# FATuM -Fast Animated Transitions using Multi-Buffers

Alexandre Perrot, David Auber

# FATuM - Fast Animated Transitions using Multi-Buffers

Alexandre Perrot, David Auber
LaBRI, Université de Bordeaux
aperrot@labri.fr, auber@labri.fr

**Abstract**

*The rise of Big Data and powerful mobile devices calls for libraries able to render a large number of visual elements and make fast animations without loss of frame rate. We introduce the FATuM library as a middleware for visualization. With a single abstraction for visual elements based on the work of Bertin and adaptation of the double buffering technique, we enable animated visualization of large datasets in native applications and in the browser using the same codebase.*

*Our system does not differentiate animated from static rendering, thus reducing code complexity and guaranteeing smooth animation. We show that our system maintains 60fps for up to 200.000 visual elements in a native application and 30fps for 100.000 visual elements in a web browser.*

*Keywords*—**Animation, library, transitions.**

## 1 Introduction

With the advent of HTML5 and the democratization of mobile devices, web browsers are an increasingly popular platform for data visualization. Many libraries have implemented visualization interfaces using the web standard client language, javascript. This has popularized data visualization outside of the scientific community. With general public, some new trends started to emerge, such as need for animation to convey meaning.

Another current trend is the developement of Big Data. It generates large datasets to be visualized and the great dynamicity of data demands fast rendering libraries, able to handle animation at large scales. Furthermore, the new Big Data Infrastructures, consisting of several distributed compute and storage nodes enable bigger datasets to be processed and saved. Their distributed nature increases the need for browser based visualization applications capable of handling the massive data stored in those repositories. Nowadays, most mobile devices are sufficiently powerful to run GPU based applications, and WebGL also enables GPU use in web browsers. This is the perfect combination to enable large dataset browser based visualization.

Motivated by these evolutions in the data visualization landscape, we describe in this paper a way to fill the gap between fast and responsive GPU technologies and user-friendly data visualization libraries. We implemented a library leveraging the speed of OpenGL and WebGL. Its interface is oriented toward data visualization rather than simple drawing. Inspired by the double buffering technique used in computer graphics, we also enable fast animated transitions between visualization states.

The paper is divided into two parts. First, we explain why we chose to expose only a single flexible type of visual element and its visual properties. Second, we develop the technique we call "double buffering for information visualization", which enables fast animation of large datasets.

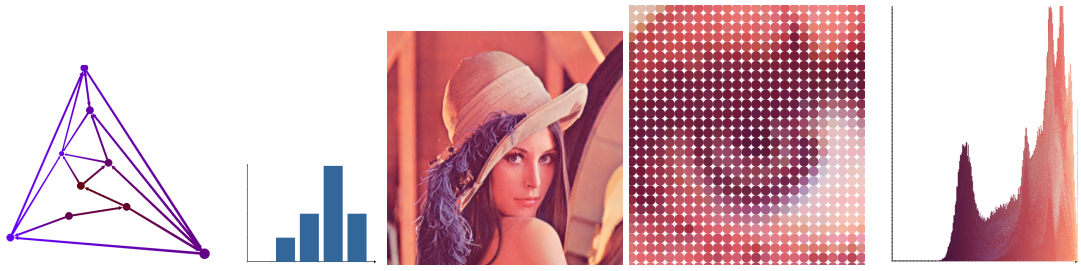
## 2 Previous Work

In this section, we summarize previous works about animation and give an overview of the major existing libraries for information visualization.

### 2.1 Animation in information visualization

Animation in information visualization is a well-studied domain. Many publications have highlighted the benefits and drawbacks of using animation for different kinds of visualizations. Archambault et al. [1] compared the effects of animation and small multiples in the context of dynamic graphs. Work about animation and mental map also include work by Bederson and Boltman [2]. Heer and Robertson [12] conducted an experiment to test the effect of animated transitions in statistical data graphics. They found that carefully designed transitions improve change perception. The general consensus is that using animated transitions can be beneficial to help track changes in the visualization, but must be carefully designed to avoid inducing unnecessary occlusion, distraction or ambiguity, as described by Chevalier et al. [6].

