Online Dynamic Graph Drawing

Yaniv Frishman and Ayellet Tal

***Abstract*— This paper presents an algorithm for drawing a sequence of graphs online. The algorithm strives to maintain the global structure of the graph and thus the user's mental map, while allowing arbitrary modifications between consecutive layouts. The algorithm works online and uses various execution culling methods in order to reduce the layout time and handle large dynamic graphs. Techniques for representing graphs on the GPU allow a speedup by a factor of up to 17 compared to the CPU implementation. The scalability of the algorithm across GPU generations is demonstrated. Applications of the algorithm to the visualization of discussion threads in Internet sites and to the visualization of social networks are provided.**

***Index Terms*— Graph layout, GPU.**

I. Introduction

Graph drawing addresses the problem of constructing geometric representations of graphs [1]. It has applications in a variety of areas, including software engineering, software visualization, databases, information systems, decision support systems, biology, chemistry and social networks.

Many applications require the ability of *dynamic graph drawing*, i.e., the ability to modify the graph [1]–[3], as illustrated in Figure 1. Sample applications include financial analysis, network visualization, security, social networks, and software visualization. The challenge in dynamic graph drawing is to compute a new layout that is both aesthetically pleasing as it stands and fits well into the sequence of drawings of the evolving graph. The latter criterion has been termed *preserving the mental map* [4] or *dynamic stability* [2].

Most existing algorithms address the problem of offline dynamic graph drawing, where the entire sequence of graphs to be drawn is known in advance [3], [5], [6]. This gives the layout algorithm information about future changes in the graph, which allows it to optimize the layouts generated across the entire sequence. For instance, the algorithm can leave place in order to accommodate a node that appears later in the sequence. In contrast, very little research has addressed the problem of online dynamic graph drawing, where the graph sequence to be laid out is not known in advance [7], [8].

This paper proposes an online algorithm for dynamic layout of graphs. It attempts to maintain the user's mental map, while computing fast layouts that take the global graph structure into account. The algorithm, which is based on force directed layout techniques, controls the displacement of nodes according to the structure and changes performed on the graph. By taking special care in order to represent the graph in a GPU-efficient manner, the algorithm is able to make use of the GPU to significantly accelerate the layout.

This paper makes the following contributions. First, a novel, efficient algorithm for online dynamic graph drawing is presented. It spends most of the execution time on the parts of the graph being modified. Second, it is shown how the heaviest part of the algorithm, performing force directed layout, can be implemented in a manner suitable for execution on the GPU. This allows us to significantly shorten the layout time. For example, incremental drawing of a graph of 32,000 nodes takes 0.704 seconds per layout. Finally, two information visualization applications of the algorithm are presented. The first is the visualization of the evolution over time of discussion threads in Internet sites. In this application, illustrated in Figure 1, nodes represent users and edges represent messages sent between users in discussion forums. The second application is the visualization of the growth of a social network, shown in Figure 9. Here, nodes represent users and edges represent connections between friends.

The rest of the paper is organized as follows. Section II discusses related work. Section III formally defines the problem and gives an overview of key algorithm ideas. Section IV presents the algorithm in detail. Section V discusses our implementation. Section VI presents results. Section VII discusses an application to Internet discussion threads visualization. Section VIII presents an application to the visualization of social networks. Section IX concludes the paper. A preliminary version of this research was presented in [9].

II. Related Work

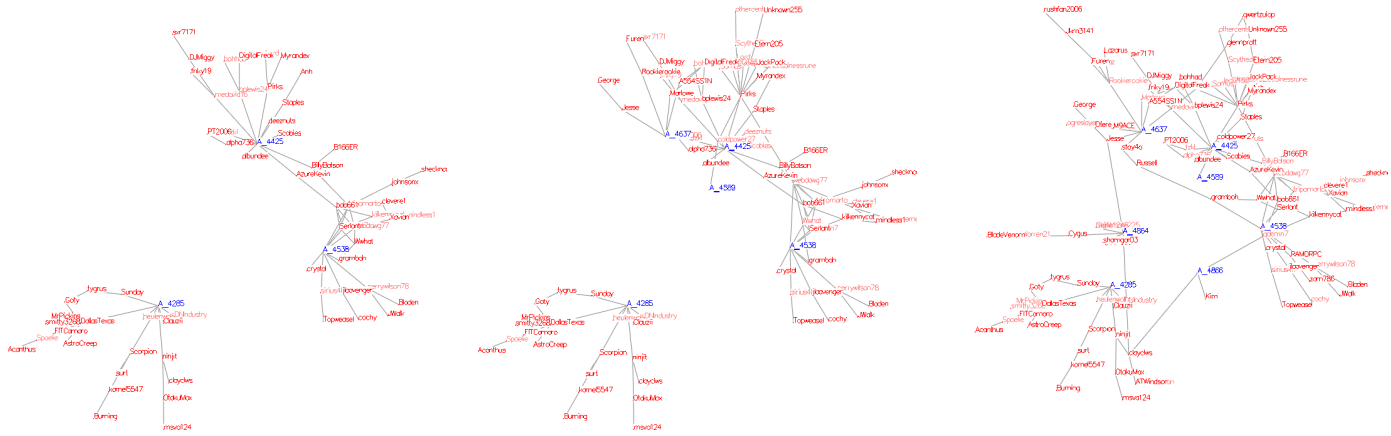
Various methods for graph drawing have been proposed , such as hierarchical, planar, circular, orthogonal, and force directed layout [1], [10]. Our algorithm builds on force directed layout [1], where forces are applied to nodes according to the graph structure and the layout is determined by convergence to a minimum stress configuration. Force directed algorithms are able to produce aesthetic layouts of general graphs, but may be computationally expensive.

Some algorithms have been proposed to perform static force directed layouts of large graphs [11]. In [12] coarser representations of the graph are recursively built using the *edge collapse* operation. The algorithm in [13] coarsens the graph using an approximation of the k-center problem. In [14] a quadtree is used to accelerate the layout. In [15] a maximum independent set filtration is used for coarsening. FM^3 [16] uses a clever $O(N \log N)$ approximation of the all-pairs repulsive forces for N nodes. In [17] a simplified energy function, allowing a more robust mathematical treatment, is used. In [18] a high dimensional embedding of the graph is used.

Several algorithms address the problem of offline dynamic graph drawing, where the entire sequence is known in advance. In [3], a meta-graph built using information from the entire graph sequence, is used in order to maintain the mental map. In [6], a stratified, abstracted version of the graph is used. The nodes are topologically sorted into a tree–like structure (before layout) in order to expose interesting features. An offline force directed algorithm is used in [5] in order to create 2D and 3D animations

Yaniv Frishman is with the Department of Computer Science, Technion - Israel Institute of Technology. E-mail: frishman@tx.technion.ac.il

Ayellet Tal is with the Department of Electrical Engineering, Technion - Israel Institute of Technology. E-mail: ayellet@ee.technion.ac.il



(a) (b) (c)

Fig. 1. Snapshots from the threads1 graph sequence, visualizing discussion threads at http://www.dailytech.com, left to right. Node labels in red show user names, edges link users replying to posted comments. Up to 119 users are shown. Discussion topics, marked as blue A_n nodes, include GPUs (A_4864, A_4285), chipsets (A_4637, A_4425, A_4538 and A_4866) and CPUs (A_4589). A total of 144 messages are visualized.

of evolving graphs. Creating smooth animation between changing sequences of graphs is addressed in [19].

A few algorithms have been proposed to address the online dynamic graph drawing problem, where the graph sequence is not known in advance. An approach based on Bayesian networks is described in [20]. A cost function that takes both aesthetic and stability considerations into account, is defined in [8]. Unfortunately, computing this function is very expensive (45 seconds for a 63 node graph). An algorithm for visualizing dynamic social networks is discussed in [21]. Drawing constrained graphs has also been addressed. Incremental drawing of DAGs (directed acyclic graphs) is discussed in [2]. In [7] dynamic drawing of clustered graphs is addressed. Dynamic drawing of orthogonal and hierarchical graphs is discussed in [22]. The current paper aims at producing online layouts of general graphs efficiently.

In recent years, GPUs have been successfully applied to numerous problems outside of classical computer graphics [23]. Protein folding [24] and simulation of deformable bodies using mass-spring systems [25], [26] are related to our application. However, while the mass-spring algorithms take only nodes connected by edges into account, the force directed algorithm considers all the nodes when calculating the force exerted on a node. GPUs have also been used to simulate gravitational forces [27], where an approximate force field is used to calculate forces. A GPU-based implementation of the MDS (multidimensional scaling) algorithm is discussed in [28]. Accelerating static graph drawing on the GPU has been addressed by several authors [29]–[31]. A GPU accelerated force directed layout algorithm using an Euler method is presented in [30]. Although a very large acceleration is achieved, the complexity of the underlying algorithm is $O(|E| + |V|^2)$ for $|E|$ edges and $|V|$ nodes. In [31] spectral partitioning is used to create a hierarchy of graphs. The focus of the current paper, however, is on creating stable layouts of changing graphs.

III. Overview

Given, online, a series of undirected graphs $G_0 = (V_0, E_0), G_1 = (V_1, E_1), \ldots, G_n = (V_n, E_n)$, the goal of the algorithm is to produce a sequence of layouts $L_0, L_1, \ldots, L_n$, where L_i is a straight-edge drawing of G_i. The updates U_i that can be performed between successive graphs G_{i-1} and G_i, include adding or removing vertices and edges.

A key issue in dynamic graph drawing is the preservation of the mental map, i.e. the stability of the layouts [4]. This is an important consideration since a user looking at a graph drawing becomes gradually familiar with the structure of the graph. The quality of the layout can be evaluated by measuring the movement of the nodes between successive layouts, which should be small, especially in unchanged areas of the graph. In addition, each layout in the sequence should satisfy the standard requirements from static graph layouts, such as minimization of edge crossings, avoidance of node overlaps and layout symmetry [1].

