TclDG - A Tcl Extension for Dynamic Graphs

John Ellson, Lucent
Stephen North, AT&T

TclDG - A Tcl Extension for Dynamic Graphs

John Ellson
john.ellson@lucent.com
Stephen North
stephen.north@att.com

Abstract

TclDG is a toolkit extension for the manipulation, rendering, and interaction with abstract graphs and dynamic graph views. This paper introduces TclDG by examples that progressively build up to an implementation of a multiple-view graph editor.

1 Introduction

Drawings of abstract graphs and networks are an important component of user interfaces that focus on relationships or connections between objects. Effective techniques for viewing static graphs are fairly well understood [RDM+87, Him89, PT90, GKNV93], but many applications could benefit from dynamic graph layouts. Dynamic layouts are needed when the underlying data modeled by graphs can change. Some plausible examples include communication networks whose connectivity changes, and dynamic data structures displayed as graphs in an interactive debugger. Dynamic layouts are also especially relevant if the user can adjust the set of visible objects for browsing graphs or networks too large to be drawn in their entirety, such as linked WWW documents [Dom95].

Besides dynamic displays, another key property of advanced graph browsers is that they be easily customized and extended to handle application-specific features. This suggests a toolkit approach.

TclDG is a practical user interface toolkit that incorporates incremental graph layouts. It was written specifically for prototyping user interfaces to distributed network management systems. TclDG is a direct descendent of TclDot [EN95]. Where TclDot was built on the same graph libraries as the Unix tool "dot," TclDG is built on the next-generation of those graph libraries. It retains much stylistic compatibility with TclDot but its incremental layout mechanisms and its technique for coupling with Tk's canvas are quite different.

Figure 1 shows three interactive views of a sample graph. Users can insert or delete nodes or edges in any view and the resultant graph is updated in all views. Though different layout algorithms are employed, the views are kept in synchrony to depict the same base objects. The 100-line TclDG script that implements this three-view graph editor is discussed in this paper.

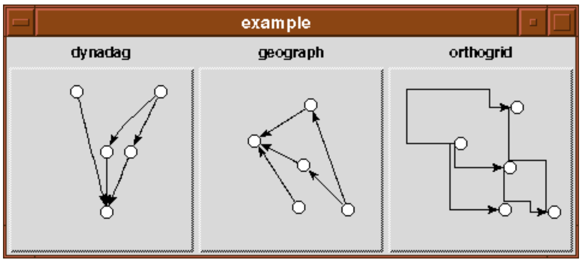

Figure 1

2 Architectural Framework of TclDG

The conceptual framework of TclDG combines graphs, views, and canvasses:

A graph is a set of nodes and edges data structures maintained by Libgraph, but without necessarily any intrinsic positional properties. The sets can be nested into subgraphs, and all items (nodes, edges, graphs, and subgraphs) can have arbitrary application-defined attributes.

A view is also a graph, but one which has been annotated with positional information generated by a layout engine. A view in TclDG does not provide support for application defined attributes since these can be stored in a graph and might be usefully shared between multiple views.

Tk's canvas provides a rendering of a view. Nodes and edges may be further decorated on the canvas. TclDG node and edge handles can be used as tag values with the associated canvas items allowing user events to be related back to the abstract objects represented by the graph. There is no hard dependency on Tk's canvas, so other rendering devices can be used instead. Two useful ones are: GIF-Draw (gd1.2 with tclgd [Bou95, Tho95]) and Pad++ [Con95].

TclDG allows multiple views per graph and multiple canvasses per view. Also, the graph data structure facilities of TclDG can be useful in applications without any views or canvasses for maintaining graph data structures in non-graphical applications.

