# GLUG: GPU Layout of Undirected Graphs

Stephen Ingram[*]
University of British Columbia

Tamara Munzner[†]
University of British Columbia

Marc Olano[‡]
University of Maryland Baltimore County

October 15, 2007

## Abstract

We present a fast parallel algorithm for layout of undirected graphs, using commodity graphics processing unit (GPU) hardware. The GLUG algorithm creates a force-based layout minimizing the Kamada Kawai energy of the graph embedding. Two parameters control the graph layout: the number of landmarks used in the force simulation determines the influence of the global structure, while the number of near neighbors affects local structure. We provide examples and guidelines for their use in controlling the visualization. Our layouts are of comparable or better quality to existing fast, large-graph algorithms. GLUG is an order of magnitude faster than the previous CPU-based $FM^3$ algorithm. It is considerably faster than the only previous GPU-based approach to force-directed placement, a multi-stage algorithm that uses a mix of CPU partioning and GPU force simulation at each step. While GLUG has a preprocessing stage that runs on the CPU, the core algorithm operates entirely on the GPU.

[*]e-mail: sfingram@cs.ubc.ca
[†]e-mail:tmm@cs.ubc.ca
[‡]e-mail:olano@umbc.edu

## 1 Introduction

Most graph layout algorithms assign cartesian coordinates for each node in a graph by attempting to maximize a set of aesthetic criteria. Some of these criteria, such as minimizing edge crossings, require discrete combinatorial optimization techniques, while others, such as isometry, permit modeling with differentiable functions. Force based layout techniques simplify the minimization of several differentiable cost functions with physical models. Their success derives from the fact that the stable states of the force model generally produce quality layouts and may even contain a state coinciding with the global minimum.

This paper focuses on minimizing the cost function $\sigma$, also called the *stress* or the Kamada Kawai energy, of an embedding of a graph

$$\sigma(\Delta, D, W)^2 = \sum_{ij} w_{ij}(\delta_{ij} - d_{ij})^2$$

Here, $\delta_{ij}$ are elements of the distance matrix $\Delta$, whose entries represent the graph theoretic distance, or shortest number of *hops*, between the nodes $i$ and $j$. The matrix $D$, whose entries are $d_{ij}$, is the distance matrix of the same nodes in the embedding or *screen* space. Finally, $W$ is a matrix of *weights* for fine-tuning the importance of one graph distance or set of distances over another. Commonly, $w_{ij} = 1/d_{ij}^2$. Clearly, when $\sigma = 0$ an isometry is achieved and any nonzero value indicates distortion in the embedding, with larger values of $\sigma$ indicating greater and greater distortion. The rationale for isometry as a good aesthetic criterion is that isometric embeddings exhibit graph symmetry and employ the proximity percept such that nearby/distant graph neighbors are nearby/distant screen neighbors.

To optimize $\sigma$ many graph algorithms differentiate with respect to screen *coordinates* and use gradient descent or function majorization to converge to a local minimum. Alternatively, one can model $\sigma$ with a force model where nodes are particles who experience forces proportional to the derivative of $\sigma^2$ with respect to screen *distance* and layout convergence is detected when the particles reach a steady state. Both techniques are succeptible to local minima, though they are ususally different minima, and are $O(V^2)$ per iteration.

We present GLUG, an algorithm for rapid layout of graphs by minimizing the $\sigma$ function. It achieves high speeds using a $O(V)$ per iteration, force-based particle simulation designed to execute in parallel on a Graphics Processing Unit (GPU). Only one previous system, by Frishman and Tal [1], has used the GPU to accelerate force-based graph layout. Their system addresses dynamic graph layout using a multi-stage algorithm that uses a mix of CPU partitioning and GPU force simulation at each step. While GLUG has a pre-processing stage that runs on the CPU, the core algorithm operates entirely on the GPU. GLUG is relatively simple to implement, and has two parameters that permit users to control the global and local fidelity of a layout. By comparing the visual quality and speed of GLUG with competitive graph layout systems, we find that GLUG achieves visually superior or comparable layouts in times an order of magnitude faster than CPU systems and substantially faster than the recently proposed GPU accellerated method.

## 2 Previous Work

The idea of using a force simulation to compute a graph drawing was popularized by the *spring embedder* of Eades [2]. In this model, edges were replaced with springs and non-adjacent vertices emit repulsive forces. Fruchterman and Reingold (FR) introduced several improvements to this simple model [3] including more forces between vertices for faster convergence and a spatial partitioning scheme for reducing computation. Kamada and Kawai (KK) devised a different formulation than the Eades local spring model, based on the entire system energy [4]. This energy function, introduced above as $\sigma$, is typically more costly to minimize but uses more spatial information from the graph than FR energy. Although many approaches in the graph drawing literature use force simulation for computing FR energy [5, 6, 7, 1], we know of no previous work that uses force simulation for KK energy minimization. Here force simulation refers to using physical analogies to update the entire state of all the interacting particles at each time step using integration.

