Interactive 3D Force-Directed Edge Bundling

Category: Research

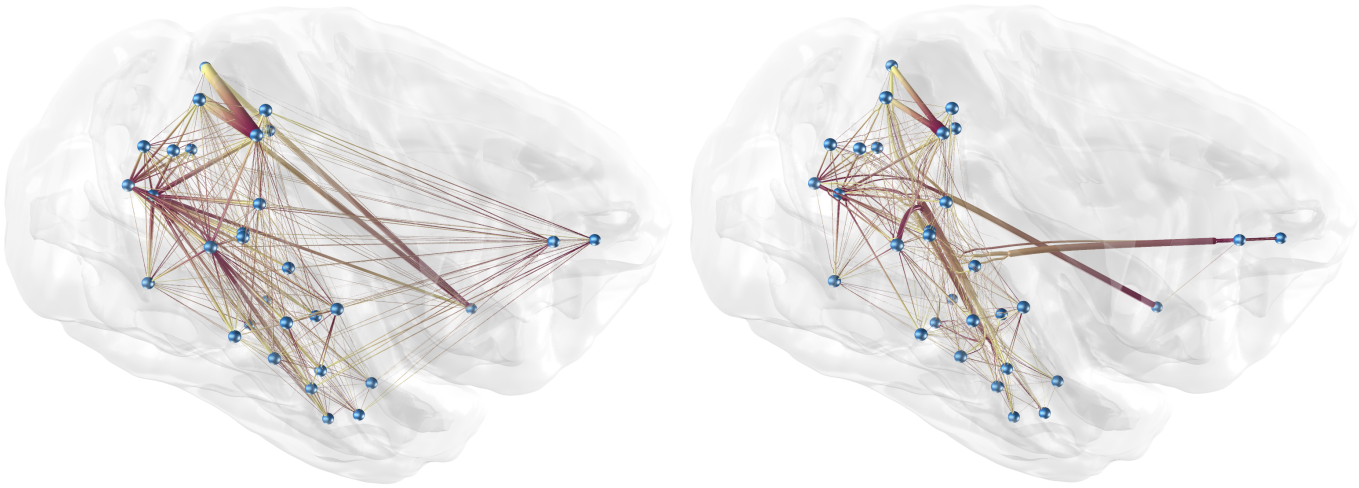

Fig. 1: Node-link diagram of an almost fully connected, bidirectional graph, originating from a NEST simulation based on a macaque's brain [14]. This images depicts 32 each of which represents a brain region. The edges are the regions' interconnectivity. **Left:** original graph; **Right:** the same graph after edge bundling; the edges are directed from purple to yellow.

Abstract—Interactive analyis of 3D relational data is challenging. A common way of representing such data are node-link diagrams as they support analysts in achieving a mental model of the data. However, naïve 3D depictions of complex graphs tend to be visually cluttered, even more than in a 2D layout. This makes graph exploration and data analysis less efficient. This problem can be addressed by edge bundling. We introduce a 3D cluster-based edge bundling algorithm that is inspired by the force-directed edge bundling (FDEB) algorithm [17] and fulfills the requirements to be embedded in an interactive framework for spatial data analysis. It is parallelized and scales with the size of the graph regarding the runtime. Furthermore, it maintains the edge's model and thus supports rendering the graph in different structural styles. We demonstrate this with a graph originating from a simulation of the function of a macaque brain.

Index Terms—3D Information visualization, graph visualization, edge bundling, clustering, physical simulation.

1 Introduction

A graph is an ubiquitous data structure, which describes relational data and is often visually inspected in its representation as node-link diagram. Ware and Mitchell showed [31] that when enriched with the appropriate depth cues, such as provided by immersive virtual environments, graphs laid out in 3D, support data analysis. This is especially true for graphs with a natural spatial embedding, for example brain region connectivity data [1, 4, 5], where a reduction to 2D results in a loss of information. However, when laid out in 3D, graphs are prone to visual clutter, even more than in a 2D layout. This makes graph exploration and data analysis less efficient. Aside from methods that change the positioning of vertices, like in [11], this problem can be addressed by methods changing the course of the edges. The latter can be classified into methods that perform local changes, such as EdgeLense [33] or Edge Plucking [32], and global methods such as edge bundling [15, 26]. Edge bundling is a method that combines geometrically close edges into bundles, which use much less screen space. This work introduces a 3D cluster-based edge bundling algorithm that is inspired by the force-directed edge bundling (FDEB) algorithm [17] and fulfills the requirements to be embedded in an interactive framework for spatial data analysis.

Interactivity, on the one hand, imposes the need of keeping the system's response time within 100ms [24], while navigating and interacting with the visual representation of the graph and updating it at 30 frames per second or even more frequent when used with current 3D image projection devices. On the other hand, it requires algorithms to be fast enough so that they can be added to the workflow of an analyst. The requirement due to runtime is hard to quantify, since this is affected by several conditions. A duration in the order of a few seconds is acceptable [24], until it does not have to be performed every few seconds.

Furthermore, the algorithm should be applicable to general graphs. This enables the possibility to be embedded into a general analysis framework. Moreover, special graphs, like weighted and directed ones, should be supported without restrictions. For this purpose, the presented approach is meant to maintain the model, or rather an explicit geometry of the graph, as this allows applying the same visual representation to the graph when unbundled.

Our main contributions are a native 3D edge bundling algorithm that is optimized for interactive data analysis. Furthermore, it offers flexible rendering styles based on an explicit bundle topology. Finally, the algorithm scales with the graph size regarding runtime.

The rest of the paper is structured as follows. First, we propose an edge cluster-based (see Section 3.1) edge bundling algorithm (see Section 3.2). Within this algorithm, the bundles' topology is maintained, which supports rendering the graph in different structural styles (see Section 3.3). Furthermore, the algorithm is parallelized (see Section 3.4) and scales with the size of the graph regarding the runtime (see Section 4.1). This allows the algorithm to be embedded in an interactive framework for spatial data analysis (see Section 4.2). Furthermore, we discuss our approach and present future work in Section 5. Finally, we conclude our approach in Section 6.

ANONYMIZED

2 Related Work

The representation of relational data as node-link diagram is prone to visual clutter. Holten et al. [15, 17] introduced the approach of merging geometrically close edges to bundles with the objective of reducing edge clutter for general graphs. For this purpose, their force-directed edge bundling algorithm (FDEB) attaches spring and electrostatic forces to segmentation points of the edges that then attract each other. This algorithm works very well for smaller graphs and is generalizable to 3D, but does not scale regarding runtime [3], as it has a quadratic runtime in the number of edges. Selassie et al. extend this algorithm to work better on directed graphs since antiparallel bundles are explicitly laid side by side [29].

