Accelerating Web-based Graph Visualization with Pixel-Based Edge Bundling

Jieting Wu, Jianxin Sun, Xinyan Xie, Tian Gao, Yu Pan, Hongfeng Yu
University of Nebraska-Lincoln, Lincoln, NE, USA

***Abstract*—We present a novel web-based framework, named Pixel-Based Edge Bundling (PBEB), for effectively and interactively visualizing large graphs. Our framework combines an image-based edge-bundling method and a parallel texture-based processing scheme, allowing us to effectively and efficiently compute edge similarities using kernel density estimation and subsequently group these edges into bundles based on their similarities. We discuss several challenges related to developing large-graph visualization on web-based platforms. To accelerate the edge bundling process and enable interactivity in web-based environments, we leverage texture-based parallel processing, a standard feature of WebGL. Our framework optimizes an end-to-end process, from bundling to rendering, enabling practical and interactive visualization of large graphs in a web-based setting. We demonstrate the superior performance of our framework by conducting comparisons with existing web-based and CUDA-based edge-bundling methods using various standard graphics cards on different devices.**

***Index Terms*—graph visualization, edge bundling, similarity, GPU, WebGL.**

I. Introduction

Graphs or networks are extensively employed to represent data relationships in diverse fields, such as biology, transportation, deep learning, and so on [1]. Visualization, especially web-based solutions (e.g., D3 [2]), have provided a versatile way for researchers and practitioners to visually analyze and detect potential patterns within graph datasets. However, as data sizes continue to increase substantially, an effective and interactive visualization of large graphs on web-based platforms becomes an increasingly formidable task. While researchers have introduced various approaches to tackle large-graph visualization, effectively addressing issues related to visual clutter and performance bottlenecks remains a non-trivial task. These challenges have hindered the successful integration of interactive large graph visualization into web-based applications.

First, large graphs typically contain numerous nodes and edges that present intricate interconnections between nodes. On the other hand, many displays, in particular those found on portable devices, often have limited screen space. Consequently, a straightforward visualization method (e.g., a node-link diagram) for large graphs can easily result in extensive overlapping and crisscrosses among nodes and edges, incurring server visual clutter that is difficult to interpret. While some exiting efforts [3]–[5] have developed applications that can adjust to different devices for web-based graph visualization, they have primarily dealt with moderately sized graphs. The challenge of effectively visualizing large graphs with reduced visual clutter with constrained scree sizes remains an open problem.

Second, it is challenging to implement interactive processing and rendering for large graphs on web-based platforms that often possess limited processing capabilities. The availability of standard parallel programming libraries and architectures, like CUDA and OpenCL, is constrained in web-based environments. This limitation becomes especially problematic when there is a need for client-side processing to enable interactive visualization of large graphs on a web browser. While resorting to server-side processing [6] can partially mitigate this issue, it cannot guarantee full interactivity, as it may still encounter potential problems like network latency and service interruptions.

We aim to tackle the above challenges related to visual clutter and performance bottlenecks by developing an effective and efficient method, named *Pixel-based Edge Bundling* (PBEB), for visualizing large graphs on web-based platforms. Our approach is based on edge bundling, a technique that has been demonstrated to be practical for visualizing large graphs across various applications and scientific domains [7]. Edge bundling methods can reduce visual clutter by visually grouping or clustering the edges of a node-link diagram of a graph based on a similarity metric. This strategy can reduce excessive edge crossings and unveil the high-level patterns within the graph, thus mitigating visual clutter.

To enable parallel processing on web-based platforms, we harness the parallel computing capability of WebGL within PBEB to accelerate the edge bundle process when accessed through web browsers. When handling an input graph, PBEB employs the concept of kernel density estimation [8] to iteratively calculate the density map of the graph edges to form edge bundles. In addition, we develop a pixel-based method in PBEB to leverage the parallel functionalities of WebGL within a web browser. This approach enables the parallelization of edge bundling processing, which traditionally demanded advanced GPU parallel processing platforms [8]–[11].

Our contribution is twofold. First, we develop a web-based framework to visualize large graphs effectively by leveraging edge bundling to identify edge similarities. Our framework yields visualization results that can uncover the overall structures and backbones of large graphs within web-based environments. Second, we address the challenge of parallel processing in edge bundling using a web-based method, minimizing data transfers between CPU and GPU to enhance real-time interaction performance. Our experiments demonstrate that our web-based framework outperforms the

existing approach [11] by a significant speedup. At the same time, we ensure the high quality of our visualization results on web-based platforms.

II. Related Work

This paper focuses on building a web-based framework for effective and interactive large graph visualization. We refer those interested in information visualization to the following work and surveys. Liu et al. [12] presented a comprehensive survey and key insights into the fast-rising information visualization domain. Landesberger et al. [13] and Beck et al. [14] surveyed many techniques for large graphs and dynamic graphs, respectively. Herman et al. [15] and Vehlow et al. [16] had conducted comprehensive surveys for graph visualization. In this section, we will first recap some of the web-based tools and studies that tackle node-link diagram visualization for large graphs. We will also recap some of the edge bundling techniques since our framework is based on an edge bundling method. For a comprehensive survey of edge bundling methods, we redirect readers to the edge bundling survey [7].

Web-based visualization systems have provided a portable means for designers and researchers to build customizable visualization systems. Protovis [17] was a visualization system that employed HTML, JavaScript, and Scalable Vector Graphics (SVG) to generate interactive web-based visualization. A web-based library Document-Driven Documents (D3) [2] emerged as a new visualization system to directly manipulate page content elements by binding data to document elements. The performance of D3 was demonstrated to be twice as fast as Protovis. However, when the problem size reaches a certain large amount, e.g., visualizing thousands of edges with force-directed layouts (i.e., force-directed node-link diagram or force-directed edge bundling graph) [3], the performance of the interactive visualization drops drastically. A web-based implementation [4] embedded Multilevel Agglomerative Edge Bundling (MINGLE) [18] in graph visualization for web access. To overcome the performance issue of the traditional web-based visualization system, Texture-based Edge Bundling (TBEB) [5] employed the texture feature of WebGL to accelerate bundling processes using the capability of standard GPUs for web-based force-directed edge bundling graphs. It achieved $240\times$ speedup compared to the implementation of D3 using a graph with approximately two thousand edges. However, this work can not handle very large graphs due to the texture size limitation. Hence, it is less scalable for large graphs. Our PBEB framework has enhanced the scalability and enabled the visualization of larger graphs on web-based platforms.

