Texture-Based Edge Bundling: A Web-Based Approach for Interactively Visualizing Large Graphs

Jieting Wu
Computer Science and Engineering
University of Nebraska-Lincoln
Email: jwu@cse.unl.edu

Lina Yu
Computer Science and Engineering
University of Nebraska-Lincoln
Email: lina@cse.unl.edu

Hongfeng Yu
Computer Science and Engineering
University of Nebraska-Lincoln
Email: yu@cse.unl.edu

***Abstract*—Directly visualizing a large graph as a node-link diagram often incurs visual clutter. Edge bundling can effectively address this issue and concisely reveal the main graph structure with reduced visual clutter. Although researchers have devoted noticeable efforts to develop acceleration methods, it remains a challenging task to efficiently conduct edge bundling on devices with a limited computing capacity, such as ubiquitous smart mobile devices. We present a new method for visualizing a node-link diagram based on force-directed edge bundling. We use textures to encode the data of lines and forces, and employ shaders to conduct the iterative line refinement on GPUs. We name this method as Texture-Based Edge Bundling (TBEB) as the major steps are done using textures. We demonstrate the high performance of TBEB using standard graphics cards. TBEB makes it feasible to interactively visualize large graphs on web-based platforms.**

***Keywords*-graph visualization; edge bundling; network visualization; mobile devices; GPU computing.**

I. Introduction

Graphs are widely used to model relationships between data entities in diverse scientific and engineering applications, such as systems biology, software engineering, social science, and so on. Visualization is an effective means to reveal salient graph structures and patterns, as well as similar communities and subtle outliers. Among various graph visual representations, node-link diagram is one of the most prevailing techniques because of its simple and intuitive structure [6], [12]. However, a node-link diagram of dense graph often suffers from visual clutter, as its visual readability can be quickly degraded with an increasing number of nodes and edges.

To address visual clutter of node-link diagram, a new concept of *edge bundling* has been proposed, and has been applied to hierarchical graphs [7] and general graphs [8]. These algorithms can concisely reveal the main structure of a graph by grouping related edges together into a set of smooth curved bundles. However, their execution typically involves a considerable computing cost to achieve a visually appealing result for a large graph. Although researchers have experimented with acceleration methods [14], these solutions mostly reply on features dedicated to special graphics hardware or libraries. Therefore, it remains a challenging task to interactively construct and visualize edge bundles of large graphs on devices with a limited computing capacity, in particular, on ubiquitous smart mobile devices.

We present a new method for constructing and visualizing edge bundles of large graphs using standard functions of graphics cards. We employ the computation model of force-directed edge bundling [8], encode the data of lines and forces into standard texture images, and conduct interactive line refinement using vertex and fragment shaders that are the standard functions of graphics systems. As the major steps are conducted through the usage of textures, we name this method as Texture-Based Edge Bundling (TBEB).

The method uses the standard graphics functions and libraries, and can be nearly universally applied on devices with standard graphics processing units. In addition, we present a web-based implementation that can be easily migrated across devices and platforms without any significant overhead. We demonstrate the effectiveness of our method using multiple data sets on devices with standard graphics cards. We achieve interactive frame rates to conduct and render the edge bundles of a graph with thousands of edges in a web browser. Overall, our TBEB method is easy to understand and implement.

II. Related Work

We refer interested reader to the surveys conducted by Herman et al. [6] and Vehlow et al. [12] for an overview of graph visualization, and to the surveys conducted by Beck et al. [1] and Landesberger et al. [13] in specific techniques for dynamic graphs and large graphs. This paper focuses on node-link diagrams.

Although the node-link representation is one of the most popular ways to visualize graphs, visual clutter can be easily generated if we render a large number of edges as straight lines. To remedy this problem, Holen [7] presented a new technique, named *hierarchical edge bundles*, that constructs smooth curved bundles to represent a set of edges at the relevant levels of a hierarchical graph. This technique can effectively reveal the high-level structure of a graph, while significantly reduce visual clutter. This method has been

further evolved to force directed edge bundling (FDEB) for visualizing a general graph without hierarchy [8].

Many researchers have made efforts to generalize and improve edge bundling. Cui et al. [2] introduced a geometry-based edge clustering method. They first generated a control mesh at different levels of detail with respect to underlying graph patterns, and then used this mesh to guide edge-clustering and achieve different visualization results. Telea et al. [11] used an image-based technique that leverages distance-based splatting and shape skeletonization to render a hierarchical edge clustering of a graph. This method has been further extended to a skeleton-based edge bundling method that combines edges clustering, distance fields, and 2D skeletonization to construct edge bundles [4]. Based on fast agglomerative clustering techniques, Gansner et al. [5] adopted a guiding principle to save ink for drawing edges and enhancing the effectiveness of edge bundling. Selassie et al. [10] proposed divided edge bundling to improve bundling results by considering graph topology.

