CUBu: Universal real-time bundling for large graphs.

Matthew van der Zwan, Valeriu Codreanu and Alexandru Telea

Abstract—Visualizing very large graphs by edge bundling is a promising method, yet subject to several challenges: speed, clutter, level-of-detail, and parameter control. We present CUBu, a framework that addresses the above problems in an integrated way. Fully GPU-based, CUBu bundles graphs of up to a million edges at interactive framerates, being over 50 times faster than comparable state-of-the-art methods, and has a simple and intuitive control of bundling parameters. CUBu extends and unifies existing bundling techniques, offering ways to control bundle shapes, separate bundles by edge direction, and shade bundles to create a level-of-detail visualization that shows both the graph core structure and its details. We demonstrate CUBu on several large graphs extracted from real-life application domains.

Index Terms—I.3.3 [Computing Methodologies]: Computer Graphics–Picture/Image Generation; I.3.6 [Computing Methodologies]: Computer Graphics–Methodology and Techniques

1 Introduction

Very large graphs and networks have become pervasive in data-intensive applications such as traffic and network monitoring, software engineering, bioinformatics, and telecom applications. When the size of such datasets exceeds certain limits, understanding them becomes challenging. Edge bundling methods have become an important tool for this task: Given a large graph having a 2D spatial embedding of its nodes, bundling produces a simplified view of the graph structure by grouping spatially-close and semantically-related edges, so that edge-crossing clutter is reduced and the graph's main connectivity patterns become better visible. Bundling was used for applications in vehicle traffic [42], [21], [27], program understanding [8], [41], multivariate data analysis [29], and medical visualization [4].

While many bundling methods have been proposed, several key challenges exist with respect to their usability and usefulness:

Scalability: Recent techniques can bundle graphs of tens of thousands of edges in subsecond time [14], [22], [31]. While impressive, this speed is still insufficient for graphs such as Internet connectivity patterns, worldwide traffic flows, or call graphs of large software systems of millions of edges. Moreover, bundling time-dependent (dynamic) graphs, or changing bundling parameters during interactive data exploration, asks for bundling methods that are one to two orders of magnitude faster. Such methods do not exist yet.

Directions: Most bundling methods cannot separately bundle edges having different directions. This creates a high amount of *overdraw*, which precludes users from reasoning about the directions of edges in a given bundle [30], [20]. Current directional bundling methods, which aim to solve this issue, are however too slow to cope with interactive exploration of large graphs [43], [40], [38].

- *M. van der Zwan and A. Telea are with the University of Groningen, the Netherlands. Email: m.a.t.van.der.zwan@rug.nl, a.c.telea@rug.nl.*
- *V. Codreanu is with SURFsara. Email: valeriu.codreanu@surfsara.nl*

Level of detail: As bundling highlights the main connectivity patterns in a graph, users also need *level-of-detail* techniques able to emphasize the importance of a given bundle for the overall pattern. Various shading techniques have been used for this, *e.g.*, colormapping and alpha blending [17], [27], [20] and shaded tubes [46], [12]. While detail shading can effectively provide level-of-detail cues, it cannot (yet) be computed in real time and it is relatively complex to implement.

Generality: A final challenge is the proliferation of bundling techniques. While each such technique may excel in specific ways, *e.g.*, speed, ease of use, or achieving specific constraints on the resulting layout, achieving *all* these goals with a *single* algorithm is still hard [53]. Hence, users face the dilemma of implementing and using a large set of algorithms, or settling with the (dis)advantages of a specific algorithm.

We propose a single bundling algorithm: CUBu (CUDA-based Universal Bundling) to address all above challenges. We tackle scalability by a GPU bundling method that achieves average speed-ups of 50 up to 100 times *vs* the fastest existing general-graph bundling techniques [22], [14], [31]. Next, we propose two directional bundling extensions that are fast, robust, and easy to use. Thirdly, we show how to create multiscale bundled visualizations having the same quality as comparable methods [46], [12] and with a much simpler, and faster, implementation. Finally, we show how to create bundle styles proposed by widely different methods [9], [12], [17], [27] with our single method. As CUBu achieves all the above, we dub it a 'universal' bundling algorithm. We illustrate CUBu's speed and quality by several applications on large real-world graphs.

The structure of this paper is as follows. Section 2 covers related work. Section 3 describes our general bundling algorithm. Section 4 presents applications of CUBu in several domains. Section 5 discusses our method. Section 6 concludes the paper.

2 Related Work

We overview existing bundling methods based on the four feature, or requirements, classes listed in Sec. 1:

Scalability: Early bundling methods for compound graphs, *e.g.*, HEB[17], achieve a high scalability, by exploiting explicit hierarchical information present in the input graph. Methods for bundling general graphs (without hierarchical information) evolved from slow approaches, such as force-directed edge bundling (FDEB [18]), to advanced schemes to detect edge proximity and thus achieve faster bundling, such as control meshes (GBEB [9]), Voronoi diagrams (WR [27], [26]), medial axes (SBEB[12]), and radial kernel splatting (KDEEB [20], 3DHEB [31]). The MINGLE method uses multilevel edge clustering to accelerate the bundling process [14]. Scalability is critical for, *e.g.*, streaming (time-dependent) graphs: For the well-known *US airlines* dataset (900 edges), FDEB [18] achieves 19 seconds/frame on a 1.7GHz PC (see [18], Sec. 4.2); StreamEB achieves 6 seconds/frame on similar hardware [32]; KDEEB reaches 0.17 seconds/frame [20]; and 3DHEB yields 0.4 seconds/frame [31]. Such speeds are yet insufficient for real-time bundling of large static graphs or large dynamic graphs having hundreds of thousands of edges or more, such as the *wiki* or *amazon* graphs discussed in [14].

Directions: A bundle contains several edges placed close to or atop of each other. Bundles separate high-density edge groups by large white space areas, and thereby help perceiving the overall graph structure. Yet, bundling creates an undesired *overdraw* issue: We cannot display edge-specific attributes for (all) edges in a bundle, since these share the same screen space. This, in turn, makes it harder to reason about a bundle's semantics. The standard solution to this issue is to aggregate attributes at overlapping edges, *e.g.*, using averaging done by alpha blending [27], [17], [46], [18]. While this works well for quantitative scalar attributes, it yields wrong results for other types of attributes, such as edge *directions*. To address this, one can group same-direction edges into different bundles and, next, directionally color-code edges to show directions. Such *directional* bundling methods include divided edge bundling (DEB), which extends FDEB to include edge-direction compatibility [43]; attribute-driven edge bundling (ADEB), which extends KDEEB by a flow map encoding local edge-compatibility metrics [38]; and 3DHEB, which bins edges per-direction-interval and uses KDEEB to bundle each bin separately [31]. Yet, all these methods are much slower than undirected edge bundling, making them unsuitable for interactive large-graph exploration.

Level of detail: Graph splatting first proposed to depict a graph as a continuous image-space density field, computed by convolving the graph drawing by a Gaussian filter whose size controls the level-of-detail [51]. Accuracy issues on computing such dof the node ensity fields are discussed in [28], [42]. This image-space idea was extended in the LaGO tool by aggregating edges connecting local node-density maxima [54]. Recent bundling methods highlight a graph's main structure (dense bundles *vs* isolated edges) by alpha blending [27], [18], [17]. Shaded cushions [52], adapted to create crisp curved-and-shaded tubes along bundles [46], [12], better separate different pattern scales in a graph than splatting or alpha blending, and also separate crossing bundles better than alpha blending or straight-line edge aggregation. Yet, creating shaded bundle tubes is very complex, involving operations such as distance transforms and medial axes [46], [12], which are far from real-time performance.

Generality: Globally, we see three major styles in existing bundling methods: (1) *smooth* bundles, having few inflection points between end-nodes, are created by FDEB, KDEEB, and WR, and help easy visual following of large bundles; (2) strongly ramified bundles, showing a hourglass-like pattern between end-nodes, are created by HEB, GBEB, and SBEB, and help an easy detection of connection branching points; and (3) straight-line bundles, showing a highly simplified graph connectivity pattern, are created by LaGO, and help an easy recognition of end-to-end node connections. All these drawing styles have their merits (and limitations). The problem is that obtaining each such drawing style entails changing the bundling method being used. This is undesirable in terms of application-design simplicity and also forces users to learn parameter settings of many such methods.

3 Proposed method

CUBu's input is a graph drawing $G = (V, E)$ with vertices $V = \{\mathbf{v}_i \in \mathbb{R}^2\}$ and edges $E = \{\mathbf{e}_i \subset \mathbb{R}^2\}$. In detail: $\mathbf{v}_i$ are the positions of vertices, and $\mathbf{e}_i$ are the drawings of edges in G. For conciseness, we shall however call these next vertices and edges, respectively. Edges $\mathbf{e}_i$ can be straight lines or 2D curves, which covers bundling of both straight-line graphs [18], [9], [27] and spatial trajectories [21], [42], [20], [19]. Edges $\mathbf{e}_i$ are modeled as uniformly-sampled polylines, *i.e.*, $\mathbf{e}_i = \{\mathbf{x}_j\}$, where $\|\mathbf{x}_j - \mathbf{x}_{j+1}\|$ is a user-given sampling-step σ. The control points $\mathbf{x}_j$ of these polylines are further called *sites*. Edges can be either directed or undirected. Our algorithm's first phase creates a bundling $B \subset \mathbb{R}^2$ of G. For this, we propose a set of techniques which lead to massive performance improvements of CUBu as compared to all existing bundling methods (Sec. 3.1). Next, we describe several changes to the basic bundling algorithm that lead to the generation of different bundle shapes and styles (Sec. 3.2). Finally, we describe several methods to render the bundled graph drawing B using suitable shading, transparency, and color-mapping to emphasize various structures of interest on different spatial scales (Sec. 3.3).

3.1 Bundling algorithm

To achieve our goal of real-time bundling of large graphs, we efficiently use parallel computing architectures such as NVidia's CUDA or OpenCL. After studying the wide family of general-graph bundling methods, we found that KDEEB, one of the fastest bundling methods, is the most suitable to such parallelization. Yet, a careful study of KDEEB reveals several performance and accuracy issues. We next describe these issues, and how CUBu corrects them.