Among the different classes of graph drawing algorithms, the force directed algorithm class [1], [10] is a natural choice in our case, for several reasons. First, different layout criteria can be easily integrated into these algorithms. Second, in some of these algorithms, it is possible to update node positions in parallel, thus making it possible to efficiently employ the GPU's parallel computation model. Finally, it is possible to use a convergence scheme that resembles simulated annealing, in which nodes are slowly frozen into position [32]. This is suitable for use in dynamic layout, where nodes have different scales of movement.

Our algorithm utilizes several key ideas. In order to maintain the mental map, we perform the following. First, nodes are initially placed using local graph properties and information from the previous layout. Second, a movement flexibility degree is assigned to each node, according to the changes in the graph. This allows the algorithm to "focus" on nodes that may have large displacements. Third, an approach similar to simulated annealing is used, where the graph slowly freezes into its final position. Fourth, the changes between graphs are smoothly animated. In order to reduce the layout time while maintaining layout quality, the graph is partitioned so that forces from distant nodes can be approximated, and the GPU is used to accelerate the layout. Moreover, in order to quickly compute aesthetic layouts, a multi-level force directed scheme is used.

IV. Algorithm

Given a sequence of graphs $G_0, \ldots G_n$, our algorithm computes layouts $L_0, \ldots L_n$. This section describes the algorithm in detail.

We begin with describing how the online dynamic layouts $L_i, i \geq 1$ are computed, given L_{i-1} and G_i. Next, we discuss the algorithm used to compute the initial layout L_0.

A. Computing Dynamic Layouts

Given a set of undirected graphs $G_1, G_2 \ldots G_n$, the goal of the dynamic algorithm is to compute online layouts $L_1, L_2, \ldots L_n$. Algorithm 1 is used to compute the layouts. Figure 2 visualizes the main steps of the algorithm. We elaborate on these steps below.

Algorithm 1 Dynamic layout of graph $G_i, i \geq 1$

input: G_i, L_{i-1} **output:** L_i

1) Merging: Merge layout L_{i-1} and graph G_i to produce an initial layout.
2) Pinning: Assign *pinning weights* to the nodes, which control the allowed displacement of each node.
3) Coarsening: Set $C^0 = G_i$. Compute $C^1, C^2, \ldots, C^{coarsest}$ where $C^{k+1} = edge\_collapse(C^k)$. Set $l = coarsest$.
4) Compute a geometric partitioning of the nodes of C^l.
5) Perform incremental layout of C^l. If $l = 0$ goto step 7 and use the layout of C^0 as L_i (the layout of G_i).
6) Interpolation: Update the initial layout of C^{l-1} using the layout of C^l. Set $l = l - 1$, goto step 4.
7) Animation: Smoothly morph L_{i-1} into L_i.

Merging (Step 1): Computing a good initial position is vital for reducing the layout time and maintaining dynamic stability [15], [33]. The coordinates of nodes that exist both in G_{i-1} and in G_i are copied from L_{i-1}. Nodes in G_i that do not exist in G_{i-1} are assigned coordinates while considering local graph properties, as follows.

Each un–positioned node v is examined in turn. Let $PN(v)$ be the set of neighbors of node $v \in V_i$ that have already been assigned a position. If v has at least two positioned neighbors, v is placed at their weighted barycenter: $pos(v) = \frac{1}{|PN(v)|} \sum_{u \in PN(v)} pos(u)$. If v has a single positioned neighbor, u, then v is positioned along the line between $pos(u)$ and the center of the bounding box of L_{i-1}. This procedure is performed in a BFS (breadth–first search) manner, starting from the positioned nodes. The nodes that cannot be placed by this procedure are placed in a circle around the center of the bounding box of L_{i-1}.

A *Positioning score* $\Gamma(v) \in [0, 1]$ is assigned to each node, based on the method used to position it. These scores indicate the "confidence" in the node's position. The higher the positioning score, the better the initial placement is considered. The scores are used to control the movement of nodes, as described in Step 2. The highest score is assigned to nodes whose neighborhood has not changed between G_{i-1} and G_i, since we are most confident with their positions. A lower score is assigned to nodes that are positioned according to two or more neighbors. An even lower score is assigned to nodes positioned according to one neighbor. Finally, the lowest score is assigned to nodes for which no good initial guess is known, and are therefore placed near the center of the bounding box of the graph. In our implementation, scores of 1, 0.25, 0.1 and 0 are assigned to nodes positioned according to their coordinates at L_{i-1}, at the barycenter of two or more neighbors, according to one neighbor (in a direction pointing away from the center of the bounding box of the graph), and at the center of the bounding box of L_{i-1}, respectively. Figure 2 (b) shows an example of computing the positioning score Γ. Note that darker nodes, with a lower Γ are relatively localized. These changes are propagated to the reset of the graph in the next step.

Pinning (Step 2): After all the nodes are placed, their pinning weights, $w_{pin}(v) \in [0, 1]$, which reflect the stiffness in the positions of the nodes, are computed [6], [7], [20]. The position of a node with a pinning weight 1 is fixed during layout, while a node with a pinning weight 0 is completely free to move during layout.

Pinning weights are assigned using two sweeps. The first sweep, which is local, uses information regarding the positioning scores Γ of the node and its neighbors:

$$w_{pin}(v) = \alpha \cdot \Gamma(v) + (1 - \alpha) \frac{1}{degree(v)} \sum_{u:(u,v) \in E} \Gamma(u).$$

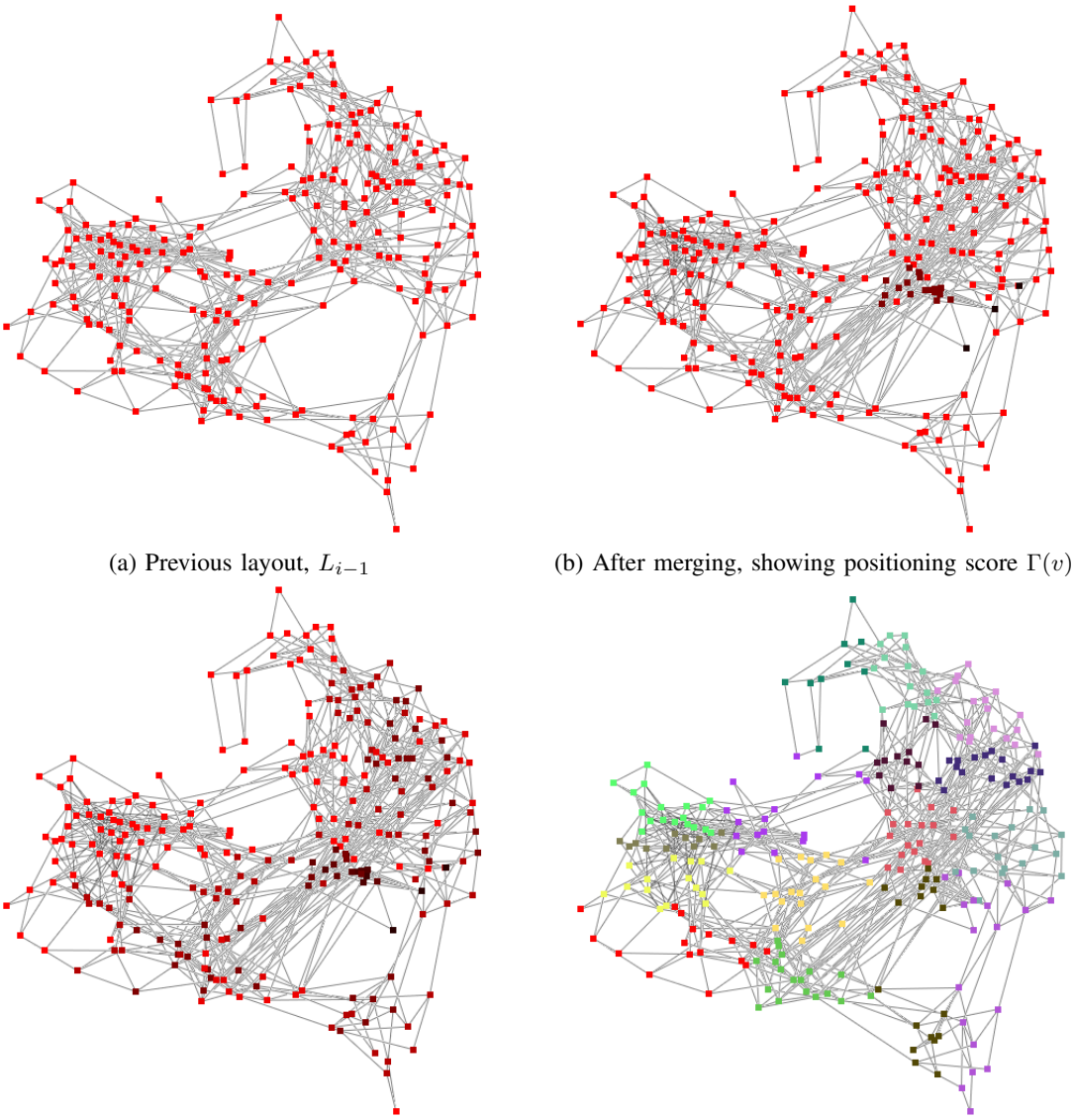
Taking the neighbors of v into account amounts to performing low pass filtering of the pinning weights, according to graph connectivity information. This mimics the creation of flexible ligaments in the graph around areas that were modified. Using a higher α value will reduce the influence of the neighbors of a node on its displacement. In our implementation $\alpha = 0.6$.

