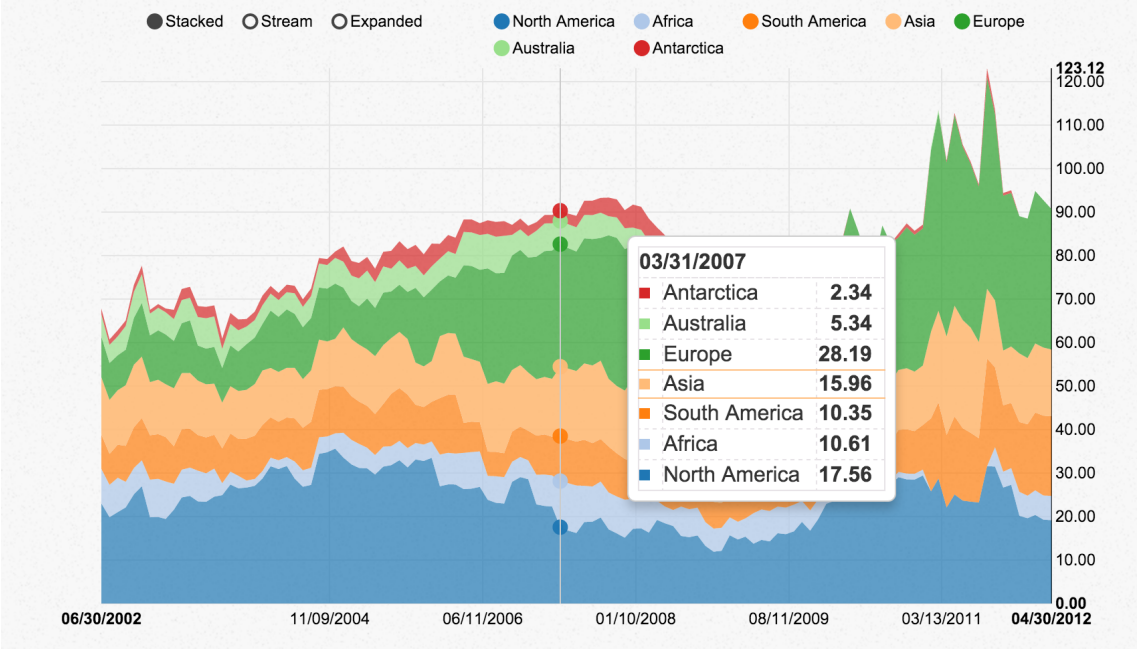


Figure 2.6. Stacked area chart component of NVD3. It can be switched to stream and expanded mode, and variables can be disabled by clicking on circles. By hovering mouse cursor over the graph, the data are shown in detail.

2.2.8 Paper.js

`Paper`[1]) is an open source vector graphics scripting framework that runs on top of the HTML5 Canvas. It offers comfortable and clean programming interface to create and work with vector graphics and bezier curves.

- It is too abstract and focused generally on 3D graphics, rather than focused on data visualization. Figure 2.7 shows path intersection detection.
- Supports every browser which supports Canvas.
- Active project github and twitter.
- Lots of tutorials on web page.
- Easy to learn but hard to master.
- MIT license. Possible to extend the code.

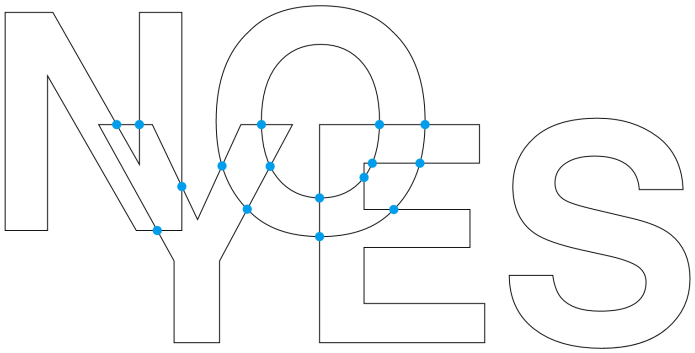


Figure 2.7. Path intersection detection created with Paper.js. Words are moved by user and intersections are shown in real time.

[1]) `http://paperjs.org/`

2.2.9 Processing.js

`Processing`[1]) is library for data visualization, digital art, interactive animations, graphs, video games and so on. It uses popular `Processing`[2]) visual programming language.

- Huge framework focused on 3D graphics in general.
- Lots of plugins created by big community.
- Active github, forum, blog and lots of tutorials.
- MIT license. Possible to extend the code.

2.2.10 Raphael

`Raphael`[3]) is small library to simplify work with vector graphics. It is not focused on data visualization, even though it is possible, as Figure 2.8 shows.

- Simple
- Same as Paper.js, it is not focused on data visualization.
- Perfect and readable documentation.
- Small community, inactive twitter.
- MIT license. Possible to extend the code.

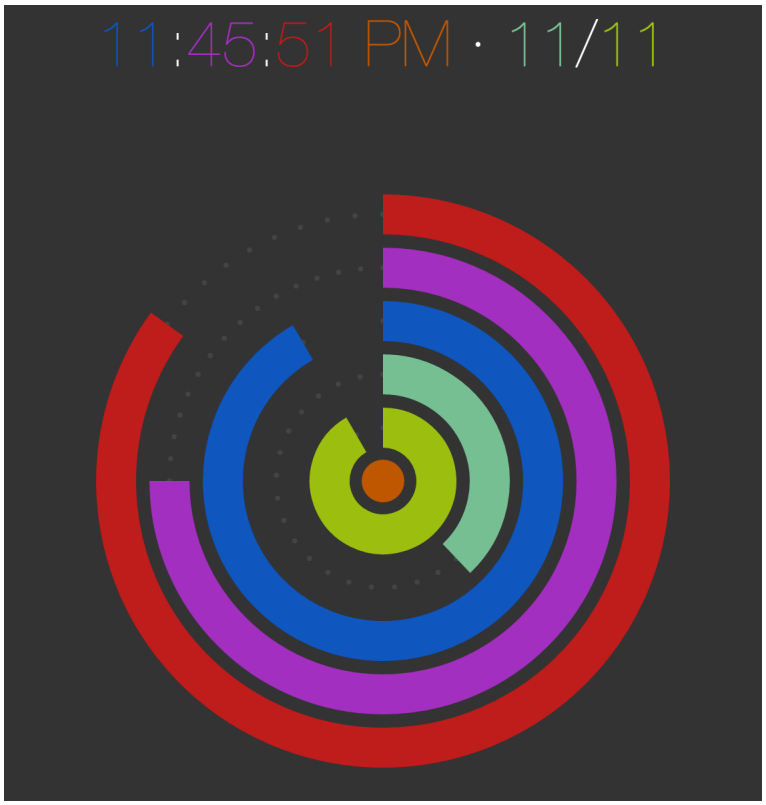


Figure 2.8. Time visualization with Raphael.

[1]) `http://processingjs.org/`
[2]) `https://processing.org/`
[3]) `http://raphaeljs.com/`

2.2.11 Rickshaw

`Rickshaw`[1]) is a D3 plugin for visualizing time series.

- More options compared to Cubism.js, for example scatterplots, line charts, stacked bars or color interpolation. Figure 2.9 shows time series of tabular data.
- Easy to use, only requires data in specific format.
- Depends on D3.js.
- Weak documentation, the only source of information is readme file on github.
- Inactive github.
- MIT license. Possible to extend the code.

