;login:

THE MAGAZINE OF USENIX & SAGE

October 2001 • Volume 26 • Number 6

inside:

PROGRAMMING

Writing the Tk Geometry Managers in Tcl

By Sergei Babkin

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

writing the tk geometry managers in tcl

by Sergey Babkin

Sergey Babkin works for Caldera Systems on UnixWare/ OpenUNIX. He is also a FreeBSD developer and participates in a few smaller Open Source projects.

babkin@users.sourceforge.net

If you open a book on Tk programming, the internals of the geometry managers are usually considered a rather advanced topic. And the fact that there are not many geometry managers for Tk correlates well with this observation. A major obstacle in writing a geometry manager is that they are normally written in the C language and then linked with Tcl (or possibly Perl). To overcome this obstacle, I wrote a small module that implements the operations missing in Tcl/Tk but necessary for writing a geometry manager. This made writing the new geometry managers and experimentation with them much easier, so I want to share this experience.

Of course, an important question to consider is: why would someone want to write another geometry manager? Are not the managers provided with Tcl/Tk enough? Well, it depends on what do you want to do with them. For my project "Not A Commander" (see its home page at *http://nac.sourceforge.net*) they were not. I started this project in order to learn Tcl/Tk while doing something useful. And I had a quite good idea of what this useful thing would be: an X11 file manager done the way I like it.

I like to have the modal dialog windows shown within the main window of the application. When they are created as new top-level windows, one of two things usually happens: either a Netscape dialog pops up when I'm typing something in xterm, and my typing gets diverted to this dialog which immediately disappears; or the dialog gets lost among the other windows so that I am surprised when the main Netscape window refuses to react to any typing and mouse-clicking. Both cases annoy me greatly.

In a program I write for my pleasure, such dialogs should be shown in the main window, preferably as big as their natural size but not bigger than the size of the main window. Achieving this effect with the standard geometry managers is possible but far from easy, and even at its best does not work very well. For this reason, I decided to write my own geometry managers. And since I wanted to experiment with them easily, they had to be written in Tcl.

General Principles

The general principles of the geometry managers are described in the classic book *Tcl and the Tk Toolkit*, by John Ousterhout. The only caveat regarding this book is that it describes a quite old version of Tcl/Tk, so the details of the geometry managers' implementation have changed significantly. But the general principles are the same.

In short, they work as follows: as the slave widgets (or windows – for the purposes of this discussion these terms are synonyms) are configured, they calculate the size they need. They send these size requests to the geometry manager. The geometry manager collects these requests and places the slave widgets inside the master widget according to its policies and to the size of the master widget. If the size of the master widget is not enough to satisfy the requests of all the slaves, the geometry manager will usually resize some of the slaves to a smaller size than they requested.

The geometry manager also calculates the needed size of the master window that would fully satisfy the requests of its slaves and passes this request further up the hierarchy, to the master's master. Also, if the master widget gets resized, the geometry man-

ager must recalculate the placement of the slave widgets in it accordingly. Because usually many slaves are created or changed at once, the geometry managers try to avoid the unnecessary work of doing the full recalculation on each and every change. They postpone the actual recalculation until the script becomes idle and then sweep up all the changes at once.

Writing the Geometry Manager

To write the geometry managers in Tcl I needed to implement in C the special operations not normally available to the Tcl scripts:

- Notify Tk that a slave widget will be managed by this manager.
- Notify Tk that a slave widget will no longer be managed by this manager.
- Map a slave widget within a master widget.
- Unmap a slave widget from a master widget.
- React to a geometry request from a slave.
- Send a geometry request further up.
- React to a reassignment of a slave to another geometry manager.
- Get the border width of the master window.

The last operation is theoretically not really necessary because the border width information usually can be obtained by running the `cget -bd` command of a widget, but in practice getting this information directly is safer and faster.

Two more operations are needed to connect the Tcl part of the geometry manager with the C services:

- Register a geometry manager with the C services.
- Unregister a geometry manager with the C services.

For reasons of space, I won't include the full listing of the C part; it can be downloaded (the file `geom.c`) as part of the Not A Commander (NAC) project. I will only describe the commands that are visible to the Tcl side and how they map to the Tk's C calls.