KK energy is identical to the *stress* function of multidimensional scaling (MDS) [8]. MDS can embed points in arbitrary dimensions, whereas graph drawing is typically restricted to two- or three-dimensional layouts, but the same techniques are applicable to both problems. While the cost of classic MDS is $O(V^3)$, a large number of algorithms have been proposed to reduce the complexity and improve the visual quality. Of particular interest are the stress majorization technique developed by de Leeuw [9] and force-based methods of Chalmers [10] and Ingram [11]. Stress majorization has seen many useful applications and extensions in graph layout and has been lauded for its monotonic convergence properties [12] and the ease of introducing layout constraints [13]. Although the full majorization is $O(V^2)$ per iteration in complexity, Koren [14] introduced a sparsification strategy which reduces runtimes on large graphs while maintaining much of the visual quality. Force-based methods, also $O(V^2)$ time per iteration in naive implementations, have demonstrated rapid reduction of the stress function using particle simulations. Chalmers' linear-time iteration particle system used fixed-size index caches per point which were updated by a stochastic search. GLUG is much faster than all of these previous approaches at minimizing the KK/stress objective function.

Recent work has used the low-cost, widely available parallelism of programmable GPUs to dramatically speed up layout computation. Ingram recently introduced two algorithms for MDS that run entirely on the GPU: the multilevel Glimmer approach, and the single-level GPUSF approach that Glimmer runs as a subsystem[11]. The latter, inspired by the work of Chalmers, was the springboard for GLUG. GLUG is faster than GPUSF for graph layout because eliminates stochastic search, which is cheap for MDS but expensive for graphs, and exploits the given graph topology.

Frishman and Tal's Online Dynamic Graph Drawing [1] is the first approach to graph layout that exploits the GPU. Their algorithm begins with an initial KK layout stage that runs on the CPU. It uses a multi-stage pipeline, where a FR force simulation stage that runs on the GPU is interleaved with a geometric partioning stage that runs on the CPU. This partioning is carried out in screen space, and must be recomputed as the layout progresses. In contrast, GLUG partions only once, based on graph-theoretic distance. After this linear-time preprocessing step is run on the CPU, the core GLUG algorithm runs entirely on the GPU, thus achieving greater speed and simplicity.

# 3 GLUG Algorithm

We begin with a discussion of general purpose programming on GPUs. We then give an overview of the GLUG algorithm, present the CPU-based preprocessing stage, and finish with the full details of the entirely GPU-based core force simulation.

## 3.1 General Purpose Parallel Computation on GPUs

Modern GPUs include a pipeline of programmable processing stages, each of which is highly parallel. We primarily use the pixel stage, which runs a program, or shader, on a stream of pixels. The GPU pixel processors can be considered as a single-instruction multiple data (SIMD) unit operating in parallel on a subset of pixels in the stream, where the SIMD size varies from 16 to 1024 in recent GPUs. This unit has random read access to data stored in texture memory, so textures can be used in place of arrays. Computation occurs when a textured polygon is rendered using a shader. Typical computations take multiple rendering passes, where the only communication channel between processing units is writing a texture in one pass, then reading from it in a later pass.

GLUG is designed to exploit GPU parallelism. Although it could be executed on the CPU, the speed benefits would be minimal. Like its predecessors, GLUG does not depend on specific hardware features of a particular GPU, running on any card that supports pixel shaders.

## 3.2 Overview

GLUG works through a single-level force simulation where each node in the graph is represented by an infinitesimal particle of unit mass. Forces are computed proportional to the discrepancy between graph-theoretic distance and screen distance between a single node and the nodes in two sets. The first set, *Near*, contains a point's $K$ nearest neighbor nodes and is typically unique to each point. The second set *Landmark* is shared by all the nodes in the graph and contains a set of $l$ landmark nodes. The size and contents of both sets are computed in $O(V+E)$ time on unweighted graphs and fixed at the beginning of the algorithm.

GLUG is strongly influenced by the GPUSF/Chalmers algorithms for MDS. The key difference is the removal of the stochastic search strategy. In an MDS context, where pairwise distances between points are either precomputed or cheap to compute, the stochastic search algorithm incurs no penalty. However, in graph layout, computing an arbitrary distance $\delta_{ij}$ can have a high cost, requiring breadth first search or Dijkstra's algorithm. We fix the *Near* and *Landmark* sets, and compute their contents in a preprocessing step.

The idea of using a resticted number of nodes in the computation to minimize $\sigma^2$ is also used by Koren in a majorization-based technique[14]. In his algorithm, however, the nearest neighbor set size is determined by all the nodes within a fixed distance and not by set cardinality. This unbounded sizing does not readily translate to GPU computation.

## 3.3 GLUG Preprocessing

GLUG preprocessing consists of two steps executed on the CPU. The first step is the computation of $L$ landmarks and their corresponding shortest-path distances to the other nodes. Computation of distances is done using a breadth first search on an unweighted graph or Dijkstra's algorithm on a weighted graph. The selection of landmarks is done using the approximation algorithm to the k-centering problem proposed with HDE [15]. The second step is the computation of $K$ nearest neighbors. This computation is done with a depth-limited graph search from each vertex which can be done in a constant number of operations on an unweighted graph.