KDEEB follows the mean shift principle [7]: The set P of all sites $\mathbf{x}_j$ on all edges in E is convolved with a parabolic (Epanechnikov) radial kernel K of radius p_R, to obtain an edge density map $\rho : \mathbb{R}^2 \to \mathbb{R}^+$ as

$$\rho(\mathbf{x} \in \mathbb{R}^2) = \sum_{\mathbf{y} \in P} K\left(\frac{\|\mathbf{x} - \mathbf{y}\|}{p_R}\right). \quad (1)$$

Next, the sites $\mathbf{x}_j$ are advected upwards in the normalized gradient of ρ with a distance p_R, or in other words

$$\mathbf{x}_j^{new} = \mathbf{x}_j + p_R \frac{\nabla\rho}{\|\nabla\rho\|}. \quad (2)$$

Edges are next resampled to get an uniform and dense spatial distribution of the sites $\mathbf{x}_j$ over G, needed for a good kernel density estimation. Finally, a 1D Laplacian filter is applied on the edges to remove small-scale jitters created by the finite-step advection (Eqn. 2). The above four steps are repeated p_N times, while p_R is decreased from an initial user-specified value p_R by a small fraction at each step. We next discuss our changes to the four steps of the process (density map computation, advection, resampling, and smoothing), and their reasons to be.

Density map computation (step 1): KDEEB computes ρ using OpenGL by splatting the radial kernels K, encoded as 2D floating-point textures at the site locations $\mathbf{x}_j$, and accumulating the result ρ in a floating-point image buffer. All texture and image buffers in CUBu are square and of identical resolution, further denoted as $p_I \times p_I$ pixels. While simple to implement, KDEEB's OpenGL splatting takes about 40% of the total bundling time for the graphs discussed in [22]. For large graphs of over 1M sites, we measured that splatting reaches over 60% of the total bundling time. This is due to the fact that splatting uses a *scattering* model: Data from sites $\mathbf{x}_j$ is scattered to neighbor pixels within the kernel radius p_R, so parallelization is severely limited by the many concurrent image-writes – for a set of p_S sites and kernel radius p_R, computing the density map takes $p_S \cdot p_R$ pixel writes, many of which 'overlap' at the same pixel locations. We address this by replacing scattering with a *gathering* strategy: We compute $\rho(\mathbf{y})$ for each pixel $\mathbf{y}$ in the image ρ by summing up the contributions of all sites $\mathbf{x}_j$ closer to $\mathbf{y}$ than p_R. This reduces the number of pixel writes from $p_S \cdot p_R$ to p_I^2. Additionally, we split the 2D convolution (Eqn. 1) into two 1D passes, one over the rows and one over the columns of the image ρ, and treat several blocks of rows, respectively columns, in parallel via CUDA kernel invocations.

For gathering to work, we need to know, for each pixel $\mathbf{y}$, all sites $\mathbf{x}_j$ for which $\|\mathbf{y} - \mathbf{x}_j\| \leq p_R$. Typical solutions for this use spatial search structures, *e.g.*, kd-trees. While such techniques exist on CUDA [15], [6], they all need costly reinitializations after each advection step that moves the sites (Eqn. 2). We propose a faster solution: We first create a per-pixel site-density buffer $C : \mathbb{R}^2 \to \mathbb{N}$, where $C(\mathbf{x})$ gives the number of sites in E that fall inside pixel $\mathbf{x}$, by rendering all sites into the image C, using `atomicAdd` operations to take care of concurrent writes. Next, we compute $\rho(\mathbf{y})$ as

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

with $T(\mathbf{y})$ being a disk of radius p_R centered at $\mathbf{y}$. In contrast to scattering, where speed is bounded by the integral of ρ (Eqn. 1) over $\mathbb{R}^2$, the speed of our gathering is bounded by the much lower number of concurrent writes occurring when computing C, *i.e.*, the probability that two or more sites fall over the same pixel, *i.e.*, the number of edge intersections in the graph drawing. To further decrease this probability, rather than sampling edges uniformly with a step of Δ units (as in KDEEB), we use a sampling step of $\Delta + \frac{\Delta}{10}\delta$, where δ is a random real number uniformly distributed in $[-1, 1]$, computed by CUDA's *cuRAND* library. This decreases the chance that closely-spaced edges, appearing in later bundling iterations, have clusters of closely-spaced sites, separated by gaps of size Δ. A good by-product of our random sampling is that sites are more evenly distributed in image space. This leads to a better estimation of the density gradient $\nabla\rho$ used for advection, which is computed by finite differences. The setting of CUBu's core parameters p_K, p_N, p_I, and p_S is further discussed in Sec. 5.1.

Advection and resampling (steps 2 and 3): KDEEB advection (Eqn. 2) strictly follows the mean-shift idea, *i.e.*, moves sites upwards in the density gradient. Yet, since $\operatorname{div} \nabla\rho$ is practically never zero for our bundles (see analysis in [12]), advection increases the local edge-sampling density in negative-divergence areas, and decreases density in positive-divergence areas. To ensure a nearly constant sampling density (important for density estimation [22]), KDEEB resamples edges after *each* advection iteration. We measured this resampling cost on a wide family of graphs, and found it to be about 30% of the total bundling time, in line with [22]. Let us analyze what happens when advecting a site: The site's shift $\mathbf{x}_j^{new} - \mathbf{x}_j$ (Eqn. 2) can be decomposed into a drift along the tangent

$$\tau(\mathbf{x_j}) = \frac{\mathbf{x}_{j+1} - \mathbf{x}_j}{\|\mathbf{x}_{j+1} - \mathbf{x}_j\|} \quad (4)$$

of the graph edge sampled by $\mathbf{x}_j$ and a motion along the normal $\mathbf{n}_j$ to the edge at $\mathbf{x}_j$. Tangent drift $((\mathbf{x}_j^{new} - \mathbf{x}_j) \cdot \tau(\mathbf{x}_j))\tau(\mathbf{x}_j)$ causes sampling-density variations. We cancel this drift by advecting sites along $\mathbf{n}_j$, *i.e.*, replace $p_R\nabla\rho/\|\nabla\rho\|$ in Eqn. 2 by its projection along $\mathbf{n}_j$. In other words, Eqn. 2 is replaced by

$$\mathbf{x}_j^{new} = \mathbf{x}_j + p_R \left(\frac{\nabla\rho}{\|\nabla\rho\|} \cdot \mathbf{n}_j \right) \mathbf{n}_j. \quad (5)$$

This change, which has negligible computational cost, yields a much more uniform sampling density. We can now resample edges every three or four advection iterations, instead of every iteration, as in KDEEB. This gives a performance boost of about 20..30% as compared to KDEEB.

Implementation-wise, both advection and resampling are done in parallel with CUDA. For advection, the set P of sites is split into equal sizes among the available threads, which next apply Eqn. 5 in parallel to each site. For resampling, the set E of edges is similarly split into equal sizes among the available threads. Each edge $\mathbf{e} = \{\mathbf{x}_j\}$ is resampled by creating new sites $\mathbf{x}_j^{new}$, spaced equally along the polyline $\{\mathbf{x}_j\}$, with the user-given sampling-step σ (see the introduction of Sec. 3). After the complete set $\{\mathbf{x}_j^{new}\}$ is computed, it replaces the old set $\{\mathbf{x}_j\}$ in the description of $\mathbf{e}$.

Smoothing (step 4): As the final step of the bundling process, we perform one Laplacian smoothing iteration every 3..4 iterations rather that after each iteration (as in KDEEB). As for resampling, we need to perform smoothing less frequently than the original KDEEB since our constrained advection is more accurate than the KDEEB one, being controlled not just by the (imprecise) density gradient, but also by the shape of the edges themselves (due to normal projection). As advection, smoothing is implemented by splitting the site set P equally among the available threads

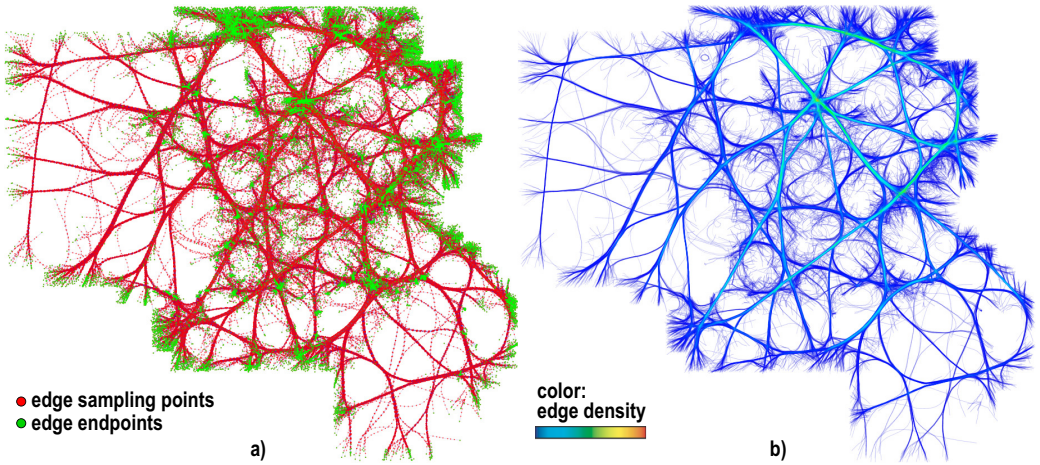

Fig. 1. a) Bundled graph with nodes (green) and edge sampling points (green). b) Density map for the same graph.

and next moving each site $\mathbf{x}_i$ to the location

$$\mathbf{x}_i^{smooth} = (1-\phi)\mathbf{x}_i + \phi\frac{\sum_{j=i-L}^{i+L}\mathbf{x}_j}{2L+1}. \quad (6)$$

Here, L represents the width (in sites) of the 1D Laplacian smoothing kernel, and ϕ represents the smoothing strength. Following similar applications of Laplacian smoothing in bundling, we set are $\phi = 0.5$ and $L = 10$ [18], [22], [12].

The modifications to the original KDEEB described in this section give a major performance boost to the CUBu algorithm. Performance is discussed in full detail further in Sec. 5.2.

3.2 Control of bundle geometries

Besides computational efficiency, a second main aim of CUBu is to offer simple but flexible ways to parameterize the geometry of the resulting bundles. This section describes two types of such parameterizations – bundle shape control and directional bundling – and how these can be easily integrated in the general-purpose four-step algorithm described in Sec. 3.1.

Bundle shape control: Different bundling techniques create different bundle *styles*, in terms of their shape, curvature, and thickness (see also Fig. 2a-e): HEB creates typical 'hour-glass' shapes by its B-spline control polygons that capture the underlying hierarchy tree. HEB shapes have been found effective for tasks involving finding high-level connections between node groups [8], [47]. HEB bundles also constrain edge directions close to their node endpoints, helping one to visually match edges with node glyphs. FDEB and related methods, *e.g.*, KDEEB, create bundles with less inflection points and smoother curvature variation than HEB, which are easier to follow visually [18], [32], [16], [22]. SBEB, GBEB, and WR create highly ramified bundles, which are good in showing splitting/merging of paths between node groups. While not typically seen as bundling methods, several techniques route spatially close edges along constrained paths, yielding highly simplified graph drawings [54], [27]. Separately, the TGI-EB framework enhances the classical FDEB algorithm with four different edge-compatibility metrics to create four bundling styles called centrality-based, topology-based, radial, and orthogonal, which support different tasks such as centrality analysis and finding cohesive groups of nodes [33]. Summarizing, different bundle styles support different analysis tasks and/or user preferences.