Another class of edge bundling approaches implicitly bundles edges together by routing them through nearby, static control points [8, 21, 27]. These points are obtained by a regular grid or a mesh geometry, which is generated from the graph's structure. These approaches avoid the costly edge-to-edge comparisons but require a healthy geometry generation. If the density of points is too low, edges are sharply bent, or in the opposite case, are not bundled at all. Additional challenges arise with the extension to 3D. For example, there are a lot more possible routes for the edges to take and it is more difficult to aid them in taking a common one. However, for a limited three-dimensional case, i.e., with the vertices and edges bound to the surface of a sphere, Lambert et al. [20] showed that it is possible, in general.

A number of recent approaches take advantage of the pixel pipelines of today's GPUs [9, 19, 30]. This enables massively parallel processing in the pixel space. These image-based techniques are to some extent similar to the ones discussed before, as the result of the GPU accelerated calculations, e.g., a density field, can be described as a set of control points, even if equipped with a weight and an expansion in 2D space. These approaches are very fast, but in addition to the problems of the geometry-based techniques, they are even more difficult to adapt to 3D, since when the view is changed, the complete edge bundling has to be recomputed. Depending on the size of the graph, this still could be possible in real time. However, without major changes in the algorithms, the resulting bundling depends on the view, which would be very confusing for the user, as it would be hard to obtain a consistent mental model of the graph. But simultaneously, view changes are very common in interactive applications. As mentioned in the introduction, graphs laid out in 3D support data analysis [31], such as provided by immersive virtual environments. But, in projection setups like CAVEs, [7] the user usually does not look orthogonally at the projection screens, or even worse, simultaneously looks in different angles on different projection screens. This would lead to artifacts at the borders while using image based techniques. In summary, it is very difficult to handle image space techniques for edge bundling in an interactive 3D application.

Böttger et al. [4, 5] were the first to construct an algorithm explicitly adapted to full 3D. It combines methods from FDEB and kernel density estimation edge bundling (KDEEB) [19]. They compute all pairwise edge compatibilities and then move an increasing count of support points to a weighted mean of the surrounding points. However, the authors stated that their approach is not in general able to process weighted graphs due to hardware limitations.

Gansner et al. designed Multilevel Agglomerative Edge Bundling for Visualizing Large Graphs (MINGEL) consisting of two components [12]. First, they reduced the complexity of the problem by clustering the edges and building a proximity graph. Second, the decision to combine a pair of neighbored edges is taken by calculating if there is a saving of "ink" in this case. They use the quantity of ink as a metaphor for the used pixel space and this again is a metric for display clutter. Our algorithm also follows a two-stage approach. But in the first step, it is based on the combination of a more drastic separation of edge clusters, that are calculated in advance, and in the second step, we use a variant of the original FDEB algorithm for the edge bundling instead. This produces smoother edges, which is important as you want the user to be able to track the shape of connections without being distracted by sharp bends fetching the attention.

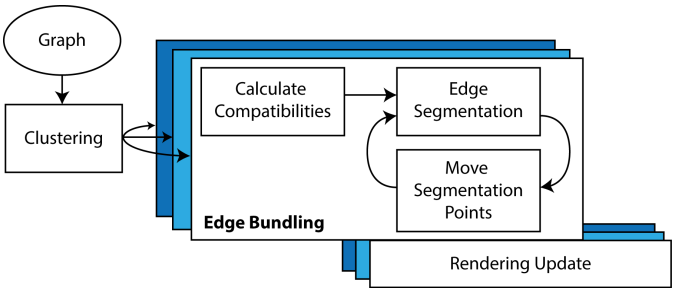

Fig. 2: Edge bundling pipeline. The edges are clustered and then, per cluster task-parallel, bundled and drawn.

3 Method

In this paper, we define a Graph G as an ordered tuple (V,E), where $V \in \mathbb{R}^3$ is a finite set of vertices and $E \subseteq V \times V$ a set of edges. Without loss of generality, we assume that an edge e is an ordered tuple (u,v), with $u,v \in V$, as we can order u and v in case of an undirected graph, e.g., by the order of their occurrence in the underlying data structure. So, in case of a directed as well as of an undirected graph, we can identify an edge's origin, destination, and direction. Thus, for a given $e = (u,v)$, we define $\vec{e} = v - u$. For the purpose of drawing, an edge is described as a set of points $e = \{e^{(i)}\}_{0 \le i \le k}$, with $k \in \mathbb{N}$. Furthermore let $n = |V|$ and $m = |E|$.

To avoid the complete edge-to-edge comparisons and reduce the runtime complexity, we take advantage of the fact already stated by Holten et al. [17] that only very few edges really influence the positioning of any given one. Thus, we break down the edge bundling into a two step process (see Fig. 2). In Section 3.1 we describe how the edge population is divided into clusters, without using a full pairwise comparison approach. From these edge clusters, subgraphs are created and a modified FDEB algorithm is calculated in parallel on those, which is described in Section 3.2. While these calculations are still running, the rendering takes place to provide the user with intermediate results (see Section 3.3). Here, various rendering styles are offered to the user. For example, a way of reasonably drawing weighted edges, or bundles, is presented which is challenging with holding just an implicit model of the bundled graph and is therefore usually not considered by other approaches.

3.1 Edge Clustering

Most of the runtime of FDEB originates from the calculations of the compatibility for every pair of edges and of the force every single edge exerts on the current one. Notably, the compatibility and consequently the forces for most of the pairs are in most cases almost zero. Therefore, we cluster edges with high compatibility in advance and only do further calculations within these clusters. To measure the necessary edge similarity, we define some basic edge metrics. These metrics are inspired by the ones used to calculate the pairwise edge compatibility in FDEB, such as edge length corresponding to the scale from one edge to the other

$$\|v-u\| \in \mathbb{R},$$

the edge's gradient corresponding to the angle between two edges

$$\left(\frac{(v_i-u_i)}{\|v-u\|}\right)_{1\le i\le 3} \in \mathbb{R}^3, \text{ with } v=(v_1,v_2,v_3)^T$$

and the edge's position

$$\frac{u+v}{2} \in \mathbb{R}^3.$$

These metrics are composed of very basic geometric measures and can be complemented or replaced by more application-specific ones, e.g., metrics describing graph topology or edge weights if it is desired that bundled edges share, e.g., common origin/target vertices or only similar-weighted edges should be laid together. It is important to note