The basic edge bundling methods used in our framework are Kernel Density Estimation-based Edge Bundling (KDEEB) [8] and CUDA Universal Bundling (CUBu) [10], which are the image-based edge bundling methods. Telea et al. [19] first introduced the image-based method for edge bundling visualization of graphs. Given a graph layout, they rendered the clusters based on an image-based technique that used kernel splatting. The bundles were visualized using the overlapping shaded shape generated by the kernel splatting, such that the coarse-scale graph structures were emphasized. The following work, KDEEB [8], built a skeleton for edge merging. Using kernel density estimation, the method first transformed the input graph into a density map. Sample points of edges moved towards the local density maxima to form bundles. Bottger [20] worked towards visualizing the 3D correlations between neurons inside the human brain for medical data. It used the advection method of KDEEB. Peysakhovich et al. [9] extended the basic idea of KDEEB to distinguish bundles based on different edge attributes. CUBu [10] used GPU parallelization to accelerate density assessment and enabled bundling of a graph with a million edges at interactive frame rates. It also coupled different attributes and metrics in its density assessment, enriching a wide variety of styles of graph layouts. The Fast Fourier Transform Edge Bundling (FFTEB) method [11] addressed the scalability of density estimation by transforming the density space to the frequency space and was slightly faster than CUBu. Furthermore, the approach of Edge Bundling Using Moving Least Squares Approximation (MLSEB) [21] utilized a distance-minimizing approximation function to generate high-quality bundle effects. It demonstrated scalability and efficiency, particularly when applied to large graphs. However, these methods require advanced GPU parallel processing capabilities often unavailable on web-based platforms. Our PBEB framework exploits standard WebGL texture functionality and allows web-based edge bundling of large graphs.

III. Algorithm

To enable effective visualization of large graphs on web platforms, we choose edge bundling methods that can group or cluster edges based on their similarities and generate a set of edge bundles to largely decrease the excessive edge crossing (visual clutter) in the final drawing. Many existing works have shown the effectiveness of bundling techniques [7], [16], [22]. After carefully studying the family of edge bundling methods, we find that the *Kernel Density Estimation based* (KDE-based) methods are the most appropriate ones for our framework. The rationale for choosing KDE-based methods is discussed in Section III-A.

Among the fastest KDE-based methods, CUBu and FFTEB use parallel computing architectures such as NVidia's CUDA or OpenCL. It is non-trivial to deploy these methods on a web-based platform because the support of the parallel computing library is limited. In our framework, we propose to use WebGL's texture feature and shaders programming to handle parallel computing. That enables a real-time edge bundling construction on any standard web browser. The modification required to adapt the KDE-based method on a web-based platform and the implementation details will be discussed in Section III-B.

A. Bundling Algorithm

Given an input of a graph drawing G = (V, E) where vertices V = $\{v_i \in \mathbb{R}^2\}$ and edges E = $\{e_i \subset \mathbb{R}^2\}$, our framework aims to visually bundle the edges based on a similarity metric, and

the constructed bundles can be rendered in real-time on a web platform. In an edge bundling visualization, an edge e_i is visualized as a straight line or curve and can be modeled as a polyline. The polylines' control points, namely *sites*, are non-uniformly or uniformly sampled by a predefined sampling step σ. Formally, the sites are presented as x_j and $e_i = \{x_j\}$ accordingly [8], [10], [11], [23]. In this paper, we use KDE-based edge bundling methods. We will describe the basic pipeline of the KDE-based methods and how we adapt the idea to our implementation.

The typical KDE-based edge bundling methods [8]–[11] essentially use the mean shift principle [24]. The idea is to iteratively group the edges using the normalized gradient of an edge density map ρ of E. To obtain the edge density map $\rho : \mathbb{R}^2 \to \mathbb{R}^+$, all sites x_j on all edges of E are convolved with a radial kernel K:

$$\rho(x \in \mathbb{R}^2) = \sum_{y \in P} K(\frac{||x - y||}{p_R}), \tag{1}$$

where K is a kernel function (Epanechnikov or Gaussian), p_R is the radius of K, and P is the set of all sites x_j on all edges of E. Next, the sites x_j are advected according to the normalized gradient of ρ with a distance p_R. Formally,

$$x_j^{new} = x_j + p_R \frac{\nabla p}{||\nabla p||}, \tag{2}$$

CUBu, one of the KDE-based edge bundling methods, proposes a solution that a per-pixel site-density buffer $C : \mathbb{R}^2 \to \mathbb{N}$ is created to accumulate the sites that are rendered into C. $C(x)$ gives the number of sites of E that fall inside pixel x. Hence, for each pixel y, the density map ρ is computed as:

$$\rho(y) = \sum_{x \in T(y)} K(||x - y||)C(x), \tag{3}$$

where $T(y)$ is a disk of radius p_R centered at y. Equation 2 is applied to advect sites after ρ is computed. Every edge e_i is resampled by replacing x_j with x_j^{new} after every advection iteration. At the final step of the bundling process, a 1-D Laplacian smoothing is performed to remove the jitter effect of the resulting polylines. The complexity of KDE method is $O(p_I \cdot p_N \cdot p_S)$, where p_I, p_N, and p_S are the image resolution, the number of bundling iterations, and the counts of all sites, respectively.

We adopt the principle of KDE-based methods in our framework. Using KDE-based methods on web-based platforms has multiple advantages. First, optimizing the implementation of the mean shift principle on GPUs, KDE-based methods reveal several accuracy and scalability advantages [7]. Edges are sampled and accumulated into a buffer of pixels, and normalized gradients are thus generated to advect the sampled points. The computational complexity is independent of the problem size of the number of edges and vertices. Hence, it is suitable to bundle very large graphs. Second, it is feasible to parallelize KDE-based methods. The density map can be computed using OpenGL, WebGL, or other web-based rendering tools by encoding the sites' locations x_j with a 2D floating-point texture and accumulating the sites in a floating-point image buffer. Similarly, the calculation of normalized gradients can also be encoded with 2D floating-point textures and advect the sites in a ping-pong buffer manner. Third, the web-based rendering tool we use in our framework is WebGL. To our best knowledge, it supports all web browsers and thus ensures portability. Therefore, we choose to use the texture feature of WebGL to implement KDE-based methods to visualize large graphs on web-based platforms. Next, we will describe the parallelization of KDE-based methods on web platforms.

B. Parallelization

To enable interactive edge bundling on web-based platforms, GPU acceleration is required for the bundling process. However, the main challenge in performing parallel computing on web-based platforms is that the current web-based techniques lack direct GPU memory access support. We build our framework upon the method of TBEB [5]. The basic idea of TBEB is to leverage the texture feature and shader programs of WebGL to access and operate on GPU memory. For GPU memory read, data can be first uploaded from the main memory to the GPU memory via texture binding and then access texture data via texture lookup in vertex and fragment shader programs. To perform GPU memory writes, a Framebuffer Object (FBO) can be created and bound with a texture. This setup allows data to be written into the FBO by rendering it onto the texture that is bound to the FBO. The rendering function can be customized using programs of vertex and fragment shaders so that the data written to the FBO can be manipulated. It is less feasible in WebGL to simultaneously read and write one texture. To address this problem, the ping-pong buffering (or double buffering) method [25] is used to read the input and write the output through different textures.

