

High-performance Web-based Visualizations for Streaming Data

Eric Whitmire

emwhit@cs.washington.edu

1 INTRODUCTION

Researchers and engineers who design sensor systems often need to visualize real-time signals that update in microseconds. Many times, these visualization tasks support exploratory signal processing. An engineer might tweak a filter to remove noise in a signal or change a threshold to support event detection. In these scenarios,

Developers of such sensing systems often work in C/C++, MATLAB, and Python to facilitate signal processing. Developing real-time visualizations in these platforms is time-consuming, tedious, and often results in poorly optimized rendering that is tightly coupled to a particular use case. As a result, it can be difficult to rapidly explore different signals and intermediate processing steps. In contrast, web-based visualization frameworks have seen significant attention and advancement in recent years. Browsers have become optimized for graphics-intensive tasks and offer a convenient platform for designing dashboards.

The goal of this project is to design a lightweight web-based visualization library to support high-performance visualizations of high-speed streaming data. This work aims to bring smooth, 60 Hz rendering to signal processing tasks without adding significant compute overhead. Specifically, the contributions of this work are:

- (1) A streaming line-plot component that supports semantic zoom and hundreds of signals.
- (2) A streaming spectrogram component that supports geometric zoom and over 16M points.
- (3) A set of supporting components that facilitate integration with existing tools
- (4) A brief performance analysis of the rendering performance

2 RELATED WORK

At the hobbyist level, many users rely on Arduino [1] and Processing [6] for visualization. This supports rich visualizations, but there is tight coupling between the visualization and the signal processing, often in the form of a loop function which contains all functionality.

Python is commonly used for many real-time processing tasks. Matplotlib [4] is a widely used visualization package, but it is optimized for publication-quality graphics. Tools like PyRealtime [9] attempt to enable performant rendering for streaming data using Matplotlib. Bokeh [2] is a Python

library that supports web-based visualization, suitable for use with high-speed data. While this is a promising option for Python users, this work targets does not restrict use to a particular language.

In the web space, tools like Vega [8] and Vega-lite [7] support declarative specification of visualizations, which is ideal for the "fire-and-forget" type of functionality desired. These tools, while they support dynamic datasets, are not optimized for high-speed data and rendering. Plotly.js [5] is another commonly used web-visualization framework, but it also struggles with high-speed data. Cubism.js [3] is a D3 plugin for time series visualization that supports incremental render. Though it uses a similar implementation, it is designed for slowly (several hertz) updating horizon plots and does not scale for high-speed signal processing tasks.

3 METHODS

A high-level diagram of the components of this system are shown in Figure 1. Scrolling line plots and spectrograms are two commonly used visualizations for real-time signal processing. While support for other visualizations would be useful, this project focuses on these two plots as they are both challenging and ubiquitous. In general, each visualization uses two superimposed containers—an svg container for elements like axes and labels and a canvas container for the actual plot. Each plot has separate update and render functions. A call to update supplies new data that is buffered until render is called and the plot is updated. This distinction is important, as it allows data updates at rates faster than can be rendered. Each plot also supports zoom by brushing as well as customizable axes. All axes are set up and drawn using standard d3 functions. The specific rendering techniques and zoom functionality are customized for each plot and are detailed in the following sections.

Streaming Line Plots

A depiction of the rendering process is shown in Figure 2 (left). The line plot is implemented using a single canvas element, sized to match the requested dimensions at screen resolution. Initialization of a line plot requires specifying the number of series/lines and the width/history of the plot in samples. When the plot is rendered, the canvas pixels are shifted left by a number of pixels corresponding to the width

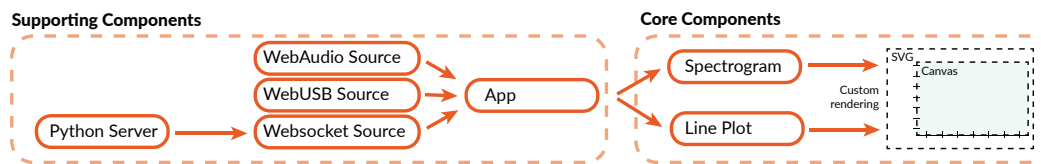
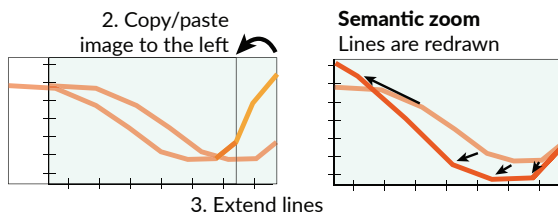