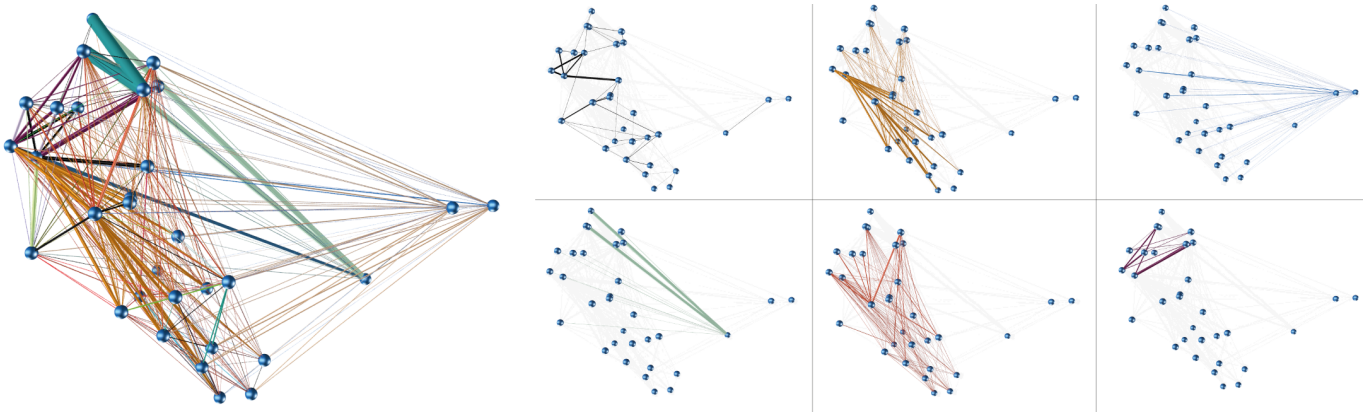

Fig. 4: Graph from Fig. 1 shown with color-coded clusters consisting of similar edges; black edges are unclustered, i.e., not similar to any other. **Left:** Complete graph; **Right:** 6 example clusters, where the upper left depicts not a cluster, but all unclustered edges.

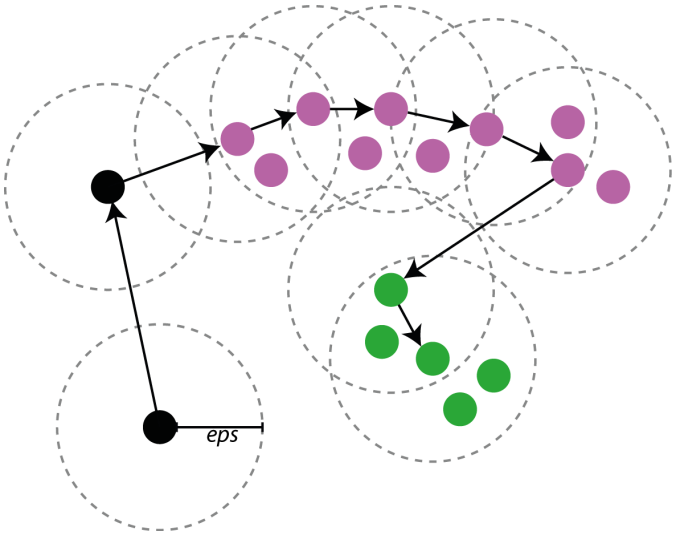

Fig. 3: Example of a DBSCAN cluster calculation [10], with minimal cluster size is 2. The dashed circles, each framing one data point, depict the destiny parameter *eps*. Whenever a data point falls within the radius of any existing point of the current cluster, it is added recursively. The result is a green and a purple cluster. The two black data points are marked as unclustered.

that the chosen metrics should not refer to pairs of edges but instead are computable per edge. Otherwise the quadratic complexity of FDEB just would be moved to a preprocessing step. In most cases they do not have the same expressiveness as their compatibility counterparts, but it is completely sufficient to find a preselection. For example, the angle between two edges is more meaningful than comparing their gradient. This means that the found clusters do not have to be perfect, but a superset of similar edges, as the connected edge bundling algorithm will do the precise work in the following.

This basic set of the three metrics finally specifies a seven-dimensional feature vector for every edge. After normalizing every component to the interval $[0,1]$, these vectors are inserted into an R*-tree [2], which enables an efficient access to this multi-dimensional data with spatial queries. Compared with an R-tree, an R*-tree guarantees a better aligned indexing within a longer setup time but faster access during runtime. For the clustering we use DBSCAN [10], a density-based clustering algorithm, whose parameter for the minimal cluster size is set to a value of 2. This is because we want to bundle sets of similar edges starting with a size of 2. The number of clusters, or the similarity of edges, necessary to form a cluster is determined by the density parameter *eps*. It is a threshold for deciding if two data points, according to their Euclidean distance in the parameter space, are close enough to be assigned to a common cluster (see Fig. 3). A default *eps* value is chosen by first precomputing an interval of reasonable values $[l,u]$, starting with a lower bound where most of the edges first become assigned to any cluster and an upper bound where the result is only one cluster. Finally the value is set to $l+(u-l)/3$ as this often has turned out to be a good choice in our experiments, which means there is no loss of bundling quality while a good runtime performance is achieved. However, always computing the "right" density parameter for the clustering algorithm is not feasible. On the one hand, it could be very different for two graphs, and a good parameter selection strongly depends on the current analysis task on the other hand. But since the clustering is fast, it allows us to put the user in the loop of interactively choosing or changing the value on demand within the precalculated interval (see Section 4.2). After selecting an *eps* value, a color-coded visual representation of the resulting clusters is instantly shown (cf. Fig. 4).

Now, out of the resulting clusters we create new subgraphs, whereas from the implementation view they only hold references to the edges of the original one, so that changes in the subgraphs propagate to the root graph. Note that the subgraphs are not necessarily connected, even if the parent graph was. They are processed by a modified FDEB algorithm, described in the next section, below.

3.2 Edge Bundling

Our bundling strategy is based on the force directed edge bundling algorithm described by Holten and van Wijk [17]. This approach merges geometrically similar edges to common bundles. For this purpose, it attaches spring and electrostatic forces to segmentation points upon the edges that mutually attract each other (see Fig. 5). We made major changes and extensions to the algorithm to achieve especially three goals:

(G1) *De-parameterization*
Of course it could be asked why especially an approach with the aim to be interactive wants to reduce parameters, as user involvement with direct feedback is a good method to find suitable parameters. However, it should be kept in mind that edge bundling is a tool for supporting data analysis and the user should need as little time as possible to tweak algorithm parameters.

(G2) *More stable force system*
The simulation of forces is often very fragile, e.g., to numerical instability. We want to clear out the problems we identified.

(G3) *Increase computational efficiency*
This goal addresses the desired interactivity of the edge bundling.