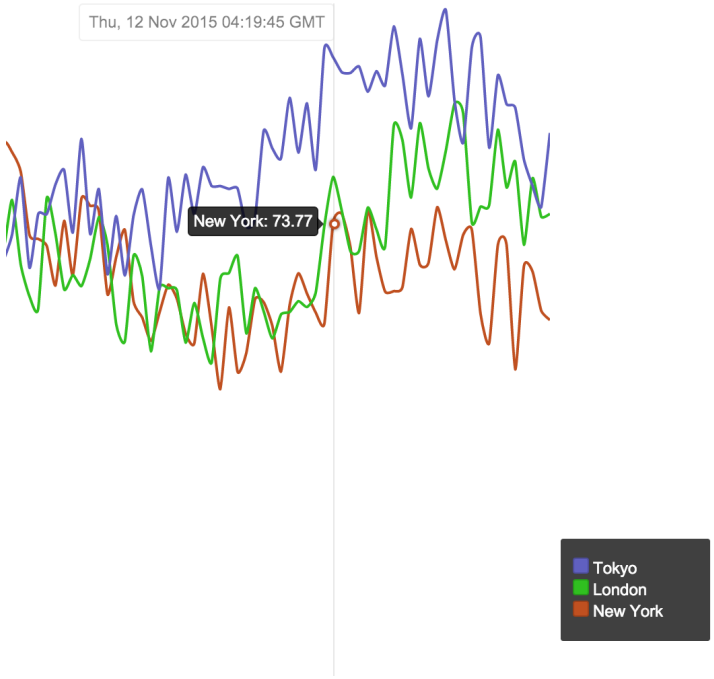


Figure 2.9. Time series of tabular data visualized in line chart.

2.2.12 Sigma.js

`Sigma`[2]) is library focused on graph drawing of networks. Example of such visualization is provided in Figure 2.10.

- Relational and hierarchical data visualization.
- Easy to use, only requires data in specific format.
- Interaction possibilities.
- It is small library, so web and documentation are short.
- Active github.
- MIT license. Possible to extend the code.

[1]) `http://code.shutterstock.com/rickshaw/`
[2]) `http://sigmajs.org/`

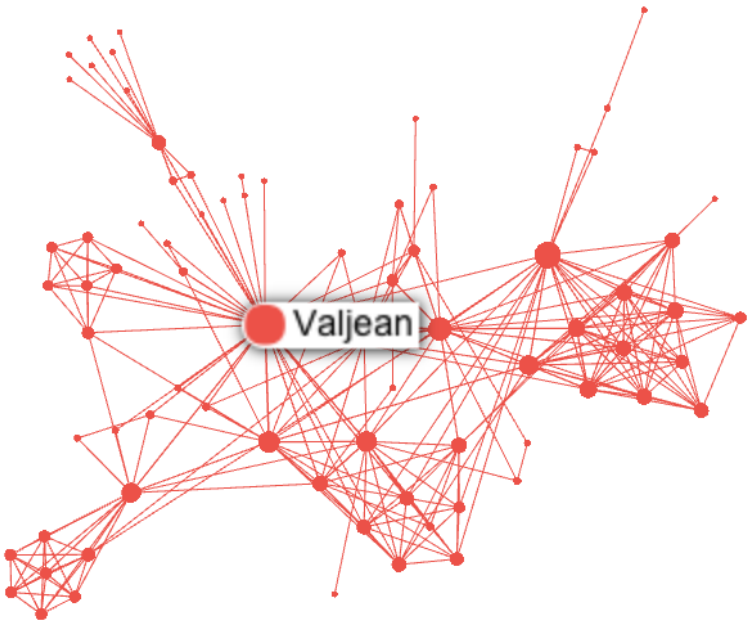


Figure 2.10. Network graph created with Sigma.

2.2.13 Three.js

`Three`[1]) is framework for programming in WebGL. It is layer just above WebGL, so programmer works with shaders, lights, scenes and so on. Just like Processing, it is possible to create anything 3D. Figure 2.11 shows 3D graph in Canvas.

- Huge community, arguably the most popular 3D framework for the web.
- Active twitter and github.
- Not focused on data visualization.
- Requires knowledge of WebGL.
- Great documentation and lots of examples and tutorials.
- MIT license. Possible to extend the code.

[1]) `http://threejs.org/`

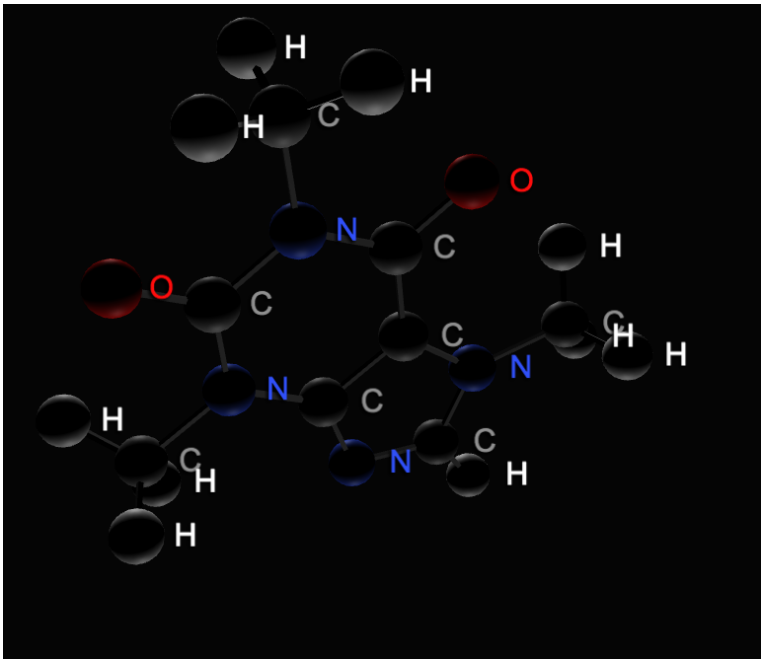


Figure 2.11. 3D representation of chemical element implemented with Three.js.

2.3 Final overview

In this section I provide the final overview of the analyzed tools, as shown in Table 2.1. I compare the libraries by activity of development, learning curve, and data types which can be visualized. Since there is a lot of data types categories, I discuss only three major categories: tabular data, relational data and spatial data. We can understand tabular data as items(rows) and their attributes(columns), relational data as items(nodes) with attributes and links between nodes and spatial data as grids with positions and cells and their attributes. A common example of spatial data can be seen in geographical maps.

Tool	Maintained	Difficulty	Data types
Arbor.js	No	Medium	T, R
Cola.js	Yes	Hard	T, R, S
Cubism.js	No	Hard	T
D3.js	Yes	Hard	T, R, S
Google Chart Tools	Yes	Easy	T, R
Kartograph	No	Medium	T, R, S
NVD3.js	Yes	Medium	T, R
Paper.js	Yes	Hard	-
Processing.js	Yes	Hard	T, R, S
Raphael	No	Easy	-
Rickshaw	No	Hard	T
Sigma.js	Yes	Easy	T, R
Three.js	Yes	Very hard	T, R, S

Table 2.1. Summary of analyzed tools. T stands for tabular, R relational, S spatial. Dash is undefined, because tools are not directly created for data visualization, however, there is possibility that they are able to create visualizations.

Difficulties are following: easy, medium, hard and very hard. Easy difficulty is just giving the data to the library controller function, medium requires learning library API environment which is not very complex. Hard is the same as medium, but more complex, and very hard requires additional knowledge of computer graphics.

2.4 Choice of framework

I analyzed arguably the most popular and relevant tools for visualization and graphics development on the web. For my topic, which is visualizing the Google Drive file system, I go for D3.js. The reason for it is big community, lots of examples and the power which is given to the developer. It does not necessarily mean, that other tools like Processing or Three.js are worse for the given issue, D3 just makes more sense, because it is data driven, and I work with data fetched from Google API.

Next section offers more detailed analysis of D3.js.

2.5 D3.js in detail

In this section I try to elaborate the basic idea behind D3 in introduction, following subsections outline techniques for creating and updating visualizations and conclusion sums everything up.

2.5.1 Introduction

D3 is described as a JavaScript library for manipulating documents based on data. D3 helps to bring data to life using HTML, SVG, and CSS. To understand this, we must forget how are we used to manipulate the DOM.

One of the common tools for visualizing on the web is SVG. Typical approach to visualize data with SVG using pure JavaScript or jQuery would be to iterate through some data array, and insert SVG element with calculated properties to the DOM. Then, if we need to update the visualization, the most naive approach is to select and remove an old SVG elements from DOM and insert the new ones. In the more specific cases we would need to compare old SVG elements which do not need to be removed, update the existing ones and insert or remove the changes. This is a mess, because we would also need to store old information to the DOM, typically to *id* or *data* attributes, if we want to compare it with new data.