In the second sweep, the local changes are propagated, in order to create a global effect. A BFS-type algorithm assigns each node a *distance-to-modification* measure, as follows. The distance-zero node set, D_0, is defined as the union of the set of nodes with a pinning weight of less than one and the set of nodes adjacent to an edge that was either added or removed from G_{i-1}. The distance-one set, D_1, is defined as the subset of nodes in $V \setminus D_0$ adjacent to a node in D_0. In general, D_i is the subset of nodes not yet marked, which are adjacent to a node in D_{i-1}. This process continues until all the nodes in V are assigned to one of the sets $D_0, D_1, \cdots, D_{dmax}$. Note that according to this definition, the nodes in set $D_i, i \geq 1$ were assigned $w_{pin} \equiv 1$ in the first sweep. In the second sweep, as described below, some of these nodes are assigned a lower pinning weight. This gives the layout algorithm more flexibility in adopting to changes in the graph.

Pinning weights are assigned to nodes based on their *distance-to-modification*. In particular, nodes that are farther than some cutoff distance $dcutoff$, are assigned a pinning weight of one, thus remaining fixed, since they are far away from areas of the graph that were changed. The movement of other nodes depend on the set D_i they belong to. This is done as follows. Given $dcutoff = k * dmax$, the nodes in D_i, $i \in [1, dcutoff]$ are assigned pinning weights:

$$w_{pin} = (w_{pin}^{initial})^{(1 - \frac{i}{dcutoff})}.$$

This assignment creates a decaying effect in which nodes farther away from D_0 are assigned higher pinning weights. The constant $w_{pin}^{initial}$ is used to determine the decay in pinning weight. The nodes in D_{j+1} are assigned a pinning weight that is $(w_{pin}^{initial})^{(\frac{-1}{dcutoff})}$ times the pinning weight of nodes in D_j. Note that a larger k results in a more global effect, possibly trading layout stability for better layout quality (since nodes are more free to move). Setting a higher $w_{pin}^{initial}$ will make the graph more rigid, thus limiting the displacement of nodes already existing in the previous layout. In our implementation $k = 0.5$ and $w_{pin}^{initial} = 0.35$.



(a) Previous layout, L_{i-1}

(b) After merging, showing positioning score $\Gamma(v)$

(c) After merging, showing pinning weight $w_{pin}(v)$

(d) Final layout of L_i, showing partitions

Fig. 2. Dynamic layout steps: (a) previous layout, L_{i-1} (b) merged graph (Step 1), color coded according to the positioning score $\Gamma(v)$. Brighter nodes have a higher Γ. Here, nodes with $\Gamma \in \{0.1, 0.25, 1\}$ are shown. (c) Pinning weights $w_{pin}(v)$ (Step 2). Brighter color corresponds to a higher $w_{pin}(v)$ (d) Final layout (Step 5), color coded according to the partitioning (Step 4)

Figure 2 (c) shows an example of computing the pinning weights. Note how the local changes in (b) are propagated to a larger portion of the graph. Also note the decaying effect as the distance from the modified part, in the middle of the graph, increases. This reflects the requirement that nodes further from the changed areas should undergo fewer modifications during layout.

While pinning weights were proposed in the past [6], the approach taken here is different. In the current paper pinning weights are used as part of setting the allowed displacement of nodes, prior to computing the layout. This controls the movement flexibility of each node. In [6], nodes are displaced according to a combination of two different forces. The relative strength of the forces is determined by weights that are modified as the layout iterations progress.

Coarsening (Step 3): In this step a series of reduced versions of the graph, which include initial positions, are constructed. These are used to compute increasingly detailed "skeletons" of the final layout. At each level, given a fine graph, a coarser representation is constructed by performing a series of *edge collapse* operations. This is done by replacing two connected nodes and the edge between them by a single node, whose weight is the sum of the weights of the nodes being replaced. The pinning weight of the new node is set to the geometric mean of the pinning weights of the replaced nodes. The new node is placed at the weighted average position of the corresponding fine nodes, biased according to their weights. The weights of the edges are updated accordingly. (The weight of a node/edge in the finest graph is 1.)

The order of the edge collapse operations is determined as follows. First, nodes, which are candidates to be eliminated, are sorted by their degree (so as to eliminate low-degree nodes first). An adjacent edge of an un-paired low-degree node is chosen for collapse by maximizing the following measure: $\frac{w(u,v)}{w(v)} + \frac{w(u,v)}{w(u)}$, where $w(x)$ is the weight of node x and $w(x,y)$ is the weight of edge (x,y). This function helps to preserve the topology of

$frac_{done} = 0$, $K = 0.1$, $t = K * \sqrt{|V|}$, $\lambda = 0.9$
do iteration_count times,
update partitioning (Alg. 1 Step 4, Alg. 2 Step 3) if required
parallel_foreach partition $P_i \in P$,
(1) calculate partition center of gravity $CG(P_i) = \frac{\sum_{v \in P_i} pos(v)}{|P_i|}$
parallel_foreach node v, $v \in P_i$ where $frac_{done} > w_{pin}(v)$,
(2) $F_{int}^{repl}(v) = \sum_{u \in P_i, u \neq v} K^2 \frac{pos(v)-pos(u)}{\|pos(v)-pos(u)\|^2}$
(3) $F_{ext}^{repl}(v) = \sum_{P_j \in P, P_j \neq P_i} K^2 |P_j| \frac{pos(v)-CG(P_j)}{\|pos(v)-CG(P_j)\|^2}$
(4) $F_{tot}^{repl}(v) = F_{int}^{repl}(v) + F_{ext}^{repl}(v)$
(5) $F^{attr}(v) = \sum_{u:(u,v) \in E} \frac{\|pos(u)-pos(v)\|(pos(u)-pos(v))}{K}$
parallel_foreach node v where $frac_{done} > w_{pin}(v)$,
(6) $F^{total}(v) = F_{tot}^{repl}(v) + F^{attr}(v)$
(7) $pos_{new}(v) = pos(v) + \frac{F^{total}(v)}{\|F^{total}(v)\|} min(t, \|F^{total}(v)\|)$
$t* = \lambda$, $frac_{done} += iteration\_count^{-1}$

Fig. 3. Parallel force directed layout algorithm

the graph by "uniformly" collapsing highly connected nodes. Coarsening is used in [12], where a different ordering of the edge collapse operations is used.

In our implementation, the coarsening stops either when the graph is reduced to several hundred nodes or after four coarsening steps. Coarsening further may lead to diminishing results due to the inaccuracy in the computed pinning weights of the coarse graph.

Geometric partitioning (Step 4): The partitioning step is used to accelerate the layout step, discussed below. There are three requirements that should be satisfied by partitioning. First, the partitions should be geometrically localized, thus the nodes in each partition should be relatively close to each other. This will let us represent each partition using a single "heavy" node. Second, the number of nodes in each partition should be similar. This is important in order to achieve good load balance between the parallel processors of the GPU, as discussed in Section V. Third, the algorithm should be fast.

We have chosen to use a KD-tree-type partitioning. The algorithm works top down. Given the positions of all nodes, they are sorted according to the X coordinate and the index of the median node is located. The nodes are partitioned into two sets: one with indices below the median and one with indices equal or greater to the median index. The algorithm proceeds recursively with the two subsets. This time, sorting is performed according to the Y coordinate. The algorithm alternates between computing the median X and Y coordinates. The recursive subdivision terminates when the size of the subset is below the required partition size. Figure 2 (d) shows an example of computing a geometric partitioning of a graph.

Layout (Step 5): This step of the algorithm computes the layout. Our algorithm builds on the basic Fruchterman-Reingold (FR) force directed algorithm [32], which is modified, so as to make it suitable both for incremental layout and for efficient implementation on the GPU. The basic algorithm is thus modified in three ways. First, an approximate force model is used in order to speedup the calculation. Second, node pinning allows individual control over the movement of each node. Third, the algorithm is reformulated in a manner suitable for efficient implementation on the GPU.

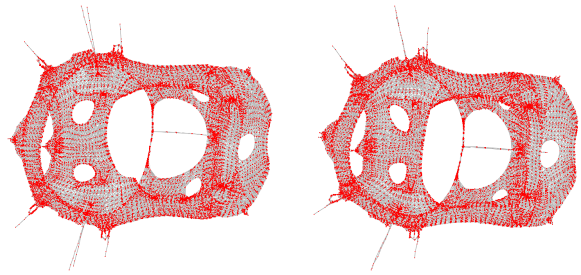
Figure 3 outlines our algorithm. The input is a graph $G = (V, E)$ decomposed into partitions P_i, nodes with initial placement $pos(v)$, and their pinning weights $w_{pin}(v)$. The output is the positions for all nodes. The key idea of the algorithm is to converge into a minimal energy configuration, which usually leads to aesthetically pleasing layouts.

The initialization of the algorithm includes setting the optimal geometric node distance K (that affects the scale of the graph), the initial annealing temperature t, the temperature decay constant λ, and the fraction of the iterations done $frac_{done} \in [0, 1]$.

Partitioning is used to accelerate the algorithm. Instead of calculating all-pair repulsive forces, as is customary, approximate forces are calculated. An exact calculation is performed only for nodes contained in the same partition, while an approximate calculation is performed for nodes belonging to different partitions. The center of gravity is found for each partition P_i and is used to replace the nodes in P_i.

Our experiments show that there is flexibility in the number of nodes in each partition, e.g. Figure 4 shows that using twenty times fewer nodes in each partition has little effect on the final layout. Moreover, it is not necessary to re-partition at every iteration, except for the initial iterations of the initial layout (Algorithm 2, Step 4), where the nodes may have a high displacement. During the incremental layout, the merge stage (Algorithm 1, Step 1) already gives a good approximation of the final layout. In cases where there are large changes between consecutive graphs, performing several re-partitioning steps may improve the results. These cases can be identified using the following formula: $\frac{1}{|V|} \sum_{v \in V} (1 - w_{pin}(v))$, whose value is proportional to the changes performed to the graph. This is so since the number of iterations during which each node v moves, is proportional to $(1 - w_{pin}(v))$ (see Figure 3).