Since the code was written specifically for the NAC project, it doesn't try to be a proper module with, for example, namespace isolation. For the purposes of study, the simplified code seems to me more of an advantage than a drawback. Also it uses the NAC object model and conventions. For the examples in this article, I've gotten rid of most of these dependencies and dragged in only a minimal amount of them. The most important one is that the code is generally organized into "classes." The "classes" are not exactly what is meant by this word in the world of object-oriented programming but are a reasonable approximation. The procedures and global variables of a class are prefixed with the class name followed by a colon:

```
<class_name>:<object_name>
```

The object names that start with underscore are intended for the use of procedures of this class only (sort of like "private" and "protected" in C++ but without enforcement). More of these conventions will be described when we get to the Tcl code.

The commands implemented in C are:

Register a Tcl geometry manager:

```
nacgeom:register <manager_prefix>
```

`manager_prefix` is the class name of this manager. The implementation has an array of Tk per-manager structures, in which it finds a free slot and remembers the name.

Then it returns a manager ID which may be used for the subsequent calls. The most important part of the implementation is:

```
managers[freeid].name = strdup(argv[1]);
snprintf(interp->result, TCL_RESULT_SIZE, "%d", freeid);
```

UNREGISTER A TCL GEOMETRY MANAGER:

```
nacgeom:unregister <manager_prefix>
```

The implementation marks the structure in the registration array as free.

NOTIFY TK THAT THIS MANAGER WILL TAKE CARE OF THE SLAVES:

```
nacgeom:ofslave <mgr_id> <slave_window>...
```

mgr_id is the ID returned by nacgeom:register. More than one window can be specified. The most important call repeated for each slave window is:

```
Tk_ManageGeometry(win, &managers[mgr_id], (ClientData) mgr_id);
```

NOTIFY TK THAT THIS MANAGER RELEASES THE SLAVES:

```
nacgeom:freeslave <mgr_id> <slave window>...
```

The most important call in the implementation (repeated for each slave) is:

```
Tk_ManageGeometry(win, NULL, (ClientData) 0);
```

MAP A SLAVE WINDOW WITHIN A MASTER WINDOW:

```
nacgeom:map <slave_window> <master_window> <x> <y> <width>
<height>
```

x, y, width, and height are the position and size of the slave window in the master window. Tk provides a convenient function that takes care of all the necessary details:

```
Tk_MaintainGeometry(slave, master, x, y, width, height);
```

The Tk standard geometry managers separate a special case when the slave window is an immediate child of the master window; in this case they do all the mapping, positioning, and resizing by calling the low-level functions directly to improve the performance. However, the geometry managers written in Tcl are slow enough by themselves, so a little more overhead traded for convenience won't hurt them noticeably.

UNMAP A SLAVE WINDOW:

```
nacgeom:unmap <slave_window> <master_window>
```

This is implemented as another convenience call:

```
Tk_UnmaintainGeometry(slave, master);
```

REACT TO A GEOMETRY REQUEST TO A SLAVE.

This function passes the control in the opposite direction, it is called from Tk and calls a Tcl callback procedure that should be defined in the Tcl code:

```
<manager_prefix>:_geometry <slave_window> <width> <height> <border>
```

Width, height, and border width are the requested dimensions of the slave window. This function gets two arguments: the manager ID (as passed to Tk in Tk_ManageGeometry) and the slave window ID. Its important part is:

```
snprintf(bf, sizeof bf, "%s:_geometry %s %d %d %d", managers[id].name,
    Tk_PathName(win), Tk_ReqWidth(win), Tk_ReqHeight(win),
    Tk_InternalBorderWidth(win) );
Tcl_GlobalEval(my_interp, bf);
```

Send a geometry request up the hierarchy:

```
nacgeom:request <master_window> <width> <height>
```

This translates to the call

```
Tk_GeometryRequest(master, width, height);
```

React to a loss of slave due to its reassignment to another geometry manager.
Transfers the call to another Tcl callback function:

```
<manager_prefix>:_lost_slave <slave_window>
```

The implementation is similar to another callback:

```
snprintf(bf, sizeof bf, "%s:_lost_slave %s", managers[id].name,
    Tk_PathName(win));
Tcl_GlobalEval(my_interp, bf);
```

Get the border width of a window:

```
nacgeom:infobd <window>
```

This is implemented as

```
snprintf(interp->result, TCL_RESULT_SIZE, "%d",
    Tk_InternalBorderWidth(win));
```

The full text of the C support (geom.c) and the Makefile are parts of Not A Commander, and can be downloaded from *http://nac.sourceforge.net*.