Although these edge bundling based approaches can achieve visually appealing results, they are characterized with high algorithmic complexities. The execution of such an algorithm may take several to hundreds of seconds to compute bundles for large graphs. This performance lag hinders the usage of these methods on interactive applications. A few efforts have been perceived to accelerate the construction of edge bundles. Ersoy et al. [4] used GPUs to accelerate their skeleton-based edge bundling. Zhu et al. [14] presented a parallelized FDEB on GPUs, and achieved a 11× speedup compared to the original FDEB. However, these methods rely on NVIDIA's CUDA architecture, which cannot be easily adopted to general graphics cards.

III. Background

In this paper, we aim to accelerate force direct edge bundling (FDEB) [8] using standard functions of general graphics cards, and make edge bundling deployable and compatible on web-based platforms.

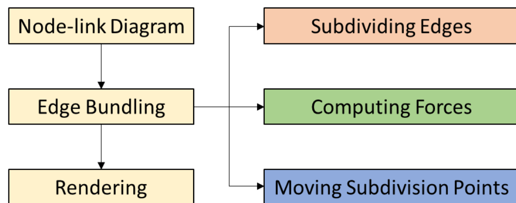


Figure 1. Edge bundling based graph visualization.

Figure 1 shows a general framework of graph visualization using edge bundling. Given an input node-link diagram, we first use edge bundling to group the edges and build a set of smooth curved bundles, and then send the generated geometry data of bundles to GPUs for rendering. This framework typically suffers from two main performance bottlenecks for large graphs: First, edge bundling requires considerable computing power for a large amount of edges. Second, if we conduct edge bundling using a CPU, transferring the data of bundles from the main memory to the GPU memory is costly for a large graph. For an interactive graph visualization, edge bundling and rendering need to be carried out frequently to response each user interaction. Thus, these two operations can become a severe problem in interactive applications of large graphs.

A. Force Directed Edge Bundling

We first revisit the computational model of FDEB. As shown in Figure 1, FDEB takes a node-link diagram with straight edges as its input, and uses an iterative simulation to refine the bundling. There are total C simulation cycles. In each cycle, FDEB first subdivides each edge and then iteratively moves each subdivision point to a new position by modeling and computing forces among the points.

In the first simulation cycle C_0, FDEB starts with P_0 subdivision points for each edge . For example, in Figure 2(a), each edge has two end points (in orange) and one subdivision point (in blue), and is subdivided into two segments (i.e., $P_0 = 1$ in this case). Then, two types of forces, the spring forces F_s and the electrostatic forces F_e, are modeled at a subdivision point p_{ij}, where p_{ij} is the jth point on an edge l_i, as shown in Figure 2(a). F_s is defined as:

$$F_s = k_p(\| p_{i(j-1)} - p_{ij} \| + \| p_{ij} - p_{i(j+1)} \|), \quad (1)$$

where k_p is a spring constant, and $p_{i(j-1)}$ and $p_{i(j+1)}$ are the neighboring points of p_{ij} on the edge l_i. F_e is defined as:

$$F_e = \sum_{m \in E} \frac{1}{\| p_{ij} - p_{mj} \|}, \quad (2)$$

where E is a set of edges where each edge l_m interacts with l_i. p_{mj} is the corresponding subdivision point on such an interacting edge l_m. Thus, the combined force $F_{p_{ij}}$ exerted on p_{ij} is:

$$F_{p_{ij}} = F_s + F_e. \quad (3)$$

The position of p_{ij} is updated by moving it a small distance in the direction of $F_{p_{ij}}$. This is an iterative process in that F_e, F_s, and $F_{p_{ij}}$ are also updated according to the new position of p_{ij}. A specific number of iteration steps I is conducted to move the subdivision points to reach an equilibrium between forces. Figure 2(b) shows the equilibrium state of C_0. The edge subdivision and the subdivision point movement are continued in the following simulation cycles. Figure 2(c) and (d) illustrate the process of C_1.