The key to efficient implementation of this algorithm on the GPU is deciding which nodes will be processed by the *parallel_foreach* loops. In order to reduce layout time and maintain dynamic stability, only some of the nodes are displaced in each



(a) $0.5\sqrt{|V|}$ partitions (b) $10\sqrt{|V|}$ partitions

Fig. 4. Partition size effect on layout, graph bcsstk31, $|V| = 35588$, $|E| = 572916$

layout iteration. For each node v, $w_{pin}(v)$ is compared to the current fraction of layout iterations done, $frac_{done}$. Only nodes that satisfy $frac_{done} > w_{pin}(v)$ are processed. This makes it possible to control the relative displacement of nodes. Nodes with a low pinning weight will be displaced during more iterations of the algorithm. Thus, the pinning weight, assigned according to the changes performed in the vicinity of each node, controls the stability of node locations. Because the allowed displacement is decreased from one iteration to the next, setting a higher pinning weight limits the total displacement of nodes.

Using this method, the algorithm spends computation time only on nodes which should be displaced in each layout iteration. The amount of work done depends on the changes performed to the graph. Areas which did not change are not processed, thereby reducing the layout time. It is often possible to accelerate the incremental layout time by a factor of two using this technique.

The algorithm computes the total force acting on each node in several steps. First, the centers of gravity of all partitions are computed. Next, the set of active nodes, which are allowed to be displaced in the current iteration, is determined. For each such node, the repulsive forces $F^{repl}_{int}, F^{repl}_{ext}$ and the attractive force F^{attr} acting on it, are calculated. Finally, the nodes are displaced by an amount bounded by the current temperature of the algorithm, which slowly decays, mimicking particles freezing into position.

Interpolation (Step 6): In this stage the computed layout of graph C^l is interpolated and used to update the initial layout of the higher-resolution graph C^{l-1}. Given a node $v \in C^{l-1}$, which was mapped to node $p \in C^l$, node v is displaced by the following amount:

$$(1 - w_{pin}(v)) \frac{A(Bbox^{old}(C^l))}{A(Bbox^{new}(C^l))} (pos^{new}(p) - pos^{old}(p)),$$

where $A(Bbox^{old}(C^l))$ is the area of the bounding box of graph C^l computed during the coarsening step, $A(Bbox^{new}(C^l))$ is the area computed during the layout step, $pos^{old}(p)$ is the position of node p computed during the coarsening step and $pos^{new}(p)$ is the position of p computed during the layout step. The motivation for using this formula is as follows. The amount $1-w_{pin}(v)$ is used to displace nodes according to their pinning weights. Nodes with a higher pinning weight are allowed a smaller displacement. Doing so helps maintain the stability of the graph. Nodes with a lower pinning weight are allowed greater flexibility in order to compute a high-quality layout. The displacement is scaled according to the change in the area of the coarser C^l due to the layout step. Finally, node v is displaced according to the movement of the corresponding lower-resolution node p.

Morphing (Step 7): The old layout L_{i-1} is morphed into the new layout L_i. The animation, showing a gradual change, helps the user maintain the mental map of the graph. Node positions are linearly interpolated. Removed nodes and edges fade out, then the nodes and edges move to their new position and finally added nodes and edges fade into view.

Complexity: The asymptotic complexity of the merging, pinning, coarsening and interpolation steps is $O(|E| + |V|)$. The complexity of the partitioning step is $O(|V| \cdot \log(|V|))$: finding the median is linear at each level in the partition tree which contains $O(log|V|)$ levels. Assuming that each partition contains C_s nodes, the running time of each layout iteration is $O(|E| + |V| \cdot (C_s + \frac{|V|}{C_s}))$. This expression is minimized when $C_s = \sqrt{|V|}$, resulting in a total complexity of $O(|E| + |V|^{1.5})$. When $|E| \approx |V|$, the dominating term is $|V|^{1.5}$. Although this may look relatively high, the simplicity of the calculation and its parallel implementation on the GPU give good results, as discussed in Section VI. We use 50 layout iterations [12].

B. Computing the Initial Layout L_0

Algorithm 2 is used to compute a static layout of the first graph, G_0. This algorithm uses a multi-level force directed scheme in order to quickly compute an aesthetic layout. Both the Kamada-Kawai (KK) [34] and Fruchterman-Reingold (FR) [32] algorithms are employed. We elaborate on the steps of the algorithm below.

Algorithm 2 Static layout of the first graph, G_0

input: G_0 **output:** L_0

1) Coarsening: Set $C^0 = G_0$. Compute $C^1, C^2, \ldots, C^{coarsest}$ where $C^{k+1} = edge\_collapse(C^k)$. Set $l = coarsest$.
2) Perform KK layout of $C^{coarsest}$.
3) Compute a geometric partitioning of the graph nodes.
4) Perform layout of C^l. Update the partitioning (step 3) every few iterations. If $l = 0$ terminate and use the layout of C^0 as L_0 (the layout of G_0).
5) Interpolate the layout of C^l to form an initial layout for C^{l-1}. Set $l = l - 1$, goto step 3.

Coarsening (Step 1): A similar method to Algorithm 1, Step 3 is utilized to create a series of reduced versions of the graph, which are used to compute increasingly detailed "skeletons" of the final layout. The coarsening continues recursively until a small graph of several hundred nodes is created. This graph is then efficiently handled in the next step and is used as a basis of a series of resolution-increasing layouts. Note that unlike the incremental case, initial coordinates for the constructed graphs C^k, are not available.

KK layout (Step 2): The KK algorithm [34] is used to compute a force-directed layout of the coarsest graph, $C^{coarsest}$. This algorithm is used in conjunction with the FR [32] force-directed algorithm (in Step 4) in order to produce an aesthetic layout. While the KK algorithm is good at producing a good placement from an arbitrary initial position, the FR algorithm produces a "smoother" layout, is quicker, but is more sensitive to the initial conditions given to it. Hence, combining the algorithms gives a fast and aesthetic result. In our implementation 2000 iterations of the KK algorithm are performed. Note that during incremental layout (Section IV-A) combining our multi-level approach while

reusing the previous layout as a starting point gives fast and good results without incurring KK's performance penalty.

Geometric partitioning (Step 3): The same algorithm as in step 4 of Algorithm 1 (Section IV-A) is used here.

FR layout (Step 4): In this step we perform force-directed layout of the current graph in the hierarchy, C^l. The algorithm is described in detail in Step 5 of Algorithm 1 (Section IV-A). Unlike the dynamic case, here pinning weights are not used and all nodes are free to move in every layout iteration. In order to get improved results, we update the node partitioning (Step 3) several times during the layout. The center of gravity of each partition is updated every iteration, though. The algorithm terminates when the layout of $C^0 = G_0$ is computed.

Interpolation (Step 5): In this stage the existing layout of C^l is interpolated to form an initial layout for the higher-resolution C^{l-1}. Nodes in C^{l-1} are initially placed near the position of their parent in C^l.

V. Implementation

This section discusses the implementation of the algorithm. As will be shown in Section VI, performing incremental layout, i.e. Algorithm 1, Step 5, (and similarly Algorithm 2, Step 4) on the GPU can significantly accelerate the overall running time of the algorithm. Therefore, in this section we focus on describing the GPU implementation of this step.

On the GPU, parallel computation is achieved by rendering graphics primitives that cover several pixels. The GPU runs a program called a *kernel program* for each pixel candidate, called a *fragment*. The key to high performance on the GPU is using multiple fragment processors, which operate in parallel. The GPU suits uniformly structured data, such as matrices. The challenge is representing graphs, which are unstructured, in a manner that makes efficient use of GPU resources.

Implementing static force directed layout on the GPU has been discussed in [31]. While the algorithm used here for static layout is different, the GPU implementation is similar. This section reviews the GPU implementation and focuses on the changes needed for dynamic layout.

Several textures are used on the GPU to represent the graph: the textures represent the nodes, the partitions, the edges, and the forces. The *location* texture holds the (x,y) positions of all the nodes in the graph. Each graph node has a corresponding (u,v) index in the texture. As shown in Figure 5 (a), the nodes in each partition are stored in a rectangular region in the location texture.

Bucket-sort is performed on the pinning weights of the nodes in each partition. Nodes are placed into the texture in a left to right, top to bottom order, according to the bucket they belong to, as shown in Figure 5 (b). The number of buckets is set to the number of iterations of the layout algorithm. Sorting creates contiguous regions of nodes with similar w_{pin} values. This allows the algorithm to control the set of nodes whose positions are updated at every layout iteration. Using appropriate rendering commands, the GPU is instructed to process only the relevant nodes in each iteration, as discussed below.

The *partition center of gravity* texture holds the current (x,y) coordinates of the center of gravity of each partition. Graph edges are represented using the *neighbors* texture and the *adjacency* texture. The *adjacency* texture contains lists of (u, v) pointers into the location texture, representing the neighbors of each node. The *neighbors* texture holds for each node v, a pointer into

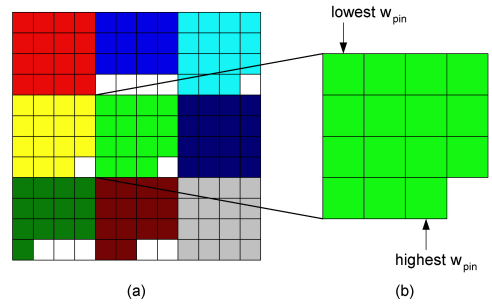


Fig. 5. Sorting nodes by pinning weight w_{pin} on the GPU. (a) : A location texture separated to regions, color coded by the partition each node belongs to. (b) : Nodes in each region are sorted from low w_{pin} to high w_{pin}.

the adjacency texture, to the coordinates of the first neighbor of the node. Pointers to additional neighboring nodes are stored in consecutive locations in the adjacency texture. The neighbors texture also holds the degree of each node. The forces computed during layout are stored in two textures: the *attractive force* texture and the *repulsive force* texture. The *attractive force* texture contains for each node the sum of the attractive forces F^{attr} exerted on it by its neighbors. The *repulsive force* texture holds the sum of repulsive forces, both by nodes in the same partition – F^{repl}_{int} and by the other partitions in the graph – F^{repl}_{ext}.

