Proposal—Dygraphs Data Handler API

David Eberlein

April 24, 2013

Problem

Overview

At Sauter we use custom data for our charts, since we have additional information which will then be used by our custom plotters to correctly plot the data. Up until now we added a hack to make this possible.
However, while looking through the dygraphs code and trying to find a good way to integrate this feature I came along the complicated checks for the different formats supported by dygraphs, scattered all over the place. (especially errorBars / customBars).

Dygraphs already provides the API to add custom data plotters, which is very powerful. Additionally, dygraphs has hard-coded options that allow special forms of drawing the graph (e.g. errorBars, CustomBars etc.) which are dependent on the plotters and the data format. It would be very useful to be able to also add proprietary data with more information than x and y values and be able to render them with a custom plotter. The error bars and custom bars options already do this, but they do it in a very hard-coded and non-generic way. The code would become less cluttered, faster, more generic and easier to maintain if there was a clear definition of data formats and a generic handling of the data.

Data Formats

Loader

Dygraphs has two levels of data handling. The first one is the actual data that is inserted into Dygraphs (e.g. CSV / JSON / ...). This data gets parsed initially into a JS Array. I don't want to handle this initial data parsing in this proposal, although I want to reference to Robert Konigsberg's proposal for generic Loaders[1].

Initial Format

Here I want to focus on the second level of data handling. After the data is initially parsed, an n-dimensional array is present that has the following format:

x (e.g. time)	**Series1**	**Series2**	**...**	**SeriesN**
x1	y1(x1)	y2(x1)	...	yN(x1)
x2	y1(x2)	y2(x2)	...	yN(x2)
...	...	...	...	...

which in json would look like:

```
data = [
  [ x1, valueSeries1, valueSeries2, ..., valueSeriesN ],
  [ x2, valueSeries1, valueSeries2, ..., valueSeriesN ],
  ...
]
```

1https://docs.google.com/document/d/1TFea6KjCSRCOjVjivDiQ07WJs2MTwxQZZMHj8ZLxldc/edit#

The values of the series again may be one of the following:

Option	Y-Value Data Format	Y-Value Content	Description
default	num	yVal	single js number
errorBar	[num,num]	[yVal,yError]	2-element array with y value and the error part
customBar	[num,num,num]	[yMin, yVal, yMax]	3D-element array containing a min / mid (e.g. avg) / max value

Per Series Format

This initial array is then extracted into one 2D array per series with the following format:

```
series[n] = [
  [ x, y],
  [ x, y],
  ...
]
```

Again, the y-value may have one of the formats described above:

Option	Series Row
default	[x, yVal]
errorBar	[x, [yVal,yError]]
customBar	[x, [yMin, yVal, yMax]]

Generalized Format

The rolling average method changes the format of the errorBar and customBar data once more. (this is not very clean since one wouldn't suspect this when reading the method name.)
The result of this method is a n-dimensional array per series with the following format:

Option	Generalized Series Row
default	[x, yVal]
errorBar / customBar	[x, yVal, yTopVariance, yBottomVariance]

So from three possible data formats we are now down to two. Additionally now, independent of the prior format, the primary x and y values of each point of a series may now be found in: `series[n][point][0] = x`, and `series[n][point][1] = y`. This will be useful for the proposal.

Point Creation

Before rendering the data, a point array is created for each series. This point array is then passed to the plotters for rendering. Each point is an own object containing different values relevant for plotting.

Conclusion

Dygraphs supports various formats that are adapted and unified several times from their initial input into dygraphs up until the point where they are actually plotted.

Proposal

General

The idea is to define a common, generic data format that works for all data that can be displayed in dygraphs. Additionally a DataHandler interface is added that is implemented for different data types supported by Dygraphs and returns Dygraphs-compliant formats.
By default, the correct DataHandler is chosen based on the options set. Optionally, the user may use his own DataHandler (similar to the plugin system).

Benefits

- The code would become much clearer because a lot of if/else checks would be done in the beginning (preDraw()) and wouldn't be scattered all over the code.
- Handling of one sort of data would be centralized in one place.
- Adding new special graph types could be easily done by adding new DataHandlers and Plotters.
- Performance optimization since each DataHandler can assume that the date is in the correct format which would obsolete the if / else checks
- Users may add their own DataHandlers if the provided ones aren't sufficient.
- Given that plotters can be defined per series and that all DataHandler methods are on a per-series basis, one could even make DataHandler a per-series option, allowing users to handle and plot different sorts of data in one graph.

Costs

- Minor duplications of code since some of the code snippets are used in more than one handler.
- Nothing else I am aware of in the moment. Please add your comments if you can think of something.