Making a single bundling technique generate all above bundling styles is not easy. Small changes can be done by parameter tuning, *e.g.*, the amount of Laplacian smoothing or edge relaxation [18], [12]. More complex style changes, *e.g.*, creating HEB-style bundles with FDEB or a schematic bundled layout with KDEEB, are hard or not even possible without changing the bundling algorithm itself. We achieve all such styles by changing a few parameters in CUBu, as follows.

HEB and FDEB styles: To create HEB bundles, we modulate site advection by an edge-profile function, *i.e.*, replace Eqn. 5 by

$$\mathbf{x}_j^{new} = \mathbf{x}_j + p_R\lambda(t(\mathbf{x}_j))\left(\frac{\nabla\rho}{\|\nabla\rho\|}\cdot\mathbf{n}_j\right)\mathbf{n}_j. \quad (7)$$

Here, $\lambda : [0,1] \rightarrow [0,1]$ controls the amount of motion of each site $\mathbf{x}_j$ as a function of the parametric arc-length position $t \in [0,1]$ of $\mathbf{x}_j$ along its edge. Using a hourglass-like function $\lambda_{HEB}(t) = ((1-8|t-0.5|)^3)^4$, having two symmetric inflections at its endpoints $t = 0$ and $t = 1$ and a plateau of value 1 in the middle (see also Fig. 4), creates HEB-style bundles, by gradually limiting the advection of sites close to bundle endpoints. Using a function $\lambda = 1$ produces the classical smooth fan-out common to general-graph bundling algorithms for both static graphs [18], [27], [12], [22] and dynamic graphs [32]. We call this style the FDEB style, giving credit to the FDEB technique [18], which was the first to pioneer it.

Small-world style: When exploring small-world graphs, one wants to see how a compact and strongly-related node group is connected (or not) to other node groups. Several methods do this by pre-clustering nodes based on connectivity and distance, and next drawing straight-line connections between nodes and their cluster centers, followed by drawing connections between cluster centers themselves [49], [50], [2], [54]. This yields a typical 'linked star' pattern akin to the one in a bubble graph layout [3], where stars show node clusters and links between star centers show higher-level connections between node clusters.

We obtain the same effect by edge bundling, without needing to explicitly cluster the input graph, as follows. Let $\mathbf{e}$ be an edge with node endpoints $\mathbf{v}_i$ and $\mathbf{v}_j$. We first run CUBu's basic advection step (Eqn. 2) on the set of all nodes

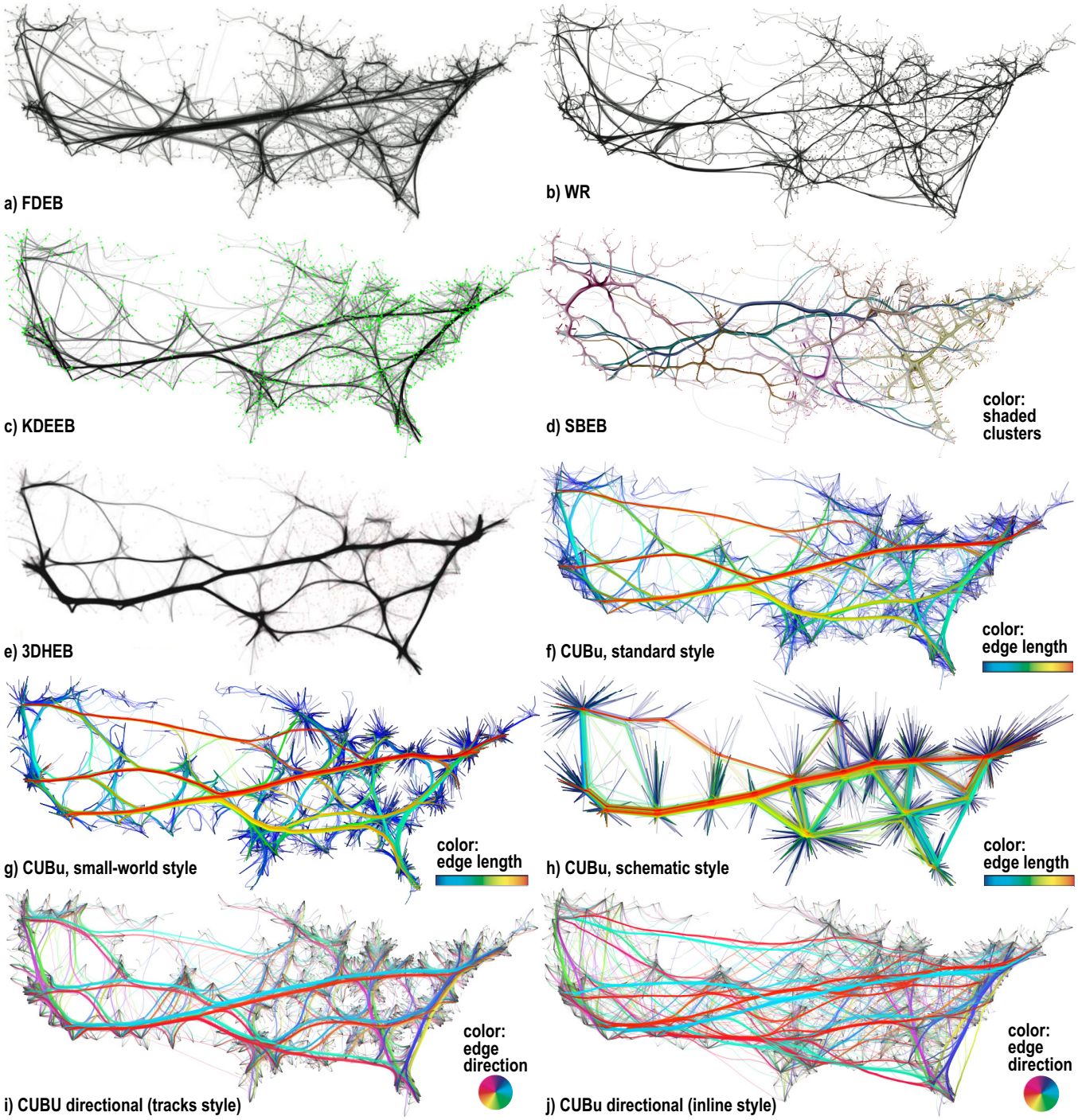

Fig. 2. Bundling styles for *migrations* graph. (a-e) Existing algorithms. (f-j) Styles produced by our single CUBu method.

$\mathbf{v} \in V$, rather than on all sites (sampling points) of all edges. Given that Eqn. 2 is essentially a mean shift process (see Sec. 3.1), this shifts each node $\mathbf{v}$ to a location $\mathbf{v}^c$ close to the center of its local neighborhood. Next, we insert the points $\mathbf{v}_i^c$ and $\mathbf{v}_j^c$ as the second, respectively one-but-last, sites on the sampled edge $(\mathbf{v}_i, \mathbf{v}_j)$. Finally, we apply the standard CUBu bundling on the resulting set of edge-sites.

Figure 2g, shows the effect of this technique. Several star structures appear, showing groups of related nodes. Bundles now link the centers of these stars, showing the node-group to node-group main connectivity patterns more explicitly than via standard bundling (*e.g.*, FDEB or KDEEB). More whitespace is left between the bundles, as edges will first automatically agglomerate by going to the node-group centers $\mathbf{v}_i^c$ and $\mathbf{v}_j^c$ prior to bundling proper. This further helps better visual separation of bundles. The kernel radius p_R used for mean shift clustering of edge endpoints gives the desired size of node neighborhoods: Large p_R values create fewer and larger node clusters, *i.e.*, show the coarse-level small-world graph structure. Small p_R values create more and smaller node clusters, *i.e.*, show the fine-level small-world structure. The lower bound of $p_R \leq 1$ pixels yields the standard KDEEB bundling.

Schematic style: Schematic bundled-graph drawing uses simple edge shapes and, at the same time, groups

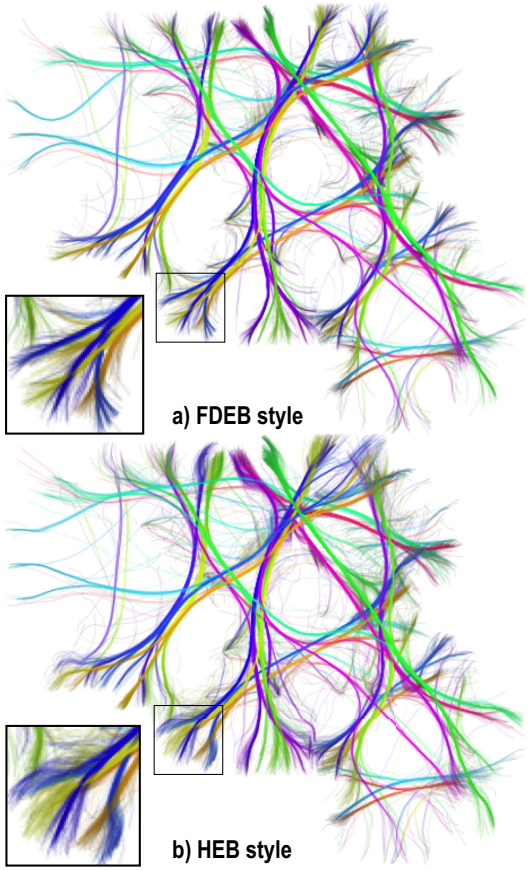

Fig. 3. FDEB *vs* HEB styles for *airlines* graph, using 'parallel tracks' directional bundling.