Based on results of these studies, systems have been designed with animation as a core feature. The ScatterDice [9] system uses animation to explore multi-dimensional data. The data is represented as a 2D-scatterplot, which undergoes a 3D rotation animation when changing one of the dimensions. This usage of animation enables to track data points across dimensions. The DiffAni [16] system also uses animation in combination with difference maps to visualize dynamic graphs.



(a) (b) (c) (d) (e)

Figure 1: Examples of visualizations built with FATuM : (a): node-link view of a graph. (b): histogram of the nodes' degrees of the graph in (a). (c): image visualization of the popular "Lenna" image. Each pixel of the image is represented by a mark. (d): detail of the image. We can clearly see the marks representing the pixels, with a circular shape to distinguish them. (e): the frequency histogram of the red component from the "Lenna" image. Recreating work from Chevalier et al.[7], the marks representing the pixels are moved to form the histogram.

### 2.2 Libraries for information visualization

Many libraries are available to build visualizations. Despite their diversity, they can be divided into two types : low-level drawing libraries and high-level visualization libraries.

The low level drawing libraries only expose a way to draw graphics, through specialized specifications. For example the **SVG**[1] language specifies elements to draw vector graphics. SVG renderers are now implemented into web browsers, enabling its use for a wide variety of applications. **Raphael**[2] is a javascript library aimed at providing a cross-browser interface for SVG graphics. **OpenGL**[3] is a graphics library enabling the use of GPU. Drawing anything with OpenGL requires many lines of code and is difficult to get right. However, it ensures best performance. A web implementation exists in the form of **WebGL**[4], allowing for GPU based graphics rendering in web browsers. **Processing**[5] is a full programming language aimed at artists and designers. Its API is very close to OpenGL, and as such, suffers from the same drawbacks regarding visualization design. A javascript implementation called Processing.js is available for web browsers. The **Three.js** library[6] is built on top of WebGL as a 3D graphics library. While this eases the use of WebGL for a 3D context, it does not help for data visualization.

All those libraries suffer from a common drawback : lack of visual abstractions. They are designed for flexible and efficient drawing primitives, but are not suited as information visualization libraries.

On the other hand, there exists a number of high-level visualization libraries. The most widely used is Bostock's et al. **D3** [5]. Implemented for web browsers in javascript, this library uses the webpage's Document Object Model as a data backend. Its very simple and powerful API ensured its adoption outside of the InfoVis community. The user only has to specify changes to the state of data and not the entire visualization to produce a dynamic visualization. However, the implementation of D3 prevents it from handling a large number of visual elements. Since every graphical object of the visualization is an element in the DOM, the size of the visualization is inherently limited. Other libraries have tried to overcome this difficulty by adding WebGL hardware acceleration capabilities to D3, but the major part of the visualization still uses the DOM.

Seeing the respective drawbacks and advantages of both categories of library, we propose the FATuM library to fit between those two categories.

## 3 The system

In this section we present the design of the system, its goals, its interface and details of the implementation.

### 3.1 Library Philosophy

The library presented here is a 2D visualization rendering library, half-way between the low levels graphics libraries and the specialized visualization libraries. It only focuses on the rendering process. We see it as a *middleware* between the visualization designer and the graphics capabilities of the computer, by analogy with web applications, where the middleware enables communication between the front-end user interface and the back-end storage space. It will allow other libraries to provide another interface more suitable for visual design, without worrying about rendering speed.

Our system is totally detached from any data model. Thus, we do not provide any data management layer in our library, unlike D3. Managing the correspondence between

[1] http://www.w3.org/Graphics/SVG/
[2] http://raphaeljs.com/
[3] http://www.khronos.org/opengl/
[4] http://www.khronos.org/webgl/
[5] http://processing.org
[6] http://threejs.org