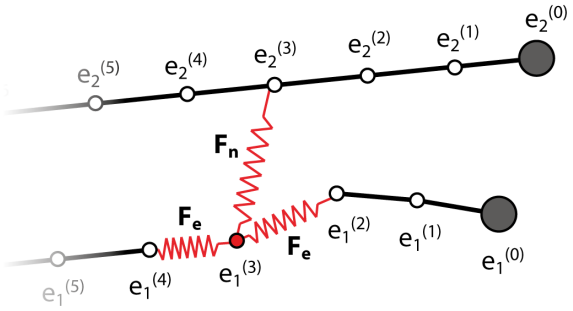

Fig. 5: Segmented edges e_1 and e_2. Example force calculation for the 3rd segmentation point of e_1, assumed there is only one other relevant edge e_2: $F(e_1^{(3)}) = F_e(e_1^{(3)}) + F_n(e_1^{(3)})$. All other points are static. Figure based on [17].

In the initialization step, the compatibilities for all edge pairs are calculated with more precise metrics than within the clustering and stored (**G3**, cf. Algorithm 1, line 1). They are needed later, as it depends on the comparability of two edges to which extent they attract each other. It is possible to compute them now, as the comparability measurements only depend on the origin and target point of the edges, as described below, which are assumed not to change during the algorithm. If it is required to change single vertex positions during the simulation, a first naïve solution to this would be to restart the bundling for the affected clusters. If the changes provoke changes even in the clustering, the computations have to be restarted.

The comparability criteria start with the angle between two edges $C_a(e_1,e_2)$, with

$$C_a(e_1,e_2) = \left| \frac{\vec{e_1}\cdot\vec{e_2}}{\|\vec{e_1}\|\cdot\|\vec{e_2}\|} \right| \in [0,1].$$

The second one describes the scale between two edges $C_s(\vec{e_1},\vec{e_2})$ with,

$$C_s(e_1,e_2) = \frac{\max(\|\vec{e_1}\|,\|\vec{e_2}\|)}{\min(\|\vec{e_1}\|,\|\vec{e_2}\|)} \in (0,1].$$

At last, we rate the positioning to each other $C_p(\vec{e_1},\vec{e_2})$ with,

$$C_p(e_1,e_2) = \frac{\frac{\vec{e_1}+\vec{e_2}}{2}}{\frac{\vec{e_1}+\vec{e_2}}{2} + \|mid(e_2) - mid(e_1)\|} \in [0,1],$$

$$\text{with } mid(e) = \frac{e^{(0)}+e^{(k)}}{2}.$$

We intentionally leave out the so called visibility criterion, described by Holten et al. [17], as we argue that it is only a weighted combination of the other criteria. The reason for adding this fourth metric are edges that are comparable with respect to the other metrics, but not comparable in the intended sense. As an example, the opposite edges of a skewed parallelogram are mentioned. However, in our opinion this already is covered by the position metric.

Now, the total comparability $C(e_1,e_2)$ is calculated as follows:

$$C(e_1,e_2) = C_a(e_1,e_2)\cdot C_s(e_1,e_2)\cdot C_p(e_1,e_2) \in [0,1]$$

After the initialization, a number of cycles c are simulated (see Algorithm 1, line 2). At the beginning of a cycle, every edge is subdivided by adding new segmentation points, refining the resulting curves (line 3). With every cycle, the number of points is doubled by evenly distributing the new ones along the risen edge line segments and simultaneously dropping the old ones. Within these cycles, we avoid an explicit number of iterations but instead iterate until the total energy in the system does not further decrease. This eliminates a parameter

Algorithm 1: Edge Bundling

Data: $G = (\text{Vertices}, \text{Edges})$,
numOfCycles,
maxDisp *// maximum Displacement*

```
1  comp = CalcComparabilityMap(Edges);
2  for numOfCycles do
3      DoEdgeSegmentation(Edges);
4      while newEnergy < oldEnergy do
5          oldEnergy = newEnergy;
6          newEnergy = 0;
7          foreach Edge e do
8              foreach Edge e_x do
9                  foreach segmentation point i do
10                     // e^(i) is the ith segmentation point of e
11                     force = CalcForces(e^(i), e_x^(i)) · comp[e,e_x];
12                     e^(i) = e^(i) + min(maxDisp, force);
13                     newEnergy = newEnergy + force;
14                 end
15             end
16         end
17     end
18 end
```

(G1) and in addition assures that edges are not bundled with different densities **(G2)**, a problem already described by Böttger et al. [4]. Within every iteration, for every segmentation point of any edge, the total force it is affected by is calculated. This consists of two components (see Fig. 5). First, the force F_n holding the segmentation point $e^{(i)}$ in between its neighbors, with spring constant k

$$F_n(e^{(i)}) = k\cdot(\|e^{(i-1)} - e^{(i)}\| + \|e^{(i)} - e^{(i+1)}\|)^2.$$

We changed the force from a linear to a quadratic behavior, which leads to more stable results **(G2)**, as we want to better imitate the behavior of a physical spring and penalize the points in running away much harder. The second force component F_e is the sum over all forces to the corresponding segmentation points of the other edges

$$F_e(e^{(i)}) = \sum_{e_x\in E} \left(C(e,e_x)\cdot\|e_x^{(i)} - e^{(i)}\|\right)^2.$$

Here again we take a squared distance, for the same reasons as before. The total force on an edge segmentation point then is

$$F(e^{(i)}) = F_n(e^{(i)}) + F_e(e^{(i)}).$$

Finally, we move the given point along the force vector by the step size $s_c = s/2^c$, with c the number of the current cycle run. But as an additional difference to FDEB, we cap this value by the length of the total force vector **(G2)**. This prevents an edge being attracted by another edge when they are not similar, as a point originally was moved along the combined force with a defined step size, as long as the force was not exactly 0. From the increasing number of segmentation points, Holten et al. also expect smoothed edges. We think that the bundling, with respect to **G3**, is too expensive for this task and therefore recommend to choose as few cycles as possible and as many as necessary, and to instead smooth the edges in the following rendering step (see Section 3.3).

It should be mentioned that we added the option to perform an ambiguity free variant of the algorithm as described in [22], where only edges with either a common origin or a common target are bundled. This would guaranty that there is no loss of information in the bundled representation, but in general reduces less clutter.

