Splatterplots in WebGL

Chris Hopman

Introduction

Splatterplots are a new technique for displaying point data that scales well with the number of points. With a large number of points, the amount of information exceeds what can be displayed. Splatterplots show dense regions as contour-bounded filled areas and subsamples points outside those areas. In this way, overall shapes, relationships between sets, and the range of outliers is preserved, even when the number of data points is much greater than the number of pixels.

There is currently an implementation of Splatterplots (with accompanying Splatterplot Matrix) written in C++ (referred to as the desktop implementation). This report addresses the challenges and changes made to recreate that implementation in WebGL. To learn more about the design and implementation of Splatterplots, refer to [1].

We will discuss the changes and challenges encountered when translating each step of the Splatterplot pipeline.

Kernel Density Estimation

Point aggregation into contours is achieved through density estimation and thresholding. As a first step we compute a density scalar field from the data points of each subgroup. We use a simple version that allows us to leverage the GPU for efficient computation. KDE estimates continuous density values by summing the contribution of discrete samples around a point based on a kernel function.

Our implementation accumulates the points into a framebuffer texture creating a 2D scalar field of point counts per pixel. As WebGL does not support single-channel textures, we render one data group into each channel of a texture - this allows us to do almost all of the following steps in parallel on four groups at a time.

The scale field is convolved with a 2D gaussian kernel using a two pass approach. One major limitation of WebGL is that loop variables and conditions must be compile-time constants (they cannot use uniforms for loop conditions). This means that the KDE shader must be able to accomodate the largest kernel window that is allowed. There is a large performance cost to the unnecessary loop iterations when they are not needed

(which is always the case for the Splatterplot Matrix discussed below). For this reason, we actually use four different kde shaders each with a different number of loop iterations (16, 32, 64, 96). Which shader to use is decided such that the number of iterations is at least three times the kernel bandwidth parameter.

Max Density and Thresholding

To initialize the distance transform in the next step, we must calculate the maximum density and threshold the density field to determine which points lie within the contours. The desktop implementation does this on the CPU, i.e. it reads the density texture from the previous step and finds the maximum density on the CPU. Due to the limitations of WebGL (or at least of the current implementations of it), it is extremely slow to read any data off of the GPU, and so the WebGL implementation must do this step on the GPU.

This is done on four channels in parallel essentially in the same way you could generate mipmaps. That is, repeatedly create a smaller sized texture where you reduce a 2x2 block of points to a single point with the maximum values across each channel.

Distance Transform

In order to ensure that outlier points do not clutter contours we filter out any points that lie too close to the contours. Instead of calculating the distance to the contour for each data point, we use a distance transform that calculates the distance to the closest point on the contour at each point in space. This creates a scalar field that we can efficiently sample. A distance transform also facilitates the anti-aliased rendering of contour outlines in a shader by allowing each pixel to query whether it falls in the immediate vicinity of the contour.

The desktop implementation uses the Jump Flooding Algorithm to propagate the position of the closest boundary point and the distance to that point. The WebGL version takes a similar approach, but only propagates the distance. This is not as accurate, but only requires a single channel per data group.

Sampling

We want to retain any points not aggregated by contours without creating visual clutter or overdraw. To do this, we limit how close outlying points can be to each other and how close they can be to the contours. For the sampling stage, we divide the current viewing area into a grid of bins, and only render a single point from each bin.

This is another step that is easily done in the CPU and is done that way in the desktop implementation. Again, in WebGL reading any data off the GPU is extremely slow and so we need to find a different approach to this step.

To do sampling in the GPU we use a two-phase approach. First, we render the points into a rectangle where each point corresponds to a single bin. The rendered color of a point encodes the actual position of that point. In this way, after rendering we have rendered at most one point into each bin. Note that as we are encoding two values for each data group, we do this step one group at a time (rather than four in parallel). It would be possible to do this in two parallel steps rendering x-position into one texture on the first step and y-position into another on the following step.

Once we have rendered the points into bins, we then convert that into the sampled points drawn at their actual position. For each pixel, we look up the position stored in that pixel's respective bin, and if that point is close enough to the pixel we draw this pixel as the point.