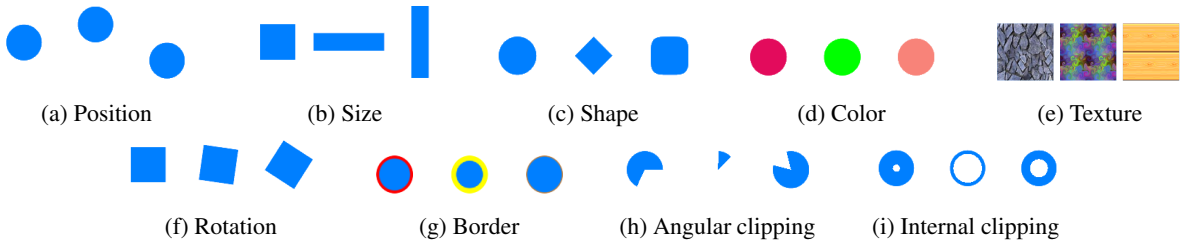



Figure 2: Different values for the properties available for the marks.

data and the visual entities is the responsibility of the user. While this approach can be considered a supplementary burden for the user, we argue that it allows more flexible data mapping, other than a simple one-to-one mapping.

The library handles the visual objects in a 3D world, using an orthographic projection to create a 2D visualization. Thus, it handles model and screen coordinates spaces and enables zoom and pan using matrices operations. Furthermore, a layer system enables to control the types of objects rendered in the final visualization.

### 3.2 Visual primitives

Many visualization libraries propose a large selection of graphical elements to represent data, ranging from circles and squares to more exotic shapes like stars. They also generally provide lines, polylines, arrows, curves, areas or polygons. During an animated transition, a data point often has to change a number of its visual variables, such as shape, size or position. When that happens, the datum represented by this point has not necessarily changed. Only the representation of the datum has changed. We wanted to reflect that in our library by providing a common abstraction on top of which it would be possible to build more classical graphical elements. Since Bertin [3], such elements are called ”marks”, but visualization libraries tend to differentiate them based on shape, possibly for implementation purposes. Following the work done on Protovis [4], which already identifies marks as visual primitives, we propose to go one step further by exposing only a single type of entity for marks, allowing dynamic modification of its visual properties, including its shape.

Additionaly, in many visualizations, marks can be connected to one another. For instance, in a node-link graph visualization, marks representing nodes are connected through edges, in a line chart, a mark is connected to the next one. To model this relationship between marks, we introduce the *connection* visual entity.

### 3.3 Visual properties

Bertin [3] defined properties for the marks, which we retain as follows:

- Position: X, Y and Z position of the center of the mark. Thanks to the orthographic projection, the Z coordinate enables to control which mark is rendered on top of the others.
- Size: Width and height of the mark. No size is provided on the Z axis, since it would not be seen in the visualization.
- Shape: The shape of the mark. All standard shapes are possible, such as circle, square or diamond. More complex shapes are also provided thanks to a distance field. There is also a special shape called ”none”, which makes the mark invisible.
- Color: The color of the mark, including the alpha channel for transparency. This relates to the value and color visual variables from Bertin.
- Rotation: This property enable to rotate the mark around the Z axis. This is the ”orientation” variable from Bertin.
- Texture: We enable the user to specify a texture for the mark.

Additionaly, we introduce new properties for more flexibiliy. First are the properties relative to the mark's border and second are clipping properties, useful for radial visualization, as they enable to easily make radial charts, such as pie charts and donut charts using marks as angular sectors:

- Border size: Size of the border in pixels. The border size is defined in screen space, meaning that its apparent size will remain constant, even upon zooming.
- Border color: The color of the border can be different from the color of the mark.
- Radial clipping: Defines an angular portion of the shape to render. This enables to use angle as an encoding variable, as suggested by Mackinlay in [13].
- Internal clipping: Defines a portion of the inside of the shape that should not be rendered

Example values for the properties are diplayed in Figure 2.

As exposed previously, marks are the primary visual primitives. Connections on the other hand encode relationship. As such, they do not have their own visual properties. Their aspect is derived from the aspect of the marks they connect. The connections link the two marks at their position. Color and size of the connection are deduced from those of the marks. They can be rendered as arrows or as

simple lines. The rendering defaults to line when connecting to a mark with the "none" shape.