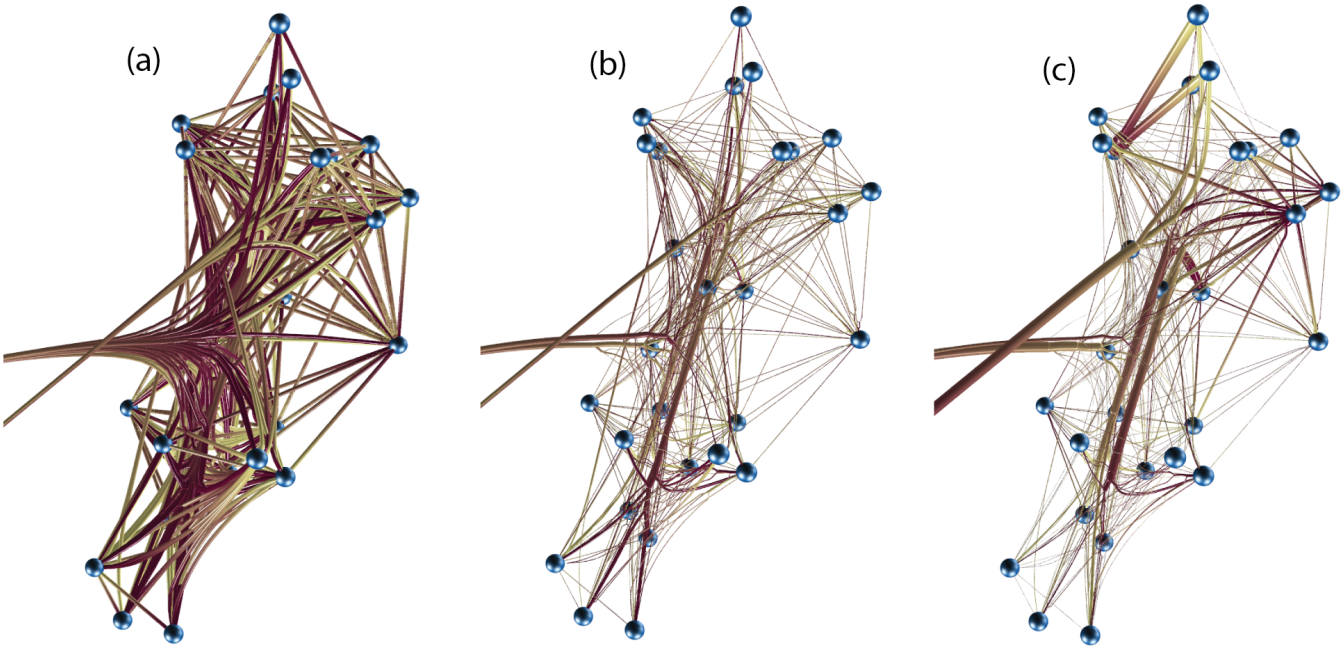

Fig. 7: A section of the graph shown in Fig. 1 depicting different rendering styles: **(a)** edge rendering style, **(b)** bundle rendering style, line width represents the number of edges combined in this bundle, **(c)** bundle rendering style, line width represents the combined edge weight in this bundle.

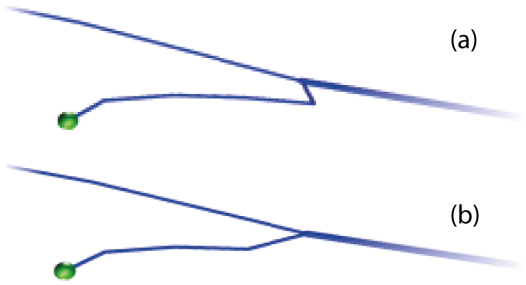

Fig. 6: Two bundles coming from the left become one bundle, with their widths stacked. (a) without an additional smoothing step. (b) with an additional smoothing.

3.3 Rendering

According to the model-view-controller design pattern, we include in our framework a graph view, which renders a graph and consists of a vertex view and an edge view. So a graph view with an empty edge view and any vertex view is created for the parent graph. Normally, the latter draws simple spheres for every vertex. But due to the framework's flexibility, it is possible to add different vertex representations. For example, if the data represents a social network, pictures of the people can be rendered on the position of the vertices, or if the data represents a neural network, cell nuclei can be rendered instead. Furthermore, for every subgraph we create an additional graph view, this time with an empty vertex view. These views are responsible for drawing the edges. We also could add a single edge view upon the parent graph as the subgraphs are holding only references to the edge segmentation points stored in the parent graph. But with the current setup, we have to update much less data on the graphics card when one parallelized edge bundling task (see Section 3.4) has finished and triggers an update of the corresponding edge segments. In addition, keeping the separation helps to reduce computational costs of all following steps.

3.3.1 Edge Rendering Style

During all computations, the graph and the changing edge segments are redrawn every frame with a so-called edge rendering style (see Fig. 7a). Thus, the user is able to interactively explore a largely cleaned graph, while the bundling is being processed. This is positively affected by the observation that the main structural changes, specifically the reduction of edge clutter, take place during the first cycles of the bundling algorithm, whereas the following ones mainly just refine the result.

The edge segmentation points are supplied to the OpenGL pipeline as line segments, where they are rendered as polylines using a geometry shader. In the case of a directed graph, the direction is encoded by the intensity, from dark-to-light according to a study of Holten et al. [16]. Although, a tapered representation is even higher rated, this would interfere with the edge weight coding of the other views (see Section 3.3.2) and for reasons of usability, we prefer to keep it consistent. All together, the chosen representations are just examples and the framework allows to change the representations and even interactively switch between a set of them, further demonstrated in the following (also see Section 4.2).

3.3.2 Bundle Rendering Styles

The result of most edge bundling approaches are lines, or in our case tubes, laying on each other. However, as bundles are only implicit, this means there is neither explicit knowledge about which edges become part of a bundle, nor when or where this happens. This is sufficient for various applications, but not for every one. For instance, the user could be interested in the number of edges forming a bundle, but usually this information is lost. Of course, there are work-arounds, by, for example, drawing edges with an stacking alpha [35] or just beside each other [27]. But as work-arounds they all come with their limitations, as bundles stay implicit and the representation is not exchangeable. Another prominent example are weighted edges. To overcome this issue, we extract the explicit bundling topology based on the segmented edges. For this purpose, we have to decide when sets of edges form bundles and when not. As a result of the quasi-continuous physical simulation, the edge segmentation points do not necessarily lie precisely on top of each other. However, considering single points is

Algorithm 2: Segmentation to explicit Bundles

```
Data: G = (Vertices, Edges),
      // e^(i) is the ith segmentation point of e and
      // k = 2^(cycles-1) - 1
      ∀ Edges e : ∃_1 (e^(0), .., e^(k)),
      // density parameter
      eps
Result: // every bundle is a strip of segments
        Bundles
        // tracks for every edge the bundles it participetes in
        Topology

new segment = (Vector3D, Vector3D);
foreach Edge e do
    segment[0] = e^(0);
    segment[1] = e^(1);
    R*Tree.Add(segment);
end
for i = 1..k do
    Cluster = DoClustering(R*Tree, eps);
    R*Tree.Clear();
    // consider unclustered segments as a set with size 1
    foreach set of segments {s_0, .., s_r} in Cluster do
        // {e_0, .., e_r} corresponding set of Edges
        foreach e ∈ {e_0, .., e_r} do
            UpdateTopology(e);
            segment[0] = GetMeanSegEndPoint({s_0, .., s_r});
            segment[1] = e^(i+1);
            R*Tree.Add(segment);
        end
        if {e_0, .., e_r} were a cluster before then
            ExtendBundle({s_0, .., s_r});
        else
            AddBundle({s_0, .., s_r});
        end
    end
end
```

usually not a good idea, as co-located points do not clearly imply partially co-located edges. The edges could cross each other in this point or could even be antiparallel.