Coloring, Texturing, Contouring, and Combining

In the desktop implementation, coloring, texturing, and contouring is done in the GPU and then that result and the sampled points are read into program memory and composed together on the CPU. Again, we do this all on the GPU. In fact, we simply add combining to the same shader that does coloring/texturing/contouring.

Splatterplot Matrix

The WebGL approach to the Splatterplot Matrix greatly differs from that of the desktop implementation. On the desktop implementation, each element of the matrix is its own fully functional Splatterplot which supports zooming and panning. The WebGL implementation does not allow this. This allows several optimizations in the WebGL implementation.

For the Splatterplot Matrix, each element is rendered into a 128 by 128 block of a 1024 by 1024 texture. In this way, we will work on an 8 by 8 grid (enough for a matrix of 11 data variables) of these matrix elements all in parallel. This leads to some small changes to each of the steps described above. As each of the changes are very minor, we use the same shader programs for each step with only slightly changed arguments.

The initial scalar density field is straightforward, the points are simple rendered into

a small area of the framebuffer texture for each data group. Since we do not allow zooming or panning in the matrix (and no other adjustable parameters have an effect on this step), we only need to do this once for a given dataset.

The kernel density estimation also needs only a small change: we ensure that the blurring does not cross the border into neighboring elements. As we are working in a 128 by 128 block rather than 1024 by 1024 we also adjust the kernel bandwidth parameter.

Next we must calculate the maximum density. For this step, we simply do not take as many steps. Since the smaller block has power of two width and height, the maximum values will naturally be contained within the correct block. The later steps that use the maximum value need to know to look at the correctly offset position when they are working on the matrix.

The only change required for the thresholding is the lookup in the maximum texture mentioned above.

For the Splatterplot Matrix we do not calculate the distance transform. As discussed below, we do not dynamically sample the points for the Splatterplot Matrix and so calculating the distance transform would only be used for contour outlines which are much less helpful in the limited space that matrix elements are drawn in.

In the matrix, we sample points only once per dataset. This is mostly done as the performance of dynamically sampling the points (at least when done the straightforward way) is extremely limiting. Since we do this without the contours available, we can not reject points to close to the contours. In the limited space that the matrix elements are drawn in, we are willing to make that trade.

The coloring and combining step is very similar except that now we must reject points that lay on their own data group's contour in this step as we could not do it in the last step. In addition, we do not do texturing of contours in the matrix as it tends to greatly clutter that small area.

With these changes, every calculation that is required when the input parameters are adjusted can work on all of the up to 256 elements (4 channels, 8 by 8 grid) in parallel, and scales independent of the number of data points in the dataset. This allows us to dynamically update the Splatterplot Matrix along with the main view with only a minor performance penalty, even for very large datasets.

Future Improvements

The one major step that is skipped in the splatterplot matrix is the sampling of points. This was done because the straightforward approach requires re-rendering all the points in the dataset many times. However, there is a way to do this that does not require re-rendering the points. Since we do not allow zooming or panning, we can take the already rendered points and use a process similar to how the maximum density is calculated to also collect points into their respective bins. Once the points are “rendered” into their bins, the process would continue as in the main view.

When panning or zooming, the bucket that a point falls into in the sampling phase will change. This leads to different points popping into and out of view as the user pans or zooms. The desktop implementation addresses this by offsetting the bucket calculation based on the zoom/pan state so that (at a fixed zoom level) a point always falls into the same bucket. The WebGL version does not yet incorporate this improvement.

Conclusion

There were several challenges to porting the splattplot visualization implementation to WebGL. The most important limitations to deal with were the fact that in WebGL it is incredibly slow to read any data off of the GPU. This may be due, in part, to Chrome’s off process WebGL command buffer (Chrome has a flag to enable in-process rendering, but the use of this was not explored) and the requirement that all loops (and vector/ array lookup indices) be compile time constants.

The slowness of reading data off of the GPU required us to move all of the calculations that were done on the CPU to the GPU. The only thing that could not be done on the GPU was the shape optimization discussed in the paper, which was not actually being done in the desktop implementation either.

Everything considered, we were quite successful in porting this implementation to WebGL.

[1] Splatterplots: Overcoming Overdraw in Scatter Plots