Figure 1: High-level architecture of primary system components

Line Plot Implementation

1. Buffer incoming data



Spectrogram Implementation

1. Buffer incoming data

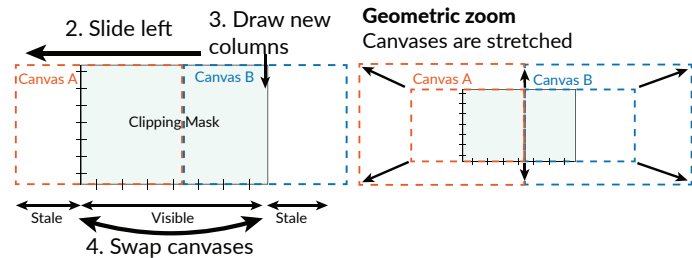


Figure 2: Illustration of implementation of both the line plots and spectrograms

of the buffered data, computing by scaling the number of new data points by the canvas width and history length. The shift operation was carefully chosen to maximize performance. It is implemented by drawing the canvas onto itself with a negative horizontal offset while the canvas compositing operation is set to copy. This effectively replaces the leftmost portion of the canvas with the shifted image in a single operation. Experimental evaluation revealed this method to be much faster than using `putImageData` directly to shift the image. Others have reported similar performance results¹.

To render the new data, standard line drawing functions are used. To maximize smoothness and continuity, the last two data points are overdrawn. For example, if there are 7 new data points to draw, the line drawn consists of 9 vertices. Without this, there could appear to be a gap in the plot depending on the concavity of the plot at the seam. One downside of this approach is the possibility of slightly darker areas where the transparent antialiased pixels overlaps. Another practical complexity of this approach is the consideration of layering effects where multiple lines cross. In this implementation, a fixed ordering is used and the source-over compositing operation is used for the line drawing. This minimizes layering artifacts, but is not a perfect solution. A more robust implementation could consider overlap explicitly and redraw these stale pixels.

A user can zoom in on the plot by brushing. Double-click returns to the default zoom. Since the plots contain streaming data, it can be confusing to the user to zoom in on the leftmost side of the plot, effectively viewing a delayed stream. To eliminate any confusion, the rightmost axis is locked to 0, regardless of the brushed zoom. The zoom for this plot is a semantic zoom; after a zoom, the lines are simply drawn using the scaled transform. This ensures the line width remains constant and the lines remain crisp. In this implementation, the canvas is cleared upon a zoom operation. Future work could consider maintaining a separate buffer of the data to redraw the canvas at the new zoom level immediately after the zoom.

Streaming Spectrogram

Because the spectrogram is a much denser representation of data, additional optimizations are used. The approach for the spectrogram rendering is shown in Figure 2 (right). As before, an `svg` container is used for the axes and the data is drawn on canvas. However, with the spectrogram, two canvas elements are used. Spectrograms are generally computed using specific, fixed parameters that result in a specific output resolution. For example, a 1024 point FFT with 512 sample overlap will produce a 512 element vector every 512 samples. These parameters are usually set based on properties of the signal, independent of the desired size or zoom-level of the spectrogram visualization. Consequently, the approach here conveniently leverages the canvas elements as the datastore for the visualization. Each canvas element is sized in pixels

¹<https://stackoverflow.com/questions/8376534/shift-canvas-contents-to-the-left>

to match the specified FFT size (divided by two) and history, but scaled in CSS to match the desired size on screen. As a result, the image shown by on the canvas may be higher or lower than the screen resolution. Because the canvases extend outside the plot area, a CSS clipping mask is applied to restrict the view.

When the plot is rendered, both canvas elements are shifted left in screen-space via CSS. As with the line plots, the shift size is determined based on the buffer size, width of the plot (in pixels), and history of the plot (in samples). Unlike the line plots though, there are no copy operations here; the entire canvas is shifted. This is an extremely fast operation that is optimized by the browser and requires no pixel manipulation. The new columns of data are then drawn in the appropriate canvas columns. When the first canvas is filled, drawing seamlessly continues onto the second canvas. When a canvas fully exits the left side of the plot, it is simply moved over to the far right side and the process continues.