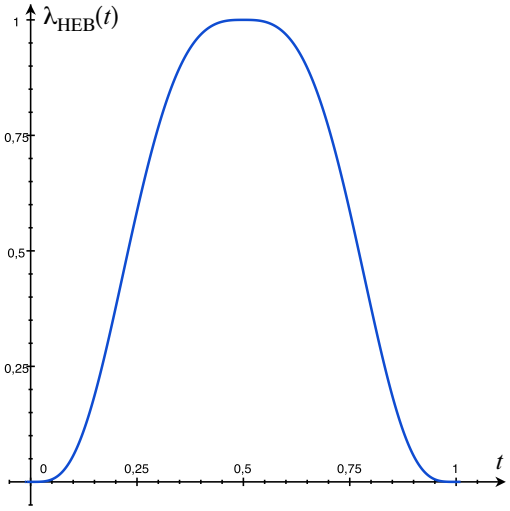

Fig. 4. HEB profile function λ_{HEB}. See Eqn. 7. and related text.

spatially close edges into bundles. Examples are orthogonal layouts used for software diagrams [10] and metro map layouts [45], [34]. CUBu can generate a particular type of schematic drawings, in which (a) spatially close edges are bundled and (b) bundles have shapes given by simple polylines consisting of a few segments. For instance, Fig. 2h can be seen as the schematic simplification of the bundlings in Figs. 2f or 2g. To achieve this bundling style, we simply disable the edge resampling step after each advection iteration (see Sec. 3.1, steps 2 and 3). As outlined in Sec. 3.1, this makes advection create a highly non-uniform sampling-point density consisting of a few spatially-separated dense point clusters. Edges consist of segments linking such clusters, thus have the desired coarse polyline structure.

Directional bundling: Drawing graphs with edges separated by direction is of recognized importance. Only a few bundling methods can do this: DEB [43] adapts FDEB's edge-compatibility function [18] to add directional similarity atop of spatial closeness. However, this method is quite slow. Similarly, ADEB [38] and 3DHEB [31] extend KDEEB's edge-density function to include a flow field capturing edge directions (ADEB) and respectively compute H edge-density maps, uniformly sampled over the 2π edge orientation range. FDEB and 3DHEB are 5 to 10 times slower than KDEEB, although implemented on the GPU. Other fast bundling techniques, *e.g.*, MINGLE, are hard to adapt to use directional compatibility. We describe next two directional bundling techniques that can be easily added to CUBu to produce similar results to DEB, 3DHEB, and ADEB, while keeping scalability.

Parallel tracks: Given a graph with edges $\mathbf{e}_i$ and sites $\mathbf{x}_j$, bundled by an undirected bundling method (*e.g.*, FDEB, SBEB, KDEEB, WR, or any similar method), we move each site $\mathbf{x}_j$ in the final bundled graph with a small distance $\epsilon\lambda(t(\mathbf{x}_j))$ in the direction $\tau(\mathbf{x}_j) \times \mathbf{d}$, where $\tau(\mathbf{x}_j)$ is the normalized tangent vector to $\mathbf{e}_i$ at $\mathbf{x}_j$ (Eqn. 4), $\mathbf{d} = (0, 0, 1)$ is the vector normal to the 2D layout plane, and t and λ are the edge arc-length parameterization and edge-profile functions λ used earlier for bundle shape control. In detail, we replace the $\mathbf{x}_j$ by

$$\mathbf{x}_j^{tracks} = \mathbf{x}_j + \epsilon\lambda(t(\mathbf{x}_j))\,(\tau(\mathbf{x}_j) \times \mathbf{d}). \tag{8}$$

This effectively 'splits' each bundle into two parallel railway-like 'tracks' separated by a maximal distance 2ϵ, so that all edges in a bundle that go in the same direction stay in one such track. As Figs. 3 and 2i show, this creates a uniform and regular separation of bundles into two thinner, parallel-running, bundles which can be next easily color-coded to show edge directions. The value ϵ is controlled by the user, typically set to about 10 pixels for good results. Parallel tracks can be applied to any of the bundling-geometry styles described in Sec. 3.2 with negligible cost. Moreover, parallel tracks can be applied as a simple 'postprocessing' step to the bundling itself, precisely as relaxation, *i.e.*, just before the visual exploration of the bundled graph (Sec. 3.3).

Inline directional bundling: Our second directional bundling technique takes place during bundling itself rather than as a postprocessing step. For this, we modify Eqn. 3 to compute, for each edge site $\mathbf{y}$, a density gradient $\rho(\mathbf{y})$ that accounts only for sites $\mathbf{x}$ within a radius p_R from $\mathbf{y}$ which have an edge-tangent vector $\tau(\mathbf{x})$ (Eqn. 4) that is compatible with the tangent $\tau(\mathbf{y})$. In detail, we replace Eqn. 3 by

$$\rho(\mathbf{y}) = \sum_{\mathbf{x}\in T(\mathbf{y})} K(\|\mathbf{x}-\mathbf{y}\|)C(\mathbf{x})\kappa(\mathbf{x},\mathbf{y}), \tag{9}$$

where $\kappa(\mathbf{x},\mathbf{y}) = \tau(\mathbf{x}) \cdot \tau(\mathbf{y}) \in [-1, 1]$. This bundles close same-direction edge fragments as usual, but forces close opposite-direction edge fragments to repel each other. In the above, we use for $\tau(\cdot)$ the tangent directions of the *original* (unbundled) edges, as we want to estimate directional compatibility based on the input, and not on the bundled, graph.

Compared to parallel tracks (Fig. 2i), inline directional bundling (Fig. 2j) creates larger separations between edges having different directions, and an overall more natural effect, quite similar to DEB. Bundle shapes can now adapt more freely, as they are only constrained by directionally compatible edges.

Ease of integration: Summarizing the above, it is important to note that all bundling styles described in this section can be integrated in the core CUBu algorithm (Sec. 3.1) with minimal changes, as follows:

- *FDEB style:* This style is created by default by the CUBu bundling algorithm described in Sec. 3.1;
- *HEB style:* To achieve this, we replace Eqn. 5 by Eqn. 7;
- *Small-world style:* To achieve this, we perform the advection step (Eqn. 2) on the node positions prior to the standard CUBu algorithm;
- *Schematic style:* To achieve this, we disable the resampling step (Sec. 3.1, step 3);
- *Parallel tracks style:* To achieve this, we apply Eqn. 8 to the final site positions of a bundled drawing;
- *Inline directional style:* To achieve this, we replace Eqn. 3 by Eqn. 9 in the density map computation (Sec. 3.1, step 1).

3.3 Visualization enhancements for bundled graphs

Besides computational scalability (Sec. 3.1) and control of the bundles' geometry (Sec. 3.2), a third contribution of CUBu targets the visualization of the resulting bundled graphs. In this section we describe two visual additions that help exploring the information conveyed by bundled graphs: color-and-opacity control for better visibility of short edges, and multiscale shading for emphasizing the coarse bundle structure. As for the bundle geometry control (Sec. 3.2), the visual additions discussed here integrate easily and keep the computational efficiency of the overall CUBu framework.

Color and opacity: Existing bundling techniques use color to encode geometric edge properties, *e.g.*, direction or length; or edge attributes, *e.g.*, time, height, or speed when bundling trail datasets [27], [20]. Alpha blending typically shows edge density ρ, *i.e.*, bundle importance, as opacity [17]. Yet, tuning alpha blending is not easy: Too high values make the drawing cluttered in high edge-density areas [11]; too low values make outliers, like sparse bundles and isolated edges, hard to see. Note that this is a separate problem from the issue of normalizing the opacity of the drawn bundled graph so that it lies between 0 and 1, so as to prevent opacity saturation during the drawing of the bundled edges. Such normalization can be easily done by simply using the normalized density ρ as per-pixel opacity. However, doing this for large graphs, where the amount of edge overdraw can vary hugely between dense-bundle zones and sparse edges, will make isolated edges appear almost transparent. This can be further corrected by using a non-linear opacity transfer function, that would emphasize low opacities. However, this still does not show well short edges: Due to their small length, such edges get a smaller chance to overlap other edges during bundling, as compared to long edges. As such, they will not be well visible, unless one considerably boosts the low opacity range. In turn, doing such a boosting can make the entire image too opaque, thus cluttered.

We propose an alternative color-and-opacity mapping scheme that takes into consideration the edge lengths. The scheme works as follows: Each edge site $\mathbf{x}$ has an $HSVA$ (hue, saturation, value, alpha) attribute. We allow setting H and S based on any edge attribute. Examples in this paper of setting H and S include both local and global attributes, such as the edge direction at $\mathbf{x}$ and the edge length respectively. V and A are set using a parabolic cushion profile function c, where

$$V(\mathbf{x}) = l/l_{max} + (1 - l/l_{max})c(\mathbf{x}), \tag{10}$$

$$A(\mathbf{x}) = \alpha \cdot (1 - l/l_{max} + l/l_{max}c(\mathbf{x})), \tag{11}$$

$$c(\mathbf{x}) = \sqrt{1 - 2 \cdot |t(\mathbf{x}) - 1/2|}. \tag{12}$$

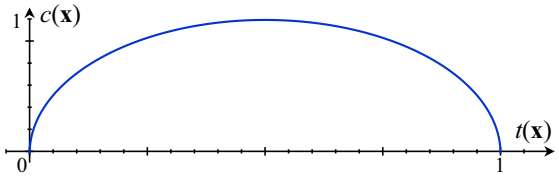

Fig. 6. Parabolic profile $c(\mathbf{x})$. See Eqns. 10-12.

Here, l is the length of the current edge; l_{max} is the length of the longest edge in E; t is the edge arc-length parameterization explained in Sec. 3.1; and $\alpha \in [0, 1]$ is a parameter that controls the drawing's overall opacity. The parabolic profile $c(\mathbf{x})$ is shown in Fig. 6. Note how the luminance V and opacity A are functions of both the position of the site along the edge $\mathbf{x}$ and also of the relative edge length l/l_{max}. As such, short edges get constant opacity A but a parabolic luminance V profile, whose gradient makes them appear more salient in the image, which helps seeing them better. Long edges get a flat luminance V and parabolic opacity profile. This way, their end fragments become less opaque, and thus make more 'room' around their end vertices for short edges that may exist in these areas to be better visible. Their flat luminance de-emphasizes them, as opposed to short edges, since their length is a strong enough visual cue to make them visible.

Figures 2f-j show how the proposed color and opacity design works: Compared to Figs. 2a-e, which use classical alpha blending, we now see many more detail edges that connect to isolated nodes.

Multiscale shading: Edges in bundled drawings can be hard to follow end-to-end due to crossings [30], [46]. This can be helped by a shading effect that makes bundles appear like 3D overlaid tubes rather than flat 2D shapes. The shading gradient (high across a bundle, low along it) makes the visual separation between crossing bundles easier. This can be done by explicitly computing separated bundles, by clustering edges belonging to the same bundle, and rendering generalized shaded cushions atop such bundles [52]. Such shading involves complicated and costly operations: edge clustering, distance transforms, and medial axes [46], [12]. For graphs of tens of thousands of edges or more, doing this at interactive rates is not possible.