### 3.4 Visualization using marks and connections

Here are ways to realize common visualizations using marks and connections, examples can be seen in Figures 3 and 1 :

- A scatterplot can be implemented using only marks for each point. A multivariate scatterplot can be derived by using different shapes for the marks.
- In a barchart, each bar is a square mark. The bars can be displayed horizontally or vertically.
- A linechart can be derived from the previous barchart by changing the shape of marks to the special invisible shape and adding connections. Each one will link two consecutive marks. We can consider that in this visualization, connections encode temporal adjacency. A variation of the linechart could be to use a visible shape for the marks. Using the same design process, parallel coordinates can also be displayed.
- A node-link view of a graph is straightforward to implement : use marks for nodes and connections for links. Another common representation of graphs is the matrix view. It can also be realized using marks. The vertices at the beginning of the columns and lines of the matrix are made using marks, and the edges, generally depicted by a point at the intersection of the line of its source and the column of its target is also a mark. This is a good exemple of the flexibility of our library, since each vertex in the graph data model is represented by two marks and each edge by a single mark.
- A pie chart can be made by placing circular marks at the same position and using the radial clipping properties to get wedges. The donut chart, a common variation of the pie chart, is realized with the help of the internal clipping property. By correctly using a combination of these two properties, it is also possible to make a sunburst plot by stacking several donut charts.

### 3.5 API design

Designing an API for an information visualization library has been extensively studied by Heer and Bostock in several publications [4, 5, 11]. We chose to follow their major guidelines. As such, our API design is much inspired by D3.js's API.

On a general level, we followed the intention of the Facade design pattern described by Gamma et al. [10]. The entrypoint of the system is provided through a class called Fatum. It enables to add a new mark to the system, get a mark based on its id, connect two existing marks, and get all the marks and connections currently registered.

The Mark object is the main visual primitive we expose. For each visual property, we provide two functions named after this property. The first is a getter, which returns the value of the property, the second is a setter, which changes the value of the property. For instance, the shape property can be accessed using the function *shape()* and modified using *shape(Shape newshape)*. Several shorthands are also provided, such as *alpha*, *width* and *height* to access specific parts of properties with multiple dimensions, such as color, position and size. Every setter returns a reference to the Mark itself, in order to allow method chaining. This is strongly encouraged by [11] and proven successful by the D3 API. We also think it allows for simpler visualization declarations. In addition to the visual properties access and modification, the Mark object can be connected to another mark in an inline fashion using method chaining. While this is redundant with the FATuM Facade interface, it avoids breaking the declarative flow to add a connection. Furthermore, dynamic visualizations often need to transform a single existing entity into multiple new ones, for instance when dynamicaly exploring a hierarchy. Thus, we enable to *clone()* a Mark to have a copy of it freely modifiable.

The last object in our system is the connection. Its interface only consists of source and target access and a deletion operation.

### 3.6 Technologies used

As a rendering library, we want performance to be highly scalable with the number of marks registered. We used C++ and OpenGL to implement our system and the rendering engine, as those technologies were the fastest available. The interface we designed allows to harness the performance of these technologies, while still using visualization concepts.

The choice of these technologies directly enables Windows, MacOS, and Linux native versions of our library. Furthermore, using the Emscripten[7] [17] compiler, we are able to compile our library to a highly optimisable subset of javascript called ASM.js and to WebGL. This enables direct integration of our library in the browser.

## 4 Double buffering for Visualization

Double buffering is a technique widely used in computer graphics to avoid graphical artifacts when updating the display. It consists of maintaining two different buffers to hold pixel values. One is called the "front buffer" and the other the "back buffer". The front buffer is the one used when reading from the buffers to draw on the display. The back buffer is used when writing to the buffers to change the displayed image. When an update operation is finished, i.e. drawing a new image, the two buffers can be exchanged. This operation is called "swap". This way, the displayed image is not changed until the new one is