Preprocessing yields a sparse non-symmetric distance matrix $\Delta$. The landmark preprocessing computes entire columns of the matrix while the nearest neighbor preprocessing computes $K$ entries per row. We apply the following strategy to efficiently pack the sparse matrix into texture memory that can be easily processed by a pixel shader. We allocate two textures *Index* and *Delta*. The $u$ and $v$ coordinates of any texel indicates the source vertex index. The content of an $Index$ texel gives the destination vertex index, and the value of the texel at the same position in *Delta* contains the distance between the source and destination vertex.

## 3.4 Parallel Force Simulation

GPU algorithms use texture memory to store program data. Efficient organization of program data on this memory is important for fast and simple pixel shader code. One important feature of GLUG is the exclusive use of fixed-size data structures. This choice makes the organization of program data in texture memory very straightforward and permits us to eliminate any dynamic resizing or reorganization of program data, thus avoiding shader program complexity. In contrast, existing graph-layout sparsification strategies all rely on variable size data structures [14] or CPU-based repartitioning [1].

The force simulation divides its information across the following textures of $O(V)$ size

- *Index* - indices of Near and Landmark vertices used by each vertex in the simulation, contents are static and computed during preproccessing
- *Delta* - graph theoretic distances between each vertex and the Near and Landmark vertices pointed to in $Index$, contents are static and computed during preprocessing
- *Distance* - embedding or screen distances between each vertex and the vertices pointed to in *Index*
- $Force$ - the force vector for each vertex
- $Velocity$ - the velocity vector for each vertex
- *Embedding* - the embedding coordinates for each vertex, initialized randomly within a bounding box

The simulation is divided into five stages, one for each texture updated in a rendering pass or series of passes.

**Stage 1** GLUG first calculates the Euclidean distance between each vertex and the vertices in the index set, using a single fragment shader that renders to *Distance* and reads from *Embedding*.

**Stage 2** We then calculate the force vector to apply to each vertex in two parts. First, we compute the normalized vector betwen each vertex and the vertices in its index sets. These vectors are then scaled by the difference between corresponding values from *Delta* and *Distance* and subsequently damped with values read from $Velocity$. Damping is designed to inhibit excessive particle oscillation and improve convergence. The GLUG damping scheme computes the relative velocity vector between each vertex and its indexed vertices and subtracts it from the force vector between these vertices. Second, the sum of all these damped force vectors is computed and stored in $Force$. This stage is computed in $\lceil (K+L)/4 \rceil$ rendering passes, one pass for each set of four vertices in the index sets. The division by 4 is a side effect of the organization of texel data into groups of 4 floats, corresponding to the RGBA or XYZW quanities used in graphics programming.

**Stage 3** We compute the velocity of the vertex into $Velocity$ by integrating the force vector from $Force$ in a single pass. We use a simple Euler integration scheme where the current velocity is simply updated by adding the force times a desired timestep.

**Stage 4** We update the *Embedding* texture by integrating the velocity vectors using the integration scheme from the previous step.

**Stage 5** At the end, for a termination condition we check whether the velocities of the moving nodes have converged to a stable value, using the same approach as Glimmer [11]. We compute the normalized sum of point velocities in $\log_4(V)$ sum-reduce rendering passes. The velocity is low-pass filtered to avoid spring oscillation problems, and computation is terminated when the filtered velocity falls below a threshold set to 1/1000 of the maximum velocity observed.

Note that unlike many force simulations, forces are asymetrically applied. That is, a given force vector calculated between two vertices $A$ and $B$ that is applied to a vertex $A$ is not guaranteed to also be applied to $B$ unless $B$ also contains $A$ in its index set. Implementing such a strategy on the GPU would require a *scatter* or random-access write operation, unsupported by current GPU architectures without costly workarounds.

# 4 Results and Discussion

We analyze the complexity of GLUG, examine the behavior of its two parameters, and compare its speed and quality on several graphs to previous work.

## 4.1 Complexity

GLUG preprocessing consists of solving the single source shortest path problem once for each landmark chosen and so the complexity is different for weighted and unweighted graphs. We can use a $O(V + E)$ breadth first search for unweighted graphs, or the $O(V \log V + E)$ Dijkstra's algorithm for weighted graphs. Likewise, we compute the $K$ neighbors by limiting the number of vertices accessed in a breadth first search to $K$. This technique produces a constant number of operations per vertex and so has $O(V)$ time complexity. Naturally, this only computes the true nearest neighbors in the unweighted case.

Each GLUG iteration consists of several texture rendering passes whose cost time is proportional to the number of pixels processed. Each rendered texture is of $O(V)$ size and is rendered a constant number of times using pixel shaders with a constant number of instructions, implying that each iteration of GLUG is of $O(V)$ time. While GPU parallelism does not affect the asymptotic complexity of the algorithm, it theoretically provides a speedup factor proportional to the number of processors specific a given graphics card. Current GPU hardware is has up to 1024 parallel processors for shader programs

We consider the complexity of GLUG to be the preprocessing complexity plus $O(V)$ per iteration for $C$ iterations equals a total complexity of $O(E + CV)$ in the unweighted case or $O(E + V(C + \log V))$ in the weighted case. $C$ is the total number of iterations required before the force simulation converges. It is well known that $C$ is difficult to analyze [3]. We have observed empirically that for mesh-like graphs, $C$ is roughly $\log(V)$, while $C$ approaches $V$ in graphs with with many cliques or biconnected components.