The pixels within a column are computed using a color scale. This actually happens to be one of the slowest parts of the rendering. If further performance gains are needed, this would be a possible area for optimization.

Unlike the line plots, zoom is implemented geometrically. Upon zoom, the canvases are simply scaled (in screen-space) to match the brushed region. This has the advantage of being efficient and requiring no explicit rerendering or scaling. When zoomed in, the pixels in the rightmost (newest) columns that are vertically out of view are still drawn so that the image is consistent when zooming back out. Like the line plots, the zoom is restricted such that the rightmost column always corresponds to the most recent data.

Supporting Components

Although the two plot components represent the primary contributions of this project, a few additional components are implemented to facilitate ease of use and integrations.

App. A *App* class is provided that wraps a data source and sets up rendering. It parses incoming data and delegates new data to the appropriate plot component using prespecified keys. It also calls the render method of each plot in sync with the browser's animation frame, which is usually called at 60 Hz.

Data Source - Web Audio. For audio applications, the Web Audio API provides convenient access to streaming audio data and analysis. This data source sets up a web audio context and can use either a audio file or the user's microphone as an audio source. It then generates spectral data suitable for plotting on a Spectrogram at regular intervals.

Data Source - Web USB. The Web USB API is an experimental API that allows raw USB access within the browser. This

reference component sets up a connection to a USB device and begins streaming data from it. This component could be used for direct access to devices like Arduinos, without any additional client code.

Data Source - Websocket. Websockets provide a convenient protocol to communicate between the webpage and arbitrary software on the PC. This data source connects to a websocket server and accepts JSON packets that contain a dictionary that maps new data onto each plot key. This source can be used to interface with arbitrary code used for traditional signal processing. A reference demo is provided that uses Python to interface between a sensor and a web dashboard.

4 RESULTS

Performance Analysis

Because rendering performance is a major motivator of this project, this section compares the present work to two other plotting libraries—plot.ly and Vega. These libraries were chosen because of they are modern, full-featured, and commonly used. Two benchmark tasks were devised.

Line Plot Analysis. For this, only Plot.ly was used for comparison. However, this analysis compares Plot.ly's standard renderer with the WebGL renderer. For the task, a 2000 pixel wide line plot was created and *N* lines were added. The number of lines were varied from 10 to 40, with a few larger points tested on the present implementation. The history was set to 2000 points. For the Plot.ly implementation, the *extendTraces* method was used to add new data to the plot. The render time was computed using the Chrome performance analysis tool. These times were used to computed frames per second. For the line plots from this project, the FPS was capped at 60 Hz, even when the rendering time was much less.

Figure 4 shows the result of this analysis. This should be interpreted as an expected frame rate for each scenario. For 10-40 lines, the line plots from this project rendered in under 1.6 ms each frame. The number of lines was increased to unreasonably high level to stress test the plot and it maintained 60 Hz performance until around 1000 lines were added. The Plot.ly implementations dropped under 30 Hz between 10 and 20 lines. Surprisingly, even the WebGL implementation struggled under this load.

Spectrogram Analysis. This analysis compares this work, Plot.ly and Vega. A square spectrogram of size 128 to 4096 was constructed. Random data was added to the spectrogram each frame. Figure 5 shows the result of this analysis. Neither Plot.ly nor Vega could handle a 128x128 spectrogram at interactive frame rates. The present work supported 60 Hz all the way up to 4096x4096 images.

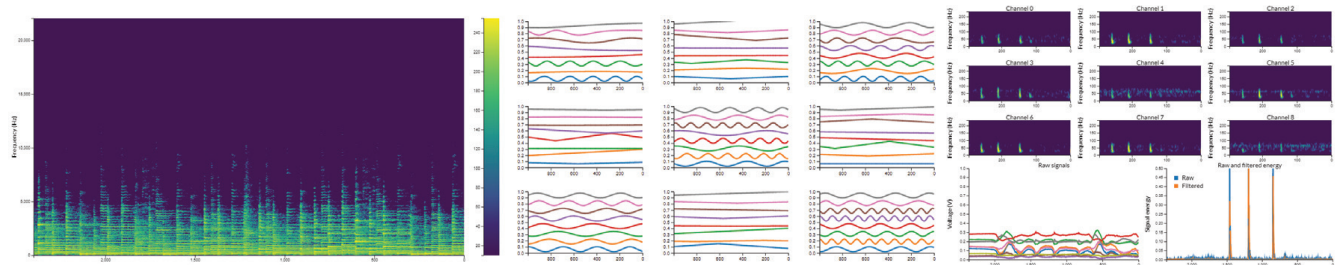


Figure 3: Example applications showing live audio spectrograms, high-density signals, and a real-world sensor dashboard