The limitation of TBEB is that it requires non-uniform sampling, i.e., each edge has the same number of sampling points. It encodes each sample point as a 4-component pixel in a 2-D floating-point texture. It works well with some modest datasets with just up to a few thousand edges. It cannot address very large graphs since the subdivision sampling may make the sample points very large and exceed the maximal texture size. TBEB and FDEB typically require a large number of sampling points to generate a pleasing result [26].

To solve the scalability problem for large graphs, we exploit the idea of the KDE-based methods. As described in Section III-A, we use a uniform sampling method to sample edges, such that the number of sample points is significantly reduced. Then, the sample points can be accumulated on an image buffer, technically, an FBO. The subsequent operations, including kernel splatting, gradient computing, and position updating, can be conducted in a double-buffering manner. Leveraging the KDE-based methods and the parallel computing of WebGL together, we realize an effective and interactive edge bundling visualization on web-based platforms.

We name our new algorithm Pixel-based Edge Bundling (PBEB), as it primarily operates on pixels for the scalability

Algorithm 1 PBEB

1: // **Initialization**
2: $I \leftarrow$ the number of iteration steps
3: $D \leftarrow$ the decreasing factor
4: $R \leftarrow$ the kernel radius
5: $I_i \leftarrow I_0$ // initialize the iteration number
6: $E \leftarrow$ the edges of the graph
7: $\sigma \leftarrow$ the sampling step
8: $\{x\} \leftarrow Function(E, \sigma)$ // sample E based on σ
9: $T_I \leftarrow$ indexing texture
10: $T_O \leftarrow$ indicator texture
11: $T_{in} \leftarrow$ input position texture
12: $T_{out} \leftarrow$ output position texture
13: Build location array L and indexing array T_I based on $\{x\}$ and E
14: $T_{in} \leftarrow T_I$ // initialize T_{in} with T_I
15: **while** $I_i < I$ **do**
16: // **Step 1: Compute histogram**
17: $T_H \leftarrow$ histogram texture
18: $F_H \leftarrow$ histogram FBO
19: Bind F_H with T_H // bind texture with FBO
20: Render $\{x\}$ into F_H // accumulate pixel
21: Synchronization
22: // **Step 2: Compute gradients**
23: $T_G \leftarrow$ gradient texture
24: $F_G \leftarrow$ gradient FBO
25: Bind F_G with T_G // bind texture with FBO
26: /* render F_G through T_G with T_H and R to compute the gradient */
27: PARALLELGRADIENT(T_H, T_G, R)
28: Synchronization
29: // **Step 3: Update site positions**
30: $F_P \leftarrow$ position FBO
31: Bind F_P with T_{out} // bind texture with FBO
32: /* render F_P through T_{out} with T_{in}, T_O and T_G to update position*/
33: PARALLELUPDATE(T_O, T_G, T_{in}, T_{out})
34: Synchronization
35: // **Step 4: Resample**
36: $F_R \leftarrow$ resampling FBO
37: Bind F_R with T_{in} // bind texture with FBO
38: /* render F_R through T_{in} with T_{out} to resample sites */
39: PARALLELRESAMPLE(T_{in}, T_{out})
40: Synchronization
41: // **Step 5: Smooth**
42: $F_S \leftarrow$ smoothing FBO
43: Bind F_S with T_{out} // bind texture with FBO
44: /* render F_S through T_{out} with T_{in} to smooth edges */
45: PARALLELSMOOTH(T_O, T_{in}, T_{out})
46: Synchronization
47: Swap T_{in} and T_{out} // swap texture for next iteration
48: $I_i \leftarrow I_i + 1$
49: $R \leftarrow R \times D$
50: **end while**

Algorithm 2 PARALLELGRADIENT($T_H : input\_texture$; $T_G : output\_fbo$; $R : kernel\_radius$)

1: **for** each pixel p_{ij} of T_G **in parallel do**
2: **for** each pixel p_θ inside the disk centered at p_{ij} **do**
3: // apply $Gaussian$ splatting with radius R
4: Use texture lookup from T_H, apply weighted function W, and copy the sum of weighted values to p_{ij}
5: $p_{ij} \leftarrow W(p_{ij}) + W(p_\theta)$.
6: **end for**
7: Render p_{ij} into T_G.
8: **end for**

Algorithm 3 PARALLELUPDATE($T_O : input\_texture$; $T_G : input\_texture$; $T_{in} : input\_texture$; $T_{out} : output\_fbo$)

1: $G_{p_{ij}}$ // the advection vector of T_G in the position of p_{ij}.
2: **for** each pixel p_{ij} of T_{out} **in parallel do**
3: Fetch the corresponding sites in T_{in} using texture lookup and learn the position of the advection vector in T_G.
4: Fetch the corresponding sites in T_G using texture lookup and use them to copy the value of the advection vector to $G_{p_{ij}}$.
5: **if** p_{ij} **is a site** according to T_O **then**
6: Move p_{ij} based on $G_{p_{ij}}$.
7: **end if**
8: Render p_{ij} into T_{out}.
9: **end for**

of edge bundling. Next, we describe our implementation and technical details. Algorithm 1 lays out the pipeline of PBEB.

Lines 1 to 14 of Algorithm 1 correspond to the initialization of PBEB. Each edge e_i of an input graph G is sampled with a sample step σ. The sample points, namely sites x_i, are stored in a texture T_I. We refer to this texture as an *indexing texture*. It stores the 2D coordinates of x_i by encoding the coordinates of the sites with 4-component RGBA pixel values. In particular, we intentionally subdivide each edge into an odd number of segments so that a 4-component RGBA pixel value stores a segment with 4 coordinate values (one segment consists of 2 points with a 2D coordinate value for each point), which facilitates a relatively simple encoding. Hence, the width of T_I can be defined:

$$I_w = \begin{cases} \dfrac{p_S}{2} & \text{if } \frac{p_S}{2} < max, \\ max & \text{otherwise,} \end{cases} \tag{4}$$

where max is a predefined parameter to control the width of an allocated texture to be not larger than the maximum texture width allowed by WebGL systems. Recall that p_S is the count of all sites. The height I_h of the texture T_I is then defined as:

$$I_h = \mathbf{ceil}(\frac{\frac{p_S}{2}}{max}), \tag{5}$$

Additionally, we need to create a texture T_O that records a point in T_I is either an endpoint or a site since endpoints of

