Using WebGL as an Interactive Visualization Medium: Our Experience Developing SplatterJs

Alper Sarikaya, *Student Member, IEEE* and Michael Gleicher, *Member, IEEE*

Fig. 1. Our WebGL implementation of the Splatterplot technique [9], showing a subsample of five different sampled Gaussian distributions (about 7.5k points per series). The web-based implementation allows for interactive exploration of hundreds of thousands of two-dimensional points.

Abstract—With web-technologies gaining popularity in use, designed information visualizations can now enjoy wide dissemination without the need for operating system-specific implementation. The process of porting existing visualizations that use GPU-enabled programming (such as OpenGL) to WebGL enables the instantiation of efficient, interactive data visualizations that can scale to larger datasets and larger canvases. In this paper, we present our porting of the Splatterplot system to a WebGL implementation, which we call *SplatterJs*, enabling interactive viewing and summarization of hundreds of thousands of two-dimensional points in the browser. We describe our experiences in implementing this raster-based visualization in WebGL, taking care to retain interactive rendering performance. In particular, we discuss using the GPU as a computational unit to transform data, or alternatively using binary data-streaming facilities built into HTML5 for using backend systems to supply transformed data.

Index Terms—WebGL, Splatterplots, data transformations, GPGPU algorithms

1 Introduction

There are many challenges for effective, scalable visual representation of large datasets. Many of these core challenges for rendering effective representations lie in creating scalable visual designs as well as efficient implementations that allow for interaction. Scalable visualizations allow the viewer to obtain an overview of trends in the dataset, while interactive elements (e.g. zooming, expanding particular trends) allow the viewer to recover individual details upon closer inspection. Interactive methods and implementations are needed in order to tackle the challenge of aggregating and summarizing many elements, while retaining interative speeds for exploration. Information visualizations can use the power of the client's GPU to bring interactive speeds to the scalable display of data. In addition, given the ubiquity of browsers, visualizations implemented in WebGL can enjoy wider reach to potential viewers. However, implementing visualizations in this environment imposes constraints, from the comparatively slow performance of JavaScript to the communication pipeline between JavaScript and the GPU, both of which require additional consideration. In this paper, we present our experiences in implementing the Splatterplot system [9] for WebGL (named *SplatterJs*), with discussion on how we worked within constraints for maintaining client interactivity in the browser.

- *Alper Sarikaya is with the University of Wisconsin—Madison. E-mail: sarikaya@cs.wisc.edu.*
- *Michael Gleicher is with the University of Wisconsin—Madison. E-mail: gleicher@cs.wisc.edu.*

Manuscript received 31 Mar. 2015; accepted 1 Aug. 2015; date of publication xx Aug. 2015; date of current version 25 Oct. 2015. For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

In our iterative development of SplatterJs, we ran into several challenges when porting the native-code OpenGL implementation to one using WebGL. One of the most significant issues was the amount of CPU-based computation done in the original model, which had a significant adverse effect on performance when directly ported to JavaScript and WebGL. This forced us to re-evaluate how we per-

formed operations on the data, including the consideration of moving these computations to the GPU through the use of WebGL. In order to perform reduction and subsampling operations, we used general-purpose GPU (GPGPU) algorithms and stored the results to textures to be used in downstream rendering steps. We expand on the specifics of the WebGL implementation of SplatterJs, and note some general lessons from our experience using WebGL to architect visual scalability into an information visualization.

2 Architecting Visualizations for WebGL

The GPU (graphics processing unit) is a powerful piece of hardware that excels at the massively parallelizable operations such as determining the color of each pixel. The power of the graphics primitive pipeline to take data through programmer-defined vertex shaders, fragment shaders, and various compositing operations is a convenient tool to have in the visualization designers' toolbox. The allure of WebGL is in the marriage of GPU control coupled with the increasingly ubiquitous nature of internet browsers—it is an opportunity to bring GPU-accelerated graphics to the mainstream, without the overhead of installing a native application for the viewer.

WebGL itself is growing in popularity, due to the adoption of WebGL as the graphics standard for many mobile devices. Though the functionality of the standard is considerably behind the current version of OpenGL (WebGL 1.0 currently implements similar functionality to OpenGL ES 2.0 [7]), the opportunity that WebGL presents by providing an interface to utilizing clients' GPUs as a computational unit is very promising for designing visualizations that can handle, process, and render constantly increasing amounts of data. Previous work has started to examine the utility of GL in information visualization. In particular, the work by McDonnel and Elmqvist [10] and Andrews and Wright [1] look at using OpenGL and WebGL shaders, respectively, to render common information visualization designs.