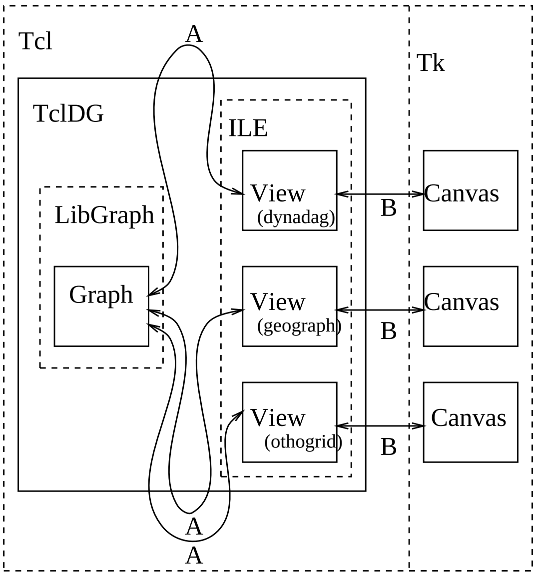

Figure 2

Figure 2 diagrams the architecture of TclDG as instantiated in the example presented in this paper. TclDG provides a Tcl script-level interface to instances of graphs, supported by Libgraph, and to instances of views, supported by the various Incremental Layout Engines (ILEs).

One, possibly novel, aspect of this architecture is the decoupling of the abstract graphs from views. At the level of the C libraries, ILE does not depend on the Libgraph interface, instead ILE is a "black box" interface employing 32 bit identifiers embedded in succinct descriptors for layout elements. The descriptors contain geometric coordinates and some related fields. The namespace of identifiers is controlled by clients; IDs usually refer to client-side descriptors. Engines inform clients of changes to views by invoking callback functions that pass descriptors. Our original design did explicitly layer ILE on Libgraph, but we found it convenient to allow ILE clients to define their own graph data structures.

TclDG provides a "handle" for each node (or edge) in the main graph that can be used in the views, and as a tag on canvas items.

The Tcl script writer has control over all interactions between graphs and views (A), and views and canvasses (B), because interconnection is programmed by Tcl methods and bindings, not hard-wired. In the example script presented in this paper the Tcl processing between graphs, views, and canvasses is minimal, but in a more advanced TclDG application these interfaces can be quite valuable. For example, the underlying graph can be larger (contain more nodes and edges) than any individual view; the scripts at points (A) can filter the graph according to some application defined predicate for inclusion in the view. This can be used to produce say: inheritance diagrams, include file dependencies, and ER diagrams, all from a common graph representing a set of source code.

In the example graph editor application, graphs, views, and edges are "connected" by bindings such that user events propagate from the user, through one view, to the graph and from there distributed back out again to the user via all the views. In this way, all the views are updated to be consistent with the underlying abstract graph, although they may use different layout engines and offer other stylistic differences in rendering.

3 Graphs

In this section we describe the methods and bindings of graphs. The next section will provide a similar introduction to views; then we will combine these elements to present an example graph editor script.

To reiterate, a graph is a set of nodes interconnected by edges. Nodes can be added and deleted from the graph, and edges can be added or deleted between pairs of nodes. String attributes of nodes and edges can be set and modified. Primitives are supplied for editing node and edge sets and defining and changing attributes. A detailed description of each command can be found in the TclDG man page. Here we introduce a selected subset of the methods by means of a snapshot of an interactive session with TclDG.

3.1 Graph, node and edge construction

The interpreter is started from the shell with the `tcldg` command that produces an unmodified tclsh "%" prompt. TclDG may also be dynamically loaded into an unmodified tclsh executable.

```
$ tcldg
%
```

A new empty graph is created with the `dgnew` command that returns a handle to the new graph. Graphs can be directed or undirected, strict or not strict. A strict graph is one with no self-edges or parallel edges, a directed graph is one in which an a-b edge is considered different than a b-a edge.

```
% set g [dgnew digraph]
graph0
```

Nodes can be added to the graph with the `addnode` method that returns the handle of the new node.
All handles in TclDG are also registered as new Tcl commands, so handles are typically assigned to variables that are then used with method parameters to perform operations on the object.

```
% set n [$g addnode]
node0
% set m [$g addnode]
node1
% set p [$g addnode]
node2
```

Similarly edges can be added to the graph with `addedge`, but since node handles are also registered as commands edges can alternatively be added by using the slightly shorter `addedge` method of nodes.

```
% set e [$g addedge $n $m]
edge0
% set f [$m addedge $p]
edge1
```

3.2 Graph, Node and Edge Attributes

Nodes and edges can have user defined attributes whose values are strings. Once an attribute name has been declared for one node, then all nodes have an attribute of that name.
There is a similar namespace for edge attributes.
In real applications attributes usually play a crucial role in supporting application-specific graph semantics.

```
% $n set a 3
3
% $n set a
3
% $m set a

%
```

