Cljs-gravity

A reliable 3D force-directed layout running in the browser

Geoffrey GAILLARD*
University of Eastern Finland, Ïta-Suomen Yliopisto
Université de Lorraine, France

December 27, 2015

Abstract

This project's purpose is to design and implement a 3D graph visualization library running in the browser, using technologies like WebWorkers, Clojure (Script) and WebGL. This library sit on top of D3.js force-layout algorithm, which is a 2D implementation of the Barnes-Hut approximation algorithm, and THREE.js for 3D visualization.

This report cover design requirements, technology, and extension of the D3.js Barnes-Hut implementation to add the third dimension and how to project the mouse in a 3D space. It brings question on multi-threading, code isolation, fault tolerance and simplicity in a JavaScript library.

* https://github.com/ggeoffrey

Table of Contents

1. Introduction

1.1 Lack of a true dynamic 3D force-directed graphs

The web is full of JavaScript libraries for graphs generation, manipulation or visualization. But none of them give you the ability to generate a 3D, perspective, dynamic force-directed graph, **based on quasi-standard tools**. I think especially about nGraph[1]. nGraph is a fast graph manipulation and visualization tool that has been demonstrated being the fastest. But being the fastest in not always the goal. This project exist not to create the fastest 3D graph visualization tool but the more reliable. This mean finding the balance between security, performances, code maintainability and simplicity. There is already a lot of proven and powerful tools allowing graph manipulation and it is useless to create another one, so this project exist also to do one thing and do it well: display a graph and allow the user to manipulate the visualization, not the graph itself.

Data Driven Documents (D3.js) is a JavaScript library allowing DOM manipulation with data bindings in a functional style. It implements a force-directed graph layout based on the Barnes-Hut approximation algorithm. Including D3.js into your productions can bring you more than DOM manipulation. D3 bring data manipulation, data classification, events handling and is actively developed and maintained as one of the most popular GitHub repository. Using D3.js over nGraph is the first step towards a more reliable library.

1.2 Security

Developing for browsers mean dealing with JavaScript. Even if JavaScript is a powerful language, it is difficult to implement secure code with it. The main reason in regard to this project is side effects. In JavaScript collections are mutable and the memory is dynamically allowed and cleaned by garbage collection. Manipulating or displaying big trees of graphs can easily bring memory leaks and unwanted side effects. Worst, the best way to clone an object is to import a library to do it[2] or by serializing it to a string and reparsing it again. This bring two questions:

1. How can we be sure that our library will not affect the overall state of the user's application?
2. How can we be sure that the user application cannot affect our library internal state?

If users, by manipulating data, succeed to affect the internal state of our library, this

1 https://github.com/anvaka/ngraph (look at it, it's worth it !)

2 #1 result on Google for the question "Best way to clone an object in JavaScript": http://stackoverflow.com/questions/122102/what-is-the-most-efficient-way-to-clone-an-object

mean that our library have security leaks. It can make the web page crash or worst: display a wrong graph representation.

Ensuring a perfect isolation between the user and the internal library state is mandatory to guaranty stability and an accurate graph visualization.

1.3 Performances

A browser tab execute JavaScript code in a single threaded environment. This mean that long running scripts can block the user interface rendering or make the page appear as "not responding". A well designed library must avoid it even if heavy computation is involved.

With big graphs come big memory consumption and ensuring that no memory leak appear can be difficult in an environment where everything is mutable. How can you be sure that the user is not keeping a reference to a node? The library need a reliable and predictable way to ensure that memory will be available for garbage collection sooner or later.