## 4.2 Near and Landmark Size Parameter Effects

The GLUG algorithms has two adjustable parameters, the number of landmarks and number of nearest neighbors to use in the force calculation. In this section we examine the qualitative and quantitative results of choosing several parameter values on different graphs.

Figure 1 shows the effect of altering the landmark parameter while keeping the number of nearest neighbors fixed. Choosing only 4 landmarks yields faster run-times, but considerable distortion. We have observed that quality is generally good enough when 32 landmarks are chosen.

More interesting is the choice of number of nearest neighbors. This parameter appears to control the "bushiness" of graph layouts. If the nearest neighbor

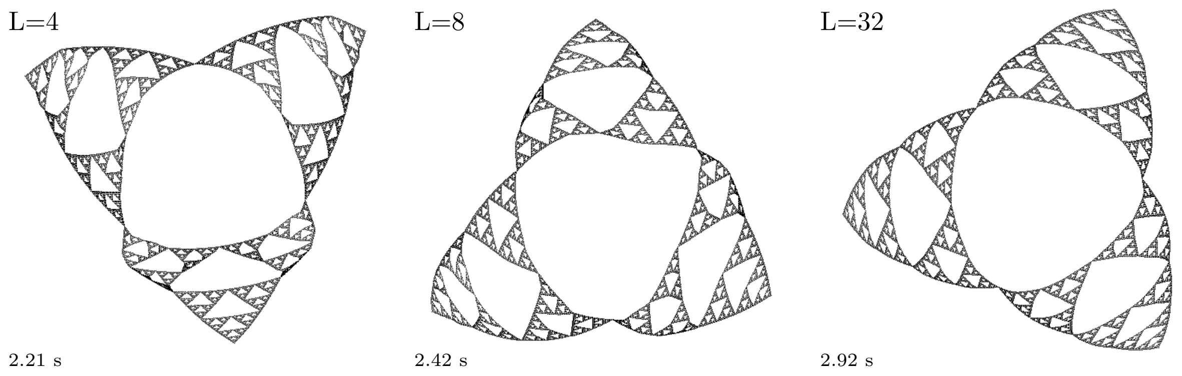


Figure 1: Increasing the number of landmarks $L$ used in the force simulation yields improved global structure of `sierpinski_08` while imposing a slight performance cost.

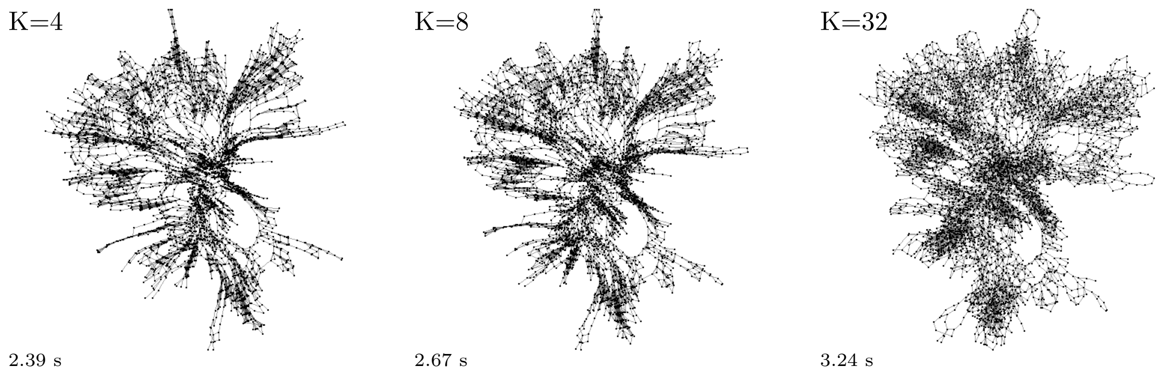


Figure 2: Increasing the number of nearest neighbors $K$ yields improved local structure of `bcspwr_10` while imposing a slight performance cost. Whether the fidelity of local structure improves a layout depends on the graph and the goals of the visualization.

set size is small enough, large branches on trees will not have enough spatial information to avoid occluding each other. Figure 2 shows that increasing the nearest neighbor parameter while keeping the number of landmarks fixed eventually obscures the appearance of a level-set structure of nodes in the graph. We propose that the nearest neighbor parameter should be changed to suit the aims of the underlying visualization.

## 4.3 Experimental Comparison

Following Hachul and Jünger's experimental comparison of algorithms for large graph layout, we compare GLUG to previous work using both wall clock time and qualitative comparison of graph drawings [7]. Table 1 shows timings and sizes for the graphs we tested, alongside performance figures quoted from the $FM^3$ [7] and Frishman and Tal [1] papers.