In typical programming practice, data manipulation (such as abstraction, filtering, projection, and subsampling) is often done in CPU code. Once the data is transformed appropriately, it is handed off to the rendering procedures. In the browser, this process can be done in one of two modes: those methods that bind data to shapes and modify those shapes depending on the data characteristics (e.g. creating an SVG drawing using D3.js [2]), or methods that raster individual pixels (2D canvas or WebGL), which use the GPU to compose the final visualization. In contrast to operations manipulating the data directly, some methods operate over image space, transforming the visual abstractions of the data. These methods may operate over shapes generated and bound to data in a SVG image, or pixels generated on a canvas.

In natively-compiled code, data-space computations are typically done on the CPU, while image-space operations are often done on the GPU. This paradigm doesn't necessarily port well to WebGL —though the GPU is a powerful computational unit, the latency of transferring data from memory to the GPU is relatively high. This performance cost becomes more precarious using WebGL within the context of the browser. To maintain responsiveness for the viewer, care must be taken to minimize the unnecessary data transfer across this interface. Our experiences in porting the Splatterplot application to WebGL shows that minimizing this data transfer is key to maintaining interactive speeds for the user.

To minimize the transfer of transformed data from main memory to a GPU buffer, one possible solution could be to push computations to a backend server. While this operation may seem expensive, recent developments in HTML5 have enabled the transfer of binary data directly to WebGL through well-typed arrays in JavaScript called *arraybuffers* [7]. Arraybuffers can be filled manually, through XML HTTP requests (XHR), WebSockets (essentially TCP connections directly to the client), or WebWorkers ("multi-threading" for JavaScript) [6]. Using these interfaces (e.g. setting the `messageType` to `arraybuffer`), one could conceivably use a database or computational backend to stream new or transformed data directly to the viewers's GPU buffer for immediate visualization. Previous work has started to explore this concept of loading data using well-formed blobs, such as the *imMens* system [8]. *imMens* uses specially-designed PNGs to bring a data cube client-side, which supports interactive brushing and linking of large amounts of data, using the client's GPU as a processing unit to pull relevant aggregations from the data cube. We note that this area is potentially ripe for additional work, and encourage the exploration of this space.

An alternative solution to moving data-space computation to the backend could be to move data-space computation directly to the GPU. The utility of this solution depends on the feasibility of porting the data transformation to the GPU, and managing the pipeline of data-space and image-space computations done on the data from data ingest to visualization rendering. The data-space transformations of abstraction, filtering, projection, and subsampling can be performed in WebGL using fragment shaders by employing GPGPU algorithms and re-purposing image-space algorithms for data. For example, we can use the GPGPU pattern of *reduction* [3] to find the maximum value in a texture that stores point density. We can also subsample points by using the depth test, and compute distance fields using algorithms such as the jump-flooding algorithm [12]. We elaborate on this solution with a discussion of our experiences implementing Splatterplots in WebGL, describing the possible ways that data-space computation can be adapted to use the client's GPU.

3 Adapting Splatterplots to WebGL

Splatterplots [9] is an information visualization technique that handles overdraw that occurs when plotting thousands to millions of individual points in a scatterplot. If many points occupy the same x- and y-position in a scatterplot, it can be difficult to distinguish whether one or multiple points are at a particular position in a conventional scatter plot, and more difficult still to provide a density comparison of stacked points (e.g. 3 points at this position against 17 points at another position).

Splatterplots address these issues of overdraw by utilizing kernel density estimation (KDE), which abstracts low-level features (individual points) to provide the viewer with an idea of the density of points in space. The key idea in Splatterplots is to use a screen-space KDE that performs abstraction at overview scales and revealing details at smaller scales, while also highlighs representative outlier points outside of the thresholded region. These heuristics combine to create a visual paradigm that can handle visual scalability for scatterplots at a high-level overview, while also supporting interactivity (through its screen-space parameterization of its KDE) to recover individual points and positions at smaller scales.