We propose a new fast way to compute shaded bundles from a bundled graph drawing. Consider the density map ρ (Eqn. 1) as a height plot surface $z = \rho(\mathbf{x})$. At each pixel $\mathbf{x}$, we estimate the normal $\mathbf{n}(\mathbf{x})$ to this surface as the

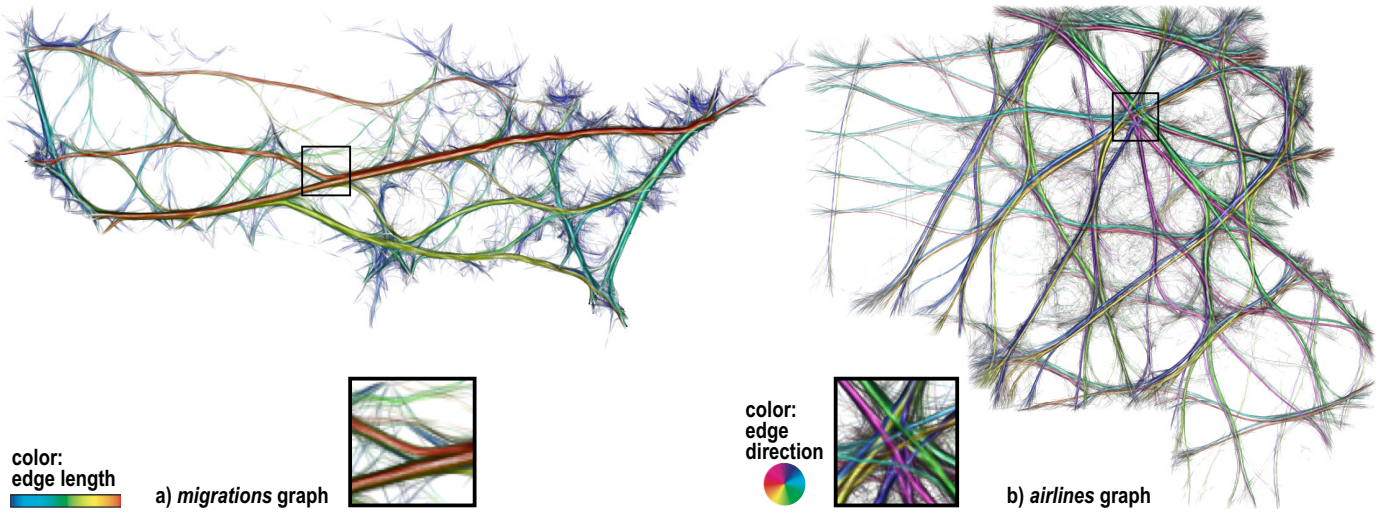

Fig. 5. Multiscale tube shading for the *migrations* and *airlines* graphs. Insets show shading details (Sec. 3.3).

normalized 3D vector $\left(\frac{d\rho}{dx}, \frac{d\rho}{dy}, -1\right)$, with derivatives of ρ computed by central differences. We next use $\mathbf{n}(\mathbf{x}_i)$ to shade each edge site $\mathbf{x}_i$ by the classical Phong lighting model, like [27]. However, this approach has a key problem: The density ρ can vary hugely over its domain: Along dense and tight bundles, ρ derivatives have high values, yielding almost horizontal normals, thus no illumination (if we use a vertical light vector); along sparse bundles of just a few edges, ρ derivatives get very low, yielding an almost vertical normal $\mathbf{n}$, thus maximal illumination. This is opposite to our goal to de-emphasize dense bundles and emphasize sparse ones. We handle this by a different shading model: For each pixel $\mathbf{x}$, we compute the maximal value $\rho_{max}(\mathbf{x})$ of ρ over a disk of radius r centered at $\mathbf{x}$. Next, we use $\rho(\mathbf{x})/\rho_{max}(\mathbf{x})$ as height in the above lighting computations. This locally normalizes ρ so that dense bundles appear as shallower bumps and sparser ones as taller ones, respectively. The parameter r controls the smoothing amount in a multiscale way: Larger values make bundles have larger highlights and less sharply-defined borders; smaller values create sharper highlights and bundle borders. Good values for r range between 5 and 15 pixels, which is roughly the thickness of a bundle in the image. Computing bundle shading is easily implemented in CUDA by a single pass over all pixels $\mathbf{x}$ of the density image ρ. The complete shading takes a few milliseconds, since the partial derivatives of ρ are already available from the gradient computation (Eqn. 2), and the remaining operations are very simple. In contrast, the methods in [46], [12] involve (a) clustering of edges in E based on geometric similarity, which takes a few seconds for graphs of hundreds of thousands of sites; (b) the computation of the Euclidean distance transform of the edge drawing, its thresholding, and the computation of the skeleton (medial axis) of the thresholded image, which takes, when using fast GPU-based methods [5], about 50..100 milliseconds on images of 512^2 pixels; and (c) the evaluation of a Phong-like lighting model using the above skeletons and distance transforms. For full details, we refer the reader to [12], [46]. Overall, the costs of this type of tube shading are about two orders of magnitude larger than our method.

Figure 5 shows our tube shading results. As visible, dense (important) bundles stand out easily, in terms of color and shading.

4 Applications

Huge graphs: Our first example uses the *amazon* graph, which records about 900K co-purchase relations between about 520K items on *amazon.com* [14]. Figure 7 shows this graph, visualized with MINGLE, KDEEB, and CUBu. We see that MINGLE only exposes the edge-density pattern in the graph; KDEEB shows some structure, but cannot outline the main connection patterns. In contrast, CUBu clearly shows these connection patterns. We also see how CUBu generates very similar results for two different sampled versions of this graph, taken from [14] (Figs. 7c,e); and how tuning the radius r used for multiscale shading generates coarser *vs* finer-grained visualizations of the graph structure (Fig. 7c *vs* d). Figures 7f-i show the bundling of the *wiki* graph, which contains about 105K edges representing relations between Wikipedia pages, computed with four different methods: MINGLE [14], LaGO [54], KDEEB, and CUBu. We notice how LaGO mainly shows the node density, and a few edges (in red) between nodes selected by the user, but cannot reveal the entire edge structure. MINGLE succeeds in showing that the graph consists of a dense 'core' of nodes (light blue) to which peripheral nodes connect. In contrast, CUBu's FDEB style shows clearly that the graph part that surrounds the code has a quite regular structure. CUBu's small-world style shows even better how groups of nodes far from the core connect to the core. Note also that following a bundle from its end outside the core to where it ends inside the core is much easier in the CUBu images (Fig. 7h,i) than in the MINGLE image (Fig. 7f).

Flight exploration: We consider dynamic trail-sets formed by the trajectories of airplanes over given space-time regions. Such data is provided by either air-traffic control (ATC) [20], [23], or gathered by hobbyists that record ADS-B signals [1] used by planes to transmit their name, position, altitude, call sign and status information, and consolidated into a global server [39]. Dynamic bundling can be used to show changes of the main structure of the connectivity pattern implied by the trails [20], [25]. However, if undirected bundling is used, trail-sets being

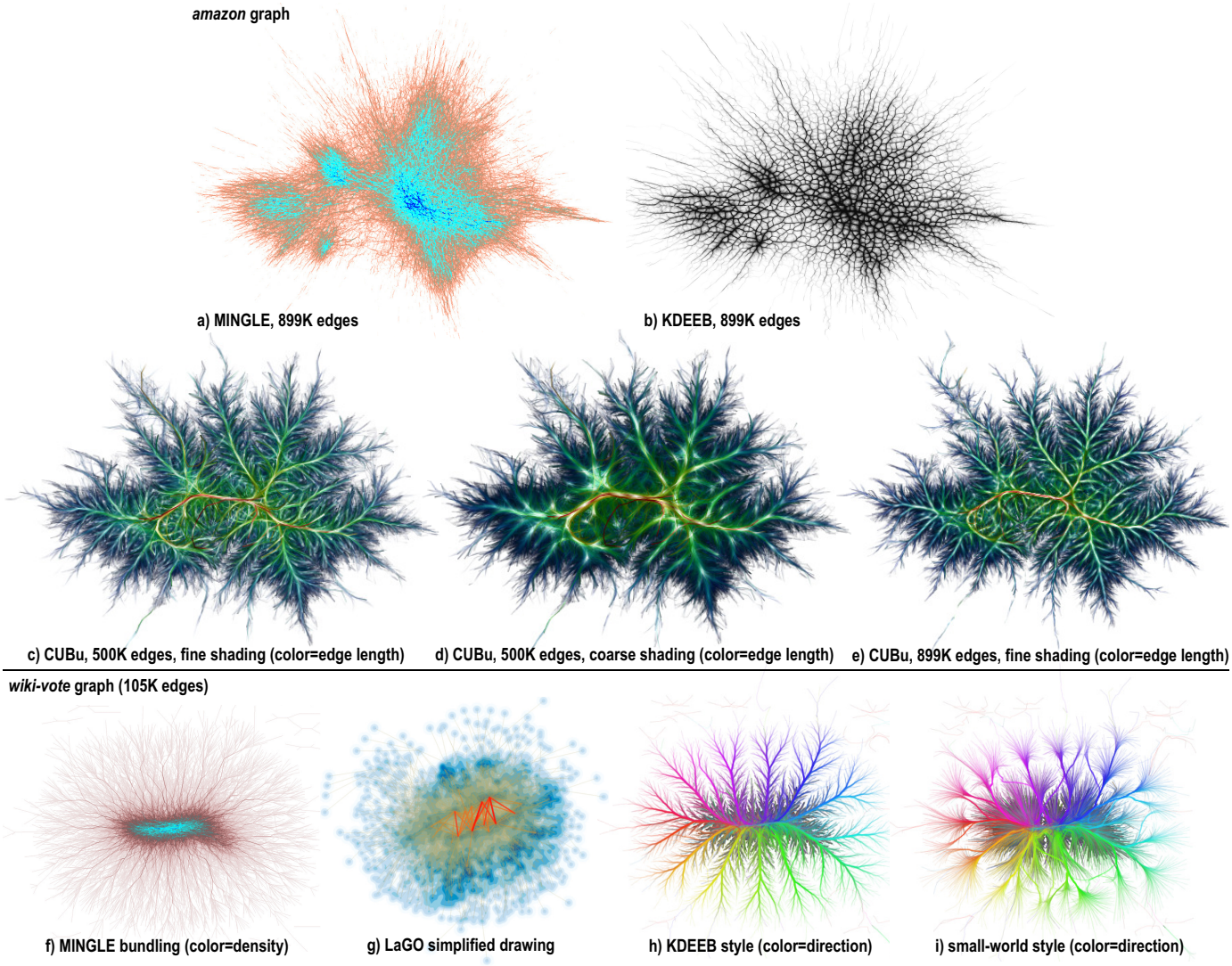

Fig. 7. Multiscale visualization of large graphs (*amazon* and *wiki*) at different sampling resolutions and shading scales (Sec. 4).

close to each other and going in opposite directions get bundled together. In turn, when using color-mapping to show trail attributes, *e.g.*, direction, colors mapping different values get blended together, resulting in wrong insights. Existing directed bundling methods [43], [18], [38] are orders of magnitude too slow for the real-time requirements of dynamic trail exploration, where we need to create tens of frames (thus, bundled layouts) per second to yield a smooth animation.