[7] http://emscripten.org/

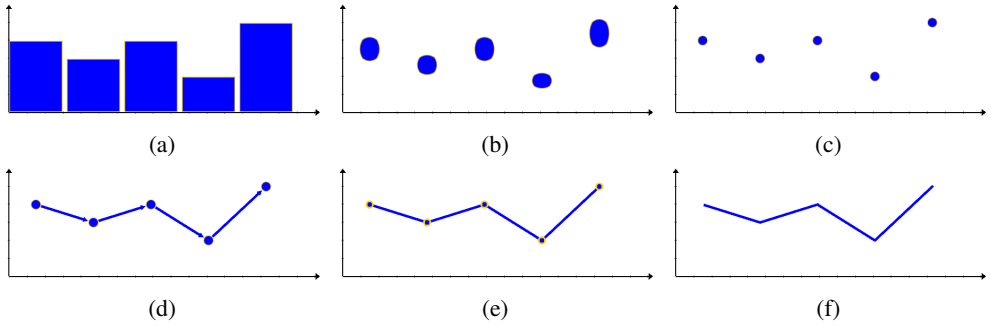



Figure 3: Dynamic transitions of visualizations for the data {4,3,4,2,5} with FATuM. In order to transition between those visualizations, the marks were not recreated, only their properties were changed. (a): A bar chart. Each bar is implemented as a mark. The positions, size are data dependant and the shape is set to "square". (b): Transition to a dot chart [8]. The shape, size and vertical position of each mark were modified. We can see the smooth shape deformation caused by the modifcation of the shape and size properties. (c): The final dot chart state. (d): Connections between the marks were added and are rendered as arrows. (e): Animated transition to a linechart by changing the shape of the mark to "none". Notice the contraction of the marks toward their center. (f): Final linechart state.

complete. This has the advantage of changing the whole image at once and not little by little during long and complex updates. The swap can be implemented either as a copy from the back buffer to the front buffer or only as a role exchange, the back buffer becoming the front buffer and vice-versa. This technique has been used by Munzner et al. [14] to allow iterative rendering of large trees.

We took inspiration from this technique to design the way animation is handled in our library. We also maintain a front and back buffer, but instead of storing pixel values, they are used to store the marks' properties. The front buffer is the one displayed, and the next properties values are stored in the back buffer. This effectively creates two states for every Mark and Connection : the current and next animation states.

### 4.1 Rendering double buffered marks

We use custom OpenGL shaders to render the marks. A shader is a program running on the graphics hardware. Using them permit to harness the parallel nature of GPU.

When rendering the visualization, the two states of every visible element are sent to the OpenGL drawing pipeline. In order to get an animated rendering, an interpolation parameter is used. It varies from 0 at the beginning of the animation to 1 at the end of the animation. The current animation state is computed at the time of the drawing, generaly using a linear interpolation with the formula: $current = front \times (1.0 - interpolation) + back \times interpolation$. The variation of the interpolation parameter enables to control the direction of the animation, making it vary from 1 to 0 will generate the reverse animation. While the interpolation computation is made every frame and for every visible element, it is inherently parallel. Deffering the animation calculation to the drawing pipeline shifts the compute load from the CPU to the GPU. This transfer also has the advantage that no animation computation is made for elements that are not rendered. Thus it is needed to render only elements that are visible. A CPU clipping pass is performed for every frame rendered, taking animation into account to determine for each mark if it is visible or not. This involves verifying if the mark intersects the screen, if its shape is displayed (not "none" at both ends of the animation) and if it is not fully transparent throughout the entire animation. This decomposes the rendering into two phases : the clipping phase and the drawing phase. The clipping phase runs on every mark, but is very quick for each one, while the drawing phase involves more work per mark, but only runs on visible marks. The combination of those phases enables to scale to a large number of marks while retaining interactive framerates. Furthermore, it allows animation on detailed views of huge datasets to be smoothly animated even if the global view cannot be rendered at satisfying framerates.

This technique allows animated rendering at a marginal additional cost. Rendering a static visualization is identical to rendering an animated one, only the interpolation parameter does not vary.