The processing pipeline for every rendered frame in Splatterplots is shown in Figure 2. Each operation is colored based on which computational unit it utilized in the original implementation. For each data series, the points are drawn to a texture that collects the density of points on each pixel (*overdraw*). This density is then approximated by KDE, using a Guassian kernel. The maximum density is recorded, a thresholded region is defined (by default, those pixels containing 50% of the maximum density), and representative points are randomly subsampled at regular intervals outside of the thresholded area to reinforce that data exists outside of the thresholded region. Finally, each series is composited together to form the final Splatterplot, using generated colors for each data series that are selected to be the most discriminable. Of particular note in our discussion here are the operations that find the maximum density value (an operation also done by previous visualization systems using density textures, *cf.* [5]) and the subsampling of points for drawing representative outliers.

Many of these methods (shown as hexagons in Figure 2) were suited directly for implementation in WebGL. For example, OpenGL is well-suited for collecting the per-pixel density for every pixel in the viewport: disable the depth test, enable blending, and change both the blend equation to add (`gl.FUNC_ADD`) and the blend function to one (`gl.ONE`). These changes result in counting points that fall in each fragment, effectively creating a texture that stores point density data. Some operations, however, were conceptually easier to implement in native code, such as determining the maximum density (reading the texture into memory and iterating through the buffer) or subsampling

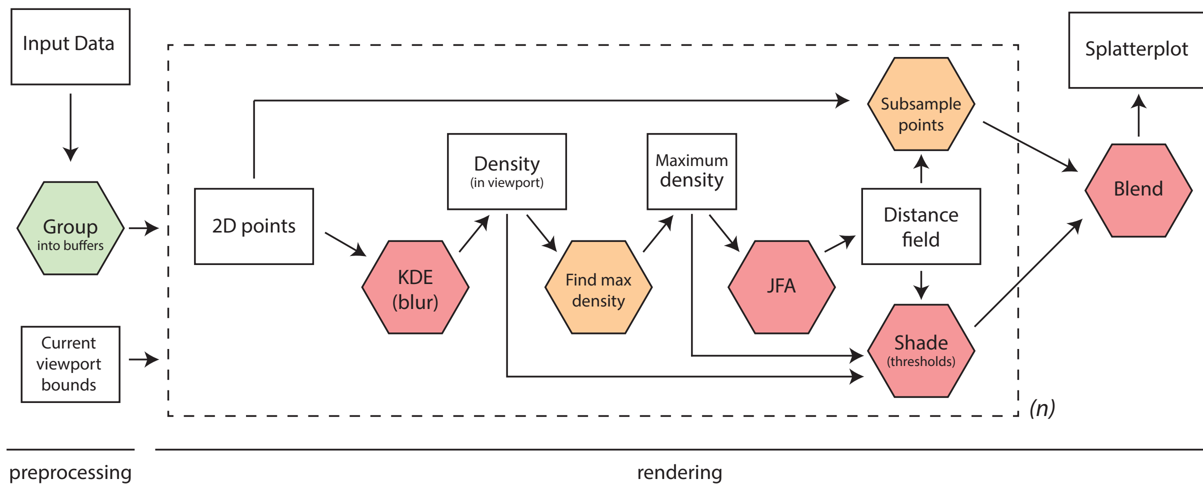

Fig. 2. Data flow through the Splatterplot visualization technique. All processes (hexagons) produce outputs (rectangles), which take the form of textures. Those computations that are performed on the CPU in the original implementation are in orange, while GPU implementations are in red and the one-time preprocessing computation is in green. We adapted all rendering steps to the GPU, including rendering steps that transform input data for downstream rendering (§3.1).

the points (using a spatial data structure to iterate and subsample points to use as outliers). While potentially inefficient for performance, CPU-native code (the project was implemented in both C++ and C#) swallows the performance cost, and the technique remains interactive for the viewer when panning and zooming the dataset (see the Splatterplot manuscript [9] for detail in the scale of the data).

After the Splatterplot technique was presented [9], the visualization stirred interest, but potential users wanted an online solution that would let them quickly visualize their own data and see higher-level patterns. A natural choice to implement this system was WebGL, given the similarity between the GL interfaces. In the implementation of this system, however, several issues were encountered, and needed alternative implementations to maintain the real-time interactivity of the visualization. Most notably, performing transformations of the data (hundreds of thousands to millions of points) to support the various heuristics of Splatterplots in JavaScript proved to bog down clients on even the most advanced systems. Reading a (float-encoded) texture back into local memory is an illegal operation in WebGL 1.0 [7] (see §5.14.12), the method used by the original implementation (though WebGL workarounds exist). Randomly subsampling data points to select exemplar outliers iteratively in JavaScript proved to be too slow.

