5.20.2015

GRAPHEAD

(⌒/? /\|^ |-| [- /\|)

Graphead is a graph editor. It uses the HTML5 canvas element for rendering. It works in all modern browsers, and supports touch-enabled devices.

Quick Start Guide

1. Include the Graphead JavaScript files in your HTML

```
<script type="text/javascript" src="https://code.createjs.com/easeljs-
0.8.0.min.js"></script>
<script type="text/javascript">var cjs = createjs;</script>
<script type="text/javascript" src="Graphead.js"></script>
<script type="text/javascript" src="Grid.js"></script>
<script type="text/javascript" src="regression.min.js"></script>
```

Notice the CreateJS library. EaselJS is REQUIRED. Easel is a part of the CreateJS suite available from createjs.com. The createjs namespace has been abbreviated to "cjs".

2. Include a <canvas> element in your HTML

```
<canvas id="MY_Canvas" width="400" height="400"></canvas>
```

3. Create a new instance of Graphead

```
var MyGraph = new Graphead();
MyGraph.init("My_Canvas", "My_Graph_Name", {});
```

This is the simplest configuration possible. The 'init' method takes 3 arguments: the ID of your HTML canvas element, a name to assign to the instance, and a configuration object. In this case, the configuration object is empty: {}

Graphead has a DEFAULT configuration. Without customization, it will render a 10 x 10 graph with 30px wide squares.

Your instance code should be called only after the page (DOM) has loaded. Use window.onload or a similar function.

4. Customize your grid

Without any customization, Graphead renders a simple 10 x 10 grid. To override the default, provide a JSON object with different property values. There are LOTS of properties you can customize: colors, labels, fonts, sizes, etc. The complete list is documented elsewhere. Snap to grid is enabled by default. To enable touch support, set the touch property to true.

This will render a 5 x 5 graph with axis labels:

```
var MyGraph = new Graphead();
MyGraph.init("My_Canvas", "My_Graph_Name", {unitsWide:5, unitsHigh:5,
axisTitleX: "time", axisTitleY: "money"});
```

This will render a 5 x 5 graph with Y-axis incremented by 100 (0, 100, 200, 300, 400, 500)

```
var MyGraph = new Graphead();
MyGraph.init("My_Canvas", "My_Graph_Name", {unitsWide:5, unitsHigh:5,
scaleFactorY:100});
```

This will render a 20 x 15 graph with white lines on a black background, snap to grid is disabled:

```
var MyGraph = new Graphead();
MyGraph.init("My_Canvas", "My_Graph_Name", {snap:false, unitsWide:20,
unitsHigh:15, bgColor:"#000000", lineColor:"#ffffff"});
```

This will render a 20 x 15 graph, every 5th line is labeled on the axes: 0 . . . . 5 . . . . 10 15

```
var MyGraph = new Graphead();
MyGraph.init("My_Canvas", "My_Graph_Name", {unitsWide:20, unitsHigh:15,
labelIntervalX:5, labelIntervalY:5});
```

5. Customize your data (LAYERS)

The visual style of plotted data is organized by LAYERS. A layer is a named collection of styles: colors, shapes, sizes, etc.

You must create a least 1 layer to plot any data. All data is associated with a layer name. In that sense, a layer also represents a data set.

A graph may have multiple layers. You could, for example, customize one layer to draw points in red and another in blue. Or, you could customize a layer to draw squares rather than circles.

For interactive graphs, a layer can be "locked." A locked layer cannot be changed or deleted. This is helpful if you would like to pre-plot data for the user; it will persist regardless of any interaction.

```
MyGraph.addLayer([], "My_layer", false, {});
```

This is the simplest configuration possible. The 'addLayer' method takes 4 arguments: an array of points or lines (if you're pre-plotting), the NAME of your layer, a boolean flag to lock (persist) the layer, and a configuration object. In this case, the configuration object is empty: {} It is not necessary to provide plotting data when creating a layer because data can always be added later (**see addData()**).

Graphead has a DEFAULT configuration for layers. Without customization, Graphead renders points as small red circles and lines with thin red strokes. To override the default, provide a JSON object with different property values. There are many properties you can customize: color, shape, stroke size, etc. The complete list is documented elsewhere.