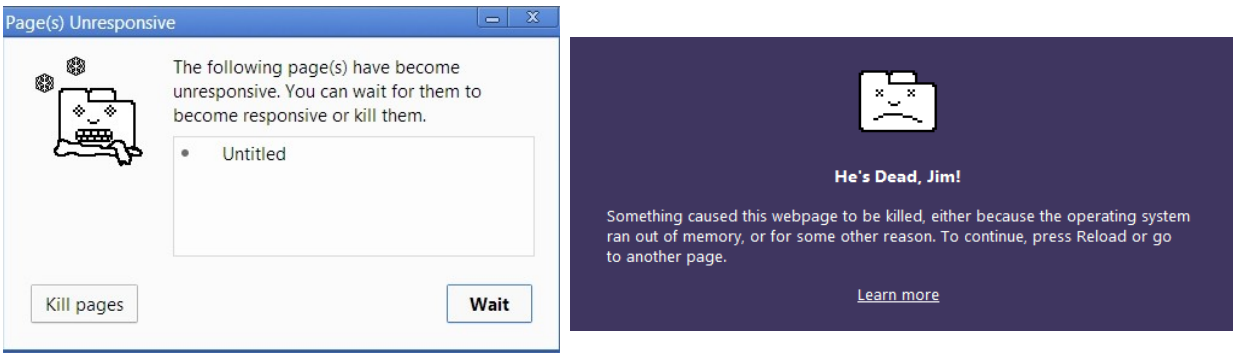


Figure: modern "x stopped working" and the horrible CSOD (Chrome Screen Of Death)

1.4 Simplicity

Users have habits and expect common sense or clever things from tools they use. To satisfy these needs, cljs-gravity expose only one function called **create**. This function return a graph instance allowing the user to listen to events and to inject data in the graph. Events name are explicit. If the user lost reference to the graph instance the memory is guaranteed to be made available for garbage collection. As this library sit on top of D3.js, the same API is provided for the force layout.

On the code side, code need to be as small as possible, clean, simple and self-explanatory to be easy to understand and maintain.

2. Methods

2.1 Clojure: elegant coding for a more civilized age

In the world of JavaScript libraries, developers can be sure about at least one fact: you can find anything that will do what you need, but you can never be 100% sure that it is doing it the good way. There is no standardized way to packages and distribute your code and worst, no standard way to write code, only "good practices". If you include a library in your project you can never be sure if the creator took care of the good practices or even if he is aware of them. The only solution is to check the code yourself or to assume that a popular project is well-driven and can be trusted. Secondly, even if JavaScript is expressive, it allows you to write spaghetti code easily[3] and can be a true nightmare to maintain. Finally, JavaScript objects are not truly typed and are always mutable. This bring problems for concurrency, stability and performances.

Clojure is a LISP dialect that compile to JVM or CLR bytecode and JavaScript. It put a strong emphasis on functional programming with pure functions and immutable data structures. The syntax is almost non-existent and Clojure programs are short, simple, expressive and allow total and symbiotic communication with the host virtual machine. In this project, Clojure brings three main advantages:

1. pure functional programming mean no side effects and predictable code,
2. immutable data structure make more reliable code and garbage collection easier,
3. Clojure philosophy allow us to write short and simple functions that chains in an elegant way, thus reducing the overall project size and improving maintainability.

Clojure provide simplicity[4] through its syntax. Namespaces are explicits, asynchronous code looks synchronous, private and public functions are explicitly annotated, files are small and the code is expressive.

2.2 Isolation and channel based communication

To implement the security aspect, the user data are deeply cloned and transformed to Clojure immutable data structure. Clojure data structures like lists, vectors, sets and maps are immutable by default. The user can affect the internal state with specific accessors. It ensure users cannot set or get any mutable object. To improve fault tolerance and security, the library is divided in 3 major parts:

1. the front part, sending events to the user and providing accessors,
2. the view part, listening to DOM events (click, mouse movements, etc.) and