Algorithm 4 PARALLELRESAMPLE(T_{in} : $input\_texture$; T_{out} : $output\_fbo$)

1: $|\cdot| \leftarrow$ euclidean distance
2: $H \leftarrow$ distance threshold
3: $pre_n \leftarrow$ number of previous adjacent sites that are close to the current site p_{ij}
4: $next_n \leftarrow$ number of next adjacent sites that are close to the current site p_{ij}
5: $pre_{p_{ij}} \leftarrow$ the previous adjacent site of the current site p_{ij}
6: $next_{p_{ij}} \leftarrow$ the next adjacent site of the current site p_{ij}
7: **for** each pixel p_{ij} of T_{in} **in parallel do**
8: **while** $|pre_{p_{ij}} - p_{ij}| < H$ **do**
9: $pre_{p_{ij}} =$ the previous sites of $pre_{p_{ij}}$
10: pre_n++;
11: **end while**
12: **while** $|next_{p_{ij}} - p_{ij}| < H$ **do**
13: $next_{p_{ij}} =$ the next sites of $pre_{p_{ij}}$
14: $next_n$++;
15: **end while**
16: **if** $pre_n <= next_n$ **then**
17: Move p_{ij} a small distance $\frac{|pre_{p_{ij}} - p_{ij}|}{pre_n+1}$ towards $pre_{p_{ij}}$.
18: **else**
19: Move p_{ij} a small distance $\frac{|next_{p_{ij}} - p_{ij}|}{next_n+1}$ towards $next_{p_{ij}}$.
20: **end if**
21: Render p_{ij} into T_{in}.
22: **end for**

Algorithm 5 PARALLELSMOOTH(T_O : $input\_texture$; T_{in} : $input\_texture$; T_{out} : $output\_fbo$)

1: $\{pre_{p_{ij}}\} \leftarrow$ the previous adjacent sites of p_{ij} ($\{pre_{p_{ij}}\}$ and p_{ij} are in the same edge)
2: $\{next_{p_{ij}}\} \leftarrow$ the next adjacent sites of p_{ij} ($\{next_{p_{ij}}\}$ and p_{ij} are in the same edge)
3: **for** each pixel p_{ij} of T_{out} **in parallel do**
4: Fetch the previous sites $\{pre_{p_{ij}}\}$ of p_{ij} from T_{in}.
5: Fetch the next sites $\{next_{p_{ij}}\}$ of p_{ij} from T_{in}.
6: Convolve the position of p_{ij} with $\{pre_{p_{ij}}\}$ and $\{next_{p_{ij}}\}$.
7: Render p_{ij} into T_{out}.
8: **end for**

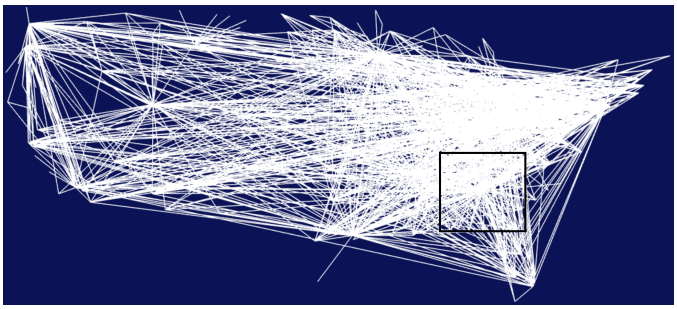


Fig. 1. Visualizing a US Airlines dataset (2100 edges) using a node-link diagram that can easily incur visual clutter.

edges and sites may overlap in some pixels.

Next, PBEB builds edge bundling by iteratively conducting kernel density estimation through the steps of histogram computing, gradient computing, site position updating, edge resampling, and edge smoothing. The algorithm stops when it reaches the predefined number of iteration steps.

First, in the step of histogram computing (Lines 17 to 21 of Algorithm 1), similar to CUBu [10], we accumulate sites on an image buffer. In our implementation, we render all the sites of the graph into an FBO F_H. The width and height of F_H are set to be the resolution of the display. F_H is bound with a 2D floating-point texture T_H. After rendering, T_H gives a histogram, where the number of sites falling into every pixel is recorded. The pixel's value in the histogram is encoded as either a R, G, or B value in T_H. Here, we restrict the rendering point size to be exactly 1 pixel, i.e., one site can only be rendered in one pixel in T_H. To accumulate the number of rendering sites in T_H, the source and destination factors of the blending function must be set to be *GL_SRC_COLOR*. The width and height of F_H and T_H are the same as the width and height of the display resolution.

Second, after the histogram is built, we can compute the normalized gradients by splatting a kernel function (Lines 23 to 28 of Algorithm 1). To do that in WebGL, we need to bind an FBO F_G with a 2D floating-point texture T_G. The 2D gradient values are encoded into the color components of the texture T_G. The kernel splatting is conducted in a customized fragment shader program. T_H is also passed to the GPU memory during the rendering process. In the fragment shader program, kernel splatting can be easily done by applying a *Gaussian* weighted function W. Then, a gradient filter is used to compute the normalized gradient for each pixel in T_G. After rendering, every color component of T_G contains an advection vector for later position updating. The width and height of F_G and T_G are identical to the width and height of the display resolution. Algorithm 2 shows the PARALLELGRADIENT fragment shader.

Third, we update the positions of the sites (Lines 30 to 34 of Algorithm 1). Recall that we have created an indexing texture T_I and T_O. We can update the position of each site of T_I in a customized fragment shader program with T_I, T_O, and T_G as inputs. We first make $T_{in} = T_I$. We need another texture buffer T_{out} to be bound with an FBO F_P. We then pass T_{in}, T_O, and T_G to the GPU memory and render the new positions as a color component into T_{out}. Note the sizes of the aforementioned textures T_H and T_G are identical to the display resolution, while the width and the height of T_{in} and T_{out} are I_w and I_h, respectively. The key to this step is the translation between the texture coordinates and the indexes of sites (pixels). When we encode a data set into a texture, a data point is accessed through a 2D texture coordinates (x, y), where the x or y component of the texture coordinate is typically between 0.0 and 1.0. If we encode a 2D $n \times t$ matrix into a 2D texture, the transformation between an entry at the ith column and the jth row of the matrix and its 2D texture coordinates (x, y) can be