This is not the case of D3. D3 offers three core methods, *data*, *enter*, and *exit*, which handle the DOM updates themselves using D3's `general update pattern`[1]). Programmer only has to tell them, what to do with existing, new and old elements. D3 takes a burden of complicated DOM manipulation off programmer's shoulders. This is what makes D3 unique.

To demonstrate it with example, I use the data from Table 2.2 and following code snippet to demonstrate how to visualize this data in treemap, provided in Figure 2.12. The code snippet shows both D3 code and output of it in the HTML. D3 code outlines how data focused D3 is. There is no need for any DOM operations, just bind *myData*

[1]) `http://bl.ocks.org/mbostock/3808218`

variable which contains data from Table 2.2 to an SVG *rect* element and set its properties like height, width and styles. Note that this is iteration and we always have access to every array item which is passed as argument to the function. In addition, *width* and *height* properties have been already calculated by *d3.layout.treemap* based on their size property, and appending the SVG *g* and *text* elements is also not outlined there and it needs to be done. I wrote it to SVG output part of the code to make it consistent with Figure 2.12. Aim of this example is just to demonstrate, how to access the data and create their graphical representation in D3. Also to clear any confusion, the *svg* variable is already created by D3 by appending it to a DOM element, for example a *div* container.

```
//D3 code
 svg.selectAll("rect")
    .data(myData)
    .attr("width", function(d) {return d.width;})
    .attr("height", function(d) {return d.height;})
    .style("fill", "#ffbb78")
    .style("stroke", "#fff");

//SVG output in the HTML file
<g class="cell">
    <rect x="0" y="0" width="187" height="131" fill="#ffbb78"
        stroke="#fff">
    </rect>
    <text x="93.5" y="65.5" text-anchor="middle">
        The Wold of Wall Street
    </text>
</g>
<g class="cell">
    <rect x="187" y="0" width="227" height="294" fill="#ffbb78"
        stroke="#fff">
    </rect>
    <text x="300.5" y="147" text-anchor="middle">
        Man of Steel
    </text>
</g>
<g class="cell">
    <rect x="0" y="131" width="187" height="163" fill="#ffbb78"
        stroke="#fff">
    </rect>
    <text x="93.5" y="212.5" text-anchor="middle">
        The Great Gatsby
    </text>
</g>
```

Film	Veľkosť v GB
The Wolf of Wall Street	2
The Great Gatsby	2.5
Man of Steel	5.45

Table 2.2. Multimedia files and their size.

Figure 2.12. Visualized size of multimedia files using treemap.

2.5.2 Selections

D3 employs declarative approach to modifying the DOM. Compared to imperative approach of `W3C DOM API`[1]), it makes the code more readable and understandable. Following code demonstrates the same thing done by pure JavaScript and D3.

```
//JavaScript using W3C DOM API
var paragraphs = document.getElementsByTagName("p");
for (var i = 0; i < paragraphs.length; i++) {
  paragraphs.item(i).style.setProperty("color", "white", null);
}

//D3
d3.selectAll("p").style("color", "white");
```

2.5.3 Dynamic properties

Compared to jQuery or Prototype, styles, attributes and other properties can be specified as functions of data in D3, not just simple constants. This is the place where we can calculate properties of elements based on the data. Following code snippets shows styling of every even and odd paragraph.

```
d3.selectAll("p").style("color", function(d, i) {
  return i % 2 ? "#fff" : "#eee";
});
```

2.5.4 Appending the elements

Initially, we want to append SVG to container, or to append SVG elements to the group. This is done with D3 *append()* method. Note that in contrast with jQuery, D3 *append()* returns created element, so we can continue to work with it, as demonstrates the following code snippet.

[1]) `https://www.w3.org/DOM/DOMTR`

```
var svg = d3.select("body")
            .append( "svg" )
            .attr("width", 100)
            .attr("height", 300);
```

2.5.5 Data

The *data()* method is the core of D3. It binds the data to document elements. In the easiest case, the array is bound in the order. Specifically, the array of [200, 300, 400] binds value 200 to the first rectangle, 300 to the second and 400 to the last. Also, if we want to update the data, we use this method and D3 figures out which SVG elements need to be redrawn. This method was already mentioned in the previous example, so I won't go to more details.

2.5.6 Enter and Exit

The *data()* method only works for updating the arrays of the same length. The *enter()* method declares, what to do with new data, while *exit()* specifies, what to do with the old, untouched data. Exit is typically used, if we want to remove old elements from the DOM, and enter vice versa. Following code demonstrates the example of data manipulation in the both cases.

```
var newData = [800, 200, 300, 600, 800];

//two elements are new, append them to the svg
svg.selectAll( "rect" )
   .data( newData )
   .enter()
   .append( "rect" )
   .attr( "x", function(d,i) {return i*2;} )
   .attr( "width" , 15);

var evenNewerData = [50, 2, 3];
svg.selectAll( "rect" )
    .data( evenNewerData )
    .append( "rect" )
    .attr( "x", function(d,i) {return i*2;} )
    .attr( "width" , 15);

//two elements are left, remove them
selection.exit()
    .remove();
```

2.5.7 Conclusion

In this section I demonstrated how D3 works and how it lets a programmer to focus on what he really is doing–visualizing the data. It handles the DOM updates really fast and code is readable and maintainable. But the framework itself is pretty hard to learn and even harder to master. I hope I have covered the basics in understandable way, and more advanced usage will be showcased in the application code of this work.

Chapter 3
Building the application

3.1 Introduction

The aim of the implementation part is to choose the visualization techniques for use cases described in the introduction of this work and implement web application using these techniques.

Firstly I analyze the Google Drive API and identify, what can be fetched from it. Application should be able to give every user the opportunity to visualize his own data on demand. Thereafter I choose the most suitable visualizing techniques for the given data and outline the business logic of the application. I set up the project environment and explain the project structure. I briefly touch the back-end, front-end and build tools which make development faster, cleaner, maintainable and with less bugs. Finally, I outline the implementation of the chosen techniques for the visualization.

3.2 Google Drive API

In this section I examine the `Google Drive API`[1]) for JavaScript.

API provides the service for fetching Google Drive metadata asynchronously via AJAX. It is available for multiple languages including Java, Python, PHP, Ruby, JavaScript and so on. It requires to create a project in `Google Developers Console`[2]). This is the control point of the application where we can see quotas of enabled APIs and other settings of the project, like setting how the OAuth modal with look like. Figure 3.1 shows the OAuth modal of my project, which handles user authentication and authorization and pops up when user requests the Google Drive data.

Google Drive API maximum quotas are set to 1,000,000,000 requests per day with possibility to apply for higher quotas.

One API call returns array of 100 metadata objects and next page token. In order to get the whole Google Drive structure, I need to call the API in the loop until no next page token is received. Getting the Google Drive structure by parts is good, because I can create a loader which gives feedback to users about loading progress. Implementation of the loader service is in *app/services/DriveAPILoader.js* file.

I have examined the fetched API object structure in Table 3.1. We can see that it has some good attributes for visualization, i.e. creation and modification times, size, mime type, or owners and parents, which can be visualized as relational data.

[1]) `https://developers.google.com/drive/v3/web/quickstart/js`
[2]) `https://console.developers.google.com/`

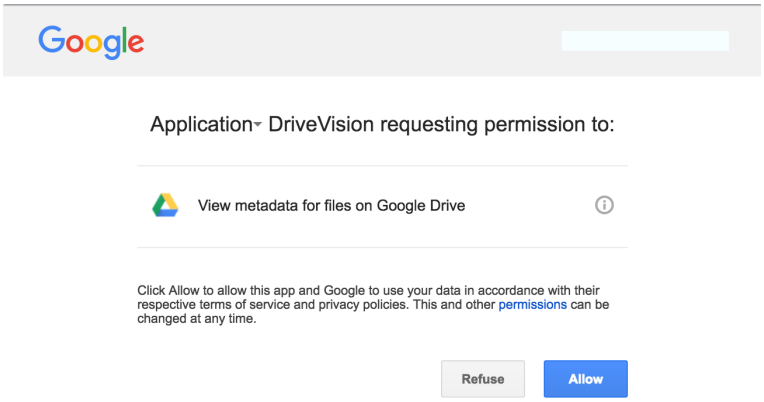


Figure 3.1. Google handles security automatically. DriveVision is the name of the project created in Google Developers Console.