An Example

Now let's look at an example of a full geometry manager that uses this interface: a simplified version of the post geometry manager from NAC. It allows posting of the slave widgets at the center of the master widget (if multiple slaves are posted, then they will overlap each other). The size of the slave widgets is limited only by the size of the master window.

The full text of the example is available for download from:
http://nac.sourceforge.net/pub/post.tcl.

The procedures in the example do not follow the Tk convention of one command with many subcommands but instead follow the usual NAC naming conventions. The meaning of commands implemented in this manager is similar to the standard Tk geometry managers but simplified:

```
post:add <slave-widget>... [-in <master-widget>] – Manage the slave widgets
post:forget <slave-widget>... – Stop managing the slave widgets
post:slaves <master-widget> – Return the list of slaves posted in this master
```

To save space, the less essential parts are not shown here and are only briefly described. The script starts with loading the C part:

```
load ../geom.so nacgeom
```

The script expects that it would be placed in a subdirectory one level under the base directory of NAC.

Then three auxiliary procedures are defined. These procedures can be obtained by including the files gman.tcl and util.tcl from NAC but are defined in the script explicitly to avoid extra dependencies.

nacgeom:assert_ancestor <slave> <master> checks that the master and slave widgets conform to the proper ancestral relations, or throws an error otherwise.
bind_adduniqtag <window> <position> <tag> adds the binding tag to the binding list of the window at the specified position unless it's already on the list.
bind_rmclass <window> <tag> removes the tag from the binding list of the window (opposite of bind_adduniqtag).

The first action of the geometry manager itself is its registration with the C support code:

```
set post:gmid [nacgeom:register post]
```

The information about the widgets is stored in the global associative arrays indexed by the widget names. The value at the empty string ("") index is used to set the default values for the newly associated windows. For a master widget two variables are defined:

```
set post:slaves("") {}
set post:calcid("") {}
```

post:slaves contains the list of slaves posted in this master. post:calcid contains the delayed command ID of the scheduled geometry recalculation procedure. As I said before, when a Tk geometry manager gets a new slave or a geometry change request from a slave, it does not recalculate its geometry immediately because there is a good chance that more changes will follow immediately. Instead, it schedules its recalculation procedure for the time when the Tk process becomes otherwise idle.

For a slave widget the variables

```
set post:mymaster("") {}
set post:reqwidth("") 1
set post:reqheight("") 1
```

contain the name of its master and the size that it requested. This size can also be obtained by the Tk commands winfo reqwidth and winfo reqheight, but storing it in variables is more convenient. The procedure

```
proc post:_globals {} { ...
```

imports all the class global variables (those with names starting with post:) into the current function. When imported, these variables lose the class prefix in their names; for example, post:slaves simply becomes slaves. Normally, in NAC such a procedure (and a bit more) for a class would be generated automatically by calling

```
defclass post
```

But again to reduce dependencies in the example it's defined explicitly.

The binding tags are defined for the master and slave widgets:

```
bind post.master: <Destroy> {post:_forgetmaster %W}
bind post.master: <Configure> {post:_schedcalc %W}
bind post.slave: <Destroy> {post:forget %W}
```

When a master widget is destroyed, all of its slaves are freed. When a slave widget is destroyed, it's just forgotten as usual. When the master widget is resized by its own

master, a geometry recalculation must be done for it. As always, this recalculation is not done immediately but scheduled for later.