3.3 Introspection and Navigation of a Graph

Methods are provided for several kinds of graph searches and traversals. Here is a small sampling of the methods.
Methods that return handles are convenient for use in compound statements like

```
[$e headof] listoutedges
```

Methods that return lists of handles are convenient for use in `foreach` loops.

```
% $g listnodes
node0 node1 node2
% $g listedges
edge0 edge1
% $m listinedges
edge0
% $m countoutedges
1
% $e tailof
node0
```

3.4 Deletion

Nodes, edges, and graphs all have a `delete` method. If a node is deleted then all subtending edges are also deleted. If a graph is deleted then all the nodes and edges of the graph are deleted along with it.

```
% $f delete
% $n delete
% $g listnodes
node1 node2
% $g listedges
%
```

3.5 Graph Bindings

Bindings in TclDG are analogous to the bindings in Tk, except that these support a different set of events completely independent of X events. Graph bindings are provided to allow views to be notified of events that occur in the graph, such as a node insertion or an attribute modification. All the bindings in TclDG share a common set of "%" substitution parameters. These are:

- `%g` The graph handle of the graph generating the event.
- `%v` The view handle of the view generating the event.
- `%n` The node handle of the node generating the event.
- `%e` The edge handle of the edge generating the event.
- `%A` The attribute name (or shape type)
- `%a` The attribute value (or boundary point list).

Not all of these are appropriate in all TclDG events - an empty string is supplied for parameters that cannot be substituted.

Here we set up the three node bindings and illustrate the resulting callbacks by generating each of the event types: insertion, modification, and deletion.

```
% $g bind insert_node {puts "insert: %g %n"}
% $g bind modify_node {puts "modify: %g %n %A %a"}
% $g bind delete_node {puts "delete: %g %n"}
% set n [$g addnode]
insert: graph0 node0
node3
% $n set a 9
modify: graph0 node3 a 9
9
% $n delete
delete: graph0 node3
```

Multiple bindings may be attached to the same event. This can connect a graph to multiple views, or a view to multiple canvasses. Additional bindings are indicated by a leading "+" in the binding string.

```
% $g bind insert_node {+puts "insert2: %g %n"}
% $g addnode
insert: graph0 node3
insert2: graph0 node3
node3
```

The full set of bindable events can be found by the introspection mode of the bind method, i.e. without any parameters. The current bindings for any event can also be determined.

```
% $g bind
insert_graph modify_graph delete_graph
insert_node modify_node delete_node
insert_edge modify_edge delete_edge
% $g bind insert_node
puts "insert: %g %n"
puts "insert2: %g %n"
```

Debugging trick: this one-liner binds a debug statement to all the possible events of a graph.

```
% foreach b [$g bind] {$g bind $b
        "+puts \"$b %g %e %n %A %a\""}
```

4 Views

A "view" is a specialized data structure supported by the ILE library. Objects in views (nodes, edges, and higher-order structures) are represented by descriptors that contain an object ID and associated geometric coordinates. The principal operations in ILE are:

- open or close a view
- insert a new object
- modify an object's position or shape
- delete an object
- hold or release callbacks

(The operations to hold and release callbacks are not exposed to the Tcl programmer but used by TclDG to provide atomic operations.) A few other functions assist with mapping between IDs and descriptors and walking the contents of a view.

The set of layout engines is extendible. The selection includes:

DynaDag	- Hierarchical directed graphs, splined edges [Nor96]
OrthoGrid	- Incremental manhattan layout [MHT93]
GeoGraph	- User-defined node placement, straight edges. (For graphs in which nodes are manually placed, or in which nodes have intrinsic location properties that dictate placement)

Figure 3 shows two successive frames in the execution of DynaDag. The frame on the right shows the result of incrementally adjusting the layout on the left, after adding an edge from the rightmost to the middle node on the second level.

4.1 Node and Edge Insertion into Views

When a nodes is inserted into a view, information about its shape (bounding polygon), and an optional initial coordinate (mouse click hint) are provided to the view's engine. The shape descriptor permits computing the correct separation between center points of adjacent nodes and in clipping edge splines or polylines to the node's boundary. (In a simple graph editor [Ous94], one usually draws the edge to the center of the node and obscures the end by drawing the node shape on top. However this technique does not lend itself to drawing arrowheads on directed edges.)

