The gWidgets2 package

John Verzani

May 16, 2012

The **gWidgets2** provides a programming interface for writing graphical user interfaces within R. The package depends on one of several different interfaces to underlying toolkit libraries: tcltk, RGtk2, or Qt. [1] The package provides many of the common features of the above toolkits using more or less standard R idioms, making it fairly easy for the R user to quickly begin building user interfaces.

The **gWidgets2** package is a rewrite of the **gWidgets** package. [2] Most changes are not visible to the end user, but there were significant enough ones that the package name was changed. The goals of the rewrite included:

- simplifying of the code base using the powerful combination of **devtools** and GitHub (`github.com/jverzani`) making it much easier to document, test, and deploy changes of the underlying packages;
- replacing S4 methods with reference class methods resulting in a cleaner code base;
- better consistency across toolkits;
- speed improvements; and
- better documentation.

For the most part this has been achieved. Most **gWidgets** scripts run unchanged, save for, perhaps, a few minor adjustments. Some others will be problematic. (To make more cross-toolkit, some of the drag and drop features of **gWidgetsRGtk2** have been dropped.)

[1] A related – but independent – package, **gWidgetsWWW2**, provides an interface for programming web GUIs.

[2] The **gWidgets** package is document in an 2007 *R News* article "An Introduction to gWidgets" and the book *Programming Graphical User Interfaces in R*, by Lawrence and Verzani, CRC Press.

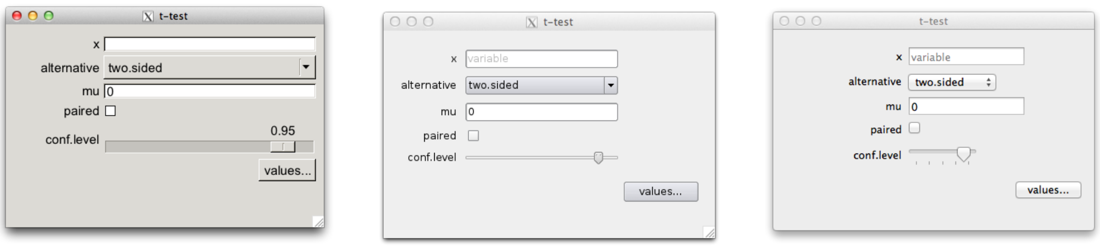

Figure 1: Screenshots of the same **gWidgets2** GUI using RGtk2, tcltk, and qtbase.

As with the original package, **gWidgets2** provides the abstract interface. A toolkit package (one of **gWidgets2RGtk2**, **gWidgets2tcltk**, **gWidgets2Qt**) provides the link to the underlying R toolkit that binds to the graphical toolkit. Typically, the only issues with installation arise from the underlying toolkits, which often require separate downloads. [3] The **devtools** package can be used to install the development versions from GitHub.

1 Requisite "Hello World" example

We begin with a simple example to illustrate what a script looks like:

```
library(gWidgets2)
options(guiToolkit="RGtk2")
##
win <- gwindow("Basic example", visible=FALSE)
gp <- gvbox(container=win)
btn <- gbutton("click me for a message", container=gp)
##
addHandlerClicked(btn, handler=function(h,...) {
  galert("Hello world!", parent = win)
})
##
visible(win) <- TRUE
```

The above will produce a titled window with a button. When the user clicks the button, a dialog will appear with a very important message.

[3] Well, not always. The **tcltk** is a base package and the underlying libraries are usually available, the **RGtk2** and **Qt** packages make an effort to install the underlying libraries.

The first two lines load the package and force the choice of toolkit. If the second line is not given, the user is prompted for a toolkit choice, as needed.

The structure of programming in **gWidgets2** is mostly illustrated above. The basic components – containers, controls, and dialogs – are defined by constructors. In the above we see the window constructor (`gwindow`), a vertical box container constructor (`gvbox`), a button widget constructor (`gbutton`) and a dialog constructor (`galert`).

The top level window constructor has two arguments specified, one for the title, the other to suppress the initial drawing. The latter allows the toolkit to compute the requested size before drawing, which can make the initial appearance seem faster.

The box container and button constructors have the argument `container`. This is used to define the widget hierarchy and layout of the GUI. The dialog has the similar `parent` argument. This argument does not imply the child is rendered within the parent, as `container` does, but rather indicates a relationship. In this case, the alert dialog will be drawn relative to the parent window.