Unified Data Format

A great performance and readability benefit lies in a unified data format. (Discussed below.) Dygraphs already shows that a Generalized data format is possible (see Generalized Format above). The proposed data format is very similar:
series[n][point] = [x,y,(extras)]
This format contains the common basis that is needed to draw a series extended by optional extras for more complex graphing types. It contains a primitive x value as first array entry, a primitive y value as second array entry, and an optional extras object for additional data needed.

x must always be a number.
y must always be a number, NaN of type number, or null.
extras is optional and must be interpreted by the DataHandler. It may be of any type.

In practice this might look something like this:

Option	Unified Series Row
default	[x, yVal]
errorBar / customBar	[x, yVal, [yTopVariance, yBottomVariance]]
custom	[x, yVal, {valueReliability, min, max}]

The unified data is generated by the DataHandler described in the next chapter.

Data Handler

The data handler is responsible for all data-specific operations. All of the series data it receives and returns is always in the unified data format. Initially the unified data is created by the extractSeries method:

DataHandler.prototype.extractSeries(g, rawData, seriesIndex)

The extract series method is responsible for extracting a single series data from the general data array. It must return the series in the unified data format. It may or may not add extras for later usage.

DataHandler.prototype.rollingAverage(g, unifiedData, rollPeriod)

The rolling average method is called if the rollPeriod is larger than 1. It is supplied with the series data generated by extractSeries and the rollPeriod.

It must return an array that is again compliant with the unified data format. Extras may be used if needed.

DataHandler.prototype.getExtremeYValues(g, unifiedData, dateWindow)

This method computes the extremes of the supplied rolledData. It may be pruned compared to the data returned by the DataHandler.rollingAverage method, but generally contains the data returned from it. The given dateWindow must be considered for the computation of the extreme values. Extras may be used if needed.

~~DataHandler.prototype.onPointCreated(g, seriesPoint, unifiedDataSample)~~

~~Based on the provided x and y values, seriesPoints for each sample of a series are created. This additional callback is called for every seriesPoint created. The original unifiedDataSample is also given so that additional extras may be extracted and added to the seriesPoint. (e.g. the DataHandlers for bars add y_top and y_bottom here which is needed to draw the error bars.)~~

DataHandler.prototype.onLineEvaluated(g, seriesPoints, dataset, setName)

Because of performance reasons, the onPointCreated callback was replaced by this method[2]. The only difference is that this method is only called once per series, and not for every point of the series. This saves us several method calls as well as several option reads that are done in the onPointCreated.

2See chapter Dygraphs-Perf Benchmarks

Implementation

General

I have done a full implementation of my proposal on one of our branches[3] that you could have a look at.

This is all it basically needs for us to use custom formats; however, one would have to look at the rest of the code to see which other bits should also be extracted. e.g. providing a separate method for rollingAverage is questionable. The computations done in this method could either be done in the extract series method or in a more generic applyFilters method.

See the new folder datahandler and the adaptations in dygraphs.js preDraw() (this.dataHandler_ was added there).

Don't worry about the enormous amount of code added to the extremeValues method. This fixes "Issue 458: Autoscal", and doesn't have anything to do with the datahandler concept.

Data Handlers:

Default Handlers

The “default” handler is a very simple implementation used for normal x/y line line plots. It is the default handler if no special option is set.

The “default-fractions” handler extends it and adds support for simple line charts based on fractions data.

Bar Handlers

The “bar” handler is the base handler for all other bar handlers. It implements the getYExtremeValues and onPointCreated methods since the unified data format allows us to reuse these methods for all other bar handlers. Format specific extractions are done in the inherited bar handlers.

Compatibility

The new implementation is option-wise 100% compatible to the current Dygraphs version. This means that all options, including errorBars, customBars, fractions, and rollPeriod still work exactly the same.

This also means that all current data formats including CSV, Gviz, and different types of JS Arrays still work with 100% the same format.

All changes made for this proposal only concern the internal data handling. I have added several test to the roll-period.js auto test that verify this.

Automated Tests

All automated tests pass. Since the options have stayed compatible, only a handful of tests which called private Dygraphs methods had to be adapted.

I have also added several rolling-period tests since the test coverage of this option was not very good.

Performance

I have added a benchmark test to the automated tests (deactivated because it takes quite a while to run.) I let the benchmarks run both on my proposed version and on the current Dygraphs master branch.

