Eurographics/ IEEE-VGTC Symposium on Visualization (2007)
Ken Museth, Torsten Möller, and Anders Ynnerman (Editors)

Online Dynamic Graph Drawing

Yaniv Frishman[†1] and Ayellet Tal[‡2]

[1]Department of Computer Science, Technion, Israel
[2]Department of Electrical Engineering, Technion, Israel

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 8 *compared to the CPU implementation. An application to visualization of discussion threads in Internet sites is provided.*

Categories and Subject Descriptors (according to ACM CCS):
I.3.8 [Computer graphics]: Applications H.4.3 [Information systems applications]: Communications applications

1. Introduction

Graph drawing addresses the problem of constructing geometric representations of graphs [KW01]. It has applications in a variety of areas, including software engineering, software visualization, databases, information systems, decision support systems, biology, and chemistry.

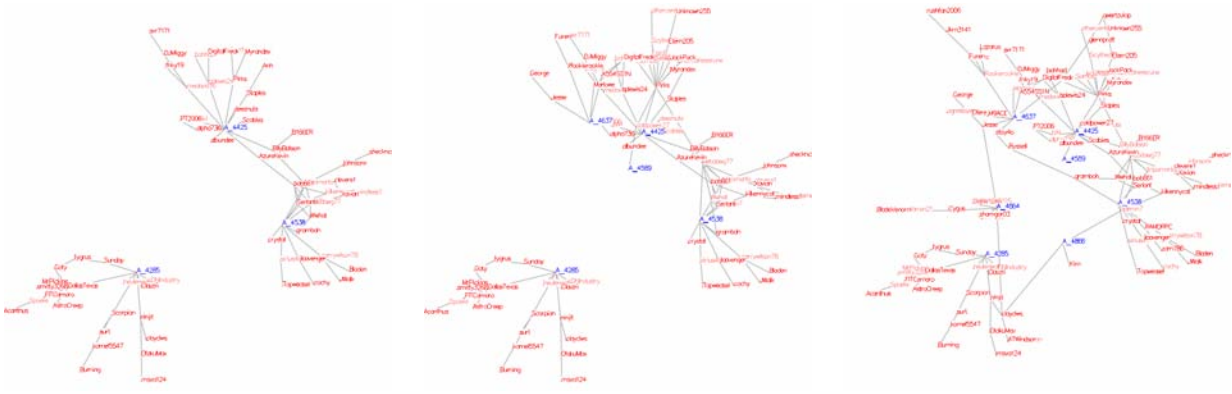
Many applications require the ability of *dynamic graph drawing*, i.e., the ability to modify the graph [Nor95, DG02, KW01], as illustrated in Figure 1 . Sample applications include financial analysis, network visualization, security, social networks, and software visualization. The challenge in dynamic 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* [MELS95] or *dynamic stability* [Nor95].

Most existing algorithms address the problem of offline dynamic graph drawing, where the entire sequence of graphs to be drawn is known in advance [DG02, EHK*03, KG06]. This gives the layout algorithm information about future changes in the graph, which allows it to optimize the layouts generated across the entire 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 [FT04, LLY06].

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 1.12 seconds per layout. Finally, the algorithm is applied to 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.

† e-mail: frishman@tx.technion.ac.il
‡ e-mail: ayellet@ee.technion.ac.il



(a) (b) (c)

Figure 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.*

2. Related Work

Various methods for graph drawing have been proposed [KW01]. Our algorithm builds on force directed layout [KW01], where forces are applied to nodes according to the graph structure and the layout is determined by convergence to a minimum stress configuration. To accelerate the force directed layout, several approaches have been proposed. These attempt to reduce the number of calculations performed, by using multiple levels of detail to represent the graph [HK02, Wal03, HJ04, KCH03, BH86, ATAM04].

Dynamic layout using force directed methods is introduced in [BW97]. Several algorithms address the problem of offline dynamic graph drawing, where the entire sequence is known in advance. In [DG02], a meta-graph built using information from the entire graph sequence, is used in order to maintain the mental map. In [KG06] a stratified, abstracted version of the graph is used to expose its underlying structure. An offline force directed algorithm is used in [EHK*03] in order to create 2D and 3D animations of evolving graphs. Creating smooth animation between changing sequences of graphs is addressed in [BFP05].