### 4.3.1 Experimental Setup

Our algorithm was executed on a 3.1 GHz Pentium 4 CPU running WindowsXP with a NVIDIA GeForce

| Name | $\|V\|$ | $\|E\|$ | K | L | Preproc | Sim | GLUG Total | $FM^3$ Total | ODGD |
|---|---|---|---|---|---|---|---|---|---|
| sierpinski_08 | 9843 | 19683 | 8 | 32 | 0.19 | 0.39 | 0.58 | 16.8 | |
| crack | 10240 | 30380 | 8 | 32 | 0.282 | 0.42 | 0.66 | 23.0 | |
| bcsstk_33 | 8738 | 291583 | 8 | 32 | 0.34 | 0.53 | 0.88 | 23.8 | |
| ug_380 | 1104 | 3231 | 8 | 32 | 0.02 | 0.94 | 0.95 | 2.1 | |
| add_32 | 2075 | 9462 | 8 | 32 | 0.06 | 0.92 | 0.98 | 12.1 | |
| dg_1087 | 7602 | 7601 | 8 | 32 | 0.13 | 0.91 | 1.03 | 18.1 | |
| spider_C | 10000 | 22000 | 8 | 32 | 0.19 | 1.13 | 1.31 | 16.41 | |
| bcsstk_31 | 35586 | 572913 | 8 | 32 | 1.16 | 0.86 | 2.06 | 83.6 | |
| ubc | 40011 | 191659 | 8 | 32 | 1.33 | 1.19 | 2.52 | 84.56 | |
| bcsstk_32 | 44609 | 985046 | 8 | 32 | 1.64 | 2.53 | 4.17 | 110.9 | |
| sierpinski_10 | 88575 | 177147 | 8 | 32 | 2.44 | 1.89 | 4.28 | 162.0 | |
| finan_512 | 74752 | 261120 | 8 | 32 | 2.17 | 2.55 | 4.67 | 158.2 | |
| fe_ocean | 143437 | 409593 | 8 | 32 | 4.15 | 3.14 | 7.203 | 355.9 | |
| snowflake_B | 9701 | 9700 | 8 | 64 | 0.20 | 7.05 | 7.28 | 166.5 | |
| flower_C | 90030 | 1308441 | 4 | 32 | 2.17 | 10.72 | 12.89 | 121.4 | |
| 3elt | 4720 | 13722 | 8 | 32 | 0.06 | 0.39 | 0.45 | | 1.51 |
| fe_pwt | 36519 | 144794 | 8 | 32 | 0.88 | 0.83 | 1.70 | | 6.05 |
| 4elt | 15606 | 45878 | 4 | 64 | 0.55 | 1.26 | 1.81 | | 2.89 |

Table 1: GLUG is an order of magnitude faster than $FM^3$ on all large graphs. The pre-processing stage runs on the CPU, the rest of the force-directed simulation runs on the GPU, and the total GLUG time is the sum. The $FM^3$ and Online Dyanmic Graph Drawing timings are quoted from previous work [16, 1].

8800GTX GPU. We surmise that faster CPUs will primarily improve preprocessing times and faster GPUs will improve force simulation times.

### 4.3.2 Running Times

Table 1 compares the layout time for GLUG versus $FM^3$ and Frishman and Tal's approach. We see order of magnitude speed improvements on most large graphs compared to the $FM^3$ performance figures, and significant improvements over Frishman and Tal's approach.

The GPU-based architecture of GLUG provides an alternative to producing static layouts. By using vertex shaders, vertices can be quickly fetched from texture memory and vertex drawing can be done with only a slight performance penalty of 1 to 10 seconds depending on the size of the graph. Drawing every frame of the layout is useful not only for debugging the algorithm, but also permits user detection of convergence if the velocity-based termination condition terminates too soon or too late. The `snowflake_B` and `flower_C` running times reflect manual termination when the layout appeared to converge, because our automatic check caused premature termination in these two cases. Refining our termination check would be interesting future work.

### 4.3.3 Drawings

We argue that GLUG produces graph drawings of comparable, and in some cases superior, quality to existing fast algorithms for drawing large graphs. Figure 3, 4, 5, and 6 show drawings for many of the graphs in Table 1, and we comment below on only a few for space reasons. Readers are encouraged to compare these drawings with those in the previous experimental comparisons [16, 17, 1].

Of particular interest is the `finan512` graph. Koren[18] points out that his computationally expensive stress minimization technique successfully shows its microstructure, which is hidden by the spring embedder stragegy of $FM^3$. Figure 5 shows that GLUG reveals the full symmetry and microstructure of the graph with a layout far faster than previous iterative graph drawing methods methods.

On the other hand, the `dg_1087` graph highlights a shortcoming of the GLUG sparsification strategy with regard to trees with high degree edges. In such graphs, leaf nodes tend to get artificially lumped together.

## 5 Conclusion

GLUG is a faster alternative to existing force-based graph drawing systems. While previous graph drawing work has interleaved GPU and CPU stages to partially accelerate force simulations, we provide a simpler approach which requires only a single CPU preprocessing step and then runs entirely on the GPU, converging in less time. The quality of the graphs is comparable to more computationally demanding CPU algorithms. GLUG can readily be incorporated into a multilevel scheme and we believe this will be a fruitful direction for future work.