The overall storage complexity is $O(|V|+|E|)$: every node and edge is stored a fixed number of times. Each node is represented as four 32-bit floating-point values in the following textures: location (two textures), forces (two textures) and neighbors. Each edge is represented twice in the adjacency texture (once for each of the nodes in its endpoints), whose entries are also four 32-bit floating-point numbers. Due to performance reasons, information about the graph partitions is stored in three textures holding four 32-bit floating-point numbers each. These textures have the same size as the textures representing nodes.

Hence, in the current implementation, a total of 32 32-bit numbers are stored per node and 8 32-bit numbers are stored per edge in the different textures. This amounts to about 8MB of texture memory for the fe_pwt graph with $(V, E) = (32045, 112395)$. Modern graphics cards have hundreds of megabytes of texture memory, making accommodation of very large graphs possible. Note that for implementation ease, textures holding four 32-bit numbers are used in all cases. This in not always required, and can further reduce the memory footprint.

Computing each layout iteration is done in several steps, which are implemented as kernel programs that run on the GPU. The *partition CG* kernel calculates the center of gravity of each partition, as shown in the line numbered (1) in Figure 3. The *repulse* kernel calculates the repulsive forces exerted on each node. This kernel first calculates for each fragment it processes, the internal forces, e.g. forces exerted by nodes contained in the partition that the fragment belongs to. Then, it approximates the forces by all other partitions. See lines (2)-(4) in Figure 3. The *attract* kernel is used to calculate the attractive forces caused by graph edges. For each node, the kernel accesses the neighbors texture in order to get a pointer into the adjacency texture, which contains the (u,v) location texture coordinates of the node's neighbors. For each neighboring node, the attractive force is calculated and accumulated. This corresponds to line (5) in Figure 3. Finally, the *anneal* kernel calculates the total force on each node, F^{total}, and

graph	rimzu		threads1		threads2		newcomb		3elt		fe_pwt	
metric	Δpos	$\|U^{total}\|$	Δpos	$\|U^{total}\|$	Δpos	$\|U^{total}\|$	Δpos	$\|U^{total}\|$	Δpos	$\|U^{total}\|$	Δpos	$\|U^{total}\|$
non-incr	31.4	4418	1.45	39.2	1.06	9.72	0.48	1.82	25.9	2.73x10^5	105.5	9.59x10^5
basic-incr	4.62	4435	0.333	40.4	0.297	9.81	0.221	1.81	2.3	3.06x10^5	10.7	9.37x10^5
ours	0.274	3418	0.042	30.3	0.048	5.55	0.099	1.94	0.968	2.79x10^5	3.62	8.1x10^5

TABLE I

LAYOUT QUALITY - VALUES ARE AVERAGES FOR A SEQUENCE OF LAYOUTS

Graph name	avg. $\|V\|$	avg. $\|E\|$	3GHz Pentium + 7900GS GPU				2.4GHz Core 2 + 8800GTS GPU			
			initial layout		dynamic layout		initial layout		dynamic layout	
			CPU	CPU+GPU	CPU	CPU+GPU	CPU	CPU+GPU	CPU	CPU+GPU
3elt	4097	10468	2.72	1.49	0.764	0.249	1.72	1.27	0.436	0.2
4elt	14588	40176	17.6	2.98	5.91	0.777	10.4	2.22	3.38	0.39
bcsstk31	32715	48495	50.4	9.28	21.2	4.74	34	9.61	12.1	1.38
fe_pwt	32045	112395	47.7	6.03	21	2.1	28.8	4.27	12	0.704

TABLE II

GRAPH SEQUENCE INFORMATION AND RUNNING TIME [SEC.]. RUNNING TIMES OF THE CPU ONLY AND GPU-ACCELERATED IMPLEMENTATION OF THE ALGORITHM ARE SHOWN. ALL TIMES SHOWN ARE TOTAL RUNNING TIMES FOR COMPUTING A LAYOUT. DYNAMIC LAYOUT TIMES ARE AVERAGED OVER A SEQUENCE OF LAYOUTS.

displaces nodes accordingly, as shown in lines (6),(7) in Figure 3. This kernel updates a second copy of the location texture. This double buffering is required since the GPU can not read and write to the same texture.

In total, the *partition CG* kernel performs $O(|V|)$ operations; the *repulse* kernel performs $O(|V|^{1.5})$ operations; the *attract* kernel performs $O(|E|)$ operations; and the *anneal* kernel $O(|V|)$ operations. On the GPU, the computations executed in each kernel, are run in parallel. Since, as discussed below, only some of the nodes are operated on during each layout iteration, in practice the average number of operations performed by each kernel is lower than the maximum values presented above.

Recall that the nodes in each partition are sorted according to w_{pin}, as shown in Figure 5 (b). This allows us to control the nodes processed in each layout iteration, thus spending GPU time only on the nodes which should move. Before each layout iteration, for each rectangular texture region representing a partition of the graph, the rows which contain nodes for which $frac_{done} > w_{pin}(v)$ are determined. A set of quadrilaterals which cover the corresponding parts of each region are rendered. This instructs the GPU to process only these nodes. OpenGL *display lists* are used in order to efficiently send these rendering commands to the GPU. Note that this method operates on a per-row basis, potentially causing a small amount of extra fragments to be processed for each region. The processing of these extra fragments is avoided by conditionally updating the location of a node only if $frac_{done} > w_{pin}(v)$.

Note that our implementation does not require copying data from GPU memory (textures) to CPU memory while performing the layout iterations. Keeping the data on the graphics card enables full utilization of the GPUs compute and memory bandwidth resources.

VI. RESULTS

Two criteria are used to measure the quality of the resulting dynamic layouts: *average displacement of nodes between each pair of successive layouts* and *potential energy*. The first criterion measures the stability of the layout. The second criterion judges the quality of the layout. Lower energy (in absolute value) implies low stress in the graph, corresponding to a good layout. The energy U is derived from the relation $\vec{F} = -\nabla U$. Hence, given the force $\vec{F}$, the energy can be derived by integrating. Given two nodes at positions $\vec{u}, \vec{v}$, connected by an edge , the attractive force acting along the edge is

$$\vec{F}^{attr} = \frac{1}{K}\|\vec{u}-\vec{v}\|(\vec{u}-\vec{v}) = -\nabla U^{attr},$$

hence

$$U^{attr} = \frac{-1}{3K}\|\vec{u}-\vec{v}\|^3.$$

The repulsive force between two nodes is

$$\vec{F}^{repl} = \frac{-(\vec{u}-\vec{v})}{\|\vec{u}-\vec{v}\|^2}K^2 = -\nabla U^{repl},$$

hence

$$U^{repl} = \frac{1}{2}K^2 log(\|\vec{u}-\vec{v}\|^2).$$

The total energy is computed by summing over all edges and over all node pairs: $U^{total} = U^{attr} + U^{repl}$, e.g.

$$U^{total} = \sum_{u:(u,v)\in E} \frac{-1}{3K}\|\vec{u}-\vec{v}\|^3 + \sum_{u,v\in V, u\neq v} \frac{1}{2}K^2 \log(\|\vec{u}-\vec{v}\|^2).$$

Other static graph layout quality criteria are indirectly handled by the underlying force directed algorithm. Note that other criteria have also been used to measure mental map preservation. For example the orthogonal ordering of nodes [4].

The quality of the layout is compared to two algorithms. The first is a force-directed non-incremental algorithm that lays each graph in the sequence independently. This algorithm, which is expected to produce the best layouts since it has no constraints, is used to check the quality of our dynamic layouts. The second is a variant of our dynamic algorithm which does not use pinning weights (e.g. $w_{pin} \equiv 0$). This algorithm demonstrates that simply using the previous placement is insufficient for generating stable layouts. Note that the running time of these two algorithms is much higher than the running time of our algorithm since they process all nodes in each layout iteration.

Several well–known graphs (3elt, 4elt, fe_pwt, bcsstk31) are used to demonstrate our algorithm [38]. The dynamic sequences are generated by performing random changes on the graphs, modifying $|E|$ and $|V|$ by up to 15%. In addition, the sequences marked threads1,2 and Rimzu come from real data, discussed

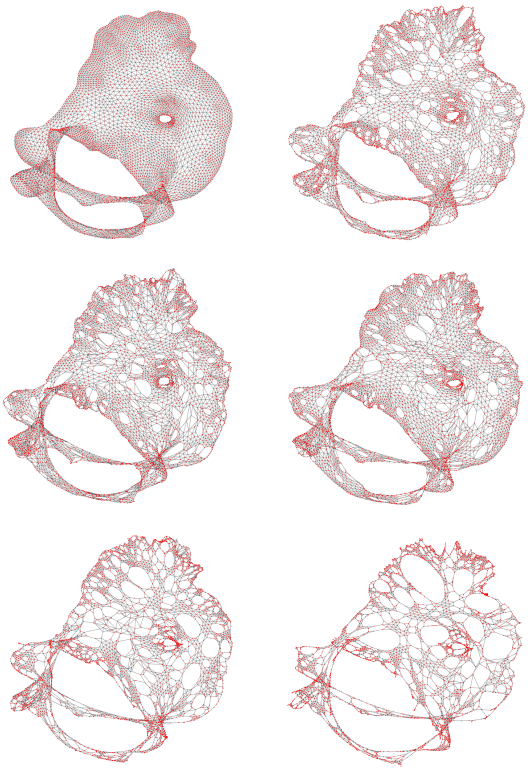


Fig. 6. Snapshots from layouts of the 3elt sequence ($|V| \approx 4000$, $|E| \approx 10,500$), left-to-right, top-to-bottom

in Sections VII, VIII. Figure 6 shows a few snapshots from the dynamic graph layout of 3elt.