An online algorithm is discussed in [LLY06], where a cost function that takes both aesthetic and stability considerations into account, is defined and used. Unfortunately, computing this function is very expensive (45 seconds for a 63 node graph). Drawing constrained graphs has also been addressed. Incremental drawing of DAGs is discussed in [Nor95]. In [FT04] dynamic drawing of clustered graphs is addressed. Dynamic drawing of orthogonal and hierarchical graphs is discussed in [GBPD04]. 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 [OLG*05]. Protein folding [Pan06] and simulation of deformable bodies using mass-spring systems [TE05, GEW05] 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 [NHP04], where an approximate force field is used to calculate forces.

3. Algorithm Outline

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 [MELS95]. 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 [KW01].

Among the different classes of graph drawing algorithms, the force directed algorithm class [KW01] 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 [FR91] . This is suitable for use in dynamic layout, where nodes have different scales of movement.

Our algorithm utilizes several key ideas. 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, allowing 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. The above are performed in order to maintain the mental map. In addition, in order to reduce the layout time while maintaining layout quality, forces from distant nodes are approximated. Finally, the GPU is used to accelerate the layout.

Given a sequence of graphs $G_0, \ldots G_n$, our algorithm computes layouts $L_0, \ldots L_n$ using the following stages:

1. Initialization: compute layout L_0.
2. Merging: Merge layout L_{i-1} and graph G_i to produce an initial layout.
3. Pinning: Assign *pinning weights* to the nodes. These weights control the allowed displacement of each node.
4. Geometric partitioning: The nodes in the graph's initial layout are partitioned, in order to allow us to perform approximate calculations on partitions instead of on individual nodes.
5. Layout: a modified force directed algorithm is used to compute the final layout L_i.
6. Animation: smoothly morph L_{i-1} into L_i, goto step 2.

Computing layout L_0 involves executing steps 1, 4 and 5. Subsequent layouts involve steps 2-6.

4. Algorithm

This section describes the algorithm in detail.

Initialization (Step 1): L_0 is computed using a multilevel force-directed layout scheme, where coarser representations of the graph are recursively built. At each level, given a fine graph, a coarser representation is constructed using a series of *edge collapse* operations. A collapse operation replaces 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 weights of the edges are updated accordingly. (The initial weight of a node/edge is 1.) The algorithm is similar to [Wal03]. However, the order of the edge collapse operations is different: 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 a 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 the graph by "uniformly" collapsing highly connected nodes.

The coarsest graph is laid out using the Kamada-Kawai algorithm [KK89]. It is not used on the finer graphs since it is expensive. However, it gives good results on the coarse graph. To recursively lay the finer graphs, a series of interpolations and layouts is performed, as described in Steps 4–5, until the initial, finest graph is laid out.

Merging (Step 2): Computing a good initial position is vital for reducing the layout time and maintaining dynamic stability [Coh97, GGK04]. 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 G_{i-1}. This procedure is performed in a BFS manner, starting from the positioned nodes. The nodes that cannot be placed by this procedure are placed in a circle around $center(L_{i-1})$.

A *Positioning score* $\Gamma(v)$ is assigned to each node, based on the method used to position it. 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, and at the center of L_{i-1}, respectively. These scores indicate the "confidence" in the node's position and will be used in the next step of the algorithm.

Pinning (Step 3): 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 [FT04,KG06]. 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, as if the layout is static.

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:

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

where in our implementation $\alpha = 0.6$. 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.

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 $D_0, D_1, \cdots, D_{dmax}$.

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:

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

In our implementation $k = 0.5$ and $w_{pin}^{initial} = 0.35$. This assignment creates a decaying effect in which nodes farther away from D_0 are assigned higher pinning weights. 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).

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 5. Third, the algorithm should be fast.

This step creates a KD-tree-type partitioning. Proceeding from coarse to fine, the nodes are partitioned according to their median, alternating between computation of the median of the X and Y coordinates. The recursive subdivision terminates when the size of the subset is below the required partition size.

Layout (Step 5): This step of the algorithm computes the layout. Our algorithm builds on the basic force directed algorithm [FR91], 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 2 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 is shown to correlate with an aesthetic layout.