The procedure that adds the slaves to a master is one of the two larger ones (another large procedure is for the geometry recalculation). Its arguments are like the Tk command place but with only one option supported.

```
proc post:add {args} {
    post:_globals
    set optpos [lsearch -regexp $args {^[^.].*}]
```

It starts with importing the class globals and finding where the options start in the argument list. All the widget names start with a dot, so anything not starting with a dot is considered an option.

```
    if {$optpos >= 0} {
        set opts [lrange $args $optpos end]
        set args [lreplace $args $optpos end]
    } else {
        set opts {}
    }
```

If any options are found, they are separated from the list of the new slave widgets. Since the packing order for this widget manager does not matter, the slaves are always added to the end of the list. And the only supported option is -in to select the master:

```
    set omaster {}
    foreach {opt val} $opts {
        switch — $opt {
        {-in} {set omaster $val}
        default { error "unknown post option $opt"}
        }
    }
    if {$args == ""} {
        # nothing to do
        return
    }
```

Having parsed the options, we add the slaves one by one:

```
    foreach win $args {
        if {$omaster == ""} {
            set master [winfo parent $win]
            if {$master == ""} {
                error "can't manage the root window as a slave"
            }
        } else {
            nacgeom:assert_ancestor $win $omaster
            set master $omaster
        }
```

If the master was specified explicitly, we need to assert that it's appropriate for this particular slave.

```
        if [info exists mymaster($win)] {
            post:forget $win
        }
```

If this slave is already posted, we need to forget it first. If the slave was previously managed by another geometry manager, Tk will notify that geometry manager automatically when we take over its slave. But if the slave was already managed by the same manager, then Tk will not send this notification to us, so we have to check for it.

```
        if { ![info exists slaves($master)] } {
            post:_initmaster $master
        }
        lappend slaves($master) $win
```

If this master has no slaves managed by this manager yet, we need to initialize our global variables and bindings for it. Then we add the new slave to its list.

```
        set mymaster($win) $master
        set reqwidth($win) [winfo reqwidth $win]
        set reqheight($win) [winfo reqheight $win]
        bind_adduniqtag $win end post.slave:
```

Then we initialize our global variables for the slave and add the post.slave binding to the end of its binding list.

```
        nacgeom:ofslave $gmid $win
```

We let Tk know that we take over the management of this slave.

```
        post:_schedcalc $master
    }
}
```

Finally, for each posted slave we schedule the geometry recalculation procedure for its master. And that completes the adding of slaves.

The procedure for the opposite action, forgetting the slaves, is:

```
proc post:forget {args} {
    post:_globals
    foreach win $args {
        if {![info exists mymaster($win)]} {
            continue
        }
        nacgeom:unmap $win $mymaster($win)
        nacgeom:freeslave $gmid $win
        post:_lost_slave $win
    }
}
```

It does not take any options. Each slave is checked whether it's managed. The managed slaves are unmapped and Tk is notified that we don't manage them any more. Finally, we clean up our variables, and this cleanup happens to be the same as when Tk notifies us that the slave was moved by the user to another geometry manager, so we just call that procedure.

The last procedure of the user API returns the list of slaves for a master:

```
proc post:slaves {master} {
    post:_globals
    return slaves($master)
}
```

If no slaves are managed by this geometry mamager for this master it throws an error on an undefined variable.

The scheduling and unscheduling of the geometry recalculation is done with the following procedures:

```
proc post:_schedcalc {master} {
    post:_globals
    if {![info exists calcid($master)] || $calcid($master) == ""} {
        set calcid($master) [after idle "post:_recalc $master"]
    }
}
proc post:_unschedcalc {master} {
    post:_globals
    if {$calcid($master) != ""} {
        after cancel $calcid($master)
        set calcid($master) {}
    }
}
```

post:_schedcalc schedules the run only if it was not already scheduled, because one recalculation run is enough to process all the changes.

The geometry recalculation routine is the heart of a geometry manager:

```
proc post:_recalc {master} {
    post:_globals
    if {$slaves($master) == ""} {
        post:_forgetmaster $master
        return
    }
```

First we check that there are some slaves to manage. If no slaves are left, then this master does not need any more geometry management from us.

```
    set calcid($master) {}
```

Since the delayed command already has been called, its ID is not usable anymore, so we clean it. This is also a sign to post:_schedcalc that when called it must schedule a new delayed execution of the geometry recalculation.

```
    set bd [nacgeom:infobd $master]
    set maxwd [expr [winfo width $master] - $bd *2]
    set maxht [expr [winfo height $master] - $bd *2]
```

The geometry manager should respect the internal border of the master window and not use it for placing the slaves. For this simple manager this just means that the border width must be deducted from the available size.