CUBu's fast directional bundling solves both above problems. Figure 8 shows a frame of the dynamic bundling of all world flights in the database [39], corresponding to the morning of June 1^{st}, 2013 (about 26K flights from a total of 750K flights in the database). The unbundled flight display (Fig. 8 a) shows several clutter areas, where we see only a single direction-color (A) or false colors, not even existing in our directional colormap (B, C). Undirected bundling also creates false colors, see *e.g.* the detail in Fig. 8 c, corresponding to a small zone above Paris. Directed bundling (Fig. 8 b) clearly separates trail-sets running in opposite directions. We now see that in all regions $A..C$ there is symmetric traffic in both directions. Also, no false colors are created (Fig. 8 d).

Projection analysis: Multidimensional projections are efficient and effective tools for mapping multivariate datasets, having tens or hundreds of attributes per data point or observation, to a 2D scatterplot, so that high-dimensional similarities between points are preserved in this scatterplot [36], [37], [35], [44], [24]. As all existing projection techniques cannot *faithfully* preserve high-dimensional distances, showing erroneously-projected points is crucial to using the resulting projections [29]. One way to spot wrongly-projected points is to draw point-to-point connections (edges) and color these by the projection error [29]. However, this creates a very large, and cluttered, set of lines. Figure 9a shows this for a relatively small set of 2300 19-dimensional points from the well-known *segmentation* dataset describing image fragments [13], projected to 2D using the LAMP technique [24]. Here, point-to-point projection errors are color-coded using a rainbow colormap (blue=low projection errors, red=high projection errors). Little structure, in terms of point-groups sharing similar projection errors, can be seen. Using CUBu, we can create a bundled layout of these point-to-point error edges, which now better shows that the main errors affect the top *vs* bottom point groups (Fig. 9b). Using our tube-like shading, we now can better spot the top-to-bottom bundle,

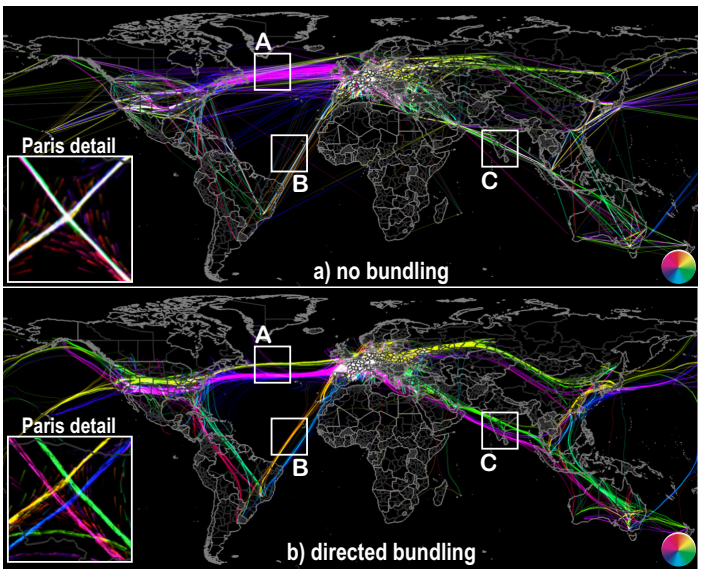

Fig. 8. World flights (June 1^{st} 2013), raw *vs* directed bundling (Sec. 4).

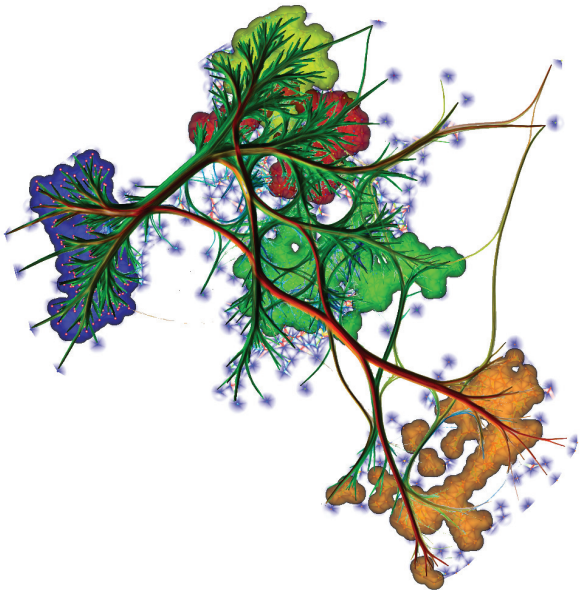

Fig. 10. Bundled graph showing projection errors between point-groups in a multidimensional projection. Image from cover of [48].

and also see that an important left-to-right bundle exists (Fig. 9c). Finally, using HEB-style bundles allows us to better see how individual projected points participate in bundles, *i.e.*, are affected by projection errors (Fig. 9d). All these visualizations are generated in real-time on a commodity PC, due to our fast CUBu bundling technique.

Figure 10 shows a more complex usage of CUBu to depict projection errors in a multidimensional projection of a high-dimensional dataset. The projected points, visible as blue dots, are grouped into five clusters, based on their attribute similarity. Each cluster is shown by a colored shaded cushion, using five categorical colors. Bundles show similar points which are placed far away from each other by the projection, *i.e.*, the most important projection errors, and are colored by the projection error magnitude. The image is taken from the cover of a recent visualization book [48].

Eye tracking: Our final example shows the eye tracking

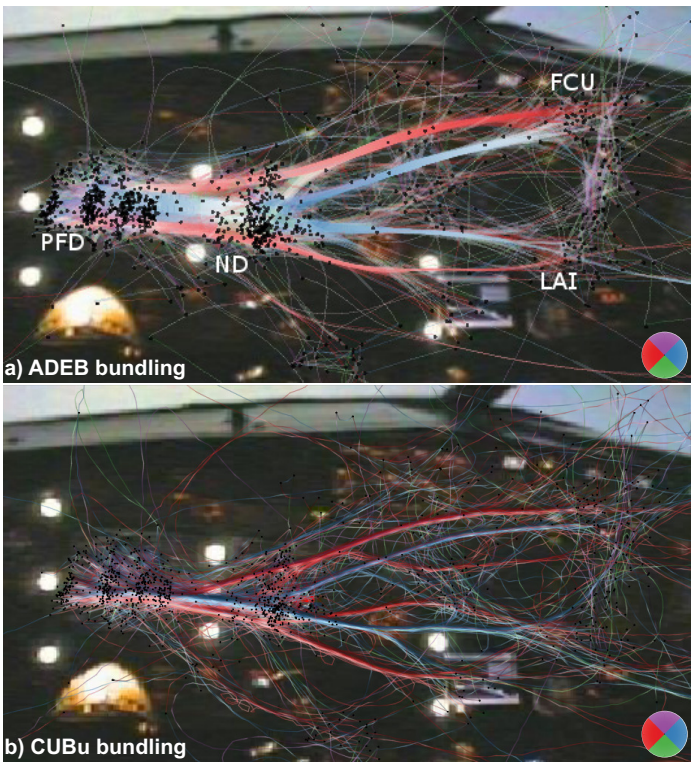

Fig. 11. Eye tracking analysis of pilot gaze (Sec. 4).

data set used in [20], which comes from tracking the eye movement over the instruments of an airplane, in a scenario involving a pilot performing a landing manoeuvre in a flight simulator. The aim of this experiment was to test a new cockpit instrument providing landing assistance and its use in combination along the other instruments in the cockpit [20]. In Fig. 11 a, this new instrument is indicated as LAI (Landing Assistance Interface). The vertices in the graph are the points to which the eyes where drawn (so-called *fixation points*) and the connecting edges indicate eye-movement between the vertices (so-called *saccades*). For this dataset, fixation points and saccades were obtained from raw eye-tracking data following the protocol detailed in [20].

Drawing the raw saccades between fixation points generates a completely cluttered image, from which high-level connections between fixation points cannot be inferred. However, bundling can be used to de-clutter and simplify such an image [38]. Figure 11 compares this approach using the attribute-driven edge-bundling method (ADEB) [38] and our CUBu method. The figure shows bundles generated by ADEB *vs* those generated by CUBu, both using the same color scheme to show the main direction of the original (unbundled) edges. Like the image generated with the ADEB method, the CUBu bundled image (Fig. 11 b) shows the same connections between instruments. However, we can also see a smaller but still significant bundle along the central axis of the image due to the different rendering style allowed by CUBu. Also, CUBu is roughly 100 times faster than ADEB for this dataset (see also Tab. 1).

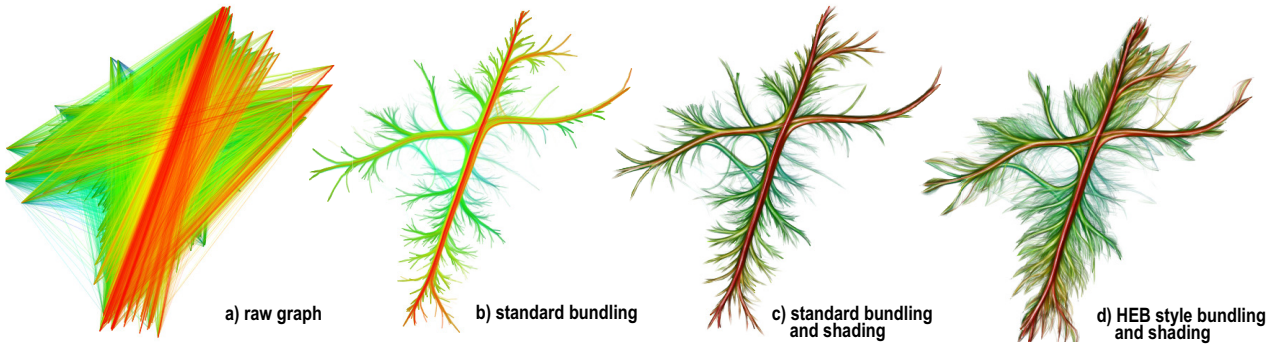

Fig. 9. Projection errors. Color maps edge lengths (Sec. 4).