Attribute	Type	Commentary
capabilities	Object	Object of capabilities, i.e. canWrite.
createdTime	String	Time of file create.
explicitlyTrashed	Boolean	Whether the file is trashed explicitly.
iconLink	String	Link to file icon.
id	String	Unique ID of the file in Google Drive.
isAppAuthorized	Boolean	Whether opened by the requesting app.
kind	String	This is always drive#file.
lastModifyingUser	Object	Who modified a file most recently.
mimeType	String	Type of the file.
modifiedByMeTime	String	The last time the file was modified by me.
modifiedTime	String	The last time the file was modified.
name	String	Name of the file.
ownedByMe	Boolean	Whether it is owned by me.
owners	Array	Array of user objects, who own the file.
parents	Array	Array of strings, parents IDs.
permissions	Array	Array of user objects with permissions.
quotaBytesUsed	String	The number of storage quota bytes.
shared	Boolean	Whether the file is shared.
spaces	Array	Whether 'drive', 'appDataFolder' or 'photos'.
starred	Boolean	Whether the file is starred.
trashed	Boolean	Whether the file is in trash.
version	String	A monotonically increasing version number.
viewedByMe	Boolean	Whether the file has been viewed by me.
viewedByMeTime	String	Time of the last view-time.
viewersCanCopyContent	Boolean	Whether readers can copy the file.
webViewLink	String	A link for opening the file.
writersCanShare	Boolean	Whether writers can share the file.

Table 3.1. Object returned by Google Drive API represents single file metadata.

3.3 Choosing the techniques

In this section I reason about the techniques I use to visualize the Google Drive file system. In the introduction of this work I have presented the following tasks to visualize:

- To navigate through hierarchy and see file system structure
- To spot files by their size
- To spot files by their types
- To spot files by their owners and permissions to these files

Following the first task, the first thing which comes to mind when visualizing the data which have some hierarchy is to display them in some sort of graph with nodes and links between them. First we must define the terminology. Following definitions outline the graph, its paths and cycles according to Diestel [3].

Definition 3.1. A graph is a pair $G = (V, E)$ of sets such that $E \subseteq [V]^2$; thus, the elements of E are 2-element subsets of V. To avoid notational ambiguities, we shall always assume tacitly that $V \bigcap E = \emptyset$. The elements of V are the vertices (or nodes, or points) of the graph G, the elements of E are its vertex edges (or lines).

Definition 3.2. A path is a non-empty graph $P = (V, E)$ of the form $V = x_0, x_1, \ldots, x_k$, $E = x_0x_1, x_1x_2, \ldots, x_{k-1}x_k$, where the x_i are all distinct. The vertices x_0 and x_k are linked by P and are called its endvertices or ends; the vertices $x_1, \ldots, x_{k-1}$ are the inner vertices of P . The number of edges of a path is its length, and the path of length k is denoted by P_k. Note that k is allowed to be zero; thus, $P_kP_0 = K_1$.

Definition 3.3. If $P = x_0 \ldots x_{k-1}$ is a path and $k \geq 3$, then the graph $C := P + x_{k-1}x_0$ is called a cycle.

Diestel describes a tree as an acyclic graph, one not containing any cycles. The vertices of degree 1 in a tree are its leaves, the vertices of degree greater than 1 are nodes. We can understand the degree of vertex as the number of edges connection to it. Figure 3.2 shows an example of a non-trivial tree.

Figure 3.2. Non-trivial tree.

There are two categories of hierarchical data visualization–node-link diagrams and space-filling diagrams. Node-link diagrams are categorized by their layouts to indented layout, traditional layout, force layout, dendrogram, circular layout or circular dendrogram, hyperbolic trees and so on. Space-filling diagrams include, but are not limited to icicle diagrams, sunburst diagrams, circle packing, and Voronoi treemaps.

To fulfill the first task, navigating through hierarchy and seeing file system structure,

we can use any type of those diagrams. But not every of them scales good or uses the screen space well with growing amount of data. For example, dendrograms or traditional layouts, where root is on the top and leaves at bottom, grow to width very fast and need to be zoomed out.

I aim to use every pixel possible, because most of the users of this application do not have large screens. I do not expect the large, multimedia studios to use it. I expect the application to be used by admins, or common users, who want to control the files in the company cloud, or their own files, respectively. And those users usually work on desktops and laptops.

For this reason, the ideal candidates are all space-filling diagrams and circular layout, circular dendrogram and force layout from node-link diagrams. This application uses two visualization views, so choosing the visualizations able to cover all four cases is ideal.

The first task is already covered, and to visualize file sizes, all space-filling diagrams are ideal. I choose tree map visualization, because it makes sense to visualize files which take space on the storage by rectangles which take space on the screen. Also, compared to i.e. circle packing, tree map visualizes large data better, because rectanges can use 100 percent of screen space, while circles can not.

Johnson and Shneiderman [4] have proposed two basic types of tree map layout. First type is nested tree-map, containing nesting offset to better visualize hierarchy. Second type is common, non-nested tree map, eliminating the nesting offset. Figure 3.3 provides the experimental implementation of this type of tree map. In the figure, we can clearly see two basic problems. First, if we want to express the type information of files with color, we need to add borders, otherwise the information would split. But how to choose color and width of a border to not confuse the user? If we colorize border, user may think that it expresses some other information, i.e. creator of a file. In the figure is deeply nested directory structure, separated with black border. It visualizes size of files, and their types as well. There is also information about hierarchy–it is obvious, that smaller rectangles are children of the bigger ones covering them. But we can not clearly see which ones are directories and which ones are children of specific directories.

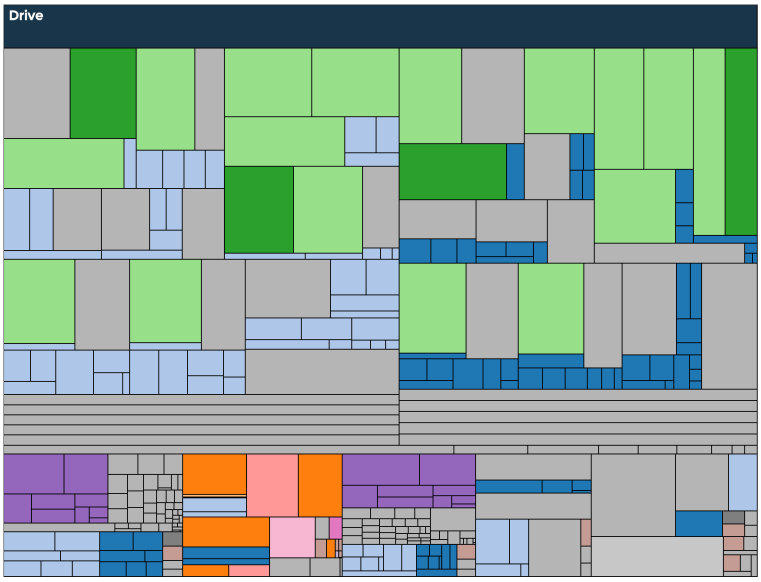


Figure 3.3. Non-nested tree map. Directories are gray-colored.

The nested tree map solves this issue. Following Figure 3.4, I experiment with adding offsets to rectangles to make hierarchical information absolutely clear. Note that this example uses the same data as shown in Figure 3.3, only visualizing the depth of four levels. And just now it becomes clear that structure is mostly composed from directories colored with gray.