The number of iteration steps during the first cycle is I_0. We can easily see that the number of subdivision points is doubled to smoothen the edges after performing a cycle. Meanwhile, the number of iteration steps I is decreased by a factor R. The original paper of FDEB [8] reported that a

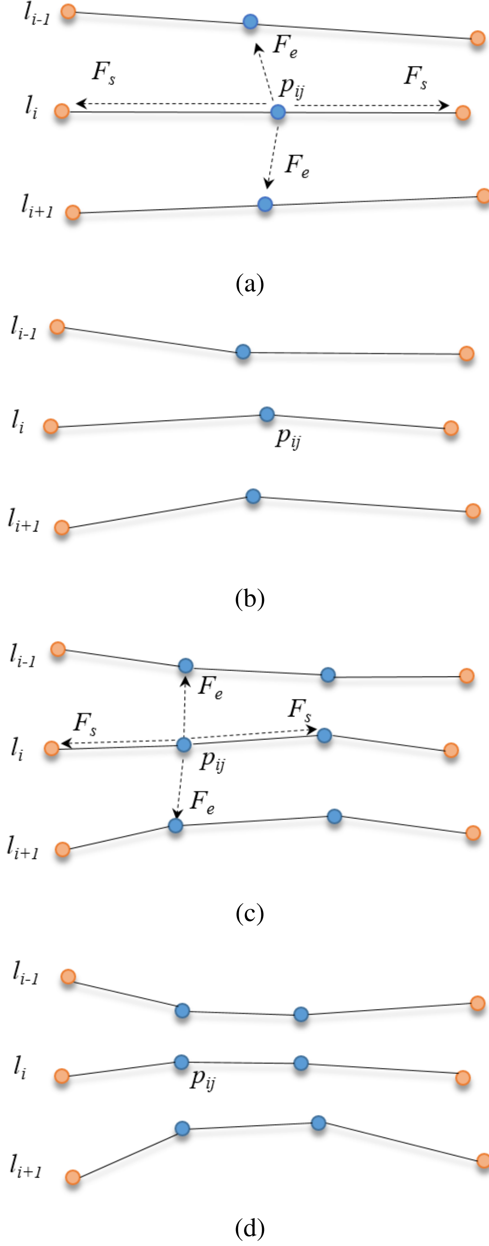


Figure 2. Edge subdivision and subdivision point movement in FDEB.

configuration of $P_0 = 1$, $C = 6$, $I_0 = 50$, and $R = 2/3$ leads to appropriate results.

The size of E can be significant for a large graph, and thus increase the computing cost. To address this, FDEB uses four criteria, angle, scale, position, and visibility, for edge compatibility measures. Only the edges with the compatibility measures greater than a threshold are considered as interacting edges. In this way, FDEB can control the amount of interaction between edges, and thus reduce the computing overhead.

B. Parallelization Strategy

We characterize FDEB and observe several possibilities to parallelize the computing with respect to simulation cycles and iteration steps:

- We can see that each line can be subdivided independently in each simulation cycle. On the other hand, each line should have the same number of subdivision points to compute the combined force. This implies that the line subdivision needs to be synchronized in each simulation cycle. That is, asynchronous line subdivision across simulation cycles is not allowed.
- We can see that in each iteration step, the combined force of each subdivision point can be computed independently. Similarly, the position of points can be updated in parallel as well.
- Given a specific configuration of P_0, C, I_0, and R, the total amount of subdivision points of a graph is deterministic after the last simulation cycle of FDEB. This implies that we can pre-allocate the memory to accommodate the data of final result.

Based on these observations, we can derive the following parallelization strategy:

First, we use a 2D matrix to represent the graph. Each row corresponds to one edge l_i, and each entry records the position of a point p_{ij}. The total matrix entry number is equal to the total point number that will be generated by FDEB in the final cycle. Initially, only the first two columns are filled, which correspond to the two end points of the original straight edges. We then insert the subdivision points into the 2D matrix at each simulation cycles. For example, Figure 3 (a) and (b) show the matrixes of C_0 and C_1, respectively, where the orange entries correspond to the end points, and the blue entries correspond to the subdivision points at each cycle. Compared to Figure 3 (a), we can see that a new blue entry (i.e., a new subdivision point) has been inserted into each row (i.e., an edge) of the matrix in Figure 3 (b). This process will be continued until the whole matrix is filled.

Second, given the positions of all points, we can compute the combined force $F_{p_{ij}}$ of a point p_{ij} using Equations 1, 2, and 3. The position value of p_{ij} then is updated by moving it a small distance according to $F_{p_{ij}}$. The updated position is directly recorded into the entry of p_{ij} in the 2D matrix.

Figure 3 illustrates this parallelization strategy. We can see that the computing task associated with p_{ij} is largely independent with each other. Intuitively, we can assign each task of p_{ij} to a thread, and execute them in parallel. Algorithm 1 shows a realization of our parallelization strategy[1]. Lines 12-18 correspond to the operation of edge subdivision, and Lines 21-29 correspond to the operations of force calculation and subdivision point movement. Special care has been taken

[1]To simplify the discussion, the technical detail of compatibility measure is not described, but its timing result is included in our performance evaluation.