Except for dialogs, the constructors return objects which have S3 methods defined for them. Table `tab:methods` lists most of the new generics defined by the package. [4]. The last line shows the `visible` assignment method. For top-level windows, this can be used to either hide or make the window visible. The most common method is `svalue`, which is defined to return or set the most basic property of a widget.

The interactivity of **gWidgets2**'s GUIs is introduced by event handlers. Handlers for the most common event can be assigned during the construction of the object, but more commonly handlers are done after the layout of the GUI is finished, as above.

The above handler simply calls the `galert` dialog to create a transient message.

There are other examples available in the demos for the toolkit packages, e.g., `demo("gWidgets2RGtk2")`.

2 Overview

The basic example above shows the main areas of **gWidgets2**. Here we give a bit more detail.

[4]Constructors return reference class objects, so they also have reference class methods defined. There are a few cases where these are standardized across toolkits, for example, the `set_borderwidth` argument for box containers. Otherwise, though the reference class methods may be used directly, doing so will often break portability of the code to other toolkits.

Method	Description
`svalue<-`	set main property
`enabled<-`	set if widget is sensitive to user input
`visible<-`	adjust "visibility" of widget
`focus<-`	give widget keyboard focus
`editable<-`	adjust if widget can be edited
`font<-`	adjust font of widget
`tab<-`	store property in widget's environment
`size<-`	adjust size of widget
`tooltip<-`	add tooltip to widget
`dispose`	dispose of widget or part of widget
`delete`	delete child from parent container

Table 1: Generic methods defined in the **gWidgets2** package to manipulate the state of the GUI objects. Most "setters" have corresponding "getter" versions.

2.1 Containers

A GUI is laid out by nesting child components within containers, which in turn may be nested within other containers. All the children sit within some top-level window. In **gWidgets2**, top-level windows are constructed with `gwindow`.

The most basic containers are the "box containers", `ggroup` (and its shortcut `gvbox`), `gframe`, `gexpandgroup`). Box containers pack in their child components from left to right or top to bottom. The packing of each child can be adjusted through specifications of `expand` (to have the allocated space grow if the window size grows), `fill` (to have the child widget grow to fill the allocated space), or `anchor` (to align the child widget within the allocated space). The spacing between children is controlled by the main property and border spacing, by the aforementioned reference class method `add_borderwidth`.

Other containers are

glayout A grid layout container

gformlayout A special case of a grid layout container for quickly building up forms.

gpanedgroup A container to hold two children, separated by a sash. This is used to allow the user to allocate the allotted space.

gnotebook A notebook widget uses tabs to organized different "pages" from which the user can easily select.

gstackwidget Like `gnotebook`, though no tabs are presented so selection is done programmatically.

gbasicdialog A dialog that can be used to create a modal window

The containers can be nested to build up a hierarchy. This examples shows how nested groups can be used to create a button group at the bottom of a page:

```
## Some filler
lorem <- "Lorem ipsum dolor sit amet, consectetur adipiscing elit."
##
win <- gwindow("Nested groups")
g <- gvbox(container=win)
g$set_borderwidth(10L)
txt <- gtext(lorem, container=g, expand=TRUE, fill=TRUE) ## text widget
bg <- ggroup(cont=g)
addSpring(bg)
gbutton("dismiss", container=bg, handler=function(h,...) dispose(win))
gbutton("about", container=bg, handler=function(h,...) {
  gmessage("Shows lorem ipsum text", parent=win)
})
```

In the above, we have nested box containers, `g` a vertical one, and `bg` a horizontal one. (The `gvbox` call is a simpler to type version of `ggroup` with `horizontal=FALSE`.) The `set_borderwidth` reference method call is not essential, but does give some extra space around the edges of `g`. The `svalue` method is used to set padding between children, this call sets the border within the box. The spring is added to `bg` container to right justify the buttons. When buttons are packed into box containers, they expand to fill the direction orthogonal to the direction of packing. As such, it is best to put buttons in a horizontal box container, as done above.

2.2 Widgets

Containers are needed for organizing the GUI, but more importantly are the controls that allow the user to enter information into a GUI or get information out of a GUI.

A summary of the basic controls is given in Table 2.

The basic use is straightforward. A widget has some data that is needed to configure it, and most widgets have common arguments to specify the parent container (`container`) and a handler for the most common event (`handler`, which just maps to `addHandlerChanged`).