$frac_{done} = 0$, $K = 0.1$, $t = K * \sqrt{|V|}$, $\lambda = 0.9$
do iteration_count times,
  update partitioning (step4) if required
  parallel_foreach partition $P_i \in P$,
    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 $frac_{done} > w_{pin}(v)$,
    $F_{int}^{repl}(v) = \sum_{u \in P_i, u \neq v} K^2 \frac{pos(v)-pos(u)}{\|pos(v)-pos(u)\|^2}$
    $F_{ext}^{repl}(v) = \sum_{P_j \in P, P_j \neq P_i} K^2 |P_j| \frac{pos(v)-CG(P_j)}{\|pos(v)-CG(P_j)\|^2}$
    $F^{attr}(v) = \sum_{u:(u,v) \in E} \frac{\|pos(u)-pos(v)\|(pos(u)-pos(v))}{K}$
    $F^{total}(v) = F_{int}^{repl}(v) + F_{ext}^{repl}(v) + F^{attr}(v)$
  parallel_foreach node v where $frac_{done} > w_{pin}(v)$,
    $pos_{new}(v) = pos(v) + \frac{F^{total}(v)}{\|F^{total}(v)\|} min(t, \|F^{total}(v)\|)$
  $t* = \lambda$, $frac_{done} += iteration\_count^{-1}$

Figure 2: *Parallel force directed layout algorithm*

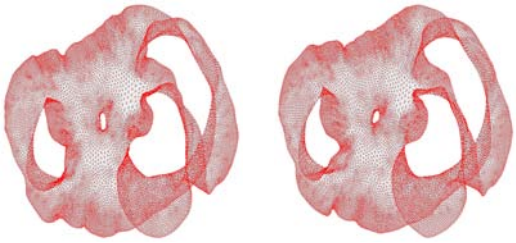
The initialization of the algorithm includes setting the optimal geometric node distance K (that affects the scale of the graph), the initial annealing temperature t, the temperature decay constant λ, and the fraction of the iterations done $frac_{done} \in [0, 1]$.

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 partiton 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 3 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 (step 1), where the nodes may have a high displacement. During the incremental layout, the merge stage (step 3) already gives a good approximation of the final layout.

The key to efficient implementation of this algorithm on the GPU is the use of the *parallel_foreach* loops. Each iteration of the algorithm operates only on the subset of the nodes in G for which $frac_{done} > w_{pin}(v)$. 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. Because the allowed displacement is decreased from one iteration to the next, setting a higher pinning weight limits the total displacement of nodes. Hence, the pinning weight controls the stability of node positions.

The algorithm computes the total force acting on each



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

Figure 3: *Partition size effect on layout, graph 4elt,* $|V| = 15606$, $|E| = 45878$

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 $F_{int}^{repl}, F_{ext}^{repl}$ 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.

The asymptotic complexity of the merging and pinning steps is $O(|E|+|V|)$. The complexity of the partitioning step is $O(|V|*\log(|V|))$: 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 $O(|E|+|V|*(C_s+\frac{|V|}{C_s}))$. This expression is minimized when $C_s = \sqrt{|V|}$, resulting in a total complexity of $O(|E|+|V|^{1.5})$. In many cases $|E| \approx |V|$ and 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 5. We use 50 layout iterations [Wal03].

5. Implementation and Results

This section describes our GPU implementation of the layout stage (step 5, see Figure 2), which is the most time consuming stage, and our results. 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.

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 and their partition number. Each graph node has a corresponding (u,v) index in the texture. 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 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_{int}^{repl} and by the other partitions in the graph – F_{ext}^{repl}. The overall storage complexity is $O(|V|+|E|)$: every node and edge is stored a fixed number of times.

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. 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 partition. 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. Finally, the *anneal* kernel calculates the total force on each node, F^{total}. 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.

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 implies low stress in the graph, corresponding to a good layout. The energy U is derived from the relation $F = \nabla U$. For attractive forces acting on edges, $F^{attr}(\vec{x}) = -K^{-1}\|\vec{x}\|\vec{x} = \nabla U^{attr}$, hence $U^{attr} = -(3K)^{-1}\|\vec{x}\|^3$. For repulsive forces acting between all node pairs, $F^{repl}(\vec{x}) = K^2\vec{x}\|\vec{x}\|^{-2} = \nabla U^{repl}$, hence $U^{repl} = 0.5K^2 log(\|\vec{x}\|^2)$. The total energy is $U^{total} = U^{attr} + U^{repl}$. Other static graph layout quality criteria are indirectly handled by the underlying force directed algorithm.