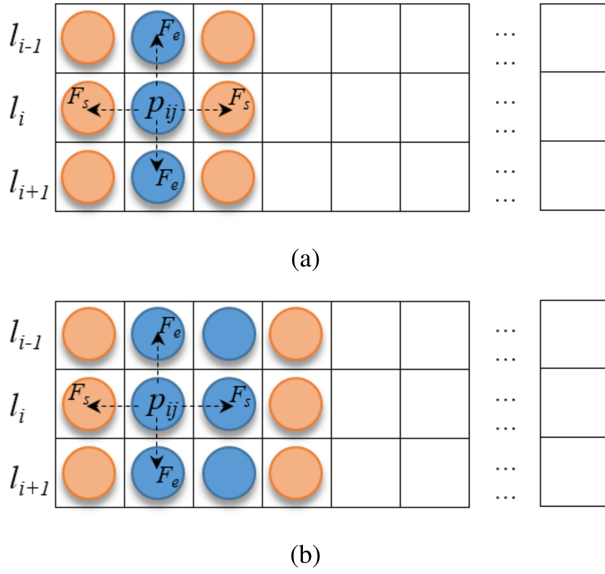


Figure 3. The 2D matrix representation of graph used in our parallel FDEB strategy.

Algorithm 1 ParallelFDEB

1: // Initialization
2: $C \leftarrow$ the number of simulation cycles
3: $I_0 \leftarrow$ the number of iteration steps in the first cycle C_0
4: $R \leftarrow$ the decreasing factor
5: $t \leftarrow$ the number of points of each edge after the last cycle
6: Given n edges, create one 2D matrixes M with n rows and t columns.
7: Fill the two end points of all edges into the first and third columns of M, respectively.
8: $C_i \leftarrow 0$; $I_i \leftarrow I_0$
9: // Simulation
10: **while** $C_i < C$ **do**
11: // Subdivide each edge
12: **for** each entry p_{ij} of M **in parallel do**
13: **if** p_{ij} is an end point at C_i **and** $C_i > 0$ **then**
14: Copy the corresponding end point at C_{i-1} to p_{ij}.
15: **else**
16: Compute p_{ij} as a subdivision point according to the previous subdivision points at C_{i-1}.
17: **end if**
18: **end for**
19: Synchronization
20: // Iteratively move the subdivision points
21: **for** each iteration step of I_i **do**
22: **for** each entry p_{ij} of M **in parallel do**
23: Compute F_s with respect to p_{ij}.
24: Compute F_e with respect to p_{ij}.
25: $F_{p_{ij}} \leftarrow F_s + F_e$.
26: Move p_{ij} a small distance in the direction of $F_{p_{ij}}$, and update p_{ij} in M.
27: **end for**
28: Synchronization
29: **end for**
30: $C_i \leftarrow C_i + 1$
31: $I_i \leftarrow I_i \times R$
32: **end while**

to ensure the synchronization across the simulation cycles (Line 19) and the iteration steps (Line 28).

Conceptually, Algorithm 1 can be easily implemented using multi-thread techniques on CPUs or GPUs (Line 12 and Line 22). For example, one implementation has been proposed using CUDA on NVIDIA graphics cards [14]. GPU-based implementations are especially attractive as a significant speedup can be possibly achieved by leveraging the massive parallelism of a GPU. However, these implementations often rely on specific graphics cards or graphics features, such as direct GPU memory write, that are not universally supported. In particular, mobile devices are often equipped with lower-end GPUs, and are less capable to carry out the calculation in Algorithm 1.

IV. Textured Based Edge Bundling

We present a new method to realize the visualization framework in Figure 1 by holistically addressing edge bundling and rendering. Our method uses standard graphics features specified in OpenGL and WebGL, and thus can be easily adopted by various devices with standard GPUs. In particular, if we conduct our method using WebGL, the web page of interactive edge bundling can be displayed on most web browsers without any modification.

A. Concept

The main challenge for us to perform FDEB using OpenGL or WebGL is that there is a lack of functionality of direct GPU memory access in OpenGL or WebGL. When we use OpenGL or WebGL, GPU memory access is typically conducted through texture functions. For GPU memory read, we can first upload data from the main memory to the GPU memory via texture binding, and then access texture data via texture lookup in vertex or fragment shaders. For GPU memory write, we can first bind texture as a Framebuffer Object (FBO) and render data into FBO. The rendering function can be customized using vertex or fragment shaders so that we can control the contents that are written to texture. Because we cannot directly read and write one texture simultaneously, we can use the ping-pong buffering (or double buffering) method [9] to read the input and write the output through textures, and thus enable the operations of edge subdivision, force computing, and position updating.