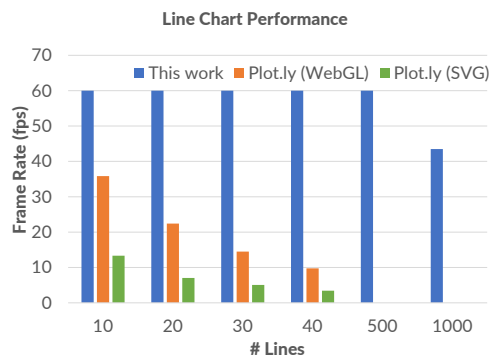


Figure 4: Benchmark performance for line plots

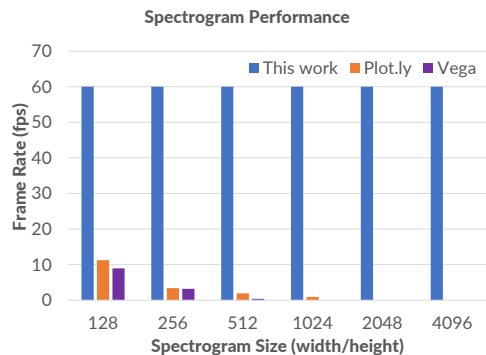


Figure 5: Benchmark performance for spectrogram

Example Use Cases

To demonstrate the functionality of these components, a number of examples are presented in Figure 3. On the left, the Web Audio datasources plots a streaming spectrogram

from live microphone data. In the center, a dashboard demonstrates composing multiple line plots in one page. The right-most figure shows a real-world use case with a sensor device that produces a reading from 9 sensors at 500 Hz. A Python process reads from the device and performs a number of filtering and analysis steps. It initializes a WebSocket server and streams data to the web client. The raw data is plotting in the lower left and each signal's spectrogram is plotted at the top. A few derived signals are plotted in the lower right. This dashboard has replaced a Matplotlib-based visualization for a real-time sensor dashboard for an ongoing research project. It supports more plots and consumes significantly fewer system resources.

5 DISCUSSION

This work has presented two visualization components that have been optimized for real-time streaming signals. This enables the construction of sensor dashboards or other visualizations without having to worry about performance. By cleanly decoupling the rendering from the signal processing, it frees the designer to focus on exploratory signal processing. The use of canvas elements powers much of the system performance and allows the system to render just what is needed.

6 FUTURE WORK

There are a number of areas of potential refinement for this work. The bulk of the effort for this project focused on rendering, but additional work is needed for API design around elements like legends and axis scales. Future work should consider whether this makes more sense as a rendering method within an existing visualization package or a lightweight standalone contribution. Investigation of WebGL techniques would be another promising route. Care must be taken to balance expected performance gains with the complexity and rigidity such an approach would require.

REFERENCES

[1] [n.d.]. Arduino. <https://www.arduino.cc/>.
[2] [n.d.]. Bokeh. <https://github.com/bokeh/bokeh>.

- [3] [n.d.]. Cubism.js. <https://square.github.io/cubism/>.
- [4] [n.d.]. Matplotlib. <https://matplotlib.org/>.
- [5] [n.d.]. plotly.js. <https://plot.ly/javascript/>.
- [6] Ben Fry, Casey Reas, and Others. [n.d.]. Processing. <https://processing.org/>.
- [7] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2017). <http://idl.cs.washington.edu/papers/vega-lite>
- [8] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. 2014. Declarative Interaction Design for Data Visualization. In *ACM User Interface Software & Technology (UIST)*. <http://idl.cs.washington.edu/papers/reactive-vega>
- [9] Eric Whitmire. 2017. PyRealtime. <https://github.com/ewhitmire/pyrealtime>.