Type	Constructor	Description
labels	`glabel`	label widget
	`gstatusbar`	messages in page bottom
	`galert`	transient messages
	`ghtml`	if supported (`Qt`)
action	`gbutton`	button action
	`gmenu`	menu bar and popup menus
	`gtoolbar`	tool bar
selection	`gcheckbox`	Boolean
	`gradio`	one of many
	`gcheckboxgroup`	one or more of many
	`gcombobox`	one of many
	`gtable`	one, or one or more of many
	`gslider`	slide to select from range
	`gspinbutton`	click to select from range
	`gcalendar`	select date
	`gfilebrowse`	select file, directory
Text	`gedit`	single line text
	`gtext`	multi line text
Data	`gtable`	select from rectangular data
	`gtree`	select from hierarchical data
	`gdf`	edit a data frame
Graphics	`ggraphics`	graphic device
	`gimage`	image file display

Table 2: List of basic controls in **gWidgets2**

The following example shows a GUI that could be used to collect arguments for a *t*-test (Figure 1). We use the convenient `gformlayout`, new to **gWidgets2**.

```
win <- gwindow("t-test", visible=FALSE)
g <- gvbox(container=win)
g$set_borderwidth(10L)
flyt <- gformlayout(container=g, expand=TRUE)
##
gedit("", initial.msg="variable",
      label="x", container=flyt)
gcombobox(c("two.sided", "less", "greater"),
          label="alternative",container=flyt)
gedit("0", coerce.with=as.numeric,
      label="mu", container=flyt)
gcheckbox("", checked=FALSE,
          label="paired", container=flyt)
gslider(from=0.5, to = 1.0, by=.01, value=0.95,
        label="conf.level",  container=flyt)
##
bg <- ggroup(container=g)
addSpring(bg)
gbutton("values...", container=bg, handler=function(h,...) {
  print(svalue(flyt))                   # replace me...
})
addSpring(g)                            # better for Qt
##
size(win) <- c(400, 250)
visible(win) <- TRUE
```

We see that some arguments are specific to the widget. For example, `initial.msg` to use an initial message for `gedit`; `coerce.with` to specify a function to coerce a character string for `gedit`, and others, or `checked` to set the initial state for `gcheckbox`.

The `expand` argument and `label` argument do not belong to the widget constructor, but are passed through `...` to the `add` method for `ggroup` and `gformlayout`. These control how the child is laid out within the parent. For `gformlayout`, the `label` argument also names the value returned by its `svalue` method.

The above, is just the part of the GUI that collects the user input. One would need to write something intelligent to do with the values. Though, if the labels are chosen

Method	Description
`addHandlerChanged`	most typical event
`addHandlerClicked`	click event
`addHandlerDoubleclick`	double-click event
`addHandlerSelect`	Select or activate event (typically the change event)
`addHandlerSelectionChanged`	Selection changes
`addHandlerFocus`	widget gets keyboard focus
`addHandlerBlur`	widget loses keyboard focus
`addHandlerKeystroke`	text widget has keystroke
`addHandler`	add callback for toolkit signal

Table 3: Methods to bind callbacks to events in **gWidgets2**

well, this can be as simple as using something like `do.call(FUN, svalue(form_layout_object))`.

2.3 Event handlers

GUIs become interactive through the use of event handlers. The basic idea being the user initiates an event through a control, the control then emits a signal, and any listeners for the signal are called. In **gWidgets2** there are various "`addHandlerXXX`" methods to attach a callback (a handler) for different events a widget may signal. The most basic event is bound to through `addHandlerChanged`, which for many widgets is an alias for a more aptly named event. (For example, `addHandlerClicked` for the button widget.) Most others are listed in Table 3, though some are not, such as the different ones for column clicks in the table widgets.

The callbacks all have the same signature, `(h, ...)`. The main argument is `h`, a list with components `obj` to refer to the emitter of the signal and `action`, an optional user-supplied value to parameterize the callback. Some events pass back more information. For example, the keystroke handler passes back key information through `h`. The `...` values are used by some toolkits to pass back information given by the toolkit about the signal. This may be of interest to some, but using it breaks portability of **gWidgets2** code across toolkits.

The `addHandlerXXX` methods return an ID that can be used to disconnect the callback or temporarily block the callback. See the help pages for `removeHandler`, `blockHandler`, and `unblockHandler`. All events can be removed or temporarily blocked through `removeHandlers`, `blockHandlers`, and `unblockHandlers`.