Algorithm 2 TEXTUREBASEDEDGEBUNDLING

1: // Initialization
2: $C \leftarrow$ the number of simulation cycles
3: $I_0 \leftarrow$ the number of iteration steps in the first cycle C_0
4: $R \leftarrow$ the decreasing factor
5: $t \leftarrow$ the number of points of each edge after the last cycle
6: Given n edges, create two 2D matrixes M_a and M_b. Each matrix has n rows and t columns.
7: Fill the two end points of all edges into the first and third columns of M_a, respectively.
8: $T_{in} \leftarrow M_a$ // Bind M_a as the input texture T_{in}
9: $T_{out} \leftarrow M_b$ // Bind M_b as the output FBO T_{out}
10: $C_i \leftarrow 0$; $I_i \leftarrow I_0$
11: // Simulation
12: **while** $C_i < C$ **do**
13:    TEXTUREBASEDSUBDIVISION(T_{in}, T_{out}, C_i)
14:    Swap T_{in} and T_{out} (i.e., the previous T_{out} becomes the input texture, and the previous T_{in} becomes the output FBO).
15:    // Iteratively move the subdivision points
16:    **for** each iteration step of I_i **do**
17:       TEXTUREBASEDUPDATE(T_{in}, T_{out}) Line
18:       Swap T_{in} and T_{out}
19:    **end for**
20:    $C_i \leftarrow C_i + 1$
21:    $I_i \leftarrow I_i \times R$
22: **end while**

B. Algorithm

The main algorithm of our method is illustrated in Algorithm 2. Given a specified configuration (i.e., P_0, C, I_0, and R), we can predict the number of points of each edge, t, after the last cycle. Given a node-link diagram with n edges, we construct two 2D matrixes M_a and M_b, and each matrix has n rows and t columns. Our goal is to fill each entry of one matrix with the final position of a subdivision point after the completeness of simulation.

Initially, we fill the two end points of all edges into the first and third columns of a matrix M_a. We then bind M_a as the input texture T_{in}, and bind M_b as the output FBO T_{out}. The 2D or 3D position coordinates are encoded into the color components of the input texture M_a. Then, during each simulation cycle, we use a fragment shader TEXTURE-BASEDSUBDIVISION to conduct edge subdivision (Line 13 in Algorithm 2), and then iteratively call another fragment shader TEXTUREBASEDUPDATE (Line 17 in Algorithm 2) to compute the forces and update the point position.

Algorithm 3 lays out the TEXTUREBASEDSUBDIVISION fragment shader. Each pixel p_{ij} of T_{out} represents the jth point of an edge l_i at a simulation cycle C_i. If p_{ij} is an end point at C_i, we can directly fetch the corresponding end point p_{ij} in T_{in} using texture lookup and copy the value to p_{ij} in T_{out}. Otherwise, we compute p_{ij} as a subdivision point according to the previous subdivision points in T_{in} using texture lookup. We then render p_{ij} as a color into T_{out}. The key of this step is the transform between texture coordinates and the indexes of lines and points. When we encode a data set into a texture, a data point is accessed through a 2D texture coordinate (x, y), where the x or y component of texture coordinate is typically within the range of 0.0 and 1.0. If we encode a 2D $n \times t$ matrix into a 2D texture, for an entry at the ith column and the jth row of the matrix, its 2D texture coordinate (x, y) is computed as:

$$x = i/(n-1), y = j/(t-1). \quad (4)$$

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

$$i = \mathbf{ceil}(x \times n), j = \mathbf{ceil}(y \times t). \quad (5)$$

We use this transformation to read (write) the points from (into) the textures.

Algorithm 4 lays out the TEXTUREBASEDUPDATE fragment shader. According to the texture coordinate of each pixel p_{ij} of T_{out}, we can use Equation 5 to compute the corresponding indexes of line and point, and locate the indexes of the points that are needed in Equations 1, 2, and 3. These indexes are then translated into the texture coordinates in T_{in} using Equation 4, which allows us to fetch

Algorithm 3 TEXTUREBASEDSUBDIVISION(T_{in} : $input_texture$; T_{out} : $output_fbo$; C_i : $simulation_cycle$)

1: **for** each pixel p_{ij} of T_{out} **in parallel do**
2:    **if** p_{ij} is an end point at C_i **and** $C_i > 0$ **then**
3:       Fetch the corresponding end point in T_{in} using texture lookup and copy the value to p_{ij}.
4:    **else**
5:       Compute p_{ij} as a subdivision point according to the previous subdivision points in T_{in} using texture lookup.
6:    **end if**
7:    Render p_{ij} into T_{out}.
8: **end for**

Algorithm 4 TEXTUREBASEDUPDATE(T_{in} : $input_texture$; T_{out} : $output_fbo$)

1: **for** each pixel p_{ij} of T_{out} **in parallel do**
2:    Fetch the corresponding points in T_{in} using texture lookup and use them to compute F_s and F_e with respect to p_{ij}.
3:    $F_{p_{ij}} \leftarrow F_s + F_e$.
4:    Move p_{ij} a small distance in the direction of $F_{p_{ij}}$.
5:    Render p_{ij} into T_{out}.
6: **end for**

the corresponding points in T_{in} using texture lookup and use them to compute F_s, F_e, and $F_{p_{ij}}$ with respect to p_{ij}. We then move p_{ij} a small distance in the direction of $F_{p_{ij}}$, and render p_{ij} as a color into T_{out}.