I also removed borders and used shadows instead to separate the colors. According to Ware [5], it would be dull to live in a gray world, but we would actually get along just fine 99% of the time. We can divide color space into one luminance (gray scale) dimension and two chromatic dimensions. And it is the luminance dimension that is most basic to vision and understanding. Also, the light-sensing receptors in our eyes help us to recognize depth, by making the deeper directories darker. Following this statement, I set directories gray by default, manipulate the lightness by depth and separate parents from its children using shadows.

Nested tree maps have one disadvantage against non-nested tree maps. They are not able to visualize as many files. Added offsets require some pixels which would otherwise be used for more data. However, in this specific use case, user is not interested in visualizing small files deeply hidden in hierarchy most of the time. In if he is, he can still navigate deeper or navigate there from second visualization which will be more navigation-oriented.

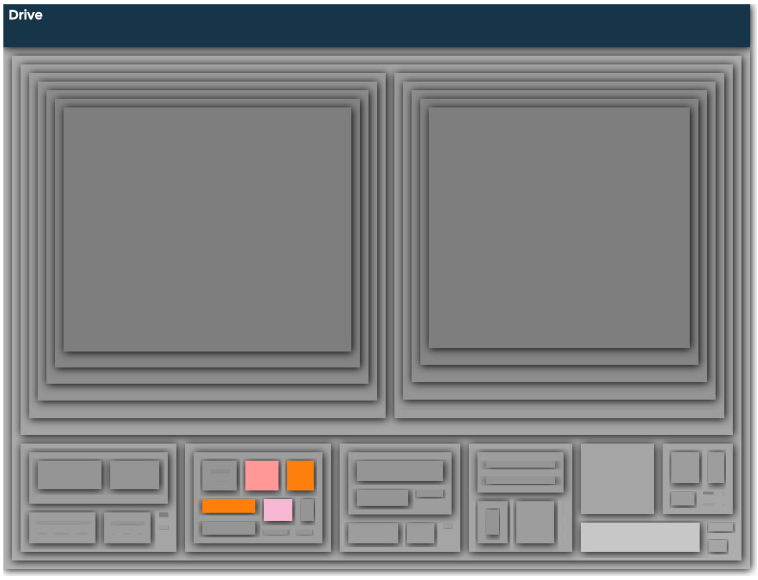


Figure 3.4. Nested tree map. Directories are gray-colored.

For the second visualization I have one task left–to visualize owners and their permissions. I investigated Google Drive metadata object in the previous section and I identify four types of permissions–owner, writer, commenter and reader. Most permissions are self-explanatory. Commenter is given to the users as a permission to comment Google files like docs, spreadsheets and so on.

I also want this visualization to be able to handle drive exploration better. Tree map gives us information about hierarchy, but we are not able to really see the whole hierarchy in the nested tree map–it has its pixels restriction, while the node-link visualization can be infinite.

To visualize owners and their permissions in node-link visualization, colors can be used. But how to colorize node, if it has multiple owners with their permissions is an issue. Figure 3.5 shows one possible technique to do that by grouping. In my specific case, I would group the graph by users and colorize areas behind with the chosen users colors. Areas would overlap, if a node has more than one owner. This is not good, graph might get chaotic and disarranged.

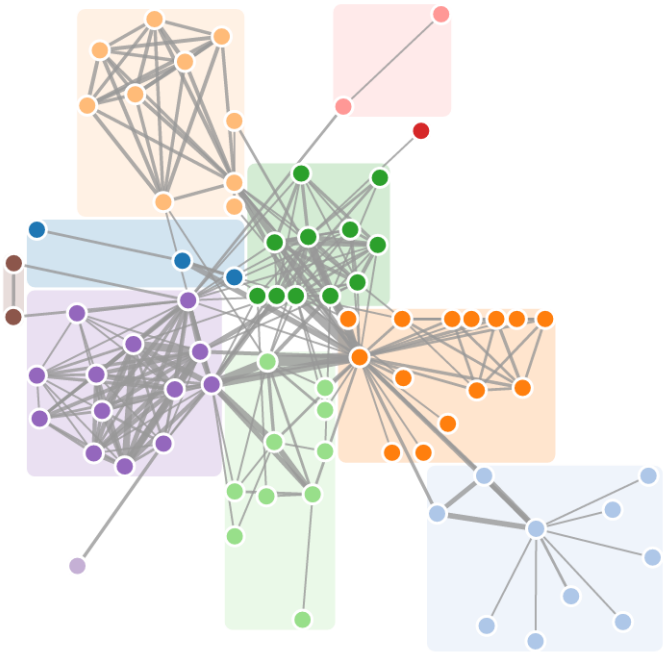


Figure 3.5. Common attribute is visualized by coloring the background.

Another potential technique is bubble chart. Each node in graph would be represented with inner bubble chart which contains circles representing users. Each circle is colorized with specific user color and permission attribute is visualized by radius of circle. Figure 3.6 presents this idea. I can visualize owners, writers, commenters and readers in the 4:3:2:1 ratio.

Figure 3.6. Bubble chart representing one node and visualizing three owners with owner, commenter and reader permissions.

Inner bubble charts in force layout is possible choice to do the task, but I also want to give the user a better by-type visualization than tree-map. According to Ware [5], a strategy for designing a visualization is to transform the data so it appears like a common environment. It complements the perception along with using colors, lightness, brightness, contrast and so on.

My experiment is to visualize file types with SVG drawings as nodes and leaves. Application uses three SVG drawings so far–closed folder, opened folder and file. There is room to add another SVG drawings for specific types, but I stick with those three in this work. The final idea is to visualize types in force layout using SVG, additionally colorize those SVGs by given type, and attach inner bubble charts next to them. Figure 3.7 presents an example outlining this idea on another data.

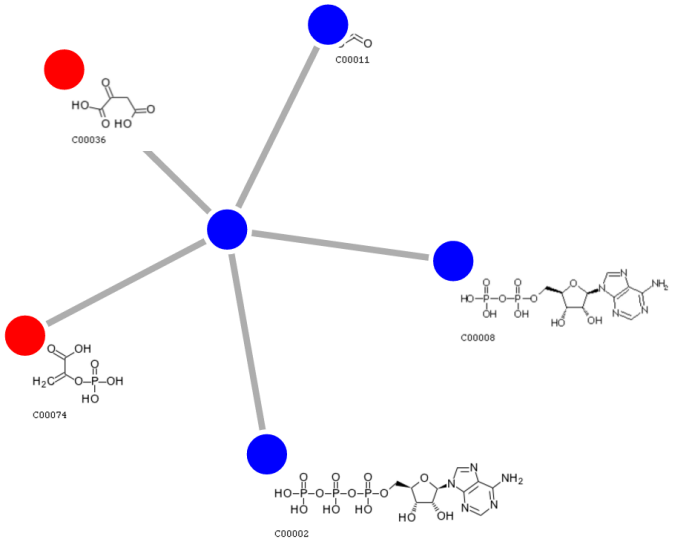


Figure 3.7. Force layout visualizing some information with colors, and another one with attached inner graphs.

I call this visualization mind map, because it has mind-mapping nature–nodes look like drawings. According to Buzan [6], mind map is a radial tree, diagramming key words in a colorful, radiant, tree-like structure. There is no strict way of doing the mind map, but Buzan recommends to stick to the following guidelines:

- Start in the center with an image of the topic, using at least 3 colors.
- Use images, symbols, codes, and dimensions throughout your mind map.
- Select key words and print using upper or lower case letters.
- Each word/image is best alone and sitting on its own line.
- The lines should be connected, starting from the central image. The lines become thinner as they radiate out from the center.
- Make the lines the same length as the word/image they support.
- Use multiple colors throughout the mind map, for visual stimulation and also for encoding or grouping.
- Develop your own personal style of mind mapping.

- Use emphasis and show associations in your mind map.
- Keep the mind map clear by using radial hierarchy or outlines to embrace your branches.

I won't necessarily stick to all guidelines. For example, the tree in my visualization is not radial. I use the force layout to use the screen space in the most efficient way. However, I start with the image of Google Drive logo. This lets user recognize the root apart from the other files.