Another example is Newcomb's fraternity data [35], which represents friendship relations between college students. This data was visualized using the SoNIA tool for social network visualization [21], [36], [37]. As discussed in [21], the Newcomb data is best visualized by the peer-influence (PI) algorithm of SoNIA, where nodes are displaced according to forces exerted by neighbors.

Table I shows average results for the layout quality metrics. (Lower values are better.) The Δpos column shows the average displacement of nodes and the $|U^{total}|$ column shows the absolute value of the potential energy of the graph. It is clear that our incremental algorithm outperforms the other algorithms and maintains dynamic stability. The potential energies achieved by all algorithms are similar, demonstrating that the quality of layouts computed by our algorithm is good. In some cases (like fe_pwt) the two incremental algorithms surprisingly perform better than the static one. This is due to the fact that the force-directed algorithm finds a local minimum which depends on the initial conditions, which are different for each algorithm used here. In summary, the results demonstrate that our algorithm computes aesthetic layouts while decreasing the movements of the nodes. This reduction does not come at the expense of layout quality. The algorithm tries to maintain the structure of the graph, using node pinning to propagate changes across the graph, allowing for new landmarks to be created, while at the same time maintaining the mental map. Note that compared to the algorithm of [9], using a multi-level incremental algorithm somewhat reduces the stability of the layout. However, this gives the algorithm an opportunity to calculate a higher quality layout.

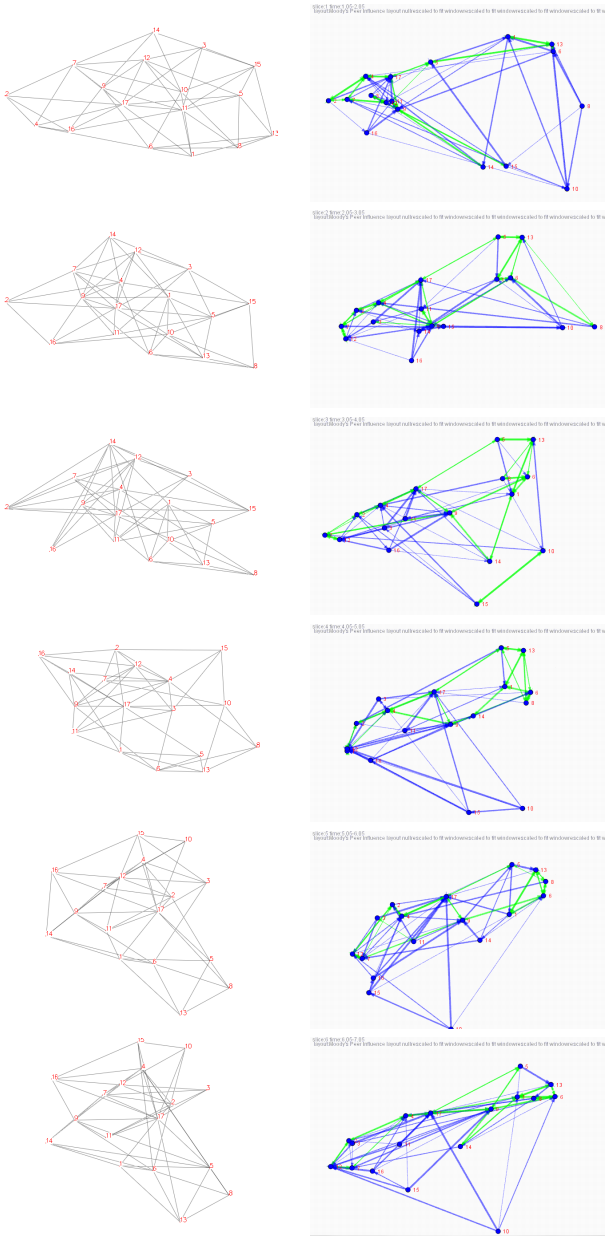


Fig. 7. Snapshots from the layouts of the newcomb fraternity data [35]. Left: our algorithm. Right: SoNIA algorithm [36], [37], used in [21].

Figure 7 shows a comparison of the SoNIA layouts using the PI algorithm and our layouts. As can be seen, one of the advantages of our algorithm is the greater stability in node positions, especially when only the edges of the graph are modified. Although both SoNIA and our algorithm are based on force-directed methods, the more sophisticated initial placement and pinning algorithms help improve the results.

For our performance tests we used two computers. The first is a PC with a 3 GHz Pentium IV CPU and an NVIDIA 7900GS GPU. The second is a newer PC with a 2.4 GHz Intel Core 2 Duo E6600 CPU and an NVIDIA 8800GTS GPU. Our algorithm was implemented using C++, Cg and OpenGL.

Table II gives information about the graph sequences and running times - when using only the CPU and when using the GPU to accelerate the computation. As can be seen in the table, our GPU implementation provides a significant speedup compared

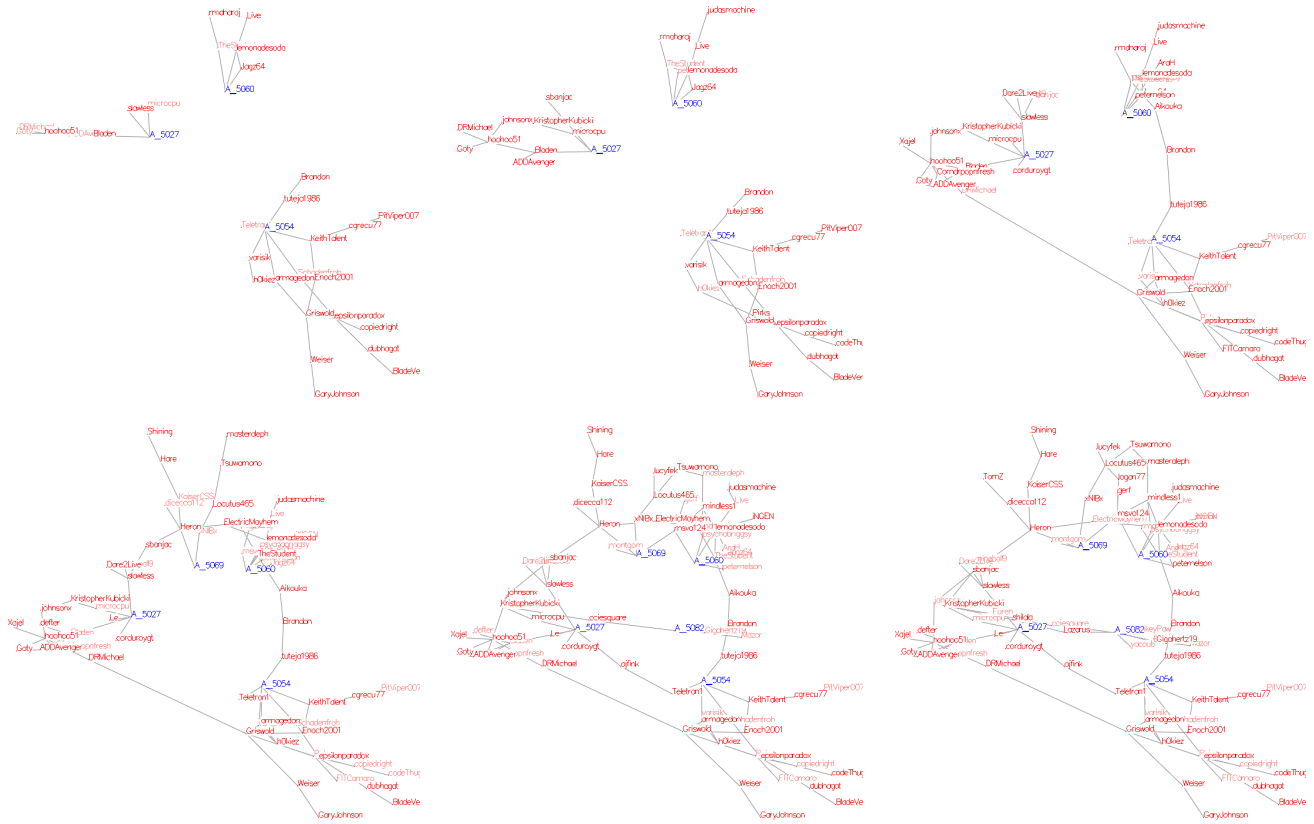


Fig. 8. Snapshots from the threads2 graph sequence, visualizing discussion threads at http://www.dailytech.com, left to right, top to bottom. 109 messages from 86 users in 5 discussion threads are shown. Discussion topics, marked as blue A_n nodes, include computer games (A_5054), nuclear fusion (A_5027), low-cost PCs (A_5060), Windows/Linux switch (A_5069) and Christmas e-shopping (A_5082) .

to the CPU. Using the older 7900GS GPU, a speedup of up to 10 times is achieved. Using the newer and faster 8800GTS GPU, the speedup increases to up to 17 times, compared to the latest CPU. Due to the high ratio of arithmetic operations to memory accesses, the algorithm is compute and not memory bound. Therefore, as demonstrated in the comparison between the PCs, the GPU implementation of the algorithm is scalable.