This draws points and lines on "My_layer" in blue:

```
var MyGraph = new Graphead();
MyGraph.init("My_Canvas", "My_Graph_Name", {});
MyGraph.addLayer([], "My_layer", false, {color: "#0000FF", fill:"#0000FF"});
```

This creates 2 layers, one red and one blue:

```
var MyGraph = new Graphead();
MyGraph.init("My_Canvas", "My_Graph_Name", {});
MyGraph.addLayer([], "My_RED_layer", false, {color: "#FF0000",
fill:"#FF0000"});
MyGraph.addLayer([], "My_BLUE_layer", false, {color: "#0000FF",
fill:"#0000FF"});
```

This creates 2 layers, red & blue, the red layer is LOCKED and has 2 data points:

```
var MyGraph = new Graphead();
MyGraph.init("My_Canvas", "My_Graph_Name", {});
MyGraph.addLayer([{x:4,y:4}, {x:7,y:7}], "My_RED_layer", true, {color:
"#FF0000", fill:  "#FF0000"});
MyGraph.addLayer([], "My_BLUE_layer", false, {color: "#0000FF", fill:
"#0000FF"});
```

6. Add interactivity

Graphead is interactive by default. A user can drag any point on a layer that is not locked. To create a completely static display, set the lock flag on each layer to: "true". (**see LAYERS**)

You can add more interactivity by enabling 'point and click' behaviors called "**click actions**". Click actions allow the user to add lines, points, and curves to the graph. These actions work similarly on touch devices. To enable touch support, set the touch property to true when initializing your graph.

Here are some examples, the complete list is documented elsewhere.

dot	each click adds a point
line	each click adds a point, every 2nd point defines a new line
curve	each click adds a point, all points are connected by a smooth curve
piecewise	each click adds a point, all points are connected as a piecewise linear function

A single graph can have multiple click actions, but only 1 can be enabled at a given time.
The "enableClickAction" method takes 2 arguments: the name of the layer to use, and the name of the click action. (Some actions accept additional arguments.)

This will enable "dots", where each click adds a new point:

```
MyGraph.enableClickAction("My_layer", "dot");
```

This will enable a piecewise linear function:

```
MyGraph.enableClickAction("My_layer", "piecewise");
```

NOTE: Use "setPointLimit", to limit the total number of points that can be added:

```
MyGraph.setPointLimit(5);
```

NOTE: "startOver" disables any click actions. Click actions must be re-enabled after a call to startOver.

7. Undo & Start Over

It's a good idea to give users the ability to undo their actions or start over completely. Graphead provides the functionality to do both. However, Graphead does not include buttons or any type of UI. Your web application should provide its own buttons, which, in turn call Graphead's public methods: undo() and startOver().

Example HTML:

```
<button onclick="window.MyGraph.undo()">UNDO</button>
<button onclick="window.MyGraph.startOver()">START OVER</button>
```

8. Have fun !

Graphead can do a lot more than what appears in this quick start guide. Read the full documentation for additional features.

GRID PROPERTIES

Graphead has a DEFAULT grid configuration. Without customization, it will render a 10 x 10 graph with 30px squares. Snapping enabled. To override the default, initialize your graph using a JSON object with different property values.

Any of the listed properties can be overwritten.
Most of the properties come in X/Y pairs, allowing one axis to be configured differently from the other.

Graph Size

unitsWide	Number	the width of your graph, the total number of units horizontal	10
unitsHigh	Number	the height of your graph, the total number of units vertical	10
unitSize	Number	the display size of each square unit (in pixels)	30

- A grid does not necessarily need to use perfect squares for units, setting different values for 'unitSizeX' & 'unitSizeY' will result in rectangular units.

unitSizeX	Number	the horizontal width of each unit (in pixels),	30
unitSizeY	Number	the vertical height of each unit (in pixels),	30

Graph Labels

axisTitleX	String	a label for the X axis	" "
axisTitleY	String	a label for the Y axis	" "

Grid Lines & Background

useBG	Boolean	use a background color	true
bgColor	String	the CSS color used to paint the background	"#FFFFFF"
labelBgColor	String	the CSS color painted behind the axis labels	"#FFFFFF"
useOutline	Boolean	draw a line along the perimeter of the grid	true
outlineColor	String	the CSS color used to draw the outline	"#000000"
lineColor	String	the CSS color used to draw grid lines	"#CCCCCC"
originLineColor	String	the CSS color used to draw the lines on the origin	"#000000"
strokeStyle	Number	the stroke width of the drawn lines	0.5

- You can show horizontal lines only, or vertical lines only, by not drawing one of the axes.
- You can have a blank graph by not drawing any axes.
- Graphead can still plot data without grid lines.