### 4.2 Implementing the buffer swap

Contrary to the computer graphics technique, where the buffer swap is instantaneous, when dealing with animated transition, the buffer swap is delayed during the animation. When the animation starts, the buffers are not yet swapped, only the interpolation parameter is growing toward 1. When the interpolation reaches 1, the state of the front buffer is not displayed anymore, the back buffer state has replaced it entirely. This marks the end of the animation. At this moment, we can finally swap the buffers and

set the interpolation parameter to 0 again. We are then in a new static state.

The user can even specify a callback to execute when at the end of an animation. By launching another animation in it, it effectively enables staged animation.

### 4.3 The need for more buffers

The animation process uses only two buffers to hold the starting and final states of the animation. However, if the user modifies a property during the animation, it will be written to the back buffer. This implies that the animation of this property will now take place as if this new value has been there from the start. This negates the transition effect of the animation and might confuse the user.

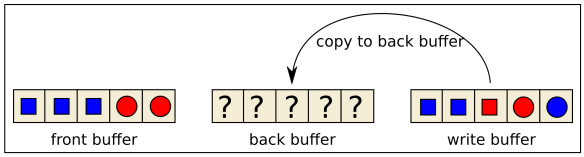
To prevent this, properties are not directly written into the back buffer, regardless of the animation state. We thus add a third buffer to the system, called "write buffer" (see Figure 4). When a property is modified, it is written into this write buffer. At the beginning of the animation, the write buffer is copied into the back buffer. The animation process can now take place with the front buffer containing the previous values and the back buffer containing the updated values coming from the write buffer. At the end of the animation, the front and back buffers are swap as described previously. Using this 3-buffers implementation, the user's modifications are stored in a temporary buffer and thus do not perturb animation. The modifications made during the animation will be used for the next animation.

With this design arises the possibility to copy the write buffer to store different states of the visualization. The states saved could consequently be reused as back buffer states. This enables to start a transition to an already existing state without having to redeclare it. This is in essence the *Memento* design pattern by Gamma et al. [10].

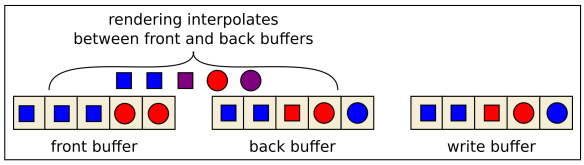
### 4.4 Implementing properties animation

Thanks to the multi-buffered architecture of our system, no additional computation is needed to set the current value of a property when animating. The initial and final values are transmitted to the OpenGL graphics pipeline. Then, using shaders, we are able to compute the current value at the time of rendering, benefiting from the parallelization on the graphics hardware.

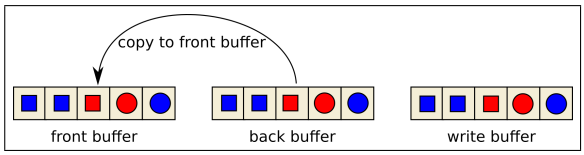
For most of the properties, a simple interpolation between the initial and final values is sufficient. This interpolation can be realized using an easing function to get different animation behaviours, such as slow-in, slow-out, bounce or elastic. We chose to use the same function for the whole animation rather than a per-property value so as not to make the animation confusing. However, the "shape" property is harder to animate. Due to its discrete nature, a simple interpolation of the value will not work. For example, if the initial state is a circle represented by 2 and the final state a square represented by 4, at half




(a) When the animation starts, the back buffer is populated




(b) During rendering, no additional computation is needed




(c) When the animation is complete, the buffer swap can be completed

Figure 4: Steps of the buffer swap for 3-buffers animated rendering.

the animation, the interpolation would give a value of 3, which could correspond to a diamond. Such an interpolation would be dependant on the order in which the shapes are defined and would not give an understandable value outside of the defined integer values.

In [12], Heer and Robertson also needed to animate shapes. They used predefined polyhedral meshes to represent shapes and a vertex correspondance to animate the mesh between two shapes. While this approach gives excellent results for arbitrary shapes, it requires a lot of geometry computations which we could not afford for scalability reasons. Instead, we used signed distance fields to implement the shapes provided in the library. A signed distance field represents for each point the distance to the isocurve of the shape. The distance is negative if inside the shape and positive outside. Formulas to compute such implicit distance fields can be found in [15]. Using the information encoded in the distance field, we are able to efficiently rasterize a shape without additional geometry.