3https://github.com/sauter-hq/dygraphs/tree/sauter-custom-datahandler

Test setup

Browser: Chrome Version 26.0.1410.64
OS: Windows 7 Enterprise 64 Bit SP1
CPU: Intel Core i7-2600 @ 3.4GHz
RAM: 12GB

TestCase1: Init Dygraphs

num of points	options	master [ms]	proposal [ms]
1,000,000	default	1200	1100
1,000,000	fractions	2500	1400
10,000	errorBars	1400	1400
10,000	errorBars, fractions	1400	1400
10,000	customBars, rollPeriod (500)	1400	1400

TestCase2: Set Data: updateOptions(file)

num of points	options	master [ms]	proposal [ms]
1,000,000	default	1200	1200
1,000,000	fractions	2900	1500
10,000	errorBars	1400	1400
10,000	errorBars, fractions	1400	1400
10,000	customBars, rollPeriod (500)	1400	1400

TestCase3: Zoom In / Out: updateOptions(dateWindow)

num of points	options	master [ms]	proposal [ms]
1,000,000	default	1300	1300
1,000,000	fractions	2900	2400
10,000	errorBars	800	800
10,000	errorBars, Fractoins	800	800
10,000	customBars, rollPeriod (500)	800	800

Conclusion

Performance-wise most of the tests come to the same results. Default data with fractions stands out a bit but that is probably tweakable. Generally one can say that the performance definitely doesn't get worse with this solution, but it is rather faster.
So performance should not stand in the way of this proposal.

Dygraph-Perf Benchmarks

General

Additionally, benchmarks based on the Dygraph-Perf[4] project have been done, to test the two implementations in their minified form.

Test setup

Engine: Phantom JS 1.9
OS: Ubuntu Linux 3.2.0-40-generic-pae i686 i386
CPU: Intel 2 Duo E7500 @ 2.93GHz
RAM: 2GB

Test Case 1: No roll period

p.p. series	series	data format	roll period	master [ms]	proposal [ms]	diff [%]
1000	100	line	1	165	174	-5.2
1000	100	fractions	1	235	220	+6.8
1000	10	errorBars	1	4752	4758	-0.1
1000	10	customBars	1	2375	2371	+0.2

Test Case 2: Roll period of 10

p.p. series	series	data format	roll period	master [ms]	proposal [ms]	diff [%]
1000	100	line	10	252	261	-3.4
1000	100	fractions	10	236	250	-5.6
1000	10	errorBars	10	741	750	-1.2
1000	10	customBars	10	2379	2381	-0.1

Optimizations

This sums up to an overall performance loss of -1.2% which is rather small, taking into account that no optimizations have been done to make the performance better. The next step is to look at what slows down the performance and add optimizations for the data handler implementation. The following optimizations could be done:

Removal of data copying

Removed copying each series in the gatherDatasets_ (dygraphs.js) method. A comment there stated that the series had to be copied due to issues when zooming with the errorBar option. I removed the copy on both my proposal and the current master branch. The zoom bug was present on the master trunk but it was not present anymore in my proposed version. The unified data format seems to fix this bug.

4https://github.com/danvk/dygraphs-perf

onLineEvaluated instead of onPointCreated

The onPointCreated callback, which was called for every single point, was responsible for most of the performance reduction in my proposal. Now instead of calling this method for every point created, a different callback method was added, called onLineEvaluated. This method is only called once per series and is given all seriesPoints and the dataset for the series.

Make onLineEvaluated optional

Since the default data handlers don't use the onLineEvaluated callback, they now set it to "undefined". dygraphs-layout.js now checks this and only calls the method if it is defined, which also boosts the performance.

Optimized Test Case 1: No roll period

p.p. series	series	data format	roll period	master [ms]	proposal [ms]	diff [%]
1000	100	line	1	165	156	+5.5
1000	100	fractions	1	235	202	+14.0
1000	10	errorBars	1	4752	4745	+0.1
1000	10	customBars	1	2375	2360	+0.6

Optimized TestCase 2: Roll period of 10

p.p. series	series	data format	roll period	master [ms]	proposal [ms]	diff [%]
1000	100	line	10	252	243	+3.6
1000	100	fractions	10	236	232	+1.7
1000	10	errorBars	10	741	738	+0.4
1000	10	customBars	10	2379	2368	+0.5

Conclusion

With the optimizations done, this sums up to a overall performance gain of +3.3% which shows that the new solution even has benefits performance-wise. Also one can see that this performance benefit is consistent for every data format tested. None of the tests perform worse than the current master which in my opinion is a great achievement. Additionally, the most commonly used data format (simple line) even outdoes the overall 3.3% with a performance gain of +5.5% (from -5.6% in the unoptimized version).

The relatively small performance gain using errorBars and customBars is mainly due to the reduced numbers of series. The gain from not having to copy all the data per series therefore doesn't come into account that much. Also in contrast to the line and fractions data handlers, the bars data handlers both implement the onLineEvaluated and therefore don't profit from the "undefined" optimization.

Sequence Diagram