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

graph	threads1		threads2		3elt		4elt		fe_pwt	
metric	Δpos	U^{total}	Δpos	U^{total}	Δpos	U^{total}	Δpos	U^{total}	Δpos	U^{total}
non-incr	1.34	38.9	1.40	9.65	28.7	$2.73\text{x}10^5$	56.3	$1.01\text{x}10^6$	99.9	$9.59\text{x}10^5$
basic-incr	0.33	39.8	0.32	9.76	1.35	$3\text{x}10^5$	2.58	$9.17\text{x}10^5$	4.65	$8.37\text{x}10^5$
ours	0.05	28.0	0.05	5.76	0.66	$2.72\text{x}10^5$	1.05	$9.87\text{x}10^5$	1.97	$8.26\text{x}10^5$

Table 1: *Layout quality - values are averages for a sequence of layouts*

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.

Several well–known graphs (3elt, 4elt, fe_pwt) are used to demonstrate our algorithm [Wal]. 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 come from real data, discussed in Section 6. Figure 4 shows a few snapshots from the dynamic graph layout of 3elt. Table 1 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 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 (fe_pwt, 4elt) 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 minimizing the movements of the nodes.

For our performance tests we used a PC equipped with a 1.86 GHz Intel Core 2 Duo CPU and an NVIDIA 7900GS GPU. Our algorithm was implemented using C++, Cg and OpenGL. Table 2 gives information about the graph sequences and running times. As can be seen in the table, our GPU implementation provides a significant speedup of up to 8 compared to the CPU. Due to the high ratio of arithmetic operations to memory accesses, the algorithm is compute and not memory bound. Such an algorithm is expected to scale well when using faster, newer GPUs.

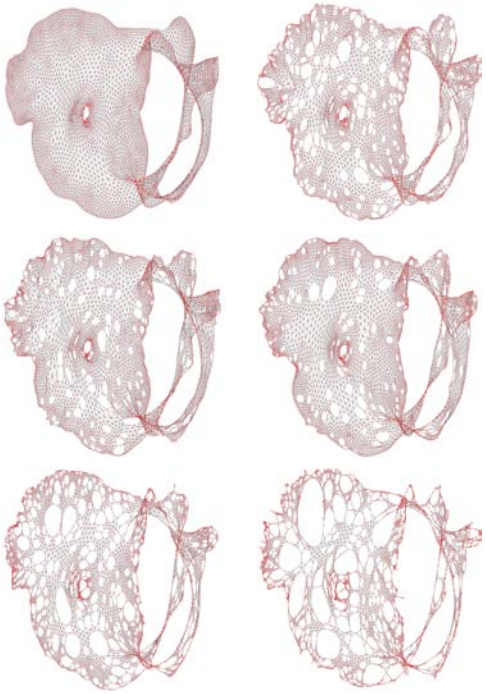


Figure 4: *Snapshots from layouts of the 3elt sequence ($|V| \approx 4000$, $|E| \approx 10,500$), left-to-right, top-to-bottom*

Graph name	avg. |V|	avg. |E|	initial layout GPU	initial layout CPU	incr. layout GPU	incr. layout CPU
3elt	4097	10468	1.51	2.2	0.1	0.34
4elt	14588	40176	2.89	13.27	0.39	2.93
fe_pwt	32045	112395	6.05	35.7	1.12	9.23
g7	9783	19179	1.9	5.94	0.18	0.85

Table 2: *Graph sequence information and running time [sec.]. The last two columns show average incremental layout time for one graph. Total running times for the CPU only and GPU accelerated variant of the algorithm are shown.*

6. 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).