We swap T_{in} and T_{out} during each simulation cycle and each iteration step (Lines 14 and 18 in Algorithm 2) to enable GPU memory read/write via ping-pong buffering. In this way, we can always compute the new subdivision points and write them into the output texture according to the previous points stored in the input texture. Similarly, we can iteratively compute the forces and update the position of each point by flipping the input and output textures.

The entire bundling process is completed until we perform all C simulation cycles. We obtain the output texture where the entries are filled with the final positions of subdivision points of all edges. Then we need to render the edges encoded in the texture.

C. Rendering

After the final simulation cycles, we transform the structure of 2D matrix of the output texture into a 1D array, and bind this array into a Vertex Buffer Object (VBO) [9]. Then we can interactively render a large number of lines using VBO. 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 in the conventional method shown in Figure 1.

We note that texture lookup, FBO, VBO, and fragment shaders are standard graphics functions in the specification of OpenGL and WebGL, and are nearly fully supported by all graphics vendors. Thus, our method can be adopted to most devices with standard graphics processing units.

V. RESULTS

A. Performance Evaluation

We have evaluated the performance of our method. We have compared the following methods:

- the original CPU-based FDEB [8],
- the CUDA-based FDEB [14],
- the JavaScript-based FDEB of d3 [3],
- our TBEB method using a WebGL implementation.

There were three different devices used in our experiment:

- a desktop with an 8X Intel Core i7 3.60GHz CPU and an NVIDIA GeForce GTX 660 GPU,
- a laptop with an AMD mobile quad-core A10-5750M 2.5Ghz CPU and an AMD Radeon HD 8790M GPU,
- a Nexus 9 tablet with a dual-core Denver 2.3GHz CPU and a Kepler DX1 GPU.

We first used a graph of US migration (1732 nodes, 2180 edges). Table I shows the timing results. We can clearly see that the performance of our method is significantly higher than the CPU- and JavaScript-based FDEB methods. On the desktop, our method reaches around 20 frames per second (fps) for constructing and rendering the edge bundles, while the CPU (JavaScript)-based FDEB takes around 6 (19) seconds. Our method achieves a 124× (380×) speedup compared to CPU (JavaScript)-based FDEB on the desktop. The performance of the CUDA-based FDEB is superior to our method; however, it requires NVIDIA graphics cards.

Only our TBEB and the JavaScript-based FDEB can run on all three devices. As shown in Table I, our method reaches around 10 (5) fps on the laptop (tablet), and achieves a 662× (55×) speedup compared to the JavaScript-based FDEB. Overall, our method maintains an interactive or nearly interactive frame rate on all three devices.

Table I
PERFORMANCE COMPARISON USING THE US MIGRATION GRAPH.

	Desktop	Laptop	Tablet
CPU-based FDEB	6.23s	-	-
CUDA-based FDEB	0.001s	-	-
JS-based FDEB	19.06s	59.63s	12.80s
WebGL-based TBEB	0.05s	0.09s	0.23s

We have also conducted the performance comparison using a graph of US airlines (235 nodes, 2101 edges). Table II shows the timing results. Averagely, our method achieves a 240× speedup compared to JavaScript-based FDEB for processing the graph with two thousands of edges on three devices. In particular, on the laptop, our method only uses 0.16 seconds to compute and render the edge bundles, while the JavaScript-based FDEB takes around 11 seconds. For this data set, our method also achieves an interactive or nearly interactive speed on different devices.

Table II
PERFORMANCE COMPARISON USING THE US AIRLINES GRAPH.

	Desktop	Laptop	Tablet
CPU-based FDEB	3.82s	-	-
CUDA-based FDEB	0.001s	-	-
JS-based FDEB	12.52s	36.45s	11.10s
WebGL-based TBEB	0.05s	0.09s	0.16s

Apart from the real-world data sets, we used a set of synthetic random graphs with different numbers of edges (specifically, 100, 500, 1000, and 2000 edges) to conduct a scalability test. We compared our WebGL-based TBEB with d3's JavaScript-based FDEB on the desktop. As shown in Figure 4, we can clearly see that the computing time of JavaScript-based FDEB increases dramatically with the increasing number of edges. Therefore, it is less feasible to use this method to process large graphs. Compared to JavaScript-based method, the time of our WebGL-based TBEB increases marginally for larger graphs. Table III shows the detailed quantitative timing results. Our method can maintain interactive frame rates for visualizing a graph with thousands of edges.