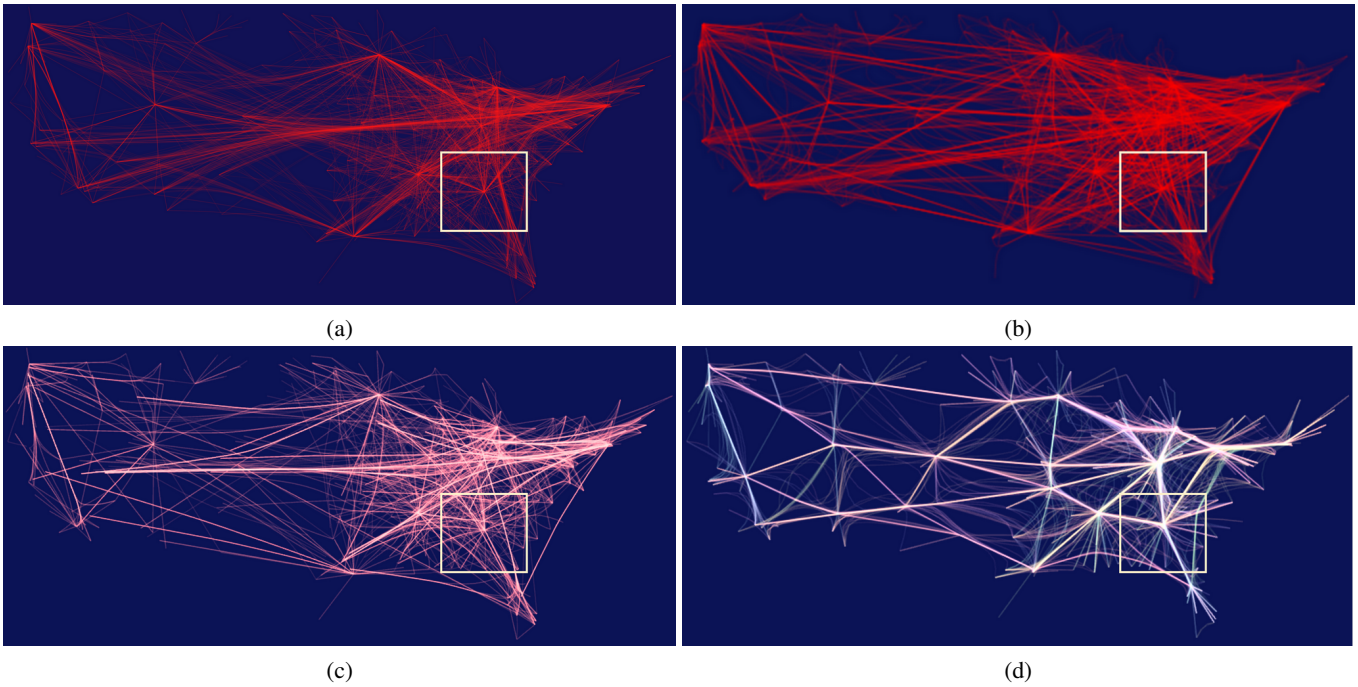


Fig. 2. The visualization results of a US Airlines dataset (2100 edges) using (a) a web-based implementation of FDEB [3], (b) a web-based implementation of MINGLE [4], (c) TBEB [5], and (d) our PBEB.

computed by

$$x = i/(n-1), y = j/(t-1). \tag{6}$$

Accordingly, a 2D texture coordinate (x, y) is corresponding to the entry (i, j):

$$i = \mathbf{ceil}(x \times n), j = \mathbf{ceil}(y \times t). \tag{7}$$

We use this transformation to read the points from the textures. In this step, the rendering resolution is set to be $I_w \times I_h$. In the fragment shader program, each pixel contains the position of one segment (4-component RGBA value stores two 2D sites). From the color component R and G or B and A in T_{in}, we can gain the index of the gradient value in T_G. Using Equations 6 and 7 to translate the corresponding coordinate in T_G, we can retrieve the gradient value for this site. We ignore the update of a point if its record in T_O indicates it is an endpoint. Algorithm 3 shows the PARALLELUPDATE fragment shader. After rendering, T_{out} contains the updated position of all sites and the original endpoints.

KDE-based methods can generate lattice artifacts. Hence, resampling (Lines 36 to 40 of Algorithm 1) and smoothing (Lines 42 to 47 of Algorithm 1) are applied to increase the readability of the final drawing. The resampling and smoothing can also be parallelized. Algorithm 4 shows the PARALLELRESAMPLE fragment shader. T_{out} has been generated from the position updating step. We need to create an FBO F_R, and bind it with T_{in}. In this step, T_{out} is the input texture, and T_{in} is the output texture. In the resampling fragment shader program, we first define a distance threshold H. For every site p_{ij}, search forward along the edge to find the previous site $pre_{p_{ij}}$ whose distance to p_{ij} is larger or equal to H. Count the site number pre_n between p_{ij} and $pre_{p_{ij}}$. So on and so forth for the backward search to find $next_{p_{ij}}$ and $next_n$. We then compare pre_n and $next_n$. If pre_n is smaller, we move p_{ij} towards pre_n with a distance of $\frac{|pre_{p_{ij}} - p_{ij}|}{pre_n+1}$, and vice versa.

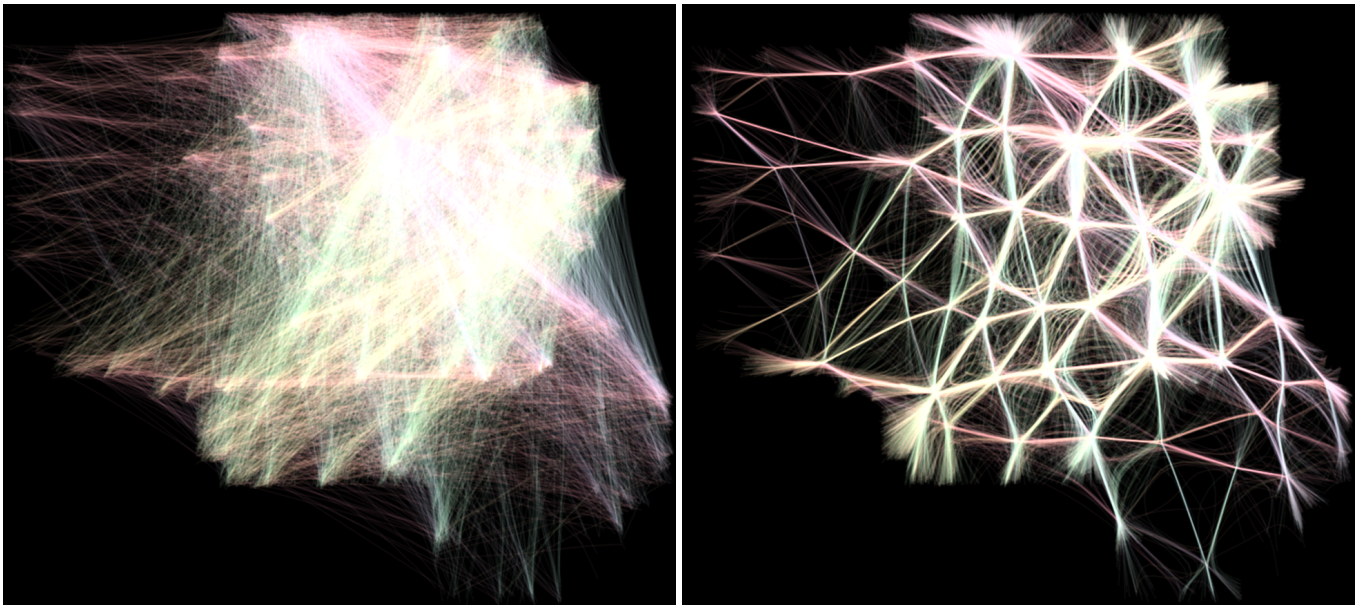
The smoothing step conducts a 1-D Laplacian smoothing on each edge, i.e., convolves the position of a site with its adjacent sites. Algorithm 5 shows the PARALLELSMOOTH fragment shader. After writing the new positions to T_{out}, we swap T_{in} and T_{out} for the next iteration. After the iterations, we bind the output texture T_{out} into a Vertex Buffer Object (VBO). Then we render the segments as polylines to the display. Because the output texture and the VBO are all located in the GPU memory, we can directly conduct rendering in GPU and avoid the costly data transferring between CPU and GPU.