Focusing on the part of the algorithm that runs on the GPU leads to interesting insights. For the fe_pwt graph, the average time for computing the FR incremental layout stage using the 7900GS GPU was 1.66 seconds. Using the 8800GTS GPU, the time dropped to 0.417 seconds. This represents a significant performance increase between GPU generations (~ 4 times), which is larger than the performance increase between the CPU generations [23]. The speedup is achieved while taking into account the overhead of instructing the GPU to perform the layouts, which can be significant in the coarser graphs. The speedup of performing the last layout stage (on the finest graph) is about 8 times.

There are several factors contributing to the increase in performance between the GPUs. The new GPU has a different architecture, which is better suited for dealing with graphs. Due to its smaller branch granularity, a smaller penalty is encountered when dealing with non-uniform data, such as graphs. In addition, the 8800GTS uses a scalar architecture, which is more efficient here, since the algorithm deals mostly with 2D and 1D quantities. Finally, the new GPU has more raw compute power.

VII. Application to Discussion Thread Visualization

We applied our algorithm to the visualization of Internet discussion forums. We collected data from several discussion threads at `http://www.dailytech.com`. This site contains various hi-tech related news items. The discussion threads visualized contain the comments people make on the news items. In the graph, each node represents a user. Edges are constructed between the user adding a comment and users which replied to that comment. Each discussion thread is represented by a node labeled A_n where n is the discussion thread number (corresponding to a news item).

In order to create the visualization, shown in Figures 1 and 8, several steps are executed. First, the graph is transformed into a connected graph, as required by the graph layout algorithm. This is achieved by adding an invisible root node and connecting it with invisible edges to all the A_n nodes representing the discussion threads. The connected graph is then handed to the incremental layout algorithm.

Second, in order to improve the visualization of the computed layout sequence, overlapping between node labels is addressed. A set of bounding boxes of drawn node labels is maintained and updated after each label is drawn. If a new label to be drawn intersects any of the bounding boxes of already drawn labels, it is drawn at the background – farther away from the viewer and with a lighter color. Doing so prevents the new label from occluding the text of any previously drawn labels. If a new label does not intersect any of the existing labels, it is drawn in the foreground. Before each node label is drawn, a rectangle with the same color

as the background is drawn behind the node label. This is done so each pixel will display the text of a single label (preventing overlaps).

Third, during animation, the nodes are drawn in a specific order which is designed to visualize the interesting features of the evolving graph sequence more clearly. The labels of important nodes should receive priority when drawn. These include nodes with a high degree, acting as central nodes and in the graph, and nodes whose neighborhood in the graph has changed. Each node is assigned a score. Nodes with a higher score are rendered before nodes with a lower score. This reduces the probability that an important node's label will be occluded. The score of each node v is set to $score(v) = degree(v) + \beta \cdot degree\_change(v)$, where $degree\_change(v)$ is the change of the degree of node v between the current and previous graphs. The score helps emphasize the main features of the evolving graph sequence. The constant β can be changed by the user. Its default value is 2.

Figure 1 shows a sample visualization of 7 discussion threads with 119 users. Although during visualization the graph more than doubles, our layout manages to preserve the mental map. Several insights can be gained from the visualization. Clusters are evident around the A_n nodes, representing each discussion thread. As time progresses, more clusters, representing new discussion threads, become visible. There are clusters of various sizes – correlating to threads drawing different levels of attention. Some users post messages on several threads while others discuss only one topic. Some users are very active and post many messages, acting as central nodes in the graph. The degree of nodes representing such users increases over time and they contribute to the connectivity of the graph. Some users, who are drawn at the boundaries of the graph, contribute only one comment.

As a second example we studied the latest headlines section of the website. We selected five items, appearing over a span of three days, from seemingly unrelated fields: computer games, nuclear fusion, low-cost PCs, Windows/Linux switch and Christmas e-shopping. The number of comments for each article varied from 15 to 31. A total of 86 users contributed to the discussion threads. Figure 8 presents several snapshots from the animation sequence showing the evolution of these discussion threads over time. A movie showing the visualization is available in the supplementary material.

Looking at the visualization, several conclusions can be drawn. The graph is initially partitioned into disconnected clusters, representing nuclear fusion, low-cost PCs and computer games. Later, connections start to appear in the graph. The threads discussing low-cost PCs and Windows/Linux switch are highly connected. Some connections exist between these clusters and the computer game cluster. Surprisingly, several users discussing nuclear fusion join both the computer games and Windows/Linux switch threads. Good correlation also exists between nuclear fusion and the Christmas e-shopping discussion.

VIII. Application to Social Network Visualization

Our algorithm was applied to the visualization of the growth of social networks. We used data from the social network at `http://www.rimzu.com`. In this network, new users can register after receiving an invitation from an existing user. Each user is able to list a set of friends among the members of the network. In the visualization, users are represented as nodes. Edges link each user to his/her friends.

Figure 9 shows a visualization of the growth of this network. The visualization shows a period in time where the network grew considerably, from 216 nodes to 962 nodes. The visualization was created by constructing the graph of the network at equally spaced intervals in time. As in the Internet threads visualization, a dummy invisible root node was added in order to make the graph connected.

Several properties of the network are evident from the created visualization. The graph has dozens of connected components. The fact that the graph is not connected is surprising since members are able to join the network only after receiving an invitation. There are many users who joined the network but did not list any friends. They are represented as a cluster of nodes with degree zero (no edges). There are components of varying complexity in the network. Some are very simple, connecting a handful of nodes, while others are large and highly connected. Several tree-like components are visible. These correspond to one user with several friends who are not linked between themselves. There is one large component which exists from the beginning of the visualization.

Coloring the nodes by age reveals more information on the graph. Some components of the graph were created in a relatively short time frame. Others, such as the large component on the right, grow continuously.

Note how the algorithm manages to compute a stable, mental-map preserving layout of the dynamic graph sequence while at the same time providing meaningful layouts from which the insights discussed above can be extracted. This is especially challenging due to the large growth of the network in the period visualized. A movie showing the visualization is available in the supplementary material.

IX. Conclusion

We have presented an online algorithm for dynamic layout of graphs, whose goal is to efficiently compute stable and aesthetic layouts. The algorithm has several key ideas. First, a good initial layout is computed. Second, the allowed displacement of nodes is controlled according to the changes applied to the graph. In particular, each node is assigned an individual convergence schedule. Third, the global interactions in the graph are approximated in order to maintain the structure of the graph and compute an aesthetic layout. Fourth, a multi-level scheme is used in order to compute high-quality layouts. Last but not least, the GPU is used to accelerate the algorithm, requiring the representation of unstructured graphs in an ordered manner that fits the GPU.

It has been demonstrated that the algorithm computes an aesthetic layout, while reducing displacement and maintaining the user's mental map between layout iterations. Our GPU implementation of the algorithm performs up to 17 times faster than the CPU version. We have applied our algorithm to the visualization of discussion threads on the Internet and to social network visualization.

There are several avenues for future research. An interesting research direction is the extension of the algorithm to drawing multi-level clustered graphs. Finding ways to implement more parts of the algorithm on the GPU will help accelerate the computation. Improving the algorithm used for morphing between layouts can further help in maintaining the mental map.