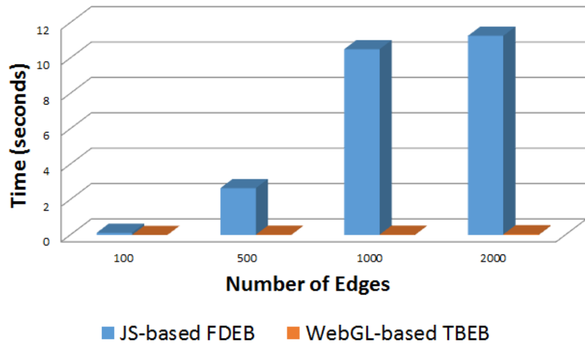


Figure 4. Scalability Comparison between JavaScript-based FDEB and WebGL-based TBEB.

Table III
SCALABILITY COMPARISON USING THE SYNTHETIC GRAPHS.

Number of Edges	100	500	1000	2000
JS-based FDEB	0.126s	2.648s	10.449s	11.255s
WebGL-based TBEB	0.004s	0.016s	0.017s	0.034s

B. *Visualization*

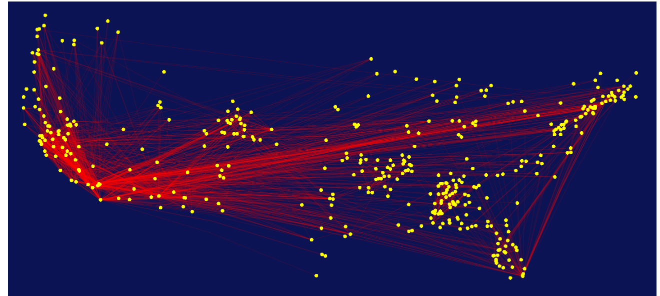
Figure 5 shows the visualization results of the US migration graph. We have compared our method with node-link diagram and FDEB. As shown in Figure 5 (a), a direct visualization of node-link diagram can easily incur visual clutter, and it is relatively difficult to perceive the main migration patterns from the graph. The high quality of our TBEB method in Figure 5 (c) is nearly identical to the original FDEB in Figure 5 (b). From both images, we can see the major migration routes between the East Coast and the West Coast and across the Midwest. The zoom-in images in Figure 5 (b) and (c) convey the detailed patterns among several cities around Texas. These patterns cannot be revealed clearly in Figure 5 (a).

Figure 6 shows the visualization results of the US airlines graph. Severe visual clutter is generated by directly visualizing the node-link diagram, as shown in Figure 6 (a). Figure 6 (b) and (c) are generated using FDEB and our TBEB method respectively, and have a similar high quality to clearly show the major bundles of airline routes that are difficult to be perceived using the node-link diagram.

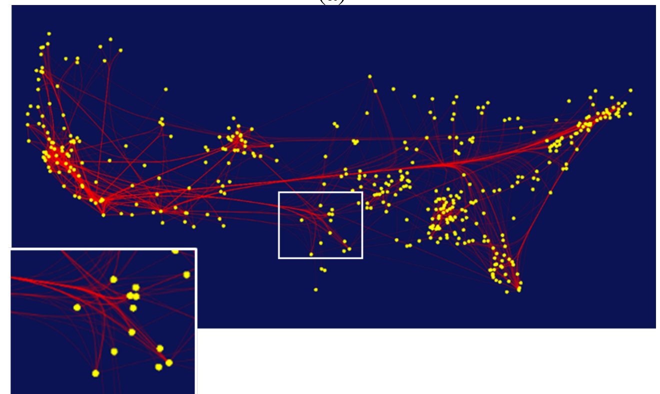
VI. CONCLUSION

We have presented a simple and efficient method for the construction and visualization of edge bundles of large graphs using standard features of consumer graphics hardware. The performance evaluation shows our TBEB method significantly improves the performance of edge bundling on different devices, and the visualization results clearly demonstrate that our TBEB method can achieve high quality visualization results as FDEB. A web-based implementation of our method can be versatilely applied on various devices, including ubiquitous smart mobile devices, without any modification. Our solution has significantly enhanced the interactivity and the usability of edge bundling, and enabled a user to possibly explore a large graph on a mobile device without visual clutter and at an interactive rate.