A simple example of using a handler might be to have the sensitivity of a button depend on whether a user has made a selection:

```
win <- gwindow("handler example", visible=FALSE)
g <- gvbox(container=win)
f <- gframe("Ethnicity", container=g)
cb <- gcheckboxgroup(c("White",
                  "American Indian and Alaska Native",
                  "Asian",
                  "Black or African American",
                  "Native Hawaiian and Other Pacific Islander"),
                container=f)
bg <- ggroup(cont=g); addSpring(bg)
b <- gbutton("Go", container=bg)
enabled(b) <- FALSE
##
addHandlerChanged(cb, handler=function(h,...) {
  enabled(b) <- length(svalue(h$obj)) > 0
})
##
visible(win) <- TRUE
```

2.4 Dialogs

The package provides a few modal dialogs, useful and familiar means to display or collect information. These include:

gmessage present a modal message

galert present a non-modal, transient message

gconfirm allows user to confirm an action to be taken

ginput collect single line of text input from user

gbasicdialog container to hold modal dialog

gfile Select a file

Modal dialogs disrupt the flow of a user's interaction through a GUI, so are used sparingly. This also prevents them from returning a meaningful object to manipulate, as this can only happen after the dialog is closed. As such, they return values such as a Boolean for `gconfirm` and the text value for `ginput`.

This example asks for confirmation before removing an object:

```
if(gconfirm(c("Remove x", "this can't be undone")))
  rm("x")
```

3 Some more examples

3.1 Selecting packages to load/unload

This example creates a GUI to load or unload a package. The main interface uses `gcheckboxgroup` for the selection of one or more from many. The argument `use.table=TRUE` is specified to use a layout with checkboxes, as otherwise scrollbars will not be provided.

We begin with some text to explain the GUI.

```
about <- "
A simple GUI to simplify the loading and unloading of packages.
This GUI uses `gcheckboxgroup`, with its `use.table` argument, to
present the user with familiar checkboxes to indicate selection.
Some indexing jujitsu is needed to pull out which value is checked to
trigger the event.
"
```

The function `installed.packages` will search a users installation for all installed packages. This call can be slow, so we store the results here as a global variable. In a more complicated setup, one would use a property, say, of a reference class to avoid name collision.

```
installed <- installed.packages() ## matrix
installed_packages <- installed[, "Package"]
```

This helper function checks the loaded namespaces against the installed packages to determine what is loaded.

```
package_status <- function() {
  ## Return if package is loaded
  loaded <- loadedNamespaces()
  sapply(installed_packages, function(i) i %in% loaded)
}
```

We begin with our main layout. Here is a common idiom – creating a top-level window with a box container to hold child components. The `set_borderwidth` reference method is used to give a little breathing room.

```
w <- gwindow("package manager", visible=FALSE)
g <- gvbox(cont=w)
g$set_borderwidth(10L)
```

Here we use the checkbox group to show the data. The items to select from are specified first, then the `checked` argument is fed a logical variable to indicate what should be initially checked. We use the `expand=TRUE` argument here to that the widget gets the maximum space possible, should the window be resized.

```
a <- package_status()
tbl <- gcheckboxgroup(installed_packages, checked=package_status(),
                      use.table=TRUE,
                      expand=TRUE, container=g)
```

The following is a standard idiom to create a button group. This button simply shows the information defined about in `about`. The specification of `parent=w` below makes the window `w1` transient for the toplevel window `w`. That means, should `w` be closed first, `w1` will also close and further, the initial positioning of `w1` depends on that of `w`.

```
bg <- ggroup(cont=g)
addSpring(bg)
gbutton("About", container=bg, handler=function(...) {
  w1 <- gwindow("About", parent=w, visible=FALSE)
  g <- gvbox(container=w1); g$set_borderwidth(10)
  glabel(about, container=g)
  gseparator(container=g)
  bg <- ggroup(cont=g)
  addSpring(bg)
  gbutton("dismiss", cont=bg, handler=function(h,...) {
    dispose(w1)
  })
  visible(w1) <- TRUE
})
```

```
Object of class GButton
```

Finally, we make the toplevel window visible:

```
visible(w) <- TRUE
```

To add interactivity to our GUI we need some means to synchronize the display of the table with the state of the loaded packages. Below, we block all handlers before updating the selected values, as otherwise this may trigger the change handler to be called.

```
update_tbl <- function(...) {
  blockHandlers(tbl)
  on.exit(unblockHandlers(tbl))

  svalue(tbl, index=TRUE) <- package_status()
}
```