## References

[1] Frishman, Y., Tal, A.: Online dynamic graph drawing. In: Proc. Eurographics/IEEE VGTC Symp. on Visualization (EuroVis'07). (2007)

[2] Eades, P.A.: A heuristic for graph drawing. In: Congressus Numerantium. Volume 42. (1984) 149–160

[3] Fruchterman, T.M.J., Reingold, E.M.: Graph drawing by force-directed placement. Software - Practice and Experience **21**(11) (1991) 1129–1164

[4] Kamada, T., Kawai, S.: An algorithm for drawing general undirected graphs. Inf. Process. Lett. **31**(1) (1989) 7–15

[5] Brandes, U.: Drawing on physical analogies. In Kaufmann, M., Wagner, D., eds.: Drawing Graphs: Methods and Models. Springer-Verlag (2001) 71–86

[6] Gajer, P., Kobourov, S.G.: GRIP: Graph dRawing with intelligent placement. In: Proc. Graph Drawing. (2000) 222–228

[7] Hachul, S., Jünger, M.: Drawing large graphs with a potential-field-based multilevel algorithm. In: Proc. Graph Drawing. (2004) 285–295

[8] Borg, I., Groenen, P.: Modern Multidimensional Scaling, Theory and Applications. Springer-Verlag, New York (1997)

[9] de Leeuw, J.: Applications of convex analysis to multidimensional scaling. Recent developments in statistics (1977) 133–145

[10] Chalmers, M.: A linear iteration time layout algorithm for visualising high dimensional data. In: Proc. IEEE Visualization. (1996) 127–132

[11] Ingram, S., Munzner, T., Olano, M.: Glimmer: Multilevel MDS on the GPU. Technical Report TR-2007-13, University of British Columbia (2007)

[12] Gansner, E.R., Koren, Y., North, S.: Graph drawing by stress majorization. In: Proc. Graph Drawing. (2004) pp. 239–250

[13] Dwyer, T., Koren, Y., Marriott, K.: IPSep-CoLa: An incremental procedure for separation constraint layout of graphs. IEEE Transactions on Visualization and Computer Graphics (Proc. InfoVis 06) **12**(5) (2006) 821–828

[14] Koren, Y., Harel, D.: Axis-by-axis stress minimization. In: Proc. Graph Drawing. (2003) pp. 450–459

[15] Harel, D., Koren, Y.: Graph drawing by high dimensional embedding. In: Proc. Graph Drawing. (2002)

[16] Hachul, S., Jünger, M.: An experimental comparison of fast algorithms for drawing general large graphs. In: Proc. Graph Drawing. (2005) 235–250