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

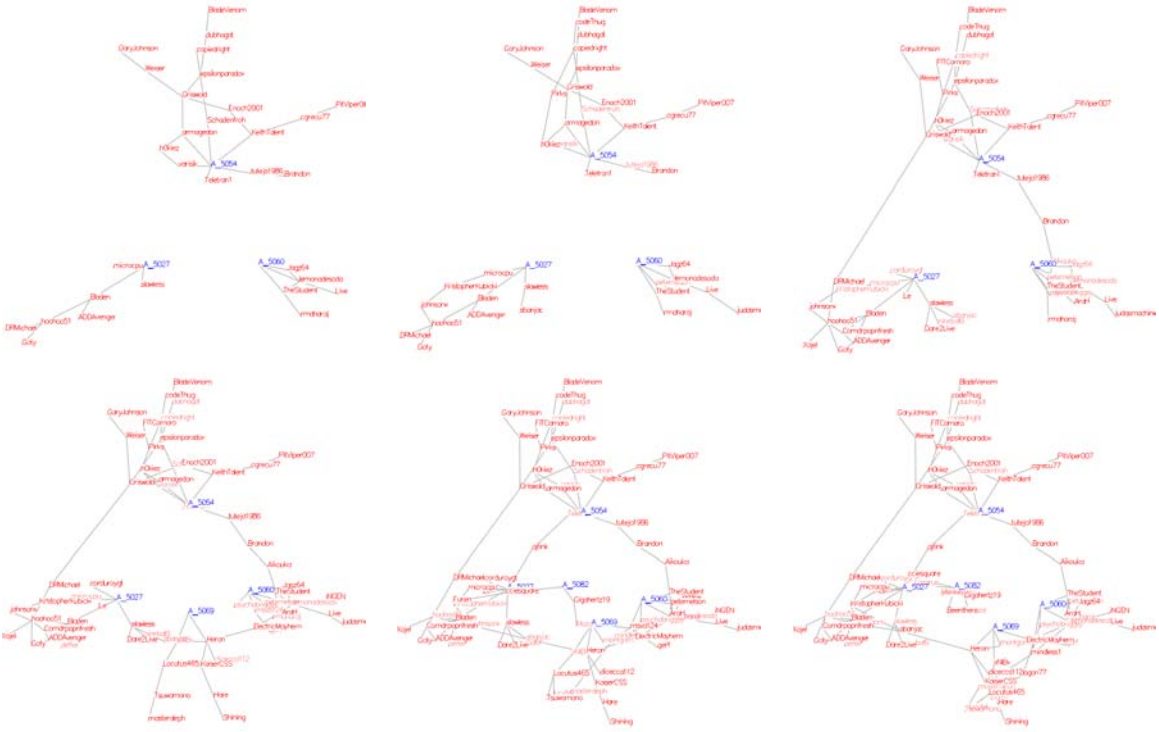


Figure 5: *Snapshots from the threads2 graph sequence, visualizing discussion threads at http://www.dailytech.com, left to right, top to bottom. 109 messages from 86 users in 5 discussion threads are shown. Discussion topics, marked as blue A_n nodes, include computer games (A_5054), nuclear fusion (A_5027),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

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 5 presents several snapshots from the animation sequence showing the evolution of these discussion threads over time.

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.

7. 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. 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 minimizing displacement and maintaining the user's mental map between layout iterations. Our GPU implementation of the algorithm performs up to 8 times faster than the CPU version. We have applied our algorithm to visualization of discussion threads on the Internet.

Acknowledgements

This work was partially supported by European FP6 NoE grant 506766 (AIM@SHAPE) and by the Fund for the Promotion of Research at the Technion.

References

[ATAM04] A. T. ADAI S. V. DATE S. W., MARCOTTE E. M.: Lgl: creating a map of protein function with an algorithm for visualizing very large biological networks. *J. Mol Biol* (2004), 179–190. 2

[BFP05] BRANDES U., FLEISCHER D., PUPPE T.: Dynamic spectral layout of small worlds. In *Proc. 13th Int. Symp. Graph Drawing, GD* (2005), pp. 25–36. 2

[BH86] BARNES J., HUT P.: A hierarchical *O(N* log*N)* force-calculation algorithm. *Nature 324*, 4 (1986), 446–449. 2

[BW97] BRANDES U., WAGNER D.: A Bayesian paradigm for dynamic graph layout. In *Proc. 5th Int. Symp. Graph Drawing, GD* (1997), pp. 85–99. 2

[Coh97] COHEN J. D.: Drawing graphs to convey proximity: an incremental arrangement method. *ACM Trans. Comput.-Hum. Interact. 4*, 3 (1997), 197–229. 3

[DG02] DIEHL S., GORG C.: Graphs, They Are Changing - Dynamic Graph Drawing for a Sequence of Graphs. No. 2528 in LNCS, pp. 23–31. 1, 2