IV. RESULTS

A. Visualization

We compare PBEB with other web-based methods on their visualization results. Figure 1 shows a US Airlines graph using direct visualization of a node-link diagram. As shown in Figure 1, the node-link diagram can easily incur visual clutter, and it is relatively difficult to perceive the main airline patterns from the graph.

Figure 2 shows the visualization results of the same graph using the edge bundling methods, including a web-based implementation of FDEB [3], a web-based implementation of MINGLE [4], TBEB [5], and our PBEB. We can see that, in general, the edge bundling results can more clearly reveal relational patterns by grouping edges according to their



(a) (b)

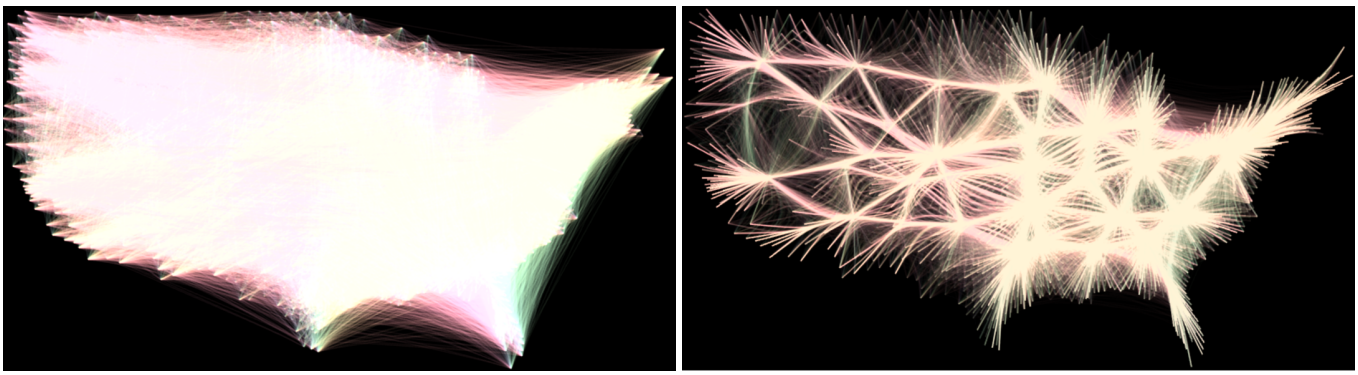
Fig. 3. The visualization results of a France Airlines graph (17273 edges) using (a) node-link diagram and (b) our PBEB, where the PBEB result can effectively alleviate the visual clutter observed in the node-link diagram, providing a clearer depiction of the main structure of the France Airlines graph.

similarities. For example, the highlighted area in Figure 1 is visually occluded. In the edge bundling visualization results shown in Figure 2, we can clearly perceive a "hub" in the center of the highlighted area. Geographically, this hub is Atlanta. From the visualization results of edge bundling methods, we can learn that Atlanta is an airline hub that is used by multiple airlines to concentrate passenger traffic and flight operations.

Moreover, the quality of TBEB (Figure 2 (c)) and PBEB (Figure 2 (d)) are arguably better than FDEB and MINGLE in terms of revealing subtle graph structure and details. From Figure 2 (a), (c), and (d), we can see a major pattern corresponding to the airline routes between the west coast of the US and the Northeastern US. Specifically, one side of the airline routes connects the Northeastern US, and then the routes split into two groups. One connects to the Northwestern US (mainly Washington), and the other connects to the Southwestern US (mainly California). There is a very subtle detail in this pattern. From the results of FDEB and MINGLE, we might misunderstand that the airline routes directly connect Northwestern and Southwestern with Northeastern. TBEB (Figure 2 (c)) and PBEB (Figure 2 (d)) show that this is not the case. Some airlines first route to another airline hub, which is geographically the Indianapolis. This subtle pattern cannot be perceived in Figure 2 (a) and (b). We can see that the results generated by FDEB and MINGLE have significantly reduced the visual clutter compared to the original node-link diagram. However, some dense areas still have visual clutter. That may mislead users in some subtle details. TBEB and PBEB reveal high-level relational patterns and preserve more subtle graph structures, avoiding the problem of FDEB and MINGLE.

The complexity of PBEB is independent of the number of edges and vertices as its complexity is $O(p_I \cdot p_N \cdot p_S)$, where p_I, p_N, and p_S are the image resolution, the number of bundling iterations and the counts of all sites, respectively. However, the model of FDEB and TBEB cannot enable an edge bundling visualization with a very large graph because of the texture size limitation. PBEB accumulates sample points on pixels such that a density histogram is generated. Gradient calculation and advection can be conducted based on the density histogram. Therefore, PBEB can avoid the issue of FDEB and TBEB and improve the visualization of large graphs on a web browser. Figure 3 shows the visualization results of a France Airlines graph (17273 edges) using the node-link diagram and our PBEB methods. In Figure 3(a), we can see a significant degree of complex node and edge overlap and intertwine, resulting in a visual clutter that makes it hard to comprehend the graph. As shown in Figure 3(b), PBEB can effectively bundle edges based on their similarities, leading to a substantial reduction in visual clutter and a clear depiction of the core structure of the France Airlines graph.

