Visualizing Huge Plots on the Web

Mikola Lysenko and Perouz Taslakian

Abstract

We consider a collection of problems that arise when one tries to render a visualization of a set of data points inside a browser, where computational power is limited and memory is scarce. The problems were posed by Plotly

1 Introduction

The best data visualizations illustrate hidden information and structure contained in a data set. As access to large data sets has grown, so has the need for interactive and scalable solutions which connect computer science, mathematics, and industry.

Plotly is an online data visualization tool designed to quickly render graphs in web browsers. The user base of the tool comprises of a wide variety of clients from industry and scientific labs. As these clients have access to larger data sets, demand to enable the rendering of huge two-dimensional plots has grown in recent years. There are various successful attempts at rendering such huge point clouds efficiently [**?**], However, not many studies have address this problem when the rendering needs to be done inside a browser (see for example [Sch]). One of the main challenges of plotting a large number of points inside a browser is doing so efficiently using a limited amount of memory (typically 1GB). With the current online Plotly interface, the maximum number of points that one can plot is about 100,000 points. The primary goal of this paper is to find an executable solution that improves the efficiency of point rendering and allows the plotting of data sets of up to 1 million elements, while preserving interactive and exportable features that the clients expect.

At the lowest level, rendering is handled by the GPU (Graphics Processor Unit), which is very efficient in processing a huge number of points, specially if many end up outside the viewport. The bottleneck in GPU processing however lies in the rasterization step when a large number of points are mapped to the same screen pixel. *WebGL* is a low level graphics API that uses GPUs directly for extremely fast rendering. It is based on OpenGL, but is designed for the web. To render an image in WebGL, functions send data to your GPU for processing, and images are drawn on top of a canvas element. Shapes are created by scripts that contain vertex and fragment shaders, that assign colors to each pixel contained in the shape. WebGL is fast, offers high performance rendering, good interactivity and excellent control.

In this report, we address the problem of rendering a huge (i.e. at least one million) number of points inside a browser (i.e. memory is limited) using WebGL. In addition to drawing considerations, our solution requires thought about how plots are stored on the server, and how stored data is downloaded to the browser. Specifically, we require an efficient way to store and retrieve data server-side based on a plot viewport. Given the

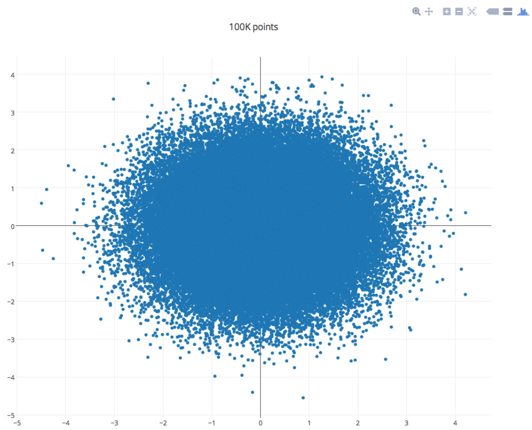

Figure 1: A scatter plot of 100,000 points in 2D.

potential bottleneck at the rasterization step of the GPU, this means that given a viewport, we need a fast way of deciding (whether on the server or on the client) which points should be sent to the GPU for rendering so that the plot looks effectively just as it would if you plotted everything. The client then can query the server on zoom/pan events to update the data.

It is also essential that existing plot features are maintained. In particular, advanced styling features (such as custom glyphs, colors, sizes, borders, etc.) must be preserved, and users must be able to highlight individual points (this is done by hovering the mouse over a point in the plot).

We concentrate on two types of two-dimensional plots: scatter plots and line plots. Our goal is to render over one million data points n by allowing at most $O(npolylog(n))$ steps during preprocessing and using $O(n)$ amount of working memory. In addition, the algorithms developed must be simple enough to allow implementation in a reasonable amount of time in order to be cost effective.

Given a set of n vectors in 2D and a shape S (which we call the *marker*), a *scatter plot* is the image formed by the union of n copies of S translated by each vector. A *line plot* is a piecewise linear curve (i.e. a polyline).

Let $P = \{p_0, p_1, \ldots, p_n\}$ be a set of primitives with $p_i \in \mathbb{R}^2$ for all $i = 0, 1, \ldots, n$. Let $s_0 > s_1 > \cdots > s_k$ be a set of scales (in our case, pixel sizes). The *cover order* of P is the filtration

$$F_0 \subseteq F_1 \subseteq \cdots \subseteq F_k = P$$

such that $\cup P \subseteq \cup (F_j \oplus C_{s_j})$ for each $j = 0, 1, \ldots k$, where C_{s_j} is a circle of radius s_j. The *level* of a primitive $level(p_i) = min_{p_i \in F_j} j$. In other words, given a pixel size s_j, we would like to determine a subset of object $P' \subseteq P$ such that every primitive overlaps with one or more primitive of P' dilated by the circle of radius s_j (ideally, we want the minimum number of objects that would cover all others). This means that for a fixed zoom level, we only need to render the primitives in P' as all other primitives are “hidden”. Therefore, given a set of points/line segments, our problem can be reduced to the problem of finding a cover order of these primitives for a given zoom level.