These sort of issues motivated us to explore re-architecting Splatterplots for WebGL. We appreciated not only having a completed prototype through this exploration, but also reusable components (such as the KDE implementation) for future visualizations using WebGL.

3.1 Performing Data Transformations in WebGL

Here, we will describe the problems we encountered in porting to WebGL, and provide our rationale and solutions to resolve them.

The "Find max density" method (Fig. 2, center) is a rate-limiting step in the rendering process, and is used to determine the thresholded region. In the original implementation, density data for all pixels were read into main memory, where iterative reduction found the maximum density value to be passed as a uniform to downstream methods. Although WebGL does not support the `readPixels` method to read values from textures with encoded floats, methods have been derived for encoding float values into four uint8 values of a RGBA texture according to the IEEE 754 specification (*cf.* [4]), then calling `readPixels` on the surrogate texture to retrieve the original float value. We elected instead to use the common GPGPU design pattern of reduction [3] that reduces values in a texture by consolidating values to a particular corner in order to find the maximum density value.

Given a texture and a reduction step size, an aggregation measure (in this case, `max`) can be done in several passes over the texture. With a step size of 8 pixels, an 8×8 square can be minimized to a single pixel by repeatedly applying the aggregation function. For a canvas of 800×600 pixels, three passes (with a step size of 8) are necessary to reduce nearly 500k pixels to two. Instead of passing a float uniform to downstream shaders that determine the thresholded region, the final max texture can be passed along, with subsequent shaders instructed to pull the value of the maximum density from the top-left corner of the reduced texture.

Representative outlier points are shown in Splatterplots to alert the analyst that data exists outside of the thresholded and shaded regions, even when viewing the dataset at an overview-level where the points would normally be blurred away. To minimize excess data display, only a single point is shown in every 25×25 pixel block (parameter-tunable). In the original native code implementation, points were iteratively picked at random at binned intervals in main memory. This approach did not scale when porting to JavaScript due to the high computational cost.

As all data points are retained in a data buffer in the GPU, we utilized a two-pass algorithm to (1) write point coordinates to a binned location in a temporary texture and (2) draw the point at the coordinates provided by each binned location. To select just one particular point from every grid cell, we associate each data point with a random value between zero and one and assign it to the z-coordinate with the depth test turned on. This has the effect of always selecting a single point for each spatial bin, as well as preventing twinkling (points winking into and out of existence) of individual outlier points when panning and zooming the display.

3.2 Implementation of WebGL Splatterplots

In the user interface of the WebGL Splatterplots application, we have added several sliders that allow the viewer control over the bandwidth of the KDE function, the threshold of the thresholded regions, as well as a outlier clutter metric (nominally the subsampling grid size). The event handlers for these elements modify the uniform parameters passed to the shaders and trigger a redraw of the canvas to interactively provide the user feedback when the slider is moved.

The application allows the viewer to upload their own data files, and asks for feedback when parsing a flat file for the two dimensions to plot (x and y dimensions), as well as an optional 'group by' column, which is used to separate a singular file into multiple data series. A working demo (allowing data uploads) and the source code of the WebGL splatterplots application are available online at `https://github.com/uwgraphics/splatterjs`.

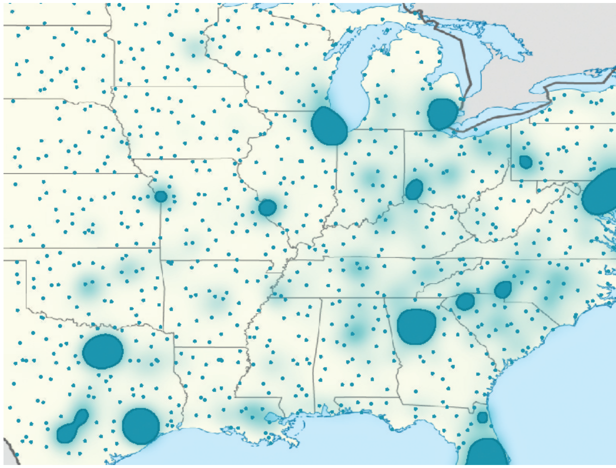

Fig. 3. A single year (2011) of FARS (Fatality Analysis Reporting System) data shown in SplatterJs. The dataset comprises nearly 31,000 points. The viewer is free to zoom and pan about the dataset, much like in typical web-mapping applications.

4 Discussion

Through this paper, we have discussed the use of WebGL for enabling web-based, interactive data visualizations that previously were only possible in natively-coded applications. Through several techniques of moving some data transformations to the GPU, we can empower users to deploy and view a complex visualization system without the additional cost of having to install and run software. A web-client's GPU can be utilized to transform data through aggregation and subsampling for use in downstream visual rendering.