useAxisX	Boolean	draw the vertical grid lines	true
useAxisY	Boolean	draw the horizontal grid lines	true

Fonts

font	String	the CSS font family used for axis labels	"Arial"
fontSize	Number	the CSS font size for axis labels	12
fontColor	String	the CSS font color for axis labels	"#000000"

Scale Numbering and Labels

- The 'scaleFactor' determines how the scale increases for each grid square. For example, setting a scaleFactor of 10, will result in: 0, 10, 20, 30, 40, . . .
- It is not necessary for your scale to display every value. You can label every other line, or every 5th line. For example, setting a labelInterval of 5, will result in: 0 . . . . 5 . . . . 10 15

- You don't need to show the scale at all.
- You may want to use custom labels on your scale. For example, the X-axis might be days of the week. 'customLabel' accepts an array of strings for this. `customLabelsX:["Monday", "Tuesday, "Wednesday", "Thursday", "Friday"]` (Custom labels do not alter the numeric value of the scale.)
- By default, NO fractional part of a decimal number is displayed. Setting the 'decimalX/Y' property will change this. Numbers will be rounded to the fixed number of decimal places.
- 'unitLabel' can add an optional suffix to indicate a unit of measurement (minutes, feet, meters, etc)

scaleStartX	Number	what number the X-axis scale should begin	0
scaleStartY	Number	the Y-axis scale should begin	0
scaleFactorX	Number	a scale factor to multiply X-axis values by	1
scaleFactorY	Number	a scale factor to multiply Y-axis values by	1
labelIntervalX	Number	which grid lines get a label	1
labelIntervalY	Number	which grid lines get a label	1
useLabelsX	Boolean	show the scale along the X-axis	true
useLabelsY	Boolean	show the scale along the Y-axis	true
customLabelsX	Array	strings to use as labels along the X-axis scale	[ ]
customLabelsY	Array	strings to use as labels along the Y-axis scale	[ ]
decimalX	Number	the number of decimal places to display	0
decimalY	Number	the number of decimal places to display	0
unitLabelX	String	a suffix to append on all values along the scale	" "
unitLabelY	String	a suffix to append on all values along the scale	" "
labelSpacing	Number	how far the labels are drawn from the grid (in pixels),	5

The Origin & Quadrants

By default the origin is the lower left hand corner. By moving the origin, you can display a grid divided into quadrants. When using a quadrant display, you can optionally choose to label the origin, or draw arrows at the end of each axis.

originoffsetX	Number	the number of units from the left to move the origin	0
originoffsetY	Number	the number of units from the bottom to move the origin	0
originoffsetZero	Boolean	label the origin (zero) when using a quadrant display	false
arrowheadPosX	Boolean	draw arrow shape at the positive end of the X-axis	false
arrowheadPosY	Boolean	draw arrow shape at the positive end of the Y-axis	false
arrowheadNegX	Boolean	draw arrow shape at the negative end of the X-axis	false
arrowheadNegY	Boolean	draw arrow shape at the negative end of the Y-axis	false

Global Attributes

snap	Boolean	enable **snap-to-grid** !	true
touch	Boolean	enable **touch** ! Supports W3C Touch API (iOS & Android) and the Pointer API (IE 11)	false

---

LAYER PROPERTIES

The visual style of plotted data is organized by LAYERS. A layer is a named collection of styles: colors, shapes, sizes, etc. **All data is associated with a layer name. In that sense, a layer also represents a data set.**

Graphead has a DEFAULT configuration for layers. Without customization, Graphead renders points as small red circles and lines with thin red strokes. To override the default, initialize your graph using a JSON object with different property values.

ANY of the listed properties can be overwritten.

size	Number	the size of the plot shape (in pixels)	8
color	String	the CSS color for lines	"#EE0000"
fill	String	the CSS color to fill plots	"#EE0000"
strokeSize	Number	the stroke width	0.5
shape	String	the plot shape ('circle' & 'square' are supported)	"circle"

- font styles for labeling points and lines (not for axes)

font	String	the CSS font family for point labels	"Arial"
fontSize	Number	the CSS font size for point labels	12
fontColor	String	the CSS font color for point labels	"#000000"

CLICK ACTIONS

You can add interactivity to your graph by enabling 'point and click' behaviors called "click actions". Click actions allow the user to add lines, points, and curves by clicking on the graph. These actions work similarly on touch devices. **A single graph can have multiple click actions, but only 1 can be enabled at a given time.** Some actions accept additional arguments.