For pointsets in 2D, a cover order can be easily computed by the use of a spacial data

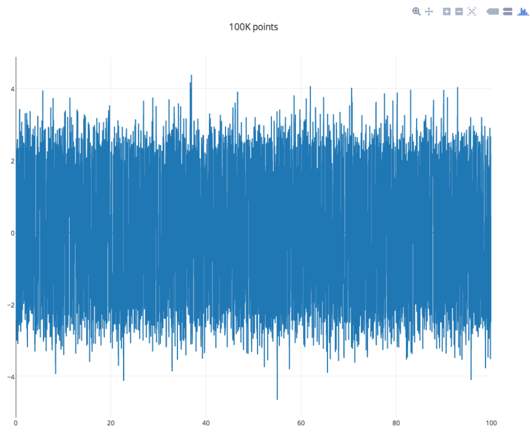

Figure 2: A line plot of 100,000 points in 2D.

structure called *quadtrees*; however, computing a cover order is not so evident in the case of line plots.

In what follows, we describe a method that will allow us to render and process a scatter plot of 100 million points within a browser. We then discuss the challenges of using the same technique for rendering scatter plots and propose a few possible alternatives.

2 Progressive Loading of 2D Scatter Plots

In this section we describe an algorithm that allows us to render 100 million points. The key idea behind the algorithm is the use of the quadtree data structure, and the building of this data structure in a way that minimizes the use of working memory.

A *quadtree* of a set of points in the plane is a geometric data structure that is constructed by recursively splitting the area (usually the square bounding box of the input pointset) into four equal-sized squares until every square contains at most one point [FB74].

Let A be a set of n points in the plane, and let z be a zoom level (which is a function of the current pixel size and screen resolution/size).

Preprocessing. Once the points A are loaded in the client (we will assume A is an array of points), we preprocess them as follows.

1. Construct a quadtree of the pointset A using a depth-first traversal of the points and storing the level of each point in the quadtree in a new array L. Store the points in array Q. Thus, the level of point $Q[i]$ is stored $L[i]$.

2. Sort the points of Q in increasing order of level and of x-coordinate using an in-place sorting algorithm (such as Quicksort). The size of array L will be equal to the height of the quadtree.

The ordering of the points by their level in the quadtree gives us a cover order for the points of the scatter plot: For a given zoom level z, this ordering of points by level gives us a way to decide which points need to be rendered and which do not because they are "hidden" behind others. Consider a square area of the quadtree, which corresponds to a node p at some level l in the tree. If the zoom level of the rendering matches with l, then the quadtree structure tells us that all vertices in that single square area will be mapped on top of each other on the screen in the final rendering on the screen. Therefore, instead of rendering all the points (p and its descendants), we can instead render only point p.

Rendering. Once array Q is constructed and sorted, we can now flush array L and send Q to the GPU. To render for a given level, we ask the GPU to draw some superset of the points that are visible on the screen and are not hidden behind others.

1. Compute the size of a pixel in the data coordinate to get the current zoom level z.
2. Compute the x_{min} and x_{max} of the screen.
3. Starting at level z, for each level above z
   (a) Find the predecessor x_p of x_{min} and successor x_s of x_{max}.
   (b) Ask the GPU to draw the points whose x-coordinates are between x_p and x_s.

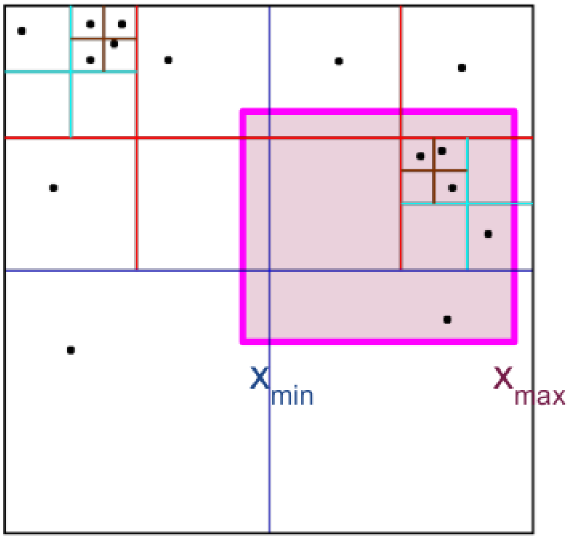

Figure 3: The partitioning of the points into quadtree partitions. The pink/bold rectangle represents the area that will appear on the screen.