3 You can write spaghetti code in every language but JavaScript is famous for it.

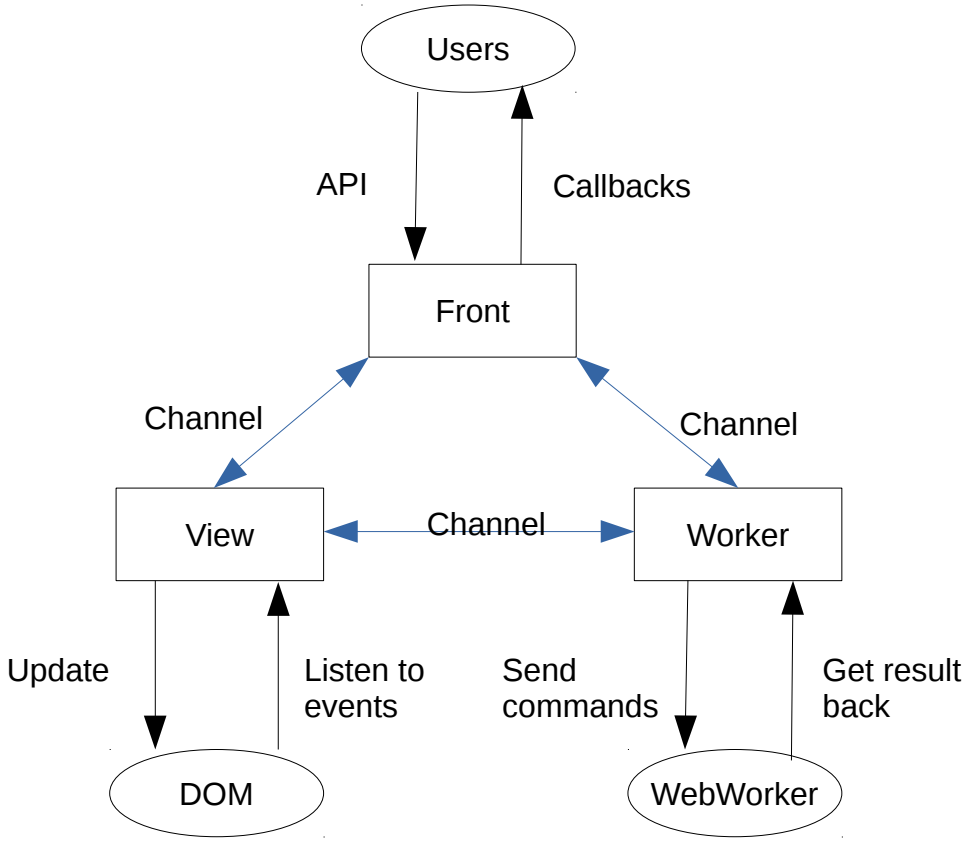
4 Simple ≠ easy. Even if Clojure syntax is simple and explicit, Clojure rely on advanced concepts and writing elegant and efficient code with it need some training.

displaying the graph,

3. the worker part, running the Barnes-Hut simulation.

Those 3 parts communicates with messages published on channels. This approach avoid callbacks' nesting and unexpected exceptions propagation. If the user throw an exception inside a callback bound to an event, the exception will never reach the view or the worker. All different parts communicate in a single and uniform manner. This make the code safe, clean and easy to read. The channel based communication make the code “looks like” synchronous code and make it easy to understand.

The front part take care of communicating with users. All incoming object (from users) are deeply cloned and transformed to immutable data structures. All outgoing objects (to users) are transformed into native JavaScript objects. The user cannot affect the library's internal state by playing with side effects.



2.3 Immutability and pure functional programming

Using immutable data structures and manipulating cloned user's data ensure that data will be available for garbage collection as soon as possible and will be garbage collected if the user remove or delete the graph visualization instance. Immutable data structures and cloning surely consume more memory than using native JavaScript data structures (and manipulating objects by their references). However it improve the garbage collection effectiveness and make the library more stable and predictable.

Clojure functional programming allows implementation of pure functions taking and producing immutable data structures. These functions are combinable to create complexes data transformation that are easy to reason about. The code is easier to read, to write and to maintain.