For geometry managers that do not change the size of the master, such as this post or the standard place, we can now start mapping the slaves. However, the geometry managers that propagate the geometry requests up the widget hierarchy should first calculate the size of the master necessary to accommodate all its slaves as requested and pass this request up. In a hypothetical case of a geometry manager that tries to make the master widget as big in each dimension as the largest size requested by a slave, this code might be:

```
set reqwd 1
set reqht 1
foreach win $slaves($master) {
    if {$reqwidth($win) > $reqwd} {
        set reqwd $reqwidth($win)
    }
    if {$reqheight($win) > $reqht} {
        set reqht $reqheight($win)
    }
}
if {[winfo reqwidth $master] != $reqwd
|| [winfo reqheight $master] != $reqht} {
    nacgeom:request $master $reqwd $reqht
    post:_schedcalc $master
    return
}
```

The smallest valid size for a widget is 1. So the calculation starts with this value. If some slave has requested a larger size, we take this larger size. After processing all the slaves we check whether our new calculated size is different from the size we calculated last time (and passed further up). If it's the same, we can start mapping the slaves. If it has changed, the request for the new size is passed to Tk, which passes it to the geometry manager that has our master as a slave. That geometry manager will schedule its own geometry recalculation for later. There is a good chance that by results of this recalculation it will change the size allocated to our master according to our request.

So for now, mapping the slaves based on the old size of the master widget would be a waste of time, and the best thing we can do is to schedule our own recalculation for later and return. If all goes well, the upper geometry manager's scheduled recalculation will run first (because presumably it was scheduled first) and set the new size to our master. Then our recalculation will run again and map the slaves according to the new size. However, if our master's master wants to resize itself as well, then our rescheduled procedure would run before the resizing happens at the upper level. But it's not a big problem: the only loss is time spent on an extra run of recalculation. Then when the upper geometry manager finally resizes our master widget, it will cause a configure event in this widget which we have bound to the recalculation request, so eventually our recalculation will run again and redo everything based on the new available size.

Now let's return from that hypothetical case to the post geometry manager. We map each slave in its turn. First we calculate its dimensions (wd and ht) and position (atx and aty) and then we actually map it:

```
foreach win $slaves($master) {
    if {$maxwd <= 0 || $maxht <=0} {
        nacgeom:unmap $win $master
        continue
```

Zero or negative maximal dimensions of the slave may occur if the size of the master widget is less than its border width. Since there is no way to display the slave, we unmap it.

```
    } else {
        set wd $reqwidth($win)
        if {$wd > $maxwd} {
```

```
            set wd $maxwd
        }
        set ht $reqheight($win)
        if {$ht > $maxht} {
            set ht $maxht
        }
        set atx [expr ($maxwd-$wd)/2]
        set aty [expr ($maxht-$ht)/2]
    }
```

The size of the slave in each dimension is limited by the available space. Then the slave's position is centered.

```
        incr atx $bd; incr aty $bd
        nacgeom:map $win $master $atx $aty $wd $ht
    }
}
```

Finally, the position is adjusted for the border width, and the slave is mapped at its calculated position.

When the first slave is added to the master, the following procedure is called to initialize the master's data structures:

```
proc post:_initmaster {master} {
    post:_globals
    set slaves($master) {}
    set calcid($master) {}
    bind_adduniqtag $master end post.master:
}
```

It also adds a bind tag which allows us to react to the master widget's destruction or resizing by the master's master.