dot

Each click adds a new point.

```
.enableClickAction('My_Layer','dot')
```

line

Each click adds a new point, every 2nd point defines a new line.
There 2 line types available:

SEGMENT	a line segment; the line is bounded by two end points
EXTENDED	the line continues to the edges of the graph

The user can keep adding new lines if the "multiple" flag is set, otherwise only 1 line exists at time. Any line can be labeled with a string.

```
.enableClickAction('My_Layer', 'line', 'SEGMENT', true)
.enableClickAction('My_Layer', 'line', 'SEGMENT', false)

.enableClickAction('My_Layer', 'line', 'EXTENDED', true)
.enableClickAction('My_Layer', 'line', 'EXTENDED', false)

.enableClickAction('My_Layer', 'line', 'EXTENDED', false, 'MY_Label')
```

connected

Each click adds a point, all points are connected by line segments in the order points were added. Connected segments will allow a user to draw closed polygons. The line can be labeled with a string.

```
.enableClickAction('My_Layer', 'connected')
.enableClickAction('My_Layer', 'connected', 'My_Label')
```

piecewise

Each click adds a point, all points are connected as a piecewise linear function. Vertical lines are not allowed.

```
.enableClickAction('My_Layer', 'piecewise')
```

curve

Each click adds a point, all points are connected by a smooth curve. Vertical lines are not allowed.

```
.enableClickAction('My_Layer', 'curve')
```

PUSH-BUTTON ACTIONS

In some situations, you may want a user to push a button first. Undo and Start Over are obvious examples. Regressions & Curves can be added with a button press. A new line can be added with a button press. Layers can be wiped clean with a button press.

See USEFUL API METHODS for the full list of behaviors that could be wired to a button.

NOTE: Graphead does not include buttons or any other type of UI. Your web application should provide its own buttons, which, in turn call Graphead's public methods.

REGRESSIONS

Graphead supports 3 regression types: **Quadratic, Exponential, Linear**.
Regressions are performed by Tom Alexander's Regression.JS
http://tom-alexander.github.com/regression-js/
(Tom Alexander's code can also return an equation. A hook to retrieve this will very likely be added in the future.)

- A regression uses all the points on a given layer for its data set.
- The method can draw the regression on a different layer than its data set. This is useful for removing a plotted regression.
- Regressions will re-calculate themselves if any points in the data set change.

The .addRegression method takes 3 arguments: the name of the layer to use as a data set, the name of the layer to draw the regression on, and the regression type:
`.addRegression(dataLayerName, regressionLayerName, regressionType)`

```
.addRegression('layerOne', 'layerTwo', 'linear')
.addRegression('layerOne', 'layerTwo', 'quadratic')
.addRegression('layerOne', 'layerOne', 'exponential')
```

CURVES

A smooth curve can be plotted using a layer as a data set.
(Curves can also be added interactively using a click action. **see CLICK ACTIONS**)
Curves are computed using equations by Jim Fife. Curves will re-calculate themselves if any points in the data set change.

addCurve takes a single argument: the name of the layer to plot.

```
.addCurve("My_Layer")
```

(The ability to draw curves on a different layer than the data set will very likely be added in the future.)

USEFUL API METHODS

NOTE: Graphead does not include buttons or any other type of UI. Your web application should provide its own buttons, which, in turn call Graphead's public methods.

.undo()

Reverse the last action taken by the user

.startOver()

Clear all data (EXCEPT layers that have been locked) and disable any click actions.

.addLine()

.addSegment()

These will add a single line to the graph. They are very similar to the "line" click action, but this method doles out lines one at a time. On non-touch devices, they have hover animations not seen in the click action form.

```
.addLine('My_Layer')
.addLine('My_Layer', 'My_Label')
.addSegment('My_Layer')
.addSegment('My_Layer', 'My_Label')
```

.addData()

Programmatically add data at any time using addData. This method takes a layer name and an array of points and/or lines to plot.