In the mind map I visualize type attribute by using colors, and owners attribute by using the inner circles alongside the nodes. Circles are colored by the owner name. This approach works until the specific number of the owners, for more than that the colors begin to split, circles are smaller and it gets harder to identify specific owner. I compensate this by giving the zoom behavior to bubble charts on hover.

Finally, I connect the views. Views use the shared HTML5 `localStorage`[1]) data and they can run in separated windows. WHATWG [2] have introduced the local storage in the HTML5 as client alternative to cookies. Unlike cookies, the storage limit is far larger and information is never transmitted to the server. In addition, we can register listeners to it, so all windows are updated, when storage data change.

Following this approach I let users to run the application on two displays and therefore use the full potential of their pixels. Additionally, it opens the window for the application to scale and add more visualizations later without having a concern about the screen space usage.

The views interact with each other in the following ways:

- Changing the color of an attribute in one view translates to another and vice versa
- Changing the depth of visualized hierarchy in one view translates to another view and vice versa (depends on user)
- Actual node selected in one view may translate to another view and vice versa (depends on user)

These interactions are handled by sidebar and custom context menu on node click. I go to details in implementation section.

3.4 Project setup

3.4.1 Server

JavaScript has come a long way and can run on the server. `Node.js`[2]) with `Express.js`[3]) offer fast and simple setup, so I can setup application server in no time and focus on the client part. The whole server implementation is in the project root, in the *server.js* file. I don't use back-end in this application, but I need some server to run the project. This is the reason, why server implementation is very shot, and without any API calls.

Server additionally provides server side rendering which can speed up and initial load of application. It will be described in isomorphic section later.

[1]) `http://www.w3schools.com/html/html5_webstorage.asp`
[2]) `https://nodejs.org/`
[3]) `http://expressjs.com/`

3.4.2 Build system

According to Heilmann [7], we should avoid globals and modularize our JavaScript. The days when we used to include the scripts in the HTML page are gone. The common practice of today is to modularize JavaScript code, import dependencies and let the build system bundle all the JavaScript code into one file. This reduces the DNS operations and load on the server. Also we don't have to include every new created script in the HTML page. Code is additionally minified in the production. Minification is the process of removing all unnecessary characters from the source code without changing its functionality. Typically variables are shortened and white spaces are removed, including comments and/or logs.

I use `Browserify`[1]) as the build system. Browserify traverses every import from the entry JavaScript file and bundles the code into one single file. In order to do that without problems, JavaScript code needs to be modular and without global conflicts. Prusty [8] describes old ways of modularizing the JavaScript with immediately invoked expressions, and new ways with named exports. Named exports, along with many other great JavaScript features like classes, have been introduced by Ecma International [9], as new JavaScript standard, and it is also used in this project. However, to reach the full browser support, I use `Babel`[2]) to compile new ES6 standard to old ES5 standard.

All these tools are used under the `Gulp`[3]) task manager, so I can automate everything and do not have to execute all the previously mentioned tools manually. Gulp magic happens in *gulpfile.js* and I won't go to more details.

3.4.3 React and Flux

Along with previously mentioned tools, I use `React`[4]) library to build user interface and `Flux`[5]) architecture for business logic. Both are fairly new tools introduced by Facebook and became very popular recenly. In this subsection I briefly describe how they work and why I have decided to use them along with D3.

Both React and D3 share the same philosophy of "give me the data, tell me what to do with them, and I figure out which parts of DOM to update". The only difference here is that D3 uses its update selections and React uses the virtual DOM diff algorithm. React is based on reusable components with props and state. Props define the component and state defines current component state. Every time the state of the component changes, the component is re-rendered. I must emphasize that although this work is about visualizations with D3, ironically, D3 will run under React. React acts like a library, but with Flux it is a complete framework. Even though it looks like this gives me a boundaries, I can easily call D3 methods from Flux components and Flux actions from D3 components.

The following code snippet shows how a visualization component might look in this application. Note that *componentDidMount()* and *componentDidUpdate()* are methods inherited from React, and their code fires after component has been mounted or updated, respectively. In my application, these methods encapsulate D3 components behavior.

[1]) `http://browserify.org/`
[2]) `https://babeljs.io/`
[3]) `http://gulpjs.com/`
[4]) `https://facebook.github.io/react/`
[5]) `https://facebook.github.io/flux/`

```
import D3Component from '../d3/D3Component'; //ES6 modularity

//React component
class VisualizationComponent extends React.Component {
    constructor(props) {
        super(props);
        this.D3Component = null;
    }
    componentDidMount() {
        this.D3Component = new D3Component();
        this.D3Component.draw(); //start visualization
    }

    componentDidUpdate() {
        this.D3Component.update();
    }

    render() {
        //this React component is just a div
        //encapsulating my SVG visualization
        return (
            <div/>
        );
    }
}
export default VisualizationComponent; //ES6 modularity
```

This component is very simple and needs more code to make it work, but the goal was just to give a basic idea. I won't go further about React, because it is not a goal of this work.

Flux is the application architecture that was introduced by Facebook along with React, for building client-side web applications. It complements React's composable view components by utilizing a unidirectional data flow. Figure 3.8 explains how the data flow in Flux.

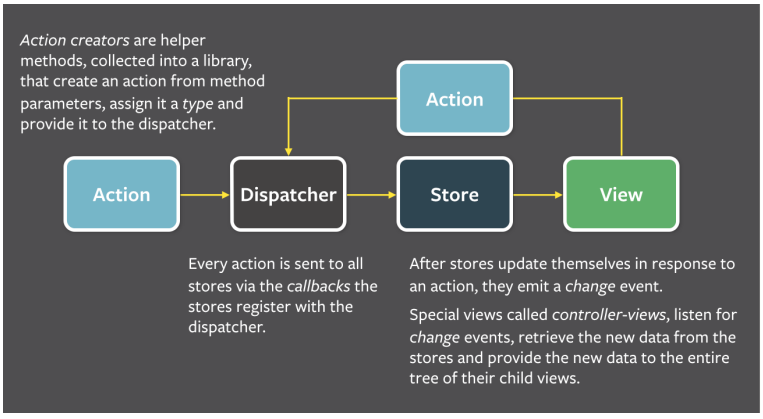


Figure 3.8. Flux architecture explained by Facebook. In the case of this work, our React and D3 components are in the view, stores contain state of the components and other business logic like Drive API fetched data. Note that stores, actions and dispatcher are all singletons.

3.4.4 Isomorphic applications with React and Node.js

My application is JavaScript heavy. I did everything I could to make it fast, from minification and bundling to using DOM efficient libraries like React and D3. But user still needs to spend some time until all the assets are loaded. Isomorphism solves this problem.

I have already mentioned in the Node.js section, that JavaScript can run on the server. So we actually can run our React code on the server, while user waits for the client-side code. React offers *ReactDOM.renderToString()* method, which can be found in the *server.js* file. This method runs the React code on the server, returns it as HTML, so user can see the UI while he waits for the client-side JavaScript. When the client side JavaScript is loaded, it bootstraps on the top of the HTML and from this point so on, application becomes client-side only.

3.4.5 Project structure

After I have introduced all the tools and features that I use, Figure 3.9 presents the final project structure. All the client-side code is in the *app/* folder. Along with Flux architecture's *actions/*, *components/* and *stores/*, there are folders for *services/*, which are used as an utilities, folder for D3 visualization components, folder for styles and *alt.js/* file with dispatcher. Bootstrap point of the application is in the *main.js/* file. *svg* folder contains classes responsible for creating SVG icons and in *factory* folder are factories responsible for creating these SVG classes.

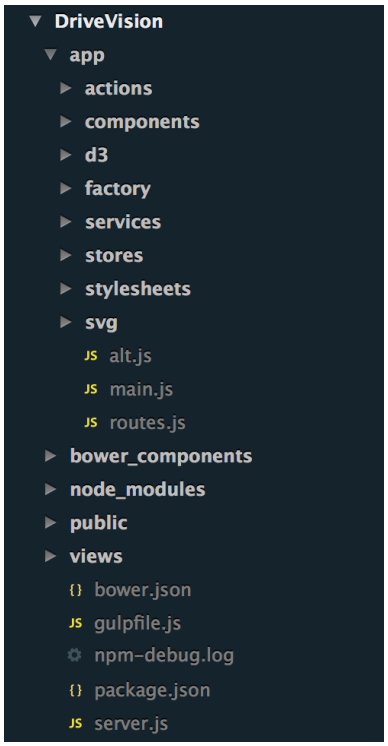


Figure 3.9. Project structure.