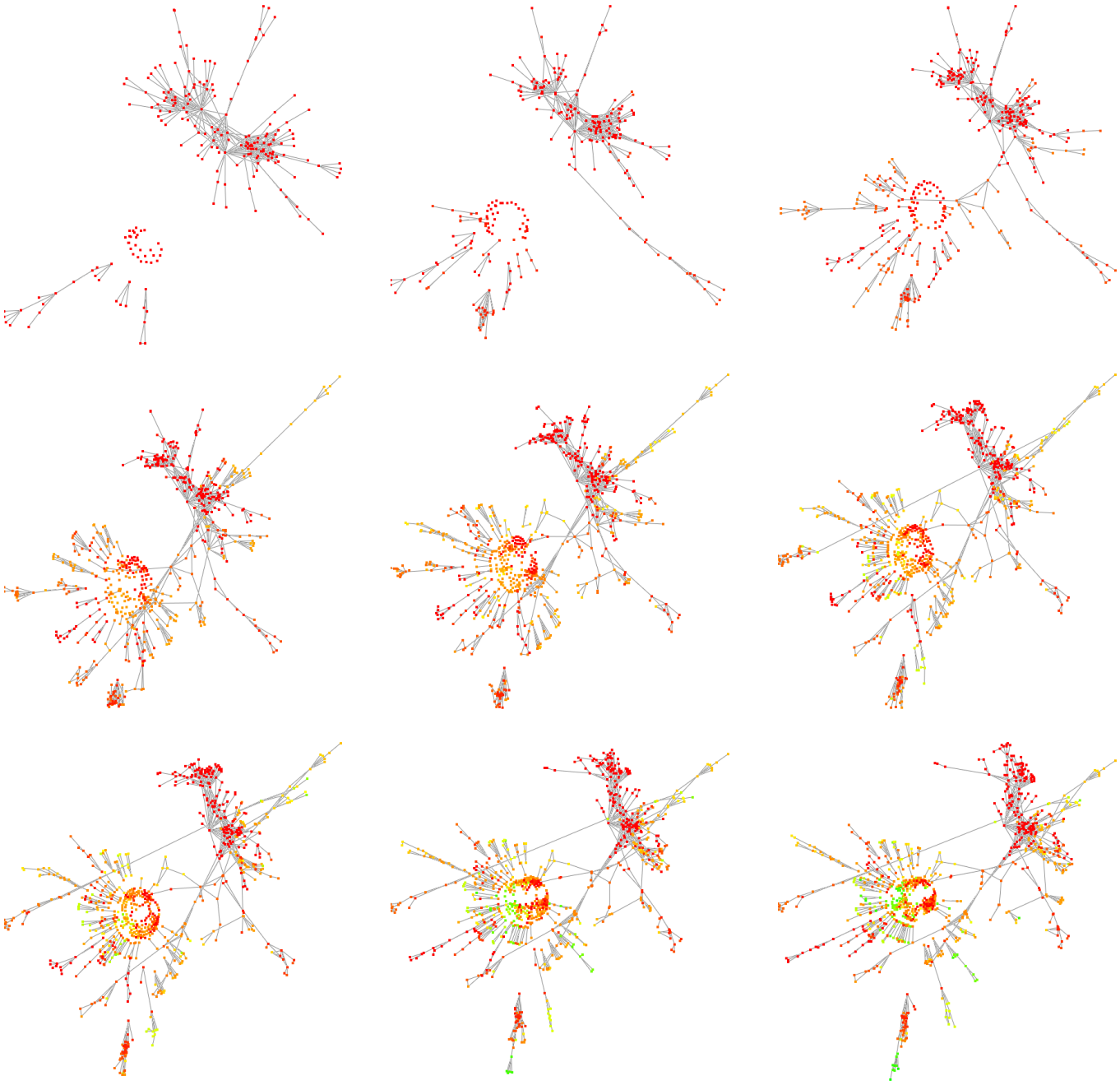


Fig. 9. Snapshots from the Rimzu graph sequence, visualizing the social network at http://www.rimzu.com, left to right, top to bottom. Nodes represent users and edges represent connections between users. In the visualization the graph grows from V=216, E=544 to V=962, E=1561. Nodes are colored by age in a $red \rightarrow yellow \rightarrow green$ scale.

Acknowledgements

This work was partially supported by European FP6 NoE grant 506766 (AIM@SHAPE), by the Israeli Ministry of Science, Culture & Sports, grant 3-3421 and by the Fund for The Promotion of Research at the Technion. We thank Miri Ben-Chen and Danny Leshem for providing the Rimzu social network data.

References

[1] M. Kaufmann and D. Wagner, Eds., *Drawing Graphs: Methods and Models*, 2001.

[2] S. C. North, "Incremental layout in dynadag," in *Proc. 3rd Int. Symp. Graph Drawing*, ser. LNCS, no. 1027, 1995, pp. 409–418.

[3] S. Diehl and C. Gorg, "Graphs, They Are Changing - Dynamic Graph Drawing for a Sequence of Graphs," ser. LNCS, no. 2528, 2002, pp. 23–31.

[4] K. Misue, P. Eades, W. Lai, and K. Sugiyama, "Layout adjustment and the mental map," *J. Visual Languages and Computing*, vol. 6, no. 2, pp. 183–210, 1995.

[5] C. Erten, P. J. Harding, S. G. Kobourov, K. Wampler, and G. V. Yee, "GraphAEL: Graph animations with evolving layouts," in *Proc. 11th Int. Symp. Graph Drawing*, 2003, pp. 98–110.

[6] G. Kumar and M. Garland, "Visual exploration of complex time-varying graphs," *IEEE Trans. on Visualization and Computer Graphics, Proc. InfoVis*, 2006.

[7] Y. Frishman and A. Tal, "Dynamic drawing of clustered graphs," in *Proc. of the IEEE Symp. on Information Visualization, InfoVis*, 2004, pp. 191–198.

[8] Y.-Y. Lee, C.-C. Lin, and H.-C. Yen, *Mental Map Preserving Graph Drawing Using Simulated Annealing*, ser. Conferences in Research and Practice in Information Technology, 2006, vol. 60.

[9] Y. Frishman and A. Tal, "Online dynamic graph drawing," in *Eurographics / IEEE VGTC Symposium on Visualization (EuroVis)*, 2007, pp. 75–82.

[10] I. G. Tollis, G. D. Battista, P. Eades, and R. Tamassia, *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[11] S. Hachul and M. Jünger, "An experimental comparison of fast algorithms for drawing general large graphs," in *Graph Drawing*, ser. LNCS, vol. 3843, 2005, pp. 235–250.

[12] C. Walshaw, "A Multilevel Algorithm for Force-Directed Graph Drawing," *J. Graph Algorithms Appl.*, vol. 7, no. 3, pp. 253–285, 2003.

[13] D. Harel and Y. Koren, "A Fast Multi-Scale Algorithm for Drawing Large Graphs," *J. Graph Algorithms Appl.*, vol. 6, no. 3, pp. 179–202, 2002.

[14] A. J. Quigley and P. Eades, "FADE: Graph drawing, clustering, and visual abstraction," in *Proc. 8th Int. Symp. Graph Drawing, GD*, ser. LNCS, no. 1984, 2000, pp. 197–210.

[15] P. Gajer, M. T. Goodrich, and S. G. Kobourov, "A multi-dimensional approach to force-directed layouts of large graphs," *Comput. Geom*, vol. 29, no. 1, pp. 3–18, 2004.

[16] S. Hachul and M. Jünger, "Drawing large graphs with a potential-field-based multilevel algorithm." in *Graph Drawing*, 2004, pp. 285–295.

[17] Y. Koren, L. Carmel, and D. Harel, "Drawing huge graphs by algebraic multigrid optimization," *Multiscale Modeling & Simulation*, vol. 1, no. 4, pp. 645–673, 2003.

[18] D. Harel and Y. Koren, "Graph drawing by high-dimensional embedding," *J. Graph Algorithms Appl*, vol. 8, no. 2, pp. 195–214, 2004.

[19] U. Brandes, D. Fleischer, and T. Puppe, "Dynamic spectral layout of small worlds," in *Proc. 13th Int. Symp. Graph Drawing, GD*, 2005, pp. 25–36.

[20] U. Brandes and D. Wagner, "A Bayesian paradigm for dynamic graph layout," in *Proc. 5th Int. Symp. Graph Drawing, GD*, 1997, pp. 85–99.

[21] J. Moody, D. McFarland, and S. Bender-deMoll, "Dynamic network visualization," *American Journal of Sociology*, vol. 110, no. 4, pp. 1206–1241, 2005, `http://www.journals.uchicago.edu/AJS/-journal/issues/v110n4/080349/080349.html`.

[22] C. Görg, P. Birke, M. Pohl, and S. Diehl, "Dynamic graph drawing of sequences of orthogonal and hierarchical graphs," in *Proc. 12th Int. Symp. Graph Drawing, GD*, ser. LNCS, vol. 3383, 2004, pp. 228–238.

[23] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Eurographics*, 2005, pp. 21–51.

[24] V. Pande, "Folding@home on ATI GPU's," 2006, http://folding.stanford.edu/FAQ-ATI.html.

[25] E. Tejada and T. Ertl, "Large Steps in GPU-based Deformable Bodies Simulation," *Simulation Modelling Practice and Theory*, vol. 13, pp. 703–715, 2005.

[26] J. Georgii, F. Echtler, and R. Westermann, "Interactive simulation of deformable bodies on gpus." in *SimVis*, 2005, pp. 247–258.

[27] L. Nyland, M. Harris, and J. Prins, "The rapid evaluation of potential fields using programmable graphics hardware," in *ACM Workshop on General Purpose Computing on Graphics Hardware*, 2004.

[28] I. Stephen, T. Munzner, and M. Olano, "Glimmer: Multilevel MDS on the GPU," University of British Columbia, Tech. Rep. UBC CS TR-2007-15, 2007.

[29] T. Hakamata, T. Caudell, and E. Angel, "Force-directed graph layout using the gpu," in *Supercomputing '06 Workshop "General-Purpose GPU Computing: Practice And Experience"*, 2006.

[30] D. Auber and Y. Chriricota, "Improved efficiency of spring embedders: Taking advantage of GPU programming," in *Visualization, Imaging, and Image Processing*, 2007, pp. 169–175.

[31] Y. Frishman and A. Tal, "Multi-level graph layout on the GPU," *IEEE Trans. on Visualization and Computer Graphics (Proc. InfoVis)*, vol. 13, no. 6, pp. 1310–1317, 2007.

[32] T. M. J. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Software—Practice & Experience*, vol. 21, no. 11, pp. 1129–1164, 1991.

[33] J. D. Cohen, "Drawing graphs to convey proximity: an incremental arrangement method," *ACM Trans. Comput.-Hum. Interact.*, vol. 4, no. 3, pp. 197–229, 1997.

[34] T. Kamada and S. Kawai, "An algorithm for drawing general undirected graphs," *Information Processing Letters*, vol. 31, no. 1, pp. 7–15, 1989.

[35] T. M. Newcomb, *The acquaintance process*. Holt, Rinehart and Winston, 1961.

[36] S. Bender-deMoll and D. A. McFarland, "SoNIA - social network image animator," `http://www.stanford.edu/group/sonia/`.

[37] S. Bender-deMoll and D. McFarland, "The art and science of dynamic network visualization," *Journal of Social Structure*, vol. 7, no. 2, 2006.

[38] C. Walshaw, "graph collection," `http://staffweb.cms.gre.-ac.uk/~c.walshaw/partition/`.

Yaniv Frishman received the B.Sc. degree in computer engineering (Summa cum Laude) from the Technion - Israel institute of technology in 1997 and the Master's degree from the Technion in 2006. He is currently a Ph.D. candidate at the department of Computer Science in the Technion. His research interests include information visualization, graph drawing, general purpose computation on graphics processing units and computer graphics.

Ayellet Tal is an Associate Professor at the department of Electrical Engineering in the Technion. She holds a Ph.D. in Computer Science from Princeton University, an M.Sc. degree (Summa cum Laude) in Computer Science from Tel-Aviv University and a B.Sc. degree (Summa cum Laude) in Mathematics and Computer Science from Tel-Aviv University. Her research interests include computer graphics, information and scientific visualization, computational geometry, and multimedia.