In order to animate the shapes, we can now interpolate the distance fields of the initial and final shapes. This results in a smooth morphing of the shape. The edge expands or contracts toward the target shape. Using this technique, even the shape animation is affected by the easing function applied to the animation. We used implicit distance fields for the shapes currently provided, i.e. distance fields calculated using mathematical function on the fly. However, our approach is completely valid for precomputed distance

fields, which could be used to easily extend the set of available shapes.

### 4.5 Handling Addition and Deletion of Marks

With the addition of multiple buffers to our system, we need a new way of handling marks addition and deletion to maintain coherence between buffers.

When a mark is added, an entry is added into each buffer. The newly created set of visual properties is populated with default values thanks to a default constructor. This ensures that the system is in a coherent state after the addition of a new mark. Once the mark has been added, it is immediately ready to be displayed, since it exists in the front buffer. That is why we carefuly chose the default property values to render the mark invisible through alpha transparency. This way, the mark is displayed, but cannot be seen and the visualization did not change. Every subsequent property modification for a newly created mark will affect the write buffer, as with any other mark. However, this means that at the next animation, the interpolation will take place between the default values remaining in the front buffer and the values specified by the user in the write buffer. This is generally not suitable, since the user might want a mark to appear a certain way in its visualization. To enable finer control on the entrance animation for a mark, we chose to add the *show()* method to the interface of Mark. This method copies the properties of the mark from the write buffer to the front buffer, immediately updating the visualization. This simple addition to the system allows the user to specify the initial (preferably still invisible) state of the newly created mark, call the *show()* method to copy this state into the front buffer and then modify it normally in the write buffer. Thanks to method chaining, this can be realized in a single line of code. For instance, the following line will create a mark, set its color to transparent blue in the front buffer and make it opaque in the back buffer :

```
addMark().color(Blue).alpha(0)
    .show().alpha(255);
```

Upon animation, this mark will progressively appear.

The usage of the *show()* method is not restricted to mark creation. It is suited for every modification of the visualization that does not need animation.

Handling mark deletion is simpler because the deletion happens in the same direction as the animation, since its effect is not immediately displayed. It can be implemented by the users directly using the multi-buffers in a similar manner to the entry. The user can modify a mark to make it invisible, then animate the visualization. Additionally we provide through the *del()* method a way to completely remove the mark from the system. When the *del()* method is called on a mark, it is tagged for deletion and will be removed after the next animation. The user can include it in its visual deletion process as well. For instance a fade-out effect of the mark *m* can be achieved using :

```
m.alpha(0).del();
```

Those system capabilities directly enable brushing. If the user can detect mouse enter on a mark and mouse exit of the mark, he can implement brushing with *c=m.color();m.color(Red).show().color(c);* on enter and *m.show();* on exit. This will change the mark's color to Red in the front buffer and immediately change it to the previous color in the write buffer. Calling *show()* on exit will set the front buffer color to the original mark's color.

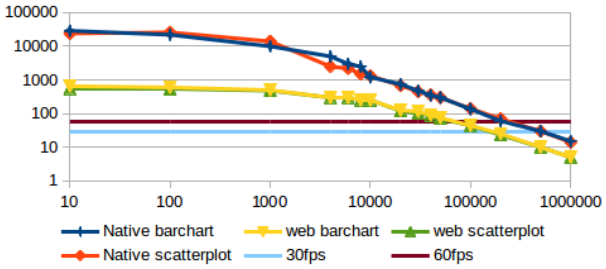



Figure 5: Frames per second versus number of marks displayed on a log/log scale, for both scatterplot and barchart visualizations, on both native and web versions.

