A note on using Gtk devices in R without the Gnome GUI

Duncan Temple Lang

October 31, 2002

This discusses separating the Gnome GUI for R and the Gtk graphics device. Specifically, we suggest a way for using GTK devices independently of the Gnome GUI. The current situation requires that if a user wants to create a graphics device using the *gtk()* function, she must start R with `--gui=GNOME`. This is not typically a hardship except for users who prefer to use a different interface such as the console, emacs or a debugger. The tight coupling of GUI and device becomes more limiting when we use the `RGtk` package to create GUIs in R and want to embed Gtk devices within a GUI. This facility now exists and requires that we can create Gtk devices without starting the R session with the Gnome interface. In the long run, we may also want to use `RGtk` to create the Gnome interface within R code itself and allow it to be dynamically created during a session, and also customized and extended at the user-level.

What follows is is a description of an organizational change to allow gtk graphics device instances to be created independently of the Gnome GUI for R. The essential user-level changes consist of adding two new C-level entry points which can be called from R.

`R_gnome_init_libs()` The effect of calling this is to load the Gnome module (*R˙gnome.so*) into the R session and initialize the gnome and glade libraries, and reads the GUI preferences. After this, one can use the Gtk device mechanism in the usual way. Somebody will have to arrange to process the Gtk event queue. This can be done via the `RGtk` package. To load the module and resolve the "methods"/function pointers this routine invokes `R_load_gnome_device_methods()` (explained next).

`R_load_gnome_device_methods()` This routine takes care of loading the Gnome module and resolving the methods for the Gtk device driver. This can be called rather than the broader and more inclusive `R_gnome_init_libs()` if one does not want the full window interface for the Gtk device. Instead, if one wants to use, for example, the `RGtk` package within an R session or within a GGobi or Gnumeric process, then one need only invoke this routine and avoid initializing Gnome and glade.

Neither of these routines create or start the Gnome GUI or do anything with the Gtk event loop. One can still use the full Gnome interface using `R --gui=GNOME`.

Currently, if one attempts to load the Gnome module and it fails, the code terminates the R session by calling `R_Suicide()`. We should make this more graceful and less terminal.

1 Creating Regular Gtk Device Windows

To create a regular Gtk device as a separate window using *gtk()*, one must cause the gnome and glade libraries to be initialized before creating the device. To do this, one can call the routine `R_gnome_init_libs()` with a vector of strings giving the command line arguments with which to initialize the gnome and glade libraries. For example, to start with a single argument (`R`) we issue the command

```
.C("R_gnome_init_libs", "R", as.integer(1))
```

and then invoke *gtk()*.

One may need to call *gdkFlush()* after calling *gtk()* as the events may not be processed immediately to show the window.

2 Embedding Gtk Devices in GUIs: Coercing a Drawing Area to an R Gtk Device

In addition to being able to create windows containing Gtk devices one can also embed a Gtk graphics device in a more customized or complex GUI. The idea is that we first create a Gtk drawing area instance. Then we use the *asGtkDevice()* function to have R treat this widget as a Gtk graphics device and draw onto it.

Let's consider an example in which we create a top-level window containing a single drawing widget.

```
library(RGtk)
gtkInit()
win <- gtkWindow(show = FALSE)
w <- gtkDrawingArea()
win$Add(w)
win$SetUsize(300, 300)
win$Show()
```

Now we should have a window on the screen and an empty drawing area.

We tell R to use that drawing area widget as a graphics device

```
asGtkDevice(w)
```

And now we can use this as a regular R graphics device. All plotting functions and device management functions will work as expected.

```
plot(1:10)
title("Foo")
```

We can switch between this and other devices. For example, we can create a new device, draw to it and then draw a new plot on the original embedded device using the following commands.

```
old <- dev.cur()
gtk()
hist(rnorm(100))
dev.set(old)
plot(rnorm(100))
```

The *asGtkDevice()* uses the newly added C routine `do_asGTKDevice()`.

3 A General Extensible Gnome GUI for R written in R

In the future (i.e. when someone has the time and interest), we can create the Gnome GUI with R code using RGtk. Then we can make the current Gnome material a dynamically loadable package. This will allow one to create and destroy one or more GUIs during the R session rather than committing to that interface for the entire session.

If we want the existing Gnome interface, we can do the equivalent of calling

```
structRstart rstart;
Rstart Rp = &rstart;

   ...

ptr_gnome_start(args, length(args), Rp)
```

Additionally, one can create different interfaces using some of the components, especially the console. This can be accessed as a regular Gtk object using the general mechanism provided by the `RGtk` package.

4 Initialization of Embedded Graphics Devices

Unfortunately, there is a slight structural problem when creating a device from a Gtk drawing area widget and drawing to it immediately. Specifically, if we create a device and plot to it within the same top-level task, things will not work as we might want. The difficulty is reasonably simple to understand and, fortunately, to remedy.

When we create the device from the widget, we do not know its size. This is normally because the device has not been realized and drawn in its parent container widget at this point. This may simply beacuse we have not called *gtkWidgetShow()* for the top-level window and its sub-widgets at this point, or simply because the display events have not been processed. Regardless of why this happens, the internal graphics code thinks that the device has zero width and height. Of course, if we attempt to draw on the device at this point, the R graphics engine will complain that the dimensions are too small and hence the margins are too large. The result will be an error and the plot will not be drawn.