In this fragment we insert node `$n` as an oval, and node `$m` as a triangle. Note that the supported shape names and coordinate syntax is designed to be compatible with Tk's canvas items.

```
% set v [dgview dynadag]
view0
% $v insertnode $n oval {-10 -10 10 10}
% $v insertnode $m polygon {-10 10 0 -10 10 10}
% $v insertedge $e
```

4.2 View Introspection

View introspection methods are helpful in situations in which a predicate for node or edge inclusion in a view depends on the nodes and edges already inserted.

```
% $v listnodes
node0 node1
% $v listedges
edge0
% $v isnodeinview $n
1
% $v isedgeinview $f
0
```

4.3 View Deletion

As in graphs, deletion methods release resources associated with objects in views.

```
% $v deleteedge $e
% $v deletenode $m
% $v delete
```

4.4 View Bindings

View bindings operate similarly to graph bindings, but view operations such as adding an edge may yield multiple callbacks as the layout engine adjusts other nodes and edges around the new one.
In insert and modify bindings `%A` and `%a` substitutions provide shape and coordinates in the format expected by the canvas create operations.
Callbacks for deletion events are issued just prior to deallocation.

```
% $v bind
insert_node insert_edge modify_node
modify_edge delete_node delete_edge
% foreach b [$v bind] {
  $v bind $b "+puts \"$b %v %e %n %A %a\""
}
% $v insertnode $m oval {-10 -10 10 10}
insert_node view0  node0 oval -10 0 10 20
% $v deletenode $n
delete_node view0  node0
```