## 5 Benchmarks

In order to test the framerate of our library, we rendered bar chart and scatterplot visualizations of random elements. We measured the framerate of the visualization with an increasing number of visual elements. This experiment was conducted on a high-range laptop equiped with an Intel Core i7-4710HQ CPU @ 2.50GHz and an Nvidia GeForce GTX 970M graphics hardware. All tests involved rendering a 800x800 pixels window. We measured the performance of both the native C++/OpenGL version, compiled with gcc 4.8.2 and the -O3 optimisation level, and the ASM.js/WebGL version, compiled with Emscripten 1.29.0, also with a -O3 optimisation level. We used firefox 35 to display the web version. Results are displayed on Figure 5. A big gap can be seen between the two versions. The native version is able to render up to 200.000 visual elements at 60fps. The web version maintains more than 30fps with 100.000 elements in both visualizations.

## 6 Conclusion & Future Work

We showed how to realize a working library to fill the gap between high-level and low-level visualization libraries. Using a common abstraction for the available visual primitives, it is possible to reduce the number of their

visual variables, following Bertin's ideas. This in turn simplifies the rendering code and enables to display a large number of those marks. In order to animate that much visual elements, we inspired from the double buffering technique used in computer graphics. By storing two states of the marks and deffering animation calculation to the OpenGL graphics pipeline, only displayed elements need to be animated, and no additional cost is generated for an animated frame with respect to a static one.

In the future, we would like to integrate custom shapes through precalculated distance fields, allow more user control on connections, text and axis. We would also like to see high-level libraries use our work as a rendering and animation back-end.

## Acknowledgements

This work has been carried out as part of "REQUEST" project supported by the French "Investissement d'Avenir" Program (Big Data - Cloud Computing topic - PIA O18062-645401).

## References

[1] Daniel Archambault, Helen C Purchase, and Bruno Pinaud. Animation, small multiples, and the effect of mental map preservation in dynamic graphs. *Visualization and Computer Graphics, IEEE Transactions on*, 17(4):539–552, 2011.

[2] Benjamin Bederson and Angela Boltman. Does animation help users build mental maps of spatial information? In *Information Visualization, 1999 Proceedings, IEEE Symposium on*, pages 28–35. IEEE, 1999.

[3] Jacques Bertin. Sémiologie graphique: les diagrammes, les réseaux, les cartes., 1967.

[4] Michael Bostock and Jeffrey Heer. Protovis: A graphical toolkit for visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 15(6):1121–1128, 2009.

[5] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. $D^3$ data-driven documents. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2301–2309, 2011.

[6] Fanny Chevalier, Pierre Dragicevic, and Steven Franconeri. The not-so-staggering effect of staggered animated transitions on visual tracking. *Visualization and Computer Graphics, IEEE Transactions on*, 20(12), 2014.

[7] Fanny Chevalier, Pierre Dragicevic, and Christophe Hurter. Histomages: fully synchronized views for image editing. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, pages 281–286. ACM, 2012.

[8] William S Cleveland and Robert McGill. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American statistical association*, 79(387):531–554, 1984.

[9] Niklas Elmqvist, Pierre Dragicevic, and Jean-Daniel Fekete. Rolling the dice: Multidimensional visual exploration using scatterplot matrix navigation. *Visualization and Computer Graphics, IEEE Transactions on*, 14(6):1539–1148, 2008.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[11] Jeffrey Heer and Michael Bostock. Declarative language design for interactive visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 16(6):1149–1156, 2010.

[12] Jeffrey Heer and George G Robertson. Animated transitions in statistical data graphics. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6):1240–1247, 2007.

[13] Jock Mackinlay. Automating the design of graphical presentations of relational information. *Acm Transactions On Graphics (Tog)*, 5(2):110–141, 1986.

[14] Tamara Munzner, François Guimbretière, Serdar Tasiran, Li Zhang, and Yunhong Zhou. Treejuxtaposer: scalable tree comparison using focus+ context with guaranteed visibility. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 453–462. ACM, 2003.

[15] Nicolas P Rougier. Antialiased 2d grid, marker, and arrow shaders. *Journal of Computer Graphics Techniques*, 3(4):52, 2014.

[16] Sébastien Rufiange and Michael J McGuffin. Diffani: Visualizing dynamic graphs with a hybrid of difference maps and animation. *Visualization and Computer Graphics, IEEE Transactions on*, 19(12):2556–2565, 2013.

[17] Alon Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011.