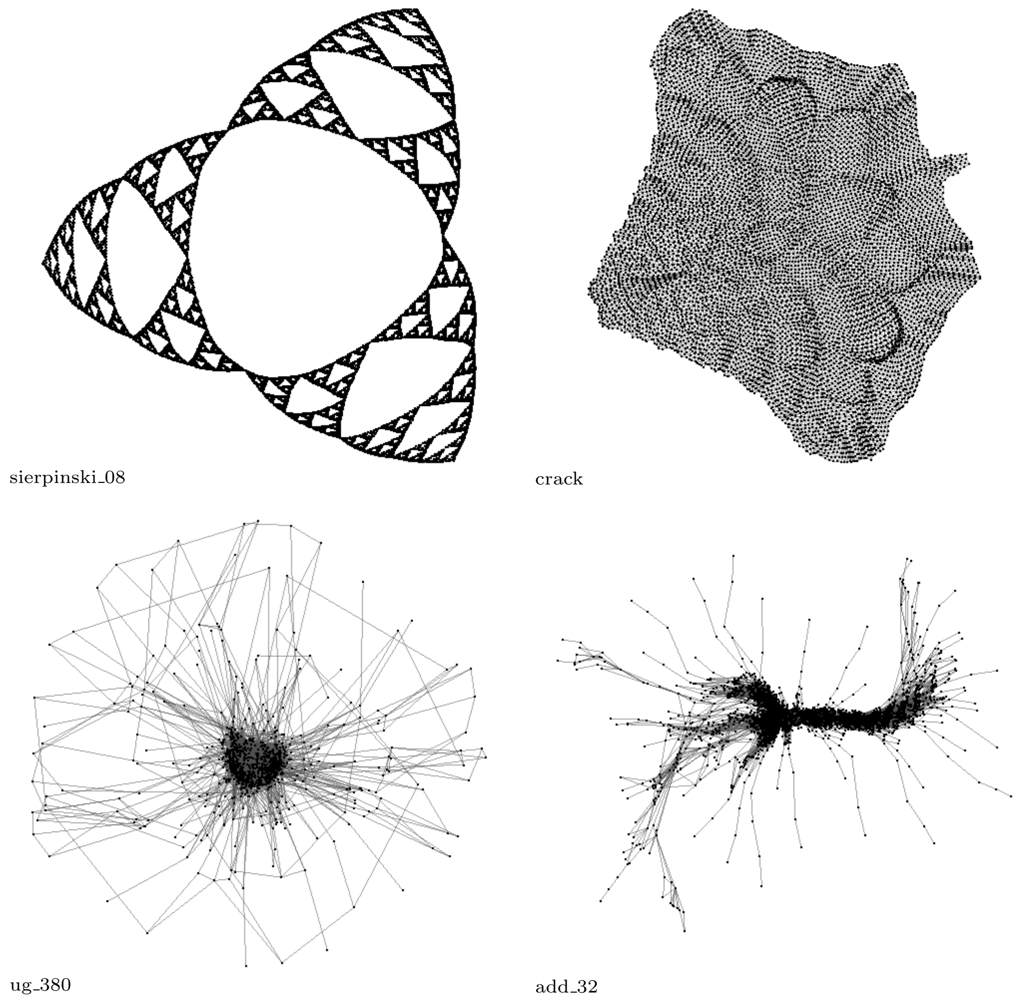


Figure 3: Graph drawings produced by GLUG

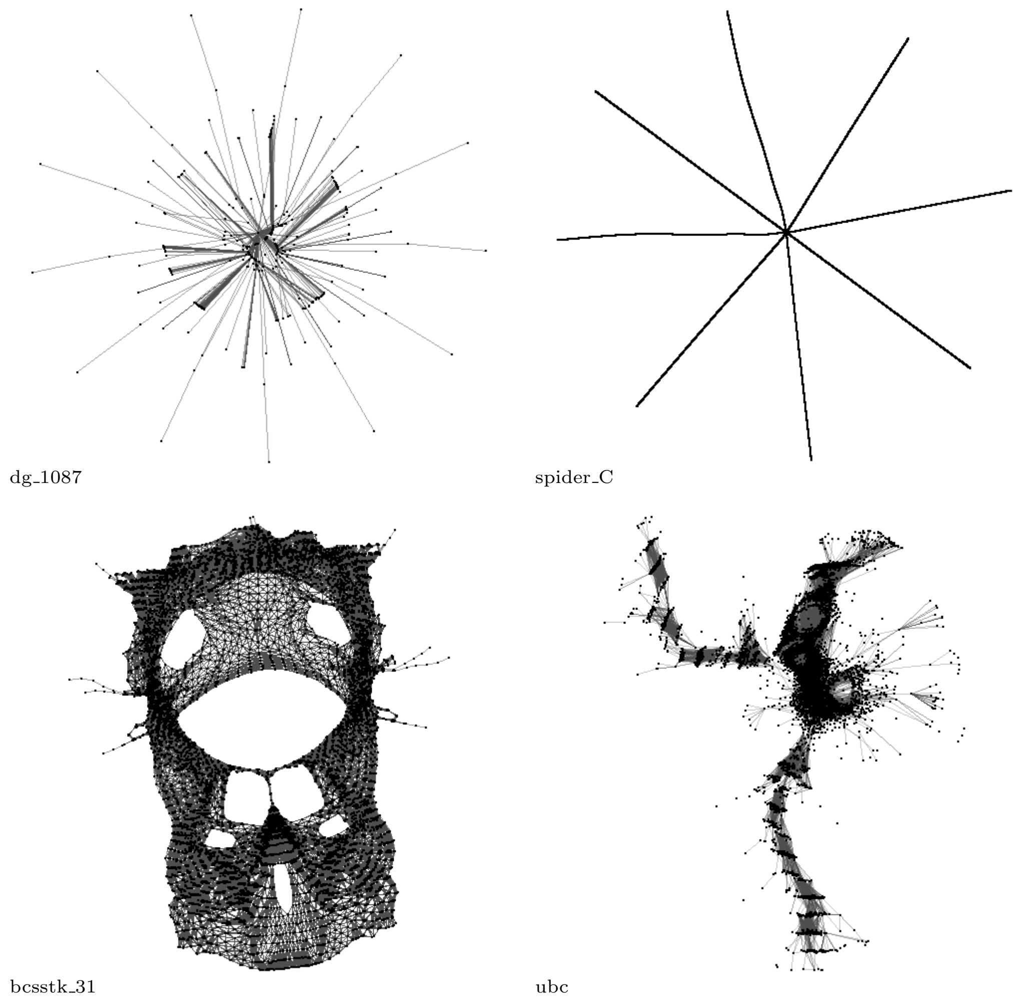


Figure 4: Graph drawings produced by GLUG

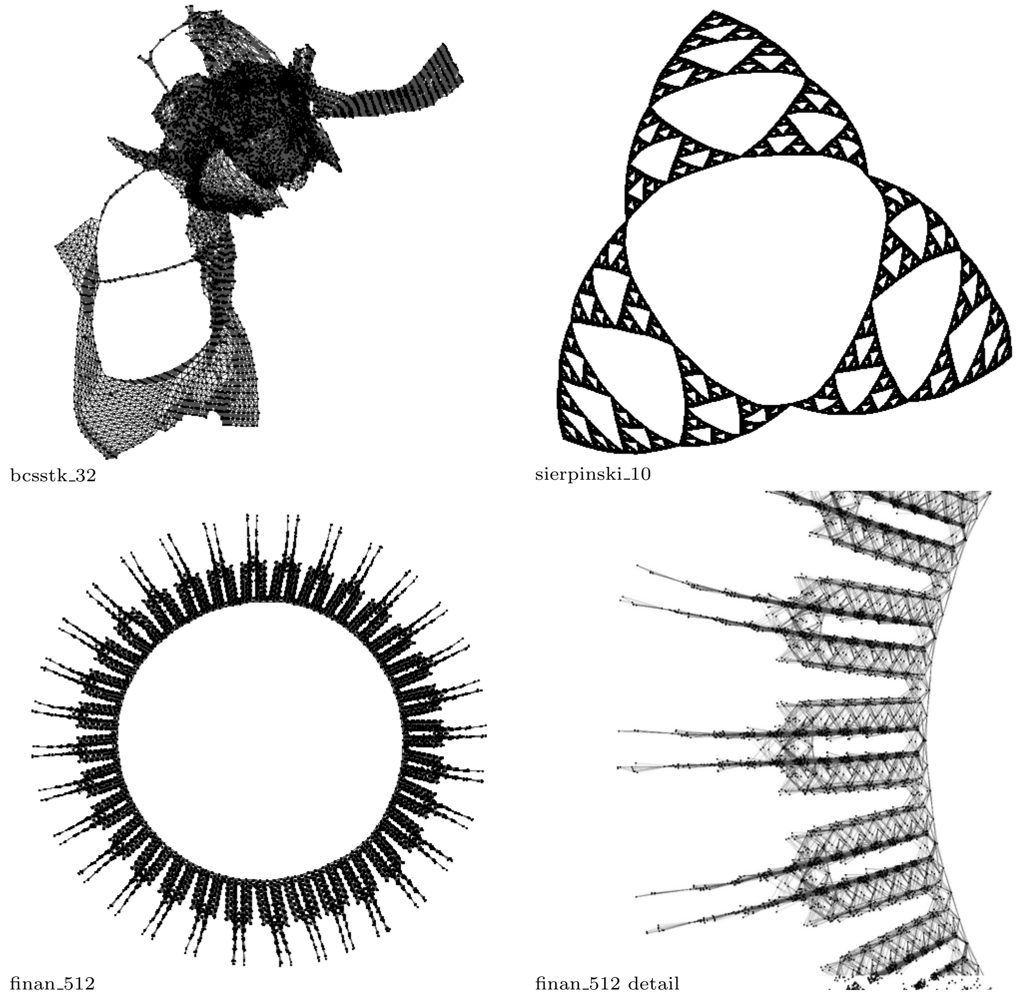
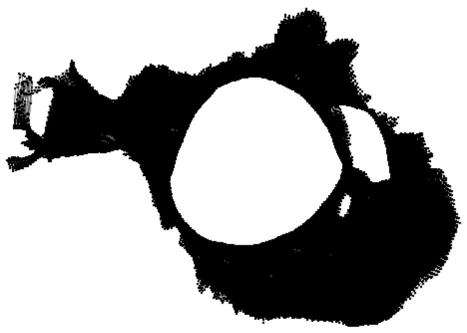
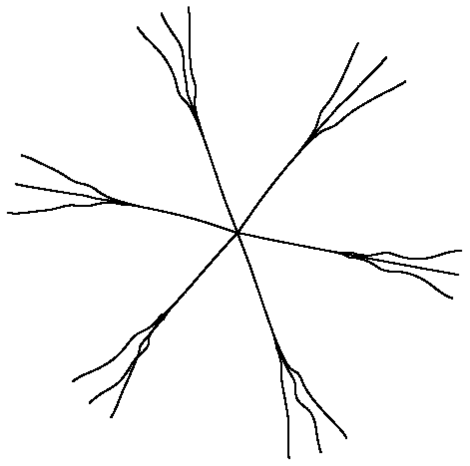


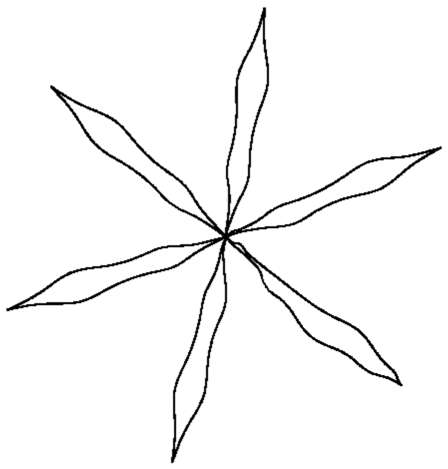
Figure 5: Graph drawings produced by GLUG



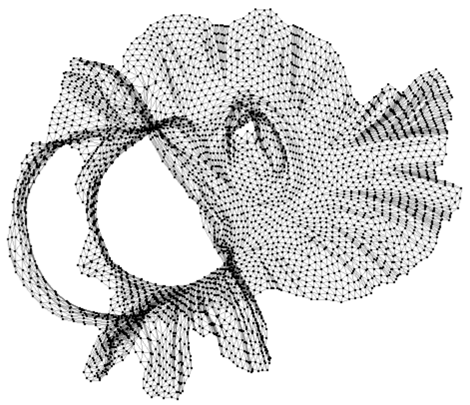
fe_ocean



snowflake_B



flower_C



3elt

Figure 6: Graph drawings produced by GLUG

[17] Archambault, D., Munzner, T., Auber, D.: TopoLayout: Multi-level graph layout by topological features. IEEE Transactions on Visualization and Computer Graphics **13**(2) (March/April 2007) 305–317

[18] Koren, Y.: Graph drawing by subspace optimization. In: VisSym. (2004) pp. 65–74