3.5 User-interface design

The great designer of Apple, Jony Ive [10] emphasizes the importance of prototyping in his biography. “But when you made a 3-D model, however crude, you bring form to a nebulous idea, and everything changes—the entire process shifts. It galvanizes and brings focus from a broad group of people. It’s a remarkable process.”

Although I do not create 3-D prototype, I want to outline the importance of having the application designed before starting the implementation. Application needs an user interface for handling interaction with visualizations. I believe that even that tiny portion of user interface should be designed before implementation to give the clear vision of how the application will look.

I have created simple paper mock-ups done in the `Balsamiq`[1]) to give the idea of the final look. It should be absolutely minimal, save the pixels for visualizations and users should be able to completely hide it from the screen. Following figures display all three use-cases of application use. Figure 3.10 previews initial load of the application. Sidebar is initially hidden are can be opened or closed. Figure 3.11 provides the designed mock-up for the mind map and tree map views.

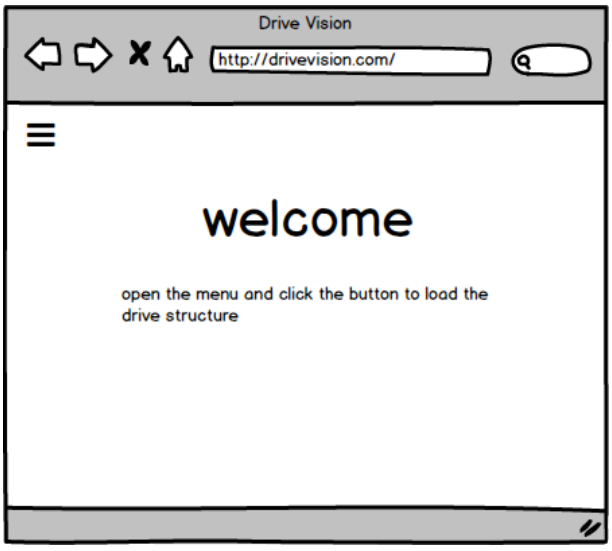


Figure 3.10. Paper mock-up of the home page. Sidebar is initially hidden. This is the only time when users see this page. After the drive structure is loaded, it is stored in localStorage and tree map loads on home page.

[1]) `https://balsamiq.com/products/mockups/`

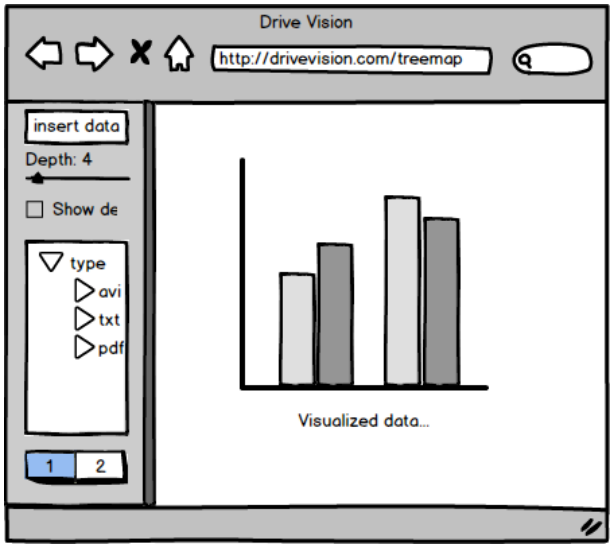


Figure 3.11. Paper mock-up of the visualization. Sidebar contains slider for choosing the depth. Additionally, there is checkbox to indicate whether to display the depth in the other visualizations or not. Next, there is a list of an colored attributes. When user clicks on attribute, a color picker shows up. Attributes are initially set up to the most fitting colors provided by D3. At the end are two buttons for switching a visualizations.

3.6 Implementation

This section explains the logic and implementation of visualizations. I do not go into deep details, just explain the basic idea how the application works under the hood.

There is a common interface for all the visualizations and visualizations extend from this interface.

```
class Visualization {
    create(){}
    update(){}
    delete(){}
}
```

The *create* method handles all initializations, *update* is executed when visualization needs to be redrawn and *delete* does the cleanup. This is the place for all D3 code. React components encapsulating these D3 classes abstract from this implementation using bridge pattern described by Gamma et al. [11]. With this approach I can have user-interface-related code and business logic code in React components and D3 code in *Visualization* classes. The React component-D3 component communication is following:

```
class ReactComponent extends React.Component {
    constructor() {
        /*...*/
        this.visualization = null;
    }
    componentDidMount() {
        this.visualization = new Visualization();
        this.visualization.create();
    }
    componentDidUpdate() {
```

```
            this.visualization.update();
        }
        componentWillUnmount() {
            this.visualization.delete();
        }
    }
```

The business logic for user interface operations like updating colors in Sidebar and so on is handled by Flux architecture. Implementation is in all *app/stores/* files. When action fires, components change state and re-render. *componentDidUpdate()* method is evoked and D3 visualization updates as well.

3.6.1 Tree map implementation

D3 provides *d3.layout.treemap()* algorithm to create a new tree map layout. Default settings are sorting by descending value, the default value accessor assumes each input data is an object with a numeric value attribute, the default children accessor assumes each input data is an object with a children array; the default size is 1×1.

```
var treemap = d3.layout.treemap()
    .children(function(d, depth) { return depth ? null : d._children; })
    .sort(function(a, b) { return a.value - b.value; })
    .ratio(height / width * 0.5 * (1 + Math.sqrt(5)))
    .round(false);
```

My algorithm skips some of the default settings. I re-calculate children values to add navigation behavior–children become at root level of size as we navigate deeper.

Also, D3 does not provide algorithm for nested tree maps and I have to calculate off sets by myself. The following code outlines adding offsets to layout calculated by D3.

```
d._children.forEach(function(c) {
    c.x = d.x + c.x * d.dx;
    c.y = d.y + c.y * d.dy;
    c.dx *= d.dx;
    c.dy *= d.dy;
    if (margin) {
        if (c.x === d.x) {c.x += mrg; c.dx -= mrg*2;} else {c.dx -=mrg;}
        if (c.y === d.y) {c.y += mrg; c.dy -= mrg*2;} else {c.dy -=mrg;}
    }
    c.parent = d;
    ref.layout(c, false, margin);
});
```

In this case, c is child and d its parent. x and y are coordinates of a rectangle on the screen and dx, dy are its width and height respectively. The idea is that if *margin* variable is true, we are going to add offsets. *mrg* variable is numeric value of the offset, in this case it is 10 pixels. If starting position of child is the same as starting position of its parent, push it by offset value and make it 2 times offset smaller. If it is somewhere else in the area, just push it.

This code has one issue–rectangles smaller than 2 times offset value get negative values and they are not drawn in drawing iteration. But user usually does not seek very small files in tree map and if he does, he can navigate deeper in hierarchy and

they become visible.

This is the basic logic of how I draw nested tree map. Other implementation details can be found in app/d3/TreemapImpl.js.

3.6.2 Mind map implementation

To create mind map, I use variety of layouts. The node-link force layout is created by cola.js. I have described cola.js in the analysis, but I do not go into details because I only use cola adapter for D3 force layout. The reason I use it is the more stable layout without jitter. The code looks exactly same as it would by using *d3.layout.force()*. In the following code I outline creating this layout using cola adapter.

```
var force = cola.d3adaptor()
    //d3.layout.force()
    .linkDistance(130)
    .size([ref.width, ref.height])
    .on("tick", function() {
        ref.tick(this, ref);
    });
```

ref holds the reference on *this*, because scope changes in iterations. Later in the code I pass the nodes data and links data, where links is an array of arrays of two objects being linked together. *tick* is function fired in time interval, trying to re-arrange the force layout.