Finally, we add the handler that is called when the selected values change. There is no means to get what checkbox actually triggered the change, so we compute that from the selected values and the actual installed packages. The `setdiff` function comes in handy here. The basic flow of this handler is clear: check which value changed. If it was a deletion, detach the package, otherwise load the package. Afterwards, update the table.

```
addHandlerChanged(tbl, handler=function(h, ...) {
  ind <- svalue(h$obj, index=TRUE)
  old_ind <- which(package_status())

  if(length(x <- setdiff(old_ind, ind))) {
    message("detach ", installed_packages[x])
    pkg <- sprintf("package:%s", installed_packages[x])
    detach(pkg, unload=TRUE, character.only=TRUE)
  } else if (length(x <- setdiff(ind, old_ind))) {
    require(installed_packages[x], character.only=TRUE)
  }
  update_tbl()
})
```

One last comment, this GUI is similar to one in RStudio, but is not nearly as nice as that. There, more information per line (a package description and an "uninstall package" icon) is included. Though one could add in a description here, say by pasting together different text to display in the checkbox labels, it wouldn't be possible to also add the uninstall feature. The package isn't as powerful as the underlying toolkits, though as you can see the creation of this table along with the interaction to retrieve and set its selected values is really quite simple, especially when compared to what this takes with some of the underlying toolkits.

3.2 Using a table to display information

A basic feature of many GUIs is the use of tables to display information. The `gtable` constructor creates an object that has data frame methods that can be used to manipulate the object. For example, `length`, `dim`, and `[`. As well, in this example we see how the `visible` method can be used to assign which rows are visible.

We begin by describing the GUI.

```
about <- "GUI to upgrade installed packages"
```

The data to be displayed here is returned by `old.packages`, of which we wish just 3 columns. Before calling that function, we programmatically set a mirror. This bit can be skipped.

```
repos <- getOption("repos")
repos["CRAN"] <- "http://streaming.stat.iastate.edu/CRAN/"
options(repos = repos)
#
pkg <- old.packages()[,c("Package", "Installed", "ReposVer")]
```

How we begin our basic GUI. First a top-level window:

```
w <- gwindow("Upgrade installed packages", visible=FALSE)
g <- gvbox(container=w)
g$set_borderwidth(10)
```

Our main interface consists of a `gedit` instance for the user to filter the table by, and a `gtable` widget to dispaly our tabular data.

```
fg <- ggroup(container=g)
glabel("Filter by:", container=fg)
```

```
Object of class GLabel
```

```
fltr <- gedit("", initial.msg="Filter by regexp", container=fg)
tbl <- gtable(pkg, chosen.col=1, multiple=TRUE, container=g, expand=TRUE)
```

Our button group is fairly standard.

```
  bg <- ggroup(container=g); addSpring(bg)
  gbutton("About", container=bg, handler=function(h,...) {
    w1 <- gwindow("About", parent=w, visible=FALSE)
    g <- gvbox(container=w1); g$set_borderwidth(10)
    glabel(about, container=g, expand=TRUE)
    bg <- ggroup(container=g); addSpring(bg)
    gbutton("dismiss", container=bg, handler=function(h,...) dispose(w1))
    visible(w1) <- TRUE
  })
```

```
Object of class GButton
```

Our update button will only be enabled if the user has a selection, hence we set it intially to be disabled. The handler below simply installs the selected packages, then updates the display.

```
  update_btn <- gbutton("Update selected", container=bg, handler=function(h,...) {
    pkgs <- svalue(tbl)
    if(length(pkgs) == 0) return()

    sapply(pkgs, install.packages)
    ## update pkg, then update talbe
    tbl[] <- pkg <<-  old.packages()[,c("Package", "Installed", "ReposVer")]
  })
  enabled(update_btn) <- FALSE
  #
  visible(w) <- TRUE
```

To enable filtering through the `gedit` widget, we connect to its keystroke event a handler that uses `grepl` to see which packages the user wishes to be displayed. The `visible` assignment method makes this very easy to do.

```
  addHandlerKeystroke(fltr, handler=function(h,...) {
    regexp <- svalue(h$obj)
    if(nchar(regexp) > 0 && regexp != "") {
      ind <- grepl(regexp, pkg[, 'Package'])
      visible(tbl) <- ind
    } else {
      visible(tbl) <- rep(TRUE, nrow(pkg))
    }
  })
```

Finally, we adjust the update button to be enabled when there is a selection. The addHandlerSelectionChanged method makes listening for changes to the selection easy. The table widget also has an addHandlerChanged method, which listens for when a use activates an item, typically by double clicking or through the return key.

```
addHandlerSelectionChanged(tbl, handler=function(h,...) {
  enabled(update_btn) <- length(svalue(h$obj))
})
```