Next, we show a visualization using PBEB on a large US migration graph with approximately half of a million edges (545881 edges). To the best of our knowledge, PBEB is the first web-based edge bundling method to handle such a large graph on web browsers. Figure 4 shows the result of the node-link diagram and PBEB methods using a Google Chrome browser. In Figure 4 (a), a considerable number of edges incur severe visual clutter. By using PBEB, the high-level patterns of migration can be perceived. Since PBEB uses the shader rendering method [10], the brighter area means the denser place where the edges converge. We can observe north-south and east-



(a) (b)

Fig. 4. The visualization results of a large US migration graph (545881 edges) using (a) node-link diagram and (b) our PBEB, where the PBEB result can effectively alleviate the visual clutter observed in the node-link diagram, providing a clearer depiction of the main structure of the large US migration graph.

TABLE I
PERFORMANCE COMPARISON USING THE US AIRLINES, FRANCE AIRLINES, AND US MIGRATIONS GRAPHS ON A DESKTOP.

Graph	**FDEB (CPU)**		**FDEB (CUDA)**		**FDEB ($\mathbb{D}^3$)**		**TBEB**		**MINGLE (JavaScript)**		**FFTEB (CUDA)**		**PBEB**	
	Sites	*Time(ms)*	*Sites*	*Time(ms)*	*Sites*	*Time(ms)*	*Sites*	*Time(ms)*	*Sites*	*Time(ms)*	*Sites*	*Time(ms)*	*Sites*	*Time(ms)*
US Airlines	270K	900	270K	105	270K	1.08K	270K	15	270K	169.75	105K	19	48.8K	5
France Airlines	2.2M	2.1M	2.2M	926	2.2M	n/a	2.2M	n/a	2.2M	1.04K	864K	380	557K	20
Large US migrations	70M	n/a	70M	n/a	70M	n/a	70M	n/a	70M	93.6K	6.4M	5.2K	2.4M	103

west migration patterns and learn that north-south migrations mainly happened in the Mideast US, while the pattern is not salient in the Western US. This observation demonstrates that PBEB is effective in visualizing large graphs using web-based techniques, whereas the original node-link diagram and other edge bundling implementations cannot achieve this on web-based platforms.

B. Performance Evaluation

We evaluate the performance of our PBEB by comparing the following methods:

- FDEB using a CPU implementation [23],
- FDEB using a CUDA implementation [27],
- FDEB using a D3 implementation [3],
- TBEB using a WebGL implementation [5],
- MINGLE using a JavaScript implementation [4],
- FFTEB using a CUDA implementation [11],
- our PBEB using a WebGL implementation.

We use the following device in the experiment:

- a desktop with an 8X Intel Core i7 3.60GHz CPU and an NVIDIA GeForce GTX 1080 ti GPU,
- a Nexus 9 tablet with a dual-core Denver 2.3GHz CPU and a Kepler DX1 GPU.

We use a Google Chrome web browser to generate the results for the web-based edge bundling applications. We use the following three typical graph datasets in the experiment:

- a US Airlines dataset with 2100 edges and 235 vertices,
- a France Airlines dataset with 17273 edges and 34194 vertices,
- a US migrations dataset with 545881 edges and 3075 vertices.

Table I shows the performance comparison on the desktop, where Sites are the number of sample points generated in the sampling step, and Time is the average elapsed time that one iteration takes for similarity (or compatibility) calculation, bundling, and rendering. For FDEB, TBEB, and MINGLE, each edge has the same number of sites in every iteration. For FFTEB and PBEB, uniform sampling is used. We use a sample step of 5% of the display size to sample each edge in the implementation of PBEB. For FFTEB, we use the uniform sampling strategy of CUBu [10]. Hence, the number of sites of different methods may be different. The timing result is formally calculated as:

$$Time = \frac{p_T}{p_N}, \tag{8}$$

where p_T is the total elapsed time of an edge bundling application to generate the final graph drawing, and p_N is the number of bundling iterations. For some existing implementations that only render once in the final result, we scale their rendering time according to the number of iterations to avoid bias. As shown in Table I, the performance of TBEB is significantly better than the CPU- and JavaScript-based methods. PBEB outperforms other methods on all datasets and achieves interactive framerates on the datasets of US Airlines and France Airlines. For the large US migrations dataset, the PBEB method achieves approximately $50\times$ speedups to the fastest state-of-the-art method, FFTEB. More importantly,

TABLE II
PERFORMANCE COMPARISON USING THE US AIRLINES AND FRANCE AIRLINES GRAPHS ON A TABLET.

Graph	FDEB ($\mathbb{D}^3$)		TBEB		MINGLE (JavaScript)		PBEB	
	Sites	*Time(ms)*	*Sites*	*Time(ms)*	*Sites*	*Time(ms)*	*Sites*	*Time(ms)*
US Airlines	270K	3423	270K	49	270K	664	48.8K	30
France airlines	2.2M	n/a	2.2M	n/a	2.2M	2653	557K	60

PBEB can be deployed on web-based platforms, which are more portable.

Next, we compare PBEB with other web-based methods, namely FDEB using D3 [3], MINGLE using JavaScript [4], and TBEB [5]. This evaluation takes place within a Chrome browser on the Nexus 9 tablet, as all these methods are designed for web browsers and are compatible with tablet devices. We use the datasets of US Airlines and France Airlines. In our experiment, we find that all the methods cannot run the Large US migrations and skip the comparison on this dataset. Table II shows that PBEB is more significantly efficient than other web-based edge bundling methods. In the case of the US Airlines graph, PBEB demonstrates approximately a $114\times$ speedup compared to FDEB and a $22\times$ speedup compared to MINGLE. PBEB achieves a marginal performance gain, i.e., about a $1.6\times$ speedup, compared to TBEB for US Airlines. However, when visualizing the France Airlines graph, it exceeds the practical scale limits for both TBEB and FDEB. Only MINGLE and our PBEB remain capable of visualizing this graph, and our PBEB attends can still visualize it, and our PBEB achieves approximately a $44\times$ speedup compared to MINGLE.

The findings presented in Tables I and II showcase the adaptability and performance of our framework. PBEB operates effectively not only on high-end machines equipped with dedicated graphics cards but also on ubiquitous smart devices, even those with comparatively constrained graph processing capabilities. The results indicate that our PBEB surpasses existing methods in terms of both performance efficiency and the capacity to handle larger graphs.