When the master widget is destroyed or loses its last slave, its data should be cleaned up. All this cleanup activity is done in the next procedure:

```
proc post:_forgetmaster {master} {
    post:_globals
    if [info exists slaves($master)] {
```

If there is no data for this master then there is nothing to clean up. This check also gives some additional safety against double calling of this function due to some race condition.

```
        eval "post:forget $slaves($master)"
```

Any slaves that are left over should be forgotten. If the slaves list is empty, post:forget will just do nothing.

```
        post:_unschedcalc $master
```

If the recalculation was scheduled, it must be canceled. Otherwise when it runs later it would find no data entries and throw an error. This cancellation can be done only after forgetting the slaves because when a slave is forgotten, the master's geometry recalculation gets scheduled.

```
        bind_rmclass $master post.master:
        unset slaves($master)
        unset calcid($master)
    }
}
```

Finally, we remove the binding tag and free the per-master data entries.

When we stop managing a slave, a cleanup should be done as well. This is handled by the procedure post:_lost_slave, which is called in the following cases: a slave is forgotten on a user's call to post:forget; a slave is destroyed and post:forget is called through the binding of the destroy event; a slave is passed to another geometry manager by the user and this procedure is called as a callback from the supporting C code.

```
proc post:_lost_slave {win} {
    post:_globals
    if {![info exists mymaster($win)]} {
        return
    }
```

As with the masters, we'd rather be safe than sorry and not try to free data that is not allocated.

```
    set master $mymaster($win)
    set idx [lsearch -exact $slaves($master) $win]
    if {$idx >= 0} {
        set slaves($master) [lreplace $slaves($master) $idx $idx]
        post:_schedcalc $master
    }
```

This slave is removed from its master's list, and the master's geometry recalculation is scheduled. The recalculation is not absolutely necessary for this particular geometry manager because the slaves are posted independently of each other. But in a generic case, forgetting a slave may cause serious changes in the master's geometry. Even for this manager, however, doing a recalculation is a good thing; if this were the last slave of this master, the geometry recalculation will catch it and free the master's data structures as well. Otherwise we would have to check here for this case explicitly.

```
    bind_rmclass $win post.slave:
    unset mymaster($win)
    unset reqwidth($win)
    unset reqheight($win)
}
```

Finally, we remove the binding tag and free the per-slave data entries.

When a slave sends a new geometry request, the C portion of the code forwards the call to the callback procedure:

```
proc post:_geometry {win wd ht bd} {
    post:_globals
    if [info exists mymaster($win)] {
        set reqwidth($win) $wd
        set reqheight($win) $ht
        post:_schedcalc $mymaster($win)
    }
}
```

If the slave is associated with a master, we remember the values it requested and schedule the geometry recalculation. Otherwise we consider this a spurious call and do nothing.

This completes the simplified post geometry manager. The post geometry manager in NAC has many more features, such as margins around dialog windows and completely different logic for the posting of menus.

Customized geometry managers open many other interesting possibilities. Some of the more complex examples that may be found in Not A Commander include:

- An auto-wrapping label widget (see the classes awlabel and awlpack in wdgt.tcl). If the label can not get enough space along the X axis, it wraps the text at the available width and tries to extend itself vertically. This is achieved by composing the widget from the Tk label subwidgets and controlling them with a highly specialized geometry manager. Of course, this effect may be achieved much more efficiently by modifying the implementation of the Tk label widget, but the internals of that widget are far from simple and are difficult to modify.
- A scrollbar displayed automatically when there is not enough space for all the slaves (see the class menupack in gman.tcl). Note that this is different from the Perl/Tk widget "Scrolled" in which the scrollbars are displayed all the time. The customized geometry managers allow the scrollbars to be displayed only when they are really necessary – that is, when there is not enough space for all the slaves.
- A pseudo-grid (see the class pgrid in gman.tcl). It implements a two-level composition: the whole grid consists of a set of row widgets, each of the rows containing a few slave widgets. The rows are combined by some other geometry manager (such as Tk's pack). The pseudo-grid manager controls the slaves within the rows so that the columns within each row are aligned with each other row. The pseudo-grid is extremely convenient for the vertical menus: each row is a composite menu button widget while the slaves are the items within these buttons. These items are arranged into non-overlapping columns: the optional radio/checkbutton indicator, the button label, and the accelerator key label.

Of course, the downside of implementing the geometry managers in Tk is that they are quite slow compared to those implemented in C. Because of this they do not scale well to a large number of managed widgets and work best either for special cases involving a small number of widgets or for prototyping with a following rewrite in C.