Therefore, we introduce an algorithm that identifies co-located segments, each consisting of a pair of segmentation points (see Algorithm 2). If two segments lie on top of each other, we can be sure that they should be part of a bundle. To find these segments, it is useful to use spatial clustering, so we use the DBSCAN algorithm [10]. Again, we have to decide on the *eps* parameter. For this purpose, we could evenly resample the edges and then calculate the *eps* value so that two directly successive segments are just not detected as lying upon each other. But as long we do not resample with the length of the smallest segment, we could possibly lose existing smoothness. Therefore, we leave that step optional but recommend to use it when there are artifacts in the results. Hence, the algorithm by default just chooses an *eps* value close to the smallest segment. Nevertheless, because of two other conditions described in the following, we should not get too confused with the clustering. First, we are still processing the edge clusters independently, which reduces the number of simultaneously processed edges, but even more important, due to the clustering conditions all included edges are very similar and it is more difficult to get confused with just crossing segments, etc. Second, the previously performed bundling algorithm constrains that only corresponding segmentation points, i.e., with the same index, attract each other. This is favored by the fact that the segmentation points of implicitly bundled edges are very close together. Furthermore, this allows us to systematically cluster the segments in groups from the origin of all edges to the target (cf. Algorithm 2). The latter additionally enables the possibility to keep track of the edges' topology. Thus, the results are not only explicit bundles, but also do not loose the edge semantic, as it is still known that a specific edge consists of these bundles in that sequence.

During the process of replacing sets of segments by bundles, segmentation points have to be averaged to single points in a bundle and as everything needs to be kept connected, although previous points have to be shifted. This could cause unaesthetic bends as illustrated in Fig. 6. Hence, we pipe the results through a simple smoothing. The resulting bundles finally are described by Catmull-Rom splines [6, 28] and supplied to the OpenGL pipeline, where they are rendered using a geometry shader. Due to the fact that we have calculated an explicit representation and kept the edge topology, we are now able to add various data to our representation. We implemented two examples, edge density and weight, stated in the beginning (see Fig. 7). Beyond that, it would be even possible to represent time varying data.

In conclusion, the user has now the opportunity to switch between different views depending on what she wants to analyze (see Section 4.2) or highlight.

3.4 Parallelization

To further increase performance, we additionally parallelized our algorithm on two different layers. For this purpose, we decided on task-based parallelization using Intel® Threading Building Blocks. TBB is initialized to use up to *number of available cores* -1 threads in parallel to exclusively reserve one core for the main thread, which is responsible for holding the frame loop in the target frame rate of 60 frames/s, in our case, because we run different immersive display systems. Everything following is task-based and only in the background mapped to threads by TBB.

Up to now, the edge clustering was only used to partition the data before it is processed by a non-scaling algorithm. But as the available edge clusters (see Section 3.1) are independently processed by the edge bundling (see Section 3.2), it is possible to create a new task for every cluster. Thus, the bundling is parallelized. The same applies for an update of the drawing (see Section 3.3). However, a large variance in cluster sizes causes load-balancing issues in static parallelization.

This leads to the second layer of parallelization within the bundling. Two outer *for* loops of the edge bundling algorithm, each iterating over all edges in a cluster, are parallelized, too. The first one is hidden in the edge segmentation function (cf. Algorithm 1, line 3) and does not need any adjustments as this happens completely independently and therefore no data races are possible. The second one is the outer one iterating over all edges within a cycle (cf. Algorithm 1, line 7). To solve data races here, we buffer the position updates and swap them outside of the loop. Finally, we synchronize the update of the total energy.

Both layers together parallelize the edge bundling and simultaneously solve possible load balancing issues.

4 Results

4.1 Runtime

In bundling an example graph with only 32 vertices but almost 600 edges (see Fig. 1), our algorithm took about 1.36 seconds ($\pm$ 0.036, in 10 measurements) with an *eps* value of 0.125 and 28 identified clusters. Thus, it was about 150 times faster than our basic 3D implementation of the FDEB algorithm, which took about 220 seconds. Without any parallelization and the same parameters our algorithm took about 5,03 seconds ($\pm$ 0.037, in 10 measurements). Measurements were performed on an Intel Xeon E5540 2.53GHz quadcore-processor running Windows 7 with 12 GB of RAM and a GeForce GTX480 graphics card. The whole measurement series was performed on the graph with different density parameters and in relation to the number of resulting clusters, which is shown in Fig. 8. As expected, the plot shows a decreasing runtime with an increasing number of clusters, although the distribution of cluster sizes was not examined here. All bundling results for the chosen *eps* interval are well-shaped and just differ in the degree of bundling.

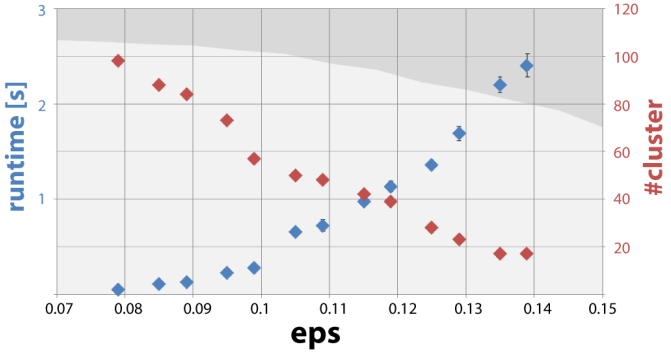

Fig. 8: Dependency of the density parameter *eps* to runtime and number of clusters, within the useful range of *eps* values for this particular graph. This means from, "there is first time a real clustering" to, "clusters start getting that small that there is no more bundling worth mentioning". Additionally in the background the ratio of unclustered edges (dark) to clustered ones (bright) is depicted.