V. CONCLUSION

We have introduced PBEB, a real-time edge bundling framework designed for the effective visualization of large graphs on web-based platforms. PBEB has leveraged the concept of kernel density estimation to address visual clutter by effectively bundling the massive edges of a large graph, thereby revealing the overall graph structures and intricate details of the graph in visualizations. Our framework harnesses the power of parallel processing through WebGL's texture capabilities and shader programming, ensuring interactive edge bundling visualization on both desktop computers and mobile devices. Notably, when compared to existing web-based edge bundling methods and applications, our solution significantly outperforms them, achieving several-fold speed improvements when handling graphs ranging from thousands to hundreds of thousands of edges. Our approach facilitates the interactive exploration of large graphs on mobile devices and significantly enhances the interactivity and usability of edge bundling across diverse platforms.

In the future, we aim to incorporate attribute-based edge bundling visualization in our framework, allowing it to convey more precise attributes and topological information of underlying graphs on web platforms. We will explore methods (e.g., [28]) for assessing the quality and precision of visual representations achieved through edge bundling. Additionally, we plan to explore the integration of various graph layouts with our work, broadening the range of available graph visualization styles. We would also like to extend the utilization of the texture feature and acceleration strategies to more visualization applications, such as interactive force-directed graph layouts and interactive scatterplots, providing a GPU-based unified framework for visualization applications on web platforms. Last but not least, we recognize that visualizing more extensive graphs, such as those containing billions or more edges, on a single device can pose a significant challenge that is expected to attract more attention as graph sizes continue to experience rapid growth.

ACKNOWLEDGMENT

This research has been sponsored in part by the National Science Foundation grant IIS-1652846.

REFERENCES

[1] N. Vesselinova, R. Steinert, D. F. Perez-Ramirez, and M. Boman, "Learning combinatorial optimization on graphs: A survey with applications to networking," *IEEE Access*, vol. 8, pp. 120 388–120 416, 2020.

[2] M. Bostock, V. Ogievetsky, and J. Heer, "D^3 data-driven documents," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2301–2309, 2011.

[3] "d3.ForceBundle," https://github.com/upphiminn/d3.ForceBundle.

[4] "MingleBundle," https://github.com/philogb/mingle.

[5] J. Wu, L. Yu, and H. Yu, "Texture-based edge bundling: A web-based approach for interactively visualizing large graphs," in *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 2015, pp. 2501–2508.

[6] N. Bikakis, J. Liagouris, M. Krommyda, G. Papastefanatos, and T. Sellis, "GraphVizdb: A scalable platform for interactive large graph visualization," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 2016, pp. 1342–1345.

[7] A. Lhuillier, C. Hurter, and A. Telea, "State of the art in edge and trail bundling techniques," *Computer Graphics Forum*, vol. 36, no. 3, pp. 619–645, 2017.

[8] C. Hurter, O. Ersoy, and A. Telea, "Graph bundling by kernel density estimation," *Computer Graphics Forum*, vol. 31, no. 3pt1, pp. 865–874, 2012.

[9] V. Peysakhovich, C. Hurter, and A. Telea, "Attribute-driven edge bundling for general graphs with applications in trail analysis," in *2015 IEEE Pacific visualization symposium (PacificVis)*. IEEE, 2015, pp. 39–46.

[10] M. van der Zwan, V. Codreanu, and A. Telea, "CUBu: universal real-time bundling for large graphs," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 12, pp. 2550–2563, 2016.

[11] A. Lhuillier, C. Hurter, and A. Telea, "FFTEB: Edge bundling of huge graphs by the fast fourier transform," in *Pacific Visualization Symposium (PacificVis), 2017 IEEE*. IEEE, 2017, pp. 190–199.

[12] S. Liu, W. Cui, Y. Wu, and M. Liu, "A survey on information visualization: recent advances and challenges," *The Visual Computer*, vol. 30, no. 12, pp. 1373–1393, Dec 2014. [Online]. Available: https://doi.org/10.1007/s00371-013-0892-3

[13] T. Von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J. J. van Wijk, J.-D. Fekete, and D. W. Fellner, "Visual analysis of large graphs: State-of-the-art and future research challenges," *Computer Graphics Forum*, vol. 30, no. 6, pp. 1719–1749, 2011.

[14] F. Beck, M. Burch, S. Diehl, and D. Weiskopf, "The state of the art in visualizing dynamic graphs." *EuroVis (STARs)*, 2014.

[15] I. Herman, G. Melançon, and M. S. Marshall, "Graph visualization and navigation in information visualization: A survey," *IEEE Transactions on Visualization and Computer Graphics*, vol. 6, no. 1, pp. 24–43, 2000.

[16] C. Vehlow, F. Beck, and D. Weiskopf, "The state of the art in visualizing group structures in graphs," in *EuroVis - STARs*, 2015.

[17] M. Bostock and J. Heer, "Protovis: A graphical toolkit for visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1121–1128, 2009.

[18] E. R. Gansner, Y. Hu, S. North, and C. Scheidegger, "Multilevel agglomerative edge bundling for visualizing large graphs," in *2011 IEEE Pacific Visualization Symposium*. IEEE, 2011, pp. 187–194.

[19] A. Telea and O. Ersoy, "Image-based edge bundles: Simplified visualization of large graphs," *Computer Graphics Forum*, vol. 29, no. 3, pp. 843–852, 2010.

[20] J. Böttger, A. Schäfer, G. Lohmann, A. Villringer, and D. S. Margulies, "Three-dimensional mean-shift edge bundling for the visualization of functional connectivity in the brain," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 3, pp. 471–480, 2014.

[21] J. Wu, J. Zeng, F. Zhu, and H. Yu, "MLSEB: Edge bundling using moving least squares approximation," in *Graph Drawing and Network Visualization: 25th International Symposium, GD 2017, Boston, MA, USA, September 25-27, 2017*. Springer, 2018, pp. 379–393.

[22] D. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 741–748, 2006.

[23] D. Holten and J. J. Van Wijk, "Force-directed edge bundling for graph visualization," *Computer Graphics Forum*, vol. 28, no. 3, pp. 983–990, 2009.

[24] D. Comaniciu and P. Meer, "Mean shift: A robust approach toward feature space analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 5, pp. 603–619, 2002.

[25] M. Pharr, R. Fernando, and T. Sweeney, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[26] Q. H. Nguyen, P. Eades, and S.-H. Hong, "Towards faithful graph visualizations," *arXiv preprint arXiv:1701.00921*, 2017.

[27] D. Zhu, K. Wu, D. Guo, and Y. Chen, "Parallelized force-directed edge bundling on the GPU," in *2012 11th International Symposium on Distributed Computing and Applications to Business, Engineering & Science*. IEEE, 2012, pp. 52–56.

[28] J. Wu, F. Zhu, X. Liu, and H. Yu, "An information-theoretic framework for evaluating edge bundling visualization," *Entropy*, vol. 20, no. 9, p. 625, 2018.