5 Discussion

We have implemented CUBu using C++ and NVidia CUDA 2.1 for the bundling part and OpenGL 1.1 for rendering. We have tested CUBu on Linux and Mac OS X with several GPUs (GT 330M, GeForce 580, and single and dual-GPU GTX 690). For the dual GPU, we split all work (density estimation, advection, resampling, and smoothing) evenly on the two GPUs. For testing, we used a variety of graphs, including all static graphs in [18], [14], [22], [29] and the dynamic graphs in [20], [25]. These range from a few hundred nodes and edges to hundreds of thousands of nodes, almost a million edges, and almost 20M edge sampling points (sites).

5.1 Parameter settings

Kernel radius p_R: The initial kernel size, specified in pixels, controls bundling coarseness. In detail, p_R tells the user the maximum distance at which two edges 'see' each other, *i.e.*, get bundled. Small values yield more, and sparser, bundles; large values yield a simpler view having less and denser bundles. A good preset for p_R is 5% of the size of the graph drawing.

Bundling iterations p_N: The number of bundling iterations should be large enough so that bundling converges to a stable structure. For all tested graphs, we verified that $p_N \in [10, 15]$ leads to convergence, although graphs having already closely spaced edges, such as trail sets [20] converge with less iterations. Hence, we conservatively preset $p_N = 15$.

Image resolution p_I: The output image size controls the accuracy of density estimation (Eqn. 3), and thus also of the gradient estimation by finite differences (Sec. 3.1). Typical applications would use $p_I = 512^2$ pixels. For high-quality results, such as the images in this paper, we used $p_I = 1024^2$ pixels.

Sampling points p_S: The edge sampling density, equal to the total number of sampling points p_S divided by the sum of all edge lengths, also affects bundling quality. Intuitively, we want the sampling density to be high enough to capture the smallest details of interest in our graph drawing, but not higher, as this decreases speed. For all our test graphs, we found this optimal density to be roughly equal to one sampling point per 10 pixels of edge length.

5.2 Performance

CUBu's performance depends on its four parameters: kernel radius p_R, number of bundling iterations p_N, image resolution p_I, and sampling point count p_S. To analyze scalability, we varied all four parameters, one at a time, while keeping the other three fixed around good default values, and measured the bundling time. Figure 12 shows our timings on the single and dual-GPU GTX 690 platform, thereby also showing multi-GPU scalability. We see that bundling speed is linear in p_N, p_S, $\sqrt{p_I}$, and roughly independent on p_R. Also, we see that CUBu scales well on a dual-GPU platform. Since our dual-GPU design simply splits workload between the two GPUs, it should also scale well on a platform having more than two GPUs. This is an important result, as none of the bundling algorithms known so far do takes advantage of multi-GPU capabilities.

FDEB	DEB	SBEB	GBEB	ADEB	3DHEB	KDEEB	MINGLE	CUBu
65	27.5	26.6	5.6	1.3	0.7	0.5	0.2	0.014

TABLE 1
Bundling times (seconds) for several methods for the *US airlines* graph (235 nodes, 2099 edges, 86K sample points), see Sec. 5.2.

The complexity of CUBu is $O(p_I p_N p_S)$, worst-case identical to KDEEB and ADEB. Yet, the highly-parallel design of CUBu ensures that it is 30 to 100 times faster than KDEEB, the fastest known undirected bundling competitor (Tab. 2). Since the principle behind KDEEB and CUBu is the same, further detailing this performance boost is worthwhile. The by far most important gain of CUBu is realized in the computation of the density map (Eqn. 1), which is done by gathering rather than scattering. As explained in Sec. 3.1, this optimally uses the GPU by minimizing the number of concurrent writes, which is the major bottleneck in the KDEEB design. As the density map computation is 40..60% of the total KDEEB time (Sec. 3.1) *and* since this is the step we improve the most, this delivers the largest part of the speed gains of CUBu. The second main speed boost is given by the fact that CUBu is entirely implemented on the GPU, while KDEEB performs the gradient computation, edge advection, edge resampling, and edge smoothing on the CPU. This, and the CPU-GPU data transfers, are the second main bottleneck of KDEEB.

Compared to ADEB, CUBu is 60 to 200 times faster, as ADEB is half the speed of KDEEB [38]. Further comparison, done on the well-known *US airlines* graph (235 nodes, 2099 edges, 86K sample points), are listed in Tab. 1. These results are not surprising given that the complexities of MINGLE, FDEB, GBEB are essentially quadratic with respect to p_S, while the complexity of CUBu is linear with respect to p_S.

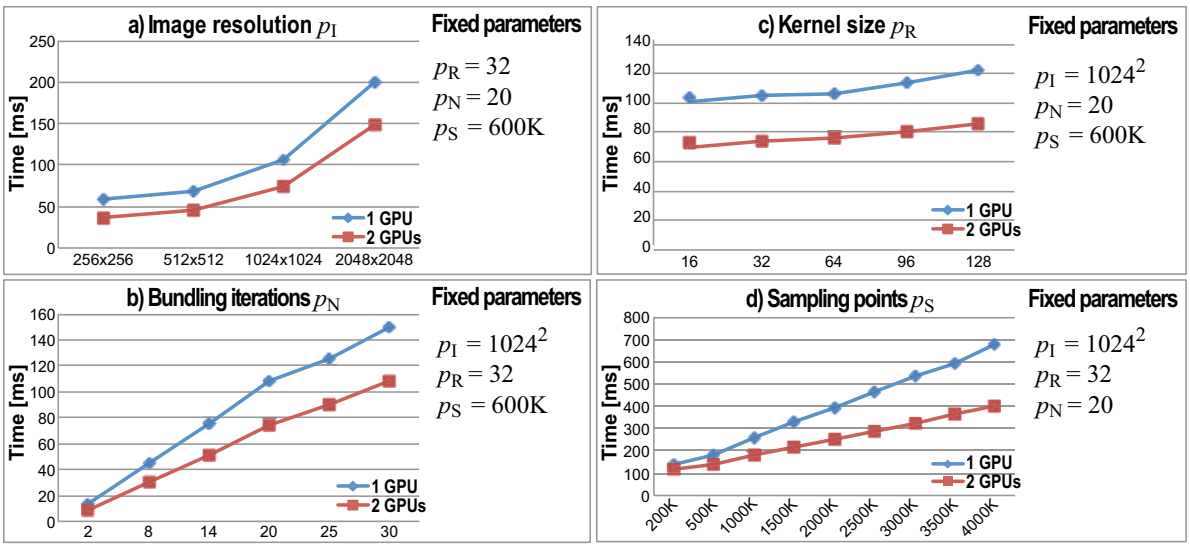

Fig. 12. Performance scalability as function of the CUBu algorithm parameters (Sec. 5.1).

Graph	Nodes	Edges	KDEEB		CUBu (1 GPU)	
			Samples	Time (ms)	Samples	Time (ms)
US airlines	235	2099	86K	500	86K	14
US migrations	1715	9780	220K	1500	221K	24
Radial	1024	4021	290K	1500	290K	23
France airlines	34550	17275	330K	1800	330K	25
Poker	859	2127	50K	400	50K	11
Amazon	738491	899791	19M	8053	19M	152

TABLE 2
Timings for CUBu and KDEEB [22] for several graphs (Sec. 5.2).

5.3 Generality

CUBu can handle graphs of any topology and having an initial edge layout given by curves or straight lines, as long as we have a 2D node layout. By varying a few parameters, we can achieve undirected or directed bundling and several bundling styles (FDEB, HEB, small-world, and schematic) and shading effects (flat, emphasizing outlier edges, and tube-like) with a single implementation. All bundling parameters, except the image resolution, can be controlled *locally*, by simply making them a function of the data values at any edge sampling point (site) or neighborhood thereof. This way, we can create a rich family of bundling variations. For example, we can set the kernel radius p_R as a function of the parametric site coordinate t to control the bundles' shapes; or we can set the directional compatibility κ as a function of edge attributes, to achieve data-driven bundling. Such variations require only minimal code changes to CUBu and incur no performance penalty, as CUBu treats all sites independently and in parallel.

5.4 Limitations

While fast, generic, and highly configurable, CUBu has a few limitations. Like all other bundling methods, its bundles are not fully controllable in terms of *exact* shape and position. Interpreting such bundles should thus be done with care, especially when spatial positions are important. To mitigate this, CUBu adds bundle relaxation [21], which allows users to interactively interpolate between bundled and original edges. Separately, the design of *effective* bundle shapes is clearly application-dependent. The styles shown in Sec. 3.1 are just a sample subset which does not claim to be generally optimal nor exhaustive. Specific applications may need different bundle styles. Such styles are easy to get by using other suitable edge profiles (Eqn. 7) and/or edge similarity functions (Eqn. 9). A final issue regards the variation of bundled layouts: Since CUBu can bundle large graphs in real-time, it means that its users can easily vary any of the bundling parameters to interactively generate a wide range of bundled layouts. While this is a powerful tool, it can also potentially confuse users during the exploration of the bundling parameter-space. Now that real-time bundling parameter control is possible, a next research step should consider how to present this parameter-space to users so as to enable intuitive and effective exploration.

6 Conclusions

We have presented CUBu, a general-purpose framework for creating high-quality bundlings from very large graphs. CUBu proposes a GPU-based design that addresses the main desirable features of existing bundling algorithms (scalability, directional bundling, level-of-detail visualization of bundled results) in a single unified framework. CUBu is 50 to 100 times faster than state-of-the-art bundling methods, thereby opening the door to real-time bundling of graphs of millions of edges. Separately, CUBu can produce bundling styles similar to a wide variety of existing graph visualization algorithms, such as hierarchical edge bundling, skeleton-based edge bundling, force-directed edge bundling, schematic graph drawing, image-based edge bundles, and dynamic-graph bundling. We compare CUBu with seven related bundling algorithms and show its scalability and generality on several graphs and trail-sets up to one million edges.

As future work, we aim to adapt CUBu to address graph and trail-like datasets generated by additional application domains, such as diffusion tensor imaging (DTI) bundling [4] and road-and-maritime traffic datasets [42]. Separately, we aim to extend our image-based bundle shading to visualize several per-edge and per-node attributes, in the direction of supporting multivariate graph visualization. Studying how CUBu can integrate additional bundling

styles such as the ones described in [33] is also an interesting topic for future work.