Conceptually, given a zoom level z, we will ask the GPU to draw all the points that have level *at most* z, thus ignoring all points that are deeper than z in the quadtree. We further trim the number of rendered points by excluding the ones that are outside the vertical strip that encloses that sides of the screen. See Figure 4.

Analysis. A quadtree of depth d storing a set of n points has $O((d+1)n)$ nodes and can be constructed in $O((d+1)n)$ time. In general, the depth of a quadtree of a set of points in the plane is at most $\log(s/c) + \frac{3}{2}$, where s is the length of one the edges of the bounding box and c is the distance between the closest pair of points. Thus, in the worst case, the depth of a quadtree can be arbitrarily bad.

Preprocessing may take a long time if the quadtree ends up having height (n). This happens when, for example, the input consists of a big cluster of points that are far from the rest of the points. To avoid such bad cases, we introduce a slight modification to the way we construct the quadtree: After every split of the area into four quadrants, we check each quadrant to see if any contains greater than 90% of the points. If such a quadrant is found, then split the point cluster arbitrarily into two equal sets and construct a new quadtree for each one separately. Doing this will preserve the level information for each point, which is all that we need to render effectively, while at the same time decreasing the height of the tree.

The algorithm described in this section has made it possible to render 100 million points inside a browser while maintaining reasonable interactivity.

A note on transparent markers of the same color and shape. So far we have assumed that the markers we are rendering are of solid color. This assumption made it possible to replace all points that are stacked on top of each other with a single point on the screen. If the markers are partially transparent however, replacing overlapping points with a single, equally-transparent marker will not work as we need to blend the colors of all the markers that are hidden. Blending different colors assumes an ordering of the points, which makes the problem even more complex. Here we propose a simple fix to the case of partially transparent markers that have the same color and shape. To blend colors properly, we need to know the number of points hidden behind the point we want to render. We can save this information while creating the quadtree in the preprocessing step — for each node of the quadtree, we keep track of the number of descendants in its subtree and store this number in a separate array. While rendering, we can then use this information to blend the colors appropriately.

3 Displaying 2D Line Plots

Unlike for scatter plots, computing a (reasonable) cover order for line plots is a more challenging task. In the line plot setting, the use of quadtrees is not efficient because a segment may hit $O(2^h)$ boxes in level h resulting in a huge quadtree.

In the line plot setting, we want to build covers inductively and solve for $F_j\ F_{j-1}$. In other words, we want the smallest set of line segments that cover $P\ F_{j-1}$ at scale s_j.

Ideas.

1. *Set Cover Approximation.* The cover ordering problem for line segments can be reduced to the problem of Set Cover (which is NP-complete). This allows us to use a known approximation algorithm for finding set covers. One such algorithm has an approximation factor of $\log n$, with a running time of $O(n^3)$.

2. *Greedy Longest Segment.* In this approach, we sort the segments and process them by decreasing order of length. The running time is $O(n^2)$.

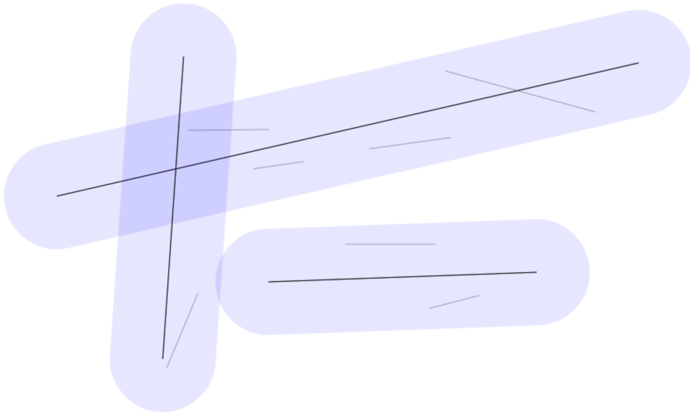

Figure 4: The three thick line segments constitute a cover for all the segments for a given zoom level. Only these three segments need to be rendered by the GPU.

3. *Divide-and-Conquer.* We recursively split the segments (arbitrarily) into two equal groups and process separately. The running time is $O(n \log n)$.

Acknowledgements

We would like to thank the participants and organizers of the Sixth Montreal Industrial Problem Solving Workshop where the problem was first posed.

References

[FB74] Raphael A. Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[Sch] Markus Schütz. Rendering large point clouds in web browsers. In *Proceedings of CESCG 2015: The 19th Central European Seminar on Computer Graphics*.