> *"It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures."*
>
> —Alan J. Perlis [1]

2.4 Web workers and ownership transfer

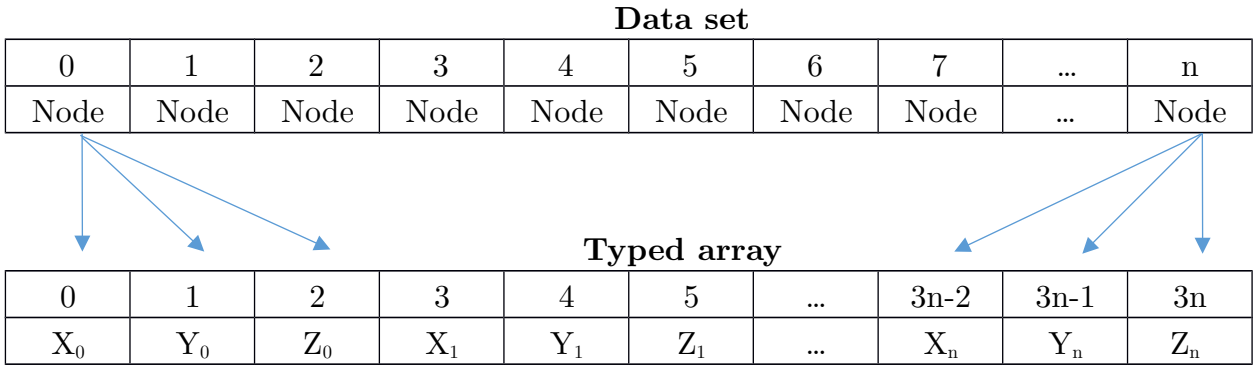
The Barnes-Hut simulation algorithm is running in $O(n \times \log(n))$ time complexity **for each step**. The animation will appear smooth if at least 25 steps are displayed every seconds. This will surely slow down the UI rendering and responsiveness if the dataset is big. A solution to improve performances and responsiveness is to use native threads, implemented as WebWorkers. A WebWorkers cannot access to the DOM and can only communicate with the parent thread by messages passing. A message is passed by serializing the content or by transferring ownership of a typed array between native threads.

To implement this behavior, a new WebWorker is assigned for each new created graph. The WebWorkers will run the Barnes-Hut simulation and emit events to the main thread. These events can be messages about the application state or data arrays. For each simulation steps, the WebWorker will create a new fixed length typed array, fill it with the graph nodes coordinates and transfer its ownership to the main thread. The main thread will then perform a view update by simply reading the received data array. The overhead[5] is:

- Creating the fixed length, typed data array: $O(1)$
- Copying nodes coordinates to the typed array: $O(n)$
- Transferring ownership: $O(1)$
- Updating the view: $O(n)$

The total complexity of a step in the worker is $O(n \times \log(n) + n)$ and $O(n)$ in the main thread.

5 Compared to a classic single-threaded implementation.



2.5 Barnes-Hut simulation

The Barnes-Hut approximation algorithm [2] is implemented in the D3.js [3] library for a 2D simulation. I created a D3.js plug-in supporting the third dimension.

The plug-in add the d3.layout.force3d() function that return a 3D force layout and is available as an independent file to import after D3.js. It does not affect or modify the 2D force layout returned by d3.layout.force().

2.6 3D visualization

It is possible to implement a simulation of 3D visualization with D3.js and without plug-in. However, this techniques rely on DOM+SVG and the third dimension is simulated by changing objects size (i.e. circle radius). This technique works with really small graphs and consume a lot of resources (DOM update on each step and 3D simulation). To improve performances and obtain a better quality, cljs-gravity use WebGL through the popular THREE.js library. WebGL provide access to the GPU capabilities and THREE.js provide hand-on tools to create rich 3D content. On the other side, rendering to a canvas with a 3D context, which end up being just an array of pixels, mean that all events generation (mouse, keyboard, window, etc.) have to be implemented by hand. For this reason a fourth part of the library is dedicated to events generation. It can produce a rich set of events that propagate through channels:

- node-over: The mouse cursor is over a node,
- node-blur: The mouse cursor was over a node and left it,
- node-select: A node has been selected,
- void-click: A click has been performed on nothing,
- node-click: A node has been clicked,
- node-dbl-click: A node has been double clicked,
- drag-start: A node is being dragged,
- drag-end: A node has been dragged,

- ready: The 3D view, the worker and the event listener are all ready, the user can start displaying a graph.