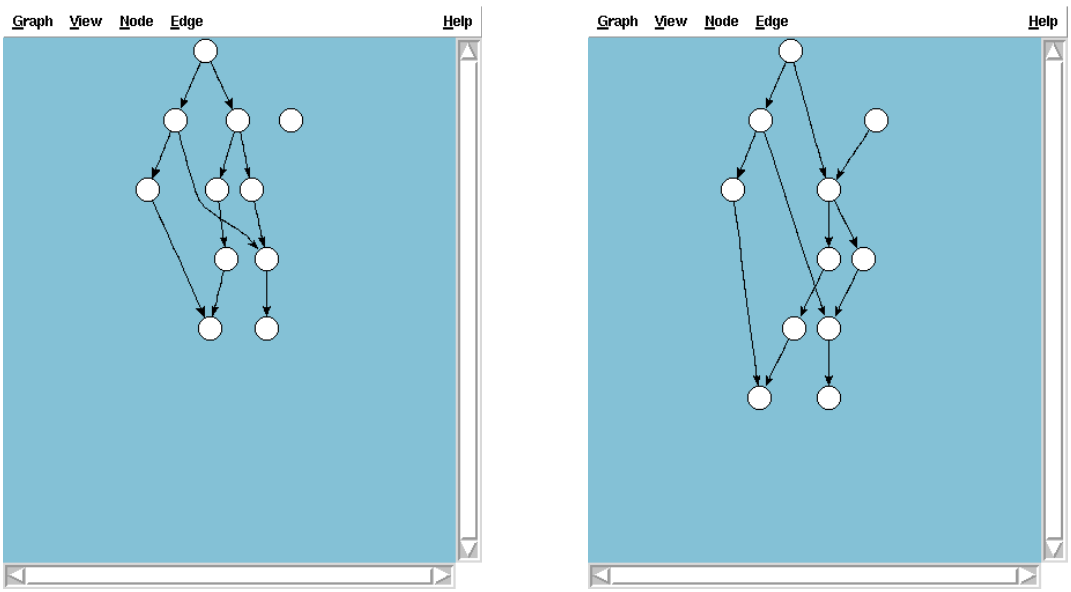

Figure 3

5 Putting it all together

This is the script for the three view graph editor shown in Figure 1. The bulk of the code handles user input events. The most interesting section of the code is the `newview` proc, in which connections are established between views and graphs, and between graphs and canvasses.

```
#!/usr/add-on/tcl7.5/bin/tclsh
load /usr/add-on/tcl7.5/lib/libtcldg.so
load /usr/add-on/tcl7.5/lib/libtk.so

# example multi-view graph editor
#
# mouse button 1 inserts node, or edge if pressed over an existing node.
# mouse button 3 deletes node or edge under mouse
#
# insert or delete in any view, result is displayed in all views.

# set to 1 to trace graph and view events
set debug 0

# define shape for nodes
set shape oval
set boundary [list -5 -5 5 5]

# set global x,y according to canvas-relative mouse position
proc adjust_xy {c wx wy} {
    global x y
    set x [$c canvasx $wx]; set y [$c canvasy $wy]
}

# find a graph object under the mouse of one of the types
#   given in $args, the handle of the object is left in $obj
proc find_obj {c args} {
    global x y
    upvar 1 obj obj
    foreach item [$c find overlapping $x $y $x $y] {
        foreach obj [$c gettags $item] {
            foreach type $args {
                if {[string first $type $obj] == 0} {return 1}
            }
        }
    }
    set obj {}
    return 0
}

# start a node or edge insertion, if started over a node then insert edge
proc b1_down {c} {
    global oldx oldy x y shape boundary id tail
    if {[find_obj $c node]} {
        set id [$c create line $x $y $x $y -fill red]
    } {
        set id [$c create $shape $boundary -fill red]
        $c move $id $x $y
    }
    set oldx $x; set oldy $y
    set tail $obj
}

# move node before completing insertion, or extend edge
proc b1_move {c} {
    global oldx oldy x y id tail
    if {$tail != {}} {
        $c coords $id $oldx $oldy $x $y
    } {
        $c move $id [expr $x-$oldx] [expr $y-$oldy]
        set oldx $x; set oldy $y
```

```
    }
}

# complete node or edge insertion
#    x,y are left in global vars for use by the event bindings
proc b1_up {c g} {
    global id tail
    $c delete $id
    if {$tail != {}} {
        if {[find_obj $c node]} {$tail addedge $obj}
    } {
        $g addnode
    }
}

# delete a node or edge
proc b3_down {c} {
    if {[find_obj $c node edge]} {$obj delete}
}

# create a new view of a specific type on a given graph
proc newview {c g viewtype} {
    global debug

    canvas $c -bd 2 -relief raised -height 150 -width 150
    $c create text 35 10 -text $viewtype
    set v [dgview $viewtype]
    if {$debug} {foreach b [$v bind] {$v bind $b "+puts \"$b %v %n %e %A %a\""}}

    # connect graph events to the view.
    # use the current position of mouse as the placement hint
    $g bind insert_node "+$c create \$shape \$boundary -tag %n -activefill yellow -fill white
        $v insertnode %n \$shape \$boundary -at \$x \$y
        $c configure -scrollregion \[$c bbox all\]"
    $g bind insert_edge "+$v insertedge %e
        $c configure -scrollregion \[$c bbox all\]"
    $g bind delete_node "+$v deletenode %n"
    $g bind delete_edge "+$v deleteedge %e"

    # connect view events to the canvas
    $v bind insert_node "+$c move %n %a"
    $v bind insert_edge "+$c create %A %a -tag %e -activefill yellow -arrow last"
    $v bind modify_node "+$c move %n %a"
    $v bind modify_edge "+$c coords %e %a"
    $v bind delete_node "+$c delete %n"
    $v bind delete_edge "+$c delete %e"

    # bind mouse events in this canvas to the event procs
    bind $c <1> "+adjust_xy $c %x %y; b1_down $c"
    bind $c <B1-Motion> "+adjust_xy $c %x %y; b1_move $c"
    bind $c <B1-ButtonRelease> "+adjust_xy $c %x %y; b1_up $c $g"
    bind $c <3> "+adjust_xy $c %x %y; b3_down $c"

    # return canvas for packing
    return $c
}

# create a graph
set g [dgnew digraph]
wm minsize . 20 20
if {$debug} {foreach b [$g bind] {$g bind $b "+puts \"$b %g %n %e %A %a\""}}

# create one example of each view type
pack [newview .1 $g dynadag] -side left -expand true -fill both
pack [newview .2 $g geograph] -side left -expand true -fill both
pack [newview .3 $g orthogrid] -side left -expand true -fill both
```

6 Foundation Libraries

The layout algorithms underlying TclDG are implemented in C, and share a common external interface (ILE) and internal engine foundation library. We will briefly sketch the external interface. A layout engine offers the following primitives to clients:

- open and close a view
- insert, modify, and delete items in a view
- issue pending changes to a view
- return contents of views or attributes of items in views

At the time a new view is opened, the client passes a discipline structure that lists client-side callback functions for setting placement of objects in views, as well as the desired minimum separation and precision for placing layout objects, and a flag indicating if callbacks should be issued immediately when objects are moved (otherwise only issues on demand by the client.) The open call returns a handle to a new view that is bound to the given engine and discipline. Thereafter a client may edit the contents of a view, and receive callbacks corresponding to layout updates, and eventually close the view to release its resources.

The engine interface is not tied specifically to TclDG, so engines can be reused in other environments. For example a colleague wrote an OLE-compliant graph editor in Microsoft Visual C++ with this same collection of layout engines.

Layout engines are written on top of a library that supplies a common foundation for maintaining graph models of views, managing callbacks, fitting and clipping edge splines, and primitive geometric operations.

As previously described, TclDG relies on Libgraph for graph representation and the engines we have written also use Libgraph internally. Libgraph has about 60 interface functions. The main operations are

- open and close graphs and subgraphs
- read and write graph files
- search or traverse node or edge sets
- edit node or edge sets
- define and update string attributes
- bind runtime storage records to graphs, node, edges

Beyond basic node and edge set maintenance, Libgraph offers richer, more flexible semantic for more convenient programming. For example Libgraph has client-defined "disciplines" to customize memory allocation, I/O, object ID allocation, and callback management. This flexibility is particularly useful in TclDG. It supports general (not just node-induced) nested subgraphs, and a general purpose graph file language. Internally, Libgraph uses Phong Vo's Libdict[1] to store node and edge sets. This provides efficient amortized log-time random access set operations using splay trees. To simplify efficient implementation of graph algorithms, certain key data access operations can be performed by in-line code. For example, node and edge sets in "read-only" mode may also be linearized or flattened into lists for traversal by in-line pointer-chasing code, and there is a similar way of binding run-time storage records for in-line access. For further background information, see the 1993 USENIX paper on earlier editions of Libdict and Libgraph [NV93].

Regarding layout algorithm efficiency, the engines we have written are advanced engineering test-beds, so we have not spent much effort on efficiency tuning. Our view is that the main problem at this stage is simply to understand what animation sequences of dynamic graphs look best, before trying to optimize their computation. DynaDag is usable on graphs up to several dozen nodes depending somewhat on the specific layout. An update always takes at linear time in the viewed graph size (because all object coordinates are recomputed); in the worst case adjusting a bad layout (such as one having many long edges or many edge crossings) could incur quadratic cost. Orthogrid employs a shortest path calculation on a set of tiles that is also potentially quadratic in graph size; though some performance enhancements borrowed from the DEC Contour tile-based VLSI router seem highly relevant [DM95].

7 Summary and Future Work

To recapitulate some of the virtues of TclDG's design, as compared with conventional graph editor toolkits:

- Scriptable and customizable in Tcl.
- Extendible via other Tcl/Tk packages.

[1] Recently extended to incorporate stacks and queues, and renamed LibCDT for C Data Types.

- Extendible via other programs using Libgraph's file language.
- Employs portable, efficient dynamic graph layout libraries written in C.

One of the main areas we have not addressed is efficient incremental layout when graph edits are compound, and perhaps large. An important example is batch loading of graphs from "save files" or externally-created diagrams. For acceptable performance, layout engines must load entire graphs more efficiently than by merely inserting individual nodes or edges and updating the entire diagram each time. They should also accept pre-existing coordinates as a strong hint about future placement. Our existing engine interface may be able to accomodate compound operations, such as batch loading, through the "immediate callback" flag. Compatible engines will need to record edits in a buffer, only process the edits and recompute layouts on demand. This implies some reworking of the engine foundation library.

Another idea is that the current framework of linked views in separate canvasses suggests literally embedding (nesting) diagrams inside other diagrams. In other words, this means implementing client requests to insert one view in another. Representing a nested diagram as a node (or some collection of proxy nodes and edges) with certain dynamic properties seems to be a natural approach, but new complications arise. For example, which engine is responsible for drawing edges that span nesting levels, and how does that engine obtain the edge's bounding path? Also, containers and containees potentially affect each other's layouts, so how should events be propagated between nested diagrams? Do nested layouts have fixpoints, or if not, how should convergence be enforced?