WebGL has proven to be a very portable way to present and disseminate a data visualization. From our experience, however, there are several factors to consider when evaluating the use of WebGL in an information visualization. Chief among these factors is the reality that most of the data-space computation will need to be done in WebGL. While it may be more natural for the programmer to implement data-space operations using JavaScript, the nature of loading data repetitively from the client's browser to the client's GPU and vice versa has shown to be an expensive operation. If possible, all time-consuming data-space operations will have been done before WebGL receives the data, or these operations must be possible with vertex and fragment shaders. The method of delivering data to the client must be reliably quick—although we use flat files in this case, we can also take advantage of binary interfaces for efficiently loading large amounts of data. Finally, the image-space operations required for visualization rendering must fit the GL paradigm: data must either be discrete and aggregated for display by (multiple) shaders, or be encoded in such a way that the data element maps directly to a graphics primitive. Although we have shown just two particular data-space implementations on the GPU, we believe that exploring the space of data transformation implementations in WebGL can help enable visualizations of larger scale and greater complexity in an implementation space more accessible to viewers.

We have concentrated on methods for transforming data on the GPU, but we note that future work can use binary data scaffolds in Javascript to support off-client computation. We transform the data in *SplatterJs* from uploaded comma-delimited files, but we also have had success in other applications using XHR requests for binary data (using `xhr.responseType = arraybuffer`) to fill well-typed arrays in JavaScript, and subsequently loading WebGL buffers with that data. Issues of varying endianness must be supported, however. [11]. Loading binary data to a JavaScript application is not just limited to XHR requests; WebWorkers and WebSockets can also handle binary data, which potentially enable receiving streaming binary-packed data from database and computational sources. Additionally, using the `DataView` construct available in ECMAScript v5 allows for parsing of heterogeneous binary streams. We see this functionality in conjunction with WebGL's ability to handle streaming data as a ripe area for future exploration.

Acknowledgments

We thank Adrian Mayorga, Deidre Stuffer, and organizers of the Data Systems for Interactive Analysis workshop for their helpful comments in framing this work. This work was supported by NSF award IIS-1162037 and NIH award 5R01AI077376-07.

References

[1] K. Andrews and B. Wright. FluidDiagrams: Web-based information visualisation using JavaScript and WebGL. In N. Elmqvist, M. Kennedy, and J. Hlawitschka, editors, *EuroVis—Short Papers*. The Eurographics Association, 2014.

[2] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(March):2301–2309, 2011.

[3] I. Buck and T. Purcell. A toolkit for computation on GPUs. In *GPU Gems*. Addison Wesley, 2004. (Chapter 37).

[4] Carlos Scheidegger. encodeFloat() — Lux. `https://github.com/cscheid/lux/blob/master/src/shade/bits/encode_float.js`. Accessed: 2015-09-15.

[5] E. R. Gansner, Y. Hu, S. North, and C. Scheidegger. Multilevel agglomerative edge bundling for visualizing large graphs. In *Proceedings of the IEEE Pacific Visualization Symposium*, pages 187–194. IEEE, 2011.

[6] Khronos Group. Typed array specification. `https://www.khronos.org/registry/typedarray/specs/latest/`. D. Herman and K. Russell, editors. Accessed: 2015-07-20.

[7] Khronos Group. WebGL 1.0 specification. `https://www.khronos.org/registry/webgl/specs/latest/1.0/`. D. Jackson and J. Gilbert, editors. Accessed: 2015-07-20.

[8] Z. Liu, B. Jiang, and J. Heer. *imMens*: Real-time visual querying of big data. *Computer Graphics Forum*, 32(3pt4):421–430, 2013.

[9] A. Mayorga and M. Gleicher. Splatterplots: Overcoming overdraw in scatter plots. *IEEE Transactions on Visualization and Computer Graphics*, 19(9):1526–1538, 2013.

[10] B. McDonnel and N. Elmqvist. Towards utilizing GPUs in information visualization: a model and implementation of image-space operations. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1105–1112, 2009.

[11] Mozilla Developer Network. DataView - JavaScript. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/DataView`. Accessed: 2015-07-20.

[12] G. Rong and T.-S. Tan. Jump flooding in GPU with applications to voronoi diagram and distance transform. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, pages 109–116. ACM, 2006.