Users can listen to these events and obtain the original event (DOM event) and the corresponding node when available.

2.7 Using the mouse in a 3D space

The process of displaying a 3D scene on a 2D computer screen is called rasterization. The mouse pointer operate on this 2D (rasterized) version of the scene that miss the third dimension. To be able to click and interact with objects in the 3D space, the mouse pointer need to be projected to the third dimension. This operation is called unprojecting. Here is a way to do it:

1. create a vector with x and y coordinates at the mouse pointer's position,
2. unproject this vector using the scene's camera. The vector now point to the correct direction.
3. use a THREE.js ray caster to extend this directional vector until it goes out the camera's field of view,
4. intersect this vector with the 3D objects,
5. get the list of objects that intersect the vector and keep the first one,
6. you got the object currently under the mouse pointer.

3. Results

3.1 Final assets

The production file set is made of 3 JavaScript files:

- D3.js plug-in: **d3-3d.js**
- main file: **gravity.js**
- the web-worker loader: **gravity-worker.js**

The gravity.js file is optimized and minified using the Google closure compiler.

You need to import:

- official THREE.js library,
- official D3.js library,
- the d3-3d.js plug-in,
- gravity.js

A demonstration is available at http://ggeoffrey.github.io.

3.2 Configuration

The only exposed function "gravity.graph.create()" take a map of parameters, here are the default values:

```
{
  "color": d3.scale.category10(),
  "worker-path": "./gravity-worker.js",
  "stats": false,
  "force": {
    "size": [1, 1, 1],
    "linkStrength": 1,
    "linkDistance": 20,
    "friction": 0.9,
    "charge": -30,
    "gravity": 0.1,
    "theta": 0.8,
    "alpha": 0.1
  },
  "webgl": {
    "antialias" : true,
    "background" : false,
    "lights" : true,
    "shadows" : true,
  }
}
```

See D3 force-layout [3] details for the corresponding parameters.

3.3 Size and maintainability

The overall production is made of 3 namespaces and 10 files containing a total of 1580 lines of code ($\bar{x}=158, \sigma=115.84$).

This make the code very small and easy to explore. After spending more then a month working on other projects, it took me less than five minutes to refresh the project in my mind, to understand again the role of each parts, and to be ready to work on it again.

4. Discussion

Of course, like everything, cljs-gravity is far from perfect (starting with the name). The library need a lot of refactoring to reduce the huge lines of code standard deviation between files. It also need to be statically typed, seriously documented and properly tested. However, I am more than confident on one thing: A small and simple code base is more reliable and maintainable than a huge one. This project has been implemented before in TypeScript and even if the static typing made the code more structured in some way, it generated a lot more files, lines of code and was way more difficult to evolve and maintain. The asynchronous aspect and the different independent parts' separation (view, worker, event, front) was less reliable and more tangled or blurry.

I think that even if the code is still a poof-of-concept, this implementation appear to be a success and that Clojure brings good ideas and concepts that slightly force developers to think about concurrency, isolation, purity and simplicity into their design, and that it is the good way to think and build modern and reliable web applications.

Bibliography

1: Alan J. Perlis, Epigrams in Programming, 1982, Yale University
2: VENTIMIGLIA T., WAYNE K., The Barnes-Hut Algorithm, 2003, Princeton University, http://arborjs.org/docs/barnes-hut, Visited: December 2015
3: BOSTOCK M., MEGILL C. and contributors, D3 Force Layout, 2015, , https://github.com/mbostock/d3/wiki/Force-Layout , Visited: December 2015