There is a good opportunity to improve the graphical presentation of dynamic layouts. Though the layout engines move objects atomically, smooth animation or fade-in/fade-out techniques are more understandable than simply switching frames instantaneously. On the other hand these graphical techniques also slow down dynamic graph displays so they may not be as appropriate for high volumes of graph data.

There is much interest in remote front ends written in Java [Ous95]. We are designing a prototype Java interface for graph diagrams. With this we hope to build distributed, multi-user graph applications in which a server maintains an abstract graph and the clients present various views to the users and send user events back to the server.

We are writing a spring embedder layout engine for undirected graphs. Previous spring embedder layout programs have lacked efficient methods of resolving node-node, node-edge, and edge ordering constraints. Such constraints are vital to making layouts that are as readable as good hand-made diagrams. We are addressing these areas.

8 Release Plans and Tcl Compatibility

Our plan is to release TclDG within the next few months under the same license arrangements as graphviz.

TclDG is a loadable extension for tcl7.5 and tk4.1. TclDG is expected to be portable to all platforms on which Tcl runs, including Macs and PCs (to be verified, but all major parts have already been ported.)

TclDG requires DASH [Nij] and SPLINE patches (ours) to tk4.1.

References

[Bou95] Thomas Boutell. gd1.2 - a graphics library for fast gif creation, 1995. http://www.boutell.com/gd.

[Con95] Pad++ Consortium. Pad++ zoomable interface, 1995. http://www.cs.unm.edu/pad++/.

[DM95] Jeremy Dion and Louis M. Monier. Contour: A tile-based gridless router. Technical Report 95/3, Digital Western Research Laboratory, 250 University Ave., Palo Alto CA 94301 USA, March 1995.

[Dom95] Peter Domel. Webmap: A graphical hypertext navigation tool. *Computer Networks and ISBN Systems*, 28(1):85–97, 1995.

[EN95] John Ellson and Stephen North. graphviz/tcldot release, 1995. http://www.research.att.com/orgs/ssr/bookreuse.

[GKNV93] E.R. Gansner, E. Koutsofios, S.C. North, and K.P. Vo. A technique for drawing directed graphs. *IEEE-TSE*, March 1993.

[Him89] M. Himsolt. Graphed: An interactive graph editor. In *Proc. STACS 89*, volume 349 of *Lecture Notes in Computer Science*, pages 532–533, Berlin, 1989. Springer-Verlag. http://www.uni-passau.de/~himsolt/GraphEd/graphed.

[MHT93] Kanth Miriyala, Scot W. Hornick, and Roberto Tamassia. An Incremental Approach to Aesthetic Graph Layout. In *Proc. Sixth International Workshop on Computer-Aided Software Engineering*, pages 297–308. IEEE Computer Society, July 1993.

[Nij] Jan Nijtmans. dash patch. ftp://ftp.nici.kun.nl/pub/tkpvm/tk4.1dash.patch.gz.

[Nor96] Stephen North. Incremental Layout in DynaDAG. In *Proc. Graph Drawing '95*, volume 1027 of *Lecture Notes in Computer Science*, pages 409–418. Springer-Verlag, 1996. ftp://ftp.research.att.com/dist/drawdag/dynadag.ps.gz.

[NV93] Stephen C. North and Kiem-Phong Vo. Dictionary and graph libraries. In *USENIX Winter Conference*, pages 1–11, 1993.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*, chapter Simple Interactive Graph Editor, pages 213–215. Addison-Wesley, 1994.

[Ous95] John Ousterhout. The relationship between tcltk and java, October 1995. http://www.sunlabs.com/research/tcl/java.html.

[PT90] F. Newbery Paulish and W.F. Tichy. Edge: An extendible graph editor. *Software–Practice and Experience*, 20(S1):1/63–S1/88, 1990.

[RDM+87] L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan. A browser for directed graphs. *Software–Practice and Experience*, 17(1):61–76, January 1987.

[Tho95] Spencer W. Thomas. Tcl gd extension, 1995. http://guraldi.hgp.med.umich.edu/gdtcl.html.