But much more interesting than these raw numbers, as they are only expressive for the explored graph, is the question concerning runtime scalability [3] regarding graph size. Following the processing pipeline (see Fig. 2), first, setting up the R*-tree and clustering all edges costs $O(m \log m)$, with m number of the edges. Unfortunately, due to the changed termination condition within an edge bundling cycle (see Section 3.2), we are no longer able to guarantee any meaningful theoretical runtime complexity. This is because it is theoretically possible that there is an epsilon-small decrease in the total energy with every iteration and so the algorithm does not terminate. Nevertheless, to get an idea of the runtime complexity we could assume a constant number of iterations or a threshold. This would lead to a quadratic runtime with respect to the number of edges in a cluster. When considered together, we get an overall complexity of $O(|C_{max}|^2)$, with C_{max} the largest edge cluster, which in the worst case again is $O(m^2)$. Based on the runtime analysis, the proposed algorithm does not scale in the worst case.

But indeed, with examples that are not specially designed, the algorithm behaves very well. This is because $|C_{max}|$ does not necessarily increases with the size of the input as many graphs and networks in real world applications are sparse and have a low connection. Thus, with increasing size, graphs can be broken down to an increasing number of components. Edges in different components are very likely not similar, regarding the used metrics (see Section 3.1 and 3.2), and therefore would be assigned to different clusters.

In conclusion, graphs with the mentioned properties will not generate larger edge clusters, in general and therefore support a scaling of the proposed algorithm. Of course, these considerations do not hold for all graphs. In Fig. 4 a nearly fully connected, bi-directed graph is shown. With every vertex added to such a graph, the mean size of the clusters will grow, so that the runtime is quadratic, as within the clusters we still operate for every edge on every other one. Nevertheless, the clustering still reduces the problem, as observable in Fig. 4.

4.2 Application

As proof of concept, we embedded our approach in an existing tool for interactive brain analysis, called ANONYMIZED, which visualizes data originated from a NEST simulation [14]. The added part is responsible for presenting spatial, weighted brain region connectivity data (see Fig. 1 and 7). The vertices have static positions, each in the center of the brain region they represent. The application is designed to run in a CAVE-like environment, but it is possible to use it with other VR devices as well as on a desktop PC. A basic set of interaction techniques is given by the application context. First, there is basic navigation and selection. Second, there are some options to directly manipulate the graph, as limiting the depicted edges by their weight, change colors, or move vertices.

We added the possibility to initiate a bundling of the current graph. In addition, it is possible to enable a bundling mode, where with every change in the graph structure, the bundling is automatically refreshed. This method and all options are accessible via an extended pie menu [13]. The usual work flow starts with the clustering of the graph with the precalculated *eps* value (see Section 3.1). The result is color coded as depicted in Fig. 4. The colors are evenly distributed within the CIE L*a*b* color space, which is perceptually uniform [18, 23]. However, currently we are not optimizing the cluster coloring with respect to the clusters' distance to each other. Altogether, the visualization is not sufficient to analyze the clusters in detail, but it is for obtaining a quick overview of whether the calculated clusters are meaningful or not. If this is not the case, the user is able to change the *eps* value via a slider within the precalculated interval (see Section 3.1) and the cluster visualization will update. When satisfied with the clustering, the user starts the edge bundling and simultaneously gets feedback, as the updated positions of the edge's segmentation points are directly drawn. She usually starts inspecting the graph, by rotating and translating her view point, while the bundling is finished. Now it is possible to switch between the rendering styles (see Section 3.3). Depending on the active style, additional parameters can be adjusted, as for example the minimal, maximal, or both line widths relating to an edge parameter. Finally, it is possible to continuously fade between the Edge Rendering Style (see Section 3.3.1) and the unbundled graph, which can be interesting for finding just the right ratio between clutter reduction and information loss through edge bundling.

5 Discussion & Future Work

In the following, we want to discuss single aspects of the presented approach in sequence of their appearance in the pipeline, according to Fig. 2, and start with the clustering of the edges.

For the calculation of edge clusters, we have used a density-based approach, which expects a density parameter. To support the user in choosing the right parameter, we precalculate an interval of meaningful parameters. To avoid patronizing the user, we used a very conservative method for this purpose, starting with a lower bound where most of the edges first become assigned to any cluster and an upper bound where the result is only one cluster. This can lead to the situation that still large parts of the interval are not of interest, e.g., because there is only one big cluster for larger parts of the interval. It may be sufficient to be more restrictive for this part and further analyze the clustering behavior in future.

Then, we took the calculated edge clusters and started an optimized, force-directed edge bundling algorithm on each cluster independently and in parallel.

Furthermore, we described that the clusters are drawn independently, too. This can be an issue with directed graphs and antiparallel bundles, as it is possible that they are drawn at the same position and so cover each other. Although we could change the clustering metrics in a way that these edges fall in one common cluster, there is no applicable solution yet to place those bundles relative to each other in 3D. In a general 2D case, Selassie et al. [29] placed bundles side by side with respect to a highway metaphor. Unfortunately, this metaphor does not work in a three dimensional space, as there is no left and right anymore.

We embedded the proposed method in a data analysis tool and added user interaction, such as the possibility to fade in and out between the bundled and unbundled graph. Some of this interaction could be even more beneficial if it would be possible to perform them only on parts of the graph. These parts can often be expected to semantically correspond to the calculated clusters. The only missing component for a realization is finding a working selection metaphor for the clusters, which is not straightforward as the clusters are usually interwoven or hidden. A first idea to achieve this is to combine a standard selection strategy, with the possibility to select a specific cluster from a list of small multiples, as depicted in Fig. 4 (right). This additionally would support the pure cluster visualization.

We argued that the proposed algorithm should scale for most graphs appearing in data analysis. This statement in future could be supported by an empirical study. Nevertheless, this represents a challenge due

to the sample size. On the one hand, graphs would have to be synthetically created, and on the other, they need to be meaningfully distributed with respect to their properties.

Finally, future work includes investigating the relevance of edge bundling as a misleading factor in inferring paths or structures in the data that are actually not there. For example, axon paths in the brain do not have to follow the paths taken by the bundles, but can be interesting for neuroscientists and for this reason are noticed from another view point as for a specialist from a different domain. Of course, this is not only a risk, but an opportunity too. If, for instance, the real pathways are known, the bundles could be attracted by these.

6 Conclusion

Interactive analysis of 3D relational data is challenging. A common way of representing such data are node-link diagrams as they support analysts in achieving a mental model of the data. However, naïve 3D depictions of complex graphs tend to be visually cluttered. This makes graph exploration and data analysis less efficient. This problem can be addressed by edge bundling, which combines geometrically close edges into bundles. We have presented a native 3D, edge cluster-based and parallel edge bundling algorithm that fulfills the requirements necessary to be embedded in an interactive framework for spatial data analysis. Furthermore, it maintains the topology of the edge bundles and thus supports rendering of the graph in different structural styles. Finally, the proposed algorithm scales in runtime with the number of edges for most of the graphs.