[EHK*03] ERTEN C., HARDING P. J., KOBOUROV S. G., WAMPLER K., YEE G. V.: GraphAEL: Graph animations with evolving layouts. In *Proc. 11th Int. Symp. Graph Drawing* (2003), pp. 98–110. 1, 2

[FR91] FRUCHTERMAN T. M. J., REINGOLD E. M.: Graph drawing by force-directed placement. *Software—Practice & Experience 21*, 11 (1991), 1129–1164. 3, 4

[FT04] FRISHMAN Y., TAL A.: Dynamic drawing of clustered graphs. In *Proc. of the IEEE Symp. on Information Visualization, InfoVis* (2004), pp. 191–198. 1, 2, 3

[GBPD04] GÖRG C., BIRKE P., POHL M., DIEHL S.: Dynamic graph drawing of sequences of orthogonal and hierarchical graphs. In *Proc. 12th Int. Symp. Graph Drawing, GD* (2004), vol. 3383 of *LNCS*, pp. 228–238. 2

[GEW05] GEORGII J., ECHTLER F., WESTERMANN R.: Interactive simulation of deformable bodies on gpus. In *SimVis* (2005), pp. 247–258. 2

[GGK04] GAJER P., GOODRICH M. T., KOBOUROV S. G.: A multi-dimensional approach to force-directed layouts of large graphs. *Comput. Geom 29*, 1 (2004), 3–18. 3

[HJ04] HACHUL S., JÜNGER M.: Drawing large graphs with a potential-field-based multilevel algorithm. In *Graph Drawing* (2004), pp. 285–295. 2

[HK02] HAREL D., KOREN Y.: A Fast Multi-Scale Algorithm for Drawing Large Graphs. *J. Graph Algorithms Appl. 6*, 3 (2002), 179–202. 2

[KCH03] KOREN Y., CARMEL L., HAREL D.: Drawing huge graphs by algebraic multigrid optimization. *Multiscale Modeling & Simulation 1*, 4 (2003), 645–673. 2

[KG06] KUMAR G., GARLAND M.: Visual exploration of complex time-varying graphs. *IEEE Trans. on Visualization and Computer Graphics, Proc. InfoVis* (2006). 1, 2, 3

[KK89] KAMADA T., KAWAI S.: An algorithm for drawing general undirected graphs. *Information Processing Letters 31*, 1 (1989), 7–15. 3

[KW01] KAUFMANN M., WAGNER D. (Eds.): *Drawing Graphs: Methods and Models*. 2001. 1, 2

[LLY06] LEE Y.-Y., LIN C.-C., YEN H.-C.: *Mental Map Preserving Graph Drawing Using Simulated Annealing*, vol. 60 of *Conferences in Research and Practice in Information Technology*. 2006. 1, 2

[MELS95] MISUE K., EADES P., LAI W., SUGIYAMA K.: Layout adjustment and the mental map. *J. Visual Languages and Computing 6*, 2 (1995), 183–210. 1, 2

[NHP04] NYLAND L., HARRIS M., PRINS J.: The rapid evaluation of potential fields using programmable graphics hardware. In *ACM Workshop on General Purpose Computing on Graphics Hardware* (2004). 2

[Nor95] NORTH S. C.: Incremental layout in dynadag. In *Proc. 3rd Int. Symp. Graph Drawing* (1995), no. 1027 in LNCS, pp. 409–418. 1, 2

[OLG*05] OWENS J. D., LUEBKE D., GOVINDARAJU N., HARRIS M., KRÜGER J., LEFOHN A. E., PURCELL T. J.: A survey of general-purpose computation on graphics hardware. In *Eurographics* (2005), pp. 21–51. 2

[Pan06] PANDE V.: Folding@home on ati gpu's, 2006. http://folding.stanford.edu/FAQ-ATI.html. 2

[TE05] TEJADA E., ERTL T.: Large Steps in GPU-based Deformable Bodies Simulation. *Simulation Modelling Practice and Theory 13* (2005), 703–715. 2

[Wal] WALSHAW C.: graph collection. http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/. 6

[Wal03] WALSHAW C.: A Multilevel Algorithm for Force-Directed Graph Drawing. *J. Graph Algorithms Appl. 7*, 3 (2003), 253–285. 2, 3, 5