Points should be encoded as an object like this: `{x:0, y:0}`

Lines should be encoded as an **Array** with point1, point2, and the line type (SEGMENT or EXTENDED).

```
.addData("layerName", [{x:0, y:0}])
.addData("layerName", [{x:2, y:2}, {x:5, y:5}, {x:7, y:7}])
.addData("layerName", [[{x:1,y:1},{x:3,y:3},"EXTENDED"]])
.addData("layerName", [[{x:1,y:1},{x:3,y:3},"EXTENDED"],
[{x:6,y:6},{x:14,y:14},"EXTENDED"]])
.addData("layerName", [{x:2, y:2}, [{x:1,y:1},{x:3,y:3},"SEGMENT"]])
```

.setPointLimit()

Limit the total number of points that can be added to the graph. **See CLICK ACTIONS**.

```
.setPointLimit(7)
```

.disableClickAction()

Programmatically disables any click actions.

.clearLayer()

Remove all plotted data from a layer and redraw. This has no effect on layer that have been locked.

```
.clearLayer("My_Layer")
```

.deleteLayer()

Delete all plotted data, the layer styles, and the layer object itself.

```
.deleteLayer("My_Layer")
```

.customText()

This allows you to add text anywhere on the canvas. Although it takes a layer name as a parameter, the layer is used for its style properties only. Custom text has no effect on data. The third argument should be a simple point object. The coordinates of that object are global coordinates on the canvas NOT coordinates on your graph.

```
.customText("hello world", "My_Layer", {x:50,y:60});
```

.getData()

Returns ALL information about the graph: plot data, layers, canvas ID, instance ID, grid configuration, etc. The string returned by getData can be fed to setData, which, can restore a graph to its original state.

.setData()

Restores a previous Graphead instance to its original state. setData expects the string generated by getData().

CLEAN UP

.cleanUp()

This is a very important method! When your application is finished using Graphead, cleanUp will remove all event handlers, wipe all data, and remove object references so memory can be freed by garbage collection. **It is always a good idea to clean up when you're done using Graphead.**

HACKING

Grid.js can be used stand alone from Graphead. Grid.js does one thing, and one thing only: it draws a grid with scale & axis labels. It returns the grid as an Easel 'container' object, which, can be used in any Easel (CreateJS) application. **Grid knows absolutely nothing about points or lines**. It can't plot anything. It has 1 public method: getOriginPt(). This will return the stage coordinate of the origin.

Knowing the location of the origin and the basic size of the grid units, you can mathematically work out where a plot should be drawn, which, is what Graphead.js does.

Graphead's object model is pretty simple and there are more than enough utility functions to help you build something new.

It is important to understand the types of "point" objects used in the code:

- "dataPt" is USER DATA, these are formatted numbers for human interpretation.
- "coorPt" is a global COORDINATE on Easel's stage for drawing purposes.
- getCoorPoint() & getDataPoint() translates a DATA point to a COORdinate and vice versa, this is IMPORTANT !
- "ComplexPoint" combines "dataPt" & "coorPt" into a single object (abbreviated as "plexPoint")
- A "ComplexLine" combines 2 ComplexPoints.
- Easel's point object "cjs.Point" is a simple data structure like this: `{x:x, y:y}`, it is the underlying data structure for the other point objects.
- However: regression.js and plotSmoothCurve use a 2 element array for a data structure: `[x, y]`
- There are some utility functions to switch between these 2 structures.
- A "**DOT**" is a synonym for a ComplexPoint that is NOT associated with a line or other structure.
- I also use "DOT" to avoid saying "point" too often in the code. ☺
- All points and lines are associated with a layer.
- Layers can be stacked on top of one another.

Here are some methods worth knowing about:

getAllPoints(layerName)
LayerName is optional, but if provided, points will be filtered.

getAllDots(layerName)
This method returns all points that are not associated with a line or curve or piecewise function.
LayerName is optional, but if provided, points will be filtered.

getCoorPoint(pt, noSnapFlag)
Given a mathematical data point, i return the internal x,y coordinates for the stage.

getDataPoint(pt, noSnapFlag)
Given internal x,y coordinates on the stage, I return the mathematical data point

drawLayer()
drawAllLayers()

Understanding Lines, Curves, & Regressions
Graphead tries to maintain Points as the primary object type (ComplexPoint). Lines, curves, and regressions are essentially collections of point objects. The drawLayer method will loop through these collections and decide how to draw them. Any new object types should attempt to follow this model. In the future, this could be a place for further abstraction. Possibly each object type supported by the graph could be a specialized instance of an abstract collection object.

2014

Graphead was written by Jason Bonthron in 2014 with contributions from Jim Fife and Keith Kiser.
Regressions use Tom Alexander's Regression.js
http://tom-alexander.github.com/regression-js/
EaselJS was written by Grant Skinner.
http://www.createjs.com