References

[1] B. Alper, B. Bach, N. Henry Riche, T. Isenberg, and J.-D. Fekete. Weighted Graph Comparison Techniques for Brain Connectivity Analysis. *In Proc. of Conference on Human Factors in Computing Systems*, pages 483– 492, 2013.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. *In Proc. of ACM SIGMOD International Conference on Management of Data*, 19:322–331, 1990.

[3] A. B. Bondi. Characteristics of Scalability and Their Impact on Performance. *In Proc. of 2nd ACM International Workshop on Software and Performance*, pages 195–203, 2000.

[4] J. Böttger, A. Schäfer, and G. Lohmann. Three-Dimensional Mean-Shift Edge Bundling for the Visualization of Functional Connectivity in the Brain. *IEEE Transactions on Visualization and Computer Graphics*, 20(3):471–480, 2013.

[5] J. Böttger, R. Schurade, E. Jakobsen, A. Schäfer, and D. S. Margulies. Connexel Visualization: A Software Implementation of Glyphs and Edge-Bundling for Dense Connectivity Data Using BrainGL. *Frontiers in Neuroscience*, 8:15, 2014.

[6] E. Catmull and R. Rom. A Class of Local Interpolating Splines. *Computer Aided Geometric Design*, 74:317–326, 1974.

[7] C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart. The CAVE: Audio Visual Experience Automatic Virtual Environment. *Communications of the ACM*, 35(6):64–72, 1992.

[8] W. Cui, H. Zhou, H. Qu, P. C. Wong, and X. Li. Geometry-Based Edge Clustering for Graph Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1277–84, 2008.

[9] O. Ersoy, C. Hurter, F. V. Paulovich, G. Cantareiro, and A. Telea. Skeleton-Based Edge Bundling for Graph Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2364–73, 2011.

[10] M. Ester, H. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. *In Proc. of ACM Knowledge Discovery and Data Mining*, pages 226–231, 1996.

[11] T. Fruchterman and E. Reingold. Graph Drawing by Force Directed Placement. *Software – Practice and Experience*, 21:1129–1164, 1991.

[12] E. R. Gansner, Y. Hu, S. North, and C. Scheidegger. Multilevel Agglomerative Edge Bundling for Visualizing Large Graphs. *In Proc. of IEEE Pacific Visualization Symposium*, pages 187–194, 2011.

[13] S. Gebhardt, S. Pick, F. Leithold, B. Hentschel, and T. Kuhlen. Extended Pie Menus for Immersive Virtual Environments. *IEEE Transactions on Visualization and Computer Graphics*, 19(4):644–51, 2013.

[14] M. Gewaltig and M. Diesmann. NEST (NEural Simulation Tool). *Scholarpedia*, 2(4):1430, 2007.

[15] D. Holten. Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–8, 2006.

[16] D. Holten and J. J. van Wijk. A User Study on Visualizing Directed Edges in Graphs. *In Proc. of 27th International Conference on Human Factors in Computing Systems*, pages 2299–2308, 2009.

[17] D. Holten and J. J. van Wijk. Force-Directed Edge Bundling for Graph Visualization. *Computer Graphics Forum*, 28(3):983–990, 2009.

[18] R. S. Hunter. Photoelectric Color Difference Meter. *Journal of the Optical Society of America*, 48(12):985, 1958.

[19] C. Hurter, O. Ersoy, and A. Telea. Graph Bundling by Kernel Density Estimation. *Computer Graphics Forum*, 31(3pt1):865–874, 2012.

[20] A. Lambert, R. Bourqui, and D. Auber. 3D Edge Bundling for Geographical Data Visualization. *In Proc. of 14th International Conference Information Visualisation*, pages 329–335, 2010.

[21] A. Lambert, R. Bourqui, and D. Auber. Winding Roads: Routing Edges into Bundles. *Computer Graphics Forum*, 29(3):853–862, 2010.

[22] S.-J. Luo, C.-L. Liu, B.-Y. Chen, and K.-L. Ma. Ambiguity-Free Edge-Bundling for Interactive Graph Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 18(5):810–21, 2012.

[23] K. McLAREN. XIII-The Development of the CIE 1976 (L* a* B*) Uniform Colour Space and Colour-Difference Formula. *Journal of the Society of Dyers and Colourists*, 92:338–341, 1976.

[24] J. Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., 1993.

[25] C. Nowke, M. Schmidt, S. J. V. Albada, J. M. Eppler, R. Bakker, M. Diesmann, B. Hentschel, and T. Kuhlen. VisNEST – Interactive Analysis of Neural Activity Data. *In Proc. of IEEE Symposium on Biological Data Visualization*, pages 65–72, 2013.

[26] D. Phan and L. Xiao. Flow Map Layout. *In Proc. of IEEE Symposium on Information Visualization*, pages 219–224, 2005.

[27] S. Pupyrev, L. Nachmanson, S. Bereg, and A. Holroyd. Edge Routing with Ordered Bundles. *In Proc. of 19th International Conference on Graph Drawing*, pages 136–147, 2011.

[28] I. Schoenberg. *Cardinal Spline Interpolation*. Society for Industrial and Applied Mathematics, 1973.

[29] D. Selassie, B. Heller, and J. Heer. Divided Edge Bundling for Directional Network Data. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2354–63, 2011.

[30] A. Telea and O. Ersoy. Image-Based Edge Bundles: Simplified Visualization of Large Graphs. *Computer Graphics Forum*, 29(3):843–852, 2010.

[31] C. Ware and P. Mitchell. Visualizing Graphs in Three Dimensions. *ACM Transactions on Applied Perception*, 5(1):1–15, 2008.

[32] N. Wong and S. Carpendale. Supporting Interactive Graph Exploration Using Edge Plucking. *Visualization and Data Analysis*, 6495, 2007.

[33] N. Wong, S. Carpendale, and S. Greenberg. EdgeLens: An Interactive Method for Managing Edge Congestion in Graphs. *In Proc. of IEEE Symposium on Information Visualization*, 2003:51–58, 2003.

[34] D. Zielasko, B. Weyers, B. Hentschel, and T. W. Kuhlen. Interactive 3D Force-Directed Edge Bundling on Clustered Edges. *In Poster Abstracts of IEEE Symposium on Information Visualization*, 2014.

[35] M. Zinsmaier, U. Brandes, O. Deussen, and H. Strobelt. Interactive Level-of-Detail Rendering of Large Graphs. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2486–2495, 2012.