None of this is an issue if one creates and displays the device in one top-level task and then draws to it in another subsequent top-level task. This combined with the knowledge that we want to draw only when we know the widget size suggests a solution to the problem. We can register a callback/signal handler for expose (or configuration) event for the drawing area widget. The first time this callback function is invoked, we create the desired plot contents. At this point, the dimensions of the widget will be known and the graphics engine will be able to correctly determine the layout of the "page". Within this callback, we can remove that very callback so that it is not called for subsequent expose events.

The code below illustrates how this can be done. We define a function *testDev()* which creates a Gtk window containing a drawing area widget. We coerce this widget to a device and then show the window. We specify a callback for the *expose˙event* signal and this creates the plot on the device and removes the callback so it won't be called again.

```
testDev <-
function()
{
 win <- gtkWindow(show = FALSE)
 w <- gtkDrawingArea()
 win$Add(w)
 win$SetUsize(400, 400)


 id <- w$AddCallback("expose_event",
                function(w, event) {
                       plot(1:10)
                       w$DisconnectCallback(id)
                })

 .C("R_load_gnome_device_methods")
 asGtkDevice(w)
 win$Show()
}
```

A simpler, more kludgy way of doing things is to declare that the device has some arbitrary dimensions (e.g. 300×300) at the time we coerce the widget to a device. Then, we can draw and the graphics system will not complain because it has adequate space. While this resulting plot is incorrect, nobody sees it since the device has not been displayed and the events processed. When the window is exposed, it will recompute the size and update the display, computing the plot contents appropriately.

5 Additions

Two C-level routines exported from base `R_load_gnome_device_methods()` and `R_gnome_init_libs()`. We also add a routine `do_asGTKDevice()` which currently is a primitive but can be made a regular *.Call()*-routine.

`devGTK.c` in `modules/gnome/` is where we add the C-level argument corresponding to the *no.window* parameter in *gtk()*. We separate out the initialization of the device into a separate routine so that it can be used when creating a device from an existing widget as well as creating a new widget itself.

We also add a routine which provides access to the drawing area widget within the locally-defined structure describing the Gtk device. This is used when create a regular Gtk device (i.e. not supplying our own drawing area widget) but do not create a top-level window. More on this below.

In `system.c`, we define the routine (`gnome_init_libs()`) for initializing the Gnome libraries. This allows the gnome and glade libraries to be initialized after the R session has been started and separates the use of the Gnome GUI and the initialization of the gnome libraries so that we can use the latter without the former.

In `devUI.h`, we add some "methods" or function pointers for facilities provided by the gnome module. Specifically, we add `GTKDeviceGetDrawingWidget()` and `asGTKDevice()`. These methods are declared and resolved in `gnome.c`. Also, this file provides declarations for stubs that provide degenerate implementations of these when Gnome is not available.

`gnome.c` provides the changes needed to provide the additional entry points to interface to the Gnome module code. These are the routines to load the Gnome module, initialize the Gnome and Glade libraries, resolve the Gtk device methods This also provides stubs for the new routines when Gnome support is not compiled for R. Also, it reorganizes how the symbols from the Gnome module are resolved.

`devices.c` provides the implementation of the `do_asGTKDevice()`. Additionaly, this file provides the changes to `do_gtk()` to handle the extra argument *no.window*.

`stubs.c` reflects the addition of the *no.window* argument to the the *gtk()* function. Also, it provides an implementation of the *asGTKDevice()* routine when Gnome is not available.

`system.c` reflects the addition of the *asGTKDevice()* and the need for another method to be initialized to the default stub.

In `x11.R`, we add a logical *no.window* argument to *gtk()* and add the function *asGtkDevice()*. We have added a primitive `do_asGTKDevice()`. We may make this accessible via a regular *.Call()* interface.

6 The no.window argument

I originally added a *no.window* argument the *gtk()* function. This is intended to be a logical value that by default (**F**) creates a regular Gtk device in a new top-level window, but if **T**, creates a new Gtk graphics device without the top-level window. In this case, it returns an external pointer object containing the address of the underlying `GtkDrawingArea`. Using the `RGtk` package, one can then convert this into a regular Gtk object in R and add it to a GUI.

Since we have also added the *asGtkDevice()*, one technically does not need this *no.window* argument. Instead, one can create the `GtkDrawingArea` widget and coerce it to a device. So, in the interest of simplicity and the status quo, we might want to drop this addition. Of course, that involves changing the changes and re-testing, etc.

7 Issues

Ideally, I would like to be able compile parts of the current Gnome module under Windows (and the Mac) and then have the Gtk interface run there. Perhaps the simplest way to do this is to extract the Gtk device into a package and then make the package platform independent.

Integrating the Gtk event loop with R's event loop doesn't handle the idle and timed events correctly. We can add some hacks to do this. However, I want to overhaul the R event loop so that it accepts the event loops of other applications/systems properly when those systems are embedded in R, and also when R is embedded within other systems and needs event loop support for things such as graphics devices, connections, etc. In many respects it would be nice to make the event loop configurable at the start of the session and to have an interface defining primitives for adding entries to the event loop with different implementations that would be swapped in according to which event loop was desired. Of course, true threading will make all this go away. But in the absence of threads (being used), we will still need something that works.