As CUBu largely solves the scalability challenge of generating bundled drawings of large graphs at interactive rates, a separate important task for future work is to use this algorithm in the construction of interactive exploratory visualizations for such graphs. This way the main added-value of the scalable bundling and various drawing styles provided by CUBu can be optimally leveraged to solve real-world problems involving such graphs.

REFERENCES

[1] ADS-B. Automatic dependent surveillance broadcast, 2014. www.ads-b.com.

[2] D. Auber, Y. Chiricota, F. Jourdan, and G. Melan con. Multiscale visualization of small world networks. In *Proc. IEEE Infovis*, pages 75–81, 2003.

[3] R. Boardman. Bubble trees: The visualization of hierarchical information structures. In *Proc. ACM CHI*, pages 315–316, 2000.

[4] J. Böttger, A. Schäfer, G. Lohmann, A. Villringer, and D. Margulies. Three-dimensional mean-shift edge bundling for the visualization of functional connectivity in the brain. *IEEE TVCG*, 20(3):471–480, 2014.

[5] T. Cao, K. Tang, A. Mohamed, and T. Tan. Parallel banding algorithm to compute exact distance transform with the GPU. In *Proc. ACM SIGGRAPH Symp. on Interactive 3D Graphics and Games*, pages 134–141, 2010.

[6] L. Cayton. A nearest neighbor data structure for graphics hardware. In *Proc. ADMS*, pages 192–197, 2010. people.kyb.tuebingen.mpg.de/lcayton.

[7] D. Comaniciu and P. Meer. Mean shift: A robust approach toward feature space analysis. *IEEE TPAMI*, 24(5):603–619, 2002.

[8] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *J. Sys. & Software*, 81(12):2252–2268, 2008.

[9] W. Cui, H. Zhou, H. Qu, P. Wong, and X. Li. Geometry-based edge clustering for graph visualization. *IEEE TVCG*, 14(6):1277–1284, 2008.

[10] S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[11] G. Ellis and A. Dix. A taxonomy of clutter reduction for information visualisation. *IEEE TVCG*, 13(6):1216–1223, 2007.

[12] O. Ersoy, C. Hurter, F. Paulovich, G. Cantareiro, and A. Telea. Skeleton-based edge bundles for graph visualization. *IEEE TVCG*, 17(2):2364–2373, 2011.

[13] A. Frank and A. Asuncion. UCI machine learning repository, 2013.

[14] E. Gansner, Y. Hu, S. North, and C. Scheidegger. Multilevel agglomerative edge bundling for visualizing large graphs. In *Proc. PacificVis*, pages 187–194, 2011.

[15] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using GPU. In *Proc. Intl. Workshop on Computer Vision on GPU (CVGPU)*, pages 77–83, 2008.

[16] Z. H, X. Yuan, H. Qu, W. Cui, and B. Chen. Visual clustering in parallel coordinates. *CGF*, pages 1324–1332, 2008.

[17] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE TVCG*, 12(5):741–748, 2006.

[18] D. Holten and J. J. van Wijk. Force-directed edge bundling for graph visualization. *Computer Graphics Forum*, 28(3):670–677, 2009.

[19] C. Hurter, S. Conversy, D. Gianazza, and A. Telea. Interactive image-based information visualization for aircraft trajectory analysis. *Transportation Research Part C: Emerging Technologies*, 48, 2014.

[20] C. Hurter, O. Ersoy, S. Fabrikant, T. Klein, and A. Telea. Bundled visualization of dynamic graph and trail data. *IEEE TVCG*, 20(8):1141–1157, 2014.

[21] C. Hurter, O. Ersoy, and A. Telea. MoleView: An attribute and structure-based semantic lens for large element-based plots. *IEEE TVCG*, 17(12):2600–2609, 2011.

[22] C. Hurter, O. Ersoy, and A. Telea. Graph bundling by kernel density estimation. *Computer Graphics Forum*, 31(3):435–443, 2012.

[23] C. Hurter, B. Tissoires, and S. Conversy. FromDaDy: Spreading data across views to support iterative exploration of aircraft trajectories. *IEEE TVCG*, 15(6):1017–1024, 2009.

[24] P. Joia, D. Coimbra, J. A. Cuminato, F. V. Paulovich, and L. G. Nonato. Local affine multidimensional projection. *IEEE TVCG*, 17(12):25632571, 2011.

[25] T. Klein, M. van der Zwan, and A. Telea. Dynamic multiscale visualization of flight data. In *Proc. VISAPP*, pages 232–240, 2014.

[26] A. Lambert, R. Bourqui, and D. Auber. 3D edge bundling for geographical data visualization. In *Proc. Information Visualisation*, pages 329–335, 2010.

[27] A. Lambert, R. Bourqui, and D. Auber. Winding roads: Routing edges into bundles. *Computer Graphics Forum*, 29(3):432–439, 2010.

[28] O. Lampe and H. Hauser. Interactive visualization of streaming data with kernel density estimation. In *Proc. IEEE PacificVis*, pages 232–239, 2011.

[29] R. Martins, D. Coimbra, R. Minghim, and A. Telea. Visual analysis of dimensionality reduction quality for parameterized projections. *Computers & Graphics*, 41:26–42, 2014.

[30] F. McGee and J. Dingliana. An empirical study on the impact of edge bundling on user comprehension of graphs. In *Proc. AVI*, pages 620–627, 2012.

[31] D. Moura. 3D density histograms for criteria-driven edge bundling, 2015. *arXiv:1504.02687v1* [cs.GR].

[32] Q. Nguyen, P. Eades, and S.-H. Hong. StreamEB: Stream edge bundling. In *Proc. Graph Drawing*, pages 324–332. Springer, 2012.

[33] Q. Nguyen, S.-H. Hong, and P. Eades. TGI-EB: A new framework for edge bundling integrating topology, geometry, and importance. In *Proc. Graph Drawing*, pages 123–235, 2011.

[34] M. Nollenburg and A. Wolff. Drawing and labeling high-quality metro maps by mixed-integer programming. *IEEE TVCG*, 17(5):626–641, 2010.

[35] F. Paulovich, D. Eler, J. Poco, C. Botha, R. Minghim, and L. G. Nonato. Piecewise Laplacian-based projection for interactive data exploration and organization. *Computer Graphics Forum*, 30(3):1091–1100, 2011.

[36] F. V. Paulovich, L. G. Nonato, R. Minghim, and H. Levkowitz. Least square projection: A fast high precision multidimensional projection technique and its application to document mapping. *IEEE TVCG*, 14(3):564–575, 2008.

[37] F. V. Paulovich, C. Silva, and L. G. Nonato. Two-phase mapping for projecting massive data sets. *IEEE TVCG*, 16:12811290, 2010.

[38] V. Peysakhovich, C. Hurter, and A. Telea. Attribute-driven edge bundling for general graphs with applications in trail analysis. In *Proc. IEEE PacificVis*, 2015.

[39] PlaneFinder. Live flight status tracker, 2014. http://planefinder.net.

[40] S. Pupyrev, L. Nachmanson, S.Bereg, and E. Holroyd. Edge routing with ordered bundles. In *Proc. Graph Drawing*, pages 136–147, 2012.

[41] D. Reniers, L. Voinea, O. Ersoy, and A. Telea. The Solid* toolset for software visual analytics of program structure and metrics comprehension: From research prototype to product. *Science of Computer Programming*, 79(1):224–240, 2014.

[42] R. Scheepens, N. Willems, H. van de Wetering, G. Andrienko, N. Andrienko, and J. J. van Wijk. Composite density maps for multivariate trajectories. *IEEE TVCG*, 17(12):2518–2527, 2011.

[43] D. Selassie, B. Heller, and J. Heer. Divided edge bundling for directional network data. *IEEE TVCG*, 19(12):754–763, 2011.

[44] C. Silva, F. Paulovich, and L. G. Nonato. User-centered multidimensional projection techniques. *Comput. Sci. Eng.*, 14(4):74–81, 2012.

[45] J. Stott, P. Rodgers, J. Martinez-Ovando, and S. Walker. Automatic metro map layout using multicriteria optimization. *IEEE TVCG*, 17(1):101–114, 2011.

[46] A. Telea and O. Ersoy. Image-based edge bundles: Simplified visualization of large graphs. *Computer Graphics Forum*, 29(3):543–551, 2010.

[47] A. Telea, H. Hoogendorp, O. Ersoy, and D. Reniers. Extraction and visualization of call dependencies for large C/C++ code bases: A comparative study. In *Proc. IEEE VISSOFT*, pages 81–88, 2009.

[48] A. C. Telea. *Data visualization – Principles and Practice, 2^{nd} edition*. CRC Press, 2014.

[49] F. van Ham and J. J. van Wijk. Interactive visualization of small world graphs. In *Proc. IEEE Infovis*, pages 199–206, 2004.

[50] F. van Ham and M. Wattenberg. Centrality based visualization of small world graphs. *CGF*, 27(3):972–980, 2008.

[51] R. van Liere and W. de Leeuw. GraphSplatting: Visualizing graphs as continuous fields. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):206–212, 2003.

[52] J. J. van Wijk and H. van de Wetering. Cushion treemaps: Visualization of hierarchical information. In *Proc. IEEE InfoVis*, pages 73–78, Los Alamitos, CA, 1999. IEEE Press.

[53] H. Zhou, P. Xu, Y. Xiaoru, and Q. Huamin. Edge bundling in information visualization. *Tsinghua Sci. Tech.*, 18(2):148–156, 2013.

[54] M. Zinsmaier, U. Brandes, O. Deussen, and H. Strobelt. Interactive level-of-detail rendering of large graphs. *IEEE TVCG*, 18(12):2486–2495, 2012.

Matthew van der Zwan received his B.Sc. in Computer Science and his B.Sc. in Applied Mathemathics from the University of Groningen, the Netherlands, in 2009. In 2011, he received his M.Sc. in Computer Science from the same university, where he currently is a Ph.D. student. His research interests are in computer vision, focussing on tracking applications, but also in visualization and visual analytics.

Valeriu Codreanu Valeriu Codreanu received his PhD in Electrical Engineering from the Polytechnic University of Bucharest in 2011. He next held postdoctoral positions at the University of Groningen (2011-2014) and Eindhoven University of Technology (2014-2015). He currently works as a HPC consultant at SURFsara, the Dutch national supercomputing center. His research interests cover the theory, architecture, and programming of high-performance and energy-efficient computing systems.

Alexandru Telea received his PhD (2000) in Computer Science from the Eindhoven University of Technology, the Netherlands. Until 2007, he was assistant professor in visualization and computer graphics at the same university. Since 2007, he is professor of computer science at the University of Groningen, the Netherlands. His interests include 3D multiscale shape processing, scientific and information visualization, and software analytics.