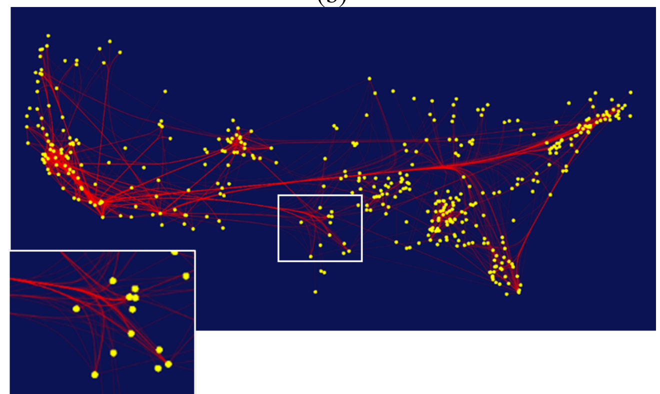
In the future, we would like to incorporate graph topological information into our method to improve the edge bundling quality. We also plan to extend our method on time-varying graphs by studying temporal visual coherence during animation. Moreover, although our TBEB method



(a)

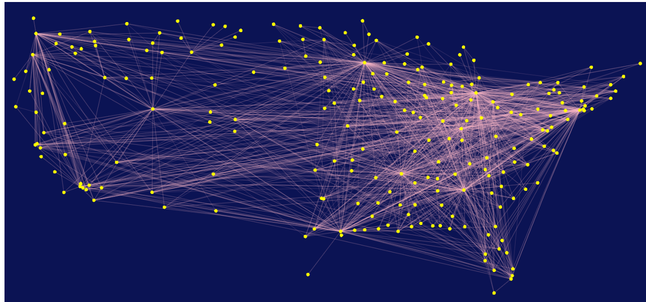


(b)



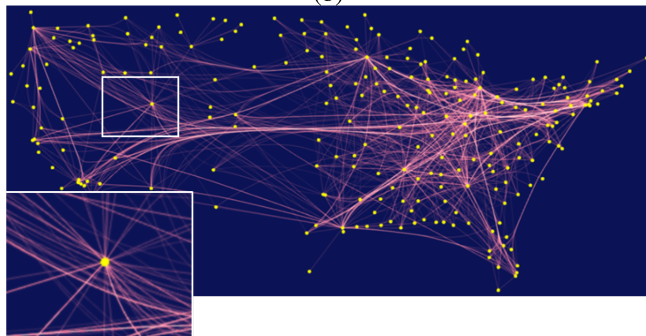
(c)

Figure 5. The visualization results of the US migration graph using a direct visualization of node-link diagram (a), the FDEB method (b) and our TBEB method (c).



(a)

(b)



(c)

Figure 6. The visualization results of the US airlines graph using a direct visualization of node-link diagram (a), the FDEB method (b) and our TBEB method (c).

targets edge bundling, we expect that the basic acceleration method and design strategy can be applied to other complex graph visualization algorithms on a device with a generic graphics computing capability. We will also investigate scalable solutions to integrate graph visualization algorithms with graph stores to provide users an interactive exploration of big and complex graph information on mobile devices.

Acknowledgment

This research has been sponsored by the National Science Foundation through grants IIS-1423487 and ICER-1541043.

References

[1] F. Beck, M. Burch, S. Diehl, and D. Weiskopf. The State of the Art in Visualizing Dynamic Graphs. In R. Borgo, R. Maciejewski, and I. Viola, editors, *EuroVis - STARs*. The Eurographics Association, 2014.

[2] W. Cui, H. Zhou, H. Qu, P. C. Wong, and X. Li. Geometry-based edge clustering for graph visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 14(6):1277–1284, 2008.

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

[4] O. Ersoy, C. Hurter, F. V. Paulovich, G. Cantareiro, and A. Telea. Skeleton-based edge bundling for graph visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2364–2373, 2011.

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

[6] I. Herman, G. Melancon, and M. Marshall. Graph visualization and navigation in information visualization: A survey. *Visualization and Computer Graphics, IEEE Transactions on*, 6(1):24–43, Jan 2000.

[7] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):741–748, Sept 2006.

[8] D. Holten and J. J. van Wijk. Force-directed edge bundling for graph visualization. In *Proceedings of the 11th Eurographics / IEEE - VGTC Conference on Visualization*, EuroVis'09, pages 983–998, 2009.

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

[10] D. Selassie, B. Heller, and J. Heer. Divided edge bundling for directional network data. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2354–2363, Dec 2011.

[11] A. Telea and O. Ersoy. Image-based edge bundles: Simplified visualization of large graphs. In *Computer Graphics Forum*, volume 29, pages 843–852. Wiley Online Library, 2010.

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

[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. In *Computer graphics forum*, volume 30, pages 1719–1749. Wiley Online Library, 2011.

[14] D. Zhu, K. Wu, D. Guo, and Y. Chen. Parallelized force-directed edge bundling on the GPU. In *Distributed Computing and Applications to Business, Engineering Science (DCABES), 2012 11th International Symposium on*, pages 52–56, Oct 2012.