To create SVG icons I append the svg path with predefined values to the node position calculated by force layout. I use the flyweight pattern described by Gamma et al. [11] to save the memory, so in memory I have just three SVG objects being drawn multiple times.

```
n.each(function(d) {
    let svgComponent = FlyweightSVGFactory.GetFlyweight(d.children ?
                       d.type+"_OPEN" : d.type);
    let el = d3.select(this);
    for (let path of svgComponent.getPaths()) {
        el
         .append("svg:path")
            .attr("d", path)
            .attr("fill", function(d) {
             return DriveDataUtils.findColorByAttr(ref.colorSettings,
               d.type);
            });
    }
});
```

Next, I append bubble charts to every node to visualize owners and their permissions to the given file using *d3.layout.pack*. Parent circle has no meaning and just reserves the space for children, and children circles represent each owner name with color and permission with its radius. Implementation of creating the each node is in *createIcon* in *app/d3/MindmapImpl.js* file.

3.7 Demonstration of the application

In this section I outline possible uses of both visualizations and sidebar controls along with final pictures of both visualizations.

Following list explains common use cases:

- Clicking on hamburger icon in top left corner expands side bar.
- Clicking on Google Drive logo in side bar executes authentication and authorization process of user and loads Google Drive files meta data. All client settings are reset.
- Clicking on the mind map and tree map buttons on the bottom of the side bars opens application in new windows with specific visualization.

Following list explains use cases of tree map:

- Clicking on the tree map rectangle navigates deeper in hierarchy.
- Clicking on the tree map label on top navigates back in hierarchy.
- Right clicking on top tree map label shows context menu with following actions: show in mind map.
- Hovering over the tree map rectangle shows more detailed information about the file.
- In the sidebar, clicking on the attribute text fires color picker with option to change the color of the given attribute in tree map.
- In the sidebar, moving the slider manipulates tree map depth.
- In the sidebar, checking the check box under the slider enables to manipulate the depth in other visualizations .as well

Following list explains use cases of mind map:

- Clicking on the mind map node svg icon expands or reduces it (depends whether it is already opened).
- Hovering over the mind map node inner bubble chart zooms it.
- Right clicking on top mind map node svg icon shows context menu with following actions: show in tree map, expand, reduce.
- Graph zooms in and out on mouse wheel.
- Dragging the nodes allow to rearrange them.
- Dragging outside the nodes moves whole graph.
- In the sidebar, clicking on the attribute text fires color picker with option to change the color of the given attribute in mind map.
- In the sidebar, moving the slider manipulates depth of graph which is expanded.
- In the sidebar, checking the check box under the slider enables to manipulate the depth in other visualizations as well.

Figure 3.12 and Figure 3.13 show final implementation of tree map and mind map, respectively.

Figure 3.12. Final tree map visualization. View is navigated to Drive/DriveVisios/public directory.

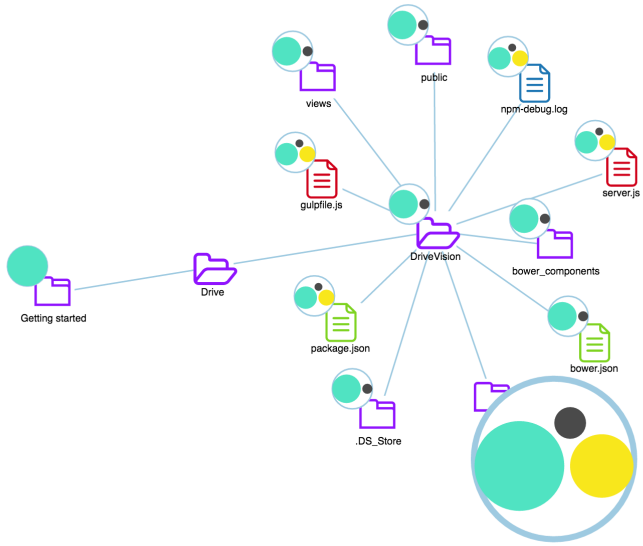


Figure 3.13. Final mind map visualization with zoomed-in inner bubble chart.

Chapter 4
Conclusion

In this work I have analyzed available JavaScript tools for visualizing data on the web and I have chosen D3 library to implement an application for visualizing Google Drive file system. Application uses two connected views which interact with data. First view is mind map, which is technically node-link tree graph with force layout and inner bubble charts, visualizing the hierarchy, file type attributes and file owners and their permissions. Second view is tree map, visualizing the hierarchy, file type attributes and file size.

The future work of this project is to add views. There are other attributes of Google Drive files which can be visualized to help users with another use cases, i.e. created time, number of files in directories and so on. Also aim is to upgrade the tree map with more SVG icons to visualize file types.

The most related work is `Mohiomap`[1]). In addition to Google Drive, Mohiomap visualizes Evernote, Dropbox and Box. In comes in paid plans and visualizes the file system with mind mapping. Figure 4.1 shows visualized file system in Mohiomap.

Mohiomap is mind mapping visualization service and does not provide other visualizations apart from user activity visualization in dashboard. This is the reason why I have decided to create this application–to give users more views and file attributes to visualize.

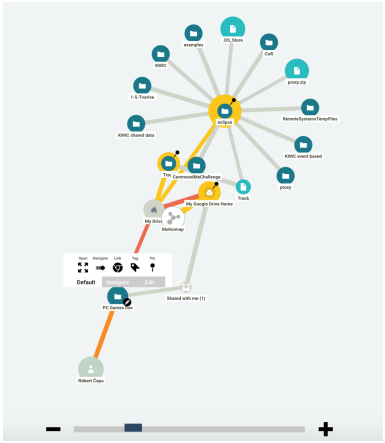


Figure 4.1. Bubble chart representing one node and visualizing three owners with owner, commenter and reader permissions.

[1]) `https://www.moh.io/mohiomap/welcome/`

References

[1] Kristin Koch, Judith McLean, Ronen Segev, Michael A Freed, Michael J Berry, Vijay Balasubramanian, and Peter Sterling. How much the eye tells the brain. *Current Biology*. 2006, 16 (14), 1428–1434.

[2] Web Hypertext Application Technology Working Group. *HTML 5*. 22 January 2008.
`https://www.w3.org/TR/2008/WD-html5-20080122/`.

[3] Reinhard Diestel. Graph theory. 2005. *Grad. Texts in Math*. 2005, 101

[4] Brian Johnson, and Ben Shneiderman. *Tree-maps: A space-filling approach to the visualization of hierarchical information structures*. In: *Visualization, 1991. Visualization'91, Proceedings., IEEE Conference on*. 1991. 284–291.

[5] Colin Ware. *Information visualization: perception for design*. Elsevier, 2012.

[6] Tony Buzan. *Make the most of your mind*. Simon and Schuster, 1984.

[7] Chris Heilmann. *JavaScript best practices*. 6 March 2009.
`https://www.w3.org/wiki/JavaScript_best_practices`.

[8] Narayan Prusty. *Learning ECMAScript 6*. Packt Publishing Ltd, 2015.

[9] Ecma International. *ECMAScript® 2015 Language Specification*. June 2015.
`http://www.ecma-international.org/ecma-262/6.0/`.

[10] Leander Kahney. *Jony Ive: The Genius Behind Apple's Greatest Products*. Penguin, 2013.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of reusable software components. 1995,

Appendix A
Installation

Application requires Node.js and NPM to run. To install on Ubuntu server:

```
sudo apt-get install nodejs
sudo apt-get install npm
```

Copy the CD contents to *folder_name*.

```
cd folder_name
npm start
```

Appendix B
Abbreviations and symbols

B.1 Abbreviations

API	Application program interface.
DOM	Document Object Model. Programming API for HTML a XML documents.
W3C	World Wide Web Consortium.
HTML	HyperText Markup Language.
SVG	Scalable Vector Graphics.
NPM	Node Package Manager.